Optimization Problems in Communication Networks

Thesis submitted for the degree of
Doctor of Philosophy
at University of Leicester

by

Matúš Mihaľák
Department of Computer Science
University of Leicester

October 2006

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

Matúš Mihaľák

Optimization Problems in Communication Networks

Matúš Mihaľák

Abstract

We study four problems arising in the area of communication networks.

The *minimum-weight dominating set* problem in unit disk graphs asks, for a given set $\mathcal{D}$ of weighted unit disks, to find a minimum-weight subset $\mathcal{D}' \subseteq \mathcal{D}$ such that the disks $\mathcal{D}'$ intersect all disks $\mathcal{D}$. The problem is $\mathcal{NP}$-hard and we present the first constant-factor approximation algorithm. Applying our techniques to other geometric graph problems, we can obtain better (or new) approximation algorithms.

The *network discovery* problem asks for a minimum number of queries that discover all edges and non-edges of an unknown network (graph). A query at node v discovers a certain portion of the network. We study two different query models and show various results concerning the complexity, approximability and lower bounds on competitive ratios of online algorithms.

The *OVSF-code assignment* problem deals with assigning communication codes (nodes) from a complete binary tree to users. Users ask for codes of a certain depth and the codes have to be assigned such that (i) no assigned code is an ancestor of another assigned code and (ii) the number of (previously) assigned codes that have to be reassigned (in order to satisfy (i)) is minimized. We present hardness results and several algorithms (optimal, approximation, online and fixed-parameter tractable).

The *joint base station scheduling* problem asks for an assignment of users to base stations (points in the plane) and for an optimal colouring of the resulting conflict graph: user u with its assigned base station b is in conflict with user v, if a disk with center at b, and u on its perimeter, contains v. We study the complexity, and present and analyse optimal, approximation and greedy algorithms for general and various special cases.

Acknowledgements

It all began when I joined the research group of Dr. Thomas Erlebach at TIK institute of ETH Zürich. This was a great opportunity for me and I am delighted to say that it was one of my best decisions. Another one was to follow him on his move to University of Leicester. Thomas is an outstanding researcher and all the more an excellent supervisor. I benefited from his instant help, advice, encouragement, answers to questions, bringing new ideas, identifying interesting problems, and tackling many of the "I will never solve this" problems, where in many cases his ideas proved to be invaluable for further progress. To all of this I am very grateful. Everything that I know about research and presentation of my ideas was shaped under Thomas' ultimate guidance. The greatest thank you goes to Thomas.

At all stages of my studies, I had the chance to meet and work with many influential persons, the co-authors of many results presented in this thesis: Prof. Peter Widmayer, Dr. Christoph Ambühl, Dr. Alex Hall, Dr. Michael Hoffman (my second supervisor at University of Leicester), Dr. Riko Jacob, Dr. Marc Nunkesser, Dr. Gábor Szabó, Zuzana Beerliová and Shankar Ram. A special mention goes to then Ph.D. students Marc and Gábor from ETH Zürich for the fruitful two years we spent on discussing many algorithmic problems.

Both in Zürich and in Leicester I found a friendly environment to work. The Theory Group at TIK institute of ETH Zürich was a small bunch of marvellous young people whose memorable "töggele" skills were cultivated after the traditional zvieri. Thank you, Alex, Danica, Sai and Stamatis. From Leicester, I will keep my memories of puzzled lunches, which were guided by Nick and Michael. I want to mention Ahmed with whom I shared the room G5 for two years and discussed many aspects of the worldwide conflicts. I also want to mention Rajeev, Rick and Fer-Jan for their own contribution to the overall good time I had in Leicester.

My very last (and big) thank you goes to my family, for all their constant support and love.

Contents

1 Introduction 7

2 Notation, Terminology and Theory 12

3 Weighted Dominating Sets in UDG 19

3.1 Problem Definition 20

3.2 Related Work and New Contributions 21

3.3 Algorithm for Minimum-Weight Dominating Sets 23

3.4 Solving the Subproblem for a Small Square 24

3.4.1 Algorithm for Disk Cover in a Small Square 26

3.4.2 Algorithm for General Weighted Disk Cover with Unit Disks 39

3.5 Connecting the Dominating Set 40

3.6 Covering Points with Unit Squares 43

3.6.1 Covering Points in a Strip 43

3.6.2 Approximation Algorithm for WSCP 44

3.6.3 MWDS in Unit Square Graphs 44

3.7 A 3-approximation Algorithm for Minimum-Weight Forwarding Sets 45

3.8 Summary of Results and Open Problems 47

4 Network Discovery and Verification 49

4.1 Problem Definitions and Preliminaries 52

4.2 Related Work and New Contributions 56

4.3 Layered-Graph Query Model 59

4.3.1 A Few Structural Properties 59

4.3.2 The Online Problem 65
4.4 The Distance Query Model 71
4.4.1 Discovering Individual Edges and Non-Edges 71
4.4.2 Structural Properties 73
4.4.3 Polynomially Solvable Cases 75
4.4.4 The Offline Problem 82
4.4.5 The Online Problem 85
4.5 Summary of Results and Open Problems 90

5 Assignment of OVSF-Codes **92**
5.1 Formal Problem Definition 94
5.2 Related Work and New Contributions 96
5.3 Folklore 98
5.3.1 Call Admission Feasibility 98
5.3.2 Irrelevance of Higher Level Codes 100
5.3.3 Arbitrary Code Tree Configuration 101
5.4 One-Step Offline CA 103
5.4.1 Non-Optimality of Greedy Algorithms 103
5.4.2 Complexity of One-Step Offline CA 105
5.4.3 An Optimal Algorithm and Fixed Parameter Tractability 108
5.4.4 An h-Approximation Algorithm for One-Step offline CA . 110
5.4.5 Proof of Lemma 5.10 116
5.5 Online CA 120
5.5.1 Greedy Strategies 121
5.5.2 Compact Representation Algorithm 122
5.5.3 Minimizing the Number of Blocked Codes 125
5.6 Summary of Results and Open Problems 128

6 Joint Base Station Scheduling **130**
6.1 Problem Definitions and Model 132
6.2 Related Work and New Contributions 135
6.3 Case on the Line—1D-JBS 137
6.3.1 Arrow Graphs 137
6.3.2 Evenly Spaced Base Stations 141

6.3.3 Serving $3k$ Users with 3 Base Stations in k Rounds 144
6.3.4 Exact Algorithm for the k-Decision Problem 147
6.3.5 Approximation Algorithm 149
6.3.6 Different Interference Models 152
6.4 The General Case—2D-JBS 153
6.4.1 $\mathcal{NP}$-Completeness of the k-2D-JBS Problem 154
6.4.2 Base Station Assignment for One Round 154
6.4.3 Approximation algorithms 156
6.5 Summary of Results and Open Problems 165

7 Conclusion **167**
7.1 Future Work 168

Chapter 1

Introduction

This thesis has two main goals. Besides the ultimate goal of presenting the research and research results of the author, the other goal is to deliver an interesting and compact reading for a passing by "pedestrian". The problems studied and presented in this thesis dare to ask to satisfy the first goal, whereas the problems' different background and nature makes this thesis a reading of 4 rather independent stories and therefore it is more challenging to address the second goal, which could be understood as presenting a fluent, one-shot reading. In spite of the fact that the problems were studied independently from each other, there are some common elements that unite the thesis under one roof.

The very common point of each of the presented stories is that they all deal with combinatorial problems that are tackled by algorithmic techniques (we talk about the algorithm-theoretic frame in Chapter 2). The cultivating medium for this thesis is the area of *communication networks* and all problems that are presented in this thesis originate within that area. A communication network is, as defined in [13], an organization of stations capable of intercommunications (but not necessarily on the same channel). Leaving this abstract definition, we can describe a communication network as communication entities connected by communication links that allow communication to be passed from one part of the network to another over multiple communication links. Here, communication links are understood as a means of delivering data from one place to another and can be of various nature: wires, fiber optic, radio, or a combina-

tion thereof. Building a communication network demands a variety of engineers and researchers to cooperate together. The diversity of the field offers also a big diversity of research problems. We are interested in problems that can be modeled and studied by means of algorithmic theory. Still, these problem can vary substantially, as the 4 problems that are studied in this thesis demonstrate and whose brief description follows.

Weighted Dominating Sets in Unit Disk Graphs

The dominating set problem is a classical graph-theoretic optimization problem. For a given graph $G = (V, E)$, a set $D \subseteq V$ is called a dominating set if every vertex from V is in D or a neighbour is in D. We consider graphs whose vertices have weights associated with them. The minimum-weight dominating set problem (MWDS) is to find a dominating set of minimum total weight (i.e., the sum of weights of vertices from the dominating set). The minimum-weight connected dominating set problem (MWCDS) asks for a dominating set of minimum weight such that the induced graph by the dominating set is connected. In this thesis we are interested in MWDS and MWCDS in a special class of graphs—unit disk graphs. A unit disk graph is a graph for which every vertex is associated with a disk in the plane. The radius of the disk is one. There is an edge between two vertices if the corresponding disks intersect. Thus, for a given set $\mathcal{D}$ of unit disks in the plane, where every disk has a weight associated with it, the MWDS problem in unit disk graphs (the MWCDS in unit disk graphs) is to select disks $\mathcal{D}' \subseteq \mathcal{D}$ of minimum total weight such that every disk from $\mathcal{D}$ is selected in $\mathcal{D}'$ or intersects a disk from $\mathcal{D}'$ (and the intersection graph of $\mathcal{D}'$ is connected).

The problem is studied in Chapter 3.

Network Discovery and Verification

This problem is motivated by the efforts of obtaining a map of large scale, self-organizing networks, such as the internet. A map of a network (and the network itself) is modeled as a graph $G = (V, E)$. The nodes V represent the communication entities (such as Autonomous Systems in the internet) and the edges represent direct communication links. We assume that the information

about links is not known and the goal is to discover all the edges and non-edges (a non-edge of a graph is a pair of vertices that do not form an edge). This can be done by querying (known) vertices of the network. Each query at vertex v gives some information about the network. The network discovery problem is to discover all edges and non-edges of the graph using a minimum number of queries. We consider the following 2 query models: the query at vertex v returns distances from v to all other vertices of the network; and the query at vertex v returns all shortest paths from v to every other vertex. Since the information about the network's edges is gained on the fly by querying vertices of the network, this is an online problem. The offline version of the problem can be stated as follows. A graph G is given, both the vertices and edges are known, and the goal is to compute a minimum number of queries that discover the graph. Since the edge set of the graph is known, the result of queries at every vertex is known in advance. The offline version of the network discovery problem is also called the network verification problem.

The problem is studied in Chapter 4.

Assignment of OVSF-Codes

In mobile telecommunication systems of the third generation (3G), for example in UMTS, the sharing of bandwidth can be accomplished via Orthogonal Variable Spreading Factor Codes (OVSF-codes). In a communication cell served by a base station, users are assigned these codes to use them when communicating with the base station. These codes can be viewed as nodes of a complete binary tree T, where a code from level l of the tree guarantees a bandwidth proportional to 2^l. The users can distinguish the signal of the base station if all the codes that are in use are orthogonal, i.e., if no two codes lie on the same leaf-to-root path in T. Users request, upon arrival, a code from a certain level, reflecting their bandwidth demand. It is the task of the base station to decide which code from the requested level to assign to the user. Over time, as users enter and leave the cell, it can happen that the new user cannot be assigned a code—all codes from level l are in use or lie on a common leaf-to-root path with an assigned code—but a different code assignment of codes to existing users would allow the new user to get her code. The code reassignment causes extra communication

with the users, resulting subsequently in delays and users disturbance. Thus, a natural goal is to keep the number of users that have to change their code as low as possible—either at the moment of a new user's request or over a period of time.

The problem is studied in Chapter 5.

Joint Base Station Scheduling

Another means of sharing bandwidth in mobile communications in a communication cell with one base station is to assign the full bandwidth to a user for a certain period of time. Thus, users share time as a resource. The standard approach is that the base station decides itself about which user gets the bandwidth at what time. Because of interference phenomena, a neighboring cell's QoS (Quality of Service) can be influenced when the user is at the border of the cells. We consider the scenario when the base stations cooperate, i.e., they share the knowledge of positions of the users and their communication demands and the base stations decide together upon which base station communicates with the user and also at which time. The model we adapt is that the base stations adjust their power to reach the user. This results in the so called *interference disks*. An interference disk is a disk with its center placed at the communicating base station (which is modeled as a point in the Euclidean plane) and with a radius equal to the distance between the base station and the user that is communicating with the base station.

The optimization problem that we study in this thesis is motivated by the above example. Let B and U be sets of points in the Euclidean plane, representing the base stations and users, respectively. We consider discrete time $\{0, 1, 2, \ldots\}$ and we want to assign each user $u \in U$ to a base station $b \in B$ at time t_u such that there is no interference at the user's side, i.e., for every $u \in U$, there is exactly one interference disk intersecting u at time t_u, namely the interference disk formed by the base station that is assigned to u, among all interference disks that are considered at time t_u. The optimization goal is to minimize the maximum time t that is assigned to any user.

The problem is studied in Chapter 6.

A Note on a Divide and Conquer Approach in Writing a Thesis

We studied the OVSF-code assignment problem (Chapter 5) and Joint Base Station Scheduling problem (Chapter 6) together with Thomas Erlebach, Riko Jacob, Marc Nunkesser, Gábor Szabó and Peter Widmayer. As postgraduate regulations suggest, unique and original results are expected to be presented in a thesis of a PhD student. As both Marc Nunkesser and Gábor Szabó were PhD students and interested in using the results in their thesis, some kind of "I write this and you write that" splitting would be politically correct but would lose on the readability of the thesis. Moreover, some of the results were discovered in common discussions and joint investigations and cannot be easily contributed to one person. Therefore, with the agreement of all co-authors and Ph.D. supervisors, some of the results that are presented here appeared already in the theses of Gábor Szabó [96] and Marc Nunkesser [83] (Sections 5.4.2, 5.4.3, 5.5.2 and 5.5.3 for the OVSF-code assignment, and Sections 6.3.1, 6.3.2, 6.3.4, 6.3.5, 6.3.6, 6.4.1 and 6.4.1 for the Joint Base Station Scheduling problem). We believe that it is better to mention most of the results here, in order to make the reading fluent and to put the results in context. We stress, where appropriate, to whom the credits go and in which thesis the reader shall find the missing/detailed proofs/discussions on a certain topic.

Chapter 2

Notation, Terminology and Theory

In this chapter we want to introduce (and reference) the necessary minimum of notation and terminology that is used throughout this thesis. We suppose the reader is familiar with a general concept and terminology of discrete mathematics, graph theory, linear algebra and theory of algorithms. Nonetheless, we recall some of the elementary notion here and also present some less common terminology.

Graphs

A *graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* E. The vertices are also called *nodes*. The set of vertices and the set of edges of a graph G is also denoted by $V(G)$ and $E(G)$, respectively. For an *undirected* graph $G = (V, E)$, the edges are 2-element subsets of V. The number of vertices is denoted by $n := |V|$ and the number of edges is denoted by $m := |E|$. A *neighbour* of vertex $v \in V$ is a vertex $w \in V$ such that $\{v, w\}$ is an edge, i.e., $\{v, w\} \in E$. The number of neighbours of v is called the *degree* of v. The *maximum degree* of a graph G, denoted by Δ_G or Δ, is the maximum degree of vertices of G.

A *path* P in graph G is a sequence $v_1, v_2 \ldots, v_\ell$ of vertices from V such that no two vertices on the path are the same and for every $1 \leq i \leq \ell - 1$, $\{v_i, v_{i+1}\}$

is an edge, i.e., $\{v_i, v_{i+1}\} \in E$. The path P is said to be a path between v_1 and v_ℓ or a path from v_1 to v_ℓ. The *length* of a path P is the number of edges $\{v_i, v_{i+1}\}$ in that path, i.e., the number of vertices minus one. Graph G is said to be *connected* if for every two vertices $u, v \in V$ there is a path between u and v. The *distance* of two nodes u and v in graph G is the length of the shortest path between u and v, and is denoted by $d_G(u, v)$, or, if it is clear which graph we are referring to, by $d(u, v)$.

A *clique* is a graph $G = (V, E)$ such that for every $u, v \in V$, $\{u, v\}$ is an edge. A connected graph $G = (V, E)$ is a *tree* if it has $n - 1$ edges. In that case, for every u and v there is a unique path from u to v. A tree is often denoted as T (instead of G) and often has a special vertex—the *root*. Tree with root v is usually depicted in layers according to the distances of the vertices to the root, i.e., vertex u at distance i from the root v belongs to layer i. The height of tree T is the maximum of the distances from root v to all other nodes. A *binomial* tree T_k of order k is a tree with the follwing recursive definition: T_0 is just a single node; T_k, $k > 0$, is a node (root) with k neighbours, where the i-th neighbour, $0 \leq i < k$, is a root of binomial tree T_i.

Graph $H = (V', E')$ is a *subgraph* of graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. H is an *induced subgraph* of G if H is subgraph of G and for every $u, v \in V'$, if $\{u, v\} \in E$ then also $\{u, v\} \in E'$. An induced subgraph H of graph G with vertex set V' is denoted by $G[V']$.

The *chromatic number* of a graph $G = (V, E)$, denoted by $\chi(G)$, is the minimum number of colours such that each vertex of G is assigned one colour and for every edge $\{u, v\} \in E$ the colours assigned to u and v are different. The *clique number* of a graph $G = (V, E)$, denoted by $\omega(G)$, is the maximum number of vertices of G that induce a clique. An *independent set* of graph $G = (V, E)$ is a subset $V' \subseteq V$ such that for every $u, v \in V'$, $\{u, v\}$ is not an edge. The *independence number* of graph G, denoted by $\alpha(G)$, is the number of vertices in a maximum-size independent set of G. A *dominating set* of graph $G = (V, E)$ is a subset $V' \subseteq V$ such that for every vertex $v \in V$, v is in V' or a neighbour of v is in V'.

A graph G is called *perfect* if for every induced subgraph H the chromatic number $\chi(H)$ is equal to the clique number $\omega(H)$. The so-called strong perfect

graph theorem states that G is perfect if and only if G contains no odd *hole* and no odd *antihole*. A hole of G is a chordless cycle of length at least four. An antihole is a complement of such a cycle. The parity of the number of vertices of holes and antiholes determines whether the hole (antihole) is odd or even.

More on graph theory can be found e.g. in the standard textbook by Reinhard Diestel [46]. The first chapter therein deals entirely with the basic terminology.

Geometry

We now introduce some notation for the problems dealing with geometric objects. We assume all problems are considered in a 2-dimensional plane—this can be seen as a set $\mathbb{R} \times \mathbb{R}$—which is part of the 2-dimensional Euclidean space, denoted by $\mathbb{E}^2$ (the Euclidean space is also called Euclidean plane). The *points* are the elements of $\mathbb{R} \times \mathbb{R}$. The distance between two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$ is denoted by $d(p, q)$, $|p - q|$, or $||p - q||_2$ (also called the 2-norm). A *conflict* graph $G = (V, E)$ of a set D of objects is a graph for which the vertices V are in one-to-one correspondence with the objects of D and there is an edge $\{u, v\}$ in G if the corresponding objects are in *conflict*. We will be talking about conflict graphs of geometric objects (e.g., lines, circles, disks, rectangles, etc.), where there is a conflict between two geometric objects, if the two objects intersect. In such a case we say that G is a (*geometric*) *intersection* graph of D.

Complexity Theory and Optimization Problems

We make a very brief (and coarse) excursion into the basics of complexity theory, and approximation and online algorithms of optimization problems. A *decision problem* (such as the Satisfiability Problem) is a problem with a solution "yes" or "no". A decision problem P is given by all instances I_P of the problem together with a mapping $f : I_P \to \{\text{yes}, \text{no}\}$. An instance that maps to yes is called a yes-instance and an instance that maps to no is called a no-instance. A decision problem P is in $\mathcal{NP}$ if there is a nondeterministic polynomial time algorithm that *solves* the problem, i.e., if for any instance x of the problem the algorithm decides in polynomial time whether x is a yes-instance. A decision problem P is in $\mathcal{P}$ if there is a deterministic polynomial time algorithm that decides for any

instance x of P whether x is a yes-instance. A decision problem P_1 *reduces* to a decision problem P_2 if there exists a polynomial time deterministic algorithm that for any instance x of P_1 produces an instance y for P_2 such that x is a yes-instance if and only if y is a yes-instance. Reductions are used to establish how difficult two problems are: here, P_2 is at least as difficult as P_1 since an algorithm that solves P_2 can be used to solve P_1, losing only polynomial time in the reduction. A decision problem P is said to be *$\mathcal{NP}$-complete*, if P is in $\mathcal{NP}$ and every decision problem P' from $\mathcal{NP}$ can be reduced to P. A decision problem P is said to be *$\mathcal{NP}$-hard* if every decision problem P' from $\mathcal{NP}$ can be reduced to P. We switch now to complexity of optimization problems. An *optimization problem* P is characterized by

1. the set of *instances* I_P of the problem P,
2. a set of *feasible solutions* $S_P(i)$ for every instance $i \in I_P$,
3. *objective function* obj_P, that assigns a nonnegative rational number to each pair (i, s), where i is an instance and s is a feasible solution for i, i.e., $s \in S_P(i)$, and
4. the goal of the problem P, which can be either *minimization problem* or *maximization problem.*

For maximization problems, the objective function is also called the *profit function* and for minimization problems, the objective function is also called the *cost function.* An *optimal solution* $\text{OPT}_P(i)$ for an instance i of a minimization (maximization) problem P is a feasible solution that achieves the smallest (largest) objective function value. We use OPT for short, if it is clear what problem and instance we are talking about. We abuse the notation sometimes, and refer by OPT to the actual value of the optimal solution (and we make it clear what we mean by OPT when considering particular problems). An algorithm *solves* the optimization problem if it returns an optimal solution. In this thesis, we are interested in polynomial time algorithms only. Every optimization problem P can be naturally modified to a corresponding decision problem (called a decision version of P) by giving a bound on the optimal solution (which is given as part of the input). Clearly, a polynomial time algorithm for an optimization

problem P solves the decision version—simply compute the optimum solution and compare it to the given bound. Hence, hardness established for the decision problem carries over to the optimization problem. In general, an optimization problem P is said to be *$\mathcal{NP}$-hard*, if using an optimal solution of P can solve an $\mathcal{NP}$-complete decision problem P' in polynomial time.

An algorithm A is said to be a *factor δ approximation algorithm* (or δ-approximation algorithm) for a minimization (maximization) problem P, if, for each instance i of P, the algorithm runs in polynomial time and produces a feasible solution s such that $\text{obj}(i,s) \leq \delta \cdot \text{OPT}(i)$ ($\text{obj}(i,s) \geq \frac{1}{\delta} \cdot \text{OPT}(i)$), where $\text{OPT}(i)$ denotes the cost of an optimum solution of the instance i and δ is a function that for a given size of the input i gives a positive rational number. For algorithm A and an instance i of an optimization problem P, by $A(i)$ we usually denote the solution delivered by A or the cost of this solution. We always make clear which meaning of $A(i)$ we use. To make the life even more complicated, A itself can sometimes refer to the actual solution (or cost of the solution) of the algorithm.

Online optimization problems are characterized by partial knowledge—the input is given to the algorithm in parts, which can be seen as a sequence, one item of input at a time, and an *online* algorithm must decide how to act on incoming items without knowledge of future inputs (or if there is any future item at all). In a *competitive analysis* of online algorithms, the quality of the solution produced by an algorithm A on input i (denoted by $A(i)$) is compared with a solution of an optimal algorithm OPT that knows the whole input i in advance. (Such an algorithm is called *offline* algorithm.) An online algorithm is called *c-competitive* for an online minimization problem if for every finite input sequence i, $A(i) \leq c \cdot \text{OPT}(i)$. An online algorithm is called *weakly c-*competitive, if an additive constant α in the quality of $A(i)$ is allowed, i.e., if $A(i) \leq c \cdot \text{OPT}(i) + \alpha$ for every input i. Notice that we do not require the algorithms to run in polynomial time, but in practice the running time is always an issue. The competitive ratio of an algorithm for online maximization problems is defined similarly as for approximation algorithms. The analysis of an online algorithm is often seen as a "game" between two players—the online algorithm and an adversary. The adversary has full knowledge of the online

algorithm and is responsible for creating an input sequence that maximizes the competitive ratio of the algorithm.

If randomization is used in algorithms for optimization problems, the terms of approximation ratios and competitive ratios are defined for randomized algorithms as well. In all the above definitions, the actual cost $A(i)$ of the solution produced by an algorithm A on input i is substituted by the expected cost $\mathbf{E}(A(i))$ of the algorithm A on input i.

A *polynomial-time approximation scheme* (PTAS) is a family of approximation algorithms, one for every constant $\epsilon > 0$, with approximation ratio $1 + \epsilon$. A *fully polynomial-time approximation scheme* (FPTAS) is a family of approximation algorithms, one for every $\epsilon > 0$, with approximation ratio $1 + \epsilon$ and running time polynomial in $\frac{1}{\epsilon}$ and in the input size.

Different optimization problems admit different approximation guarantees. An optimization problem belongs to class $\mathcal{APX}$, if there exists a constant-factor approximation algorithm for the problem. Assuming $\mathcal{P} \neq \mathcal{NP}$, not every optimization problem belongs to $\mathcal{APX}$. Clearly, every problem for which there exists PTAS belongs to $\mathcal{APX}$. There are problems that cannot be approximated arbitrarily, unless $\mathcal{P} = \mathcal{NP}$ (i.e., there are problems in $\mathcal{APX}$ for which no PTAS exists). This result is a consequence of the celebrated PCP theorem. The theorem shows that $\mathcal{MAX}$ $\mathcal{SNP}$-hard problems do not admit PTAS. (Thus, to show that a problem does not admit PTAS we can show that the problem is $\mathcal{MAX}$ $\mathcal{SNP}$-hard.) $\mathcal{MAX}$ $\mathcal{SNP}$ is a special class of optimization problems that is defined using second-order logic, together with the notion of approximation-preserving reduction, and thus the notion of completeness and hardness. In general, to show hardness of approximation of a problem, a so called *gap introducing reduction* (or a *gap technique*) is often used. Let P' be an $\mathcal{NP}$-complete decision problem and let P be a minimization problem (for maximization the reduction works similarly). Let us suppose there is a reduction from P' to P that for every instance i' of problem P' creates an instance i of problem P such that the value $\mathrm{OPT}(i)$ of the optimum solution of i is $c(i')$ if i' is a yes-instance, and $c(i')(1+\mathrm{gap})$ if i' is a no-instance (c is some function that can be computed in polynomial time). Then clearly there is no ρ-approximation algorithm for P, where $\rho < 1 + \mathrm{gap}$.

An exact algorithm A for an optimization problem is called *fixed parameter tractable with respect to parameter* k, if it solves the problem in time bounded by $f(k) \cdot n^{o(k)}$, where f is an arbitrary function.

A good introduction to algorithms is [37]. A standard reference for complexity theory and $\mathcal{NP}$-complete problems is [62]. Approximation algorithms are covered in [14, 101], and online algorithms and online computation is the topic of [21]. The class $\mathcal{MAX}$ $\mathcal{SNP}$ is defined in [86]. The PCP theorem was first proved in [9]. For more on the topic of fixed parameter tractable algorithms, a good starting point is [48] or a more recent [82].

Linear Programming

Linear programming is a technique that provides a unified way to describe and solve a plentiful amount of optimization problems. A *linear program* (LP for short) is an optimization problem, where the aim is to minimize a linear function $c \cdot x^T$, where c and x are n-dimensional vectors (that can be seen as row-vectors), c is a constant vector and x is the vector of variables. The values of x have to satisfy a set of constraints in the form of linear inequalities $Ax^T \geq b^T$, linear equations $A'x^T = b'^T$ and also constraints $x_i \geq 0$, where A and A' are matrices, b and b' are vectors, and $x = (x_1, x_2, \ldots, x_n)$. *Integer linear programming* (ILP for short) is a linear programming problem, where additional constraints on x are imposed, which are not in the form of linear equations—each x_i has to be an integer, i.e., $x_i \in \mathbb{Z}$. We note that LP can be solved in polynomial time, if the number of constraints is polynomial in n, whereas ILP does not possesses this property, unless $\mathcal{P}$=$\mathcal{NP}$. In case the number of constraints of LP is more than polynomial, the problem can be solved in polynomial time, if there exists a polynomial time *oracle* that for any x identifies the constraints that are violated.

Linear programming and its use in optimization problems is covered e.g. in [85] and [91].

Chapter 3

Weighted Dominating Sets in Unit Disk Graphs

We study the problem of finding a minimum-weight (connected) dominating set in the given unit disk graph—for a given set $\mathcal{D}$ of disks in the plane, find a minimum subset $\mathcal{D}' \subseteq \mathcal{D}$, such that every disk from $\mathcal{D} \setminus \mathcal{D}'$ intersects a disk from $\mathcal{D}'$. The problem is mainly motivated by (besides the fact that it is a classical problem in graph theory on its own) its applications in the field of wireless ad-hoc networks. These networks consist of a set of devices that can communicate with each other (and with no-one else) in a wireless manner. It can be just a couple of laptops communicating with each other, but much of the recent research attention has been devoted to devices that are usually very small and have limited power, memory and computational capabilities (the networks consisting of these devices are called sensor networks). Ad-hoc networks lack any centralized network management. Unlike wired networks or cellular networks, no physical backbone infrastructure is present in wireless ad-hoc networks. Communication between two nodes is established either via direct radio transmission (if the parties are close enough, i.e., in the reach of their signalling strength)—which is called a single-hop (or one-hop) communication, or via passing the message onto intermediate nodes—which is called multi-hop (or many-hop) communication. The single-hop connectivity, i.e., the topology of a wireless ad-hoc network, can be modeled via graph $G = (V, E)$—nodes V represent the wireless devices and

there is an edge between two nodes u and v in the edge set E if the devices are in the transmission range of each other. We assume for simplicity the devices to be in the plane. Mostly, all devices have the same technical capabilities, including their transmission range, and therefore the underlying graph G is a unit disk graph, i.e., a conflict graph of disks (in the plane) of unit radius.

A vital question is how devices send messages to each other, i.e., how to establish a routing mechanism in such networks. Although no physical backbone infrastructure is present, a virtual backbone can be formed for routing purposes. Dominating sets of the corresponding unit disk graph have been proposed for construction of routing backbones (see, e.g., [41, 6]). A message that is broadcast by all nodes of a dominating set will be received by all nodes of the network. Therefore, a small connected dominating set is an energy-efficient routing backbone. Recent work has emphasized that ad-hoc networks are often heterogeneous as different nodes have different capabilities. Therefore it is meaningful to assign weights to the nodes (giving small weight to nodes that have a large remaining battery life, for example) and aim to determine a (connected) dominating set of small weight [102].

3.1 Problem Definition

For a given undirected graph $G = (V, E)$, a subset $D \subseteq V$ of its vertices is called a *dominating set* if every vertex in V is contained in D or has a neighbour in D. A vertex in D is called a *dominator*. A dominator *dominates* itself and all its neighbours. The goal of the *minimum dominating set problem* (MDS) is to compute a dominating set of smallest size. In the weighted version, the *minimum-weight dominating set problem* (MWDS), each vertex of the input graph is associated with a weight, and the goal is to compute a dominating set of minimum weight.

A dominating set $D \subseteq V$ is called a *connected dominating set* in the graph $G = (V, E)$ if the subgraph induced by D is connected. The minimum connected dominating set problem (MCDS) and minimum-weight connected dominating set problem (MWCDS) are defined in the obvious way.

We are interested in MWDS and MWCDS problems in unit disk graphs in

the following. A graph $G = (V, E)$ is called *unit disk graph* if every node is associated with a disk in the plane of a unit radius and there is an edge between two nodes if the corresponding disks intersect. The set of unit disks in the plane are called a *disk representation* of G. Our algorithm for MWDS and MWCDS in unit disk graphs works for the graphs whose disk representation is given as the input.

3.2 Related Work and New Contributions

For general graphs, MDS (and therefore MWDS) is $\mathcal{NP}$-hard [62]. Furthermore, MDS for general graphs is known to be equivalent to the *set cover* problem, implying that it can be approximated within a factor of $O(\log n)$ for graphs with n vertices using a greedy algorithm (see, e.g., [101]), but no better unless all problems in $\mathcal{NP}$can be solved in $n^{O(\log \log n)}$ time [57]. Approximation ratio $O(\log n)$ can also be achieved for the weighted set cover problem and thus for MWDS. The best known approximation ratio for MWCDS in general graphs is $O(\log n)$ as well [68].

We are, however, concerned with MWDS and MWCDS in a special class of graphs: *unit disk graphs*. Clark et al. [35] have proved that MDS is $\mathcal{NP}$-hard for unit disk graphs. Lichtenstein [77] has shown that MCDS is $\mathcal{NP}$-hard for unit disk graphs. Constant-factor approximation algorithms for MDS and MCDS in unit disk graphs were given by Marathe et al. [79]. For MDS in unit disk graphs, a PTAS was presented by Hunt et al. [72], based on the shifting strategy [15, 70]. These algorithms, however, do not extend to the weighted version. In particular, the PTAS is very much based on the fact that the optimal dominating set for unit disks in a $k \times k$ square has size at most $O(k^2)$ and can thus be found in polynomial time using complete enumeration if k is a constant. In the weighted case, there is no such bound on the size of an optimal (or near-optimal) solution, as an optimal solution may consist of a large number of disks with tiny weight. For MCDS in unit disk graphs, a PTAS was presented in [31]. For the special case of unit disk graphs with bounded density, asymptotic fully polynomial-time approximation schemes (with running time polynomial in $\frac{1}{\varepsilon}$ and in the size of the input, but achieving ratio $1+\varepsilon$ only for large enough inputs) were presented

for MDS and MCDS in [100].

Wang and Li [102] give distributed algorithms for MWDS and MWCDS in unit disk graphs that achieve approximation ratio $O(\min\{\log\Delta, \sigma\})$, where Δ is the maximum degree of the graph and σ is the ratio of the maximum weight to the minimum weight of a disk. Note that these approximation ratios are not better than the known ratios for general graphs in the worst case.

In this thesis, we present the first constant-factor approximation algorithms for MWDS and MWCDS in unit disk graphs. The results were published in [7] and appeared in more details in [8]. Our algorithm for MWDS solves the problem in two steps. First, we reduce MWDS in unit disk graphs to the problem of covering a set of points that are located in a small square using a minimum-weight set of unit disks. In the reduction we lose only a constant factor in the approximation ratio. Then, we present a constant-factor approximation algorithm for the latter problem using enumeration and dynamic programming techniques exploiting the geometry of unit disks. To solve the MWCDS problem, we first compute an $O(1)$-approximation for the MWDS problem and then use an approach based on a minimum spanning tree calculation to add disks to the solution in order to make the dominating set connected.

We also show that our techniques yield a constant-factor approximation algorithm for the weighted disk cover problem for unit disks, constant-factor approximation for the weighted rectangle cover problem and for the weighted dominating set in conflict graphs of rectangles, and a 3-approximation algorithm for the special case of the forwarding set problem (see Section 3.7 for a definition of this problem).

The remainder of the chapter is structured as follows. Our top-level approach to solving MWDS, which consists of breaking the problem into subproblems in small squares, is presented in Section 3.3. In Section 3.4, we show how the subproblem can be reduced to a special disk cover problem and give a constant-factor approximation algorithm for the latter problem. We also describe how this implies a constant-factor algorithm for the general weighted disk cover problem with unit disks. Section 3.5 shows how we can make a dominating set connected while incurring a cost that is bounded by a constant factor times the cost of the optimal connected dominating set. In Section 3.7, we apply our techniques to

obtain a 3-approximation algorithm for the forwarding set problem.

3.3 Algorithm for Minimum-Weight Dominating Sets

Let an instance of MWDS in unit disk graphs be given by a set $\mathcal{D}$ of weighted unit disks in the plane. The weight of disk $d \in \mathcal{D}$ is denoted by $w_d \geq 0$. Each disk has radius 1 and is specified by the coordinates of its center. For $U \subseteq \mathcal{D}$, we write $w(U)$ for $\sum_{d \in U} w_d$.

We partition the plane into squares S_{ij} of side length $\mu < 1$, which is the parameter of our algorithm; we can set $\mu = 0.999$. The square S_{ij}, for $i, j \in \mathbb{Z}$, contains all points (x, y) with $i\mu \leq x < (i+1)\mu$ and $j\mu \leq y < (j+1)\mu$.

For a square S_{ij} that contains at least one disk center, let $\mathcal{D}_{ij}$ be the set of disks in $\mathcal{D}$ whose center is in S_{ij}. Let $N(\mathcal{D}_{ij})$ denote the set of all disks in $\mathcal{D} \setminus \mathcal{D}_{ij}$ that intersect a disk in $\mathcal{D}_{ij}$. Thus, $N(\mathcal{D}_{ij})$ contains the neighbouring disks of $\mathcal{D}_{ij}$ in the underlying unit disk graph.

We focus on solving the following subproblem of MWDS in every square S_{ij}: Find a minimum-weight set of disks in $\mathcal{D}_{ij} \cup N(\mathcal{D}_{ij})$ that dominates all disks in $\mathcal{D}_{ij}$. We show in Section 3.4 that such a special case of MWDS in unit disk graphs admits a 2-approximation algorithm, i.e., our algorithm finds a solution U_{ij} for which $w(U_{ij}) \leq 2 \cdot w(\mathrm{OPT}_{ij})$, where OPT_{ij} is an optimal solution to the MWDS subproblem for square S_{ij}. Our algorithm outputs in the end the union of all sets U_{ij} that have been computed. It is clear that this yields a dominating set.

Theorem 3.1 *There is a constant-factor approximation algorithm for the minimum weight dominating set problem in unit disk graphs.*

Proof. Let U be the dominating set that is computed by the algorithm that was described above. The weight of U is at most $\sum w(U_{ij})$. Here and in the following, the summation is over all squares S_{ij} that contain at least one disk center. We want to compare this weight to $w(\mathrm{OPT})$, the weight of a minimum-weight dominating set OPT for the whole instance. Recall that OPT_{ij} denotes the optimum solution to the MWDS subproblem for square S_{ij}. As we will

present a 2-approximation algorithm for each subproblem in Section 3.4, we have $w(U_{ij}) \leq 2 \cdot w(\mathrm{OPT}_{ij})$. Let $\mathrm{OPT}[S_{ij}] = \mathrm{OPT} \cap (\mathcal{D}_{ij} \cup N(\mathcal{D}_{ij}))$. Note that $\mathrm{OPT}[S_{ij}]$ is a feasible solution to the subproblem for square S_{ij} and therefore we have $w(\mathrm{OPT}_{ij}) \leq w(\mathrm{OPT}[S_{ij}])$.

We get $w(U) \leq \sum w(U_{ij}) \leq 2 \sum w(\mathrm{OPT}_{ij}) \leq 2 \sum w(\mathrm{OPT}[S_{ij}])$. The sum $\sum w(\mathrm{OPT}[S_{ij}])$ adds the costs of solutions $\mathrm{OPT}[S_{ij}]$ for all squares S_{ij} that contain at least one disk center. Note that a disk d in OPT can be in $\mathrm{OPT}[S_{ij}]$ only if it is part of the optimal dominating set for the subproblem S_{ij}, which can happen only if the center of d is in S_{ij} or it intersects a disk with center in S_{ij}. Therefore, the distance between the center of d and the square S_{ij} is at most 2. Consequently, there are only $O(1/\mu^2)$ squares S_{ij} such that d can be in $\mathrm{OPT}[S_{ij}]$. More precisely, all such squares must be fully contained in a disk of radius $2 + \sqrt{2}\mu$ around the center of d, and for $\mu = 0.999$ that disk can contain at most $\lfloor (2 + \sqrt{2}\mu)^2 \pi / \mu^2 \rfloor = 36$ such squares. This means that the number of times each disk in OPT contributes its weight to $\sum w(\mathrm{OPT}[S_{ij}])$ is bounded by 36. We get $\sum w(\mathrm{OPT}[S_{ij}]) \leq 36 \cdot w(\mathrm{OPT})$ and, thus, $w(U) \leq 2 \sum w(\mathrm{OPT}[S_{ij}]) \leq 72 \cdot w(\mathrm{OPT})$. □

As a direct consequence of the proof, the approximation ratio of our algorithm is 72.

3.4 Solving the Subproblem for a Small Square

In this section we present a 2-approximation algorithm for the following problem: Given a $\mu \times \mu$ square S_{ij}, where $\mu < 1$, and the set of disks $\mathcal{D}_{ij} \cup N(\mathcal{D}_{ij})$, compute a minimum-weight set of disks that dominates all disks in $\mathcal{D}_{ij}$.

Let OPT_{ij} denote the set of disks in an optimal solution for the problem. In the following, we will often write that the algorithm "guesses" certain properties of OPT_{ij}. Such guesses are to be interpreted as follows: The algorithm tries all possible choices for the guess (there will be a polynomial number of such choices) and computes a solution for each choice. In the end, the algorithm outputs the solution of minimum weight among all solutions found in this way. Some guesses may not lead to feasible solutions; such guesses are discarded. In the analysis, we concentrate on the solution in which the algorithm makes the

right guess about OPT_{ij}. It then suffices to show that the solution the algorithm finds for that guess is a constant-factor approximation of the optimum, because the solution output by the algorithm in the end will be at least as good as the one it finds for that guess.

First, the algorithm guesses the largest weight w of a disk in OPT_{ij}. Since there are n disks in our instance, there are at most n different values for this guess, i.e., the algorithm tries at most n different values. Having the largest weight w of a disk in OPT_{ij}, the algorithm checks if there is a disk of weight w in $\mathcal{D}_{ij}$. If this is the case, the algorithm simply outputs that disk as the solution. Observe that this solution is optimal, because the disk has its center in square S_{ij} and therefore dominates all disks in S_{ij} and has the right (correctly guessed) weight. If there is no disk of weight at most w in $\mathcal{D}_{ij}$, we know that OPT_{ij} consists entirely of disks in $N(\mathcal{D}_{ij})$ of weight at most w. In this case, we first discard all disks from $N(\mathcal{D}_{ij})$ that have weight larger than w and end up at the following problem: Find a set of disks of minimum weight from $N(\mathcal{D}_{ij})$ that dominates all disks in $\mathcal{D}_{ij}$. A disk d_1 from $N(\mathcal{D}_{ij})$ dominates a disk d_2 from $\mathcal{D}_{ij}$ if and only if the distance of the centers of d_1 and d_2 is at most 2. Therefore, we can increase the radius of the disks in $N(\mathcal{D}_{ij})$ from 1 to 2 and reduce the radius of the disks in $\mathcal{D}_{ij}$ from 1 to 0 and obtain an equivalent problem: If $\mathcal{D}'$ denotes the set containing the enlarged version of the disks in $N(\mathcal{D}_{ij})$ and $\mathcal{P}$ denotes the set of centers of the disks in $\mathcal{D}_{ij}$, we need to find a minimum-weight subset of the disks in $\mathcal{D}'$ that covers all points in $\mathcal{P}$. Furthermore, we can re-normalize the setting so that the disks in $\mathcal{D}'$ have radius 1. The re-normalized square S_{ij} is now a $\delta \times \delta$ square, with $\delta = \mu/2 < 1/2$. Therefore, the problem to be solved can be stated as follows:

> **Disk cover in a small square:** Given a set $\mathcal{P}$ of points in a $\delta \times \delta$ square S, where $\delta < 1/2$, and a set $\mathcal{D}'$ of weighted unit disks, find a minimum-weight subset of $\mathcal{D}'$ that covers all points in $\mathcal{P}$.

In the following subsection, we will present a 2-approximation algorithm for this problem. In view of the discussion above, this implies that we have a 2-approximation algorithm for the problem of computing a minimum-weight set of disks that dominates all disks in $\mathcal{D}_{ij}$ for a given $\mu \times \mu$ square S_{ij}, and this is the ingredient that we needed in the previous section to obtain the constant-factor

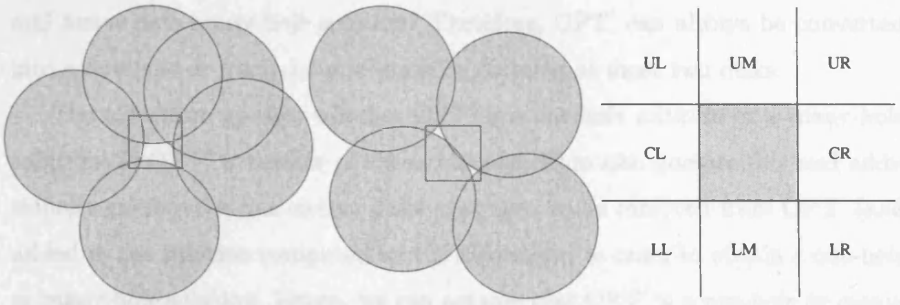


Figure 3.1: One-hole solution (left), many-hole solution (middle), naming of regions (right).

approximation algorithm for MWDS in unit disk graphs.

3.4.1 Algorithm for Disk Cover in a Small Square

We are given a set $\mathcal{P}$ of points in a $\delta \times \delta$ square S, $\delta < 1/2$, and a set $\mathcal{D}'$ of n weighted unit disks, and we want to find a minimum-weight subset of $\mathcal{D}'$ that covers all points in $\mathcal{P}$. Let OPT$'$ denote a set of disks constituting an optimal solution to this problem.

Let C be the area covered by the union of the disks in OPT$'$. A *hole* of OPT$'$ is defined to be a topological component of $S \setminus C$. Intuitively, if S was a glass window and the disks in OPT$'$ were to cover parts of this window, the holes would be the connected regions where one can still see through the window.

Definition 3.1 OPT$'$ *is a* one-hole solution *if it has exactly one hole and each disk in* OPT$'$ *forms part of the boundary of that hole (and that part consists of more than* 1 *point).* OPT$'$ *is a* many-hole solution *if it has at least two holes.*

Definition 3.1 is illustrated in Fig. 3.1. If OPT$'$ is neither a one-hole solution nor a many-hole solution, it must be of one of the following types: Either OPT$'$ has no hole at all, or it has one hole but not all disks in OPT$'$ form part of the boundary of the hole. If OPT$'$ does not have a hole, we can delete one disk d from OPT$'$ (and remove all points in d from $\mathcal{P}$) to obtain a solution with at least one hole. If OPT$'$ has one hole but not all disks are on the boundary of the hole, let d' be a disk that is not on the boundary of the hole. If we delete d' from OPT$'$ (and the corresponding points from $\mathcal{P}$), we have at least two holes

and arrive at a many-hole solution. Therefore, OPT′ can always be converted into a one-hole or many-hole solution by deleting at most two disks.

The algorithm guesses whether OPT′ is a one-hole solution or a many-hole solution. If OPT′ is neither of these, the algorithm also guesses this and additionally guesses the one or two disks that need to be removed from OPT′ (and added to the solution computed by the algorithm) in order to obtain a one-hole or many-hole solution. Hence, we can assume that OPT′ is a one-hole or many-hole solution and that the algorithm has guessed correctly which of the two is the case. In each of the two cases, we will encounter subproblems that can be solved by dynamic programming, as stated in the following lemma.

Lemma 3.2 *Let $\mathcal{P}$ be a set of points located in a strip between the horizontal lines $y = y_1$ and $y = y_2$ for some $y_1 < y_2$. Let $\mathcal{D}$ be a set of weighted unit disks with centers above the line $y = y_2$ (upper disks) or below the line $y = y_1$ (lower disks). Furthermore, assume that the union of the disks in $\mathcal{D}$ contains all points in $\mathcal{P}$. Then a minimum-weight subset of $\mathcal{D}$ that covers all points in $\mathcal{P}$ can be computed in polynomial time.*

Proof. A solution consists of some upper disks and some lower disks. All upper disks in the solution intersect the line $y = y_2$, and all lower disks the line $y = y_1$. We view the upper halfplane bounded by $y = y_2$ and the lower halfplane bounded by $y = y_1$ as special cases of disks (with weight 0). For a set $\mathcal{U}$ of upper disks and a point $p \in \mathcal{P}$ with x-coordinate x_p, we say that an upper disk $u \in \mathcal{U}$ is *active* at x_p if its lowest intersection point with the vertical line $x = x_p$ has the smallest y-coordinate among all lowest intersection points of disks $u' \in \mathcal{U}$ with that line. If there are two or more active upper disks at $x = x_p$ by this definition, we consider only the one with leftmost center. For lower disks, active disks are defined similarly (i.e., having an intersection point with $x = x_p$ of largest y-coordinate). For a given solution and a given x-coordinate x_p, there is one active upper disk and one active lower disk at x_p (and each of these could also be the respective halfplane, as mentioned above). The algorithm computes a table T_p for every point $p \in \mathcal{P}$, in order of non-decreasing x-coordinates. For ease of presentation, we assume that no two points have the same x-coordinate. Let $p_1, p_2, \ldots, p_k$ denote the points of $\mathcal{P}$ in order of increasing x-coordinates. For an upper disk u and a lower disk d, the table entry $T_{p_i}(u, d)$ denotes the

optimal weight of a solution that covers all points from p_1 up to p_i and has u and d as the active upper and lower disk, respectively, at x_{p_i}. (If u and d do not cover p_i, we say that u, d is not feasible for p_i and set the table entry to ∞.) The table T_{p_1} can be initialized by setting $T_{p_1}(u,d) = w_u + w_d$ for all pairs of disks u and d that cover p_1 and $T_{p_1}(u,d) = \infty$ if u or d does not cover p_1. Once the tables for $p_1, \ldots, p_{i-1}$ have been computed, the table entries $T_{p_i}(u,d)$ for all feasible disks u and d for p_i can be computed as follows:

$$T_{p_i}(u,d) = \min\{T_{p_{i-1}}(u',d') + [u \neq u'] \cdot w_u + [d \neq d'] \cdot w_d \mid u', d' \text{ feasible for } p_{i-1}\}$$

Here, the term $[u \neq u']$ is 1 if $u \neq u'$, and 0 otherwise (and similarly for $[d \neq d']$). Intuitively, the equation is based on the observation that an optimal solution covering $p_1, \ldots, p_i$ with u and d as active disks for x_{p_i} can be obtained by adding u and d to an optimal solution corresponding to some $T_{p_{i-1}}(u',d')$, where the weight of u or d needs to be added only if x_{p_i} is the first x-coordinate for which u or d is active. The correctness of the calculation in the case of unit disks follows from the fact that an upper or lower disk can be active in the solution only for points in $\mathcal{P}$ that are consecutive (except if the disk is actually the lower or upper halfplane mentioned above, but these special disks have weight 0 and therefore do not cause problems if their weight is added each time they become active). The weight of an optimal solution for the disk cover problem can be found by locating the minimum value $T_{p_k}(u,d)$ among all feasible disks u, d for p_k. The solution itself can be found using standard bookkeeping techniques. □

In the following two subsections, we deal with the one-hole case and the many-hole case, respectively.

One-hole Solutions

Assume that OPT′ is a one-hole solution. The boundary of the hole is formed by disks from OPT′ and, potentially, some parts from sides of the square S (we view the latter as special kinds of disks with weight 0 and infinite radius, i.e., halfplanes, and do not treat them explicitly in the following). All disks in OPT′ have their centers outside S. Using the lines that are the extensions of the sides of S, we can partition the plane outside S into 8 regions in the natural way (see

also Fig. 3.1): upper left region (UL), upper middle region (UM), upper right region (UR), central right region (CR), lower right region (LR), lower middle region (LM), lower left region (LL), and central left region (CL). The upper region (U) is the union of UL, UM and UR, and similarly for the lower region (L).

If we follow the boundary of the hole in counterclockwise direction, we will encounter disks with center in CL, then disks with center in L, then disks with center in CR, then disks with center in U. (This description of hole is now given as an intuition rather then an obvious fact; we do not use this fact anywhere in our assumptions.) The points on the boundary that are in the intersection of two consecutive disks on the boundary are called *corners*. Each corner is *determined* by two disks (the disks on whose boundaries it lies).

Among all corners that are determined by at least one disk whose center is in CL, let p_ℓ denote the one with the smallest y-coordinate and let p_u denote the one with the largest y-coordinate. Let p'_ℓ and p'_u be defined analogously with respect to CR. (The case where no part of the boundary of the hole is created by disks with center in CL or CR is easier and is not treated in detail here.) The algorithm guesses the corners p_ℓ, p_u, p'_ℓ and p'_u and the pairs of disks determining them. As there are only $O(n^2)$ pairs of disks, the number of potential guesses is polynomial.

Let d_L be the unit disk that has p_ℓ and p_u on the boundary and has its center to the left of the line $\overline{p_\ell p_u}$. Note that in general d_L is not a disk that is part of the input of the problem. Let d_ℓ and d_u be the disks from OPT$'$ that have their center in CL and contain p_ℓ and p_u, respectively, on the boundary. Let x be the intersection point of the boundaries of d_ℓ and d_u that is closer to S. Let $\mathcal{L}$ be the connected region that is delineated by the boundary of d_L between p_u and p_ℓ, and by the boundary of d_ℓ between x and p_ℓ, and by the boundary of d_u between p_u and x. See Fig. 3.2 (top) for an illustration.

Lemma 3.3 *The only disks in* OPT$'$ *that intersect* $\mathcal{L}$ *have their center in CL or in the union of UR, CR and LR. Furthermore, no disk from* OPT$'$ *with center in CL can cover a point outside* $\mathcal{L}$ *that is not already covered by* d_u *or* d_ℓ.

Proof. As p_u and p_ℓ are on the boundary of the hole, no disk in OPT$'$ can contain p_u or p_ℓ in its interior. Hence, any disk d from OPT$'$ that intersects $\mathcal{L}$

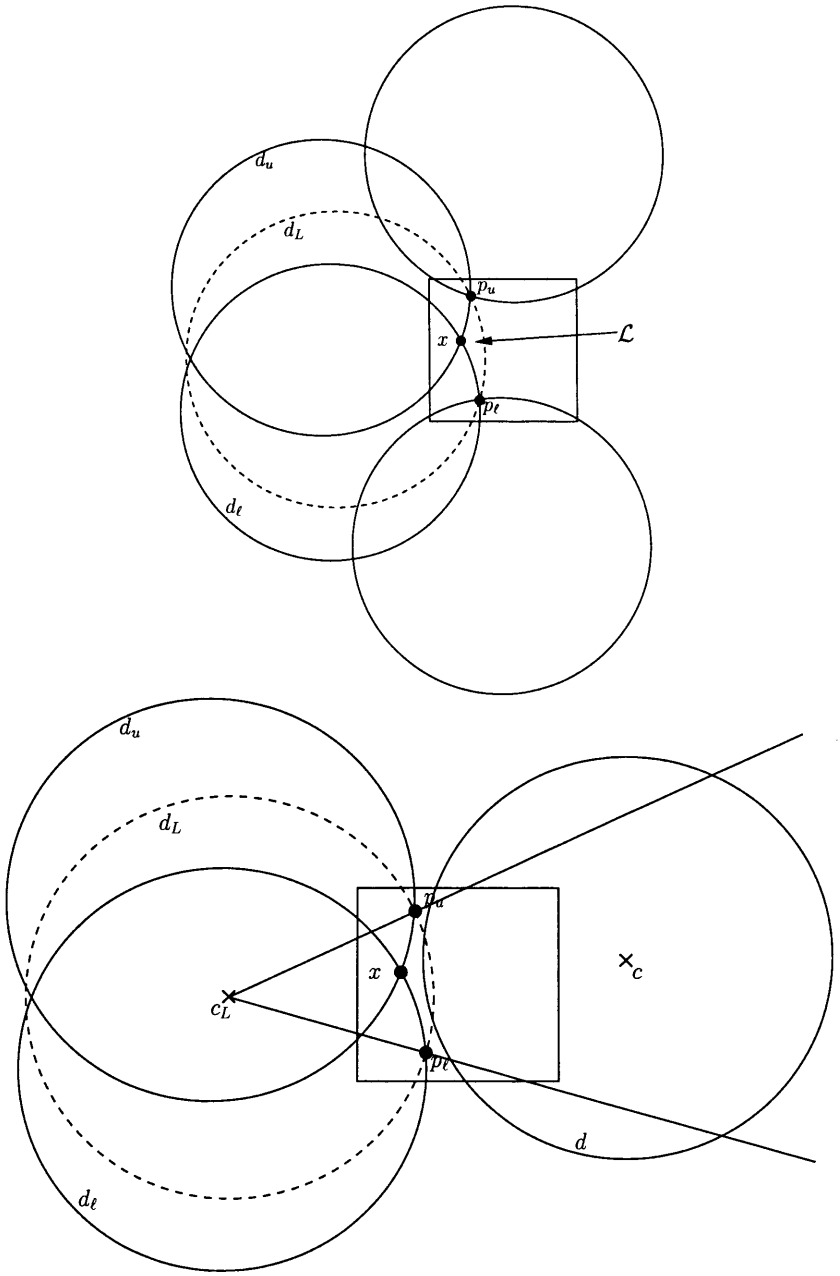


Figure 3.2: The region $\mathcal{L}$ is defined by parts of the boundaries of disk d_L, drawn dashed, and disks d_u and d_ℓ (top). A disk d with center not in CL from OPT′ intersecting $\mathcal{L}$ must have its center in the cone of two halflines starting at the center c_L of d_L and passing through p_u and p_ℓ, respectively (bottom).

must either have its center to the left of the line $\overline{p_\ell p_u}$ and intersect the parts of the boundaries of d_ℓ and d_u that define $\mathcal{L}$, or it must have its center to the right of the line $\overline{p_\ell p_u}$ and intersect the boundary of $\mathcal{L}$ twice on the part that is also a boundary of d_L. In the former case, the y-coordinate of the center of d must lie between the y-coordinates of the centers of d_ℓ and d_u, and hence d must have its center in CL. (To see this, consider the disk d' that is obtained from d by shifting it horizontally to the right until it first contains p_u or p_ℓ on its boundary; observe that the disk d_u can be rotated around p_u until it becomes identical to d', with its center continuously moving downward; the same argument can be applied to the disk d_ℓ and shows that the center of d' must have larger y-coordinate than the center of d_ℓ. By the same argument, we also have that c_L must lie in CL.) In the latter case, the center c of d must lie in the cone of points between the halflines starting at the center c_L of d_L and passing through p_ℓ and p_u, respectively, see Fig. 3.2 (bottom). We want to show that c cannot be in UM or LM. Assume for a contradiction that c is in UM (the case for LM is similar). The slope of the line connecting c_L and p_u is at most $\delta/\sqrt{1-\delta^2}$. Therefore, the largest y-coordinate of a point in the intersection of the cone and UM is bounded by $y_{p_u} + \delta^2/\sqrt{1-\delta^2}$, so the distance between p_u and any point in that intersection is at most $\delta/\sqrt{1-\delta^2}$ (see Fig. 3.3 for an illustration). Hence, for $\delta < \sqrt{2}/2$ (and we even have $\delta < 1/2$), a unit disk

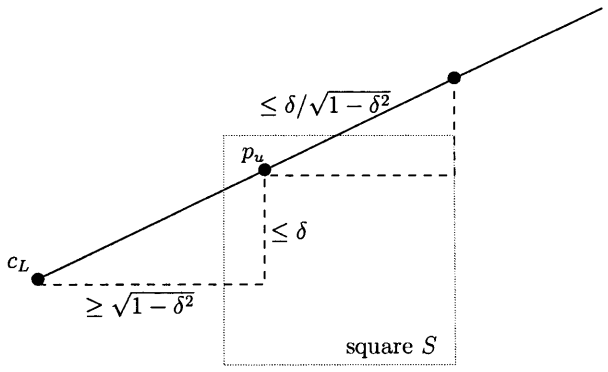


Figure 3.3: Any disk with center in the cone and in UM contains point p_u, for $\delta < \sqrt{2}/2$

with center in that intersection must contain p_u. Thus, c cannot be in UM, as d would then contain p_u in its interior. Similarly, we get that c cannot be in LM. Furthermore, c clearly cannot be in UL or LL, as it must be to the right

of p_u. Hence, we have shown that c must be in the union of UR, CR and LR.

We have shown that the only disks in OPT$'$ that intersect $\mathcal{L}$ have their center in CL or in the union of UR, CR and LR. It remains to show that no disk from OPT$'$ with center in CL can cover a point outside $\mathcal{L}$ that is not already covered by d_u or d_ℓ. Let d' be a disk from OPT$'$ with center in CL. All disks from OPT$'$ are on the boundary of the hole, and p_u and p_ℓ are the topmost and lowest corners, respectively, that are determined by at least one disk with center in CL. Therefore, d' must appear on the boundary of the hole between p_u and p_ℓ. This implies that $d' \setminus (d_u \cup d_\ell)$ consists of one region that is contained in $\mathcal{L}$ and a second region that is outside the square S (and cannot contain any points from $\mathcal{P}$). This establishes the claim. □

Similar to $\mathcal{L}$, we can define a region $\mathcal{R}$ with respect to CR, p'_ℓ and p'_u, and the analogue of Lemma 3.3 holds for $\mathcal{R}$.

Let $\mathcal{P}'$ be the set of points that is obtained from $\mathcal{P}$ by removing the points that are contained in one of the disks determining the four corner points guessed by the algorithm. For the points in $\mathcal{P}' \cap (\mathcal{L} \cup \mathcal{R})$, we can compute an optimal disk cover using Lemma 3.2 (rotated by 90°), since the points are contained in the vertical strip containing S and the only disks that need to be considered for covering them have their center to the left or to the right of the strip. The remaining points in $\mathcal{P}'$ can only be covered by disks with center in U or in L by OPT$'$, hence we can again compute an optimal disk cover for them using Lemma 3.2. If we output the union of the two disk covers, we have clearly computed a 2-approximation to the overall disk cover problem in this square.

Many-hole Solutions

Now we consider the case that OPT$'$ is a many-hole solution. For such a case, there must be two disks $d_1, d_2 \in$ OPT$'$ such that $S \setminus (d_1 \cup d_2)$ consists of two disjoint regions and each of these two regions contains a hole of OPT$'$. (As a special case, d_1 or d_2 could be any halfplane that touches a side of S but does not contain S; in this case, we would have a single disk from OPT$'$ that intersects the square in such a way that two holes are created.) We use a new coordinate system in which the y-axis contains the centers c_1 and c_2 of d_1 and d_2, respectively, and the intersection points of the boundaries of d_1 and d_2 are

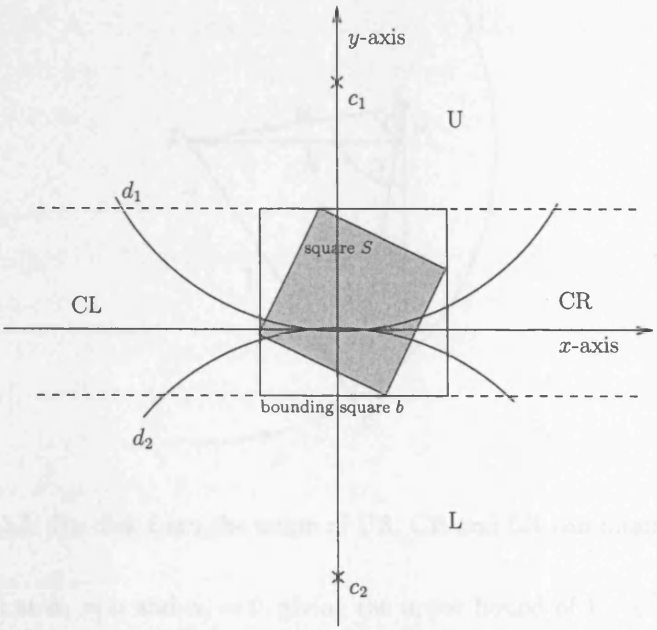


Figure 3.4: New coordinate system for the many-hole case.

on the x-axis. Let b be the smallest axis-parallel square containing the (rotated) square S. Let δ' be the side length of b. Note that $\delta' \leq \delta\sqrt{2} < \sqrt{2}/2$. See Fig. 3.4 for an illustration. As for the one-hole case, we partition the plane outside b into regions UL, UM, UR, CR, LR, LM, LL, CL, and we define regions U and L as before.

The disks d_1 and d_2 create two holes in S; we refer to the left hole as LH, and to the right hole as RH. Because OPT$'$ is a superset of $\{d_1, d_2\}$, OPT$'$ may contain more than two holes, but all the holes in OPT$'$ are contained in either LH or RH.

We begin with some observations: First of all, for any point with coordinates (x, y) that is contained in LH or RH, we have $|y| < 1 - \sqrt{1 - x^2}$. Furthermore, for any two points such that one is from LH and one from RH, the y-distance between the points is at most $1 - \sqrt{1 - \alpha^2}$, where α is the x-distance between the points. This follows from the following computation. Assume the first point has coordinates $(-\alpha_1, y_1)$ and the second point (α_2, y_2), for some $\alpha_1, \alpha_2 \geq 0$. We have $\alpha_1 + \alpha_2 = \alpha \leq \delta'$, and the y-distance of the points is bounded by $|y_1| + |y_2| \leq 1 - \sqrt{1 - \alpha_1^2} + 1 - \sqrt{1 - (\alpha - \alpha_1)^2}$. We find that this expression is

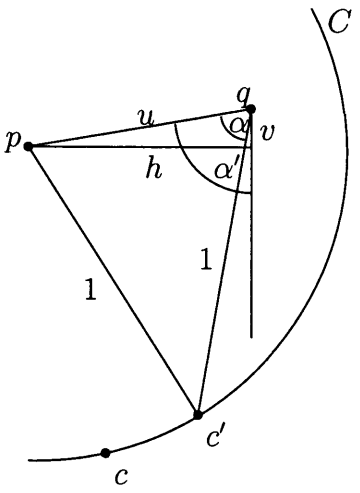


Figure 3.5: No disk from the union of UR, CR and LR can intersect LH.

maximized at $\alpha_1 = \alpha$ and $\alpha_1 = 0$, giving the upper bound of $1 - \sqrt{1-\alpha^2}$.

Lemma 3.4 *In* OPT′, *no disk with center in the union of UR, CR and LR can intersect LH and no disk with center in the union of UL, CL and LL can intersect RH.*

Proof. We consider the case of LH and the union of UR, CR and LR only. The other case is symmetric. For brevity, let R denote the union of UR, CR and LR.

Assume, for a contradiction, that there is a disk d in OPT′ that has its center c in R and intersects LH. We will show that d covers RH completely, contradicting our assumption that OPT′ is a many-hole solution with at least one hole in RH.

Assume that d does not cover RH completely. Then there must be a point q in RH that is not contained in d. Furthermore, as d does not cover LH completely, there must be a point p in LH that lies on the boundary of d. Assume without loss of generality that q is not below p. As d contains p but not q and its center is to the right of q, c must lie below q. Moreover, c lies on a circle C of radius 1 with center p, because d has p on its boundary. See Fig 3.5. Consider point c' on C at distance 1 from q and below q (observe that c' exists because p and q are at most $\sqrt{2}/2$ apart). Let α denote the angle of lines $\overline{qp}$ and $\overline{qc'}$ and let α' denote the angle of line $\overline{qp}$ and the negative y-axis. Observe that both α and α' are at most 180°. Because c lies left of c' on circle

C, it is enough to show that $\alpha \leq \alpha'$, i.e., $\cos\alpha \geq \cos\alpha'$, i.e., c is not in R (a contradiction). Let u, h and v denote the distance, the horizontal distance and the vertical distance of p and q. Then $\cos\alpha = u/2$ and $\cos\alpha' = v/u$. We have $v \leq 1 - \sqrt{1-h^2}$, i.e., $1 - h^2 \leq (1-v)^2$, i.e., $2v \leq v^2 + h^2 = u^2$, i.e., $v/u \leq u/2$, which concludes the proof. □

Our approach to the weighted disk cover problem in the many-hole case can now be outlined as follows. We will show that LH contains a region $\mathcal{L}$ such that points in $\mathcal{L}$ can be covered only by disks with center in CL by OPT′. Let $\mathcal{P}' \subseteq \mathcal{P}$ be the points in LH that are not in $\mathcal{L}$ and are not already covered by the disks the algorithm guesses to define $\mathcal{L}$. We will show that points in $\mathcal{P}'$ can only be covered by disks with center in U or L. The same approach will be applied to RH. This breaks the problem into two independent subproblems: covering points in $\mathcal{L}$ and in the corresponding region of RH using disks with center in CL or CR, and covering the remaining points using disks with center in U or L. Each of the two subproblems can be solved optimally by dynamic programming (Lemma 3.2). Since the subproblems are independent, the union of their optimal solutions gives an optimal solution to the disk cover problem in the many-hole case.

In the following we discuss this solution approach for points from $\mathcal{P}$ that are in LH in more detail. The arguments for RH are symmetric. Lemma 3.4 shows that no disk with center in the union of UR, CR and LR can intersect LH. We distinguish the following three cases concerning disks with center in CL that are contained in OPT′:

1. OPT′ does not contain any disk with center in CL.
2. OPT′ contains one disk with center in CL.
3. OPT′ contains two or more disks with center in CL.

The algorithm guesses which of the three cases holds for OPT′. In the first case we have that all points in LH are covered by disks with center in U or L by OPT′. In the second case, we have to additionally guess the disk d with center in CL that is in OPT′. The remaining points in LH (those that are not covered by d) can then again only be covered by disks with center in U or L by OPT′.

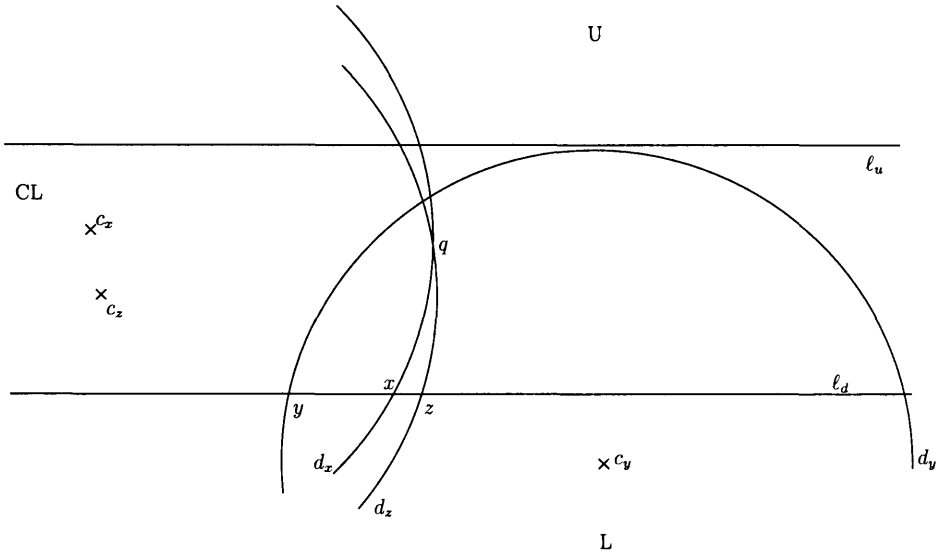


Figure 3.6: Setting for the many-hole case where more than one disk with center in CL appears in OPT' and one of the disks is not on the boundary.

It remains to deal with the third case, where OPT$'$ contains two or more disks with center in CL. We can show that in this case, all disks from OPT$'$ with center in CL form a consecutive piece of a boundary of a single hole. We start with lemma that shows for our case (the third case) that all disks from OPT$'$ with center in CL appear on the boundary of OPT$'$. In the following, we define ℓ_u and ℓ_d to be the horizontal lines that contain the upper and lower side of b, respectively.

Lemma 3.5 *If there is a disk $d_\ell \in$ OPT$'$ that has its center in CL and is not on the boundary of* OPT$'$*, then d_ℓ is the only disk with center in CL in* OPT$'$.

Proof. Assume for contradiction that there are at least two disks in OPT$'$ with center in CL and one of them does not appear on the boundary of OPT$'$. Let T denote the set of disks in OPT$'$ that have their center in CL. Consider the boundary B_T formed by disks from T inside b and observe that all disks from T appear on B_T. Let d_x and d_z be two adjacent disks on B_T, d_x above d_z, i.e., the center c_x of d_x above the center c_z of d_z, and let one of them be the disk that is not part of the boundary of OPT$'$. Let q be their common corner point on B_T, see Figure 3.6. We can assume that q is in LH, because otherwise one of the two disks d_x and d_z would be redundant in OPT$'$. Because d_x or d_z is not part

of the boundary of OPT′, point q must be covered by some disk with center in U or L (note that Lemma 3.4 shows that q cannot be covered by a disk with center in the union of UR, CR and LR). Assume that q is covered by a disk d_y with center c_y in L (the case when c_y is in U is analogous). Let x and z be the rightmost intersections of the disks d_x and d_z, respectively, with line ℓ_d. Points that are covered by d_z and not by d_x must be in the triangular region qxz. We show that the triangular region qxz is covered by d_y, and therefore d_z could be removed from OPT′, a contradiction to the fact that OPT′ is an optimum solution. Disk d_y does not cover the entire triangular region qxz if and only if y, the leftmost intersection of d_y and ℓ_d, is to the right of x. (Note that the rightmost intersection of d_y and ℓ_d is always to the right of x, if d_y has center in L.) If d_y has q on its boundary (i.e., the distance of c_y and q is 1), we claim that d_y must have its center c_y in the union of UR, CR, LR. If this is the case, observe that if the distance from c_y to q gets shorter than 1, c_y remains in the union of UR, CR and LR, which contradicts Lemma 3.4. We are left to prove the claim. Assume that $x = y$. Let h be the horizontal distance of c_y and q (see

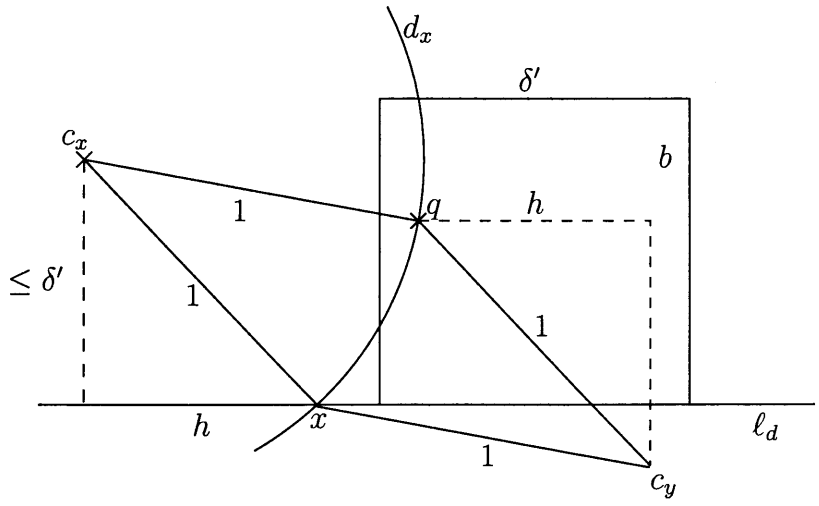


Figure 3.7: The shown setting is impossible: Center c_y must lie to the right of b because $h > \delta'$.

Fig. 3.7 for an illustration). Observe that h is equal to the horizontal distance of c_x and x (for this observe that line $\overline{c_x q}$ is parallel to line $\overline{x c_y}$). Therefore, $h \geq \sqrt{1-\delta'^2}$. For $\delta' \leq \sqrt{2}/2$ we have $h \geq \delta'$ and therefore c_y lies in the union of UR, CR and LR. If y moves to the right of x, then c_y moves to the right as

well, i.e., c_y stays in R, which is a contradiction to Lemma 3.4. □

Thus, according to this lemma (as we have at least two disks in OPT′ with center in CL), all disks from OPT′ with center in CL lie on the boundary of OPT′ (possibly on more distinct holes). Let p_u and p_ℓ be the corners with largest and smallest y-coordinate, respectively, among all corners of holes that are determined by at least one disk with center in CL. Notice that p_u and p_ℓ are different. Let d_u and d_ℓ be the disks with center in CL that determine p_u and p_ℓ, respectively, and let d_L be the disk with center to the left of p_u and p_ℓ that contains p_u and p_ℓ on the boundary. (Note that d_L is in general not a disk that is part of the input.) The algorithm guesses these points and the disks determining them. The disks d_u, d_ℓ and d_L define a region $\mathcal{L}$ in the same way as in the one-hole case, see Fig. 3.2 (top).

We show that the points from $\mathcal{L}$ can only be covered in OPT′ by disks with center in CL.

Lemma 3.6 OPT′ *does not contain any disk with center outside CL that intersects $\mathcal{L}$ in LH.*

Proof. Let d be a disk that has its center c outside CL and that intersects $\mathcal{L}$ in LH. Since c is not in CL, c has to be to the right of the line $\overline{p_\ell p_u}$ and d intersects d_L twice between the points p_u and p_ℓ. We claim that c must be in the union of UR, CR and LR. This follows by the same arguments as in the proof of Lemma 3.3 (which are applicable since $\delta' \leq \sqrt{2}\delta < \sqrt{2}/2$). However, Lemma 3.4 shows that no disk with center in the union of UR, CR and LR intersects LH, and therefore no such disk can intersect $\mathcal{L}$ in LH. □

It follows that all disks from OPT′ with center in CL lie on the boundary of the same hole:

Lemma 3.7 *p_u and p_ℓ lie on the boundary of the same hole in* OPT′*, and the boundary between p_u and p_ℓ is formed by disks with center in CL (or parts of sides of S) only.*

Proof. Observe that from the construction of the region $\mathcal{L}$ we have that all disks with center in CL that are on the boundary of the solution can appear only in the region $\mathcal{L}$. Since no other disk than those with center in CL can intersect $\mathcal{L}$ (Lemma 3.6), the results follows. □

Thus, we have characterized the disks with center in CL that appear on the boundary of OPT'. Because every disk from OPT' with center in CL must be on the same boundary that is within $\mathcal{L}$, the disks with center in CL cannot cover any other point outside $\mathcal{L}$. The points in LH that are not in $\mathcal{L}$ must be therefore covered by disks with center in U or L.

Similarly, the points in RH can be split into those that can only be covered by disks with center in CR and those that can be covered by disks with center in U or L. We create two subproblems that can be solved by dynamic programming (Lemma 3.2): one subproblem for the points to be covered by disks with center in U or L, and one subproblem for the points to be covered by disks with center in CL or CR. For both subproblems, there is a (vertical or horizontal) strip that contains all the points to be covered while all disks have their centers outside that strip. As OPT' cannot contain a disk that covers points from both subproblems, the union of the optimal solutions to the two subproblems actually gives an optimal disk cover for the square.

In summary, we have shown that in both the one-hole case and the many-hole case we can obtain a 2-approximation (in the many-hole case, even an optimal solution) of the minimum-weight disk cover for the given $\delta \times \delta$ square S. Furthermore, all other cases (no holes, or one hole with not all disks on the boundary of the hole) can be reduced to one of these cases by guessing one or two disks in the optimal solution. Therefore, we obtain a 2-approximation algorithm for the problem of computing a minimum-weight disk cover in a small square.

3.4.2 Algorithm for General Weighted Disk Cover with Unit Disks

We remark that our result on disk cover in a small square also implies a constant-factor approximation algorithm for the general weighted disk cover problem with unit disks (i.e., given a set of points and a set of weighted unit disks, find a minimum-weight set of given disks that covers all the points). We can simply partition the plane into $\delta \times \delta$ squares and compute an approximate disk cover for each square. Then we output the union of all computed disk covers as the solution. As a disk from the optimal solution can be used to cover points

in at most $O(1/\delta^2)$ different $\delta \times \delta$ squares, we lose only a factor of $O(1/\delta^2)$ in the approximation ratio by solving the problem for each square separately. More precisely, for every disk d, any square containing a point covered by d must be fully contained in a disk of radius $1 + \sqrt{2}\delta$ around the center of d, and hence there are at most $\lfloor \pi(1 + \sqrt{2}\delta)^2/\delta^2 \rfloor = 36$ such squares (for $\delta = 0.999/2$). Since our algorithm for disk cover in one square has approximation ratio 2, the overall approximation ratio of our algorithm for the weighted disk cover problem with unit disks is 72. Previously, constant-factor approximation algorithms were known only for the unweighted case of the disk cover problem [23, 25]. Moreover, the approximation ratio 72 of our algorithm improves also the (previously best) approximation ratio 108 for the unweighted case from [25]. The approximation ration 72 for the unweighted problem was also obtained independently in [81].

Observe that both the general weighted disk cover problem and the minimum-weight dominating set problem have the same approximation ratio. This connection is not surprising as the following observation shows.

Observation 3.8 *Let $\mathcal{D}$ be the set of unit disks with centers $\mathcal{P}$. Let $\mathcal{S}$ be the set of disks of radius 2 with centers identical to $\mathcal{P}$. Let C be any subset of centers $\mathcal{P}$. Then, the set of disks $\mathcal{D}' \subseteq \mathcal{D}$ with centers C dominates all disks $\mathcal{D}$ if and only if the set of disks $\mathcal{S}' \subseteq \mathcal{S}$ with centers C covers all points $\mathcal{P}$.*

3.5 Connecting the Dominating Set

In this section we consider the problem of adding disks to a given dominating set in order to produce a connected dominating set. We present an algorithm that solves this problem by adding disks of total weight at most $O(w^*)$, where w^* denotes the optimal weight of a connected dominating set for the given set of weighted unit disks. Note that the problem of connecting up a dominating set is a special case of the node-weighted Steiner tree problem; for general graphs, the best known approximation ratio for the latter problem is logarithmic in the size of the graph [68].

Let $\mathcal{D}$ be a set of weighted unit disks, and let $U \subseteq \mathcal{D}$ be a dominating set. Let G denote the unit disk graph corresponding to the disks in $\mathcal{D}$, and assume that G is connected (otherwise, G cannot have a connected dominating set). If

the subgraph of G induced by U (denoted by $G[U]$) is not connected, there has to be at least two connected components of G. We are looking for a minimum-weight set $U' \subseteq \mathcal{D}$ of disks such that the graph $G[U \cup U']$ is connected. Observe that for each component of $G[U]$ there exists a path of length at most 3 in $G[U \cup U']$ that connects the component with some other component. If this is not the case then there exists a connected component for which every path in $G[U \cup U']$ connecting the connected component with another one contains at least 5 vertices and therefore there would be a vertex in the middle of the path that is not dominated by U, a contradiction. We use this fact in constructing U' of small total weight.

We call the vertex set of a connected component of $G[U]$ a *cluster* of U. We create an auxiliary graph H. The vertices of H correspond to the clusters of U. For every path of length at most 3 in G that connects a vertex in one cluster c_1 of U to a vertex in another cluster c_2 of U and whose one or two internal vertices are not in U, we add an edge between c_1 and c_2 to H. The weight of the edge is the sum of the weights of the disks corresponding to the one or two internal vertices of the path. Note that H can have parallel edges. Next, we compute a minimum spanning tree T in H. (The proof of the theorem below shows that H is a connected graph.) Finally, we connect the dominating set U by adding all disks that correspond to internal vertices of the paths in G that correspond to the edges of T.

Theorem 3.9 *Let $\mathcal{D}$ be a set of weighted unit disks and U be a dominating set. Let w^* be the weight of a minimum-weight connected dominating set for $\mathcal{D}$. There is an efficient algorithm that computes a set U' of disks such that $U \cup U'$ is a connected dominating set and $w(U') \leq 17w^*$.*

Proof. We show that the auxiliary graph H contains a spanning tree T' of weight at most $17w^*$. This implies that H is connected. Furthermore, the weight of the set U' of disks that the algorithm adds to U is at most the weight of the minimum spanning tree, and the weight of the minimum spanning tree is upper bounded by the weight of T'. Therefore, we get $w(U') \leq 17w^*$.

It remains to show how to construct a spanning tree T' of H with weight at most $17w^*$. Let U^* be an optimal connected dominating set, $w(U^*) = w^*$. Let C be an arbitrary non-empty set of clusters of U, but not the set of all clusters

of U. Let $\bar{C}$ be the set of the remaining clusters of U. We claim that G must contain a path π from a vertex in some cluster in C to a vertex in some cluster in $\bar{C}$ such that π contains at most two internal vertices and has the property that all its internal vertices are in $U^* \setminus U$. (Note that such a path π corresponds to an edge in H.) To prove the claim, we argue as follows. Let x be an arbitrary vertex in a cluster in C, and y an arbitrary vertex in a cluster in $\bar{C}$. As U^* is a connected dominating set, there must be a path p in G from x to y all of whose internal vertices are in U^*. Let x' be the last vertex on p that is not in U and that is dominated by a vertex x'' in a cluster in C. Note that such a vertex x' must exist. Furthermore, x' or the vertex y' after x' on p must be dominated by a vertex y'' in a cluster in $\bar{C}$. Therefore, we obtain the desired path as x'', x', y'' or x'', x', y', y''.

Now we can create a spanning tree of H as follows. We start with a tree consisting of a single vertex of H (corresponding to some cluster of U) and grow the tree by repeatedly finding a path π in G that connects a vertex from a cluster in the tree to a vertex in a cluster not in the tree and has the properties discussed above. The claim above shows that such a path must exist. We can thus grow the tree by adding the edge in H that corresponds to the path π. This is repeated until we have a spanning tree T'.

The weight of each edge in the spanning tree T' corresponds to the weight of the internal vertices (which are in U^*) of a path of length at most 3 that connects different clusters of U. Furthermore, a vertex (disk) d of U^* can contribute to at most 17 edges of H: Whenever d contributes to the weight of an edge, it is an internal vertex of a path that connects two clusters of U whose closest disks have (graph-theoretic) distance at most 2 from it. However, the set of disks at distance at most 2 from d can contain at most 18 disjoint disks (see e.g. [102]) and therefore at most 18 disks from different clusters of U. As the spanning tree can contain at most 17 edges between these 18 clusters, we obtain that d contributes its weight to at most 17 edges of the spanning tree T'. Consequently, $w(T') \leq 17w^*$. □

Together with Theorem 3.1, we obtain the following corollary.

Corollary 3.10 *There is a constant-factor approximation algorithm for the minimum-weight connected dominating set problem in unit disk graphs.*

The approximation ratio of the algorithm of Corollary 3.10 is at most $72 + 17 = 89$.

3.6 Covering Points with Unit Squares

In this section we look at possible adaptations of the two main problems that were studied in this chapter—the weighted dominating set in unit disk graphs and the covering problem of points in the plane with weighted disks. We apply the techniques and ideas developed throughout the chapter to a setting where instead of disks we consider axis-parallel squares of size 1×1.

Let us consider the covering problem first. Let $\mathcal{P}$ be the set of points in the plane that is to be covered by a set of weighted (axis-parallel) unit squares $\mathcal{S}$, i.e., squares of size 1×1 (this scenario includes all covering problems with rectangles of size $a \times b$, where a and b are constants—just scale the setting appropriately). The weight of a square $s \in \mathcal{S}$ is denoted by w_s. The weight of the squares from set $\mathcal{S}' \subseteq \mathcal{S}$ is denoted by $w(\mathcal{S}') := \sum_{s \in \mathcal{S}'} w_s$. The goal of the *Weighted Square Covering Problem* (WSCP for short) is to find a minimum-weight set $\mathcal{S}' \subseteq \mathcal{S}$ that covers all points $\mathcal{P}$, i.e., $\forall p \in \mathcal{P}$: $p \in \bigcup\{s \mid s \in \mathcal{S}'\}$. We define the center c_s of a square $s \in \mathcal{S}$ to be the intersection point of the two diagonals of s. Thus, the distance from c_s to the horizontal "end" of s is $1/2$ and the distance from c_s to the vertical "end" of s is $1/2$.

Similarly to the approach of the covering problem with disks, we divide the plane into parts where we can compute a good approximation to some local covering subproblem. Our approach divides the plane into strips of width 1. Strip T_i is bounded by two horizontal lines $y = i$ and $y = i + 1$ and consists of points $\{(x, y) \mid i \leq y < i + 1\}$. For every strip T_i we solve the following covering problem: cover the points from T_i by a minimum-weight set $U_i \subseteq \mathcal{S}$. At the end we output the union of all sets U_i as the approximate solution to WSCP.

3.6.1 Covering Points in a Strip

We present an optimal algorithm for the covering subproblem for strip T_i. The algorithm is basically identical to the dynamic programming from Lemma 3.2. Let $P_i \subseteq \mathcal{P}$ be the points inside T_i. The horizontal line ℓ_i that goes through the

middle of the strip T_i (i.e., ℓ_i is a line $y = i + 1/2$) divides the squares S into two sets S_U and S_L—square s with center above ℓ_i belongs to S_U and square s with center below (or on line) ℓ_i belongs to S_L. We can now use the dynamic programming from Lemma 3.2 where instead of upper disks we use the squares S_U and instead of lower disks we use the squares S_L. The squares satisfy all the properties that are necessary for the correctness of the dynamic programming (see Lemma 3.2). Hence, the problem of covering points by weighted unit squares in a strip of width 1 can be solved optimally in polynomial time.

3.6.2 Approximation Algorithm for WSCP

We discuss the quality of the solution $U = \bigcup_i U_i$ that is produced by the algorithm. Let OPT denote the optimum solution to the covering problem. Let OPT[i] denote the set of squares from OPT that cover a point in T_i. Clearly, $w(U_i) \leq w(\text{OPT}[i])$, as U_i is an optimum solution for the covering problem in T_i. Hence, $w(U) \leq \sum_i w(U_i) \leq \sum_i w(\text{OPT}[i])$. The summation is over i such that T_i is not empty. The weight of a square from OPT[i] can be counted more than once in the above sum: always, when a square covers a point in more than one strip T_i. Each square s can cover a point in at most 2 strips (as it can intersect at most 2 strips). Hence, $w(U) \leq \sum_i w(U_i) \leq \sum_i \text{OPT}[i] \leq 2w(\text{OPT})$.

Theorem 3.11 *There is a 2-approximation algorithm for the Weighted Square Covering Problem.*

3.6.3 MWDS in Unit Square Graphs

We can use the above algorithm to compute a constant factor approximation of a minimum-weight dominating set in *unit square graphs*—geometric intersection graphs of squares $\mathcal{S}$ of size 1×1. Observe that a square $s \in \mathcal{S}$ dominates a square $s' \in \mathcal{S}$ (s' can be equal to s) if and only if a square of size 2×2 with center at c_s covers the center $c_{s'}$ of square s'. Hence, a solution to WSCP translates directly into a solution to MWDS in unit square graphs and a solution to MWDS in unit square graphs translates directly into a solution to WSCP.

Observation 3.12 *Let $\mathcal{S}$ be a set of unit squares and let $\mathcal{P}$ denote the set of centers of squares of $\mathcal{S}$. Let $\mathcal{D}$ denote the set of squares of size 2×2 with centers*

identical to centers $\mathcal{P}$. Let C be a subset of the centers $\mathcal{P}$. Then squares $\mathcal{D}' \subseteq \mathcal{D}$ with centers C cover all points $\mathcal{P}$ if and only if squares $\mathcal{S}' \subseteq \mathcal{S}$ with centers C dominate all squares $\mathcal{S}$.

As a direct application of this observation and Theorem 3.11, we obtain the following result.

Theorem 3.13 *There is a 2-approximation algorithm for the Minimum-Weight Dominating Set problem in unit square graphs.*

3.7 A 3-approximation Algorithm for Minimum-Weight Forwarding Sets

In this section we consider the *minimum-weight forwarding set problem* (MWFS). In this problem, we are given a distinguished unit disk d_o (the source disk), a set of weighted unit disks $\mathcal{D}$ with centers in d_o, and a set of points $\mathcal{P}$ in the plane outside d_o (but contained in the union of the disks in $\mathcal{D}$). The goal is to find a minimum-weight subset $\mathcal{D}'$ of $\mathcal{D}$ such that every point in $\mathcal{P}$ is covered by at least one disk from $\mathcal{D}'$. In the unweighted version of the problem (MFS), all disks have weight 1. Obviously, the problem is a special case of the disk cover problem.

MFS and MWFS arise in wireless ad-hoc networks in the context of the efficient implementation of flooding [25]. Flooding is a broadcasting mechanism where each node forwards the message to all its neighbours. This leads to many redundant messages. A more efficient implementation is obtained by letting each node forward the message only to a subset of the one-hop neighbours (the forwarding set) that covers all the two-hop neighbours. If the wireless network is modeled as a unit disk graph, the problem of determining a smallest (or minimum-weight) forwarding set for a node is just MFS (or MWFS) as defined above.

Calinescu et al. [25] devised a 3-approximation algorithm for MFS. We combine our ideas from Section 3.4 with their approach and obtain a 3-approximation algorithm for MWFS, the weighted version of the problem. The 3-approximation algorithm ALG for the unweighted case from [25] partitions the points in $\mathcal{P}$ ac-

cording to the four quadrants defined by two orthogonal lines through the center o of disk d_o, and then independently solves the covering problem for each quadrant. The union of these four disk covers is then a disk cover for all the points in $\mathcal{P}$. Clearly, the same approach can be applied to the weighted case as well.

Lemma 3.14 ([25]) *If an α-approximation algorithm is used for the (weighted) covering problem in each quadrant, the approximation ratio of ALG is at most 3α.*

The proof of this lemma is based on the observation that each disk in $\mathcal{D}$ can cover points from $\mathcal{P}$ in at most three quadrants.

We present an optimal algorithm for the weighted covering problem of points in one quadrant, thus obtaining a 3-approximation algorithm for MWFS.

Lemma 3.15 *There is a polynomial-time optimal algorithm for the minimum-weight forwarding set problem if the points $\mathcal{P}$ lie in one quadrant only.*

Proof. For a disk $d \in \mathcal{D}$, let $w(d)$ denote the weight of the disk. Let Q be the quadrant in which the points $\mathcal{P}$ lie. Let $Q' = Q - d_o$ be the *external* quadrant, i.e., the quadrant without the disk d_o. Observe that in any optimal solution $\mathcal{D}'$, each disk $d \in \mathcal{D}'$ appears in Q' on the boundary of the solution (where the boundary of the solution means the boundary of the union of the disks from $\mathcal{D}'$). To see this, consider a point $p \in \mathcal{P}$. Let $\overline{op}$ be the half-line starting at o and passing through p. Because p is covered by $\mathcal{D}'$, the half-line must intersect the boundary of $\mathcal{D}'$ at some point o' that lies beyond the point p. Let $d' \in \mathcal{D}'$ be the disk that contains o'. We say that disk d' is *active* at p. Observe that o' is in Q'. Because every disk from $\mathcal{D}$ contains o, the disk d' contains the whole segment of $\overline{op}$ between o and o'. Thus the disk d' contains p as well. Therefore, if there is a disk $d \in \mathcal{D}'$ that is not on the boundary of the solution, we could remove the disk and keep the points covered, a contradiction to $\mathcal{D}'$ being an optimal solution.

In [25] it was proved that in Q' the boundaries of any two disks from $\mathcal{D}$ can intersect in at most one point. This implies that each disk from the optimal solution appears on the boundary of $\mathcal{D}'$ in Q' exactly once (i.e., there is exactly one continuous part of the disk on the boundary).

We present a dynamic programming approach for the covering problem of points in Q (which is an adaptation of the dynamic programming from

Lemma 3.2). Assume that the points in $\mathcal{P}$ are ordered according to their polar coordinates (with the reference center being o). The algorithm computes a table with entries $T_i(d_j)$ that store the cost of an optimal solution for covering the points $p_1, \ldots, p_i$ in such a way that d_j is active at p_i. If there is no solution for which d_j is active at p_i, we set the entry to ∞. Computing $T_1(d_j)$ is straightforward: We set $T_1(d_j) = w(d_j)$ if d_j covers p_1, otherwise we set $T_1(d_j) = \infty$. For a disk d_j that covers p_i, we can compute the cost $T_i(d_j)$ using

$$T_i(d_j) = \min_k \{T_{i-1}(d_k) + [j \neq k] \cdot w(d_j) \mid d_j \text{ active at } p_i \text{ for the solution of } T_{i-1}(d_k)\}.$$

Here, the term $[j \neq k]$ is 1 if $j \neq k$, and 0 otherwise. The correctness of the computation is justified by the fact that each disk can be active only for points in $\mathcal{P}$ that are consecutive. The weight of the optimal solution can be obtained as the minimum of the table entries $T_n(d)$, where n is the number of points in $\mathcal{P}$ and d ranges over all disks in $\mathcal{D}$ that cover p_n. The optimal solution itself can be obtained using standard bookkeeping techniques. □

The discussion of this section leads to the following theorem.

Theorem 3.16 *There is a 3-approximation algorithm for the minimum-weight forwarding set problem.*

3.8 Summary of Results and Open Problems

In this chapter we have studied the Minimum-Weight Dominating Set in unit disk graphs. The problem, motivated by the real-life scenarios in mobile ad-hoc networks, is $\mathcal{NP}$-complete and we have presented the first constant factor approximation algorithm for the problem. The techniques that were developed can be successfully applied to other geometric problems of similar nature, such as the unit disk covering problem, covering problem and dominating set of squares, and the forwarding set problem. For all these problems our approach leads to new or better approximation algorithms than the previously known ones.

It is an interesting question (and a long standing open problem even for unweighted graphs) whether there is a constant factor approximation algorithm for a dominating set problem for general disks (i.e., disks are not restricted to

a unit radius) or rectangles.

Chapter 4

Network Discovery and Verification

In recent years, there has been an increasing interest in the study of networks whose structure has not been imposed by a central authority but has arisen from local and distributed processes. Prime examples of such networks are the Internet, sensor networks or unstructured peer-to-peer networks such as Gnutella. As opposed to the static, centrally planned networks, there is no central authority with a full knowledge and control over the network and which possesses a map of such a network. Obtaining an accurate map, usually modeled as a graph, is generally very difficult and costly due to the network's dynamic growth process and limitations in accessing the network. Such network maps are useful for many purposes, especially for studying and analyzing the network structure, routing aspects, robustness properties, etc.

In order to create maps of the Internet, a commonly used technique is to obtain local views of the network from various locations (vantage points) and combine them into a map that is hopefully a good approximation of the real network. There is an extensive body of related work studying various aspects of this approach, see e.g. [32, 47, 84, 64, 65, 61, 16, 95, 45, 2, 39, 40]. More generally, one can view this technique of discovering the topology of an unknown network as performing queries on the network. A query corresponds to asking for a local view of the network from one specific vantage point (a node of the

network).

As performing such a query at a node is usually very costly (in terms of time, energy consumption or money), the question of minimizing the number of such queries arises naturally. We formalize the problem of obtaining a map of an unknown network as a combinatorial optimization problem (*network discovery* problem) and study it from this perspective. The goal of the network discovery problem is to minimize the number of queries required to discover all edges and non-edges (absent edges) of the network (which is modeled as a graph). We study the problem both as an online and offline problem.

In the online network discovery problem (*network discovery* for short—we omit the word "online") we assume that the nodes of the network are known in the beginning and it is only the edges and non-edges that have to be discovered. The algorithm ascertains the edges and non-edges in the online manner and at each step has to decide where to make the next query. The only information the algorithm has is gained with the previous queries. Thus, the difficulty in selecting good queries arises from the fact that the amount of information discovered by a query may strongly depend on the parts of the network that are still unknown.

In the offline network discovery problem (called *network verification*) the full information about the network is given to the algorithm, i.e., all the nodes and edges are known (hence also all non-edges). The task is to compute a minimum set of queries that suffice to discover the network (if the network was unknown). Although an algorithm for this offline problem is hardly useful for the network discovery (knowing the network there is no need to discover it), it can be employed for a scenario where a given map of the network is to be verified whether it is still accurate.

Note that at first sight, in order to discover a network, it might seem sufficient to discover only the edges of the graph. It is, however, necessary to have a proof (i.e., discovery) for unconnected node-pairs that there is actually no edge between them, especially in view of the online setting. An online algorithm can only know that it discovered the network when both the edges and non-edges have been discovered. Taking both the edges and non-edges into account reflects in a better way what portion of the unknown network has been discovered by a

set of queries. This can be helpful when investigating the quality of published maps of the Internet.

We consider two different query models. In the *layered-graph query model* the answer to a query at a vertex v consists of all edges and non-edges whose endpoints have different distance from v. This is equivalent to obtaining all-shortest paths from vertex v. In the *distance query model* a query at a node v returns all distances from v to all other vertices of the investigated graph (network). The distance query model is much weaker than the layered-graph query model in the sense that typically a query reveals much less information about the network. The motivation for the layered-graph query model comes from the following scenario. With traceroute tools, one can determine the path that packets take in the network if they are sent from one's node to some destination. If each traceroute experiment returns a random shortest path to the destination, one could use repeated traceroute experiments to all destinations to discover all edges of the shortest-path subgraph. Making a query at node v would mean getting access to node v and running repeated traceroute experiments from v to all other nodes. If we assume that the cost of getting access to a node is much higher than that of running the traceroute-experiments, minimizing the number of queries is a meaningful goal. The motivation for studying the distance query model comes from different scenarios. In many networks it is realistically possible to obtain the distances between a node and all other nodes, while it is difficult or impossible to obtain information about edges or non-edges that are far away from the query node. For example, so called distance-vector routing protocols work in such a way that each node informs its neighbours about upper bounds on the distances to all other nodes until these values converge; in the end, the routing table at a node contains the distances to all other nodes, and a query in our model would correspond to reading out the routing table. Another scenario is the discovery of the topology of peer-to-peer networks such as Gnutella [36]. There, with the Ping/Pong protocol it is possible to use a Ping command to ask all nodes within distance k (the TTL parameter of the Ping) to respond to the sender [5]. Repeated Pings could be used to determine the distances to all other nodes.

Without doubt, both of the query models are simplifications of reality. In

real networks, routing does not necessarily use shortest paths, and traceroute experiments will often reveal only a single path (or at most a few different paths) to each destination, but not the whole shortest-path subgraph. Real peer-to-peer networks are often so large that it becomes prohibitive to send Pings for larger TTL values, and there are also many other aspects that make the actual discovery of the topology of a Gnutella network very difficult [5]. Nevertheless, we believe that our models constitute a good starting point for a theoretical investigation of fundamental issues in network discovery.

4.1 Problem Definitions and Preliminaries

A network is represented by a connected, undirected, unweighted graph $G = (V, E)$. The number of nodes is denoted by $n = |V|$ and the number of edges by $m = |E|$. For two distinct nodes $u, v \in V$ we say that $\{u, v\}$ is an *edge* if $\{u, v\} \in E$ and a *non-edge* if $\{u, v\} \notin E$. The set of non-edges of G is denoted by $\bar{E}$. The distance of two nodes u and v in graph G is denoted by $d_G(u, v)$ or $d(u, v)$ if it is clear in which graph the distance of u and v is measured. A *query* is specified by a node $v \in V$ and is called a query at v or simply the query v. We are interested in a minimum-size *query set* $Q \subseteq V$ that "discovers" the network G. As a first objective, "to discover" refers to identifying all edges and non-edges of the network G. In general, "discovering" a network can also mean "discovering" other properties of the graph, such as the diameter or maximum degree. The query model and the given discovery goal can lead to a plentiful amount of online and offline optimization problems. We now present a general framework for such network discovery problems.

To obtain a concrete variant of the network discovery or verification problem, the following two aspects need to be specified:

Query model: The query model specifies the type of queries that can be asked, and the information that is returned by a query in a given network.

Completion criterion: This criterion specifies when the task of discovery or verification is considered to be completed, i.e., when a set of queries is *sufficient.*

In all the network discovery problems, we are interested in algorithms that compute a set Q of queries (as specified in the query model) such that the information returned by the queries in Q is sufficient to satisfy the completion criterion. The objective is to minimize the cardinality of the set Q. In the online case, the algorithm can use only the information obtained through previous queries for selecting the next query. This scenario is given by the vertex set only and the edges of the graph are not known to the algorithm. In the offline version both the vertex set and edge set is known to the algorithm. The offline case is also called *network verification* problem.

Query Models. The query model specifies the information that is returned to the algorithm when a query at a node $v \in V$ is performed. We are studying the following two query models in this thesis.

For the *layered-graph query model*, the query at node $q \in V$ returns all edges and non-edges $\{u, v\}$ of G where the distances $d_G(q, u)$ and $d_G(q, v)$ are different. We refer to this query model as LG.

For the *distance query model*, the query at node $q \in V$ returns distances from q to all other nodes of V. We refer to this query model as DIST.

Completion Criteria. The completion criterion is the condition that must be satisfied so that the graph discovery or verification task is considered accomplished. In the following we describe two completion criteria.

Discovery of the edge set E and the non-edge set $\bar{E}$ of the network is the completion criterion that we consider in this thesis. The task of the algorithm is accomplished, if the algorithm computes a query set that gives enough information to identify uniquely the edge set E and the non-edge set $\bar{E}$ of the graph. This completion criterion is referred to as ALL. For this criterion, we say that the graph is discovered as synonym to saying the edges and non-edges are discovered.

Another completion criterion (which we do not consider in this, but was considered previously) is to discover the edge set E only. Clearly, this completion criterion makes sense only in the offline setting. We refer to this completion criterion as ALL-E.

Network Discovery and Verification Problems. We refer to the network discovery problem with query model X and completion criterion Y as X–Y–DISCOVERY and to its offline version as X–Y–VERIFICATION. Thus, LG–ALL–DISCOVERY and DIST–ALL-E–VERIFICATION are two examples of concrete problems arising in our framework.

Thus, the LG–ALL–DISCOVERY asks for a minimum number of queries that discover both the edges and non-edges of the network. The query q returns all edges between vertices of different distance to q. This is equivalent to obtaining all shortest paths from q to any other vertex. (To see this, observe that a query at q returns an edge $\{u, v\}$ if and only if $\{u, v\}$ lies on a shortest path from q to some node v'). For a query vertex q we can sort the vertices V into layers according to their distance from q. Layer L_i (or L_i^q, if we want to emphasize the query to which the layer is associated) is the set $\{w \in V \mid d(q, w) = i\}$. Thus, the result of the query can be viewed as a layered graph. Hence, if $u, v \in V$ and $d_G(q, u) \neq d_G(q, v)$, query q reveals what $\{u, v\}$ is—either an edge or non-edge. In such a case we say that q *discovers* the edge/non-edge $\{u, v\}$. Hence, in this model, the completion criterion is satisfied if and only if every pair $\{u, v\}$ (edge or non-edge), $u, v \in V$, is discovered by a query $q \in Q$.

The LG–ALL–VERIFICATION problem is the offline version of the LG–ALL–DISCOVERY problem. The whole network $G = (V, E)$ is given to the algorithm and the goal is to compute a minimum size query set $Q \subseteq V$ such that every edge/non-edge $\{u, v\}$, $u, v \in V$, is discovered. In this case we may also say that q verifies $\{u, v\}$ and that Q *verifies* the graph.

The DIST–ALL–DISCOVERY and DIST–ALL–VERIFICATION problems ask for a minimum-size set Q that discovers all edges and non-edges of the network. It is not clear at first sight what exactly it means to discover the edges and non-edges of the network. Intuitively, the set of queries Q discovers the network G, if the distances to all vertices that Q returns differ (in at least one distance) from the distances that Q returns when applied to any network $G' = (V, E') \neq G$ (G' is on the same vertex set as G). Formally, we label the queries in Q as $q_1, q_2, \ldots, q_{|Q|}$ and we denote by $d_G(Q, v)$ the $|Q|$-dimensional *distance vector* of distances from Q to v, i.e., the i-th component contains the distance $d_G(q_i, v)$. Let $\mathbf{D}_G(Q)$, for $Q \subseteq V$, denote the set of all distance vectors

$d_G(Q, v)$, for all $v \in V$. We write $\mathbf{D}_G(Q) \neq \mathbf{D}_{G'}(Q)$, for $G' = (V, E')$, if there exists at least one query $q \in Q$ and a node $v \in V$ such that $d_G(q, v) \neq d_{G'}(q, v)$. Conversely, $\mathbf{D}_G(Q) = \mathbf{D}_{G'}(Q)$, if $d_G(q, v) = d_{G'}(q, v)$ holds for all queries $q \in Q$ and all nodes $v \in V$.

Discovering a Graph in the Distance Query Model. A query set $Q \subseteq V$ for the graph $G = (V, E)$ *discovers the edge* $e \in E$ (*discovers the non-edge* $\bar{e} \in \bar{E}$), if for all graphs $G' = (V, E')$ with $\mathbf{D}_G(Q) = \mathbf{D}_{G'}(Q)$ it must hold that $e \in E'$ ($\bar{e} \in \bar{E}'$). $Q \subseteq V$ *discovers the graph* G, if it discovers all edges and non-edges of G.

Alternative (Equivalent) Definition. The fact that Q discovers G implies that any graph G' with $\mathbf{D}_G(Q) = \mathbf{D}_{G'}(Q)$ must have the same edges and non-edges as G, in other words: $G' = G$. Conversely, if a query set Q for G yields $\mathbf{D}_G(Q) = \mathbf{D}_{G'}(Q)$ only for $G' = G$ and for no other graph, then Q discovers G (since it clearly can discover each edge and non-edge individually). Thus, this gives an equivalent definition: A query set $Q \subseteq V$ *discovers the graph* $G = (V, E)$, if for every graph $G' = (V, E') \neq G$ at least one of the resulting distances changes, i.e., $\mathbf{D}_G(Q) \neq \mathbf{D}_{G'}(Q)$. Intuitively, the queries Q which discover a graph G can distinguish it from any other graph G' (sufficient and necessary condition).

The DIST-ALL-DISCOVERY problem is given by the vertices of a network $G = (V, E)$ and the goal of the algorithm is to discover the network by a minimum-size query set Q in the distance query model. According to the definition, an algorithm has to compute a minimum size query set Q such that for every $G' = (V, E') \neq G$, $\mathbf{D}_{G'}(Q) \neq \mathbf{D}_G(Q)$. Here, intuitively, we expect the algorithm to be aware of how the actual edge set E of G looks like at the end of the algorithm's computation. In Section 4.4.1 we show how the algorithm can discover individual non-edges and edges of an unknown network from the queries Q. Actually, we use the approach of discovering individual non-edges and edges as the "thinking environment" for the DIST-ALL-DISCOVERY problem.

We evaluate the online algorithms for the online problems within the framework of competitive analysis. The number of queries Q (that discover the unknown network G) computed by an algorithm (and denoted by $A(G)$) is com-

pared to the minimum number of queries that discover G (denoted by $\mathrm{OPT}(G)$).

For the offline setting we are interested in approximation algorithms. Here, similarly to online algorithms, the number of queries Q computed by an algorithm (denoted by $A(G)$) is compared to the minimum number of queries that discover G (also called the optimum number of queries and denoted by $\mathrm{OPT}(G)$).

4.2 Related Work and New Contributions

There are several ongoing large scale efforts to collect data representing local views of the Internet. For example, the RouteViews project [84] by the University of Oregon collects data from a large number of so-called border gateway protocol routers. Essentially for each router—which can be seen as a node in the Internet graph—the list of paths it knows (to all other nodes in the network) is retrieved. More recently, and—due to good publicity—very successfully, the DIMES project [47] has started collecting data with the help of a volunteer community, similar in spirit to SETI@Home [93]. Users can download a client which collects paths in the Internet by executing successive traceroute commands. A central server can direct each client individually by specifying which routes to investigate.

Data obtained by these or similar projects has been used in heuristics to obtain maps of the Internet, basically by simply overlaying possible paths found by the respective project, see e.g. [65, 84, 47, 32]. Another line of research aims at inferring from such local views the types of economic relationships between nodes in the Internet graph, cf. [61, 95, 45].

Bejerano and Rastogi [20] study the problem of monitoring link delays and faults in IP networks. They propose a two-phase approach. The first phase corresponds to the problem of verifying all edges of a network with as few queries as possible in a model which is related to our layered-graph query model, but where query results are trees. They give a SETCOVER [101] based $O(\log n)$-approximation algorithm and show that the problem is $\mathcal{NP}$-hard. Here and in the following, n stands for the number of nodes of the graph. They state that their reduction can be strengthened to give a lower bound of $\Omega(\log n)$ on the

approximation ratio. In contrast to Bejerano and Rastogi, we are interested in verifying (or discovering) both the edges and the non-edges of a graph. Breitbart et al. [22] consider the network monitoring problem in a setting where a query returns a shortest-path tree rooted at the query. They consider 2 variants of the problem—minimize the number of queries that verify all network links such that (a) the query may return any shortest path tree, or (b) the algorithm may choose which shortest-path tree is returned. The authors show that both variants are $\mathcal{NP}$-hard and they also present some heuristics for the problem together with an analysis and experimental evaluation of the heuristics.

Network verification in the layered graph query model is closely related to the problem of finding the metric dimension of a graph. A subset of nodes $Q \subseteq V$ such that every node in the graph has a unique vector of distances to Q is called resolvability set. The cardinality of a minimum resolvability set is the metric dimension of the given graph [62]. Such a minimum subset Q is also called a *basis* of the graph and it is easy to see that this is the same as a minimum query set in the layered graph query model. The problem of determining whether a graph G has a resolving set of cardinality at most k is $\mathcal{NP}$-complete [62]. Khuller et al. [76] investigate the problem of finding a basis of a graph (and thus a minimum query set). They present an $O(\log n)$-approximation algorithm (based on a SETCOVER greedy $O(\log n)$-approximation algorithm) and investigate special graph classes. For trees, they show that the problem can be solved optimally in polynomial time. Furthermore, they prove that one landmark is sufficient if and only if G is a path, and discuss properties of graphs for which 2 landmarks suffice. They also show that if k landmarks suffice for a graph with n vertices and diameter D, we must have $n \leq D^k + k$. For d-dimensional hypercubes, it was shown in [92] (using an earlier result from [78] on a coin weighing problem) that the metric dimension is asymptotically equal to $2d/\log_2 d$). A survey on resolvability of graphs is given in [27] (from a graph-theoretic point of view rather than an algorithmic one). Cáceres et al. [24] study the metric dimension in Cartesian products of graphs and give many helpful references, pointing out interesting connections to other closely related problems. Results for the problem variant where extra constraints are imposed on the basis (e.g., connectedness or independence) are surveyed in [90].

The LG–ALL-E–VERIFICATION problem was first considered in [87] (where the problem was called Minimum Layered-Tree Query Problem), where the authors consider special graph classes, show a lower bound on the number of queries in terms of the clique number $\omega(G)$ of the given graph G and also relate the diameter of graph G and the independence number of graph G with the maximum and minimum number of layers for a query at vertex $v \in G$. They also show that the decision variant of LG–ALL-E–VERIFICATION is $\mathcal{NP}$-complete and present a SETCOVER based $O(\log n)$-approximation algorithm for LG–ALL–VERIFICATION and LG–ALL-E–VERIFICATION. They also mention the LG–ALL–DISCOVERY problem as their prime interest and motivation, but do not consider it further.

The LG–ALL–DISCOVERY problem was then discussed in [49], where a lower bound on the number of queries in terms of the diameter of an induced subgraph was given (we recall this lower bound in our text later on). Three algorithms are proposed and experimentally compared with each other. No worst-case analysis is given.

In [17] both the LG–ALL–DISCOVERY and LG–ALL–VERIFICATION as well as LG–ALL-E–VERIFICATION are considered. It is shown that the LG–ALL–VERIFICATION problem cannot be approximated within a factor of $o(\log n)$ unless $\mathcal{P}$=$\mathcal{NP}$, thus showing that the approximation algorithm from [76] is best possible. This result was also published in [19, 18]. For LG–ALL-E–VERIFICATION a polynomial time algorithm is presented that computes the optimum number of queries for a special class of graphs—non-branching planar chordal rings. This is a class of graphs that can be build from cycles in the plane for which edges (chords; drawn as straight lines in the plane) are added that do not intersect in the plane. Further experiments for LG–ALL–DISCOVERY based on [49] were given. Also, another query model was considered—shortest path tree query model—where a query at node v returns a shortest path tree of the graph rooted at v. To specify which shortest path tree is returned by a query, weights from an interval $[1-\varepsilon, 1+\varepsilon]$ are assigned to every pair $\{u, v\}$ (edge or non-edge). The problem of deciding whether k queries discover a given graph G in this query model is shown to be $\mathcal{NP}$-complete.

In this chapter we study LG–ALL–DISCOVERY, LG–ALL–VERIFICATION,

DIST–ALL–DISCOVERY and DIST–ALL–VERIFICATION. For the layered-graph query model we first give in Section 4.3.1 a full characterization of graphs for which 2 queries discover the graph and we also investigate the change in the optimum number of queries if an edge is added or deleted from the network. For the online discovery problem we present a lower bound 3 on the competitive ratio of any online algorithm and give a deterministic algorithm that guarantees to output a query set of size at most $OPT(G)+4(|E(G)|-n)+5$. We also present a randomized online algorithm with competitive ratio $O(\sqrt{n \log n})$ (Section 4.3.2).

For the distance query model we first show how a query (or set of queries) can discover individual non-edges and edges (Section 4.4.1). In Section 4.4.2 we show lower bounds on the number of queries needed to discover or verify a graph, based on the independence number $\alpha(G)$, clique number $\omega(G)$ and size of the edge-set of the graph, $|E(G)|$. For DIST–ALL–VERIFICATION we present polynomial time algorithms for basic graph classes—chains, cliques, trees, cycles, hypercubes and grids. For general graphs, the problem turns out to be $\mathcal{NP}$-hard and a (SETCOVER like) $O(\log n)$-approximation algorithm is presented (Section 4.4.4). For DIST–ALL–DISCOVERY we show in Section 4.4.5 that no deterministic online algorithm can be better than $O(\sqrt{n})$-competitive and no randomized online algorithm can be better than $O(\log n)$-competitive. Finally, we present a randomized online algorithm with competitive ratio $O(\sqrt{n \log n})$.

Most of the results presented in this chapter were published in [19], [51] and [18].

4.3 Layered-Graph Query Model

4.3.1 A Few Structural Properties

Let us start with some observations and characterizations about the minimum number of queries for various graphs.

Theorem 4.1 ([49]) *If a graph $G = (V, E)$ contains a subgraph H of diameter D_H with n_H vertices, then* $\mathrm{OPT}(G) \geq \log_{D_H+1} n_H$.

Proof. Imagine the queries being performed sequentially. At any instant, the unknown edges and non-edges induce disjoint cliques, which we call *unknown*

groups. Two vertices are in the same unknown group if and only if they were in the same layer of all queries made so far. Consider the n_H vertices of subgraph H. Initially, all vertices form an unknown group. For each query, the n_H vertices of H will be in at most $D_H + 1$ consecutive layers of the layered graph returned by the query. Therefore, after the first query, at least $n_H/(D_H + 1)$ vertices of H will still be in the same unknown group. Similarly, after k queries, at least $n_H/(D_H + 1)^k$ vertices of H will be in an unknown group together. If k queries suffice to verify all edges and non-edges, the unknown groups must be singletons in the end. So we must have $n_H/(D_H + 1)^k \leq 1$. This proves the theorem. □

This theorem implies that a graph containing a clique on k vertices requires at least $\log_2 k$ queries, and a graph with maximum degree Δ at least $\log_3(\Delta+1)$ queries. For the former, take H to be the clique on k vertices, and for the latter, take H to be the subgraph induced by a vertex of degree Δ and its neighbours. The latter was also mentioned in [27]. This theorem can also be seen as a generalization of the result from [76], where it was stated that if k vertices discover the graph, then $n \leq D_G^k + k$.

We study now how much the optimal number of queries changes if the network is changed a bit—an edge is added or deleted, or a vertex is added or deleted. In [27] it was shown that if a vertex is added to the graph and it can be adjacent to any vertex, the number of queries can change dramatically. For example, a cycle needs 2 queries to be discovered, whereas the wheel (a cycle with one extra single vertex connected to all the vertices of the cycle) needs $\lfloor \frac{2n+2}{5} \rfloor$ queries. In [27] they also point out that for trees adding an edge can increase the number of queries by at most 1 or decrease the number of queries by at most 2. Here, we show a slightly more general result.

Lemma 4.2 *If $Q \subseteq V(G)$ discovers a graph G in* LG–ALL–DISCOVERY, *then $Q \cup \{u, v\}$ discovers $G' = G \setminus \{u, v\}$, for any edge $\{u, v\} \in E$.*

Proof. Let x, y be arbitrary vertices of the graph G. We show that there exists a query $q \in Q'$ that discovers $\{x, y\}$, i.e., leaves x and y in two different layers.

Let $q \in Q$ be a query that discovers $\{x, y\}$ in graph G. Assume w.l.o.g. that $d_G(q, x) < d_G(q, y)$. Consider the following two cases. First, assume that there is a shortest path from q to x in G of length $d_G(q, x)$ that does not contain the edge

$\{u, v\}$. In such a case $d_{G'}(q, x) = d_G(q, x)$ and therefore $\{u, v\}$ is discovered by q also in G', as $d_G(q, x) < d_G(q, y) \leq d_{G'}(q, y)$. In the second case assume that every shortest path from q to x (in G) contains the edge $\{u, v\}$. Assume w.l.o.g. that v appears after u on these shortest paths. Consider now the distances $d_G(v, x)$ and $d_G(v, y)$. As q discovers $\{x, y\}$ in G, we have $d_G(v, x) < d_G(v, y)$ (otherwise $d_G(q, x) = d_G(q, y)$). Observe that $d_{G'}(v, x) = d_G(v, x)$. Clearly, $d_G(v, y) \leq d_{G'}(v, y)$, and therefore $d_{G'}(v, x) < d_{G'}(v, y)$, i.e., v discovers $\{x, y\}$ in G'. □

We cannot make a similar claim about adding an edge into graph G, as shows an example from Figure 4.1, where $Q = \{z, v\}$ discovers G, but $\{z, u, v\}$ does not discover edge $\{x, y\}$.

As a consequence of the lemma we have the following corollary.

Corollary 4.3 *Let G be a graph and e an edge in G. Then for* LG-ALL-DISCOVERY $OPT(G \setminus e) \leq OPT(G) + 2$.

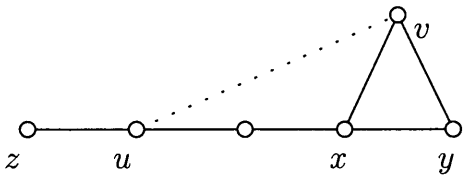


Figure 4.1: Querying $Q(G) = \{z, v\}$ and the endpoints u and v of a newly added edge into G does not discover $G' = G \cup \{u, v\}$.

In [27, 76, 49] some special graph classes were investigated. For cycles, wheels, lines, trees and complete graphs the optimum number of queries was determined. Also, a full characterization of graphs for which the optimum number of queries is 1, $n - 1$ and $n - 2$ was given. In [76] some properties of graphs for which 2 queries discover the graph were given. We now give a full characterization of such graphs. We can characterize the graphs verifiable by 2 queries by a series of simple observations. Let $Q = \{q_1, q_2\}$ be the query set verifying G.

1. Every vertex $v \in V(G)$ can by uniquely characterized by a tuple (i_v, j_v), where $i_v = d(q_1, v)$ and $j_v = d(q_2, v)$, because $\forall u, v \in V$: $d(q_1, u) \neq d(q_1, v)$ or $d(q_2, u) \neq d(q_2, v)$.

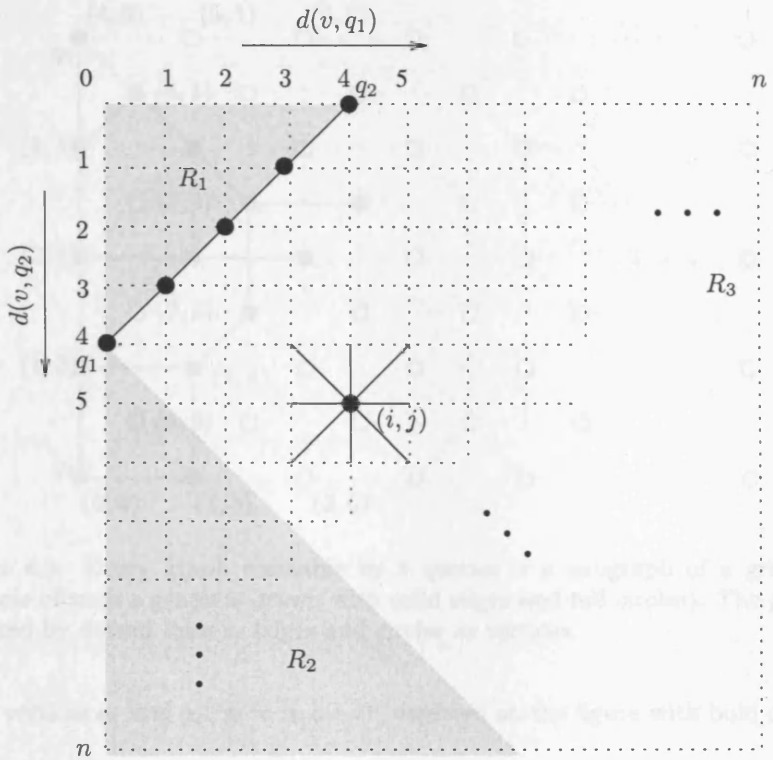


Figure 4.2: Graph verified by 2 queries q_1 and q_2 can be drawn in a grid with special properties.

Placing the vertices on a two dimensional grid of size $n \times n$ according to their coordinates (i_v, j_v) can give a better understanding of G (see Figure 4.2).

2. Every vertex (i, j) of G has at most 8 neighbours $((i-1, j-1), (i-1, j), (i-1, j+1), (i, j-1), (i, j+1), (i+1, j-1), (i+1, j)$, or $(i+1, j+1))$, i.e., G is a subgraph of a complete grid.

3. The diameter $diam_G$ of G is at least $\sqrt{n} - 1$. If not, then all the vertices of G would fit into the box of size $(diam_G + 1) \times (diam_G + 1)$, which would yield $|V| \leq (diam_G + 1)^2 < (\sqrt{n})^2 = n$, a contradiction.

4. Vertices q_1 and q_2 are placed symmetrically on the boundary of the grid: $q_1 = (0, a)$, $q_2 = (a, 0)$.

5. There is a unique path $P = p_0, p_1, p_2, \ldots, p_a$ of length a between the query

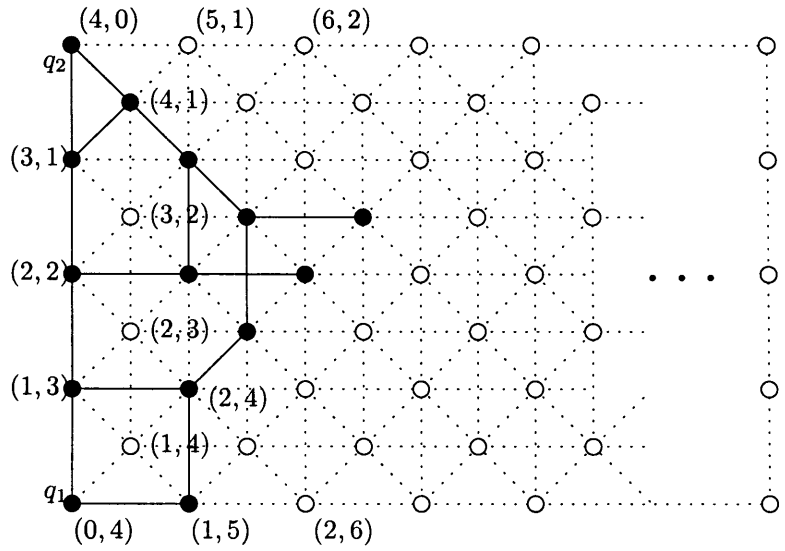


Figure 4.3: Every graph verifiable by 2 queries is a subgraph of a grid (an example of such a graph is drawn with solid edges and full circles). The grid is depicted by dotted lines as edges and circles as vertices.

vertices q_1 and q_2: $p_i = (i, a - i)$ (depicted on the figure with bold disks).

6. As a consequence of the above reasoning, there is no vertex in area $R_1 := \{(i,j) \mid i+j < a\}$. To see this, suppose for contradiction that there exists a vertex $v \in G$ such that $(i_v, j_v) \in R_1$. Then $d(q_1, q_2) \leq d(q_1, (i_v, j_v)) + d((i_v, j_v), q_2) = i_v + j_v < a = d(q_1, q_2)$, a contradiction. Similarly one can show there is no vertex from G in areas $R_2 := \{(i,j) \mid j - i > a\}$ and $R_3 := \{(i,j) \mid i - j > a\}$. E.g. using the triangle inequality, one gets for every vertex $(i,j) \in G$ that $j = d((i,j), q_2) \leq d((i,j), q_1) + d(q_1, q_2) = i + a$, i.e, $j - i \leq a$, etc. Yet another simple bound is: for every vertex $(i,j) \in G$, i and j is at most $n - 1$.

7. Thus, q_1 and q_2 are the corners of a slab of the grid into which G is embedded (the strip/lane/slab which remains when subtracting R_1, R_2 and R_3 from the integer lattice)—see Figure 4.3.

8. Both q_1 and q_2 have at most 3 neighbours each (otherwise there would be a vertex in R_1, R_2 or R_3).

9. Every vertex (i,j) except q_1 and q_2 has a neighbour in $(i-1, j-1)$, $(i-1, j)$, $(i-1, j+1)$ and a neighbour (not necessarily distinct) in $(i-1, j-1)$,

$(i, j-1)$ and $(i+1, j-1)$ (e.g. the neighboring vertex on shortest paths from (i,j) to q_1 and the neighboring vertex on shortest path from (i,j) to q_2.

Thus, every graph G which is discovered by 2 queries is $(*)$:

a subgraph of slab $\{(i,j) \mid |i-j| \leq a\}$, $a \geq 1$, of a 2 dimensional grid $\{(i,j) \mid i,j \geq 0\}$ (where also the diagonal edges are present), such that every vertex $v \in V$ at position (i,j) (except $(0,a)$ and $(a,0)$, which satisfy only one of the following constraints) of the grid has at least one neighbour in G among positions $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$ and at least one neighbour among positions $(i-1, j-1)$, $(i, j-1)$ and $(i+1, j-1)$.

It is easy to verify that every graph with such a property $(*)$ can be discovered by the two neighboring corner vertices ($q_1 = (0,a)$ and $q_2 = (a,0)$ from the grid of Figure 4.3).

We notice that these graphs don't have to be planar, as planar graphs have to satisfy $|E| \leq 3|V| - 6$.

As a generalization of the previous discussion, we can give some insight for graphs for which k queries are needed to discover them. For $Q = \{q_1, \ldots, q_k\}$ that discovers G, we can represent every vertex v by a k-tuple $(i_1^v, i_2^v, \ldots, i_k^v)$, where $i_j^v = d(v, q_j)$.

Therefore,

1. graph G is a subgraph of a k dimensional grid with diagonals.
2. $diam_G \geq \sqrt[k]{n} - 1$.
3. $\Delta_G \leq 3^k - 1$. (Every neighbour of $v = (x_1^v, x_2^v, \ldots, x_k^v)$ (including v itself) can differ in every coordinate by at most 1).
4. Every query vertex q_i has degree at most 3^{k-1}. (Here the neighbours of q_i have their i-th coordinate fixed (equal to 1) and can differ in the remaining $k-1$ coordinates).

Observe that the second and third characterization (but not the last one) follows also from Theorem 4.1.

4.3.2 The Online Problem

Lower Bound

Theorem 4.4 *No deterministic online algorithm for* LG–ALL–DISCOVERY *can have weak competitive ratio* $3 - \varepsilon$ *for any* $\varepsilon > 0$.

Proof. Let A be any deterministic algorithm for LG–ALL–DISCOVERY. We first give a simpler proof that A cannot be better than 2-competitive. This gives an insight for the ingredients we use to prove the main claim of the theorem. Consider Figure 4.4(a). We refer to the subgraph induced by the vertices labelled r, x, y, and z as a 2-*gadget*. Assume that the given graph G consists of a global root g and k, $k \geq 2$, disjoint copies of the 2-gadget, with the r-vertex of each 2-gadget connected to the global root g. One can easily verify that $\mathrm{OPT}(G) = k$ for this graph, and that the set of all x-vertices (or y-vertices) of the 2-gadgets constitutes an optimal query set. On the other hand, algorithm A can be forced to make the first query at g (as, initially, the vertices are indistinguishable to the algorithm). This will not discover any information about edges or non-edges between vertices x, y and z of each 2-gadget. The only queries that can discover this information are queries at x, y and z. In fact, a query at x or y suffices to discover the edge between x and y and the non-edges between x and z and between y and z. When A makes the first query among the vertices in $\{x, y, z\}$ of a 2-gadget, we can force it to make that query at z, since the three vertices are indistinguishable to the algorithm. The query at z does not discover the edge between x and y. The algorithm must make a second query in the 2-gadget to discover that edge. In total, the algorithm must make at least $2k+1$ queries. As the construction works for arbitrary values of k, this shows that no deterministic online algorithm can guarantee weak competitive ratio $2 - \varepsilon$ for any constant $\varepsilon > 0$.

To get a stronger lower bound of 3, we create a new gadget, called the 3-*gadget*, as shown in Figure 4.4(b). The 3-gadget is the subgraph induced by all vertices except g in the figure. We claim that A can be forced to make 6 queries in each 3-gadget, whereas the optimum query set consists of only 2 vertices in each 3-gadget (drawn shaded in the figure). If we construct a graph with k, $k \geq 2$, disjoint copies of the 3-gadget, the s-vertex in each of them connected to

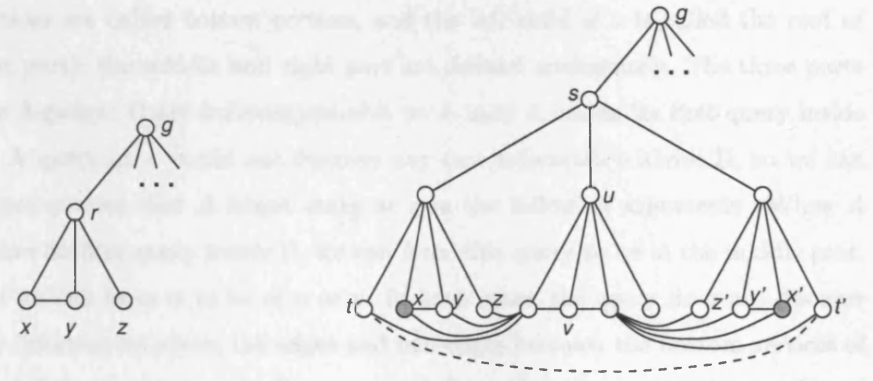


Figure 4.4: Lower bound constructions.

the global root g as indicated in the figure, we get a graph G for which we claim that $\mathrm{OPT}(G) = 2k$ and the algorithm A can be forced to make at least $6k + 1$ queries. This shows that no deterministic online algorithm can guarantee weak competitive ratio $3 - \varepsilon$ for any constant $\varepsilon > 0$.

To see that $\mathrm{OPT}(G) = 2k$, let Q be the set of queries consisting of the two shaded vertices from each copy of the 3-gadget as shown in Figure 4.4(b). We claim that Q discovers G. This can be verified manually as follows: For every vertex in a 3-gadget Π, consider the 3-tuple whose components are the distances from that vertex to the two query vertices in Π and the distance to an arbitrary query vertex from Q outside Π. One finds that each vertex in Π has a unique 3-tuple, showing that all edges and non-edges of Π are discovered by Q. Each non-edge between two different 3-gadgets is discovered by one of the queries inside these two 3-gadgets. The edges and non-edges between g and each 3-gadget are also discovered. Hence, $\mathrm{OPT}(G) \leq 2k$. We have $\mathrm{OPT}(G) \geq 2k$, because each of the edges $\{x, y\}$ and $\{x', y'\}$ (see Figure 4.4(b)) of a 3-gadget requires a separate query.

To show that $A(G) \geq 6k + 1$, we argue as follows. First, we can force A to make the first query at g. This will not reveal any information about edges within the same layer of any of the 3-gadgets. We view each 3-gadget as consisting of s and a left part, a middle part, and a right part. The left part consists of the left child of s and its four adjacent vertices below (these four

vertices are called *bottom vertices*, and the left child of s is called the *root* of that part); the middle and right part are defined analogously. The three parts of a 3-gadget Π are indistinguishable to A until it makes its first query inside Π. A query at s would not discover any new information about Π, so we can ignore queries that A might make at s in the following arguments. When A makes its first query inside Π, we can force this query to be in the middle part, and we can force it to be at u or v. In both cases, the query does not discover any information about the edges and non-edges between the bottom vertices of the left part, nor does it discover any information about the edges and non-edges between the bottom vertices of the right part, nor does it discover the edge drawn dashed. When A chooses its second query in Π, it could be in the left part, in the middle part, or in the right part. Assume that A chooses the left part; since the bottom vertices of the left part are still indistinguishable to A, we can force A to make the query either at the root of the left part or at the bottom vertex t. Similarly, in the right part we can force A to make the query at its root or at t'. In the middle part, A can make the query anywhere. In any case, the second query made by A does not discover any information about edges and non-edges between vertices in the set $\{x, y, z\}$ and in the set $\{x', y', z'\}$. Similarly as in the case of Figure 4.4(a), for each of these sets we can force A to make the first query at z (at z') and thus require a second query at x or y (at x' or y') to discover everything about these groups. In total, A must make at least 6 queries in each 3-gadget. □

With the gadget of Figure 4.4(a) one can prove easily that no randomized on-line algorithm for LG–ALL–DISCOVERY can have weak competitive ratio $4/3-\varepsilon$ for any $\varepsilon > 0$; just observe that we can force a randomized algorithm to make the first query at z with probability at least $1/3$. All lower bounds on the weak competitive ratio also hold for the (standard) competitive ratio (where no additive constant c is allowed).

A Deterministic Algorithm

We describe now an algorithm for LG–ALL–DISCOVERY that is inspired by Lemma 4.2. The algorithm makes an arbitrary query q_0. This splits the vertices of G into layers with discovered edges and non-edges between the layers. Observe

that any breadth-first-search tree rooted at q_0 contains only edges discovered by q_0. The algorithm fixes one such breadth-first-search tree T. Let Q_T be an optimal query set that discovers T (this can be computed in polynomial time due to [76]). Let Q_E be the set of endpoints of edges of G that are not in T, i.e., $Q_E = \{u \mid \exists v \in V(G) : \{u, v\} \in E(G) \setminus E(T)\}$. The algorithm makes queries at Q_T, one by one, and during this phase also at all end-points of all newly discovered edges in the process of querying, until all edges and non-edges are discovered, or there is no query to be made.

We claim that the algorithm's computed query set Q discovers G. Let Q'_E be the endpoints of the edges that were discovered by the algorithm. Suppose by way of contradiction that the algorithm terminates (i.e., it queried all vertices from $Q = Q_T \cup Q'_E$) and that there is an undiscovered pair $\{x, y\}$—a non-edge or an edge. Let $q \in Q_T$ be the query that discovers $\{x, y\}$ in T (as a non-edge). Because q does not discover $\{x, y\}$ in G, the shortest paths (in G) from q to x or the shortest paths from q to y (or both) contain some newly discovered edges (i.e., an edge not from T). Let $\{u, v\}$ be such a new edge on a shortest path from q to x or from q to y, traversed in the order u, v. The algorithm queried both endpoints u and v and because $\{x, y\}$ is undiscovered, $d_G(v, x) = d_G(v, y)$ and also $d_G(u, x) = d_G(u, y) = d_G(v, x) + 1$. Hence, we can assume that the shortest paths from q to x and from q to y agree on the part from q to v. Moreover, let $\{u, v\}$ be such a newly discovered edge with a minimum distance from v to x and y. We claim that the shortest path from v to y contains only edges from T. If not, then there would be a newly discovered edge $\{u', v'\}$ (of which both endpoints were queried) with v' closer to y and x than v is. Hence the shortest path from v to y contains only edges of T and for the same reasons the shortest path from v to x contains only edges of T, too. Thus, x and y are at the subtree T_v of T, rooted at v, and at the same depth. The query q discovers $\{x, y\}$ in T and therefore q is in T_v and the shortest path P_T from q to x in T does not contain v. Let g be the vertex from P_T that is closest to v. Let L_{i_v}, L_{i_q}, L_{i_g} and $L_{i_{x,y}}$ be the layers of q_0 with vertices v, q, g and x, respectively. Observe that the length of P_T is $\ell = (i_q - i_g) + (i_x - i_g)$. The shortest path P_G from q to x in G goes via v and therefore the length of such a path is at least $(i_q - i_v) + (i_x - i_v) > \ell$ (as all the edges of G connect neighbouring layers or lie

within layers of the initial query q_0), which is certainly not possible.

Thus the algorithm discovers G and the number of queries it makes is at most $1 + |Q_T| + |Q_E| = 1 + |Q_T| + 2|E(G) \setminus E(T)|$. Repeated usage of Lemma 4.2 (starting with G and deleting edges until we end up with T) shows that $|Q_T| = \mathrm{OPT}(T) \leq \mathrm{OPT}(G) + 2|E(G) \setminus E(T)|$.

Theorem 4.5 *There exists an online (polynomial time) algorithm for* LG–ALL–DISCOVERY *which computes a query set Q (that discovers an unknown network G) of size at most $OPT(G) + 4k + 1$, where $k = |E(G)| - (n-1)$.*

Observe that the algorithm guarantees to deliver $OPT(G) + o(n)$ queries only if the number of edges is $n + o(n)$. Otherwise, from the worst-case-analysis point of view, querying all n vertices would achieve the same upper bound.

Randomized Online Algorithm

Theorem 4.6 ([18]) *There is a randomized online algorithm with competitive ratio $O(\sqrt{n \log n})$ for* LG–ALL–DISCOVERY.

Proof. The online algorithm is shown in Figure 4.5. The algorithm consists of two phases. In the first phase, it makes $3\sqrt{n \ln n}$ queries at nodes chosen uniformly at random. In the second phase, as long as node pairs with unknown status exist, it picks an arbitrary such pair $\{u, v\}$ and proceeds as follows. First, it queries u and v in order to determine the distance of all nodes to u and v. From this it can deduce the set S of nodes from which the edge or non-edge between u and v can be discovered; these are simply the nodes for which the distance to u differs from the distance to v. Then, it queries all remaining nodes in S.

To analyze the algorithm, it is helpful to view LG–ALL–DISCOVERY as a HITTINGSET problem [62]. For every edge or non-edge $\{u, v\}$, let S_{uv} be the set of nodes from which a query discovers $\{u, v\}$. The task of the LG–ALL–DISCOVERY problem translates into the task of computing a subset of V that hits all sets S_{uv}. The goal of the first phase is to hit all sets that have size at least $\sqrt{n \ln n}$ with high probability. If this succeeds, the problem remaining for the second phase is a HITTINGSET problem where all sets have size at most $\sqrt{n \ln n}$. The algorithm of the second phase repeatedly picks an arbitrary set

```
E ← ∅;   /* discovered edges */
N ← ∅;   /* discovered non-edges */
A ← (V choose 2);   /* all pairs of distinct nodes */
/* Phase 1 */
for i = 1 to 3√(n ln n) do
    v ← randomly chosen node from V;
    (E_v, N_v) ← query(v);
    E ← E ∪ E_v;
    N ← N ∪ N_v;
od;
/* Phase 2 */
while E ∪ N ≠ A do
    {u, v} ← an arbitrary element of A \ (E ∪ N);
    (E_u, N_u) ← query(u);
    (E_v, N_v) ← query(v);
    E ← E ∪ E_u ∪ E_v;
    N ← N ∪ N_u ∪ N_v;
    S ← set of nodes from which the (non-)edge
        {u, v} is discovered;
    foreach x ∈ S \ {u, v} do
        (E_x, N_x) ← query(x);
        E ← E ∪ E_x;
        N ← N ∪ N_x;
    od;
od;
```

Figure 4.5: On-line algorithm for LG–ALL–DISCOVERY.

that is not yet hit, and includes all its elements in the solution. As the sets have size at most $\sqrt{n \ln n}$, the number of queries made in the second phase is at most a factor of $\sqrt{n \ln n}$ away from the optimum.

Let us make this analysis precise. Consider a node pair $\{u, v\}$ for which the set S_{uv} has size at least $\sqrt{n \ln n}$. In each query of the first phase, the probability that S_{uv} is not hit is at most $1 - \frac{\sqrt{n \ln n}}{n} = 1 - \frac{\sqrt{\ln n}}{\sqrt{n}}$. Thus, the probability that S_{uv} is not hit throughout the first phase is at most $\left(1 - \frac{\sqrt{\ln n}}{\sqrt{n}}\right)^{3\sqrt{n \ln n}} = \left(\left(1 - \frac{\sqrt{\ln n}}{\sqrt{n}}\right)^{\frac{\sqrt{n}}{\sqrt{\ln n}}}\right)^{3 \ln n} \leq e^{-3 \ln n} = \frac{1}{n^3}$. There are at most $\binom{n}{2}$ sets S_{uv} of cardinality at least $\sqrt{n \ln n}$. The probability that at least one of them is not hit in the first phase is at most $\binom{n}{2} \cdot \frac{1}{n^3} \leq \frac{1}{n}$.

Consider the second phase, conditioned on the event that the first phase has hit all sets S_{uv} of size at least $\sqrt{n \ln n}$. In each iteration the algorithm asks at most $\sqrt{n \ln n}$ queries. Let ℓ be the number of iterations. It is clear that the

optimum must make at least ℓ queries, because no two unknown pairs $\{u, v\}$ considered in different iterations of the second phase can be resolved by the same query.

Since $\mathrm{OPT}(G) \geq 1$ and $\mathrm{OPT}(G) \geq \ell$, the number of queries made by the algorithm is at most $3\sqrt{n \ln n} + \ell\sqrt{n \ln n} = O(\sqrt{n \log n}) \cdot \mathrm{OPT}(G)$.

With probability at least $1 - \frac{1}{n}$, the first phase succeeds and the algorithm makes $O(\sqrt{n \log n}) \cdot \mathrm{OPT}(G)$ queries. If the first phase fails, the algorithm makes at most n queries. This case increases the expected number of queries made by the algorithm by at most $\frac{1}{n} \cdot n = 1$. Thus, the expected number of queries is at most $O(\sqrt{n \log n}) \cdot \mathrm{OPT}(G) + \frac{1}{n} \cdot n = O(\sqrt{n \log n}) \cdot \mathrm{OPT}(G)$. □

4.4 The Distance Query Model

4.4.1 Discovering Individual Edges and Non-Edges

In the layered-graph query model, a query at node v resulted explicitly in a set of edges and non-edges that were discovered by the set of queries. In the distance query model, this is no longer the case. The query at node v returns (only) the distances to all other nodes of the network. We want to compute a set Q of queries that discover the graph as defined in Section 4.1. It may be not clear in the first sight, how an algorithm for the problem could discover the individual edges and non-edges. Here we give a characterization, when a query (or a set of queries) from Q discovers individual edges and non-edges.

Characterizing the Queries Discovering a Non-Edge. If we look at a particular non-edge $e \in \bar{E}$, there exists a query $q \in Q$ that confirms this non-edge to be in G:

Observation 4.7 *For $G = (V, E)$ the queries $Q \subseteq V$ discover a non-edge $\{u, v\} \in \bar{E}$ if and only if there exists a query $q \in Q$ with $|d(q, u) - d(q, v)| \geq 2$.*

Proof. The implication "$\Leftarrow$" is easy to see: Clearly, if there is a query q such that $|d(q, u) - d(q, v)| \geq 2$, then $\{u, v\}$ is a non-edge. To see the second implication "$\Rightarrow$", assume that $\{u, v\}$ is a non-edge and that (for contradiction) every query node q gives $|d(q, u) - d(q, v)| \leq 1$. We show that if $\{u, v\}$ was an

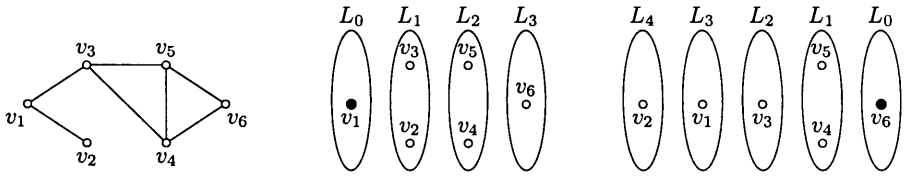


Figure 4.6: Edge $\{v_3, v_4\}$ of a graph (left) is discovered by the combination of queries at nodes v_1 and v_6; the distances to the query node v_1 (middle) and v_6 (right) are depicted via layers of the graph.

edge, the distances returned by Q would not change (a contradiction). Indeed, u and v are either in the same layer or in two consecutive layers of a query q. Therefore adding an edge $\{u, v\}$ to G cannot decrease a distance from q to any other node. □

Thus, a non-edge $\{u, v\}q$ is discovered by Q if there is a query $q \in Q$ such that $|d(q, u) - d(q, v)| \geq 2$.

Characterizing the Sets of Queries Discovering an Edge. An edge may be discovered by a combination of several queries (this is a major difference to the layered-graph query model, where the set of edges and non-edges discovered by a set of queries is simply the union of the edges and non-edges discovered by the individual queries). If a node w is in layer $i+1$ of a query q, this shows that w must be adjacent to at least one node from layer i. If layer i has more than one node, then in general it is not clear which node from layer i is adjacent to w. Figure 4.6 shows an example of how a combination of two queries can discover an edge even if each of the two queries alone does not discover the edge: The edge $\{v_3, v_4\}$ is neither discovered by a query at v_1 nor by a query at v_6 alone. The query at v_1 reveals that v_4 is connected to v_2 or to v_3. The query at v_6 identifies $\{v_2, v_4\}$ as a non-edge. From these two facts one can deduce that v_4 must be connected to v_3, i.e., $\{v_3, v_4\}$ is an edge. This discussion is generalized by the following observation.

Observation 4.8 *For $G = (V, E)$ the queries $Q \subseteq V$ discover an edge $\{u, v\} \in E$ if and only if there is a query $q \in Q$ with the following two properties:*

(i) The nodes u and v are in consecutive layers of query q, say, u in the i-th layer L_i and v in the $(i+1)$-th layer L_{i+1}, and $L_i \setminus \{u\}$ does not contain any neighbour of v.

(ii) The queries Q discover all non-edges between v and the nodes in $L_i \setminus \{u\}$.

Proof. We again start with the easy direction "$\Leftarrow$": From the result of query q in (i) one can deduce that there must be an edge from some node in L_i to v. From (ii) it follows that $\{u, v\}$ is the only possibility for such an edge.

For the implication "$\Rightarrow$", we give a proof by contradiction. Assume that the query set Q discovers the edge $\{u, v\}$. Observe that if (i) does not hold, then all queries yield the same results if $\{u, v\}$ is removed from G. To see this, consider an arbitrary query $q' \in Q$. If u, v are originally at the same distance from q', they also will be at the same distance after removing $\{u, v\}$. If u, v are originally at different distances from q', say $u \in L_{i'}$ and $v \in L_{i'+1}$, we know that since (i) does not hold, v has another neighbour in $L_{i'} \setminus \{u\}$. Therefore, we know that v is in $L_{i'+1}$ even after removing the edge $\{u, v\}$. So in this case as well, $\mathbf{D}_G(Q)$ does not change if we remove $\{u, v\}$. This contradicts our assumption that Q discovers $\{u, v\}$.

Thus, we can assume that (i) holds. For each $q \in Q$ for which (i) holds, assume that (ii) does not hold. Let q be a query for which (i) holds. Assume that u is in layer L_i of that query and v is in layer L_{i+1}. As (ii) does not hold, there must be at least one non-edge $e_q = \{u', v\}$ for some $u' \in L_i$ that is not discovered by Q. We modify the graph G as follows: We remove the edge $\{u, v\}$, and we add the edges e_q for all $q \in Q$ for which (i) holds (these edges e_q are not necessarily distinct). It is easy to see that the resulting graph G' satisfies $\mathbf{D}_{G'}(Q) = \mathbf{D}_G(Q)$, proving that Q does not discover the edge $\{u, v\}$ in G, a contradiction. □

We say that a query for which (i) holds is a *partial witness* for the edge $\{u, v\}$. The word "partial" indicates that the query alone is not necessarily sufficient to discover the edge; additional queries may be necessary to discover the non-edges required by (ii).

We conclude that a set of queries discovers a graph G if and only if it discovers all non-edges and contains a partial witness for every edge.

4.4.2 Structural Properties

In this section we show lower bounds on the number of queries needed to discover G. We relate this number to the independence number α of the graph, to the

clique number ω of the graph, and to the number of edges m. We also adopt Lemma 4.2 for the distance query model.

Lemma 4.9 *For any graph G with independence number α and diameter* $\mathrm{diam} > 2$, *at least* $\log_{\lceil \frac{\mathrm{diam}}{2} \rceil}(\alpha) - 1$ *queries are needed to discover G. If* $\mathrm{diam} = 2$, *we need at least $\alpha - 1$ queries.*

Proof. Let $A_0 \subseteq V$ be an independent set of size α. Any query q splits the nodes into at most $\mathrm{diam} + 1$ layers. In layer 0 there is only q itself. We merge each pair of consecutive layers $2i-1$ and $2i$, for $i \geq 1$, so that we obtain at most $\beta := \lceil \frac{\mathrm{diam}}{2} \rceil$ new layers $\tilde{L}_i$ (the last new layer may consist of a single original layer). Query q does not discover any non-edge whose endpoints lie within the same new layer. At least $\alpha - 1$ nodes of the independent set A_0 are distributed among the β new layers (one node of A_0 may be the query node, which is not in the new layers). Thus, there must be a new layer $\tilde{L}_i$ with at least $(\alpha-1)/\beta$ nodes from A_0. Let A_1 denote the set of these nodes. If $(\alpha - 1)/\beta > 1$, then we need at least one more query to discover the non-edges within A_1. After the second query, there is a new layer containing at least $(|A_1| - 1)/\beta \geq ((\alpha - 1)/\beta - 1)/\beta$ nodes from A_1, and the argument can be repeated. Let α_k, for $k \geq 1$, denote the size of the biggest subset of A_0 for which the queries $q_1, \ldots, q_k$ do not discover any non-edge. By the arguments above, we have $\alpha_k \geq a_k$, where $a_0 = \alpha$ and $a_k = \frac{a_{k-1}-1}{\beta}$ for $k \geq 1$. We get $a_k = \frac{\alpha}{\beta^k} - \frac{1}{\beta^k} - \frac{1}{\beta^{k-1}} - \ldots - \frac{1}{\beta}$, i.e., $a_k = \frac{1}{\beta^k}(\alpha - \frac{\beta^k - 1}{\beta - 1})$ if $\beta > 1$ and $a_k = \alpha - k$ if $\beta = 1$. If k queries discover G, we must have that $a_k \leq 1$. For $\beta = 1$ we get $k \geq \alpha - 1$. For $\beta > 1$ we get $\beta^{k+1} \geq 1 + \alpha(\beta - 1)$, i.e., $k \geq \log_\beta \alpha + \log_\beta (\beta - 1 + \frac{1}{\alpha}) - 1 \geq \log_\beta \alpha - 1$. □

Lemma 4.10 *For any graph G with clique number ω we need at least $\omega - 1$ queries to discover G.*

Proof. Consider a clique $K_\omega \subseteq G$ of size ω. Let q be the first query. The nodes of K_ω appear in at most two consecutive layers i and $i+1$ of query q. Observe that q is a partial witness of an edge from K_ω if and only if there is exactly one node v from K_ω in layer i and the rest is in layer $i+1$. Moreover, q is a partial witness only for edges incident on v. After query q, there is still a $K_{\omega-1}$ for which no query has been made that is a partial witness of any of its edges.

Therefore, by induction (using the fact that one query is necessary for a K_2 as the base case), it follows that we need at least $\omega - 1$ queries to discover G. □

Lemma 4.11 *Any graph G with n nodes and m edges needs at least $m/(n-1)$ queries to be discovered.*

Proof. Consider the layers of an arbitrary query $q \in V$. For each node v on layer i, q can be a partial witness for at most one edge $\{u, v\}$ with u in layer $i-1$. Therefore, q can be a partial witness for at most $n - 1$ edges. Since a set of queries that discovers G must contain a partial witness for each of the m edges of G, the bound follows. □

This lower bound shows that graphs with a super-linear number of edges need a non-constant number of queries to be discovered.

Lemma 4.12 *If $Q \subseteq V$ discovers a graph $G = (V, E)$ in* DIST–ALL–DISCOVERY, *then $Q \cup \{u, v\}$ discovers $G' = G \setminus \{u, v\}$, for any edge $\{u, v\} \in E$.*

Proof. Let $x, y \in V(G)$. We now distinguish two cases. Suppose first $\{x, y\}$ is a non-edge. If u or v discovers $\{x, y\}$, the lemma follows. Assume therefore that u and v do not discover $\{x, y\}$, i.e., $|d_{G'}(u, x) - d_{G'}(u, y)| \leq 1$ and $|d_{G'}(v, x) - d_{G'}(v, y)| \leq 1$. Let z be a query from Q that discovers $\{x, y\}$ in G. Assume w.l.o.g. that $d_G(z, x) \leq d_G(z, y) - 2$. Then the edge $\{u, v\}$ cannot lie on any shortest path from z to x in G (otherwise we could make the distance from z to y in G shorter going via u or v). Thus $d_{G'}(z, x) = d_G(z, x) \leq d_G(z, y) - 2 \leq d_{G'}(z, y) - 2$. Hence, z discovers $\{x, y\}$ also in G'.

Suppose now that $\{x, y\}$ is an edge. We show that there exists a partial witness in $Q \cup \{u, v\}$ for the edge in G'. Let $z \in Q$ be a partial witness for the edge in G where, w.l.o.g., $d_G(z, x) < d_G(z, y)$. If $\{u, v\}$ appears both on shortest paths from z to x and from z to y (v is closer to x than u is) then v is a partial witness for $\{x, y\}$ in G'. If $\{u, v\}$ does not appear on a shortest path from z to x, then $d_{G'}(z, x) = d_G(z, x)$ and the distance from z to y in G' cannot get shorter. Therefore z is a partial witness for $\{x, y\}$. □

4.4.3 Polynomially Solvable Cases

Lemma 4.13 *G needs 1 query to be discovered if and only if G is a chain. A clique K_n needs $n - 1$ queries to be discovered.*

Proof. If G is a chain, then clearly a vertex of degree 1 discovers the chain. On the other hand, if one query q discovers the whole graph G, observe that q cannot discover an edge or non-edge between two vertices at the same distance from q. Therefore, the vertices of G have unique distance from q and therefore G is a chain.

The second part of the statement follows from Lemma 4.10 and since with $n-1$ queries each edge has at least one incident query and therefore will be discovered. □

The example of the cycle with 4 nodes C_4 shows that there is a graph that needs $n-1$ queries to be discovered and is not a clique. (The same holds for graphs that are obtained from K_n by deleting one edge, for $n \geq 4$.) In general, for cycles the following lemma holds.

Lemma 4.14 *A cycle C_n, $n > 6$, needs 2 queries to be discovered.*

Proof. By Lemma 4.13 we have that 1 query does not discover a cycle. We show now that 2 queries are enough. We argue for n being odd, i.e., $n = 2k+1$. Similar arguments can be given for even n.

Let $V = \{v_0, \ldots, v_{n-1}\}$ be ordered according to their appearance on the cycle. Let $q_1 = v_0$ and $q_2 = v_2$ be two queries at C_n. Query v_0 divides the vertices into layers according to the distance. In every layer $i \geq 1$ there are 2 vertices v_i and v_{n-i} (see Figure 4.7). Observe that q_1 is a partial witness for all edges except $\{v_k, v_{k+1}\}$, and q_2 is a partial witness for $\{v_k, v_{k+1}\}$ (cf. Figure 4.7).

Query q_1 discovers all non-edges between vertices from non-neighboring layers. We show that q_2 discovers all the remaining undiscovered non-edges of type $\{v_i, v_{n-i}\}$, $\{v_i, v_{n-i-1}\}$ and $\{v_{i+1}, v_{n-i}\}$, for $i = 1, 2, \ldots, k-1$. Notice that $\{v_1, v_{n-1}\}$ and $\{v_1, v_{n-2}\}$ are the only unknown non-edges incident on v_1 after query q_1. Observe that if $n > 6$, query q_2 discovers these non-edges. Hence we consider an unknown non-edge $\{v_a, v_b\}$ where $a > 2$ and $a \leq k$ and $b \geq k+1$, $b \in \{n-a+1, n-a, n-a-1\}$. The distance d_a from v_2 to v_a can be used to bound the distance to v_b as follows: $d(v_2, v_b) \geq \min\{4 + (d_a - 1), d_a + 2\} \geq d_a + 2$ (by considering the lengths of the two paths from v_2 to v_b via v_0 or via v_k). Thus the distances $d(q_2, v_a)$ and $d(q_2, v_b)$ differ by at least two and therefore q_2 discovers the non-edge $\{v_a, v_b\}$.

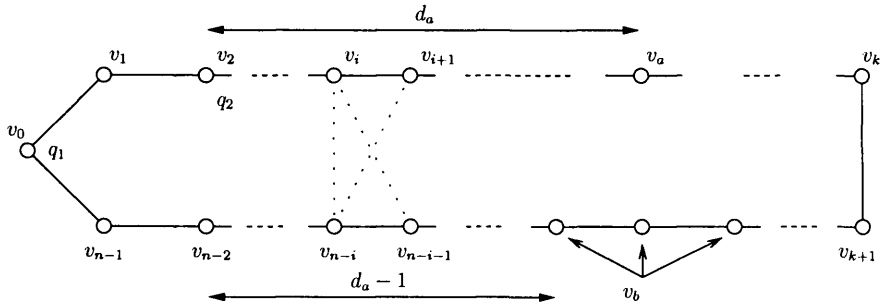


Figure 4.7: Cycle C_n can be discovered by queries at v_0 and v_2.

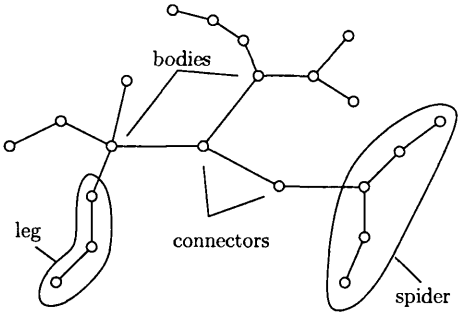


Figure 4.8: Legs, bodies, spiders and connectors in a tree.

We showed that q_1 and q_2 discover all non-edges and are partial witness for all edges. Therefore q_1 and q_2 discover the cycle C_n. □

Now we characterize the optimal query set for a tree T. For this, we define a *leg* to be a maximal path in the tree starting at a leaf and containing only vertices of degree at most 2, see Figure 4.8. Therefore, if T is not a chain, there has to be a node u of degree greater than 2 adjacent to the last vertex of the leg. We call u a *body* and we say that the leg is *adjacent* to its body u. The body u with all its adjacent legs is called a *spider*. Nodes that are not part of a spider are called *connectors* (i.e., nodes that are not in a leg and have no adjacent leg).

Lemma 4.15 *Let $T = (V, E)$ be a tree that is not a chain. Denote by $B \subset V$ the set of bodies of the graph. Let l_b, for $b \in B$, be the number of legs adjacent to b. Let $T[B]$ be the induced subgraph of T on vertex set B. Let $VC(T[B])$ denote a minimum vertex cover of $T[B]$. Then the minimum number of queries to discover T is $\sum_{b \in B}(l_b - 1) + |VC(T[B])|$.*

Proof. We first show that we indeed need at least this many queries. For this observe that if there is no query in two legs adjacent to a body, then we cannot discover the non-edges formed by vertices of the two legs at the same distance from the body. Therefore there has to be at least one query in every leg but one of any body. Moreover, if there are two legs of two different bodies which are connected by an edge then there has to be at least one query in one of the legs. Otherwise we cannot discover the non-edge between vertices of the legs at the same distance from their bodies. Therefore for any two bodies connected by an edge at least one of them has a query in every leg. Observe that the bodies with all legs containing a query form a vertex cover of $T[B]$ and therefore a minimum vertex cover gives a lower bound on the number of spiders that have a query in every leg.

To prove that the claimed number of queries is sufficient, we construct a query set Q in the following way. We compute a minimum vertex cover of $T[B]$ (which can be done in polynomial time on trees). Let u be a body. We add the leaves of $l_u - 1$ of its legs to Q. If u is in the vertex cover, we add also the leaf of the last (the l_u-th) leg to Q.

We show now that Q discovers T. We start with non-edges. Let $\{v, w\}$ be a non-edge. We distinguish several cases. First, consider the case that both v and w are from legs. Consider the following subcases.

1. v and w are from the same leg. Clearly, the non-edge is discovered by any query.

2. v and w are from different legs, and there is a query q in the leg where v or w is. This query discovers the non-edge. (Note that there must be a query in the leg of v or w if they are in different legs of the same spider, or in legs of spiders whose centers are adjacent.)

3. v and w are from different spiders centered at u and u', which are not neighbours, and there is no query in the legs containing v and w. Let the path from u to u' be $u, x, \ldots, y, u'$, where $x = y$ is possible. Let q be a query from a leg adjacent to a body b such that the path from b to u does not contain x, possibly $b = u$. Let d_v be the distance from u to v, d_w be the distance from u' to w and let $d \geq 2$ be the distance between u and

u'. If q does not discover the non-edge $\{v, w\}$ then $|d(q, v) - d(q, w)| = |d_v - (d + d_w)| \leq 1$. Then a query q' from a leg adjacent to a body b' such that the path from b' to u' does not contain y satisfies $|d(q', v) - d(q', w)| = |(d_v + d) - d_w| \geq 3$ and thus q' discovers the non-edge.

Now, consider the case that at least one of the two nodes, say, the node v, is not from a leg. Then any query in a tree of the forest $T \setminus \{v\}$ that does not contain w verifies the non-edge. Observe that such a query always exists.

Therefore Q discovers all non-edges. We claim now that Q discovers all edges. For this observe that for a tree T any query is a partial witness for every edge. To see this imagine the tree rooted at the query node. Therefore, Q discovers T, which concludes the proof. □

Lemma 4.16 *A query set discovering a d-dimensional hypercube H_d is a vertex cover and any vertex cover verifies a d-dimensional hypercube H_d for $d \geq 4$. A minimum vertex cover discovers H_3. Therefore we need 2^{d-1} queries (the size of a minimum vertex cover in H_d) for $d \geq 3$.*

Proof. First we show that a query set Q that discovers the given hypercube H_d is a vertex cover. Let $\{u, v\}$ be an arbitrary edge. Recall that we can label the nodes of the hypercube by d-dimensional vectors such that there is an edge between two vertices if and only if their labels have Hamming distance 1. Now, suppose that neither u nor v is in Q. We show that no other query is a partial witness for the edge $\{u, v\}$. Let q be a query. W.l.o.g. u is closer to q than v is. Therefore, w.l.o.g., $u = 000\ldots0$ and $v = 100\ldots0$ and $q = q_1 q_2 \ldots q_d$, where $q_1 = 0$. There must exist an $i > 1$ such that $q_i = 1$. Then $w = \underbrace{10\ldots0}_{i-1}10\ldots0$ is a neighbour of v and is at the same distance from q as u, and therefore q cannot be a partial witness for the edge $\{u, v\}$. Thus, Q does not discover H_d.

Now we show that an arbitrary vertex cover discovers H_d when $d \geq 4$. Clearly, a vertex cover discovers all edges. We show that it discovers also all non-edges. Let $\{u, v\}$ be a non-edge in H_d. If u or v are in the vertex cover, the non-edge is discovered. We assume now that neither u nor v is in the vertex cover. W.l.o.g., $u = 00\ldots0$ and $v = \underbrace{1\ldots1}_{k}0\ldots0$, $k \geq 2$. If $k = d$, i.e., v is antipodal to u then $10\ldots0$ is a neighbour of u and therefore in the vertex cover. $10\ldots0$ has a distance $d - 1$ to v and distance 1 to u and since

$(d-1)-1 = d-2 \geq 2$ the query at this node discovers the non-edge $\{u, v\}$. If $k < d$ then vertex $0 \ldots 01$ (neighbour of u and therefore in the vertex cover) has distance $k+1$ to v and distance 1 to u and therefore the distance difference is $k \geq 2$ and therefore $\{u, v\}$ is discovered.

For $d = 3$, observe that $V \setminus \{000, 111\}$ is a vertex cover, but does not discover the non-edge $\{000, 111\}$. On the other hand this is not a minimum vertex cover for H_3 and therefore a minimum vertex cover for H_3 has to contain a vertex from every antipodal pair (and therefore discovers every such non-edge). To discover a non-edge $\{u, v\}$ of vertices at distance 2 from each other, i.e., w.l.o.g., $u = 000$ and $v = 110$, note that 111 has to be in the vertex cover if none of u, v is in it, and 111 discovers the non-edge $\{u, v\}$.

Finally, we note that the size of a minimum vertex cover for H_d, $d \geq 1$, is 2^{d-1}. To see this, observe that every vertex can cover at most d edges. The hypercube has $\frac{1}{2}2^d d$ edges and therefore a lower bound on the size of any vertex cover is 2^{d-1}. One can easily check that all vertices with even Hamming distance to the origin $00 \ldots 0$ form a vertex cover and the number of such vertices is 2^{d-1}.

□

Lemma 4.17 *A two dimensional grid G of size $m \times n$, $m \neq n$, or $m = n$ and $m \geq 4$, needs $\max\{m, n\}$ queries to be discovered.*

Proof. Let G be an $m \times n$-dimensional grid. We denote the vertex from i-th row and j-th column as (i, j), $1 \leq i \leq m$, $1 \leq j \leq n$. The edges are of the form $\{(i, j), (i, j+1)\}$ and $\{(i, j), (i+1, j)\}$.

We first show that every row contains a query from Q, if Q discovers G. This follows from the fact that only vertices from the i-th row $\{(i, j) \mid 1 \leq j \leq m\}$ are partial witnesses for the edges $\{(i, j), (i, j+1)\}$ of the i-th row: It is easy to see that a vertex (i, j) is a partial witness of any such edge. We now show that vertex $q = (k, l)$, $k \neq i$, is not a partial witness for any edge of the i-th row. Let us consider an edge $\{(i, j), (i, j+1)\}$ from the i-the row. W.l.o.g. we assume that $k > i$ and $l \leq j$. The case $k < i$ and $l \leq j$ is similar (just vertically upside down) and the other cases are symmetrical, just exchange the roles of (i, j) and $(i, j+1)$ in the following arguments. Let d denote the distance from q to (i, j). Then both (i, j) and $(i+1, j+1)$ are at the same layer L_d, they are

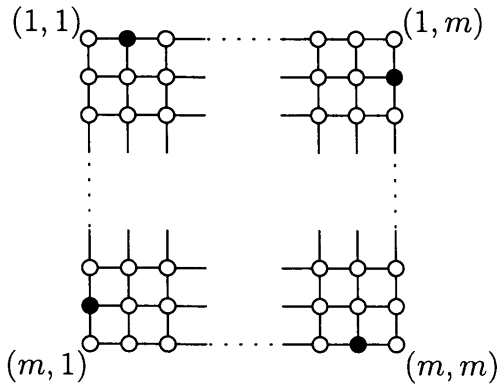


Figure 4.9: A grid $m \times m$ together with 4 queries $(1,2)$, $(m-1,1)$, $(m,m-1)$ and $(2,m)$ (shown as filled dots) that discover all non-edges.

both incident to $(i, j+1)$, and therefore q is not a partial witness for the edge $\{(i,j),(i,j+1)\}$.

For the same (symmetrical) reason we have that every column contains a query from Q. Hence, every row and column contains one query, which establishes a lower bound $\max\{m,n\}$ on the number of queries needed to discover the grid G.

We now show how to place $\max\{m,n\}$ queries in every row and every column such that every non-edge is discovered. In case $m < n$ (or $m > n$, this case is again symmetrical) we have to place in some row more than one query. Thus, we can place two queries at $(1,1)$ and $(1,n)$, which discover all non-edges (similarly as in the layered-graph query model) and then place for the remaining rows and columns the $m-2$ queries arbitrarily such that we have a query in every row and column. In case $m = n$, $m \geq 4$, we proceed as follows. We place the following 4 vertices into Q: $(1,2),(m-1,1),(m,m-1)$ and $(2,m)$. See Figure 4.9 for an illustration. Observe that no two queries lie in the same row or column. We claim that the four queries discover all non-edges. Observe first that ($\heartsuit$):

> a query (a,b) discovers all non-edges of the form $\{(a+k,b+l),(a+k+k',b+l+l')\}$ for any non-negative k,k',l,l'. Similarly, it discovers all non-edges $\{(a+k,b-l),((a+k+k',b-l-l')\}$, $\{(a-k,b+l),((a-k-k',b+l+l')\}$ and $\{(a-k,b-l),((a-k-k',b-l-l')\}$.

Let $\{u,v\}$ be a non-edge of the grid, where $u=(i,j)$ and $v=(k,l)$. We may assume $i \leq k$. Without loss of generality, $j \leq l$ (the case $j \geq l$ is symmetric—

rotate the grid counterclockwise by 90 degrees). Non-edge $\{u, v\}$ is not discovered by query $q_1 = (1, 2)$ according to the observation ($\heartsuit$) if u is from the first column, i.e., $u = (i, 1)$. The non-edge is also not discovered by query $(m, m-1)$ according to the observation ($\heartsuit$) if v is from the m-th column, i.e., $v = (k, m)$. Then $d(q_1, u) = 1 + i$ and $d(q_1, v) = m - 1 + k$. Hence, $d(q_1, v) - d(q_1, u) = m - 2 + (k - i) \geq 2$ for $m \geq 4$ and therefore q_1 discovers the non-edge $\{u, v\}$.

Hence we have shown that the four queries discover all non-edges. To discover the whole grid, we need a partial witness for every edge of the grid, i.e., we need to place a query into every row and column of the grid. We have done so in the first two and last two rows and columns already. For the remaining rows and columns, one can take the diagonal vertices as the query vertices.

We have shown that $\max\{m, n\}$ queries discover the grid, which matches the lower bound and is therefore optimum. □

Observe that the optimum number of queries for grids 2×2 and 3×3 is 3 and 4, respectively.

4.4.4 The Offline Problem

The $\mathcal{NP}$-Hardness of the Offline Problem

We consider the complexity of the DIST–ALL–VERIFICATION problem and show that it is $\mathcal{NP}$-hard. First we prove a useful lemma.

Lemma 4.18 *To discover a non-edge in a graph of diameter 2, one of its endpoints has to be a query.*

Proof. A non-edge $\{u, v\}$ is discovered by a query q, if the distances from q to u and v differ by at least 2. Since $diam = 2$, any node other than q is at distance 1 or 2 and therefore a query $q \notin \{u, v\}$ cannot discover the non-edge $\{u, v\}$. □

Theorem 4.19 *The problem* DIST–ALL–VERIFICATION *is $\mathcal{NP}$-hard.*

Proof. We present a polynomial-time reduction from the VERTEXCOVER problem to our problem. Let $G = (V, E)$ be a given graph for which a vertex cover is to be found. Let $n = |V|$. The basic idea is to create the complement $\bar{G}$

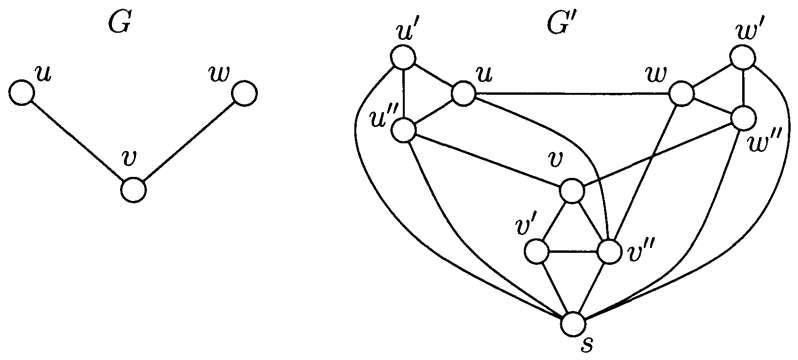


Figure 4.10: Instance of VERTEXCOVER (left), constructed instance of DIST–ALL–VERIFICATION (right).

of G and add a new node s to the graph and connect it to all other nodes. The resulting graph G' has diameter 2. According to Lemma 4.18 a query set Q verifying G' contains an endpoint of every non-edge. Thus, discovering the non-edges in G' corresponds to finding a vertex cover in G. To verify also the edges of G' we may need more queries, however, and the number of these additional queries may vary. Therefore we modify the construction of G' in order to force an additional fixed (or more precisely: tightly bounded) number of queries which discover all edges. Given an instance $G = (V, E)$ of VERTEXCOVER, we start by constructing $\bar{G}$ and then extend it as follows: For each node $v \in V$, we add two new nodes v' and v'' and the edges $\{v, v'\}$, $\{v, v''\}$ and $\{v', v''\}$. In addition, we connect v'' to all nodes $w \in V$ that are not adjacent to v in $\bar{G}$. Finally, we add an extra node s and make it adjacent to all nodes of type v' and v''. Call the resulting graph G'. An example of the construction is shown in Figure 4.10. Denote the set of all nodes of type v' by V', and the set of all nodes of type v'' by V''. We observe that G' has diameter 2. Furthermore, both V' and V'' are independent sets in G'.

Let $C \subseteq V$ be an optimal vertex cover for G. We claim that $Q_C = \{s\} \cup V' \cup V'' \cup C$ is a query set that verifies G'. First, note that Q_C contains partial witnesses for all edges of G'; in particular, the query at v' is a partial witness for all edges in G' that connect v to other nodes from V. Furthermore, Q_C verifies all non-edges. For non-edges incident to a node from $\{s\} \cup V' \cup V''$, this is obvious. For non-edges between nodes in V this follows because C, being a vertex cover in G, contains at least one endpoint of every edge in G, and

therefore at least one endpoint of every non-edge in G between nodes in V. Hence, there is a query set of size $2n + 1 + |C|$ that verifies G'.

Let Q be any query set that verifies G'. As V' and V'' are independent sets and G' has diameter 2, Q must contain at least $n-1$ nodes from V' and at least $n-1$ nodes from V'' by Lemma 4.18. Furthermore, the set $C' = Q \cap V$ must be a vertex cover of G, since it must contain an endpoint of every non-edge in G' between nodes in V. Hence, a query set Q that verifies G' yields a vertex cover of G of size at most $|Q| - 2n + 2$.

The discussion above shows that a polynomial-time algorithm computing an optimal query set for G' would give a vertex cover of size at most $(2n + 1 + |C|) - 2n + 2 = |C| + 3$. As VERTEXCOVER is $\mathcal{NP}$-hard to approximate within a factor of $7/6 - \varepsilon$ by [69], the problem DIST–ALL–VERIFICATION is $\mathcal{NP}$-hard. □

Approximation Algorithm

We present an $O(\log n)$-approximation algorithm for DIST–ALL–VERIFICATION that is based on the well-known greedy algorithm for the SETCOVER problem [62]. This technique was also used to derive the $O(\log n)$-approximation algorithm for the metric dimension and therefore for LG–ALL–VERIFICATION in [76].

Theorem 4.20 *There is an $O(\log n)$-approximation algorithm for* DIST–ALL–VERIFICATION.

Proof. We transform an instance $G = (V, E)$ of DIST–ALL–VERIFICATION into an instance of the set cover problem as follows. The edges and non-edges form the ground set $E \cup \bar{E}$ for the set cover problem. For each query $q \in V$, we introduce a subset $S_q = U_q \cup W_q$ of the ground set, formed by the set U_q of non-edges it verifies and the set W_q of edges for which it is a partial witness. By Observations 4.7 and 4.8, we can compute U_q and W_q. As a set of queries verifies G if and only if it discovers all non-edges and contains a partial witness for every edge, there is a direct correspondence between set covers and query sets that discover G. The standard greedy set cover approximation algorithm gives an approximation ratio of $O(\log |E \cup \bar{E}|) = O(\log \binom{n}{2}) = O(\log n)$. □

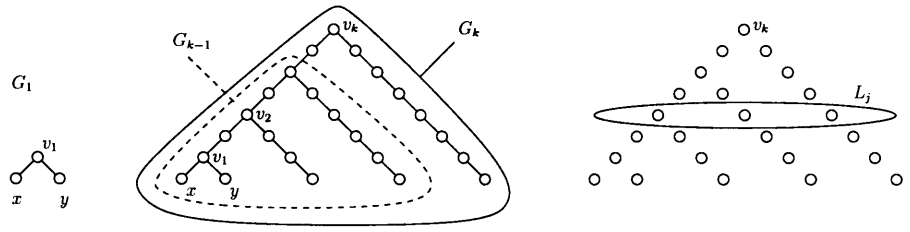


Figure 4.11: Graph used in the proof of the lower bound $\Omega(\sqrt{n})$ for on-line algorithms (left and middle); layers after query at vertex v_k (right).

4.4.5 The Online Problem

Lower Bounds for Online Algorithms

We present a lower bound of $\Omega(\sqrt{n})$ on the competitive ratio of any deterministic online algorithm for the problem DIST–ALL–DISCOVERY. Using the same ideas, we also obtain an $\Omega(\log n)$ lower bound on the competitive ratio of randomized online algorithms.

Theorem 4.21 *There is no $o(\sqrt{n})$-competitive deterministic online algorithm for* DIST–ALL–DISCOVERY.

Proof. Consider the graph G_k from Figure 4.11. It is a tree built recursively from a smaller tree G_{k-1} as depicted in the figure. Alternatively, G_k can be described as follows. Start with a chain of length $2k-1$ from x to v_k. For $1 \leq i \leq k$, the node on the chain at distance $2i-1$ from x is labelled as v_i. To each such node v_i, $1 \leq i \leq k$, we attach another chain (which we call *arm*) of length $2i-1$, starting at v_i. The number n_k of nodes of G_k satisfies $n_k = n_{k-1} + 1 + 2k$ for $k > 1$ and $n_1 = 3$. Hence, $n_k = k^2 + 2k$. G_k is a non-trivial tree and, by Lemma 4.15, the optimum number of queries is 2.

Now consider any deterministic algorithm A. As all vertices are indistinguishable to A, we may assume that the initial query q_0 made by A is at v_k. This sorts the vertices into layers according to their distance from v_k. There is no non-edge discovered within the layers. In particular, the non-edge $\{x, y\}$ in G_1 (see Figure 4.11) is not discovered. We now show that A needs at least k additional queries to discover $\{x, y\}$.

Observe that in the rightmost arm (attached to v_k) we have vertices from every layer. A picks a vertex from some layer j and, because all the vertices in

this layer are indistinguishable for A, we may force A to pick the vertex from the rightmost arm. Such a query in the rightmost arm does not reveal any new information within G_{k-1}. The vertices within one layer of G_{k-1} remain indistinguishable for A. Thus, when A places its first query in G_{k-1}, we can force it to be at a node from G_{k-1}'s rightmost arm. Clearly, we can continue recursively in this manner and therefore we can force A to query in every arm before it discovers $\{x, y\}$. This yields that A needs at least $1 + k$ queries to discover G_k.

Since $n_k = k^2 + 2k$, we have that $k = \Theta(\sqrt{n_k})$. Together with the fact that the optimum needs 2 queries, we get the desired lower bound. □

Theorem 4.22 *There is no $o(\log n)$-competitive randomized on-line algorithm for* DIST–ALL–DISCOVERY.

Proof. To show a lower bound on the competitive ratio of any randomized algorithm A against an oblivious adversary, we use Yao's principle [104]: The (worst case) expected number of queries of a randomized algorithm A (against all inputs) is at least the expected number of queries of the best deterministic algorithm for any input distribution. Thus, to show the lower bound for any randomized algorithm, we create a set of instances and a probability distribution and show that any deterministic algorithm performs badly on this input distribution in expectation.

The input set $\mathcal{G}_k$ is as follows. The graph is always isomorphic to G_k (as shown in Figure 4.11). Let layer L_i be the set of all nodes at distance i from v_k. The input distribution is constructed by permuting the labels (identities) of the nodes in each layer L_i, $1 \leq i \leq 2k - 1$, using a permutation chosen uniformly at random. Let A be any deterministic algorithm. Let E_k denote the expected number of queries made by A on an instance G_k from $\mathcal{G}_k$, assuming that a query at v_k (or at some node outside G_k, if the G_k is part of a larger tree) may have been made already but no other query inside G_k has been made. When the algorithm makes the first query q inside G_k, there are the following cases. If the query q is made at some v_i, at the parent of v_i, or at a node in the arm attached to the parent of the parent of v_i, then after the query there is still a G_i such that no query has been made in it (except possibly at its root v_i). In that case, we say that a G_i *remains*. The expected number of queries required to discover

G_i is then E_i. If the query q is made at one of the children of v_1, no G_i remains, and the algorithm may not require any additional queries. Letting p_i denote the probability that a G_i remains after the first query, we have $E_k \geq 1 + \sum_{i=1}^{k-1} p_i E_i$.

The algorithm makes the first query inside G_k at some layer j. Since the labels of the nodes of layer j have been permuted randomly, each of the nodes in layer j is equally likely to be the query node. For each layer, the probability that a G_i remains (possibly as part of a remaining $G_{i'}$ for $i' > i$) after a query in that layer is at least $\frac{1}{k+1}$ for each $i \in \{1, 2, \ldots, k-1\}$. (The minimum is achieved at the leaf layer.) Hence, we get

$$E_k \geq 1 + \sum_{i=1}^{k-1} \frac{1}{k+1} E_i$$

for $k \geq 2$ and $E_1 = 1$. This implies $E_k \geq H_{k+1} - \frac{1}{2} = \Theta(\log k)$, where $H_h = \sum_{i=1}^{h} \frac{1}{i}$ denotes the h-th harmonic number. Noting that the optimum is 2 and applying Yao's principle, we obtain the theorem (note that $k = \Theta(\sqrt{n_k})$, where n_k is the number of nodes in G_k, and thus $\log k = \Theta(\log n_k)$). □

Randomized Online Algorithm

In this section we present a randomized algorithm for DIST–ALL–DISCOVERY. The algorithm has competitive ratio $O(\sqrt{n \log n})$, which is very close to the lower bound $\Omega(\sqrt{n})$ for deterministic algorithms but leaves a gap to the lower bound $\Omega(\log n)$ for randomized algorithms.

The algorithm is a (non-straightforward) adaptation of the randomized algorithm for network discovery in the layered-graph query model presented in Section 4.3.2 and given in [19].

Theorem 4.23 *There is a randomized online algorithm with competitive ratio* $O(\sqrt{n \log n})$ *for* DIST–ALL–DISCOVERY.

Proof. The algorithm runs in two phases. In the first phase it makes $3\sqrt{n \ln n}$ queries at nodes chosen uniformly at random. In the second phase, as long as there is still an undiscovered pair $\{u, v\}$ (i.e., the queries executed so far have not discovered whether $\{u, v\}$ is an edge or non-edge), the algorithm executes the following. First, it queries both u and v. This discovers if $\{u, v\}$ is an

edge or non-edge. In case it is a non-edge, the algorithm then knows from the queries at u and v the set S of all queries that discover $\{u, v\}$: S is the set of vertices w for which $|d(u, w) - d(v, w)| \geq 2$. The algorithm then queries the whole set S. In case $\{u, v\}$ is an edge, the algorithm distinguishes three cases. First, if the queries at u and v discover a non-edge, say, $\{u, w\}$, that hadn't been discovered before, the algorithm proceeds with the pair $\{u, w\}$ instead of $\{u, v\}$ and handles it as described above. Second, if the number of neighbours of u and the number of neighbours of v is at most $\frac{\sqrt{n}}{\sqrt{\ln n}}$, then the algorithm queries also all neighbours of u and v (notice that after querying u and v we know all their neighbours). With this information we know the set S of vertices that are partial witnesses for $\{u, v\}$: a vertex w is in S if and only if the two vertices are at distances i and $i+1$ from w and all the other neighbours of the more distant vertex are at distances $i + 1$ or $i + 2$. Third, if the number of neighbours of u or the number of neighbours of v is more than $\frac{\sqrt{n}}{\sqrt{\ln n}}$, the algorithm does not do any further processing for this pair (i.e., this iteration of the second phase is completed) and proceeds with choosing another undiscovered pair $\{u', v'\}$ (if one exists).

The algorithm can be viewed as solving a HITTINGSET problem. For every non-edge $\{u, v\}$ let S_{uv} be the set of vertices that discover $\{u, v\}$. Similarly, for every edge $\{u, v\}$ let S_{uv} denote the set of all partial witnesses for $\{u, v\}$. The algorithm discovers the whole graph G if it hits all sets S_{uv}, for $\{u, v\} \in E \cup \bar{E}$. In the first phase, the algorithm aims to hit all the sets S_{uv} of size at least $\sqrt{n \ln n}$. Then, in the second phase, as long as there is an undiscovered pair $\{u, v\}$, the algorithm queries the whole set S_{uv}; if $\{u, v\}$ is an edge, it also queries all the neighbours of u and v in order to determine S_{uv}, except in the case where the degree of u or v is too large. In the case that the undiscovered pair $\{u, v\}$ is an edge for which a partial witness has already been queried before, the query at u or v must discover a new non-edge, and the algorithm uses that non-edge instead of $\{u, v\}$ to proceed.

We analyze the algorithm as follows. Let OPT be the optimal number of queries. Consider a pair $\{u, v\}$ for which the set S_{uv} has size at least $\sqrt{n \ln n}$. In each query of the first phase, the probability that S_{uv} is not hit is at most $1 - \frac{\sqrt{n \ln n}}{n} = 1 - \frac{\sqrt{\ln n}}{\sqrt{n}}$. Thus, the probability that S_{uv} is not hit throughout the

first phase is at most

$$\left(1-\frac{\sqrt{\ln n}}{\sqrt{n}}\right)^{3\sqrt{n\ln n}} = \left(\left(1-\frac{\sqrt{\ln n}}{\sqrt{n}}\right)^{\frac{\sqrt{n}}{\sqrt{\ln n}}}\right)^{3\ln n} \leq e^{-3\ln n} = \frac{1}{n^3}.$$

There are at most $\binom{n}{2}$ sets S_{uv} of cardinality at least $\sqrt{n\ln n}$. The probability that at least one of them is not hit in the first phase is at most $\binom{n}{2}\cdot\frac{1}{n^3} \leq \frac{1}{n}$.

Now consider the second phase, conditioned on the event that the first phase has indeed hit all sets S_{uv} of size at least $\sqrt{n\ln n}$. If the unknown pair $\{u,v\}$ is a non-edge, after querying u and v we know S_{uv}, and querying the whole set S_{uv} requires at most $\sqrt{n\ln n}$ queries (note that $|S_{uv}| \leq \sqrt{n\ln n}$ if $\{u,v\}$ is a non-edge that hasn't been discovered in the first phase). If the pair $\{u,v\}$ is an edge and the queries at u and v discover a new non-edge, the algorithm proceeds with that non-edge and makes at most $\sqrt{n\ln n}$ further queries (as above), hence at most $\sqrt{n\ln n}+1$ queries in total for this iteration of the second phase. Otherwise, if the number of neighbours of u and of v is bounded by $\frac{\sqrt{n}}{\sqrt{\ln n}}$, we query also all neighbours of u and v to determine the set S_{uv}, amounting to at most $2\frac{\sqrt{n}}{\sqrt{\ln n}}$ queries, and then the set S_{uv}, giving another $\sqrt{n\ln n}$ queries (since S_{uv} hasn't been hit in the first phase). In total, we make at most $\sqrt{n\ln n}+2\frac{\sqrt{n}}{\sqrt{\ln n}}$ queries in this iteration of the second phase. Consider the remaining case, i.e., the case where the unknown pair $\{u,v\}$ is an edge, no partial witness for the edge has been queried before, and u or v has degree larger than $\frac{\sqrt{n}}{\sqrt{\ln n}}$. Assume that there are k iterations of the second phase in which the unknown pair falls into this case. Note that no node can be part of an unknown pair in two such iterations. Hence, we get that $2|E| \geq k\frac{\sqrt{n}}{\sqrt{\ln n}}$ and, by Lemma 4.11, $OPT \geq \frac{|E|}{n} \geq \frac{k\sqrt{n}}{2n\sqrt{\ln n}} = \frac{k}{2\sqrt{n\ln n}}$ and therefore $k \leq 2\sqrt{n\ln n}\cdot OPT$.

Now, let ℓ denote the number of iterations of the second phase in which the set S_{uv} was determined and queried (i.e., all iterations except the k iterations discussed above). We call such iterations *good* iterations. The overall cost of the second phase is at most $\ell\sqrt{n\ln n}+2\ell\frac{\sqrt{n}}{\sqrt{\ln n}}+2k$. Clearly, $OPT \geq \ell$, because no two unknown pairs $\{u,v\}$ considered in different good iterations of the second phase can be discovered by the same query (or have the same partial witness). Therefore the cost of the algorithm is at most $3\sqrt{n\ln n}+\ell\sqrt{n\ln n}+2\ell\frac{\sqrt{n}}{\sqrt{\ln n}}+2k =$

$O(\sqrt{n \log n}) \cdot OPT$.

So we have that with probability at least $1 - \frac{1}{n}$, the first phase succeeds and $O(\sqrt{n \log n}) \cdot OPT$ queries are made by the algorithm. If the first phase fails, the algorithm makes at most n queries (clearly, the algorithm need not repeat any query). This case increases the expected number of queries made by the algorithm by at most $\frac{1}{n}n = 1$. Thus, we have that the expected number of queries is at most $O(\sqrt{n \log n}) \cdot OPT + \frac{1}{n}n = O(\sqrt{n \log n}) \cdot OPT$. □

4.5 Summary of Results and Open Problems

We have introduced a model for the problem of mapping large-scale, dynamic networks—network discovery, where a network (modeled as a connected, undirected graph) can be discovered by queries at nodes. We have presented two query models. In the layered-graph query model a query at node v returns all shortest paths from v to all other nodes. In the distance query model a query at node v returns distances to all other nodes. We have studied the complexity of discovering a network with minimum number of queries and presented optimal algorithms for specific classes of graphs, lower bounds on randomized online algorithms, and randomized online algorithms. Our randomized online algorithms have competitive ratio $O(\sqrt{n \log n})$ in both query models.

There are quite a few interesting open problems in directions for future work in the framework of network discovery:

- Is there a deterministic online algorithm with an approximation ratio similar to the competitive ratio $O(\sqrt{n \log n})$ of the online randomized algorithms presented in this thesis? How good can we approximate by a deterministic algorithm?

- Another interesting modification is to consider different query models. For example, a query specified by two nodes u and v returns all shortest paths between u and v; or a query at node v returns the distances to all other nodes that are within distance at most k from v.

- Changing the objective of the problem leads to other intersting variants. For example, one could ask for the minimum number of queries that are

required to determine the diameter or the value of some other graph parameter of the network.

Chapter 5

Assignment of OVSF-Codes

The area of mobile communication has flourished in the past years in many aspects. The high commercial interest demands new services and better solutions (whatever this means) than the existing ones. This in turn offers plenty of new research problems to the research community. Currently, the so-called third generation (3G) mobile telecommunication systems (of which Universal Mobile Telecommunications System, UMTS, is an example) are being introduced (more information on the topic can be found for example in [71], [99]). These offer a relatively large amount of data that can be communicated at a time. Sharing this bandwidth in a communication cell (served by a base station) among mobile users is one of the well studied problems. Code Division Multiple Access (CDMA) allows the users to communicate at any time and to use the whole frequency spectrum. To identify connections upon receiving users' signals, communication codes are used. More specifically, these codes are used to spread the data before transmission (creating a new data stream that allows the receiver to distinguish the data from other users' data). Orthogonal Variable Spreading Factor Codes (OVSF-codes) are used in W-CDMA (Wideband CDMA, a CDMA technology used in UMTS). The principle is that each bit of information that is to be transmitted is encoded as a binary word w. The length of w can vary, as it depends on the bandwidth that is assigned to the user. The shorter the code is, the more bandwidth the code gains for the user. OVSF-codes can be viewed as nodes of a complete binary tree, called a *code-tree*. The sharing of

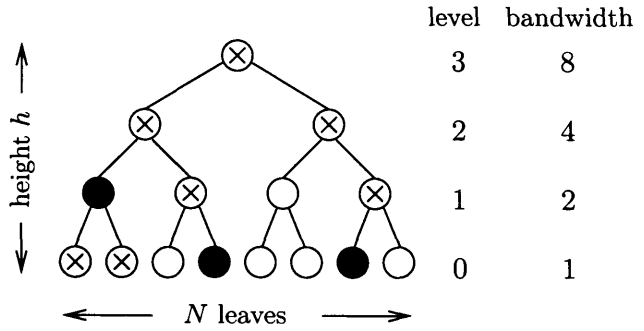


Figure 5.1: OVSF-codes represented as nodes of a complete binary tree T. The filled nodes represent codes assigned to users. The nodes with a cross represent blocked codes. The blank nodes are available codes for other users to be assigned.

the given bandwidth is accomplished by assigning the users different codes from the code-tree. The code-word w associated with a node is created in a recursive fashion as follows. The root is assigned code-word (1). For the children of a node with code-word (a): the left child carries code-word (a, a) and the right child carries code-word $(a, -a)$. The generation of OVSF-codes is described in more details in [3]. The crucial property for the receiver to distinguish the signals of two different users is that the users use codes that are mutually orthogonal. Two codes are mutually orthogonal if and only if they do not have any descendant/ancestor relationship in the code-tree. In other words, no two codes lie on the same leaf-to-root path. An assigned code therefore prevents some other codes from being assigned to another user. Theses codes are called *blocked codes*. The codes are arranged in levels, numbered from 0 to h, where level-0 codes are the leaves and the root is the level-h code. Figure 5.1 depicts an example of an OVSF-code tree, with some codes assigned to users.

Users ask for codes from a certain level according to their bandwidth preferences/needs. Over time, users enter and leave the cell, i.e., a code is assigned and released on these occasions. This can lead to a situation that a new user enters the cell, asking for a code from a certain level, but the system cannot serve the user, although the unused bandwidth is sufficient for the user's demand. Figure 5.2 gives an example of such a situation (only the bottom 3 levels of the code-tree are depicted): a new user requests a code from level 2, but all codes from level 2 are blocked or assigned. Reassigning of the codes (i.e., a user gets

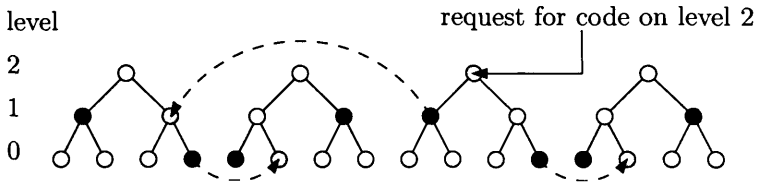


Figure 5.2: The assigned codes block all codes from level 2. A reassignment (depicted by the dashed arrows) of the already assigned codes results in a level-2 code to be available for the new user.

assigned a different code from the same level), as depicted by the dashed arrows, allows the entering user to obtain a code from level 2 (as the solid arrow shows). This problem is known as *code blocking* or *code-tree fragmentation.* Although the reassignment increases the throughput of the system, its necessary communication with the affected users causes signalling overhead, unwanted delay and users' disturbance ([80], [63]). Therefore, a natural objective stated in [80], [89] is to serve all users, if the bandwidth is sufficient, and to minimize the number of code reassignments. Since then, several authors considered the problem, suggesting several heuristics and simulations to tackle the problem, adjusting them to different statistical models capturing the process of users' entering and leaving the cell. We aim to give an algorithmic-theory insight into the problem in terms of computational complexity, online and approximation algorithms.

5.1 Formal Problem Definition

A complete binary tree $T = (V, E)$ of height h with $N = 2^h$ leaves represents the code-tree of the OVSF-codes. The codes are the nodes of the tree. For a code c from T, the subtree rooted at c is denoted by T_c. The codes are organized in levels, always codes of the same depth forming one level. The levels are counted from the leaves to the root starting at level 0. We denote by $l(v)$ the level of node (code) v. A code from level i supports bandwidth of $2^i B_L$, where B_L is the bandwidth of the leaf codes. For ease of presentation, we assume that $B_L = 1$. The supported bandwidth of a tree is also called a *capacity.* We assume that users enter the cell of a base station one at a time. Each user requests a certain bandwidth, i.e., a code from a certain level of T. All users that are allowed in the system have their requests *assigned* to codes in the tree

(and the codes are *assigned* to the requests) of the desired level, such that on every path p_j from a leaf j to the root there is at most one assigned code (i.e., no assigned code is in a subtree of another assigned code). If we denote the number of requests for a code from level l by r_l, then all requests of the users are assigned to codes from T such that for every level l there are exactly r_l codes assigned to the requests and no assigned code is in a subtree of another assigned code. We call every set of positions $F \subset V$ in the tree that fulfils these properties a *(feasible) code assignment* for a *request vector* $r = (r_0, r_1, \ldots, r_h)$. From now on, when we talk about a code tree, we always mean T together with F. The number of assigned codes is denoted by n (i.e., $n = |F|$). A maximal subtree of unassigned codes is called a *gap tree.* A request of a new user is dropped if it cannot be served. This is the case, if its acceptance would cause the users' requested bandwidth to exceed the total system bandwidth 2^h. After accepting the new request for a code on level l, a *code assignment algorithm* has to change the (old) code assignment F (for r) to a new code assignment F' for the new request vector $r' = (r_0, \ldots, r_l + 1, \ldots, r_h)$. Similarly, when a user leaves the system, the algorithm has to produce a code assignment F' for the request vector $r' = (r_0, \ldots, r_l - 1, \ldots, r_h)$ (in this case, it is straightforward to find F'–just drop the code from F that was assigned to the leaving user). The size $|F' \setminus F|$ corresponds to the number of *reassignments.* This implies that for a new code request, the new code assignment is counted as a reassignment. We define the number of reassignments as the cost function. Hence, if a code is released by a departing user, this incurs zero cost (and can be viewed as charging the departures upon arrivals).

To stress the combinatorial side of the problem, we call a reassignment a *movement* of a code and the nodes of T *positions.* This can be viewed as having pebbles on the positions of assigned codes (i.e., F are the pebbles) and moving the pebbles from F that are not present in F' onto their new positions in F'. Also, when a new request arrives, this can be viewed as inserting a new pebble into the tree. Thus we talk about a *code insertion.* Similarly, if a user leaves the system, we talk about a *code deletion.*

We state the code assignment problem (CA), as it was stated in [80], together with some of its natural variants:

one-step offline CA Given a code assignment F for a request vector r and a code request for level l, find a code assignment F' for the new request vector $r' = (r_0, \ldots, r_l+1, \ldots, r_h)$ with minimum number of reassignments.

general offline CA Given a sequence S (of length m) of code insertions and deletions, resulting into request vectors $r^1, \ldots, r^m$, find a sequence of code assignments F_i for r^i so that the total number of reassignments is minimum, assuming the initial code tree is empty (i.e., no code assigned).

online CA For a sequence S of code insertions and deletions (of unknown length), whose items appear to the algorithm one by one (in an on-line fashion), find a code assignment F_i for every code insertion/deletion r^i upon its arrival so that the total number of reassignments is minimum, assuming there is no code assigned in the beginning.

insertion-only online CA This is the online CA problem with S consisting of insertions only.

5.2 Related Work and New Contributions

Soon after W-CDMA was chosen to be the air interface for UMTS [38], awareness of a need to have some policy that assigns the codes in an effective way has arisen. Fantacci and Nannicini identify the importance of optimizing the resource allocation, i.e., having a "good" code assignment [56]. They propose a code allocation similar to our compact representation algorithm from Section 5.5.2 and use it in their simulation-based evaluation of their medium access protocol. Only three types of requests for bandwidth were allowed—voice, video and data. Here, dropping of assigned codes, or lowering the assigned bandwidth was part of their protocol. In the same journal Minn and Siu proposed a dynamic code assignment (DCA) scheme to eliminate code blocking and to minimize the number of code reassignments to support a new user to enter the system [80]. While their algorithm DCA will be shown to be erroneous in Section 5.4.1, it is this paper that explicitly defines the problem of minimizing the number of code reassignments—the one-step offline CA problem. Many follow-up papers considered the problem to be solved by Minn and Siu and concentrated

on evaluation of existing OVSF-code assignment schemes [11] or different optimization goals (such as maximizing the average level of codes assigned to users [10]), or proposed new schemes aiming to be simpler and quicker [29], or used additional mechanisms (like time multiplexing or code sharing) on top of the original problem setting in order to mitigate the code blocking problem [26, 89].

The question of which code should a new user be assigned (arising in on-line problems with different optimization goals, e.g., blocking probability, Quality of Service (QoS), number of reassignments, etc.) is addressed at the same time in [30, 12, 103]. In [30] codes are placed on the left-hand side of the tree and the larger codes on the right-hand side of the code tree. The scheme from [12] divides the code tree into several regions. Each region is reserved to support one specific data rate. The partitioning strategy is based on the users' request probabilities. A *compact index* is defined in [103] for each code and the code with the smallest compact index (among the candidates) is assigned to the user. This approach is similar to a *crowded-first* strategy introduced in [98], which computes for each code the bandwidth that is occupied in a subtree rooted at the code's parent. The code with the highest computed value is assigned to the user. The idea is to keep large areas in the code tree without any assigned code. In [89] similar ideas are used and the authors propose to decide according to a different value computed for each candidate code—the total number of codes assigned in the subtree rooted at the code's parent. Dell'Amico at al. [44] present a dynamic tree partitioning technique and evaluate it in simulations with respect to blocking probability and number of reassignments over a sequence of call arrivals and departures. Kam, Minn and Siu [73] address the problem in the context of bursty traffic and differnet QoS. They come up with a notion of "fairness" and also propose to use multiplexing. Similar in perspective is [60]. Chen and Chen [28] propose a best-fit least-recently used approach. We add that no algorithm-theoretic analysis is given in these papers.

Inspired by the above work, we studied the problems defined in Section 5.1 and present them in this chapter. A few important observations about the code assignment are presented in Section 5.3. The one-step offline CA is studied in Section 5.4. There we begin with a counter-example of the claim that the DCA-algorithm proposed by Minn and Siu is optimal [80] (Section 5.4.1). We outline

the proof of the $\mathcal{NP}$-completeness in Section 5.4.2. The full proof appeared in the thesis of Marc Nunkesser [83]. We then outline the key ideas of an optimum algorithm that runs in time $O(2^h n^h)$ time. The full details of the algorithm can be found in the thesis of Gábor Szabó [96]. Based on these ideas we show that the problem is fixed parameter tractable for various parameters (Section 5.4.3). In Section 5.4.4 we present an h-approximation algorithm. Section 5.5 deals with the online CA problem. We start by showing that no deterministic online algorithm can be better than 1.5-competitive. We show that a natural greedy algorithm that in every step minimizes the number of reassigned codes is $\Omega(h)$-competitive (Section 5.5.1). In Sections 5.5.2 and 5.5.3 we analyze two algorithms that were proposed in the literature before. The first one, called compact algorithm, is shown to be $\Theta(h)$-competitive and the second one is shown to be optimum for the insertion-only online CA problem. The main contribution of the author of this thesis is the h-approximation algorithm for the one-step offline CA, fixed parameter tractability of the one-step offline CA (together with Thomas Erlebach and Riko Jacob) and the analysis and discussion of the online algorithm that keeps the number of blocked codes on minimum at all times (joint work with Gábor Szabó).

After we have published our results, Tomamichel showed that the general offline CA problem is $\mathcal{NP}$-hard [97]. Recently, Královič et al. announced an optimum online algorithm with amortized constant number of reassignments per request [59].

5.3 Folklore

5.3.1 Call Admission Feasibility

When a new user arrives, asking for a code from level l, the system has to decide, if the user can be assigned the requested code. Thus, if F is the code assignment to the current request vector $r = (r_0, \ldots, r_h)$ of the code tree T, the task is to decide whether there exists a code assignment F' to the request vector $r' = (r_0, \ldots, r_l + 1, \ldots, r_h)$.

Consider a code assignment F of n codes in a code tree T of height h. Every assigned code on level l has its unique path from the root to the node of

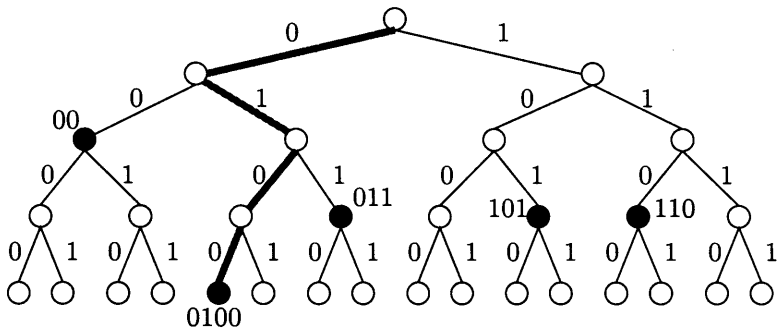


Figure 5.3: Correspondence of code assignments in tree of height 4 with codes on levels {0,1,1,1,2} (assigned codes are the filled nodes) and prefix free codes of lengths {4,3,3,3,2}.

length $h - l$. The path can be encoded by a word $w \in \{0,1\}^{h-l}$ determining whether we traverse through the left or right child. Here, 0 stands for the left child and 1 stands for the right child. From the properties of code assignments the path/node identifiers form a binary prefix-free code-set (a set of words in a given alphabet form a prefix-free code-set if none of the words is a prefix of another word). On the other hand, given a prefix-free code-set of lengths $\{h-l_1, \ldots, h-l_n\}$ we can assign a code on level l_i by following the path described by the code word (see Figure 5.3) and form a feasible code assignment. Thus, we have shown that a code assignment for codes on levels $l_1, \ldots, l_n$ exists if and only if there exists a binary prefix-free code-set of given lengths $\{h - l_1, \ldots, h - l_n\}$. The latter is a well known result in coding theory due to the Kraft-McMillan (inequality) theorem, the proof of which can be found for example in [4].

Theorem 5.1 (Kraft-McMillan inequality) *A binary prefix-free code-set of code lengths $a_1, \ldots, a_m$ exists if and only if*

$$\sum_{i=1}^{m} 2^{-a_i} \leq 1. \tag{5.1}$$

We can multiply the Equation (5.1) by 2^h to see the immediate relationship of OVSF-code assignment and binary prefix-free codes, expressed in the following corollary.

Corollary 5.2 *A code assignment for levels $l_1, \ldots, l_m$ into the code tree T of*

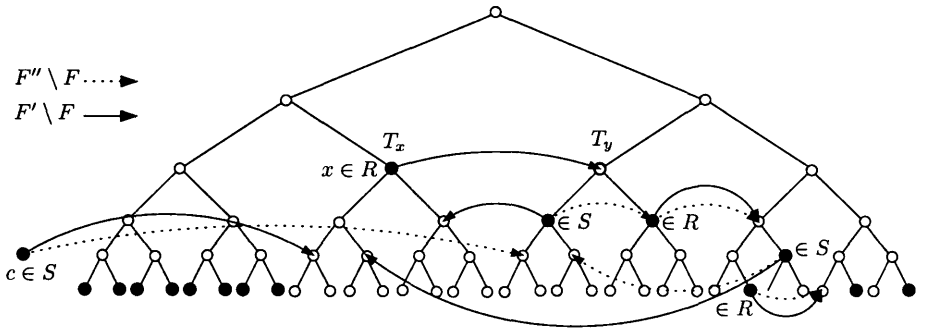


Figure 5.4: Non-optimality of a code assignment F' that reassigns codes also on higher levels than the requested level.

height h with N leaves exists if and only if

$$\sum_{i=1}^{m} 2^{l_i} \leq N.$$

Hence, checking whether one can serve the code insertion can be decided in linear time. Therefore from now on we assume, without loss of generality, that the sequence of code insertions and deletions can always fit within the tree capacity, i.e., there exists a code reassignment to insert the code.

5.3.2 Irrelevance of Higher Level Codes

In this section we show that an optimal algorithm for the one-step offline CA problem moves only codes on levels lower than the requested level l. A similar result was already given in [80], but here we give an independent and slightly different statement.

Lemma 5.3 *Let c denote a code insertion on level l into a code tree T. Then for every code reassignment F' that inserts c and that moves a code on level $k \geq l$ there exists a code reassignment F'' that inserts c and moves fewer codes, i.e., with $|F'' \setminus F| < |F' \setminus F|$.*

Proof. Let $x \in F$ be the highest code that is reassigned by F' on a level at or above the level l and let S denote the set of codes moved by F' into the subtree T_x rooted at node x. We denote by R the rest of the codes that are moved by F' (see Figure 5.4). The cost of F' is $|S| + |R|$. The code reassignment F'' is

defined as follows: Let y be the position where F' moves the code x. Then F'' moves the codes in S into the subtree T_y rooted at y, leaves the code x at the root of T_x, and moves the rest of the codes R in the same way as F'. The cost of F'' is at least one less than the cost of F' since it does not move the code x. In the example from Figure 5.4 the cost of F' is 6 and the cost of F'' is 5. □

5.3.3 Arbitrary Code Tree Configuration

For discussion about quality of algorithms for the general offline CA problem or for the online CA problem, a natural question is, how restricted the fragmentation of a code tree can be. In this section we show that any algorithm that

(a) upon a code deletion does not perform any code reassignments and

(b) upon a code insertion minimizes the number of code reassignments necessary to accommodate the new code,

can be forced (by a properly chosen sequence of code insertions and deletions) to produce an arbitrary code assignment. We remark that any optimal algorithm for the one-step CA problem behaves in exactly this way (i.e., satisfies (a) and (b)).

Let A be an algorithm following the rules (a) and (b). Let F_e be an arbitrary code assignment of a code tree T. We show how to construct a sequence of code insertions and deletions such that A ends up in F_e on that sequence. The idea of the proof is to take a detour and first attain a full-capacity code assignment F_f such that $F_e \subseteq F_f$ and then go from there to F_e. The second step is easy: It suffices to delete all the codes from F_f that are not in F_e (A must not do any reassignments during these deletions). First, we show that we can force A to produce an arbitrary chosen code assignment F_f that uses the full tree capacity.

Lemma 5.4 *Any algorithm A behaving according to (a) and (b) can be led to an arbitrary code assignment F_f (of n codes) that uses the total tree capacity (i.e., every code of the tree is blocked) by a sequence of insertions and deletions of length $m < 3n$.*

Proof. Recall that h is the height of the tree and that in the beginning there is no assigned code (technically this is not a problem, since we can make the

tree empty by deleting all the assigned codes in the beginning). We proceed level by level, starting at the top: On every level l with codes in F_f we ask for code insertions on every unblocked position on level l (of the current code assignment). A fills all unblocked codes on level l with codes. Next we delete all codes on l that are not in F_f and proceed recursively on the next level. It is clear that the created sequence of code insertions and deletions forces A to end up with code assignment F_f.

Now we have to argue about the size of the constructed sequence. Observe that we are only inserting and deleting codes above the n codes in F_f, and every node is considered for insertion or deletion at most once. Consider the binary tree for which the leaves are exactly the codes in F_f. From the properties of binary trees, the number of inner nodes of this tree is $n-1$. Hence, the number of insertions is bounded by $n+n-1$ and the number of deletions is bounded by $n-1$, which completes the proof. □

Corollary 5.5 *For any algorithm A that satisfies (a) and (b) and for any code assignment F_e of n codes into a code tree T of height h, there exists a sequence of code insertions and deletions of length $m < 4nh$ that forces A to end up with code assignment F_e.*

Proof. We define F_f from F_e by adding to F_e the roots of all gap trees of the code assignment F_e. Each code from F_e causes at most one gap tree on every level (observe that each code is inserted into some gap tree and that new gap trees can appear only within the former gap tree), hence, for every code in F_e we need to add at most h codes (the roots of the gap trees) to F_f. Altogether we have at most $n(h+1)$ codes in F_f—n codes from F_e and the at most nh codes that "fill" all gap trees. According to Lemma 5.4, we can construct a sequence of length $m < 3n(h+1)$ that forces A into F_f. By asking for code deletions of the filling codes (the roots of the gap trees of F_e) the algorithm ends up in F_e. Clearly, we need at most $4n(h+1)$ requests of code insertions and deletions in total. □

5.4 One-Step Offline CA

5.4.1 Non-Optimality of Greedy Algorithms

We consider possible greedy algorithms for the one-step offline CA problem. Recall that we want to minimize the number of code reassignments while accommodating the code insertion on level l and that according to Lemma 5.3 we do not want to move codes on higher level than l.

For a code insertion on level l, a straight-forward greedy approach is to place the code into the root of a subtree with minimum cost that is not blocked by a code above the requested level, according to some cost function. All codes in the selected subtree must then be reassigned. So in every step a *top-down greedy* algorithm (for the one-step offline CA problem) chooses the maximum bandwidth code that has to be reassigned, places it at the root of a minimum cost subtree (that is not blocked by an above assigned code), takes out the codes in that subtree and proceeds recursively. The DCA-algorithm in [80] works in this way. The authors propose different cost functions, among which the "topology search" cost function is claimed to solve the one-step offline CA optimally. As the cost function depends only on the current code assignment of the considered subtree, the following theorem implies that the DCA-algorithm is not optimal.

Theorem 5.6 *Any top-down greedy algorithm A_{tdg}, whose cost function depends only on the current code assignment of the considered subtree is not optimal.*

Proof. We construct a code assignment F of a code tree T (and specify a level l for the code insertion), for which A_{tdg} is not optimum. More specifically, we make A_{tdg} assign the new request on level l to the root of a special subtree T_0, see Figure 5.5, which illustrates the construction of F. The tree T_0 has an assigned code c_k of bandwidth 2^k on level $k < l$ and, depending on the cost function of A_{tdg}, it can have an assigned code c_{k-1} of bandwidth 2^{k-1} on level $k-1$. The code(s) from T_0 then has to be reassigned. We construct trees T_1, T_2 and T_3 (with roots on level k) to be considered as place to reassign the code(s) from T_0. The subtree T_1 has $2^{k-1}-1$ consecutive leaf codes assigned to users

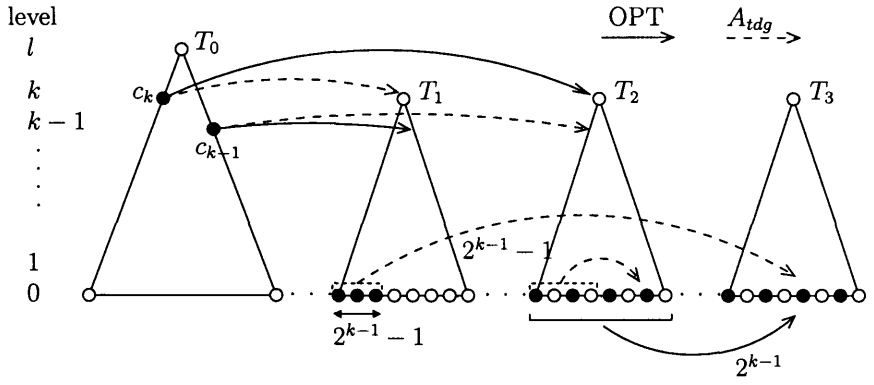


Figure 5.5: The construction of a code tree for which any top-down greedy algorithm is not optimum.

(and no other assigned codes elsewhere in the subtree). The subtrees T_2 and T_3 contain 2^{k-1} assigned codes on the leaf level, always an unassigned code following the assigned one. All other subtrees, in particular the sibling trees of T_1, T_2 and T_3 (omitted from the figure) have all their leaves assigned. With such a code assignment any optimal code reassignment that accommodates the new code on level l has to insert the new code to the root of T_0 and reassigns the codes from T_0 into the free places of T_1, T_2 and T_3 (we choose l to be sufficiently higher than k, so that all other subtrees of level l have a high number of assigned leaf codes).

Hence, A_{tdg} assigns the new code to T_0 and has to decide about where to reassign the code c_k:

case 1: The cost function of A_{tdg} evaluates T_2 and T_3 as cheaper than T_1. In this case we let the subtree T_0 contain only the code c_k. Algorithm A_{tdg} moves c_k to the root of the subtree T_2 or T_3, which causes one more reassignment than assigning it to the root of T_1, hence the algorithm fails to produce the optimal solution.

case 2: The cost function evaluates T_1 as cheaper than T_2 and T_3. In this case we let the subtree T_0 have both codes c_k and c_{k-1}. A_{tdg} moves c_k to the root of T_1 and c_{k-1} to the a child of the root of T_2 or T_3, see the dashed lines in Figure 5.5. The number of reassigned codes by the algorithm is $\frac{3}{2} \cdot 2^{k-1} + 2$ (remember that we count the initial code insertion as a reassignment). The minimum number of reassignments is $2^{k-1} + 3$, which

is achieved when the code c_{k-1} is moved into the empty part of T_1 and the code c_k is moved to the root of T_2 or T_3, see the solid lines in Figure 5.5. Therefore, also in this case A_{tdg} is not optimum.

□

5.4.2 Complexity of One-Step Offline CA

In this section we study the complexity of the decision variant of the one-step offline CA. It turns out that for a natural input decoding, the problem is $\mathcal{NP}$-complete. The full proof and discussion on this topic appeared in the thesis of Marc Nunkesser [83], we will only outline the main idea.

The decision variant of the one-step offline CA is to decide whether a new code insertion can be handled with cost less or equal to a number $s_{\max}$, which is also part of the input. Clearly, the decision variant is in $\mathcal{NP}$, because we can guess an optimal assignment and verify in polynomial time if it is feasible and if its cost is at most $s_{\max}$. To complete the proof of $\mathcal{NP}$-completeness, we reduce the three-dimensional matching problem (3DM, [62]) to the one-step offline CA problem.

Problem 1 *(3DM) Given a set $M \subseteq W \times X \times Y$, where W, X and Y are disjoint sets having the same number q of elements. Does M contain a perfect matching, i.e., a subset $M' \subseteq M$ such that $|M'| = q$ and no two elements of M' agree in any coordinate?*

For our purposes, we assume that, without loss of generality, $W = X = Y = \{1, 2, \ldots, q\}$. We define the *indicator vector* of a triplet $(i, j, k) \in W \times X \times Y$ as a zero-one vector of length $3q$ that is all zero except at the indices $i, q + j$ and $2q + k$. Hence, every element of $W \times X \times Y$ can be viewed as an indicator vector. The 3DM problem is now equivalent to finding a subset of q indicator vectors out of the indicator vectors in M that sum up to the all-one vector.

Figure 5.6 shows an outline of the construction that we use for the reduction: an input to 3DM is transformed into an initial feasible assignment that consists of a *token tree* in the left half of T and different smaller trees in the right half of T. A code insertion request is given at the level indicated in the figure. The construction is set up in such a way that the code must be assigned to the root

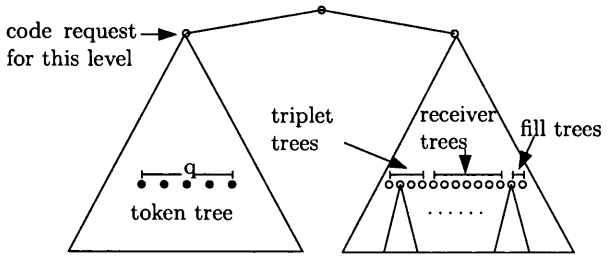
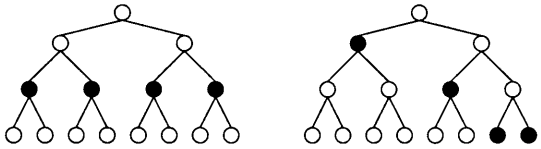


Figure 5.6: Sketch of the construction.



(a) The zero-tree. (b) The one-tree.

Figure 5.7: Encoding of zero and one.

of the left tree, the token tree, in order to minimize the number of reassignments (we can assign a lot of leaf codes in the right half of T to achieve that; the higher the tree T, the more leaf codes we have). There are q codes in the token tree, all on the same level. These codes have to be moved to the right half of T. By the construction, they are forced to be reassigned to the roots of *triplet trees* (again, we can use the leaf codes to achieve that). Each of the triplet trees represents one of the elements of M, for which we are asking the existence of a 3D matching. We want to design the tree T, such that all codes in the chosen triplet trees (by the reassignment of the q codes from the token tree) find a place without any additional reassignment if and only if these triplets represent a 3D matching.

The construction of the token tree is straightforward, we place q codes positioned arbitrarily on level l_{start} with sufficient depth (the discussion of what exactly is a sufficient depth can be found in [83, 52]). The triplet trees have their roots on the same level l_{start}. They are constructed from the indicator vectors of the triplets. We represent each of the $3q$ elements of the vector by four levels in the triplet tree. We call these four levels a *layer*. Each layer encodes either zero or one, where the encoding of zero and one are shown in Figure 5.7 (a) and (b). We have chosen the zero-trees and one-trees such that both have the same number of codes and occupy the same bandwidth, but are still differ-

ent. The *receiver trees* are constructed to receive all codes in the chosen triplet trees. By the property of 3DM, we want the union of all receiver trees to accommodate on every layer exactly one layer that encodes 1 and $q-1$ layers that encode 0 without additional reassignments. Finally, the *fill trees* are trees that are completely full (i.e., the assigned codes occupy the whole tree-bandwidth) and have one more code than the receiver trees. They are used to support the construction of the receiver trees to work in the way we want.

An interesting question is, whether this transformation from 3DM to the one-step offline CA can be done in polynomial time. This depends on the input encoding of our problem. We consider the following natural encodings:

- a zero-one vector that specifies for every node of the tree whether there is a code or not, and
- a sparse representation of the tree, consisting only of the positions of the assigned codes.

Obviously, the transformation cannot be done in polynomial time for the first input encoding, because the generated tree has roughly $2^{\Omega(q)}$ leaves. For the second input encoding the transformation is polynomial, because the total number of generated codes is polynomial in q, which is polynomial in the input size of 3DM. Besides, we should rather not expect an $\mathcal{NP}$-completeness proof for the first input encoding, because this would suggest, together with the dynamic programming algorithm (that finds an optimum reassignment and runs in $n^{O(\log n)}$ time) from Section 5.4.3, $n^{O(\log n)}$-time algorithms for all problems in $\mathcal{NP}$.

By the above construction we can prove the following theorem (the proof of which has appeared in the thesis of Marc Nunkesser [83]).

Theorem 5.7 *The decision variant of the one-step offline CA is $\mathcal{NP}$-complete for an input given by a list of positions of the assigned codes and the code insertion level.*

5.4.3 An Optimal Algorithm and Fixed Parameter Tractability

In this section we outline an optimal algorithm for the one-step offline CA based on dynamic programming. The details appeared in the thesis of Gábor Szabó [96]. Building on the techniques of the optimum algorithm we show that the one-step offline CA is fixed parameter tractable for various (natural) parameters.

Let F be a code assignment for a request vector r. Let l be the level of the new code request. We want to find a code assignment F' for the request vector $r' = (r_0, r_1, \ldots, r_l + 1, \ldots, r_h)$ that minimizes $|F' \setminus F|$, the number of code reassignments. The key idea of the optimal algorithm is to decompose the computation of optimum F' for r' into two smaller problems: Split r' into all possible pairs of vectors r'_L, r'_R such that $r' = r'_L + r'_R$; consider r'_L (r'_R) as a request vector for the left (right) subtree of T and compute the optimum code assignment F'_L (F'_R) for r'_L (r'_R); output as F that pair-union $F'_L \cup F'_R$, which has the minimum code reassignments. Observe that if $r' = (\ldots, 1)$, i.e., there is a code assigned at the root, there is nothing (difficult) to compute.

A dynamic programming for such a recursive computation can be realized as follows. We store at every node v all possible (i.e, all request vectors that are feasible, and achievable from the current code assignment F) request vectors $r^v_1, \ldots, r^v_{m_v}$ together with their *cost*. We define the cost of a request vector r^v_i to be the minimum number of assigned codes that have to move out of the subtree T_v to obtain a code assignment for the request vector r^v_i (we note that there might be some codes moving into T_v as well, but these are not considered in our cost function). Because only levels $0, 1, \ldots, l(v)$ are affected by the subtree T_v, the request vectors considered at v are all of the form $(r^v_0, r^v_1, \ldots, r^v_{l(v)}, 0, 0, \ldots, 0)$. It is easy to compute these tables at the leaves—there are only two request vectors considered: $(0, 0, \ldots, 0)$ and $(1, 0, \ldots, 0)$. To compute the cost for the request vector r^v_i we have to combine all computed requests vectors (and their costs) of the children of v. At the end, we compute at the root of T the cost of r' (the only request vector considered at the root). This cost is the value $|F \setminus F'|$, which is equal to $|F' \setminus F| - 1$, the value we are looking for, decreased by one.

The crucial aspect of the running time of the algorithm is the size of the

tables stored at every node. Since every level can have at most n assigned codes, the size of a table stored at a node is at most $(n+1)^h$. The time to combine two tables is at most $(n+1)^{2h}$. This gives the overall running time $O(2^h n^{2h})$.

Theorem 5.8 *The one-step offline CA problem can be optimally solved in time* $O(2^h n^{2h})$.

We consider now the fixed parameter tractability of the one-step offline CA. Parametrized problems are described by languages $L \subseteq \Sigma^* \times \mathbb{N}$. If $(x,k) \in L$, we refer to k as the parameter. The concept of fixed parameter tractability is described for example in [48].

Definition 5.1 ([48]) *We say that L is* uniformly fixed-parameter tractable, *if there is an algorithm A, a constant c and a function $f : \mathbb{N} \to \mathbb{N}$ such that*

1. *the running time of $A(x,k)$ is at most $f(k)|x|^c$*
2. *$(x,k) \in L$ if and only if A accepts (x,k).*

We assume that our problem is given by a pair (x,k), where x encodes the code insertion on level l and the current code assignment and k is the parameter. We assume the encoding of the code assignment in the zero-one vector form $x_1,\ldots,x_{2^{h+1}-1}$ saying for every node of the tree whether there is an assigned code. Denote for the purpose of the rest of this section by n the size of the input, i.e., $n := |x| = 2^{h+1}-1$.

We consider various variants of parameters for the problem. The most natural ones are the number of moved codes m or the level l of the code insertion. To show the fixed parameter tractability, we reuse the ideas of the exact algorithm using dynamic programming, where we store at every node a table of all possible request vectors.

We first show that the problem is fixed parameter tractable, if the parameters are both m and l, i.e., we show an algorithm solving the problem in time $O(f(m,l)p(n))$ for some polynomial $p(n)$.

Having a code insertion into the code tree for level l, we know that we only move codes from lower levels than l. Hence, when building the tables at nodes, we consider only those request vectors that differ on levels $0,\ldots,l-1$ from the

current subtree (its actual request vector). From the assumption that we move at most m codes, we have that on each of these levels, the considered request vector can differ by at most m. Hence, the number of considered request vectors in every node is at most $(2m+1)^l$. To compute all the tables, we need to combine all the tables from the children nodes, i.e., we have to consider $(2m+1)^{2l}$ pairs for every node. From this we get the running time to be $O(2^h(2m+1)^{2l})$, which is certainly of the form $f(m,l)p(n)$.

For the case, where we have only l as the parameter, we immediately get that we move from every subtree T_v at most 2^l codes, hence we bound the number of codes moved in every subtree by a parameter (we note that we did not bound the overall number of moved codes) $m = 2^l$.

Consider now the case, where only m is the parameter. Since we move at most m codes within the tree, we know that at most m codes come into the subtree and at most m go away from the subtree. Hence, assigning for each such possibility a level out of $0,\ldots,l$, we get an upper bound of at most $(l+1)^{2m}$ request vectors to be considered at every node on level l. Since $l+1 \leq h$ for $l = 0,\ldots,h-1$ we get at every node at most $h^{2m} = (\log n)^{2m}$ requests. From [88] we can use the inequality $(\log n)^m \leq (3m\log m)^m + n$ to express the size of each table in the form $g(m)+n$. To compute the table for every node, we need time $n(g(m)+n)^2$ which is certainly of the form $f(m)p(n)$.

We can summarize the results of this section in the following theorem.

Theorem 5.9 *The one-step offline CA problem is fixed parameter tractable for the following parameters:*

- *the level l of the code insertion, and*
- *the number m of moved codes.*

5.4.4 An h-Approximation Algorithm for One-Step offline CA

In this section we propose and analyze a greedy algorithm for one-step offline CA, i.e., for the problem of assigning an initial code assignment request c_0 (for level l) into a code tree T with given code assignment F. The idea of the greedy algorithm A_{greedy} is to assign the request c_0 onto the root g of the subtree

T_g that contains the fewest assigned codes among all possible subtrees. From Lemma 5.3 we know that no optimal algorithm reassigns codes on higher levels than the current one; hence the possible subtrees are all those subtrees (with roots on level l) that do not contain assigned codes on or above their root. The greedy algorithm takes all assigned codes in T_g (denoted by $F(T_g)$) and reassigns them recursively (one by one) in the same way, always processing codes of higher level first.

In the following, for ease of presentation, when speaking of codes, we mean an assigned code from T. When referring to nodes of T, we do not distinguish between assigned and unassigned codes.

At every time t the greedy algorithm has to reassign a set C_t of codes into the current tree T^t. Initially, $C_0 = \{c_0\}$ and $T^0 = T$ (i.e., for simplicity the initial request c_0 is also viewed as a code). The codes from subtree T_g that have to be reassigned are removed from T and placed into C_i (formally, once they move to C_i, they should be called code requests, but we abuse the notation here a bit). Recall that for a given position, code or request c, its level is denoted by $l(c)$.

Algorithm 1 *Greedy algorithm* A_{greedy}*:*

$C_0 \leftarrow \{c_0\}; T^0 \leftarrow T$
$t \leftarrow 0$
WHILE $C_t \neq \emptyset$ DO
  $c_t \leftarrow$ element with highest level in C_t
  $g \leftarrow$ the root of a subtree T_g^t of level $l(c_t)$ with the fewest codes in it and no code on or above its root
  /* assign c_t to position g */
  $T^{t+1} \leftarrow (T^t \setminus F(T_g^t)) \cup \{g\}$
  $C_{t+1} \leftarrow (C_t \cup F(T_g^t)) \setminus \{c_t\}$
  $t \leftarrow t + 1$
END WHILE

In [80] a similar algorithm is proposed as a heuristic for the one-step offline CA. We prove that A_{greedy} has approximation ratio h. As the following example shows, this bound is asymptotically tight. Consider Figure 5.8. The request for

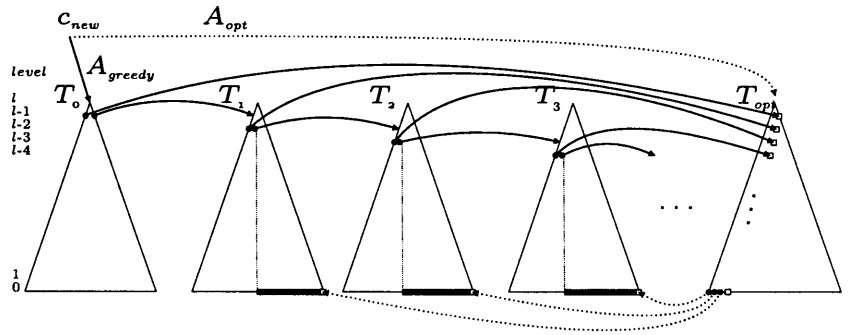


Figure 5.8: Example for the lower bound for A_{greedy}

a new code c_{new} is assigned by the greedy algorithm into the root of T_0 (which contains the least number of codes). The two codes on level $l-1$ from T_0 are reassigned as shown in the figure, one code can be reassigned into T_{opt} (with no additional reassignments) and the other one goes recursively into T_1 (whose left subtree contains 2 codes, i.e., the least among all other subtrees). The two codes from T_1 are reassigned in the same principle, one goes into T_{opt} and the second one into T_2 and so on. The reassignments stops in this fashion when the greedy algorithm processes the level 1 codes. There, it cannot accommodate a level 1 code into T_{opt} and it has to move additionally two codes from level 0. In total, the greedy algorithm does $2 \cdot l + 1$ reassignments while the optimal algorithm assigns c_{new} into the root of T_{opt} and reassigns the three codes from the leaf level into the trees T_1, T_2, T_3, requiring only 4 reassignments. Obviously, for this example the greedy algorithm is not better than $(2l+1)/4$ times the optimal. In general l can be $\Omega(h)$.

For the upper bound we compare A_{greedy} to the optimal algorithm A_{opt}. A_{opt} assigns c_0 to the root of some subtree T_{x_0}, the codes from T_{x_0} to some other subtrees, and so on. Let us call the set of subtrees to the root of which A_{opt} moves codes the *opt-trees*, denoted by $\mathcal{T}_{opt}$, and the arcs that show how A_{opt} moves the codes the *opt-arcs* (cf. Figure 5.9). By $V(\mathcal{T}_{opt})$ we denote the set of all nodes in $\mathcal{T}_{opt}$.

A sketch of the proof is as follows. First, we show that in every step t A_{greedy} has the possibility to assign the codes from C_t into positions inside the opt-trees. This possibility can be expressed by a code mapping $\phi_t : C_t \rightarrow V(\mathcal{T}_{opt})$. The key-property is now that in every step of the algorithm there is the theoretical

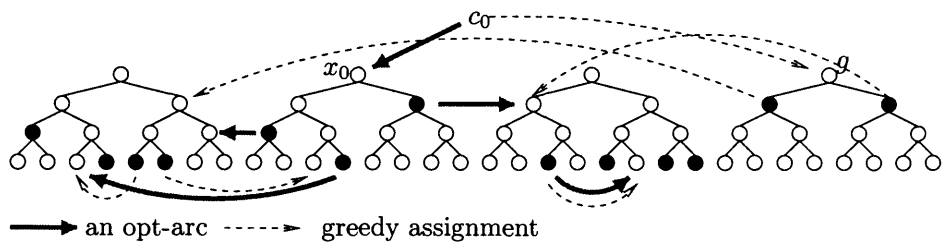


Figure 5.9: A_{opt} moves codes to assign a new code c_0 using opt-arcs. The opt-trees are subtrees to the root of which A_{opt} moves codes. Here, the cost of the optimal solution is 5. The greedy algorithm has cost 6.

choice to complete the current code assignment using the code mapping ϕ and the opt-arcs as follows: Use ϕ to assign the codes in C_t into positions in the opt-trees and then use the opt-arcs to move the codes out of these subtrees of the opt-trees to produce a feasible code assignment. We will see that this property is enough to ensure that A_{greedy} incurs a cost of no more than OPT on every level.

In the process of the algorithm it can happen that we have to change the opt-arcs in order to ensure the existence of ϕ_t. To model the necessary changes we introduce α_t-arcs that represent the changed opt-arcs after t steps of the greedy algorithm.

To make the proof-sketch precise, we use the following definitions:

Definition 5.2 *Let $\mathcal{T}_{opt}$ be the set of the opt-trees for a code request c_0 and let T^t (together with its code assignment F^t) be the code tree after t steps of the greedy algorithm A_{greedy}. An* α-mapping *at time t is a mapping $\alpha_t : M_{\alpha_t} \to V(\mathcal{T}_{opt})$ for some $M_{\alpha_t} \subseteq F^t$, such that $\forall v \in M_{\alpha_t} : l(v) = l(\alpha_t(v))$ and $\alpha_t(M_{\alpha_t}) \cup (F^t \setminus M_{\alpha_t})$ is a code assignment, where $\alpha_t(M_{\alpha_t}) = \bigcup_{u \in M_{\alpha_t}} \alpha_t(u)$.*

Note that in general, F^t is not a code assignment for all codes (for all code requests) since it does not contain the codes in C^t. The set $\alpha_t(M_{\alpha_t}) \cup (F^t \setminus M_{\alpha_t})$ represents the resulting code assignment (that again does not contain the codes in C^t) after reassignment of the codes $M_{\alpha_t} \subseteq F^t$ by α_t. We see the α-mapping as a description of how codes are moved (reassigned).

Definition 5.3 *Let T^t be a code tree, x, y be positions in T^t and α_t be an α-mapping. We say that y* depends *on x in T^t and α_t, if there is a path from x*

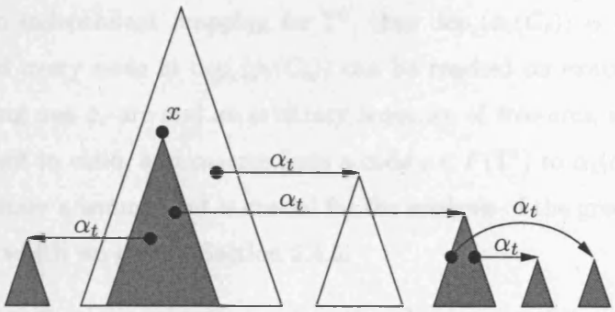


Figure 5.10: The filled subtrees represent all the positions that depend on x.

to y using only tree-edges from a parent to a child and α_t-arcs. By $\mathrm{dep}_t(x)$ *we denote the set of all positions y that depend on x in T^t and α_t. We say that an α_t arc (u, v) depends on x if* $u \in \mathrm{dep}_t(x)$.

For an illustration of this definition, see Figure 5.10. We write $\mathrm{dep}_t(X)$ for the set $\bigcup_{c \in X} \mathrm{dep}_t(x)$.

Definition 5.4 *At time t a pair (ϕ_t, α_t) of a code mapping $\phi_t : C_t \to V(\mathcal{T}_{opt})$ and an α-mapping α_t is called an* independent mapping *for T^t, if the following properties hold:*

1. *$\forall c \in C_t$ the levels of $\phi_t(c)$ and c are the same (i.e. $l(c) = l(\phi_t(c))$).*
2. *$\forall c \in C_t$ there is no code in T^t at or above the roots of the trees in* $\mathrm{dep}_t(\phi_t(c))$.
3. *the code movements realized by ϕ_t and α_t (i.e. the set $\phi_t(C_t) \cup \alpha_t(M_{\alpha_t}) \cup (F^t \setminus M_{\alpha_t})$) form a code assignment (where $\phi_t(C_t)$ stands for $\bigcup_{c \in C_t} \phi_t(c)$).*
4. *every node in the domain M_{α_t} of α_t is contained in* $\mathrm{dep}_t(\phi_t(C_t))$ *(i.e., no unnecessary arcs are in α_t).*

Independent mapping formally captures the (hypothetical) possibility of the greedy algorithm to finish the greedy assignment at time t and to follow the optimal steps, i.e., the remaining codes from C_t are assigned according to ϕ_t into some positions in opt-trees and the codes from the subtrees of $\phi_t(C_t)$ are moved according to α_t.

Note that ϕ_t and α_t can equivalently be viewed as functions and as collections of arcs of the form $(c, \phi_t(c))$ and $(u, \alpha_t(u))$, respectively. Note also that if a pair

(ϕ_t, α_t) is an independent mapping for T^t, then $\mathrm{dep}_t(\phi_t(C_t))$ is contained in opt-trees and every node in $\mathrm{dep}_t(\phi_t(C_t))$ can be reached on exactly one path from C_t (using one ϕ_t-arc and an arbitrary sequence of *tree-arcs*, which always go from parent to child, and α_t-arcs from a code $c \in F(T^t)$ to $\alpha_t(c)$).

Now we state a lemma that is crucial for the analysis of the greedy strategy, the proof of which we give in Section 5.4.5.

Lemma 5.10 *For every set C_t in algorithm A_{greedy} the following invariant holds:*

$$\textit{There is an independent mapping } (\phi_t, \alpha_t) \textit{ for } T^t. \tag{5.2}$$

We remark that Lemma 5.10 actually applies to all algorithms that work level-wise top-down and choose a subtree T_g^t for each code $c_t \in C_t$ arbitrarily under the condition that there is no code on or above the position g.

To show the approximation ratio of the greedy algorithm, we first express the cost of the optimal solution by the opt-trees:

Lemma 5.11 *(a) The optimal cost is equal to the number of assigned codes in the opt-trees plus one, and (b) it is equal to the number of opt-trees.*

Proof. Observe for (a) that A_{opt} moves all the codes in the opt-trees and for (b) that A_{opt} moves one code into the root of every opt-tree. □

Theorem 5.12 *The algorithm A_{greedy} has an approximation ratio of h.*

Proof. A_{greedy} works level-wise top-down. We show that on every level l the greedy algorithm incurs cost at most OPT. Consider a time t_l where A_{greedy} is about to start a new level l, i.e. before A_{greedy} assigns the first code on level l. Assume that C_{t_l} contains q_l codes on level l. Then A_{greedy} places these q_l codes in the roots of the q_l subtrees on level l containing the fewest codes. The code mapping ϕ_{t_l} that is part of the independent mapping $(\phi_{t_l}, \alpha_{t_l})$, which exists by Lemma 5.10, maps each of these q_l codes to a different position in the opt-trees. Therefore, the total number of codes in the q_l subtrees with roots at $\phi_{t_l}(c)$ (for c a code on level l in C_{t_l}) is at least the number of codes in the q_l subtrees chosen by A_{greedy}. Combining this with Lemma 5.11(a), we see that on every level A_{greedy} incurs a cost (number of codes that are moved away from their position in the tree) that is at most A_{opt}'s total cost. □

5.4.5 Proof of Lemma 5.10

We prove the lemma by induction on t. Assume that the code c_0 is to be inserted into the tree initially, and that A_{opt} assigns it to position x_0. For the base of the induction ($t = 0$), let $\phi_0(c_0) = x_0$ and let α_0 consist of all opt-arcs, i.e., all arcs (u, v) such that A_{opt} moves a code from u to v. It is easy to see that (ϕ_0, α_0) is an independent mapping.

Assume now that the lemma holds after $t > 0$ iterations of the greedy algorithm. We show how to construct $(\phi_{t+1}, \alpha_{t+1})$ from the independent mapping (ϕ_t, α_t). In iteration $t + 1$, the greedy algorithm A_{greedy} assigns the code c_t of highest level in C_t to a position g in T^t, such that there is no assigned code on or above g.

Case 1. There is a code c'_t in C_t with $\phi_t(c'_t) = g$. If $c'_t \neq c_t$, we exchange the ϕ_t values of c'_t and c_t while maintaining (ϕ_t, α_t) as an independent mapping for T^t. Thus, we can assume that $\phi_t(c_t) = g$. We set

$$\phi_{t+1} = \{(c, \phi_t(c)) \mid c \in C_t \setminus \{c_t\}\} \cup \{(c, \alpha_t(c)) \mid c \in F(T_g^t)\}$$

and

$$\alpha_{t+1} = \alpha_t \setminus \{(c, \alpha_t(c)) \mid c \in F(T_g^t)\}.$$

We can see it as performing one step in following the whole code reassignment given by (ϕ_t, α_t). It is easy to see that $(\phi_{t+1}, \alpha_{t+1})$ is an independent mapping for T^{t+1}.

We remark that Case 1 could also be handled in the same way as Case 2 below, but we have chosen to give a direct treatment of Case 1 in order to illustrate some of the proof ideas on a simple case.

Case 2. There is no code c'_t in C_t with $\phi_t(c'_t) = g$. In this case, T_g^t can contain a number of codes, some of which may be in the domain M_{α_t} of α_t. Furthermore, there can be ϕ_t-arcs and α_t-arcs pointing into T_g^t. An example is shown in Figure 5.11. All codes from T_g^t are going to be moved to C_{t+1} and we have to define a ϕ_{t+1}-arc for all these codes in T_g^t. Also (since no code can move to T_g^t anymore) we must find a new destination outside T_g^t for those ϕ_t-arcs

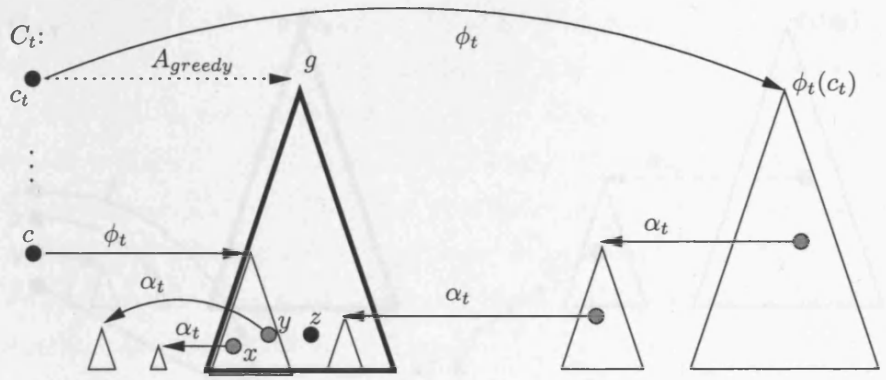


Figure 5.11: T_g^t contains the heads of α_t-arcs and ϕ_t-arcs as well as codes with and without α_t-arcs.

and α_t-arcs pointing into T_g^t that we need for the construction of $(\phi_{t+1}, \alpha_{t+1})$. The idea is to redirect these arcs into the "free space" that was reserved for c_t by $\phi_t(c_t)$ and the subsequent reassignments, but was not used by the greedy algorithm.

First, we will define an intermediate *generalized* independent mapping (ϕ, α) for T^{t+1} in which we allow *loose ends*, i.e., we allow a code c to have as head of its α-arc or ϕ-arc a *dummy tree* (that is not part of the real tree) of the required capacity. In a second step, we will fix loose ends by finding proper destinations in $\text{dep}(\phi_t(c_t))$ for them (where dep refers to the dependency induced by tree-arcs and the current α-arcs). In the end, a part of the resulting (ϕ, α) without loose ends will be used to define $(\phi_{t+1}, \alpha_{t+1})$.

We proceed as follows. For each assigned code c at a node v in T_g^t that is not in the domain M_{α_t} of α_t, define $\phi(c) = v$. For each assigned code c in T_g^t that has an α_t-arc, define $\phi(c) = \alpha_t(c)$. For all codes c in $C_t \setminus \{c_t\}$, set $\phi(c) = \phi_t(c)$. Let $\alpha = \alpha_t \setminus \{(u, v) \mid u \in T_g^t\}$. Finally, replace every α-arc or ϕ-arc (u, v) for which $v \in T_g^t$ by a loose end, i.e., an α-arc or ϕ-arc pointing from u to a dummy tree of height $l(v)$. The generalized mapping (ϕ, α) constructed in this way is indeed independent. Figure 5.12 shows the generalized mapping (ϕ, α) resulting from the situation in Figure 5.11.

Dummy trees that can be reached from $\phi_t(c_t)$ along tree-arcs and α-arcs are called *inactive*, all other dummy trees are called *active* (i.e., all dummy trees reachable from $\phi(C_{t+1})$). Active dummy trees have to be fixed (so that we can

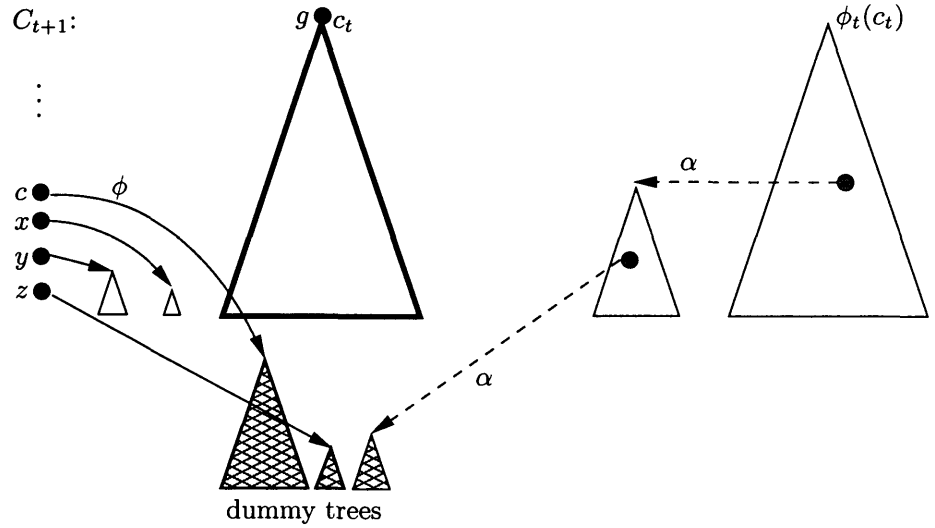


Figure 5.12: The constructed generalized independent mapping (ϕ, α). All shown α-arcs are inactive (indicated by dashed lines). The rightmost dummy tree is inactive, the other two are active.

eventually obtain an independent mapping without loose ends), while inactive dummy trees will either become active later on or will be discarded in the end. Similarly, we set all α-arcs that can be reached from $\phi_t(c_t)$ along tree-arcs and α-arcs *inactive*, and all other α-arcs *active*. Inactive α-arcs will either become active later on or will be discarded in the end as well.

Let U denote the capacity of the tree T_g, i.e., $U = 2^{l(g)}$. Note that all dummy trees were generated from independent subtrees of T_g^t. Therefore, the total capacity of all dummy trees is at most U. Let U_a be the total capacity of active dummy trees and U_i be the total capacity of inactive dummy trees. We have $U_a + U_i \leq U$.

We want to use $\text{dep}(\phi_t(c_t))$ for finding new destinations for α-arcs or ϕ-arcs that point to active dummy trees. We say that a path from $\phi_t(c_t)$ to some tree node v is *strict* if it follows tree-arcs downward from nodes without assigned codes in T^{t+1} and α-arcs from nodes with assigned codes in T_{t+1}. Now we can define the *available capacity* in $\text{dep}(\phi_t(c_t))$ to be the number of leaves that are not in dummy trees and that can be reached from $\phi_t(c_t)$ along a strict path that does not contain the head of any ϕ-arc or active α-arc. Note that a position v in $\text{dep}(\phi_t(c_t))$ can be used as the new head of an α-arc or ϕ-arc if and only if v is not in a dummy tree, there is no code at or above v, and no ϕ-arc or active

α-arc points to a position in dep(v) or to a position p such that v is in dep(p). We call v an *available* position. Otherwise, the position v is called *unavailable.*

The available capacity in dep($\phi_t(c_t)$) is $U - U_i$ initially, since only the loose ends in dep($\phi_t(c_t)$) reduce the available capacity. The total capacity of active dummy trees is $U_a \leq U - U_i$ (i.e., it suggests we have enough capacity to accommodate the ϕ-arcs and active α-arcs with loose ends). In the following we will maintain the invariant that the total capacity of active dummy trees is at most the available capacity in dep($\phi_t(c_t)$).

We fix the active dummy trees one by one in order of non-increasing levels. Assume that we are currently processing a dummy tree of level d that is the head of an α-arc or ϕ-arc (x, y). Consider all nodes v_d of level d in T^{t+1} that do not have assigned codes and are reachable from $\phi_t(c_t)$ along strict paths. Observe that a node v_d is unavailable only if it is inside an inactive dummy tree or if the path from $\phi_t(c_t)$ to v_d passes through the head of an active α-arc or a ϕ-arc. However, it is not possible that all nodes v_d are unavailable, because then the total available capacity in dep($\phi_t(c_t)$) would be zero, contradicting our invariant (it is important here that we are processing the codes in the non-increasing order and thus cannot block the available positions by some low level codes). Thus, we can find a node v_d that is available. We replace (x, y) by (x, v_d) and make all α-arcs reachable from v_d as well as all inactive dummy trees reachable from v_d active. (Note that no active dummy tree can have been reachable from v_d before this operation, since we fix the active dummy trees in order of non-increasing levels.) Let U' be the total capacity of previously inactive dummy trees that were made active now. The total capacity of active dummy trees decreases by $2^d - U'$, and the total available capacity in dep($\phi_t(c_t)$) decreases by $2^d - U'$ as well (since the part of dep($\phi_t(c_t)$) that is reachable from v_d had available capacity exactly $2^d - U'$). Therefore, the invariant is maintained and the process can be continued until no active dummy trees are left. The process terminates because the total capacity of active dummy trees never increases and in each step the number of active dummy trees of highest level decreases by one (and only dummy trees of lower levels may become active). A possible result of applying this process to the generalized independent mapping of Figure 5.12 is shown in Figure 5.13.

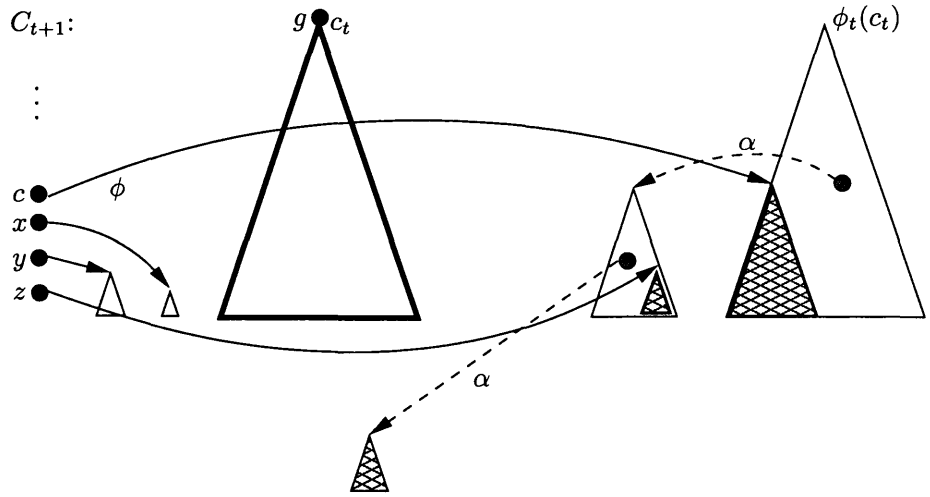


Figure 5.13: The final generalized independent mapping (ϕ, α) in which all active dummy trees have been fixed. The independent mapping $(\phi_{t+1}, \alpha_{t+1})$ is obtained by deleting the inactive α-arcs and discarding the remaining inactive dummy tree.

When all active dummy trees are fixed, we let $\phi_{t+1} = \phi$ and $\alpha_{t+1} = \{(u, v) \in \alpha \mid (u, v) \text{ is active } \}$. Since (ϕ, α) was a generalized independent mapping and $(\phi_{t+1}, \alpha_{t+1})$ does not contain loose ends, we have that $(\phi_{t+1}, \alpha_{t+1})$ is an independent mapping as required.

5.5 Online CA

The most natural version of the whole family of code assignment problems is perhaps the online version, where the codes are being inserted and deleted from the tree over time. We study the online CA in this section. We present a lower bound on the competitive ratio (of any competitive online algorithm), showing that a (small) code-tree fragmentation is (always) inevitable. We further propose and analyze several algorithms.

Theorem 5.13 *No deterministic algorithm A for the online CA problem can be better than* 1.5*-competitive.*

Proof. Let A be any deterministic algorithm for the online CA problem. Consider N consecutive leaf insertions. Whatever the algorithm's code assignment is, the adversary can ask for $N/2$ code deletions to get to the situation in Fig-

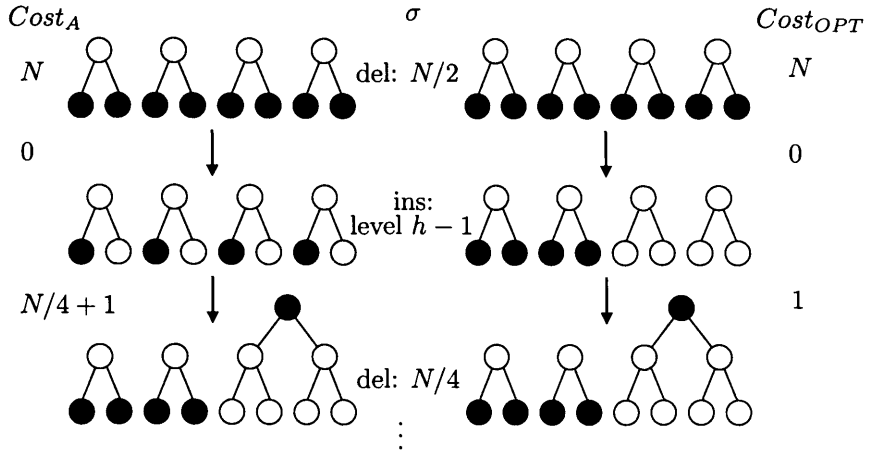


Figure 5.14: Lower bound for the online assignment problem.

ure 5.14 (i.e., deletion of every second code on the leaf level). Then a request for a code assignment on level $h-1$ causes $N/4$ code reassignments, leaving half of the tree full of assigned leaf codes. W.l.o.g., it is the left half of the tree. The adversary proceeds recursively with the left subtree of full leaf codes, deleting every second leaf code, etc. We can repeat this process $(\log_2 N - 1)$ times. The overall number of code reassignments of the algorithm A is $C_A = N + T(N)$, where $T(N) = 1 + N/4 + T(N/2)$ and $T(2) = 0$. Computing the closed form of $T(N)$ we get $T(N) = \log_2 N - 1 + \frac{N}{2}(1 - 2/N)$.

The optimal algorithm A_{opt} assigns the leaves in such a way that it does not need any code movement at all. Thus, A_{opt} needs $C_{opt} = N + \log_2 N - 1$ code assignments. If $C_A \leq c \cdot C_{opt}$ then $c \geq \frac{3N/2+\log_2 N-2}{N+\log_2 N-1} \longrightarrow_{N\to\infty} 3/2$. □

5.5.1 Greedy Strategies

We study a natural greedy algorithm that uses at every code insertion/deletion an optimal algorithm A for the one-step offline CA problem. As an optimal algorithm breaks ties in an unspecified way, the online strategy can vary for different optimal one-step offline algorithms.

Theorem 5.14 *Any deterministic greedy online strategy, i.e., a strategy that minimizes the number of reassignments for every request, is $\Omega(h)$ competitive.*

Proof. Let A be a fixed, greedy online strategy (that minimizes the number of reassignments for every request). We first insert $N/2$ codes at level 1 (i.e., we

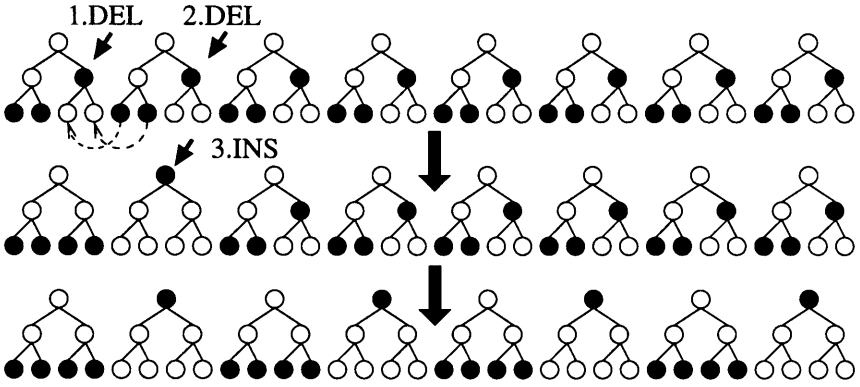


Figure 5.15: Requests that an online greedy strategy cannot handle efficiently.

fill level 1 of the tree with assigned codes). As A is deterministic we can now ask for code deletion of every second code on level 1. Subsequently, we insert $N/2$ codes at level 0, using the whole bandwidth of the tree. This leads to the situation depicted in Figure 5.15. Then we delete two codes at level $l = 1$ (as A is deterministic it is clear which codes to delete) and immediately assign a code at level $l + 1$. As it is optimal (and up to symmetry unique) the algorithm A moves two codes as depicted. An optimal strategy arranges the level-1 codes in a way that it does not need any additional reassignments. We proceed in this way along level 1 in the first round, then left to right on level 2 in a second round, and continue towards the root. Altogether we move $N/4$ codes in the first round and we assign $N/2^3$ codes. In general, in every round i we move $N/4$ level-0 codes and assign $N/2^{i+2}$ new codes on level $i + 1$. Altogether the greedy strategy needs $O(N) + (N/4)\Omega(\log N) = \Omega(N \log N)$ (re-)assignments, whereas the optimal strategy does not need any reassignments and only $O(N)$ assignments. □

5.5.2 Compact Representation Algorithm

This algorithm maintains the assigned codes in the tree T sorted according to their levels and also keeps them compact. For a given node/code $v \in T$, we denote by $w(v)$ its string representation, i.e. the description of the path from the root to the node/code, where 0 means left child and 1 right child. We use the lexicographic ordering when comparing two string representations. By U we denote the set of unblocked nodes of the tree. We maintain the following

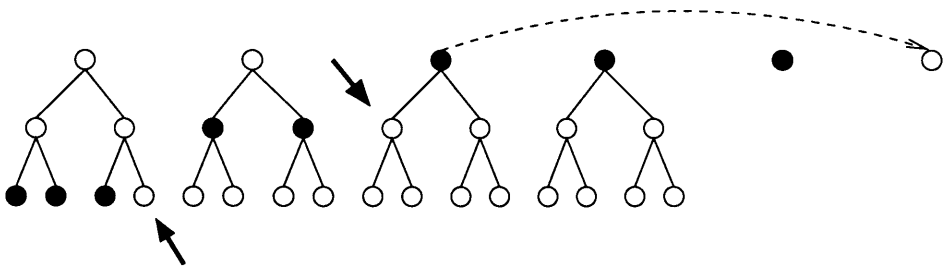


Figure 5.16: Algorithm $A_{compact}$ finds the rightmost position (blocked or unblocked) for a code insertion and reassigns at most one code at every level.

invariants:

$$\forall \text{ codes } u, v \in F: \; l(u) < l(v) \Rightarrow w(u) < w(v), \tag{5.3}$$

$$\forall \text{ nodes } u, v \in T: \; l(u) \leq l(v) \; \wedge \; u \in F \; \wedge \; v \in U \; \Rightarrow \; w(u) < w(v). \tag{5.4}$$

Following (5.3) we maintain the codes in the tree ordered from left to right according to their levels (higher level assigned codes are to the right of lower level assigned codes). Invariant (5.4) keeps the codes compact (no unblocked code to the left of any assigned code on the same level).

In the following analysis we show that this algorithm is not worse than $O(h)$ times the optimum for the offline version. We also give an example that shows that the algorithm is not asymptotically better than this.

Theorem 5.15 *There is an algorithm $A_{compact}$ satisfying invariants (5.3) and (5.4) that performs at most $O(h)$ code reassignments per request.*

Proof. We show that for both code insertion and code deletion we need to make at most h code reassignments. When we insert a code on level l, we look for the rightmost unassigned position (it can be blocked) on level l that maintains the invariants (5.3) and (5.4) among codes on level $0, \ldots, l$. Either the found node is not blocked, so that we do not move any codes, or the code is blocked by some assigned code on a higher level $l' > l$ (see Figure 5.16). In the latter case we remove this code to free the position for level l and handle the new assignment request for level l' recursively. Since we move at most one code at each level and we have h levels, we move at most h codes for each insertion request.

Handling the deletion operation is similar, we just move the codes from right

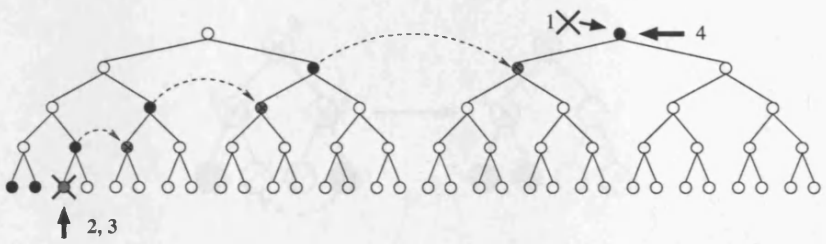


Figure 5.17: Code assignments for levels $0, 0, 1, 2, 3, 4, \ldots, h-1$ and four consecutive operations: 1. delete on level $h-1$, 2. insert on level 0, 3. delete on level 0, 4. insert on level $h-1$.

to left in the tree and move at most one code per level to maintain the invariants.

□

Corollary 5.16 *The algorithm $A_{compact}$ satisfying invariants (5.3) and (5.4) is $O(h)$-competitive.*

Proof. In the sequence $\sigma = \sigma_1, \ldots, \sigma_m$ the number of deletions d must be smaller or equal to the number i of insertions, which implies $d \leq m/2$. The cost of any optimal algorithm is then at least $i \geq m/2$. On the other hand, A_{compact} incurs a cost of at most $m \cdot h$, which implies that it is $O(h)$-competitive. □

Theorem 5.17 *Any algorithm satisfying invariant* (5.3) *is $\Omega(h)$-competitive.*

Proof. Let A_I be an algorithm that always follows invariant (5.3). Consider the sequence of requests for code assignments on levels 0, 0, 1, 2, 3,..., $h-1$. For these requests, there is a unique code assignment satisfying invariant (5.3), see Figure 5.17. Consider now two requests—deletion of the code at level $h-1$ and insertion of a code on level 0. Then A_I has to move every code on level $l \geq 1$ to the right to create space for the code assignment on level zero and maintain the invariant (5.3). This takes $h-1$ code (re-)assignments. Consider as the next requests the deletion of the third code on level zero and an insertion on level $h-1$. Again, to maintain the invariant (5.3), A_I has to move every code on level $l \geq 1$ to the left. This takes again $h-1$ code (re-)assignments. An optimal algorithm can handle these four requests with two assignments, since it can assign the third code on level zero in the right subtree, where A_I assigns the code on level $h-1$. Repeating these four requests k times, the total

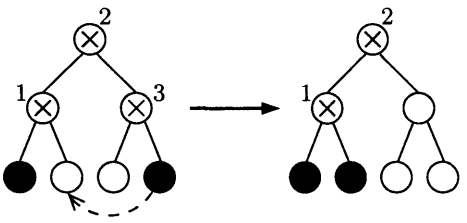


Figure 5.18: Reassignment of one code reduces the number of blocked codes from 3 to 2.

cost of the algorithm A_I is then $C_A = h + 1 + k \cdot (2h - 2)$, whereas OPT has $C_{OPT} = h + 1 + k \cdot 2$. As k goes to infinity, the ratio C_A/C_{OPT} is $\Omega(h)$. □

5.5.3 Minimizing the Number of Blocked Codes

The idea of minimizing the number of blocked codes is mentioned in [98, 89] and empirical studies are presented. In this section we given an analysis of the algorithm that satisfies the invariant:

$$\#\text{ blocked codes in } T \text{ is minimum.} \tag{5.5}$$

In Figure 5.18 we see a situation that does not satisfy the invariant (5.5). Moving a code from level 0 reduces the number of blocked codes by one.

We can prove that this approach is equivalent to minimizing the number of *gap trees* on every level (Theorem 5.18). Recall that a gap tree is a maximal subtree of unblocked codes in a code tree.

Definition 5.5 *The level of the root of a gap tree is called the* level of the gap tree. *The vector* $q = (q_0, \ldots, q_h)$, *where* q_i *is the number of gap trees on level* i, *is called the* gap vector *of the tree* T.

See Figure 5.19 for an example of the definition. Observe that the invariant (5.5) implies at most one gap tree at each level. If there were two gap trees on a level l we could move the sibling tree of one of the gap trees to fill the other gap tree, reducing the number of blocked codes by at least one (concept from Figure 5.18). As the following theorem shows, the other direction of this implication holds as well.

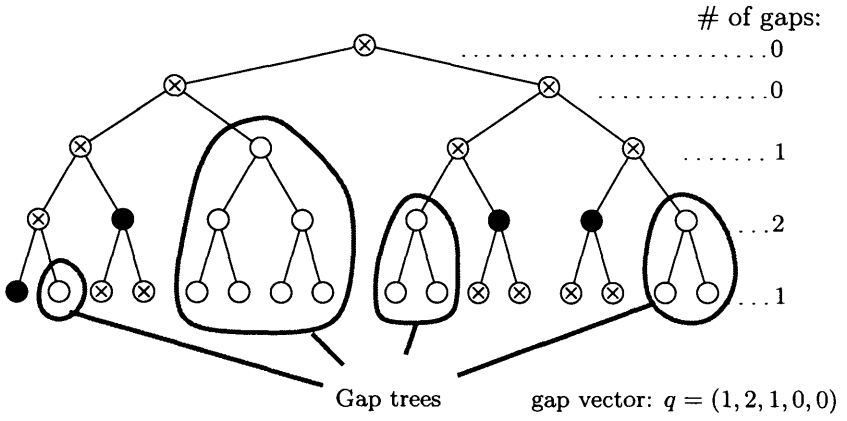


Figure 5.19: Illustration for the definition of gap tree and gap vector.

Theorem 5.18 *Let T be a code tree after realizing the code insertions and deletions from a sequence of requests σ. T has at most one gap tree on every level if and only if T has a minimum number of blocked codes.*

Proof. We are left to prove the second implication. Suppose T has at most one gap tree on every level. The minimum number of blocked codes has to be attained at some tree T' with at most one gap tree on every level, otherwise we could reduce the number of blocked codes by filling one of the two gap trees from the same level with the sibling tree of the other gap tree.

The free bandwidth capacity of T can be expressed in terms of the gap vector

$$cap = \sum_{i=0}^{h} q_i 2^i.$$

As $q_i \leq 1$, the gap vector is the binary representation of the number *cap* and thus the gap vector q is unique for every tree serving requests σ with at most one gap tree at every level (and hence also for T').

The gap vector determines also the number of blocked codes:

$$\#\text{ blocked codes} = (2^{h+1} - 1) - \sum_{i=0}^{h} q_i (2^{i+1} - 1).$$

Thus, every tree for requests σ with at most one gap tree at every level has the same number of blocked codes, i.e., T has the same number of blocked codes as T'. □

Now we are ready to define the algorithm A_{gap} (Algorithm 2). As we will show, on insertions A_{gap} never needs any extra reassignments.

Algorithm 2 *A_{gap}:*

1. *Insert:* • *Assign the new code into the smallest gap where it fits.*
2. *Delete:* • *If after the deletion a second gap tree appears on some level, move one of their sibling subtrees to "fill" the gap tree*
   • *Look for a second gap tree on a higher level and treat it recursively.*

Lemma 5.19 *The algorithm A_{gap} has always a gap tree of sufficient height to assign a code on level l and at every step the number of gap trees at every level is at most one.*

Proof. Let *cap* denote the available capacity of the tree T when processing the request for code on level l. Because the code request can be served, we have $cap \geq 2^l$. We can express the available capacity via the gap vector as $cap = \sum_i q_i 2^i$. Summing only over gap trees on level $i < l$ we get a capacity $cap' = \sum_{i=0}^{l-1} q_i 2^i \leq 2^0 + 2^1 + \ldots 2^{l-1} = 2^l - 1$. Therefore, there exists a gap tree on level $j \geq l$.

Next, consider an insertion operation into the smallest gap tree on level l' where the code fits. New gap trees can occur only on levels j, $l \leq j < l'$ and only within the gap tree on level l'. Also, at most one new gap tree can occur on every level. Suppose that after creating a gap tree on level j, we have in T more than one gap tree on this level. Then, since $j < l'$ and $l \geq j$, we could have assigned the code into this smaller gap tree, a contradiction. Therefore, after an insertion there is at most one gap tree on every level.

Consider now a deletion of a code. The nodes of the subtree of the deleted code become unblocked, i.e., they belong to some (new) gap tree. Only one new gap tree can occur in the deletion operation (and some gap trees may disappear). Thus, when the newly created gap tree is the second one at the level, we "fill" this gap tree (according to the algorithm A_{gap}) and then we recursively handle the newly created gap tree at a higher level. In this way the problem with 2 gap trees is moved to higher levels. Because we cannot have two gap trees on

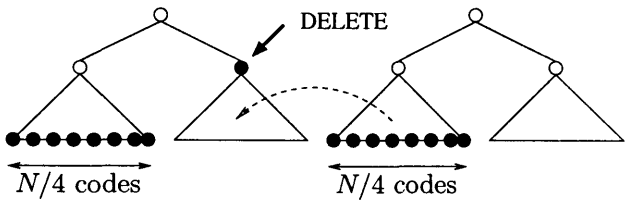


Figure 5.20: Worst case number of movements for algorithm A_{gap}.

level $h-1$, we eventually end up with a tree with at most one gap tree at each level. □

The result and its proof shows that upon insertions, the algorithm does not need any code movements.

Corollary 5.20 *The algorithm A_{gap} is optimal for the insertion-only online CA problem.*

However, similarly to the compact representation algorithm, this algorithm is $\Omega(\log N)$-competitive.

Theorem 5.21 *Algorithm A_{gap} is $\Omega(\log N)$-competitive.*

Proof. The proof is basically identical with the proof of Theorem 5.14. □

The algorithm A_{gap} has even a very bad worst case number of code movements. Consider the four subtrees on level $h-2$, where the first one has $N/4$ leaf codes inserted, its sibling has a code on level $h-2$ inserted and the third subtree has again $N/4$ leaf codes inserted (Figure 5.20). After deletion of the code on level $h-2$, A_{gap} is forced to move $N/4$ codes. This is much worse than the worst case for the compact representation algorithm.

5.6 Summary of Results and Open Problems

We have studied the OVSF-code assignment problem from the algorithm-theoretic perspective. For the one-step offline CA we have shown non-optimality of greedy algorithm that consider the current subtrees for evaluation if a position is good for a code insertion. We have presented an $O(\log n)$-approximation algorithm, an optimal algorithm and fixed-parameter tractable algorithms for various parameters. For the online CA we have shown a lowerbound 1.5 for the compet-

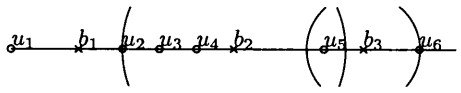
itive ratio of any online algorithm, and $\log n$-competitive algorithm. For the insertions-only CA we have presented an optimal online algorithm.

It is an interesting question, whether there is an approximation algorithm with better approximation ratio than $O(\log n)$.

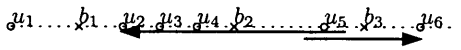
Chapter 6

Joint Base Station Scheduling

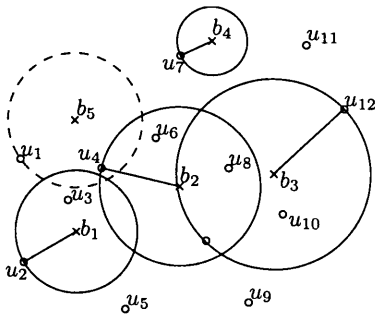
The combinatorial problem that is studied in this chapter reflects some of the real-life problems in the area of load balancing in time-division mobile networks. In such a network, mobile users are *served* by a set of base stations. Roughly speaking, serving a user means that the user communicates with one of the base stations via the common radio channel and uses the services offered by the base station (voice transmission, internet access, GPS positioning, etc.). The time is discretized into slots (rounds) 1,2,.... In each time slot/round (of the time division multiplexing) each base station serves at most one user. Conventionally, each user is assigned to the (single) base station of the cell in which the user is currently present. The user is being served by the base station until she leaves her cell or her demand is satisfied. The amount of data that a user receives depends on the strength of the signal that she receives from her assigned base station and on the interference, i.e., on all signal power that she receives from other base stations. In [42], Das et al. propose a novel approach: Clusters of base stations jointly decide which users they serve in which round in order to increase network performance. Intuitively, this approach increases throughput, when in each round neighboring base stations try to serve pairs of users such that the mutual interference is low. We turn this approach into a discrete scheduling problem in one and two dimensions (see Figure 6.1), the Joint Base Station



(a) This figure describes a possible situation in some time slot (round). Base station b_2 serves user u_2, b_3 serves user u_6. Users u_3, u_4 and u_5 are blocked and cannot be served. Base station b_1 cannot serve u_1 because this would create interference at u_2.



(b) Arrow representation of the situation from (a).



(c) A possible situation in some time slot in the 2D case. Users u_2, u_4, u_7 and u_{12} are served. Base station b_5 cannot serve user u_1 here, because this would create interference at u_4 as indicated by the dashed circle.

Figure 6.1: The JBS-problem in one and two dimensions

Scheduling problem (JBS).

In one dimension (see Figure 6.1(a)) we are given a set of n users as points $\{u_1, \ldots, u_n\}$ on a line and we are given positions $\{b_1, \ldots, b_m\}$ of m base stations (also on the line). We note that such a setting could correspond to a scenario where the base stations and users are located along a straight road. In our model, when a base station b_j serves a user u_i this creates interference in an interval of length $2|b_j - u_i|$ around the midpoint b_j. In each round each base station can serve at most one user such that at the position of this user there is no interference from any other base station. The goal is to serve all users in as few rounds as possible. In two dimensions users and base stations are represented as points in the plane. When base station b_j serves user u_i this creates interference in a disk with radius $\|b_j - u_i\|_2$ and centre b_j (see Figure 6.1(c)). Again, the goal is to serve all users in as few rounds as possible. In both dimensions, the problem can be viewed as firstly to decide which base station is going to serve each user and secondly to decide in which round the user is going to be served.

The one-dimensional problem is related to interval scheduling problems, where a set of intervals is to be scheduled such that no two intervals scheduled in the same round intersect. Here, the interference around a base station can be viewed as an interval, but because of the particular way how interference operates, the interval has a direction and the interval becomes an arrow with a head and a tail (the user that is served represents the head of the interval).

Intersection of two intervals is allowed, if only the tails of arrows intersect (intersecting tails correspond to interference that does not affect the users at the heads of the arrows).

Also, the problem has many similarities with interval graphs, with the speciality that we have conflict graphs of arrows, where the conflict rules are defined by the interference. This naturally leads to a notion of *arrow graphs*. The problem of scheduling data transmissions in the smallest number of discrete rounds can be expressed as the problem of colouring the corresponding arrow graph with the smallest number of colours, where the colours represent rounds.

6.1 Problem Definitions and Model

The setting is given by a set $B = \{b_1, \ldots, b_m\}$ of *base stations* and a set $U = \{u_1, \ldots, u_n\}$ of *users*. Each user and base station is placed in the Euclidean plane.

In the one-dimensional case users and base stations lie on a line and they are ordered from left to right, i.e., for $i < j$, user u_i is to the left of user u_j and base station b_i is to the left of base station b_j. The interference is modeled by *interference arrows*. Each pair of user $u \in U$ and base station $b \in B$ forms an interference arrow, which is an arrow pointing to the user u and having b as the midpoint (see Figure 6.1(b) for an example). Thus, the interference arrow for the pair (u_i, b_j) has its head at u_i and its midpoint at b_j. We denote the set of all arrows resulting from pairs $P \subseteq U \times B$ by $\mathcal{A}(P)$. If it is clear from the context, we call the interference arrows just *arrows*. For each user u_i we have to decide which base station serves u_i. This is equivalent to selecting one arrow (u_i, b_j) among all possible arrows from $\{u_i\} \times B$. For every user u_i we have to decide in which round she is going to be served. This scheduling can be viewed as labelling the chosen arrow (u_i, b_j) (i.e., b_j was decided to serve u_i) with a number representing the round, which can be equivalently viewed as colouring. We say that two arrows are *compatible* if no head is contained in the other arrow; otherwise, we say that they are in *conflict*. (Formally, the head u_i of the arrow for (u_i, b_j) is contained in the arrow for (u_k, b_l) if u_i is contained in the closed interval $[b_l - |u_k - b_l|, b_l + |u_k - b_l|]$.) Thus, if two users u_i and

u_k are to be scheduled in the same round, then the chosen arrows (u_i, b_j) and (u_k, b_l) have to be compatible. This models the condition that any user must not get any interference from any other base station at the time when she is being served. If we want to emphasize which user is affected by the interference from another transmission, we use the term *blocking*, i.e., arrow a_i blocks arrow a_j if a_j's head is contained in a_i.

In the two-dimensional case the positions of base stations are in the Euclidean plane. The interference is modeled by *interference disks* $d(b_i, u_j)$ with center b_i and radius $\|b_i - u_j\|_2$. We denote the set of interference disks for the user base-station pairs from a set P by $\mathcal{D}(P)$. Two interference disks are in *conflict* if the user that is served by one of the disks is contained in the other disk; otherwise, they are compatible. The problems can now be stated as follows:

1D-JBS

Input: User positions $U = \{u_1, \ldots, u_n\} \subset \mathbb{R}$ and base station positions $B = \{b_1, \ldots, b_m\} \subset \mathbb{R}$.

Output: A set P of n user base-station pairs such that each user is in exactly one pair, and a colouring $C : \mathcal{A}(P) \to \mathbb{N}$ of the set $\mathcal{A}(P)$ of corresponding arrows such that any two arrows $a_i, a_j \in \mathcal{A}(P)$, $a_i \neq a_j$, with $C(a_i) = C(a_j)$ are compatible.

Objective: Minimize the number of colours used.

2D-JBS

Input: User positions $U = \{u_1, \ldots, u_n\} \subset \mathbb{R}^2$ and base station positions $B = \{b_1, \ldots, b_m\} \subset \mathbb{R}^2$.

Output: A set P of n user base-station pairs such that each user is in exactly one pair, and a colouring $C : \mathcal{D}(\mathcal{P}) \to \mathbb{N}$ of the set $\mathcal{D}(\mathcal{P})$ of corresponding disks such that any two disks $d_i, d_j \in \mathcal{D}(\mathcal{P})$, $d_i \neq d_j$, with $C(d_i) = C(d_j)$ are compatible.

Objective: Minimize the number of colours used.

For simplicity we will write c_i instead of $C(a_i)$ in the rest of this chapter. From the problem definitions above it is clear that both the 1D- and the 2D-JBS

problems consist of a *selection problem* and a *colouring problem.* In the selection problem we want to select one base station for each user in such a way that the arrows (disks) corresponding to the resulting set P of user base-station pairs can be coloured with as few colours as possible. We call a selection P *feasible* if it contains exactly one user base-station pair for each user. Determining the cost of a selection is then the colouring problem. This can also be viewed as a problem in its own right, where we no longer make any assumption on how the set of arrows (for the 1D problem) is produced. The conflict graph $G(A)$ of a set A of arrows is the graph in which every vertex corresponds to an arrow and there is an edge between two vertices if the corresponding arrows are in conflict. We call such conflict graphs of arrows *arrow graphs.* The *arrow graph colouring problem* asks for a proper colouring of such a graph. It is similar in spirit to the colouring of interval graphs. As we will see in Section 6.3.1, the arrow graph colouring problem can be solved in time $O(n \log n)$. We finish this section with a simple lemma that reduces the number of considered arrows in the selection problem and leads to a new type of arrows.

Lemma 6.1 *For each 1D-JBS instance there is an optimal solution in which each user is served either by the closest base station to her left or by the closest base station to her right.*

Proof. This follows by a simple exchange argument: Take any optimal solution that does not have this form. Then exchange the arrow where a user is not served by the closest base station in some round against the arrow from the closest base station on the same side (which must be idle in that round). Shortening an arrow without moving its head can only resolve conflicts. Thus, there is also an optimal solution with the claimed property. □

The two possible arrows by which a user can be served according to this lemma are called *user arrows*, or *pair of user arrows.* From now on we are only interested in solutions of this kind. It follows that for a feasible selection one has to choose one arrow from each pair of user arrows.

6.2 Related Work and New Contributions

Das et al. [42] propose an involved model for load balancing that takes into account different fading effects and calculates the resulting signal to noise ratios at the users for different schedules. In each round only a subset of all base stations is used in order to keep the interference low. The decision which base stations to use is taken by a central authority. The search for this subset is formulated as a (nontrivial) optimization problem that is solved by complete enumeration and that assumes complete knowledge of the channel conditions. The authors perform simulations on a hexagonal grid, propose other algorithms, and reach the conclusion that the approach has the potential to increase throughput.

The problem of using the "scarce" radio spectrum in an efficient way in order to increase the network throughput attracted a lot of attention from researchers from various fields. Although not directly related to our work, we find it worthy to mention slightly different models and optimization goals that were studied in the context of the radio spectrum management. Many researchers focus on the assignment of channels to the communication cell itself. A channel is a telecommunication term that refers to a unit of a division of the radio spectrum, that can be used simultaneously with other channels while maintaining an acceptable received radio signal. A comprehensive survey on channel assignment algorithms is given in [74]. The main approach is to use one channel for more users in different cells, if the interference constraint allows that. Mobile users that are too close cannot communicate via the same channel. The algorithms presented there are of various types and optimization goals, but the exact solution of a mathematical model (if given at all) is often not possible to find in polynomial time and hence heuristics are used. If we look at the problem with a "discretizing" approach (graph-theoretic and algorithm-theoretic), we can relate it to the colouring problems in graphs (more precisely, to the list-colouring and T-colouring of graphs), if the base stations are assigned colours representing the channels and an edge in the graph of base stations states that the base station cannot use the same channel. The channel assignment problems are known as frequency assignment problems in the algorithm-theoretic community. A comprehensive survey and source of information on the topic can be found for example in [1, 50]. Again, we stress, the main difference is that the

channels are assigned to the cells in the planning phase of the communication network.

There is a rich literature on interval scheduling and selection problems (see [55, 94] and the references given therein for an overview). Our problem is similar to a setting with several machines where one wants to minimize the number of machines required to schedule all intervals. A version of this problem where intervals have to be scheduled within given time windows is studied in [34]. Inapproximability results for the variant with a discrete set of starting times for each interval are presented in [33].

In this thesis we study both the 1D-JBS and 2D-JBS problems. We present several results in different depth, focusing on the main contributions of the author. The results were published in [53] and [54] and appeared also in the thesis of Marc Nunkesser [83] and Gábor Szabó [96]. We show that arrow graphs are perfect and can be coloured optimally in $O(n \log n)$ time (Section 6.3.1). For the one-dimensional JBS problem with evenly spaced base stations we outline a polynomial-time dynamic programming algorithm (Section 6.3.2). The full proof appeared in the thesis of Marc Nunkesser [83]. For another special case of the one-dimensional JBS problem, where $3k$ users must be served by 3 base stations in k rounds, we also give a polynomial-time optimal algorithm (Section 6.3.3). For the general one-dimensional JBS problem, we outline that for any fixed k the question whether all users can be served in k rounds can be solved in $n^{O(k)}$ time (Section 6.3.4) (the full result appeared in the thesis of Gábor Szabó [96]). From the perfectness of arrow graphs and the existence of a polynomial-time algorithm for computing maximum weighted cliques in these graphs we derive a 2-approximation algorithm for JBS based on an LP relaxation and rounding (Section 6.3.5). We show that this result can also be generalized to a more realistic model where the interference region extends beyond the receiver (Section 6.3.6) (the result is discussed by Gábor Szabó in his thesis [96]). For the two-dimensional JBS problem, we outline that the problem is $\mathcal{NP}$-complete (Section 6.4.1). The complete discussion and the proof appeared in the thesis of Gábor Szabó [96]. The special case, deciding whether all users can be served in one round, is shown to be solvable in polynomial time (Section 6.4.2). We analyze an approximation algorithm for a constrained version of the problem,

and present lower bounds on the approximation ratios of some natural greedy algorithms for the general two-dimensional JBS problem (Section 6.4.3). The main contributions of the author are the special case when $3k$ users are served by 3 base stations in k rounds, the special case of serving users in one round, and lower bounds for greedy algorithms for 2D-JBS.

6.3 Case on the Line—1D-JBS

The one dimensional case, when all users and base stations are placed on a line, requires selecting an arrow for each user and colouring the resulting arrow graph with a minimum number of colours. Trying to understand when a selection of arrows leads to an arrow graph with small chromatic number, we first study the colouring problem for arrow graphs. We continue with some restricted cases—first we discuss the case when the neighboring base stations are evenly positioned on the line, and second we discuss the case with $3k$ users to be served in k rounds with 3 base stations. We continue with an optimum algorithm for the k-decision variant of 1D-JBS and present also a 2-approximation algorithm.

6.3.1 Arrow Graphs

Let us consider an arrow graph devised from the arrows that represent the choice of the base stations that serve the users. We are interested in the number of rounds that are necessary to serve the users by these arrows, i.e., we are interested in a colouring algorithm for arrow graphs. At the end of this section we present an $O(n \log n)$-time algorithm that optimally colours a given arrow graph. An alternative way of finding a polynomial optimum algorithm is to use some of the existing ones—one can show that the class of arrow graphs belongs to the class of perfect graphs. For perfect graphs polynomial time algorithms for the colouring problem exist.

Let us show that the class of arrow graphs belongs to the class of perfect graphs. We consider several graph classes, all perfect, and show their relationship with the arrow graphs. The class of PI* graphs consists of conflict graphs of triangles whose endpoints lie on two fixed parallel lines. An arrow graph can be represented as the intersection graph of triangles on two horizontal lines

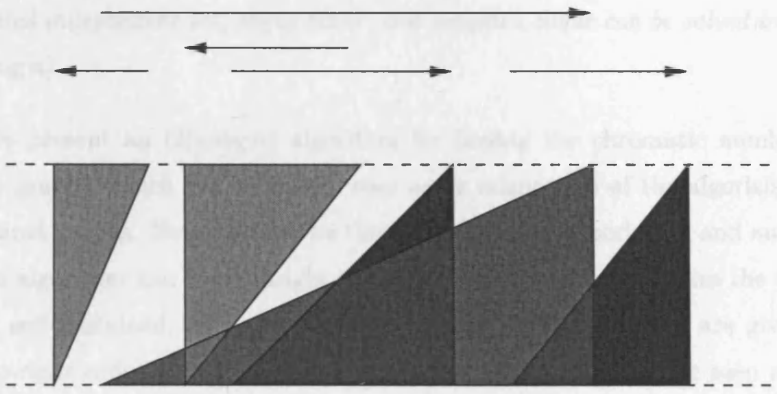


Figure 6.2: An arrow graph (top) and its representation as a PI* graph (bottom).

$y = 0$ and $y = 1$: Simply represent an arrow with left endpoint ℓ and right endpoint r that points to the right (left) as a triangle with corners $(\ell, 0)$, $(r, 0)$, and $(r, 1)$ (with corners $(r, 1)$, $(\ell, 1)$, and $(\ell, 0)$). It is easy to see that two triangles intersect if and only if the corresponding arrows are in conflict. See Figure 6.2 for an example. PI* graphs are a subclass of trapezoid graphs, which are the intersection graphs of trapezoids that have two sides on two fixed parallel lines. Trapezoid graphs are in turn a subclass of co-comparability graphs, a well-known class of perfect graphs. Therefore, the containment in these known classes of perfect graphs implies the perfectness of arrow graphs. Consequently, the size of a maximum clique in an arrow graph always equals its chromatic number.

We can also show the relationship between interval graphs and arrow graphs. Observe that an interval graph can be represented as an arrow graph simply by making each interval an arrow pointing to the right. Hence, the interval graphs are a subclass of arrow graphs.

As arrow graphs are a subclass of trapezoid graphs, we can apply known efficient algorithms for trapezoid graphs to arrow graphs. Felsner et al. [58] give algorithms with running-time $O(n \log n)$ for chromatic number, weighted independent set, clique cover, and weighted clique in trapezoid graphs with n vertices, provided that the trapezoid representation is given. Hence, the following theorem is a direct consequence of our discussion.

Theorem 6.2 *Arrow graphs are perfect. In arrow graphs chromatic number,*

weighted independent set, clique cover, and weighted clique can be solved in time $O(n \log n)$.

We present an $O(n \log n)$ algorithm for finding the chromatic number of arrow graphs, which can be indeed seen as an adaptation of the algorithm for trapezoid graphs. Nevertheless, we think the (explicit) description and analysis of the algorithm can bring insight into arrow graphs and also makes the thesis more self-contained. We assume for simplicity that the arrows are given in left-to-right order of their left endpoints (the sorting can also be seen as the first step of the algorithm). The algorithm scans the arrows from left to right in this sorted order. In step i it processes the i-th arrow a_i. The algorithm checks whether there are colours that have already been used and that can be assigned to a_i without creating a conflict. If there are such candidate colours, it considers for each such colour c the rightmost right endpoint r_c among the arrows that have been assigned colour c so far, and chooses for a_i a colour c for which r_c is rightmost (breaking ties arbitrarily). If there is no candidate colour, the algorithm assigns a new colour to a_i.

We show that this greedy algorithm produces an optimal colouring by showing that any optimal solution can be transformed into the solution produced by the algorithm.

Lemma 6.3 *Let C be an optimal colouring for a set of arrows $A = \{a_1, \ldots, a_n\}$. The colouring C can be transformed into the colouring produced by the greedy algorithm without introducing new colours.*

Proof. We prove the lemma by induction on the index of the arrows. The induction hypothesis is: *There exists an optimal colouring that agrees with the greedy colouring up to arrow $k-1$.* The induction start is trivial. In the k-th step let $C = (c_1, \ldots, c_n)$ be such an optimal colouring and let $H = (h_1, \ldots, h_n)$ be the greedy colouring, i.e. we have $h_1 = c_1, h_2 = c_2, \ldots, h_{k-1} = c_{k-1}$. We consider the colouring $C' = (c'_1, \ldots, c'_n)$ that is obtained from C by exchanging

the colours c_k and h_k for the arrows $a_k, \ldots, a_n$. More precisely, we define

$$c'_i = \begin{cases} c_i, & \text{if } i < k \text{ or } c_i \notin \{c_k, h_k\} \\ h_k, & \text{if } i \geq k \text{ and } c_i = c_k \\ c_k, & \text{if } i \geq k \text{ and } c_i = h_k. \end{cases}$$

By definition we have $c'_k = h_k$, and it remains to show that C' is a proper colouring and, therefore, the induction hypothesis is also true for k. If $c_k = h_k$ we have $C' = C$ which is a proper colouring. Otherwise, we have to show that all pairs of arrows a_i, a_j that are in conflict receive different colours in C', i.e., $c'_i \neq c'_j$. If $i, j < k$ or $k \leq i, j$ this is obvious by the fact that C is a (proper) colouring. Hence, we assume $i < k < j$; the case $j = k$ is implied by H being a proper colouring.

If h_k is a new colour, i.e., different from all of $c_1, \ldots, c_{k-1}$, then, because of the nature of the greedy algorithm, also c_k is a new colour. Hence, it is impossible that we have $c'_i = c'_j$.

Therefore h_k is an already used colour. Assume for a contradiction that we indeed have $c = c'_i = c'_j$ and the arrows a_i and a_j are in conflict. By the ordering of the arrows we know that a_i and a_k overlap. Observe that $c \in \{c_k, h_k\}$ because C is a colouring (i.e., before we made the changes, there was no conflict in C). This leaves us with two cases:

Case 1 $c = c_k$**:** Since C is a colouring, the arrows a_i and a_k are compatible, i.e., a_i is directed left and a_k is directed right. Such a configuration is depicted in Figure 6.3. By the definition of the greedy algorithm, we know that h_k is a colour of a compatible arrow. Since $h_k \neq c_k = c_i$, there must exist an arrow a_l, $l < k$, that is compatible with a_k and that ends not before a_i and that has colour h_k, i.e. $c_l = h_k$. Since a_j is in conflict with a_i (the head of a_j is within a_i), there is also a conflict between a_j and a_l. We have $c'_j = c_k$, implying $c_j = h_k$, hence we get the contradiction $c_j = h_k = c_l$ in the optimal colouring C.

Case 2 $c = h_k$**:** Because H is a colouring, a_i and a_k have to be compatible. Since a_i ends before a_k and is in conflict with a_j, also a_j is in conflict with a_k. Because $c'_j = h_k$ we know by definition of C' that $c_j = c_k$, hence there is a conflict in C, a contradiction. □

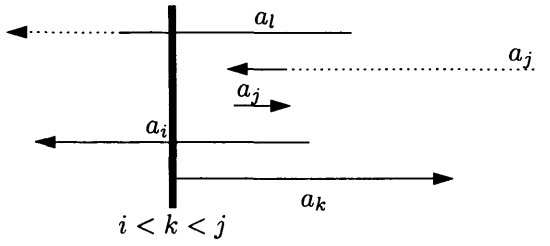


Figure 6.3: Possible configuration for the two cases. Dotted lines mean that the arrows could be extended.

The running time of the algorithm depends on the time the algorithm spends in every step on identifying an allowed colour that was previously assigned to an arrow with the rightmost right endpoint. By maintaining two balanced search trees (one tree for each direction of arrows) storing the most recently coloured arrows of the used colours (one arrow per colour) in the order of their right endpoints, we can implement this operation in logarithmic time. Together with Lemma 6.3 we get the following theorem.

Theorem 6.4 *The greedy algorithm optimally colours a given set of arrows* $\{a_1, a_2, \ldots, a_n\}$ *in* $O(n \log n)$ *time.*

6.3.2 Evenly Spaced Base Stations

On the way towards the general 1D-JBS problem, we consider the special case when all base stations are evenly spaced. From the practical point of view, this is a possible scenario, if one wants to cover, for example, a highway in an uninhabited area. We assume that m base stations $\{b_1, \ldots, b_m\}$ and n users $\{u_1, \ldots, u_n\}$ are on a line, where the distance between any two neighboring base stations is the same.

Let d denote the distance between two neighboring base stations. The base stations partition the line into two *rays* and a set of *intervals* $\{I_1, \ldots, I_{m-1}\}$. In this section we additionally require that no user to the left of the leftmost base station be further away from it than distance d, and that the same hold for the right end. Hence, we can introduce two additional intervals I_0 and I_m, where I_0 is the left neighbour of I_1 and I_m is the right neighbour of I_{m-1} on the line. Now all users lie in the union of the intervals. We outline an optimum

polynomial time algorithm that solves this special case of 1D-JBS. The complete proof appeared in [53] and in the thesis of Marc Nunkesser [83].

Observe that, since we consider only solutions that consist of user arrows only, every arrow intersects only two intervals—an arrow formed by a base station b_j intersects intervals I_{j-1} and I_j. The crucial property of the setting with evenly placed base stations is that among the optimum solutions there is always one that is *non-crossing*. A solution to 1D-JBS is said to be non-crossing if there are no two users u and w in the same interval I_i such that u is to the left of w, u is served from the right, and w from the left.

Lemma 6.5 *For instances of 1D-JBS with evenly spaced base stations, there is always an optimal solution that is non-crossing.*

Proof. Take any optimal solution s that is not non-crossing. We show that such a solution can be transformed into another optimal solution s' that is non-crossing. Let u and w be two users such that u and w are in the same interval, u is to the left of w, and u is served by the right base station b_r in round t_1 (forming an interference arrow a_r) and w is served by the left base station b_l in round t_2 (forming an interference arrow a_l); trivially, $t_1 \neq t_2$. Modify s in such a way that at t_1 base station b_r serves w and at t_2 base station b_l serves u (and modify the interference arrows a_l and a_r accordingly). This new solution is still feasible because first of all both the left and the right arrows a_l and a_r have become shorter. This implies that both a_l and a_r can only block fewer users. On the other hand, the head of a_l has moved left and the head of a_r has moved right. It is impossible that they are blocked now because of this movement: In t_1 this could only happen if there were some other arrows containing w, the new head of a_r. Such an arrow cannot come from the left, because then it would have blocked also the old arrow. It cannot come from b_r because b_r is busy. It cannot come from a base station to the right of b_r, because such arrows do not reach any point to the left of b_r (here we use the assumption that the rightmost user is no farther to the right of the rightmost base station than d, and that the base stations are evenly spaced). For t_2 the reasoning is symmetric. □

The selection of arrows in any non-crossing solution can be completely characterized by a sequence of $m-1$ *division points* d_i, such that the i^{th} division

point d_i is the index of the last user that is served from the left in the i^{th} interval I_i. (The case where all users in the i^{th} interval are served from the right is handled by choosing the i^{th} division point d_i as the index of the rightmost user to the left of the interval, or as 0 if no such user exists.) A brute-force approach could now enumerate over all possible $O(n^{m-1})$ division point sequences (*dps*) and colour the selection of arrows corresponding to each dps with the optimum greedy colouring algorithm of Section 6.3.1.

We can do better by exploiting the properties of the setting. Let $\chi_i(d_{i-1}, d_i)$ denote the cost (i.e., the minimum number of colours needed to colour the arrows) of an optimum solution of the setting with users $u_1, u_2, \ldots, u_{d_i}$ and base stations $b_1, b_2, \ldots, b_i$, such that b_i serves users $u_{d_{i-1}+1}, \ldots, u_{d_i}$. Let D_i denote all possible division points of interval I_i. Observe that we are interested in the value $\min_{d \in D_{m-1}} \chi_m(d, n)$, i.e., the cost of an optimum solution of the whole setting with U and B.

The value $\chi_{i+1}(d_i, d_{i+1})$ is obtained from some optimum solution that has $d_{i-1} \in D_{i-1}$ as a divsion point in interval I_{i-1}. Thus the arrows of b_{i+1} that are uniquely determined by the division points d_i and d_{i+1} can only be in conflict with arrows formed by b_i (that are determined by the division points d_{i-1} and d_i). Let $c_{i+1}(d_{i-1}, d_i, d_{i+1})$ denote the minimum number of colours needed to colour the arrows of b_{i+1} with respect to the colouring of arrows of b_i. Clearly, this number is equal to the minimum number of colours needed to colour the arrows of b_{i+1} with respect to the colouring of all arrows of $b_1, \ldots, b_i$. As all the colours for the arrows of b_i are different, we can assume, w.l.o.g., that the colours are $1, 2, \ldots, p_i$, where p_i is the number of arrows at base station b_i (notice that p_i is determined by the choice of d_{i-1} and d_i). Then $c_{i+1}(d_{i-1}, d_i, d_{i+1})$ can be computed by the greedy colouring algorithm using the arrows of b_i and b_{i+1} only. Then $\chi_{i+1}(d_i, d_{i+1}) = \max\{\chi_i(d_{i-1}, d_i), c_{i+1}(d_{i-1}, d_i, d_{i+1})\}$. Thus, to compute $\chi_{i+1}(d_i, d_{i+1})$ without knowing the optimum solution, we can recursively compute

$$\chi_{i+1}(d_i, d_{i+1}) = \min_{d_{i-1} \in D_{i-1}} \max\{\chi_i(d_{i-1}, d_i), c_{i+1}(d_{i-1}, d_i, d_{i+1})\}. \quad (6.1)$$

A dynamic programming approach starts with computing the values $\chi_1(0, d)$ for all $d \in D_1$. This can be done easily by the greedy colouring algorithm.

Then, setting $i = 1$ and increasing i one by one, the dynamic programming computes χ_{i+1} using the recurrence (6.1). The running time is dominated by the calculation of the $c(\cdot)$ values. There are $O(m \cdot n^3)$ such values, and each of them can be computed in time $O(n \log n)$ using the greedy colouring algorithm. An optimal solution can be found by tracking back the division points of the optimum cost solution and then colouring the arrows resulting from the division points.

Theorem 6.6 *The base station scheduling problem for evenly spaced base stations can be solved in time $O(m \cdot n^4 \log n)$ by dynamic programming.*

Note that the running time can also be bounded by $O(m \cdot u_{\max}^4 \log u_{\max})$, where $u_{\max}$ is the maximum number of users in one interval.

6.3.3 Serving $3k$ Users with 3 Base Stations in k Rounds

We study another special case of the general setting. We are given 3 base stations b_1, b_2 and b_3, and $3k$ users with k *far away* users among them, i.e., users to the left of b_1 or to the right of b_3 whose interference arrows contain b_2; the problem is to decide whether the users can be served in k rounds.

Observe that in every round every base station has to serve a user. We know that a far away user has to be served by its unique neighboring base station. Since the arrows of far away users contain b_2, all users between b_1 and b_2 have to be served in rounds in which far away users of b_3 are served, and all users between b_2 and b_3 have to be served in rounds in which far away users of b_1 are served. In particular, every round must be of one of the following two types:

Type 1: b_3 serves a far away user, b_2 serves a user between b_1 and b_2, and b_1 serves a user that is not a far away user.

Type 2: b_1 serves a far away user, b_2 serves a user between b_2 and b_3, and b_3 serves a user that is not a far away user.

For every user, it is uniquely determined whether she will be served in a round of Type 1 or Type 2.

The schedule can be constructed in the following way. Suppose we have k_1 far away users at b_1 and k_3 far away users at b_3, $k = k_1 + k_3$. First, we serve the

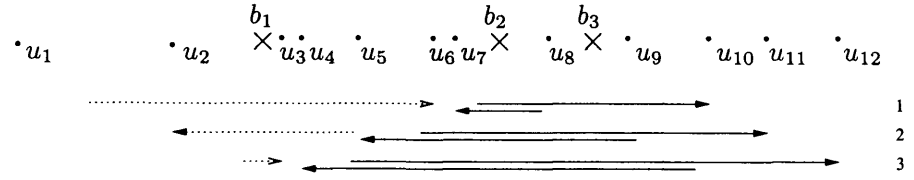


Figure 6.4: Far away users u_{10}, u_{11} and u_{12} are served by b_3 in rounds 1, 2 and 3, respectively. The solid arrows depict the selection of users for b_2 and b_3. The dotted arrows depict the resulting selection for b_1. Users u_1, u_8 and u_9 will be scheduled in a round of Type 2 (not shown).

far away users of b_3 in rounds $1, \ldots, k_3$ in the order of increasing distance from b_3. Next, we match the resulting arrows with arrows produced by b_2 serving users between b_1 and b_2 (cf. Figure 6.4). We apply a best fit approach. For every round $i = 1, 2, \ldots, k_3$, we find the user closest to b_2 that can be served together with the corresponding far away user served by b_3, and schedule the corresponding transmission in that round. Note that with this selection the size of the arrows of b_2 grows with the number of the round in which they are scheduled. Now we have to serve the remaining k_3 users (that are not far away users of b_1) with b_1. We use a best fit approach again, i.e., for every round $i = 1, 2, \ldots, k_3$, we schedule the user with maximum distance from b_1 (longest arrow) among the remaining users. This completes the description of how we obtain a schedule for the rounds in which a far away user of b_3 is served. The schedule for the remaining users (the far away users of b_1, and the users that must be scheduled in a round in which a far away user of b_1 is scheduled) can be found similarly, starting with the far away users of b_1.

We claim that if there exists a valid schedule with k rounds for the given instance of the problem, our algorithm produces a valid schedule. Without loss of generality, we consider only the schedule for the rounds in which the far away users of b_3 are served; the reasoning for the rounds with far away users of b_1 is analogous. Consider any valid schedule with k_3 rounds for the $3k_3$ users that must be served in rounds in which far away users of b_3 are served; call this schedule the *optimal schedule*. We can assume that the far away users are served by b_3 in the optimal schedule in the same round as in our schedule (the schedule produced by our algorithm). We will show that we can transform the optimal schedule into our schedule without losing validity.

First, we transform the optimal schedule in such a way that, in addition to the user served by b_3, also the user served by b_2 is the same as in our schedule in every round. Consider the first round i in which the optimal schedule does not serve the same user with b_2 as our schedule. Assume that b_2 serves user x in our schedule, but user $y \neq x$ in the optimal schedule. (Note also that the algorithm cannot get stuck while selecting a user to be served by b_2 in round i, since y is a candidate.) Note that y must be to the left of x, due to the best-fit rule our algorithm applies. The optimal schedule must serve x in some round $j \neq i$. If it serves x with b_2, we know that $j > i$, and we can simply exchange the users served by b_1 and b_2 in rounds i and j in the optimal schedule. If the optimal schedule serves x with b_1 in round j, we can let the optimum serve y with b_1 in round j and x with b_2 in round i. In both cases, we have transformed the optimal schedule so that it also serves x with b_2 in round i, without losing validity. Repeating this transformation, we obtain an optimal schedule that serves the same users with b_2 and b_3 as our schedule in every round.

It remains to handle the users served by b_1. Consider the first round i in which the optimal schedule differs from our schedule. Assume that b_1 serves user w in our schedule, but user z in the optimal schedule. Note that, by the best-fit approach, the arrow for (w, b_1) must be at least as long as the arrow for (z, b_1). The optimal schedule must serve w in some later round, say, round j, also with b_1. We can change the optimal schedule by letting b_1 serve user w in round i and user z in round j; this does not affect the validity of the schedule, since the arrow for z is not longer than the arrow for w (and thus round j remains valid) and since the arrow produced by the transmission of b_2 is shorter in round i than in round j (and thus serving w in round i must be valid if serving w in round j was valid). By repeating this transformation, we have transformed the optimal schedule into our schedule, without losing validity. This shows that our algorithm produces a valid schedule if one exists. The time complexity of our algorithm is dominated by the time for sorting the users, $O(n \log n)$. After the sorting, the schedule can be computed in linear time.

Theorem 6.7 *For the setting with 3 base stations and $3k$ users on a line with k far away users, there is an $O(n \log n)$ time algorithm that computes a valid schedule with k rounds if there is one.*

6.3.4 Exact Algorithm for the k-Decision Problem

In this section we present an exact algorithm for the decision variant k-1D-JBS of the 1D-JBS problem: For given k and an instance of 1D-JBS, decide whether all users can be served in at most k rounds. We present an algorithm for this problem that runs in $O(m \cdot n^{2k+1} \log n)$ time.

As shown in Section 6.3.1, the arrow graphs are perfect and therefore the size of the maximum clique of an arrow graph equals its chromatic number. We use this fact in our algorithm, which we call $A_{k-\mathrm{JBS}}$. The main idea is to divide the problem into subproblems, one for each base station, and then combine the partial solutions to a global one.

The set of all arrows that are considered in the selection phase is the set of all user arrows. For base station b_i, the corresponding subproblem S_i considers only those user arrows for which the arrow intersects b_i or the alternative arrow intersects b_i, i.e., if a_l and a_r are user arrows of a user, then a_l and a_r are considered at S_i if and only if a_r or a_l intersect b_i. Let us denote this set of arrows A_i. We call S_{i-1} and S_{i+1} *neighbours* of S_i. A solution to S_i consists of a feasible selection of arrows from A_i of cost no more than k, i.e., the selection can be coloured with at most k colours.

We want to find all solutions to every S_i and subsequently select one appropriate solution from every S_i and combine these selected solutions into a global solution. To find all solutions we enumerate all possible selections that can lead to a solution in k rounds. For S_i we store all such selections $\{s_i^1, \ldots, s_i^{n_i}\}$ in a table T_i. We only need to consider selections in which at most $2k$ arrows intersect the base station b_i. All other selections need more than k rounds, because they must contain more than k arrows pointing in the same direction at b_i. Therefore, the number of entries of T_i is bounded by $\sum_{j=0}^{2k} \binom{n}{j} = O(n^{2k})$. We need $O(n \log n)$ time to evaluate a single selection with the greedy colouring algorithm of Section 6.3.1. Selections that cannot be coloured with at most k colours are marked as irrelevant and ignored in the rest of the algorithm. We build up the global solution by choosing a set of feasible selections $s_1, \ldots, s_m$ in which all neighbours are *compatible*, i.e., they agree on the selection of common arrows. It is easy to see that in such a global solution all subsolutions are pairwise compatible.

We can find such a set of compatible neighbours by going through the tables in left-to-right order and marking every solution in each table as *valid* if there is a compatible, valid solution in the table of its left neighbour, or as *invalid* otherwise. A solution s_i marked as valid in table T_i thus indicates that there are solutions $s_1, \ldots, s_{i-1}$ in $T_1, \ldots, T_{i-1}$ that are compatible with it and pairwise compatible. In the leftmost table T_1, every feasible solution is marked as valid. When the marking has been done for the tables of base stations $b_1, \ldots, b_{i-1}$, we can perform the marking in the table T_i for b_i in time $O(n^{2k+1})$ as follows. First, we go through all entries of the table T_{i-1} and, for each such entry, in time $O(n)$ discard the part of the selection that affects pairs of user arrows that intersect only b_{i-1} but not b_i, and enter the remaining selection into an intermediate table $T_{i-1,i}$. The table $T_{i-1,i}$ stores entries for all selections of arrows from pairs of user arrows intersecting both b_{i-1} and b_i. An entry in $T_{i-1,i}$ is marked as valid if at least one valid entry from T_{i-1} has given rise to the entry. Then, the entries of T_i are considered one by one, and for each such entry s_i the algorithm looks up in time $O(n)$ the unique entry in $T_{i-1,i}$ that is compatible with s_i to see whether it is marked as valid or not, and marks the entry in T_i accordingly. If in the end the table T_m contains a solution marked as valid, a set of pairwise compatible solutions from all tables exists and can be retraced easily.

The overall running time of the algorithm is $O(m \cdot n^{2k+1} \cdot \log n)$. There is a solution to k-1D-JBS if and only if the algorithm finds such a set of compatible neighbours.

Lemma 6.8 *There exists a solution to k-1D-JBS if and only if $A_{k-\mathrm{JBS}}$ finds a set of pairwise compatible solutions.*

Proof. ($\Rightarrow$) Every arrow intersects at least one base station. A global solution directly provides us with a set of compatible subsolutions $\Sigma_{\mathrm{opt}} = \{s_1^{\mathrm{opt}}, s_2^{\mathrm{opt}}, \ldots, s_m^{\mathrm{opt}}\}$. Since the global solution has cost at most k, so have the solutions of the subproblems. Hence, the created entries will appear in the tables of the algorithm and will be considered and marked as valid. Thus there is at least one set of compatible solutions that is discovered by the algorithm.

($\Leftarrow$) We have to show that the global solution constructed from the partial ones has cost at most k. Suppose for a contradiction that there is a point

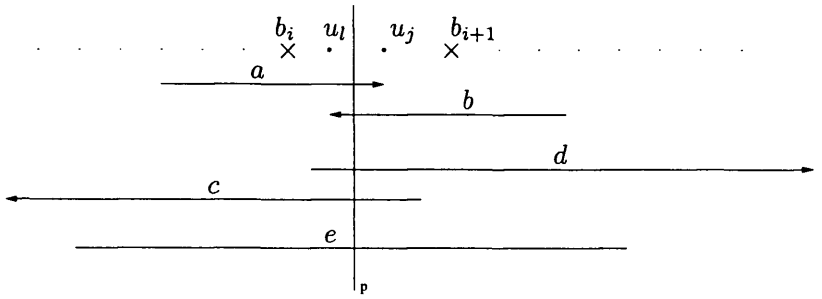


Figure 6.5: A clique of size greater than k at point p together with the different types of arrows at this point.

p where the clique size is bigger than k and therefore bigger than the clique at b_i (the left neighboring base station of p) and the clique at b_{i+1} (the right neighboring base station of p). We divide the arrows intersecting point p into 5 groups as in Figure 6.5. Arrows of type a (b) have their head between b_i and b_{i+1} and their tail to the left (right) of b_i (b_{i+1}). Arrows of type c (d) have their tail between b_i and b_{i+1} and their head to the left (right) of b_i (b_{i+1}). Finally, type e arrows intersect both b_i and b_{i+1}. For the clique at p to be bigger than that at b_i some arrows not considered at b_i have to create conflicts. The only such arrows (considered at b_{i+1} but not at b_i) are of type d (observe that arrows of type a, b and e are considered both at the table for b_i and at the table for b_{i+1}). If their presence increases the clique size at p, then no type c arrow can be in the maximum clique at p (observe that arrows of type c and d are compatible). Therefore, the clique at p cannot be bigger than the clique at b_{i+1}, a contradiction. □

Theorem 6.9 *Problem k-1D-JBS can be solved in time* $O(m \cdot n^{2k+1} \cdot \log n)$.

6.3.5 Approximation Algorithm

In this section we present a 2-approximation algorithm for 1D-JBS based on a relaxed (integer) linear program and a subsequent straightforward rounding. The main idea is to express the fact that each user u is served by her left of right neighbouring base station via two indicator variables that sum up to 1. The (deterministic) rounding then follows the choice of the indicator variable that is bigger than one half.

Let A denote the set of all user arrows of the given instance of 1D-JBS. The selection problem of 1D-JBS asks for a feasible selection $A_{\text{sel}} \subseteq A$ that minimizes the chromatic number of its arrow graph $G(A_{\text{sel}})$ (among all feasible selections). As for perfect graphs the chromatic number equals the clique number, we set our optimization goal to be the clique number of $G(A_{\text{sel}})$. The 1D-JBS problem can then be expressed as an integer linear program (ILP). We introduce two indicator variables l_i and r_i for every user i that indicate whether she is served by the left or by the right base station, i.e., if the user's left or right user arrow is selected into A_{sel}. Moreover, we ensure by the constraints that no cliques in $G(A_{\text{sel}})$ are large and that each user is served. The ILP formulation is as follows:

$$\min \quad k \tag{6.2}$$

$$\text{s.t.} \quad \sum_{l_i \in C} l_i + \sum_{r_i \in C} r_i \leq k \qquad \forall \text{ cliques } C \text{ in } G(A) \tag{6.3}$$

$$l_i + r_i = 1 \qquad \forall i \in \{1, \ldots, |U|\} \tag{6.4}$$

$$l_i, r_i \in \{0, 1\} \qquad \forall i \in \{1, \ldots, |U|\} \tag{6.5}$$

$$k \in \mathbb{N} \tag{6.6}$$

The natural LP relaxation is obtained by allowing $l_i, r_i \in [0, 1]$ and $k \geq 0$. Given a solution to this relaxation, we can use a rounding technique to get an assignment of users to base stations that has cost at most twice the optimum, i.e., we obtain a 2-approximation algorithm. Let us denote by *opt* the optimum number of colours needed to serve all users. Then *opt* is at least k, because the optimum integer solution is also a feasible fractional solution (i.e., a feasible solution to the relaxed LP). We can construct a feasible solution for the integer LP from a solution to the relaxed problem by rounding $l_i := \lfloor l_i + 0.5 \rfloor$, $r_i := 1 - l_i$. Before the rounding the size of every (fractional) clique is at most k; afterwards the size can double in the worst case (because the value of each individual variable can at most double). Therefore, the cost of the rounded solution is at most $2k \leq 2opt$. Figure 6.6 gives an example where the cost of an optimal solution to the relaxed program is indeed smaller than the cost of an optimal integral solution by a factor arbitrarily close to 2.

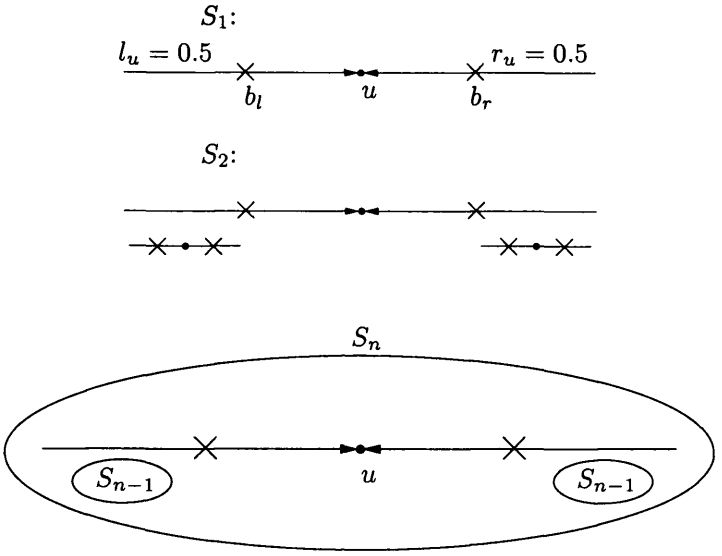


Figure 6.6: Setting S_1 with two base stations b_l and b_r and one user u in between, where both the solution of the ILP and the solution of the LP relaxation have cost 1. S_2 is constructed recursively by adding to the setting of S_1 two (scaled) copies of S_1 in the tail positions of the arrows. Here the cost of the relaxed LP is 1.5 and the integral cost is 2. The recursive approach for general n is shown in the bottom of the figure. Using setting S_1 and putting two (properly scaled) settings S_{n-1} as depicted in the picture, we get a setting S_n where $k^*(n)$, the cost of the LP relaxation for S_n, is $0.5 + k^*(n-1) = 0.5 + n/2$, whereas the cost of the ILP is n.

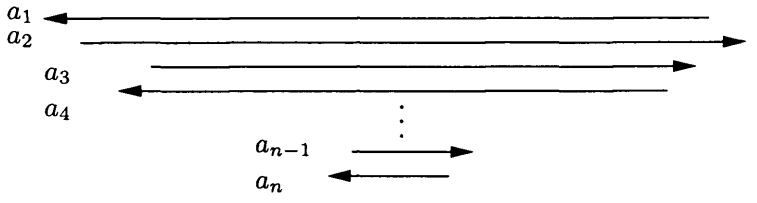


Figure 6.7: Example of an arrow graph with an exponential number of maximum cliques. For every choice of arrows from a compatible pair (a_{2i-1}, a_{2i}) we get a clique of size $n/2$, which is maximum. The arrow graph can arise from a 1D-JBS instance with two base stations in the middle and $n/2$ users on either side.

One issue that needs to be discussed is how the relaxation can be solved in time polynomial in n and m, as there can be an exponential number of constraints of type (6.3). (Figure 6.7 shows that this can really happen. The potentially exponential number of maximal cliques in arrow graphs distinguishes them from interval graphs, which have only a linear number of maximal cliques.) Fortunately, we can still solve such an LP in polynomial time with the ellipsoid method of Khachiyan [75] applied in a setting similar to [67]. This method only requires a *separation oracle* that provides us for any values of l_i, r_i with a violated constraint, if one exists. Thus, we need to find a separation oracle that runs in polynomial time. It is easy to check for a violation of constraints (6.4) and (6.5). For constraints (6.3), we need to check if for given values of l_i, r_i the maximum weighted clique in $G(A)$ is smaller than k. By Theorem 6.2 this can be done in time $O(n \log n)$. Summarizing, we get the following theorem:

Theorem 6.10 *There is a polynomial-time 2-approximation algorithm for the 1D-JBS problem.*

6.3.6 Different Interference Models

We discuss here briefly another possible interference model that can arise in the modelling phase of 1D-JBS. So far we have been dealing with the discrete interference model where the interference region has no effect beyond the targeted user. One step towards a more realistic model is to consider the interference region, produced by a base station sending a signal to a user, to span also beyond the targeted user. For the 1-dimensional case this can be modeled by using *interference segments* with the user somewhere between the endpoints of this segment (the small black circles in the segments in Figure 6.8) and the base station in the middle of the segment. The conflict graph of such interference segments is another special case of trapezoid graphs. An example of interference segments together with the corresponding conflict graph represented as a trapezoid graph is in Figure 6.8.

The transformation from interference segments to trapezoids is straightforward, if we consider an interference segment with a user between its endpoints as two "interference" arrows pointing to the user from the left and from the right. Then the two resulting triangles from the transformation of interference

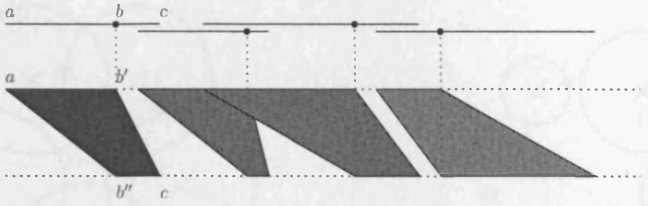


Figure 6.8: Example of interference segments for base stations and users (top; only users are depicted) and the trapezoid graph as the conflict graph of the interference segments (bottom).

arrows to trapezoid graphs (Section 6.3.1) form the trapezoid that corresponds to the interference segment. Hence, also the "interference segment graphs" are perfect and for this interference generalization of the 1D-JBS problem we have a 2-approximation algorithm using the same technique as in Section 6.3.5.

6.4 The General Case—2D-JBS

We discuss now 2D-JBS, the two-dimensional version of the joint base station scheduling problem. As the decision variant of 2D-JBS (the k-2D-JBS problem, which asks whether we can serve all users in k rounds) is shown to be $\mathcal{NP}$-complete for $k \geq 3$ (main ideas are outlined in Section 6.4.1), a natural quest for approximation algorithms tries to exploit the 2-approximation algorithm of the 1D-JBS—set up an ILP of the problem, and use some rounding of the LP formulation obtained via relaxation of the constraints of the ILP. First of all it is not clear what the rounding would be, because every user can possibly have more than 2 base stations that are considered to serve the user. Moreover, the conflict graphs of interference disks need not be perfect, as can be seen if Figure 6.9. Thus, an approximation algorithm based on an LP formulation similar to the one for 1D-JBS is more than doubtful. We give, however, an approximation algorithm whose approximation ratio depends solely upon the geometry of the setting—the minmum distance between any two base stations, and the maximum allowed radius of an interference disk. To the end we discuss several greedy algorithms and show lower bounds on their approximation ratio.

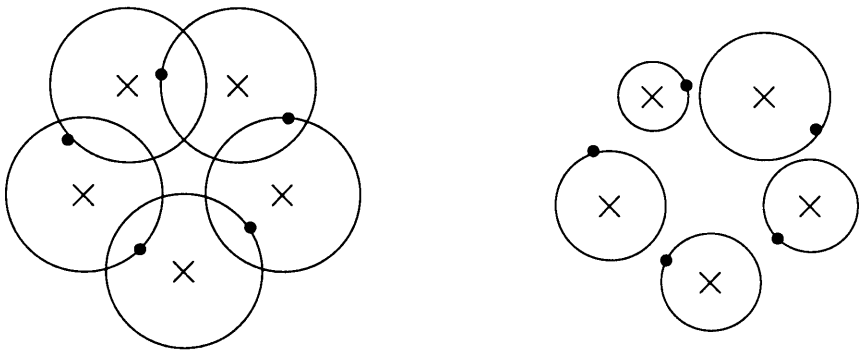


Figure 6.9: A cycle of length 5 in the conflict graph of interference disks (left). Note that it is not clear that the selection problem will ever produce this selection. Another assignment of users to base stations (an optimum one) does not lead to such a situation (right).

6.4.1 $\mathcal{NP}$-Completeness of the k-2D-JBS Problem

In this section we outline very briefly that the decision variant (k-2D-JBS) of 2D-JBS is $\mathcal{NP}$-complete. The complete proof appeared in [53] and in the thesis of Gábor Szabó [96].

The problem, k-2D-JBS, is shown to be $\mathcal{NP}$-complete by a reduction from the general k-colourability problem. Our reduction is inspired by the reduction used in [66], which proofed $\mathcal{NP}$-completeness of k-colourability of unit disk graphs (conflict graphs of unit discs in the plane). Given any graph G, we construct in polynomial time a corresponding 2D-JBS instance that can be scheduled in k rounds if and only if G is k-colourable. We use an embedding of G into the plane which allows us to replace the edges of G with suitable base station chains with several users in a systematic fashion such that the k-colourability is preserved. The result that we obtained is the following.

Theorem 6.11 *The k-2D-JBS problem is $\mathcal{NP}$-complete for any fixed $k \geq 3$. The colouring problem of k-2D-JBS is $\mathcal{NP}$-complete for any fixed $k \geq 3$.*

6.4.2 Base Station Assignment for One Round

The previous section showed that even if we have users assigned to the base stations as they are served in an optimal solution, the k-2D-JBS problem cannot be solved in polynomial time unless $\mathcal{P}$=$\mathcal{NP}$. Now we consider the complementary problem: knowing for every user the round in which the user is served in a

particular optimal solution, find an assignment of the users to the base stations such that a valid optimal schedule is obtained. We will see that this problem is solvable in polynomial time, which actually shows that, in some sense, the assignment problem is easier than the colouring one.

Knowing for every user the round in which she is served, we can consider every round as an independent problem by taking into account only the users scheduled in the corresponding round. We study therefore the problem of deciding whether we can serve all the users in one round.

We start with a simple observation that is valid also for the general 2D-JBS problem. Consider an *empty* disk $d = d(b, u)$ in the given setting of users U and base stations B, i.e., a disk containing only user u. We claim that every optimal solution can use d to serve u without changing the disks selected for other users. To see this, imagine u is served by some other base station b' in an optimal solution. Then b has to be idle in this round, because u is the closest user of b and therefore serving anybody else would block u. Moreover, d does not contain any other user, therefore we can serve u with b instead of b' in the same round without blocking anybody else.

We can adapt this idea to our setting. Suppose we can serve all the users in one round. Observe that every optimal solution is a set of empty disks. Consider the set of all possible empty disks formed by B and U. For every base station there is at most one empty disk, determined by the closest user to the base station (there is no empty disk at b if there are 2 or more closest users at the same distance from b). For every user $u \in U$ there must be at least one empty disk serving u (e.g., the disk from an optimal solution). We know that we can use any of these disks in any optimal solution, therefore we can pick any empty disk into our optimal solution. So our algorithm simply computes all empty disks and then selects for each user u an arbitrary empty disk serving u; if some user u does not have an empty disk, the users cannot be served in one round.

The algorithm can clearly be implemented to run in polynomial time. Using standard techniques from computational geometry [43], e.g., computing in $O(n \log n)$ time a Voronoi diagram for the user points and then a point location data structure so that the closest user of a base station can be determined in $O(\log n)$ time, we obtain a running-time of $O((m + n) \log n)$.

Lemma 6.12 *The problem of deciding whether all users in a given 2D-JBS instance can be scheduled in one round can be solved in time* $O((n+m)\log n)$.

Corollary 6.13 *Given the sets* $U_1, \ldots, U_r$ *of users scheduled in rounds* $1, \ldots, r$ *in an optimal solution, the problem of assigning base stations to the users such that we obtain a valid schedule of users* U_i *in round* i *can be solved in polynomial time.*

6.4.3 Approximation algorithms

We investigate possible algorithms for the general 2D-JBS problem. We discuss first a scenario of limited transmission power, which directly translates into radius-bounded interference disks. Then we consider several greedy algorithms and discuss their limits by showing lower bounds on their approximation ratio.

Bounded geometric constraints

We consider instances where the base stations are at least a distance Δ from each other and have limited power to serve a user, i.e., every base station can serve only users that are at most $R_{\max}$ away from it. We also assume that for every user there is at least one base station that can reach the user. We present a simple algorithm achieving an approximation ratio depending only on the parameters Δ and $R_{\max}$.

Consider the following greedy approach: For round $1, 2, \ldots$, the algorithm repeatedly picks an arbitrary user base-station pair (u, b), where u is an unserved user, such that the transmission from b to u can be added to the current round without creating a conflict. If no such user base-station pair exists, the next round starts. The algorithm terminates when all users have been served.

We can analyze the approximation ratio achieved by this greedy algorithm as follows. Let k be the number of rounds that the algorithm needs to schedule all users. Let u be a user served in round k, and let b be the base station serving u. Since u was not served in rounds $1, 2, \ldots, k-1$, we know that in each of these rounds, at least one of the following is true:

- b serves another user $u' \neq u$.

- u is contained in an interference disk $d(b', u')$ for some user $u' \neq u$ that is served in that round.
- b cannot transmit to u because the disk $d(b, u)$ contains another user u' that is served in that round.

In each of these cases, we see that a user u' is served, and that the distance between u and u' is at most $2R_{\max}$ (since every interference disk has radius at most $R_{\max}$). Therefore, the disk with radius $2R_{\max}$ centered at u contains at least k users (including u). Let B' be the set of base stations that serve these k users in the optimal solution. The base stations in B' must be located in a disk with radius $3R_{\max}$ centered at u. As any two base stations are separated by a distance of Δ, we know that disks with radius $\Delta/2$ centered at base stations are interior-disjoint. Furthermore, the disks with radius $\Delta/2$ centered at the base stations in B' are all contained in a disk with radius $3R_{\max} + \Delta/2$ centered at u. Therefore, we have

$$|B'| \leq \frac{(3R_{\max} + \Delta/2)^2\pi}{(\Delta/2)^2\pi} = \frac{(6R_{\max} + \Delta)^2}{\Delta^2}.$$

Furthermore, we know that the optimal solution needs at least $k/|B'|$ rounds. This yields the following theorem.

Theorem 6.14 *There exists an approximation algorithm with approximation ratio* $(\frac{6R_{\max}+\Delta}{\Delta})^2$ *for 2D-JBS in the setting where any two base stations are at least* Δ *away from each other and every base station can serve only users within distance at most* $R_{\max}$ *from it.*

General 2D-JBS

We present lower bounds on the approximation ratio of three natural greedy approaches for the general 2D problem. The algorithms proceed round by round and use certain greedy rules to determine the user base-station pairs to be scheduled in the current round.

First, consider the greedy algorithm that serves as many users as possible in every round, i.e., in each round it chooses a maximum independent set in the conflict graph of all interference disks corresponding to user base-station

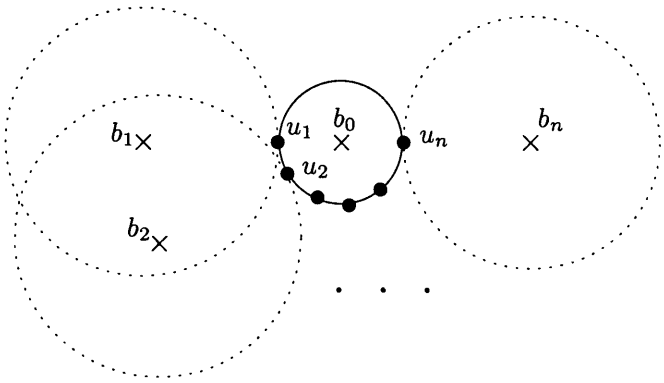


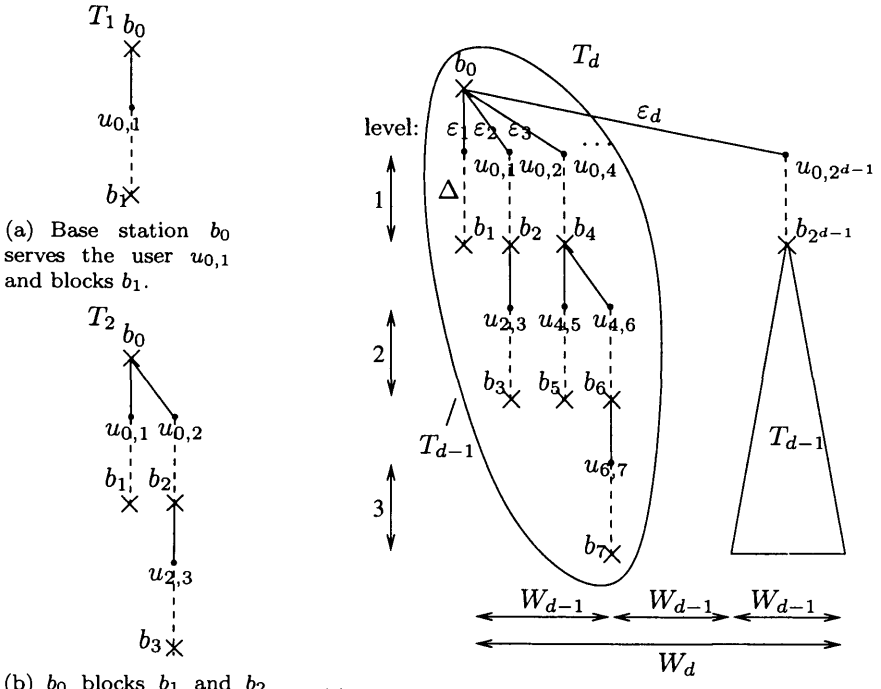
Figure 6.10: A greedy approach serves n users placed on a common interference disk in n time steps. An optimum algorithm can serve the users in one time step by assigning u_i to base station b_i, which lies on a halfline determined by b_0 and u_i.

pairs involving unserved users. We refer to this algorithm as the *maximum-independent-set* algorithm. The maximum independent set problem is $\mathcal{NP}$-hard in general graphs, and we do not know its complexity for conflict graphs of interference disks; nevertheless, we believe that it is interesting to determine the approximation ratio that can be achieved using this greedy approach, even if it is not clear whether the approach can actually be implemented in polynomial time.

Furthermore we consider greedy algorithms that, in each round, repeatedly choose an interference disk of an unserved user that can be scheduled in the current round without creating a conflict. The algorithm that chooses among the interference disks of all unserved users a disk of smallest radius is the *smallest-disk-first* algorithm, and the algorithm choosing a disk containing the fewest other unserved users is the *fewest-users-in-disk* algorithm.

For the smallest-disk-first algorithm, there is a simple example (see Figure 6.10) showing that this approach has approximation ratio $\Omega(n)$. All the user points in this example, however, lie on the perimeter of a common interference disk. As such a configuration appears to be a rare case in practice, this leads us to consider instances of 2D-JBS in general position, defined as follows.

Definition 6.1 *We say that sets of points* $(U, B) \subset \mathbb{R}^2 \times \mathbb{R}^2$ *are in* general position *if no two points from* U *lie on a circle centered at some point in* B.



(a) Base station b_0 serves the user $u_{0,1}$ and blocks b_1.

(b) b_0 blocks b_1 and b_2 blocks b_3 in the first round. In the second round, b_0 serves $u_{0,2}$ and blocks b_2.

(c) Recursive construction of instances where greedy algorithms use b_0 to serve the d users directly below b_0 in d rounds.

Figure 6.11: Outline of the construction of instances that greedy algorithms cannot handle efficiently

For points in general position we can show a lower bound of $\Omega(\log n)$ on the approximation ratio of all three considered greedy algorithms. Our construction of 2D-JBS instances leading to this lower bound is sketched in Figure 6.11. Conceptually, the arrangement of users and base stations can be viewed as a tree, where the edges of the tree are indicated by solid and dashed lines in the figure. The dashed lines represent a distance Δ, while the solid lines represent much shorter distances $\varepsilon_i \ll \Delta$. The structure of the tree is such that, after contracting the dashed edges, one obtains a binomial tree. The base stations in the figure are labelled by $b_0, b_1, \ldots$ in the order of a depth-first search traversal of the tree. Users are vertices of degree 2 and are adjacent to two base stations; our convention is to label a user adjacent to b_i and b_j with $u_{i,j}$.

The base station at the root of the tree is b_0. The aim of the construction is to have the greedy algorithms use b_0 to serve the d users $u_{0,1}, u_{0,2}, u_{0,4}, \ldots, u_{0,2^i}, \ldots, u_{0,2^{d-1}}$, which are the children of b_0, in d consecutive rounds,

whereas they can be served in a single round by an optimum algorithm by using the d base stations $b_1, b_2, b_4, \ldots, b_{2^i}, \ldots, b_{2^{d-1}}$. We show how to construct a tree T_d with this property for every value of d. In the following, the term "greedy algorithm" should be taken to refer to the smallest-disk-first algorithm. Afterwards we will discuss how the two other greedy algorithms behave on the constructed instances.

For $d = 1$, we simply put the user $u_{0,1}$ between base stations b_0 and b_1 at a position closer to b_0, see Figure 6.11(a). For $d = 2$, we have to ensure that user $u_{0,2}$ is not served by b_2 in the first round. Therefore we occupy b_2 in the first round by another user $u_{2,3}$, which will be selected by the greedy algorithm (see Figure 6.11 (b)).

Similarly, for general d, we construct the tree T_d from two trees T_{d-1}, see Figure 6.11(c). We want to ensure that the user $u_{0,2^{d-1}}$ is not served in the first $d-1$ rounds; hence, we make base station $b_{2^{d-1}}$ (which could serve $u_{0,2^{d-1}}$ from below) busy for $d-1$ rounds by creating a copy of T_{d-1} rooted at $b_{2^{d-1}}$. Equivalently, the tree T_d can be viewed as putting trees $T_0, T_1, T_2, \ldots, T_{d-1}$ below the users $u_{0,1}, \ldots, u_{0,2^{d-1}}$. Note also that every base station in the tree T_d is the root of a tree T_k for some k. With this construction, we obtain a tree with levels of users and base stations, where level i consists of the users and base stations whose distance to the root (in terms of the number of edges) is of the form $2i-1$ or $2i$. All users in a level have the same y-coordinate, and the same holds for the base stations in a level. We set $\varepsilon_1 < \varepsilon_2 < \ldots < \varepsilon_d$ and Δ in such a way that $\varepsilon_d \ll \Delta$ and $W_d \ll \varepsilon_1$, where W_d is the width of the tree T_d (see Figure 6.11). When constructing a tree T_d from two trees T_{d-1}, we always leave a free space of width W_{d-1} between these two trees. We adjust Δ such that serving a user $u_{i,j}$ from level ℓ by base station b_j blocks all the users on level $\ell+1$. Also, we adjust W_d such that a disk centered at any user u from level i with radius ε_1 contains all other users from level i. Figure 6.12 depicts the desired property. To ensure that the points are in general position, we can adjust the x-coordinates of all users by a small perturbation that is much smaller than any of the values W_i.

The tree is constructed in such a way that the greedy algorithm will pick the disks formed by base stations and users at distance ε_i in the i-th round,

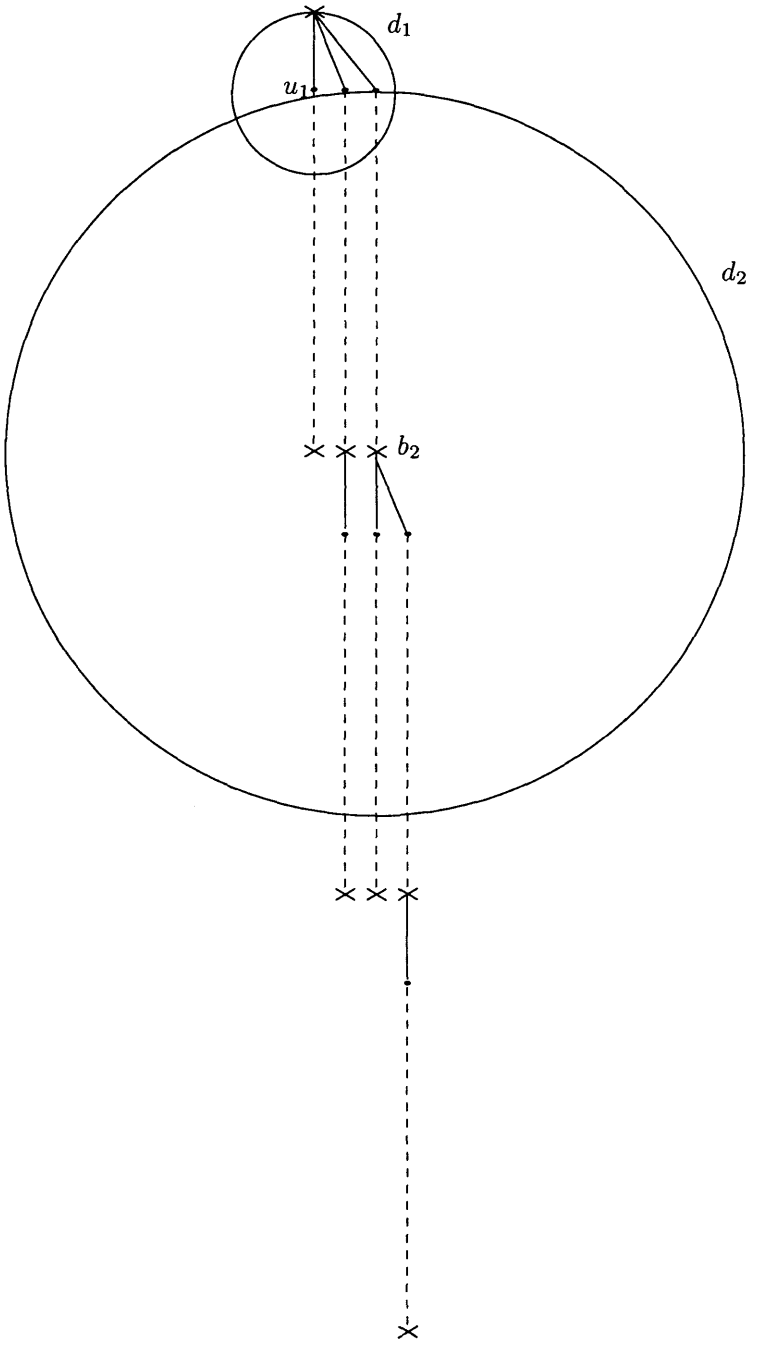


Figure 6.12: Possible realization of the "bad" instance for the greedy algorithms. The disk d_1 centered at u_1 contains all other users that are on the same level as u_1 is. The disk d_2 centered at b_2 blocks all users on the level below b_2.

$1 \leq i \leq d$. Let G_i denote the set of these disks chosen in the i-th round. An optimal algorithm can serve all users on some level ℓ in one round by using the base stations from below, at the expense of blocking all users on level $\ell + 1$. Thus, it can serve all users in two rounds by serving the odd levels in one round and the even levels in the other. Consequently, the approximation ratio of the greedy algorithm is $\Omega(d)$. The number n_d of users in the tree T_d can be calculated by solving the simple recursion $n_d = 2n_{d-1} + 1$ (which follows from the recursive construction of the tree), where $n_1 = 1$. This gives $n = n_d = 2^d - 1$ users and thus $d = \log(n + 1)$.

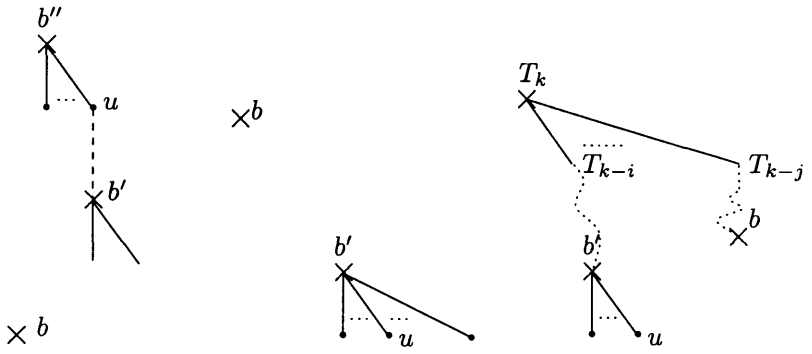
As every disk from G_i does not contain any unserved user, G_i is also a possible choice for the fewest-users-in-disk greedy algorithm. Thus we have shown the following.

Lemma 6.15 *There are instances (U, B) of 2D-JBS in general position for which the smallest-disk-first greedy algorithm and the fewest-users-in-disk greedy algorithm have approximation ratio $\Omega(\log n)$, where $n = |U|$.*

Furthermore, we can show that G_t is a maximum independent set among the interference disks of all unserved users after the first $t-1$ rounds of the algorithm, implying that the greedy algorithm maximizing the number of served users in each round can produce the same solution as the two other greedy algorithms on this instance.

To show this, we consider an arbitrary maximum independent set M_t among the interference disks of all unserved users (after serving $G_1, \ldots, G_{t-1}$ in previous rounds) and show $|M_t| = |G_t|$. The inequality $|M_t| \geq |G_t|$ is obvious. To show the second inequality, we proceed as follows. Define an *active* base station as a base station with unserved neighboring user(s) below the base station. A base station such that all neighboring users below it have been served is called *passive*. We present several transformations on M_t (exchanging one interference disk by another) that preserve the size of M_t and the independence of the disks and lead to a new maximum independent set where only active base stations are used to serve users. This shows the desired inequality $|M_t| \leq |G_t|$, since in G_t all active base stations serve a user.

First, we transform M_t so that base stations serve only neighboring users. Suppose there is user u on level i that is served by a non-neighboring base



(a) Base station b serves user u from below. If base station b' is passive, then the upper neighbour b'' of u must be idle.

(b) Base station b serves user u. The base station is to the left and above of u. Upper neighboring base station b' of u has to be idle.

(c) Base station b is to the right and above of u.

Figure 6.13: Transformations for the maximum-independent-set algorithm

station b in M_t. We perform a case study depending on the position of b.

1. Suppose b is below u (i.e., from level $i, i+1, \ldots$). Consider the bottom neighbour b' of u (from level i, cf. Figure 6.13(a)). We claim that b' is idle (i.e., b' does not serve any user); b' cannot serve any user above b', because this would block u (u is the closest user above b'); b' cannot serve any of its bottom neighbours (from level $i+1$) because all the users from level $i+1$ are blocked by the disk $d(b,u)$; b' cannot serve any of the users from a level greater than $i+1$ because this would block u. Therefore we can serve u by b' (no new interference is created because the disk $d(b',u)$ blocks only the users on level $i+1$, which were blocked also by the disk $d(b,u)$).

2. Suppose b is above u (i.e., from level $i-1, i-2, \ldots$) and to the left of u (see Figure 6.13(b)). Consider the upper neighbour b' of u (b' is from level $i-1$; note also that b must be to the left of b', by construction of the instance); b' cannot serve any user above it, because that would block u; b' cannot serve any user that is to the right of u and below b' for the same reason; b' cannot serve any user from level greater than i for the same reason; b' cannot serve any neighboring user to the left of u because these users are blocked by the disk $d(b,u)$; b' cannot serve any user from

level i to the left of its leftmost neighbour because then it would block u (as the horizontal distance is at least the width of the tree rooted at b'). Therefore, b' is idle and can serve u instead of b without introducing any new interference.

3. Suppose b is above u (i.e., from level $i-1, i-2, \ldots$) and to the right of u (see Figure 6.13(c)). Suppose b is the rightmost base station with the property of serving a user and being to the right and above of her. Consider the smallest subtree T_k containing both u and b. The children subtrees of T_k are of the form $T_1, T_2, \ldots, T_{k-1}$. Let T_{k-i} be the child subtree of T_k containing u and let T_{k-j} be the child subtree of T_k containing b. Note that $i > j$, therefore the horizontal distance from b to u is at least W_{k-j}. Thus, b also blocks all users on level i in the tree T_{k-j}, in particular, user u', which is the user in T_{k-j} whose position in T_{k-j} is identical to the position of u in T_{k-i} (note that, because the root of T_{k-j} has more children than that of T_{k-i}, there must be such a user u', and u' hasn't been served in previous rounds). We will show that u' is only in the interference disk $d := d(b, u)$, and therefore we can use b to serve u' instead of u (the interference disk gets smaller). Now if b is still to the right of u' we are left with a smaller tree T_{k-j} containing both b and u' and we can apply the same transformation rule 3 again until b is to the left of u', when we can apply the transformation 2.

   It remains to show that u' is only in the interference disk d. Suppose for a contradiction that u' is also in another disk $d'' = d(b'', u'')$. Consider the case where b'' is above u'. If user u'' is above b'' then u'' is a neighbour of b'' (transformation 1 has been applied) and therefore if d'' blocks u' then it blocks also the whole level i, u included. So u'' must be below b''. Then u'' has to be from level i (u'' being on level greater than i blocks the whole level i). If u'' is a child of b'' then also u' is, and therefore u'' is blocked by d. Therefore b'' is not a neighbour of u'' and b'' is to the right of u'' (we have applied transformation 2 already), and therefore d'' does not block any user to the left of u''. Thus u'' is to the left of u' and to the right of u. But then u'' is blocked by the disk d.

   Consider the case where b'' is below u'. If u'' is above b'', we know that u'' is

the upper neighbour of b'' (we have applied transformation 1 already), and therefore u'' is the only user above b'' that is blocked by d'', a contradiction to the assumption that d'' contains u'. If u'' is below b'', for d'' to reach u' the radius has to be at least ε_1 more than the vertical distance from b'' to u'. Therefore, d'' blocks the whole level i, u included.

Now we have a maximum independent set M_i where base stations serve only neighbouring users. If base station b' serves a neighbour below, then clearly b' is active. If b' is passive and serves its upper neighbour u, then consider b'' (cf. Figure 6.13(a)), the upper neighbour of u, which is active. We claim that b'' is idle. Because b' is passive and u is its upper neighbour, u has to be the first unserved user (in left-to-right order) among the children of b''. Therefore, if b'' were to serve any other user, it would block u. Thus, we can use b'' to serve u instead of b'. Finally, we have obtained a maximum independent set where only active base stations serve users.

Theorem 6.16 *There are instances (U, B) of 2D-JBS in general position for which the maximum-independent-set algorithm, the smallest-disk-first greedy algorithm, and the fewest-users-in-disk algorithm have an approximation ratio $\Omega(\log n)$, where $n = |U|$.*

We note that the algorithm maximizing the number of served users in every round achieves approximation ratio $O(\log n)$, as can be shown by applying the standard analysis of the greedy set covering algorithm.

6.5 Summary of Results and Open Problems

We have studied the joint base station scheduling problem—a problem of assigning base stations and time slots to users such that every user is assigned to a base station and in every time slot, there is no interference at the site of users who were to communicate at that time slot. These problems can be split into a selection and a colouring problem. In the one-dimensional case we have shown that the colouring problem leads to the class of arrow graphs, for which we have studied its relationship with other graph classes and algorithms. For the selection problem we have proposes an algorithm based on an LP-relaxation

and rounding. For the two-dimensional case we have studied several greedy algorithms as well as setting with bounded parameters, resulting in an approximation algorithm with its approximation ratio depending on these geometric parameters.

The following natural questions remain open.

- What is the complexity of the one-dimensional case?
- Can we design an approximation algorithm with approximation ratio that does not depend on the ratio $\frac{R_{\max}}{\Delta}$?

Chapter 7

Conclusion

In this thesis we have studied four optimization problems. We tackled these problems within the algorithm-theoretic framework. Thus, we were mainly interested in algorithms that run in polynomial time and that deliver solutions as close to optimum solutions as possible. Indeed, as many variants of the studied problems turned out to be $\mathcal{NP}$-hard, we had to abandon all hopes for optimum algorithms (i.e., polynomial-time algorithms delivering optimum solutions) and we concentrated on the approximation algorithms (or online algorithms) and approximation solutions. For the problem of finding a minimum-weight dominating set in unit disk graphs we have delivered the first constant-approximation algorithm and thus showed that the problem belongs to the class $\mathcal{APX}$(Chapter 3).

$\mathcal{NP}$-completness was not the only reason to study approximation algorithms. The complexity of the one-dimensional version of the JBS problem (Chapter 6) remained unresolved, which labels our 2-approximation algorithm for the problem, vaguely said, as our best polynomial-time effort. The study of this problem led to a definition of an interesting class of graphs—the arrow graphs. For this class, its relationship with various other graph classes has been presented. Furthermore we have studied several special cases of the problem in one dimension. The study of the two-dimensional case has resulted into quite opposite results when compared with the one-dimensional case. The general JBS problem is $\mathcal{NP}$-complete, but the existence of any non-trivial approximation algorithm re-

mains an open problem.

Online algorithms and online computation were in the center of our focus in the study of the OVSF-code assignment problem and of the network discovery and verification problem. For the OVSF-code assignment problem (Chapter 5), we have presented a simple strategy giving a $\Theta(h)$-competitive algorithm (h is the number of assigned codes), whereas we have shown the lower-bound to be 1.5. Also, for code insertions only, we have shown that no code reassignment is necessary. The study of the problem was motivated by a previous work that claimed to solve the problem optimally when only one code insertion is considered. We have shown that their argumentation was erroneous, and presented a $O(h)$-approximation algorithm.

We have also introduced the problem of network discovery and verification (Chapter 4), where, for a given query model, the task is to find the minimum number of queries that discover an unknown network. In the layered-graph query model the problem turned out to be equivalent to the problem of computing the metric dimension of the graph. In the design of online algorithms we have used randomization and developed a $O(\sqrt{n \log n})$-competitive algorithm for both query models. We have also studied the problem in its offline variant, and delivered several lower bounds on the optimum number of queries, and analyzed several graph classes.

7.1 Future Work

There are several possible future directions and open problems. For the minimum-weight dominating set the natural question is what is the best approximation ratio for the problem. Also, we may ask for computing the minimum dominating set in general disk graphs or general rectangle graphs.

In the study of the network discovery we are interested in deterministic online algorithms. We may also ask for different query models, or for different computational goals.

For the OVSF-code assignment an open question remains—is there a better approximation algorithm (than our $O(\log n)$-approximation algorithm) for the problem of assigning one code into a code tree?

The study of joint base station scheduling problem leaves more unanswered elementary questions. In the one-dimensional case the main unanswered question is the complexity of the problem. In the two-dimensional case, we do not know any approximation algorithm with non-trivial approximation ratio. The design of such an algorithm is an interesting open problem.

Bibliography

[1] K. I. Aardal, S. P. van Hoesel, A. M. Koster, C. Mannino, and A. Sassano. Models and solutions techniques for frequency assignment problems. Technical Report 01-40, Konrad-Zuse-Zentrum für Informationstechnik Berlin, December 2001.

[2] D. Achlioptas, A. Clauset, D. Kempe, and C. Moore. On the bias of traceroute sampling; or, power-law degree distributions in regular graphs. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing (STOC)*, pages 694–703, 2005.

[3] F. Adachi, M. Sawahashi, and K. Okawa. Tree-structured generation of orthogonal spreading codes with different lengths for forward link of DS-CDMA mobile radio. *Electronics Letters*, 33(1):27–28, January 1997.

[4] J. Adamek. *Foundation of Coding.* John Wiley, Chichester, 1991.

[5] V. Aggarwal, S. Bender, A. Feldmann, and A. Wichmann. Methodology for estimating network distances of Gnutella neighbors. In *Proceedings of the Workshop on Algorithms and Protocols for Efficient Peer-to-Peer Applications*, INFORMATIK 2004, 2004.

[6] K. M. Alzoubi, P.-J. Wan, and O. Frieder. Message-optimal connected dominating sets in mobile ad hoc networks. In *Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing (MobiHoc 2002)*, pages 157–164, 2002.

[7] C. Ambühl, T. Erlebach, M. Mihaľák, and M. Nunkesser. Constant-factor approximation for minimum-weight (connected) dominating sets

in unit disk graphs. Technical report, Department of Computer Science, University of Leicester, June 2006. Available electronically at http://www.cs.le.ac.uk/publications/techreports/2006/CS-06-008.pdf.

[8] C. Ambühl, T. Erlebach, M. Mihaľák, and M. Nunkesser. Constant-factor approximation for minimum-weight (connected) dominating sets in unit disk graphs. In *Proceedings of the 9th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, to appear.

[9] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, May 1998.

[10] R. Assarut, M. G. Husada, U. Yamamoto, and Y. Onozato. Data rate improvement with dynamic reassignment of spreading codes for DS-CDMA. *Computer Communications*, 25(17):1575–1583, 2002.

[11] R. Assarut, K. Kawanishi, R. Deshpande, U. Yamamoto, and Y. Onozato. Performance evaluation of orthogonal variable-spreading-factor code assignment schemes in W-CDMA. In *IEEE International Conference on Communications (ICC)*, pages 3050–3054, 2002.

[12] R. Assarut, K. Kawanishi, U. Yamamoto, Y. Onozato, and M. Matsushita. Region division assignment of orthogonal variable-spreading-factorcodes in W-CDMA. In *Proceedings of the 54th IEEE Vehicular Technology Conference*, 2001.

[13] ATIS Committee T1A1. ATIS telecom glossary 2000. Available online at http://www.atis.org/tg2k/. Accessed in August 2006.

[14] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties.* Springer, 1999.

[15] B. S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the ACM*, 41(1):153–180, January 1994.

[16] P. Barford, A. Bestavros, J. Byers, and M. Crovella. On the marginal utility of deploying measurement infrastructure. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 5–17, November 2001.

[17] Z. Beerliová. Rekonstruktionen von graphen. Master's thesis, Computer Engineering and Networks Laboratory, ETH Zürich, 2004. In German.

[18] Z. Beerliová, F. Eberhard, T. Erlebach, A. Hall, M. Hoffmann, and M. Mihaľák. Network discovery and verification. *IEEE Journal on Selected Areas in Communications (JSAC)*, to appear.

[19] Z. Beerliová, F. Eberhard, T. Erlebach, A. Hall, M. Hoffmann, M. Mihaľák, and L. S. Ram. Network discovery and verification. In *Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, 2005.

[20] Y. Bejerano and R. Rastogi. Robust monitoring of link delays and faults in IP networks. In *Proceedings of the 22nd Conference on Computer Communications (IEEE INFOCOM)*, 2003.

[21] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[22] Y. Breitbart, F. Dragan, and H. Gobjuka. Effective network monitoring. In *Proceedings of the 13th International Conference of Computer Communictions and Networks (ICCCN)*, pages 394–399, October 2004.

[23] H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite VC-dimension. *Discrete & Computational Geometry*, 14(4):463–479, 1995.

[24] J. Cáceres, C. Hernando, M. Mora, I. M. Pelayo, M. L. Puertas, C. Seara, and D. R. Wood. On the metric dimension of cartesian products of graphs. Manuscript, 2005.

[25] G. Calinescu, I. I. Mandoiu, P.-J. Wan, and A. Z. Zelikovsky. Selecting forwarding neighbors in wireless ad hoc networks. *Mobile Networks & Applications*, 9(2):101–111, 2004.

[26] H. Çam. Nonblocking OVSF codes and enhancing network capacity for 3G wireless and beyond systems. *Computer Communications*, 26:1907–1917, November 2003.

[27] G. Chartrand and P. Zhang. The theory and applications of resolvability in graphs: A survey. *Congressus Numerantium*, 160:47–68, 2003.

[28] J.-C. Chen and W.-S. E. Chen. Implementation of an efficient channelization code assignment algorithm in 3G WCDMA. In *Proceedings of National Computer Symposium*, pages E237–E244, Taiwan, 2001.

[29] W. T. Chen and S. H. Fang. An efficient channelization code assignment approach for W-CDMA. In *IEEE Conference on Wireless LANs and Home Networks*, 2002.

[30] W. T. Chen, Y. P. Wu, and H. C. Hsiao. A novel code assignment scheme for W-CDMA systems. In *Proceedings of the 54th IEEE Vehicular Technology Society Conference*, volume 2, pages 1182–1186, 2001.

[31] X. Cheng, X. Huang, D. Li, W. Wu, and D.-Z. Du. A polynomial-time approximation scheme for the minimum-connected dominating set in ad hoc wireless networks. *Networks*, 42(4):202–208, 2003.

[32] B. Cheswick. Internet mapping project. Published online. Project of Lumeta Corporation, an off-spin company of Lucent/Bell Labs.

[33] J. Chuzhoy and S. Naor. New hardness results for congestion minimization and machine scheduling. In *Proceedings of the 36th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 28–34, 2004.

[34] M. Cielibak, T. Erlebach, F. Hennecke, B. Weber, and P. Widmayer. Scheduling jobs on a minimum number of machines. In *Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science*, pages 217–230. Kluwer, 2004.

[35] B. N. Clark, C. J. Colbourn, and D. S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86:165–177, 1990.

[36] Clip2. The Gnutella protocol specification v0.4, 2001. http://www9.limewire.com/developer/ gnutella_protocol_0.4.pdf.

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition edition, 2001.

[38] E. Dahlman, B. Gudmundson, M. Nilsson, and J. Sköld. UMTS/IMT-2000 based on wideband CDMA. *IEEE Communications Magazine*, 36(9):70–80, September 1998.

[39] L. Dall'Asta, I. Alvarez-Hamelin, A. Barrat, A. Vázquez, and A. Vespignani. Statistical theory of internet exploration. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 71, 2005.

[40] L. Dall'Asta, I. Alvarez-Hamelin, A. Barrat, A. Vázquez, and A. Vespignani. Exploring networks with traceroute-like probes: theory and simulations. *Theoretical Computer Science*, 355, 2006. To appear.

[41] B. Das and V. Bharghavan. Routing in ad-hoc networks using minimum connected dominating sets. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 376–380, 1997.

[42] S. Das, H. Viswanathan, and G. Rittenhouse. Dynamic load balancing through coordinated scheduling in packet data systems. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 1, pages 786–796, 2003.

[43] M. de Berg, O. Schwarzkopf, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 2nd edition, 2000.

[44] M. Dell'Amico, F. Maffioli, and M. L. Merani. A tree partitioning dynamic policy for OVSF codes assignment in wideband CDMA. *IEEE Transactions on Wireless Communications*, 3(4):1013–1017, 2004.

[45] G. Di Battista, T. Erlebach, A. Hall, M. Patrignani, M. Pizzonia, and T. Schank. Computing the types of the relationships between autonomous systems. *IEEE/ACM Transactions on Networking*, 2007. Scheduled to appear in August 2007.

[46] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics.* Springer-Verlag, Heidelberg, 3rd edition, July 2005.

[47] DIMES. Mapping the Internet with the help of a volunteer community. http:// www.netdimes.org/.

[48] R. G. Downey and M. R. Fellows. *Parametrized Complexity.* Springer Verlag, 1999.

[49] F. Eberhard. Grundlagen für das mapping des internet-graphen. Semester Thesis at Computer Engineering and Networks Laboratory (TIK), ETH Zürich, July 2003. In German.

[50] A. Eisenblätter. *Frequency Assignment in GSM Networks: Models, Heuristics and Lower Bounds.* PhD thesis, Institut für Mathematik, Technische Universität Berlin, 2001.

[51] T. Erlebach, A. Hall, M. Hoffmann, and MatúšMihaľák. Network discovery and verification with distance queries. In *Proceedings of the 6th International Conference on Algorithms and Complexity (CIAC)*, 2006.

[52] T. Erlebach, R. Jacob, M. M. M. Nunkesser, G. Szabó, and P. Widmayer. An algorithmic view on OVSF code assignment. TIK-Report 173, Computer Engineering and Networks Laboratory (TIK), ETH Zürich, August 2003. Available electronically at ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report173.pdf.

[53] T. Erlebach, R. Jacob, M. M. M. Nunkesser, G. Szabó, and P. Widmayer. Joint base station scheduling. Technical report, Department of Computer Science, ETH Zürich, 2004.

[54] T. Erlebach, R. Jacob, M. M. M. Nunkesser, G. Szabó, and P. Widmayer. Joint base station scheduling. In *Proceedings of the 2nd International Workshop on Approximation and Online Algorithms (WAOA)*, 2004.

[55] T. Erlebach and F. Spieksma. Interval selection: Applications, algorithms, and lower bounds. *Algorithmica*, 46:27–53, 2001.

[56] R. Fantacci and S. Nannicini. Multiple access protocol for integration of variable bit rate multimedia traffic in UMTS/IMT-2000 based on wideband CDMA. *IEEE Journal on Selected Areas in Communications*, 18(8):1441–1454, August 2000.

[57] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[58] S. Felsner, R. Mueller, and L. Wernisch. Trapezoid graphs and generalizations, geometry and algorithms. *Discrete Applied Mathematics*, 74:13–32, 1997.

[59] M. Forišek, B. Katreniak, J. Katreniaková, R. Královič, R. Královič, V. Koutný, D. Pardubská, T. Plachetka, and B. Rovan. Optimal amortized online algorithm for OVSF code reassignment. Preprint, 2006.

[60] C. E. Fossa Jr and N. J. D. IV. Dynamic code assignment improves channel utilization for bursty traffic in 3G wireless networks. In *IEEE International Conference on Communications (ICC)*, pages 3061–3065, 2002.

[61] L. Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Transactions on Networking*, 9(6):733–745, December 2001.

[62] M. R. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, 1979.

[63] P. Goria, C. Guerrini, and A. Vaillant. Signalling delay of code allocation strategies. In *Proceedings of the 11th IST Mobile & Wireless Telecommunications Summit*, 2002.

[64] R. Govindan and A. Reddy. An analysis of internet inter-domain topology and route stability. In *Proceedings of the 16th Conference on Computer Communications (IEEE INFOCOM)*, pages 850–857, April 1997.

[65] R. Govindan and H. Tangmunarunkit. Heuristics for Internet map discovery. In *Proceedings of the 19th Conference on Computer Communications (IEEE INFOCOM)*, pages 1371–1380, Tel Aviv, Israel, March 2000.

[66] A. Gräf, M. Stumpf, and G. Weißenfels. On coloring unit disk graphs. *Algorithmica*, 20(3):277–293, March 1998.

[67] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[68] S. Guha and S. Khuller. Improved methods for approximating node weighted steiner trees and connected dominating sets. *Information and Computation*, 1:57–74, 1999.

[69] J. Håstad. Some optimal inapproximability results. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 1997.

[70] D. S. Hochbaum and W. Maass. Approximation schemes for covering and packing problems in image processing and vlsi. *Journal of the ACM (JACM)*, 32(1):130–136, January 1985.

[71] H. Holma and A. Toskala. *WCDMA for UMTS*. Wiley, 2001.

[72] H. B. Hunt III, M. V. Marathe, V. Radhakrishnan, S. S. Ravi, D. J. Rosenkrantz, and R. E. Stearns. Nc-approximation schemes for np- and pspace-hard problems for geometric graphs. *Journal of Algorithms*, 26(2):238–274, 1998.

[73] A. C. Kam, T. Minn, and K.-Y. Siu. Supporting rate guarantee and fair access for bursty data traffic in W-CDMA. *IEEE Journal on Selected Areas in Communications*, 19(11):2121–2130, November 2001.

[74] I. Katzela and M. Naghshineh. Channel assignment schemes for cellular mobile telecommunication systems: A comprehensive survey. *IEEE Personal Communications*, 3(3):10–31, June 1996.

[75] L. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979.

[76] S. Khuller, B. Raghavachari, and A. Rosenfeld. Landmarks in graphs. *Discrete Applied Mathematics*, 70:217–229, 1996.

[77] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[78] B. Lindström. On a combinatory detection problem I. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 9:195–207, 1964.

[79] M. V. Marathe, H. Breu, H. B. Hunt III, S. S. Ravi, and D. J. Rosenkrantz. Simple heuristics for unit disk graphs. *Networks*, 25:59–68, 1995.

[80] T. Minn and K.-Y. Siu. Dynamic assignment of orthogonal variable-spreading-factor codes in W-CDMA. *IEEE Journal on Selected Areas in Communications*, 18(8):1429–1440, 2000.

[81] S. Narayanappa and P. Vojtěchovský. An improved approximation factor for the unit disk covering problem. In *Proceedings of the 18th Canadian Conference on Computational Geometry (CCCG)*, 2006.

[82] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006.

[83] M. Nunkesser. *Algorithms for Manufacturing, Logistics, and Telecommunications: An Algorithmic Jam Session.* PhD thesis, Department of Computer Science, ETH Zürich, 2006.

[84] Oregon RouteViews Project. University of Oregon. http://www.routeviews.org.

[85] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, July 1998.

[86] C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. In *Proceedings of the twentieth annual ACM symposium on Theory of computing (STOC)*, pages 229–234, 1988.

[87] L. S. Ram. Tree-based graph reconstruction. Research Report of the European Graduate Program Combinatorics-Computation-Geometry (CGC), March 2003. Supervised by Thomas Erlebach.

[88] V. Raman, S. Saurabh, and C. R. Subramanian. Faster fixed parameter tractable algorithms for undirected feedback vertex set. In *Proceedings of the 13th International Symposium on Algorithms and Computation (ISAAC)*, volume 2518 of *Lecture Notes in Computer Science*, pages 241–248. Springer, 2002.

[89] A. N. Rouskas and D. N. Skoutas. Ovsf codes assignment and reassignment at the forward link of W-CDMA 3G systems. In *Proceeings of the 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, volume 5, pages 2404–2408, 2002.

[90] V. Saenpholphat and P. Zhang. Conditional resolvability in graphs: A survey. *International Journal of Mathematics and Mathematical Sciences*, 38:1997–2017, 2004.

[91] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley & Sons, April 1998.

[92] A. Sebő and E. Tannier. On metric generators of graphs. *Mathematics of Operations Research*, 29(2):383–393, May 2004.

[93] SETI. Seti@home website, 2005. http://setiathome.ssl.berkeley.edu/.

[94] F. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2:215–227, 1999.

[95] L. Subramanian, S. Agarwal, J. Rexford, and R. Katz. Characterizing the internet hierarchy from multiple vantage points. In *Proceedings of the 21st Conference on Computer Communications (IEEE INFOCOM)*, June 2002.

[96] G. Szabó. *Optimization Problems in Mobile Communication.* PhD thesis, Department of Computer Science, ETH Zürich, 2005.

[97] M. Tomamichel. Algoritmische aspekte von ovsf code assignment mit swcherpunkt auf offline code assignment. Semester thesis, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, December 2004.

[98] Y.-C. Tseng, C.-M. Chao, and S.-L. Wu. Code placement and replacement strategies for wideband CDMA OVSF code tree management. In *Proceedings of the 44th Annual IEEE Global Telecommunications Conference (GLOBECOM)*, volume 1, pages 562–566, 2001.

[99] UMTS World. Available online at http://www.umtsworld.com.

[100] E. J. van Leeuwen. Approximation algorithms for unit disk graphs. In *Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 351–361, 2005.

[101] V. V. Vazirani. *Approximation Algorithms.* Springer, 2001.

[102] Y. Wang, W. Wang, and X.-Y. Li. Distributed low-cost backbone formation for wireless ad hoc networks. In *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 2–13, 2005.

[103] Y. Yang and T.-S. Yum. Nonrearrangeable compact assignment of orthogonal variable spreading factor codes for multi-rate traffic. In *Proceedings of the 54th IEEE Vehicular Technology Conference*, 2001.

[104] A. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 222–227, 1977.