Diss. ETH No. 18599

Algorithmic Solutions for Transient Faults in Communication Networks

On Swap Edges and Local Algorithms

A dissertation submitted to

ETH Zurich
Swiss Federal Institute of Technology

for the degree of Doctor of Sciences

presented by

Beat Gfeller
Master of Science ETH
born September 15, 1980,
citizen of Röthenbach im Emmenthal, Switzerland

accepted on the recommendation of

Prof. Dr. Peter Widmayer, ETH Zurich, Switzerland
examiner

Prof. Dr. Guido Proietti, University of L'Aquila, Italy
co-examiner

2009

Abstract

Communication networks such as the Internet are ubiquitous today, and play a key role in modern society. Given their widespread use, measures must be taken to ensure the continued operation of these networks even in the presence of failures of individual parts. It is particularly important to efficiently deal with *transient faults*, since these are predominant in many network environments. This tenet is the main motivation for this dissertation, which provides algorithmic solutions for transient faults in communication networks. In this context, we consider network environments with varying characteristics, from rather static to highly dynamic networks.

For handling transient faults in static networks, we use the *swap* approach, which has been designed specifically for this environment: In the absence of any faults, a spanning tree of the underlying network is used for communication. When a link fails, the tree is split into two components, which are then reconnected in a best possible way by adding a single backup link, called a *best swap*. This small and local modification is well-suited for handling transient faults, because it quickly re-establishes communication without a large restructuring effort. Moreover, since the network topology is static, a best swap can be precomputed for every link in the spanning tree before using the network. We present efficient algorithms for precomputing best swaps, for a variety of spanning tree types. In addition, we show how to route messages efficiently through a network using the swap approach.

If too many faults occur in a network, the services it provides will break down. We therefore study the threshold of tolerable faults in networks where transient failures are very frequent. To that end, we consider one of the most fundamental problems in distributed computing, the *consensus* problem, in synchronous networks. In this problem, a number of agents, who can only communicate by exchanging messages through a network, must negotiate a globally unanimous decision. We provide a precise characterization of the number of tolerable failures for reaching such a consensus using a given network.

Finally, we investigate highly dynamic networks, such as wireless ad hoc networks, where in addition to frequent transient faults, the network topology keeps constantly changing as well. In such a

setting, it may be impossible to gather information about the entire network topology. Therefore, a completely different approach is necessary: The task of computing a global solution is distributed among the nodes of the network, and each node only communicates with nearby nodes. Surprisingly, it is possible to find such so-called *local algorithms*, for some fundamental problems in wireless networks. We give efficient local distributed algorithms for two such problems: Computing a maximal independent set, which is a basic building block for distributed coordination, and computing a small connected dominating set, which is useful for energy-efficient message broadcast.

Zusammenfassung

Kommunikationsnetze wie das Internet sind heutzutage allgegenwärtig, und spielen eine wichtige Rolle in der modernen Gesellschaft. In Anbetracht ihrer verbreiteten Nutzung ist es wichtig, Massnahmen zu treffen, um das Funktionieren dieser Netze zu gewährleisten, selbst wenn einzelne Komponenten davon ausfallen. Es ist besonders wichtig, effiziente Mittel gegen *kurzzeitige Fehler* zu finden, da solche Fehler in vielen Netzwerkumgebungen die Mehrheit ausmachen. Dieser Grundgedanke ist die Motivation für diese Dissertation, welche algorithmische Lösungen für kurzzeitige Fehler in Kommunikationsnetzen anbietet. In diesem Zusammenhang werden Netzwerkumgebungen mit verschiedenen Charakteristika betrachtet, von eher statischen bis zu hochgradig dynamischen Netzen.

Um kurzzeitige Fehler in statischen Netzen zu handhaben, benutzen wir das Konzept von *Austauschkanten*, welches für genau diesen Zweck ausgerichtet ist: Solange kein Fehler auftritt, wird zur Kommunikation ein Spannbaum des zugrundeliegenden Netzes benutzt. Wenn eine Verbindung ausfällt, zerfällt das Netz in zwei Komponenten, welche dann auf bestmögliche Weise wieder verbunden werden, indem eine Ersatzkante, die sogenannte *beste Austauschkante*, zum Netz hinzugefügt wird. Diese kleine und lokale Modifikation eignet sich für das Überbrücken von kurzzeitigen Fehlern, weil sie den Nachrichtenaustausch schnell wieder herstellt, ohne eine aufwändige Umstrukturierung des Netzes zu erfordern. Überdies kann, da das Netz statisch ist, für jede Netzkante eine beste Austauschkante vor dem Einsatz des Netzes vorberechnet werden. Wir präsentieren effiziente Algorithmen für die Vorberechnung von besten Austauschkanten, für eine Auswahl von Spannbaum-Typen. Zudem zeigen wir, wie Nachrichten auch während des Einsatzes von Austauschkanten effizient durch das Netz gelenkt werden können.

Wenn in einem Netz zu viele Fehler auftreten, kann es seine Dienste nicht mehr verrichten. Daher interessieren wir uns für das maximal erträgliche Mass an kurzzeitigen Fehlern, das ein gegebenes Netz tolerieren kann. Zu diesem Zweck betrachten wir eines der grundlegendsten Probleme im Gebiet des verteilten Rechnens, das *Konsens*-Problem, in taktgesteuerten Netzen. In diesem Problem muss eine Anzahl von Akteuren, welche nur durch den Austausch von Nachrichten über ein Netz kommunizieren können, eine global einstimmige Entscheidung aushandeln. In diesem Teil der Arbeit geben wir

eine exakte Charakterisierung für die maximale Anzahl von Fehlern in einem gegebenen Netz, welche für das Konsens-Problem tolerierbar sind.

Schliesslich untersuchen wir hochgradig dynamische Netze, wie zum Beispiel kabellose Ad Hoc Netze, in denen nicht nur viele kurzzeitige Fehler auftreten, sondern sich auch die Topologie ständig ändert. In solchen Netzen kann es unmöglich sein, eine Gesamtübersicht über die Netzwerktopologie zu gewinnen. Daher ist ein völlig anderer Ansatz notwendig: Die Aufgabe, eine globale Lösung zu berechnen, wird unter den Knoten im Netz aufgeteilt, und jeder Knoten kommuniziert nur mit Knoten in seiner Nähe. Erstaunlicherweise ist es möglich, solche sogenannten *Lokalen Algorithmen* für verschiedene grundlegende Probleme in kabellosen Netzen zu finden. Wir entwerfen effiziente lokale verteilte Algorithmen für zwei solche Probleme: Die Berechnung einer maximalen unabhängigen Menge, welche ein grundlegendes Hilfsmittel für verteilte Koordination ist, und die Berechnung einer kleinen zusammenhängenden Menge, welche nützlich für energie-effizienten Nachrichtenversand ist.

Acknowledgments

During my PhD studies at ETH, I had the pleasure of working with many interesting individuals. I am grateful to all of them.

First of all, I would like to thank my supervisor, Peter Widmayer, for his guidance and support throughout the years. His quick understanding and drive made very pleasant discussions, and I walked away from each of our meetings with fresh energy. He also introduced me to the customs of academia, and gladly gave advice on various subjects. On top of that, I admire his benevolent attitude towards people in general.

I am also grateful to Guido Proietti for agreeing to be my coexaminer. It was an honour to have one of the pioneers in swapping algorithms in my committee, and I enjoyed the new inspirations that each of his visits brought.

It was a pleasure to work with Subhash Suri during his Sabbatical visit at ETH, whose infectious enthusiasm convinced me (and others) to think about target tracking and mobile robots. I also enjoyed working on range medians with Peter Sanders during his short but productive visit. The both friendly and competitive relationship with Roger Wattenhofer's group was another source of ideas and motivation.

Furthermore, I thank Leon Peeters and Birgitta Weber for guiding me through my Master thesis and the start of my PhD, Elias Vicari for our fruitful collaboration on distributed algorithms, Shantanu Das for many good discussions on swap edges, and for proofreading this thesis, Matúš Mihalak for convincing me to bike to work, Luzi Anderegg for telling me to try out surfing, Marianna Berger for bearing my insane accounting delays, Davide Bilò for teaching me about Davenport-Schinzel sequences, Jörg Derungs for his persistence in talking Swiss German, Michele Di Lorenzi for leaving me his large flatscreen, Yann Disser for proofreading Part II of this thesis, Andreas Feldmann and Anna Zych for the great time in Göteborg, Holger Flier for taking care of our webpage, Michel Gatto for explaining me the benefits of doing a Postdoc, Luciano Gualà for his Italianità, Alex Hall for amazing me with his table soccer skills, Barbara Heller for organizing many nice cake rounds, Juraj Hromkovič's group for populating our seminar talks, Tomás Hruz for proficient IT assistance, Riko Jacob for building a physical all-pairs-shortest-paths model, Jens Maue for learning some Swiss German, Marc Nunkesser

for his legendary parties, Aurea Perez, Konrad Pomm, Reto Spöhel and Torsten Mütze for sharing their offices with me, Franz Roos for bringing some biology to our group, Marcel Schöngens for talking to me in his native German dialect, and Rastislav Šrámek for convincing me of VI's supremacy over Emacs.

Zuletzt möchte ich ganz herzlich meiner Familie danken, meinen Eltern Markus und Elisabeth, und meinen Schwestern Janine und Martina. Ihre Liebe und Unterstützung haben mich dahin gebracht, wo ich heute stehe.

Contents

Chapter 1

Transient Faults in Communication Networks

Communication networks are ubiquitous today, and provide key services in the everyday life of many people. The Internet is certainly one of the most important examples, some of whose services have largely replaced their traditional counterparts. Besides, many other technologies such as landline and mobile phones, television broadcast, and cash machines (ATMs), also rely on communication networks.

With the widespread use of communication networks grows our dependence on them. Yet, when a communication network reaches a certain size, it is inevitable that some of its components fail from time to time. For important networks, measures must hence be taken to prevent breakdown of communication in case of failures. It is often acceptable that the quality of service might degrade when a failure occurs, as long as a certain standard remains guaranteed. To guarantee communication even when some links in the network fail, for example, one can install additional backup links which are only used while a failure is present.

The nature of failures of course strongly depends on the particular type of network at hand. Some networks are relatively static, that

is, their topology does not change often, and failures are relatively rare. This description applies for example to the fiber optic networks which form the backbones of today's Internet: In these networks, failures occur roughly once a day, and they are predominantly confined to a single link [42, 56]. Furthermore, these networks are typically sparse, because the high bandwidth of fiber optic links makes it feasible and economical to bundle traffic in few links rather than in many redundant separate links of smaller capacity.

Other networks are much more dynamic, and constantly change their topology. An important class of networks with these characteristics are wireless networks: Transmitting a signal by radio is much more error-prone than using a wire, since there is almost no control over external sources causing interference. Moreover, wireless networks are often used for mobile devices that people carry around with themselves, which leads to even more changes in the network topology. Wireless networks constitute an important part of the Internet, because their use for the last hop to the end device extends Internet access to public places. More recently, so-called *wireless ad hoc networks* have emerged, which employ a more radical approach — these networks operate without any auxiliary infrastructure (such as access points), by relaying messages wirelessly from device to device. This approach is already used in some real applications, for example for environmental monitoring [7], for agricultural management targeted at Indian farmers [72], and even for keeping cows inside a virtual fence [13], and promises to be a useful concept in many additional application areas. Wireless networks have a particular structure, which can be exploited to develop algorithms that run faster than their counterparts for general topologies.

The above examples of communication networks have in common that a typical link fault does not persist for a long time: In wired networks, a link failure is immediately detected and will be repaired within a short period of time; in wireless networks, many failures are caused by random noise, perhaps a truck passing between sender and receiver, or another sender in the vicinity. In this thesis, we therefore consider *transient failures*, which are characterized by the fact that they only occur for a short period of time. Since transient failures occur in various types of networks, it is important to study solutions for dealing with them.

Outline and Overview of Results

The goal of this thesis is to extend the knowledge about how to deal with transient failures in communication networks. We focus on the theoretical foundations of these solutions, and consider extreme points of the spectrum: In Part I, we consider almost completely static networks, where at any given time at most one failure is present. In Part II we look at an intermediate case, where there can be more than one concurrent failure, and determine the maximum number of faults that any computation which requires global coordination can tolerate. Finally, in Part III, we consider highly dynamic networks. More specifically, the contents of this thesis are the following:

Part I

In the first part of this thesis, we consider networks whose topology is relatively static, with transient failures of links occurring only from time to time. A canonical example of such a network is the backbone infrastructure of the Internet, which is mostly based on fiber optic cables. In these important networks, faults will usually be repaired relatively quickly. Given this fact, it is more reasonable to remedy a link failure by a small local modification than to globally restructure the network. These networks typically have a tree-like topology, because few cables with large bandwidth suffice to provide the required capacities between all pairs of nodes. In our work, we make two simplifying assumptions: that the network used for communication is indeed a tree, and that at any time, there is at most one faulty link in the network. We study the *swapping* approach, which has been designed for exactly this scenario: In a fault-free period, a spanning tree (which optimizes some objective) is used for communication. When an edge of the tree fails, thus disconnecting the network into two components, a so-called *swap edge* (taken from the network graph) is added to the spanning tree to re-establish connectivity. A *best swap edge* is one for which the resulting *swap tree* optimizes a given objective (which is generally related to the objective for the initial spanning tree). To prepare for the failure of any edge in the initial spanning tree, it is best to precompute, for every edge in the tree, a best swap edge. We obtain efficient algorithms for precomputing best swap edges in minimum diameter spanning trees and tree spanners, and describe a simple and efficient routing scheme on trees which is suitable for the swapping

approach.

Specifically, for a graph with n nodes and m edges, we obtain the following results:

- All best swaps of a minimum diameter spanning tree can be computed in $O(m \log n)$ time and $O(m)$ space. Prior to our work, the fastest known algorithm required $O(n\sqrt{m})$ time and $O(m)$ space, which is significantly slower for sparse graphs [65].
- In a distributed setting, all best swaps of a minimum diameter spanning tree (MDST) can be computed in time linear in the hop-diameter of the network, and using $O(n^* + m)$ messages. Here, n^* denotes the size of the transitive closure of the MDST, if all its edges are directed towards the node which initiates the computation. We also give a matching lower bound for the number of messages. Hence, our algorithm is optimal in terms of messages as well as in terms of runtime. To the best of our knowledge, this is the first non-trivial lower bound on the message complexity of a distributed swap edge computation.
- For tree spanners, which are spanning trees that minimize the maximum stretch in the distance between any pair of nodes (compared to their distance in the original graph), all best swaps can be computed in $O(m^2 \log n)$ time and $O(m)$ space, and in $O(n^3)$ time and $O(n^2)$ space for graphs with unit length edges. Best swap edge computation in tree spanners had not been considered in the literature prior to our work.

We also present a compact routing scheme for trees which efficiently supports the use of precomputed swap edges. To the best of our knowledge, no such routing scheme had been previously described in the literature.

Part II

In the second part, we are interested in an intermediate case, where the network topology is still static, but transient failures are very frequent: What if many links of the network may concurrently fail at any given time? Clearly, there is a threshold for the number of bearable failures, above which any distributed computation becomes impossible. But where does this threshold lie?

To answer this question, we look at one of the most fundamental problems in distributed computing, the *consensus* problem, in synchronous networks. In this problem, a number of processors have to take a global decision, while communication is only possible through sending messages across unreliable links. The consensus problem is fundamental because it is one of the simplest distributed computations one can ask for: If the processors cannot even take a global decision, then any more advanced global computation is bound to fail. In our work, we characterize precisely how many failures (of a given type) per round in a given network topology render consensus impossible.

For a given network, modeled by a connected graph G, we are interested in the number of transmission failures that can be tolerated per round. These transmission faults can affect any link in the network, and the set of links affected by a fault may change in every round. We distinguish between three different types of faults: omissions, which are transmissions where a message is sent but none is received; corruptions, which are transmissions where the message that the receiver gets is different from the message that was sent; and additions, which are transmissions where a processor receives a message although none was sent (this models for example man-in-the-middle attacks). For this setting, we show that consensus is impossible if the number of possible faults reaches the following thresholds, where c denotes the edge-connectivity of G:

- c or more omissions
- c or more additions and corruptions
- $\lceil c/2 \rceil$ or more Byzantine faults (i.e., any combination of the three fault types is allowed).

Prior to our work, the best corresponding bounds were Δ, Δ and $\lceil \Delta/2 \rceil$, where Δ denotes the maximum degree of any node in G. Our new bounds are tight, as shown by previously known algorithms [85].

Part III

Finally, we consider highly dynamic networks, such as wireless communication networks, in which there are not only frequent transient faults, but also frequent changes of the network topology. Wireless

communication links are far less reliable than wired links, and therefore a completely new approach is necessary. Indeed, in the third part, we do not even consider failures explicitly. Instead, we strive for so-called *local* algorithms. Intuitively, local algorithms have the property that the outcome of the computation at a node x only depends on the part of the graph that is "close" to x. Formally, however, local algorithms are typically defined as distributed algorithms whose running time grows only very slowly with the number of nodes (significantly slower than linear). Some authors require that a local algorithm must even have constant running time [62, 57], while others also consider non-constant time algorithms as local [47]. A general consensus is that the running time of a local algorithm can be at most polylogarithmic in the number of nodes.

Local algorithms are well-suited for highly dynamic networks for two reasons: First, as shown in a seminal paper [6], any algorithm for a static synchronous network can be transformed into an algorithm for dynamic asynchronous networks, with the same asymptotic running time. Although the transformation works for any algorithm, the communication overhead is linear in the running time of the static algorithm. For a local algorithm, the overhead of this transformation is hence negligible. This is quite intuitive: If some small part of the network changes, then the outcome of a complete recomputation would only differ in some area close to the changed part. Hence, it suffices to repeat the computation only on the affected parts of the graph. Second, having a short running time is advantageous in dynamic networks because it minimizes the risk that a topological change will happen during some computation.

We model wireless networks by the class of so-called *growth-bounded graphs*, which include for example unit disk graphs. In this setting, we look at two fundamental structures in wireless networks: Maximal independent sets, which are a basic building block for many distributed algorithms, and can be used to coordinate the actions of the involved processors, and minimum connected dominating sets, which can serve as an energy-efficient "virtual backbone" for transmitting messages in a wireless sensor network.

For the problem of computing a maximal independent set in such a graph on n nodes, we give a randomized distributed synchronous algorithm with a running time of $O(\log \log n \log^* n)$, where each message contains $O(\log n)$ bits. Prior to our work, the fastest known al-

gorithm was deterministic and had a running time of $O(\log \Delta \log^* n)$ and message size $O(\log n)$ bits, where Δ denotes the maximum degree of a node in the graph. For the problem of computing a minimum connected dominating set in growth-bounded graphs, which is NP-hard, we give a distributed approximation scheme which completes within $O(T_{\mathsf{MIS}} + 1/\varepsilon^{O(1)} \cdot \log^* n)$ rounds of synchronous computation, where T_{MIS} is the number of rounds needed to compute a maximal independent set. We can prove that the computed connected dominating set has constant stretch, constant degree, and therefore a linear number of edges. Moreover, a recent lower bound implies that any constant approximation in unit disk graphs requires $\Omega(\log^* n)$ time [52], which shows, together with a recent optimal MIS algorithm [89], that the running time of our algorithm is asymptotically optimal.

Chapter 2

Preliminaries

2.1 Some Notions from Graph Theory

Throughout the thesis, we use a few basic graph theory concepts. For the sake of completeness, we define these notions here (for an introduction to graph theory, see e.g. [23]).

A *graph* is a pair $G = (V, E)$ of sets, where E is a subset of the 2-element subsets of V. The elements of V are the *nodes*, and the elements of E the *edges* of the graph G. We use $n = |V|$ to denote the number of nodes, and $m = |E|$ for the number of edges. Each edge $e = (u, v)$ in E is said to *join* u and v. The nodes u and v are the *endpoints* of e. We say that u and v are *adjacent nodes*, or *neighbors*, and that node u and edge e are incident with each other, as are v and e. The number of edges incident to a node u is called the *degree* of u, and denoted by δ_u. The set of neighbors of a node z (excluding z itself) in a graph G is written as $\bar{N}_G(z)$.

If $V' \subseteq V$ and $E' \subseteq E$, then $G' = (V', E')$ is a *subgraph* of $G = (V, E)$, written as $G' \subseteq G$. If $G' \subseteq G$ and G' contains exactly the edges $(u, v) \in E$ with $u, v \in V'$, then G' is an *induced subgraph* of G. We say that V' *induces* G' and write $G' = G[V']$.

A *path* $\mathcal{P}$ is a sequence $\langle p_1, \ldots, p_r \rangle$ of adjacent nodes $p_i \in V$, and it is called a *simple path* if all p_i are distinct. Throughout the thesis, we consider simple paths unless explicitly stated otherwise.

Each edge $e \in E$ has a non-negative rational *length* $l(e)$. The

length $|\mathcal{P}|$ of a path $\mathcal{P} = \langle p_1, \ldots, p_r \rangle$, $p_i \in V$, is the sum of the lengths of its edges, and the *distance* $d_G(x, y)$ between two nodes x, y is the length of a shortest path between x and y in the graph G. Similarly, we define the distance between two sets $A, B \subseteq V$ as the distance of two closest nodes $a \in A$ and $b \in B$, i.e., $d_G(A, B) = \min\{d_G(a, b) | a \in A, b \in B\}$. When the intended graph is clearly defined by the context, we sometimes omit the subscript and write $d(\cdot, \cdot)$ instead of $d_G(\cdot, \cdot)$. If all the edges of a graph have length one, we call it an *unweighted graph*. Note that in such a graph, the distance between two nodes a, b is equal to the number of edges, which is then also called *the number of hops from a to b*. For a graph with non-uniform edge lengths, the number of hops between two nodes is to be understood in the corresponding graph where all edges have length one.

A graph G is called *connected* if there exists a path between any two of its nodes. It is called *k-connected*, for $k \in \mathbb{N}$ and for any $U \subseteq V$ with $|U| \leq k-1$, the graph $G[V \backslash U]$ is connected. It is called *k-edge-connected*, for $k \in \mathbb{N}$, if between any two of its nodes, there exist k edge-disjoint paths. Let U be an inclusion-maximal subset of V for which $G[U]$ is connected (i.e., $G[U \cup \{v\}]$ is disconnected for any $v \in V \backslash U$). Then $G[U]$ is a *connected component* of G.

Given a connected graph $G = (V, E)$, we say that $T = (V, E_T)$, $E_T \subseteq E$ is a *spanning tree* of G if T is connected and $|E_T| = |V| - 1$. In this context, we call the edges in E_T *tree edges* and the edges in $E \backslash E_T$ *non-tree edges*.

A partition of V into two sets $\{V_1, V_2\}$ defines a *cut* $F \subseteq E$, which is the set of edges in E that have one endpoint in V_1 and one endpoint in V_2. We say that an edge $f \in F$ *crosses the cut*. The *size* of a cut F is $f = |F|$. For a node $v \in V$, the set of nodes in the same connected component as v in $G' = (V, E \backslash F)$ is called *v's side of the cut*.

Sometimes we will also consider *directed* graphs, in which the edges are *ordered pairs*: an edge (u, v), where $u, v \in V$, is an edge from u to v. A spanning tree $T = (V, E_T)$ can be directed by choosing a root $r \in V$: Every edge in $(u, v) \in E_T$ is then directed towards the root, as shown in Figure 2.1. We say that v is the *parent* of u, and conversely u is a *child* of v. The set of children of v is denoted by $C(v)$. A node v is an *ancestor* of u if v lies on the path in T from u to the root r. The *nearest common ancestor* of two nodes a and

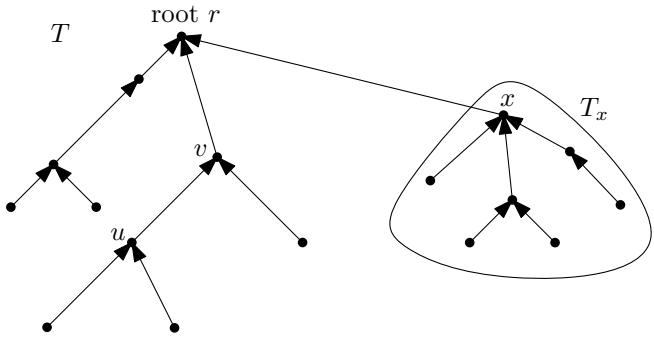

Figure 2.1: *An example of a directed tree T rooted at node r, and a subtree T_x of T rooted at node x.*

b, denoted by $\mathrm{nca}(a, b)$, is the node v which is both an ancestor of a and of b whose distance from r is maximum. Moreover, let T_x be the subtree of T rooted at node x, including x. The *height* of a subtree $T_x = (V_x, E_x)$ is defined as $\max_{v \in V_x}\{d_T(v, x)\}$.

Throughout this thesis, we use $\log n$ to denote the binary logarithm of n, and $\ln n$ for the natural logarithm of n. Furthermore, $\alpha(m, n)$ denotes a functional inverse of the Ackermann function as defined in [91], which is a function that grows very slowly.

2.2 Models of Computation

More than one model of computation is considered in this thesis: some of the results assume that a central authority has complete knowledge of the present network topology, while other results assume that the problem at hand is solved collaboratively by a number of separate entities communicating through messages. Naturally, the objectives in these two settings are quite different. In the following, we specify the models of computation and the corresponding measures used in this thesis.

2.2.1 Centralized

The model we use for our *centralized* algorithms is often referred to as a "pointer machine" in the literature. Although this term is somewhat ambiguous [8], we need not define our model more formally. The crucial point is that our algorithms are "pointer algorithms" in the sense of [8], i.e., the only way we need to access a memory location is by following pointers (we never require indirect addressing). Furthermore, we assume that identifiers of nodes and lengths of edges are atomic, that is, we cannot access their representation in our algorithms. The only operations that can be performed (at unit cost) on edge lengths are additions, subtractions, and binary comparisons. On identifiers, only comparisons can be performed, also at unit cost.

The one exception where we consider another model for a centralized algorithm is Section 3.4, where we improve upon a result which uses the RAM-model of computation. In the latter model, the edge-weights are represented as bit-strings, which can be accessed by the algorithm, and it is possible to specify arbitrary memory locations to be read or written in constant time.

The *time complexity* of a centralized algorithm is the number of unit cost operations that it performs in the worst case (expressed as a function of the input size). The *space complexity* of a centralized algorithm is the maximum amount of memory that it requires at any time during its execution, expressed as a function of the input size. In the centralized model, our two main objectives are to minimize the worst-case time and space complexity of the designed algorithm.

2.2.2 Distributed

Some of our algorithms are *distributed*, meaning that the computation is not performed by a single entity, but by a group of n individual processors which interact by exchanging messages. Typically, each processor can exchange messages only with a subset of the other processors, which are called its *neighbors*. For communication with other processors, messages have to be relayed through intermediate processors. We use a graph to model the network topology, where processors are modeled as nodes, and two nodes are joined by an edge exactly if their corresponding processors are neighbors. In the distributed setting, initially each processor only knows its neighbor-

ing processors (the *connectivity* information), and the lengths of the edges leading to them. Hence, although the complete network topology is specified by the union of the initial knowledge of all processors, no processor has a global view. Each processor has a unique identifier consisting of $O(\log n)$ bits. Every processor executes the same algorithm, whose input consists of the processor's identifier, the processor's problem-specific input, and received messages. The *message complexity* of a distributed algorithm is the total number of messages sent in the worst case during its execution. We need to distinguish between *synchronous* and *asynchronous* network models: In synchronous networks, the computation proceeds in rounds. Each round consists of three phases: In the "compute" phase, a processor performs some local computation. Then, in the "send" phase, it decides on the messages that are to be sent to its neighbors. Finally, in the "receive" phase, every processor may receive a message from each of its neighbors, provided that this neighbor has sent one. All processors start executing synchronously in the first round. The *time complexity* of a synchronous algorithm is the number of rounds required from its start until all processors complete their computation in the worst case (with respect to the network topology and to the assignment of the identifiers to the nodes). In the asynchronous model, processors may start executing their algorithm at different times. Each processor performs the "receive-compute-send" cycle, but its timing is independent of all other processors. A message sent to another processor can have an arbitrary (but finite) delay. Once the message has arrived, the receiving processor will detect the message the next time that it enters the "receive" phase. The *time complexity* of an asynchronous algorithm is the maximum number of "receive-compute-send" cycles that are performed by any processor, in a worst-case instance, assuming that sending a message requires at most one time unit.

There exists a wide variety of models for distributed computation. For our purposes, it suffices to distinguish between two classes of models, namely the $\mathcal{LOCAL}$ model and the $\mathcal{CONGEST}$ model, as described in [76]. In the $\mathcal{CONGEST}$ model, each message that is sent from one processor to another can contain at most $O(\log n)$ bits. It hence focuses on the amount of information that is sent during a computation, and on the effects of congestion on the time complexity and message complexity of an algorithm. On the other hand, the $\mathcal{LOCAL}$ model is intended to focus on the fact that the further two processors are apart, the longer it takes to exchange a message

between them. To stress this so-called "locality" aspect of distributed computation, the $\mathcal{LOCAL}$ model ignores congestion by allowing messages of unlimited size. Furthermore, this model assumes a synchronous network.

In distributed algorithms for the $\mathcal{CONGEST}$ model, which we present in Chapters 5, 7 and 11, our objective is to minimize both the time complexity and the message complexity. In distributed algorithms for the $\mathcal{LOCAL}$ model, as presented in Chapter 12, our objective is solely to minimize the time complexity. Note that in both variants of the distributed model, local computation of a processor is free, i.e., it is not part of our objective function. Yet, in our algorithms the time and space complexity of local computations will always be polynomial in the size of the input.

Part I

Swap Edge Computation

Chapter 3

Introduction

3.1 Motivation

Communication networks are ubiquitous today, and many services of everyday life depend on them. Due to this dependence, it is crucial that networks remain operational even if some of its components fail. The resilience of a network to failures is called its *survivability*, and was studied intensively, see e.g. the overview [39]. When, in the 80s, the core networks of the Internet were gradually transformed from the traditional copper wires to fiber optic cables, network survivability became even more crucial: Given the large bandwidth of fiber optic links, it is economically beneficial to use as few links as possible, i.e., making the network as sparse as possible. If all nodes are connected using the smallest number of links, the subset forms a spanning tree of the network. In addition to the economical benefits, since only one path exists between any communication pair, a spanning tree simplifies routing and allows small routing tables [31]. However, having fewer links also means that the impact of individual link faults increases — in a tree network, even a single link fault will lead to two disconnected parts.

Depending on the purpose of the network, there is a variety of desirable properties of a spanning tree. In the following, we consider the *minimum diameter spanning tree* (MDST), i.e., a tree that minimizes the largest distance between any pair of nodes, as a canonical example. The importance of MDSTs is widely recognized [12], and

it is also the subject of Chapters 4 and 5. In Chapter 6, we consider tree spanners, another important spanning tree type.

One downside of using a spanning tree is that the failure of a single link, say $e \in E_T$, disconnects the network. One extreme way to cope with such a link failure is to *reoptimize*, that is, to use the optimal spanning tree of the remaining network $G-e$ (in our example, the spanning tree of $G - e$ with minimum diameter). This solution

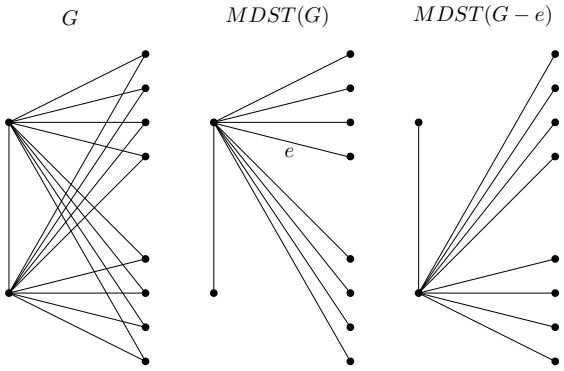

Figure 3.1: *An example of a graph G and a MDST of it, where the new MDST after the failure of an edge has only one edge in common with the old MDST.*

by definition yields the best possible spanning tree for the remaining network, but at a high cost, since the new spanning tree may be very different from the one used before the failure (see Figure 3.1 for an example). Provided that the failure of an edge is a permanent one, these high costs may be justifiable. However, in real networks, a failing link will normally be repaired, and even relatively quickly: For example, a recent study [56] shows that in IP backbone networks, about 60% of all single link faults are repaired in less than 2 minutes, and 90% of all single link faults are repaired in less than 20 minutes. The same study also shows that about 70% of the failures occurring in such a network are confined to a single link.

Therefore, in this thesis we assume that link failures are *transient*, i.e., a failed link soon becomes operational again. We further assume that at any given time, there is at most one link failing. Although this assumption is somewhat extreme, we believe that it is reasonable, for two reasons: First, if even one link fault is relatively rare, then two

or more concurrent faults are even less likely, and so preparing for more than one fault may simply not be worth the necessary overhead. Second, the approach that we will use can still be applied if there are several faults present, as long as the faults are distant enough in some sense.

When link faults are transient, the best possible way of momentarily reconnecting the network is to replace the failed link by a single other link, called a *swap* link, and leaving all other links in the networks unchanged. Among all possible swap links, one should choose a *best swap* w.r.t. the original objective[1], that is in our example, a swap that minimizes the diameter of the resulting *swap tree*. Note that the swap tree is different from a minimum diameter spanning tree of the underlying graph without the failed link. The reason for preferring the swap tree to the latter lies in the effort that a change of the current communication tree requires: If we were to replace the original MDST by a tree whose edge set can be very different, we would need to put many edges out of service, many new edges into service, and adjust many routing tables substantially — and all of this for a transient situation. For a swap tree, instead, only one new edge goes into service, routing can be adjusted with little effort, and messages can even be rerouted "on the fly" (as we will show in Chapter 7). An additional advantage of this approach is that once the failing link has been repaired, we can quickly revert back to using the original spanning tree. Interestingly, this choice of swapping against using a completely reoptimized tree often even comes at a moderate loss in the objective value: Taking the MDST as an example, the swap tree diameter is at most a factor of 2.5 larger than the diameter of an entirely reoptimized tree [65].

In order to keep the required time for swapping (and for the associated adjustment of routing tables) small, we advocate to *precompute* a best swap edge for each edge of the tree. We show in the following chapters that the computation of *all best swaps* typically has the further advantage of gaining efficiency (against computing swap edges individually), because dependencies between the computations for different failing edges can be exploited. Furthermore, precomputing all best swaps is advantageous from a management point of view,

[1] Sometimes, there is more than one natural definition for the best swap: e.g. in a shortest paths tree, minimizing the average distance of all nodes from the root is just as natural as minimizing the maximum of these distances.

because it shows in advance how much the quality of the network topology depends on its individual links (see also Section 3.3.5).

3.2 Problem Statement and Terminology

In this section, we formally define the *all best swaps* problem. We use the basic graph terminology that was introduced in Section 2.1.

We model a communication network as a 2-edge-connected, undirected graph $G = (V, E)$, with $n = |V|$ nodes and $m = |E|$ edges, where each edge $e \in E$ has a length $l(e) \in \mathbb{Q}^+$. Consider a spanning tree $T = (V, E_T)$ of G, which minimizes some objective function $\widehat{obj} : \mathcal{T} \to \mathbb{Q}^+$, where $\mathcal{T}$ denotes the set of spanning trees of G. For the following terminology on swap edges, see Figure 3.2.

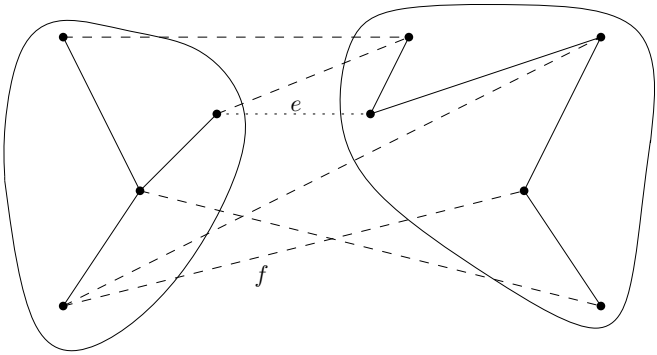

Figure 3.2: *The graph $T - e$ has two connected components. The dashed lines denote swap edges, which are the edges crossing the cut induced by these two components.*

The removal of any edge $e \in E_T$ partitions the spanning tree T into two disjoint trees. A *swap edge* f for e is any edge in $E \backslash E_T$ that (re-)connects the two trees, i.e., for which the graph $T_{e/f} := (V, (E_T \backslash \{e\}) \cup \{f\})$ is a spanning tree of G. In other words, a swap edge for e is any edge in $E \backslash E_T$ which crosses the cut induced by the two connected components of $T - e$. Let $S(e)$ be the set of swap edges for e. We define the set of *swap trees* for T as $\mathcal{T} := \{T_{e/f} | e \in E_T, f \in S(e)\}$. To formalize our objective, we define the *objective function* $obj : \mathcal{T} \to \mathbb{Q}^+$ (in the example of the

MDST, $obj(\Upsilon)$ would yield the diameter of Υ). Note that obj may depend on the input graph G (not only on the swap tree itself), but for simplicity we neglect this dependence in our notation. A *best swap edge* for e is any edge $f \in S(e)$ for which $obj(T_{e/f})$ is minimum. Throughout the thesis, we use e to denote a failing edge and f to denote a swap edge. We can now formally define the *All-Best-Swaps* problem:

Definition 3.1 (*All-Best-Swaps* (*ABS*) problem)**.** *The input to the* All-Best-Swaps *problem is a 2-edge-connected graph G, a spanning tree T of G for which $\widehat{obj}(T)$ is minimum, and an objective function obj. The output must specify, for each edge $e \in E_T$, a best swap edge f with respect to the given objective function obj.*

It is often natural to define $obj := \widehat{obj}$, that is, a best swap edge should optimize the same objective function which the given spanning tree T optimizes. This is the case for example in minimum diameter spanning trees, in minimum spanning trees, and other variants. There are exceptions, however: For example in shortest paths trees, although a characterizing objective $\widehat{obj}$ exists, the choice $obj := \widehat{obj}$ is only one of the natural options (see Section 3.3.2). Another exception arises when the computed swaps should minimize the routing adaptation time, which is completely independent of the objective that the spanning tree minimizes (see Sections 7.3 and 7.4).

In this part of the thesis, we study the *All-Best-Swaps* problem in the centralized as well as in the distributed setting. Our results for the centralized model are relatively insensitive to the precise input representation. We can assume for example that the input consists of two lists, one containing the nodes of the input graph, and one containing all edges of the input graph, where each edge which also belongs to the given spanning tree is marked. The output is a list of $n - 1$ pairs of edges, where the first edge in each pair is an edge of the spanning tree, and the second edge is a best swap edge for it.

In the distributed setting, specifying the problem is slightly more subtle: It must be specified which nodes know which part of the input at the outset of the computation, and also which nodes need to know a best swap to which edges at the end of the computation. Regarding the input specification, a customary assumption in distributed computing is that each node has a list of all its neighboring nodes in the input graph, as well as the list of all edges incident to it, among

which those edges which are also part of the given spanning tree are marked. The output specification is that at the end of the computation, each node knows a best swap for all its incident edges. This latter requirement has no customary counterpart, but is motivated by our proposed routing protocol for swapping (see Section 7).

Remark: The assumption that the underlying network graph is 2-edge-connected is made for ease of exposition. All our algorithms can be easily modified to detect the situation where there is no swap edge for some tree edges, without affecting their efficiency. After this modification, they can be applied to arbitrary connected graphs.

3.3 Related Work

In this section, we review the growing body of literature on the all best swaps problem and some related areas. We state the most relevant results, and discuss some of the techniques used.

3.3.1 Minimum Spanning Trees

The minimum spanning tree (MST), which is a spanning tree minimizing the total length of its edges, is probably the most well-known spanning tree type. For a MST, using swap edges is particularly attractive: it is easy to see that when using a best swap edge to replace a failing edge, the swap tree is again a MST of the graph deprived of the failing edge. Furthermore, the best swap for a MST edge is simply the shortest edge across the cut induced by the failing edge.

Due to these peculiar properties, computing all best swaps of a MST is closely related to the *sensitivity analysis problem*: Given a graph G and a MST T of it, compute for each edge e of T, by how much $l(e)$ can change without affecting the minimality of T. For the tree edges (i.e., edges in T), this is essentially equivalent to computing, for each edge e of T, the length of a shortest edge across the cut induced by e: if the length of e is increased above this threshold, then T is no longer a MST. Sensitivity analysis for MSTs can be performed in $O(m\alpha(m,n))$ time and $O(m)$ space using a so-called *transmuter* data structure [93]. The same algorithm also yields all best swap edges of a MST. Since transmuters are a key data structure

in various best swap computations, we describe them separately at the end of this section.

In subsequent work, a randomized algorithm for sensitivity analysis of MSTs with $O(m)$ expected running time was discovered [24], which can also be used for computing all best swaps of a MST in linear expected time. However, this algorithm requires a somewhat stronger model than we usually consider in this thesis: Although edge costs can only be compared, added, or subtracted at unit cost, "side computations" for look-up tables are performed on a random-access machine (RAM) with word size $\Omega(\log n)$ bits. Remarkably, in the same paper a deterministic algorithm is given which is proven asymptotically optimal, but the precise bound remains unknown. Recently, the $O(m\alpha(m,n))$ time bound of Tarjan [93] was reduced down to $O(m \log \alpha(m,n))$ by Pettie [80], which however also requires side computations on a RAM.

In the context of MSTs, the related problem of computing all best sets of swap edges for *node failures* has also been studied: When a node fails, the spanning tree may be split into more than two connected components, and hence adding one swap edge is not always sufficient. Yet, the best swap edges for node failures in a MST can be computed in $O(m\alpha(m,n))$ time and $O(m)$ space [68]. For the special case of planar graphs, the time bound can even be improved to $O(m)$, which is asymptotically optimal [33].

In the distributed setting, the same problem can be solved using $O(n_r^*)$ messages, where n_r^* is the size of the transitive closure of $T_r \backslash \{r\}$, where all edges are directed towards the root r from which the computation starts [29].

Using a Transmuter for Computing Best Swap Edges

A *transmuter* is an auxiliary graph, which can be used e.g. for sensitivity analysis of MSTs [93]. Given a graph G and a spanning tree T of it, let $T(f)$ denote the path in T connecting the two endpoints of f. A transmuter is a directed acyclic graph D which represents the set of fundamental cycles of G with respect to T. More precisely, D contains one source node $s(e)$ for each edge e in E_T, one sink node $t(f)$ for each edge f in $E \backslash E_T$, possibly some internal nodes, and has the property that there exists a path from a given source $s(e)$ to a given sink $t(f)$ if and only if edge e lies on the path $T(f)$ (in other words,

if and only if f is a swap edge for e). Using a transmuter D, for each edge e of T, the length of a shortest edge across the cut induced by e can be computed as follows:

Algorithm 3.2 (see also [93])**.** *Given a graph G and a spanning tree T of G, compute its corresponding transmuter D. Process the nodes of D in reverse topological order. To process a sink $t(f)$, label it with the length $l(f)$ of the edge it represents. To process a node which is not a sink, label it with the minimum of the labels of its (immediate) successors.*

Note that when all nodes have been processed, each source node $s(e)$ is labeled with the length of the shortest edge among its swap edges. It is trivial to maintain the actual edge with shortest length along with this process. Hence, computing all best swaps of a MST takes only linear time in the size of the transmuter.

For any graph G with m edges and n nodes, a transmuter can be constructed using $O(m\alpha(m,n))$ time and space [92], and so all best swaps of a MST can be computed in $O(m\alpha(m,n))$ time and space. By processing the non-tree edges in groups, the space bound can even be decreased to $O(m)$ [93]. In conclusion, all best swaps of a MST can be computed in $O(m\alpha(m,n))$ time and $O(m)$ space using a transmuter. Moreover, note that Algorithm 3.2 is not restricted to MSTs; in fact, it is applicable whenever two competing swap edges can be compared independently of the tree edge they are to replace:

Observation 3.3. *Consider any swapping problem where one can assign constants to all non-tree edges, such that whenever two swap edges f and g are both swaps for a failing edge e, the better one has a lower constant. Then, Algorithm 3.2 can be applied to compute all best swaps in $O(m\alpha(m,n))$ time and $O(m)$ space.*

This observation is used in various swapping papers discussed in the following sections. Two concrete examples of how to use the transmuter for swap edge computation are given in Section 7.4.

3.3.2 Shortest Paths Trees

The *All-Best-Swaps* problem has been thoroughly studied in single-source shortest paths trees (SPTs). A *single-source shortest paths tree* T of a given graph G with a designated source node r is a spanning

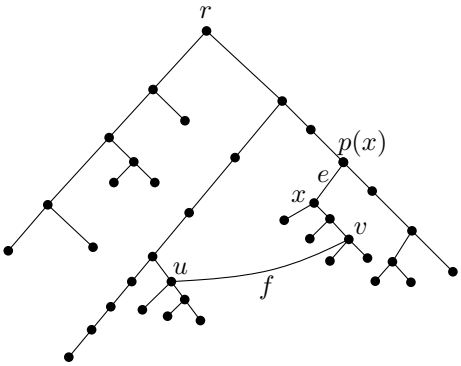

Figure 3.3: *Illustration for the different objective definitions.*

tree which contains, for each node $t \in V$, a shortest path (in G) from r to t. Consequently, it holds: $\forall t \in V : d_T(r,t) = d_G(r,t)$.

Shortest paths trees can be characterized with our formalism as minimizing the objective $\widehat{obj}(\Upsilon) := \max_{t \in V}(d_\Upsilon(r,t) - d_G(r,t))$: For any SPT T, it holds $\widehat{obj}(T) = 0$, and any spanning tree T' with $\widehat{obj}(T') = 0$ is a SPT.

Setting $obj := \widehat{obj}$ then means that the best swap should minimize the maximum additive increase of any node from the root when going from G to the swap tree. Indeed, this definition has been studied and was named "the $\{r, \Delta\}$-problem" [67]. Apart from this variant, the cited paper considers several natural alternatives. To define their objective functions, let $e = (x, p(x))$ denote the failing edge, and let x be the root of the subtree that is disconnected from the source node r (see also Figure 3.3). Furthermore, let $f = (u, v)$ be the swap edge, where v lies within the subtree that is disconnected from the source. The considered variants are:

(i) the $\{r, x\}$-problem: $obj(\Upsilon) = d_\Upsilon(r, x)$

(ii) the $\{p(x), x\}$-problem: $obj(\Upsilon) = d_\Upsilon(p(x), x)$

(iii) the $\{r, \min\}$-problem: $obj(\Upsilon) = d_\Upsilon(r, v)$

(iv) the $\{r, \max\}$-problem: $obj(\Upsilon) = \max_{t \in T_x}\{d_\Upsilon(r,t)\}$

(v) the $\{r, \Sigma\}$-problem: $obj(\Upsilon) = \sum_{t \in T_x}\{d_\Upsilon(r,t)\}$

Variants (i)-(iii), as well as the $\{r, \Delta\}$-problem, can be solved in $O(m\alpha(m, n))$ time and $O(m)$ space, using the transmuter approach as described in Section 3.3.1. Depending on the variant, the constant assigned to each non-tree edge must be chosen differently. Interestingly, in another independent work, an alternative algorithm with running time $O(m + n \log n)$ is given in [9], which works for variants (i)-(iii) as well as the $\{r, \Delta\}$-problem. This approach does not use a transmuter, but first computes a set of $O(n)$ swap candidates which contains a best swap for each edge, by computing the minimum weight spanning forest of an auxiliary graph. Note, however, that the running time of this approach is never better than using a transmuter, since $m\alpha(m, n)$ is in $O(m + n \log n)$.

Variants (iv) and (v) cannot be solved in the same way. The solutions in [67] require $O(n\sqrt{m})$ time and $O(m)$ space for variant (iv), and $O(n^2)$ time and space for variant (v). Some further variants of (iii)-(v) are studied in [67], where r is replaced by $p(x)$ in the objective function for the swap tree. These "dual" variants can all be solved similarly and with the same asymptotic complexity as their counterparts. Later, an improved solution was obtained for variant (v), with $O(m\alpha(m, m) \log^2 n)$ time and $O(m)$ space, by mapping the problem to a geometric setting [22, 10][2]. We give an alternative algorithm for variant (iv) with time $O(m \log n)$ in Section 4.7.

In addition to these theoretical results, an experimental study on shortest paths trees of random graphs shows that the average quality of a swap tree is much better than the worst-case bounds [81].

For shortest paths trees, there exist distributed algorithms which compute all best swap edges for several objectives [28, 30, 71]. These algorithms all proceed in a similar fashion, using three phases: In a preprocessing phase, some information (for example, the distance of each node to the root) is computed using $O(m)$ messages. In the second phase, for each edge $e = (x, p(x))$ of the spanning tree, a separate computation is started for computing its best swap: Using a broadcast which starts at the node x directly below the edge e, say, some information is sent to all the nodes in the disconnected subtree below e. This information allows each node inside T_x to compute a best swap candidate among all swaps incident to itself. In the third phase, a globally best among all these locally best swap edges is se-

[2]The cited manuscript corrects a detail in the original paper, which slightly increases the stated running time from $O(m \log^2 n)$ to $O(m\alpha(m, m) \log^2 n)$.

lected using a convergecast which starts at the leaves of T_x.

Although the given high-level description applies to all mentioned distributed algorithms for best swap computation, the actual information which is computed and processed in both phases differs, as it must depend on how the quality of a swap edge is defined. For the variant (i) described above, a distributed algorithm for the *All-Best-Swaps* problem uses $O(n_r^*)$ messages, where again n_r^* is the size of the transitive closure of $T_r \backslash \{r\}$, where all edges are directed towards the root [28]. Distributed algorithms with the same message complexity were later also found for the variant (ii) [71], and for the variants (iv), (v) and the $\{r, \Delta\}$-problem [30]. As a technical detail, note that in these solutions it is assumed that only the "lower" endpoint x of the failing edge e must know the best swap at the end of the computation, which is why in the n_r^* measure the root r need not be considered (therefore, e.g. in a star graph where r is the center, no messages need to be exchanged at all). In our definition of the distributed *All-Best-Swaps* problem, also node $p(x)$ needs to know the best swap.

3.3.3 Minimum Routing-Cost Spanning Tree

More recently, the *All-Best-Swaps* problem was generalized from the single-source shortest paths tree to the *minimum routing-cost spanning tree* (MRCST), which minimizes the sum of the distances from a given set of sources to all nodes in the tree. Best swaps are defined using the natural choice $obj := \widehat{obj}$. For the case of two sources, all best swap edges can be computed in $O(m \log n + n^2)$ time, and in $O(mn)$ for cases with more sources [96]. When all nodes of the tree are sources, the *All-Best-Swaps* problem can be solved in $O(n^2 + m\alpha(n, n) \log n)$ time [10]. Note that computing a MRCST is in general NP-hard. Motivated by this fact, it is also shown latter work that if the original spanning tree is an approximate MRCST with certain additional properties (such a MRCST can be constructed in polynomial time), then using a swap tree is still a constant approximation of a new optimal MRCST.

3.3.4 Minimum Diameter Spanning Trees

Computing all best swaps of a MDST was one of the first swap problems that were studied. In [65], a centralized algorithm for this prob-

lem is given which requires $O(n\sqrt{m})$ time and $O(m)$ space. For each of the $n-1$ different tree edges, this algorithm uses somewhat augmented topology trees to select $O(\sqrt{m})$ best swap candidates, then evaluates the quality of each of the $O(\sqrt{m})$ candidate swap edges in $O(1)$ amortized time, and selects the best among them. In order to obtain the $O(1)$ amortized time for computing the diameter of the swap tree associated with a given swap edge, information from a preprocessing phase is used, and combined with an inductive computation that uses path compression.

3.3.5 Most Vital Edges and Nodes

In this section, we discuss related work in an area which is closely related to best swap edge computation, namely the computation of *most vital edges*. Here, again transient failures of edges are considered. The focus is different, however: One wants to find out which edge is most vital to the network, in the sense that if it fails, the degradation of service will be worst. In fact, the concrete approaches typically define a measure for each edge, which reflects the degradation of the network if this edge fails. This knowledge is useful from a management point of view, because it highlights the important edges, so that measures can be taken to protect these particularly well from failures.

The most vital edge (MVE) problem was so far only studied for the shortest path between two nodes r and t in a graph G: There, the degradation caused by the failure of an edge e on the shortest path from r to t is defined as the length of a shortest path from r to t in $G-e$, i.e., $d_{G-e}(r,t)$. This MVE problem can be solved in $O(m+n\log n)$ time and $O(m)$ space [55, 86][3]. If one deviates from the pointer-machine model which is typically used in this area, and uses a RAM-model instead, the same problem can be solved in $O(m\alpha(m,n))$ time [64]. As an aside, we show in Section 3.4 that in the RAM-model, there is actually a linear time solution to this problem.

Depending on the motivating application, the degradation of a shortest path due to an edge failure can also be defined differently: For a failing edge $e=(x,y)$, let x be the endpoint of e which comes first on the shortest path from r to t. It can happen that a message travels from r to t on the shortest path, and notices the failure only

[3]The cited report corrects an error in the analysis of the original paper.

when arriving at x. To bypass the failure, the message then travels along a shortest path in $G - e$ from x to t. Therefore, it makes sense to define the length of the caused detour, i.e., $d_{G-e}(x,t) - d_G(x,t)$, as the degradation measure, which leads to the *detour-critical edge* problem, which can be solved in $O(m + n \log n)$ time [63]. Another variant uses the ratio $d_{G-e}(x,t)/d_G(x,t)$ to define the degradation. A published algorithm for this variant requires $O(mn)$ time [90], but since the algorithm in [63] actually computes $d_{G-e}(x,t)$ for every edge e and every node x, and $d_G(x,t)$ is trivially obtained from the input, it can be solved in $O(m + n \log n)$ time as well.

The detour-critical edge problem has also been studied for shortest paths trees, where it can be solved in $O(m\alpha(m,n))$ time [97]. Note that this solution is faster than the corresponding one for a single shortest path, which may look like a contradiction at first. The explanation is that in [97], a SPT of the original graph is part of the input, which can be used in the computation, whereas in [63] only a shortest path is given.

As in the swap edge problems, one can also consider transient node failures instead of edge failures, and determine the *most vital nodes*. For example, the most vital node of a shortest path can be computed in $O(m + n \log n)$ time and $O(m)$ space [66].

3.4 Computing the Most Vital Edge of a Shortest Path in Linear Time

We revisit the most vital edge problem of a shortest path P between two nodes r and t in a graph G. Using a RAM-model, Nardelli et al. [64] give an algorithm with running time $O(m \cdot \alpha(m,n))$. Their algorithm proceeds as follows: First, a shortest paths tree $S_G(r)$ of G rooted at r is computed, which is possible in $O(m)$ time in their model (they assume that the edge lengths are not atomic, but represented as floating point numbers). Then, a shortest paths tree $S_G(t)$ of G rooted in t is computed. They prove that a shortest path from r to t in $G - e$ can always be obtained by first traveling on $S_G(r)$, then using a single edge $f = (u,v)$ which crosses the cut induced by e, and then continuing from v on $S_G(t)$ to t. Since the length of this path is independent of the failing edge e, the problem becomes identical to a swap edge problem where the cost of a swap is constant. For

each edge which does not lie on P, the correct constant is obtained in constant time using the two computed shortest paths trees. Then, the transmuter approach mentioned in Section 3.2 is applied, which yields the degradation for each failing edge in $O(m\alpha(m,n))$ time.

We use the following observation to improve this last step: Since all considered failing edges lie on a path (as opposed to forming a general tree), every edge f not on P is a swap edge for a contiguous sequence of edges of P. Furthermore, it is easy to determine, for every such edge f, the two nodes a and b on P which delimit this sequence. Since each swap edge can be evaluated by its assigned constant, finding a best swap for each failing edge on P reduces to finding the lower envelope of a collection of $O(m)$ horizontal line segments. This problem can be solved in $O(m)$ time in the RAM-model (see [3], Section 5, the solution to problem P2). We therefore obtain an algorithm for the MVE problem running in $O(m)$ time, which is asymptotically optimal. Note that the same modification also works for the longest-detour problem discussed in [64].

Chapter 4

A Centralized Algorithm for Minimum Diameter Spanning Trees

4.1 Motivation

In this and the next chapter, we consider the swap edge approach using a *minimum diameter spanning tree* (MDST) as the communication tree, i.e., a tree that minimizes the worst-case length of the transmission path between any pair of nodes. The importance of minimizing the diameter of a spanning tree has been widely recognized (see e.g. [12]); essentially, the diameter of a network provides a lower bound (and often even an exact one) on the computation time of distributed algorithms which require global communication. Formally, the objective function here is $\widehat{obj} : \mathcal{T} \to \mathbb{Q}^+$, where

$$\widehat{obj}(T) := \max_{x,y \in V} \{d_T(x, y)\},$$

and we use the natural choice $obj := \widehat{obj}$ for defining best swaps. Consequentially, a best swap edge in our case minimizes the diameter of the resulting swap tree. The MDST minimizes the worst-case length of any transmission path, even if edge lengths are not uniform.

Interestingly, for MDSTs, the swap tree diameter is at most a factor of 5/2 larger than the diameter of an entirely reoptimized tree [65]. Hence, MDSTs are particularly well-suited for the swapping approach.

In this chapter, we consider the *All-Best-Swaps* problem in the centralized setting. The provided algorithm can serve as a planning tool for a network designer: For any given network, all swap edges can be computed efficiently, as well as the resulting diameter of every swap tree. Once a network design has been fixed, the information about the best swaps can be stored at suitable positions in the network, which then allows decentralized handling of edge failures during the actual network operation (see Chapter 7).

In some scenarios, this centralized solution for computing all best swaps is not convenient, for example if swapping is to be introduced into a network already in existence, and nobody has complete information about its entire topology. In this case, it is natural to solve the *All-Best-Swaps* problem in a distributed setting. We give a solution to the *distributed All-Best-Swaps* problem in Chapter 5.

4.2 Summary of Results

Our main result in this chapter is an algorithm for computing all best swap edges for a MDST, which proves the following:

Theorem 4.1. *Given a 2-edge-connected graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, and a minimum diameter spanning tree T of G, all best swap edges of T can be computed in $O(m \log n)$ time and $O(m)$ space.*

For $m = o\left(n^2/\log^2 n\right)$, this improves upon the time complexity of the previously best known solution [65], using $O(n\sqrt{m})$ time and $O(m)$ space, without increasing the space complexity. Our techniques also solve the $\{r, \max\}$-problem of [67], which asks for all best swap edges in a shortest paths tree, in time $O(m \log n)$ instead of $O(n\sqrt{m})$.

This improvement over the previous bounds is based on two key ingredients: First, partitioning the set of tree edges into two particular sets, and computing their best swap edges separately using two different techniques, and second, utilizing an essential observation (Lemma 4.3) to simplify the computation of the diameter in a given

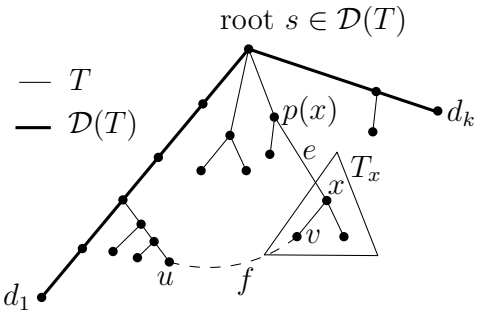

Figure 4.1: *A MDST T rooted at a node s on its diameter $\mathcal{D}(T)$, a failing edge e, and a swap edge $f = (u, v)$ for e. The bold line segments denote the diameter $\mathcal{D}(T)$ of T.*

swap tree. Our new observations allow for a simpler algorithm than the previous; we use only fundamental data structures.

This chapter is organized as follows: In Section 4.3, we introduce some terminology specific to MDSTs. Section 4.4 contains some key observations for evaluating swap edges in MDSTs. We explain how to compute all best swaps for failing edges lying on the diameter in Section 4.5, and for failing edges not lying on the diameter in Section 4.6. Finally, Section 4.7 explains how our results carry over to the $\{r, \max\}$-problem in shortest paths trees considered in [67].

The results in this chapter are the sole work of the author of this dissertation. They have been published in [35].

4.3 Terminology

The following notation is illustrated in Figure 4.1. Given a spanning tree $T = (V, E_T)$ of G, let $\mathcal{D}(T) := \langle d_1, d_2, \ldots, d_k \rangle$ denote a *diameter* of T, that is, a longest path in T. In the rest of this chapter, we measure distances in the given spanning tree T, not in the underlying graph G itself. Moreover, although G may have several MDSTs, the symbol T shall always refer to the same MDST of G, that is, the one given in the input. Similarly, we assume that $\mathcal{D}(T)$ always denotes the same diameter of T. We denote by $\mathcal{P}_{e/f}$ a longest path in $T_{e/f}$ among all paths containing the swap edge f.

4.4 The Quality of a Swap Edge f for a Failing Edge e

We start with a number of observations, all of which are used in our algorithm. Our first observation is that if the diameter of $T_{e/f}$ is longer than $|\mathcal{D}(T)|$, then the new diameter must go through f. More precisely:

Lemma 4.2. *For a given failing edge e of the MDST T, the length of the diameter of $T_{e/f}$ is*

$$|\mathcal{D}(T_{e/f})| = \max\{|\mathcal{D}(T)|, |\mathcal{P}_{e/f}|\}.$$

Proof. Let T_1 and T_2 be the parts into which T is split if e is removed. It is easy to see that

$$|\mathcal{D}(T_{e/f})| = \max\{|\mathcal{D}(T_1)|, |\mathcal{D}(T_2)|, |\mathcal{P}_{e/f}|\}. \tag{4.1}$$

Since T is a MDST, we have

$$|\mathcal{D}(T_{e/f})| \geq |\mathcal{D}(T)|. \tag{4.2}$$

Because T_1 and T_2 are contained in T,

$$|\mathcal{D}(T_1)| \leq |\mathcal{D}(T)| \quad \text{and} \quad |\mathcal{D}(T_2)| \leq |\mathcal{D}(T)|. \tag{4.3}$$

If $|\mathcal{P}_{e/f}| \geq |\mathcal{D}(T)|$, it is clear that $|\mathcal{P}_{e/f}|$ is a largest term in (4.1), so the claim holds. On the other hand, if $|\mathcal{P}_{e/f}| < |\mathcal{D}(T)|$, then either T_1 or T_2 must contain a diameter of length exactly $|\mathcal{D}(T)|$ (otherwise, either (4.2) or (4.3) would be violated). Thus, the claim holds also in this case. □

In our algorithm, we always judge swap edges only according to $|\mathcal{P}_{e/f}|$, instead of $|\mathcal{D}(T_{e/f})|$. This causes no problem because any swap edge f for which $|\mathcal{P}_{e/f}| < |\mathcal{D}(T_{e/f})|$ is a best swap edge for e, since in this case $|\mathcal{D}(T_{e/f})| = |\mathcal{D}(T)|$.

For a given tree $T = (V, E_T)$, and a given node $r \in T$, let $\mathcal{L}(T, r)$ denote the length of a longest simple path in T which starts in node r. Note that for $f = (u, v)$, $\mathcal{P}_{e/f}$ is composed of three parts: the longest path in $T - e$ starting in u, the longest path in $T - e$ starting in v, and the edge f itself. Thus, we have $|\mathcal{P}_{e/f}| = l(f) + \mathcal{L}(T - e, u) + \mathcal{L}(T - e, v)$. The following lemma shows how to compute $\mathcal{L}(T, r)$ efficiently for any given node $r \in V$.

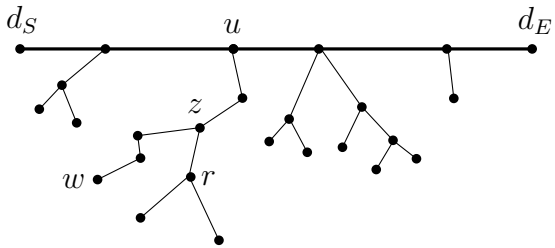

Figure 4.2: *Illustration for Lemma 4.3.*

Lemma 4.3. *Let $T = (V, E_T)$ be a tree, and let $\mathcal{D}(T) = \langle d_S, \ldots d_E \rangle$ be a diameter of T with endpoints d_S and d_E. Then, the length of a longest simple path inside T starting in $r \in V$ is*

$$\mathcal{L}(T, r) = \max\Big\{d(r, d_S), d(r, d_E)\Big\}.$$

Proof. In contradiction, assume there exists a node $w \in V$ such that it holds $d(r, w) > \max\{d(r, d_S), d(r, d_E)\}$. If r is a node on $\mathcal{D}(T)$, it follows that there exists a path of length

$$\begin{aligned} d(r, w) + \min\{d(r, d_S), d(r, d_E)\} &> \\ \max\{d(r, d_S), d(r, d_E)\} + \min\{d(r, d_S), d(r, d_E)\} &= |\mathcal{D}(T)| \end{aligned}$$

in T, a contradiction.

If r is not on $\mathcal{D}(T)$, let u be the node on $\mathcal{D}(T)$ closest to r (see Figure 4.2). If u lies on the path from r to w, then the above case shows that starting from u, no path, including the path leading to w, can be longer than $\max\{d(r, d_S), d(r, d_E)\}$. Hence, u cannot lie on the path from r to w. Let z be the node on the path from r to w closest to u (possibly, $z = r$). From $d(r, w) > \max\{d(r, d_S), d(r, d_E)\}$, we have

$$d(z, w) > \max\{d(z, d_S), d(z, d_E)\} \geq \max\{d(u, d_S), d(u, d_E)\}$$

and $d(u, w) > \max\{d(u, d_S), d(u, d_E)\}$. But this implies that the simple path from w to u and further to d_S or d_E, whichever is further from u, has length $d(u, w) + \max\{d(u, d_S), d(u, d_E)\} > 2 \cdot \max\{d(u, d_S), d(u, d_E)\} \geq |\mathcal{D}(T)|$, a contradiction. □

We now show how, given the endpoints of a diameter of T, one can compute $\mathcal{L}(T, r)$ for any given node r in constant time, after a preprocessing step requiring $O(n)$ time. We root the tree T at any node, and augment it with

- a labeling of the nodes which allows to obtain the nearest common ancestor (called $\mathrm{nca}(a, b)$ for two nodes a, b) of two given nodes in constant time [40];
- in every node x, store its distance to the root, called $\mathrm{toRoot}(x)$;

This information allows to compute the distance between two arbitrary nodes a and b in the tree T (and thus $\mathcal{L}(T, r)$) in constant time:

$$\begin{aligned} d(a,b) &= d(a, \mathrm{nca}(a,b)) + d(b, \mathrm{nca}(a,b)) \\ &= \mathrm{toRoot}(a) + \mathrm{toRoot}(b) - 2 \cdot \mathrm{toRoot}(\mathrm{nca}(a,b)). \end{aligned}$$

In our algorithm, we distinguish between failing edges on the diameter, called *diameter edges*, and failing edges not on the diameter, called *non-diameter* edges. If the given tree has several diameters, we select one and use the same throughout the algorithm. This guarantees that each edge is either a diameter edge or a non-diameter edge, and that this classification is consistent.

4.5 Best Swap Edges for Failing Diameter Edges

In this section, we show how to compute the best swap edges for all failing edges which lie on the diameter $\mathcal{D}(T)$ in time $O(m \log n)$ and $O(m)$ space.

Due to Lemma 4.2, a given swap edge f for a failing edge e can be evaluated by computing the lengths of the two longest paths starting at its endpoints. It turns out that these lengths can always be found by only considering paths which visit the diameter:

Lemma 4.4. *Consider a tree $T = (V, E_T)$, a diameter $\mathcal{D}(T)$ of it with endpoints d_S, d_E, a failing edge e on $\mathcal{D}(T)$, and an arbitrary node $r \in V$. Let u be the node on $\mathcal{D}(T)$ closest to r. One of the longest paths in $T - e$ starting from r contains the node u.*

Proof. Assume w.l.o.g. that r lies in the same connected component of $T-e$ as d_S. The proof resembles the proof of Lemma 4.3: Assume in contradiction that no longest path in T starting from r contains u, and let $\langle r, \ldots, w\rangle$ be such a path. Let z be the node on the path from r to w closest to u (possibly, $z = r$). We have $d(z, w) > d(z, d_S)$, therefore also $d(u, w) > d(u, d_S)$. This is a contradiction, because the path in T going from d_E to u and further to w would then be longer than $|\mathcal{D}(T)|$. □

Due to the above lemma, a longest path starting in any endpoint r of a given swap edge can always be found by first going to the node $u \in \mathcal{D}(T)$ closest to r. From there, a longest path may either continue to the end of the diameter (d_S or d_E), or cross at least one edge towards the failing edge e, and possibly leave the diameter again. Note that going from u towards d_S, but leaving the diameter again, cannot be longer than continuing until d_S.

For finding the length of a longest path starting from u efficiently, we compute two values $\mu_S(d_i, d_{i+1})$ and $\mu_E(d_i, d_{i+1})$ for every node d_i, $i = 1, \ldots, k-1$ on the diameter. By $\mu_S(d_i, d_{i+1})$ (respectively $\mu_E(d_i, d_{i+1})$), we denote the length of a longest path in T starting at d_S (d_E) and not crossing the edge (d_i, d_{i+1}). Formally, we have ($d_S := d_1, d_E := d_k$):

$$\mu_S(d_1, d_2) = \mu_E(d_{k-1}, d_k) = 0$$
$$\mu_S(d_i, d_{i+1}) = \max\{\mu_S(d_{i-1}, d_i), d(d_S, d_i) + h(d_i)\}$$

for $i = 2, 3, \ldots, k-1$, and

$$\mu_E(d_i, d_{i+1}) = \max\{\mu_E(d_{i+1}, d_{i+2}), d(d_E, d_{i+1}) + h(d_{i+1})\}$$

for $i = k-2, k-3, \ldots, 1$, where $h(d_i)$ denotes the length of a longest path starting in d_i, and not using any edges on the diameter $d_S, \ldots d_E$. It is easy to see that if T is rooted at a node on the diameter, then $h(d_i)$ can be computed for all d_i on the diameter in $O(n)$ time by traversing T in postorder. Thus, the values $\mu_S(d_i, d_{i+1}), \mu_E(d_i, d_{i+1})$ can be computed for all d_i on the diameter in $O(n)$ time.

The following lemma describes how to efficiently compute the longest path in $T - (d_i, d_{i+1})$ starting from any node r.

Lemma 4.5. *Consider any fixed diameter $d_S, \ldots, d_E$ of a tree T, any node r of T, and a failing edge (d_i, d_{i+1}) on the diameter. Let u be*

the node on the diameter closest to r. The length of a longest path in $T - (d_i, d_{i+1})$ starting in r is given by

$$d(r,u) + \max\{d(u,d_S), \mu_S(d_i,d_{i+1}) - d(u,d_S)\}$$

if r lies in the same connected component of $T - (d_i, d_{i+1})$ as d_S, and

$$d(r,u) + \max\{d(u,d_E), \mu_E(d_i,d_{i+1}) - d(u,d_E)\}$$

otherwise.

Proof. We only prove the case where r lies in the same connected component of $T - (d_i, d_{i+1})$ as d_S (the other is completely symmetric). Lemma 4.4 shows that a longest path first goes from r to u without loss of generality. We now distinguish two cases, depending on whether a longest path starting from u inside $T - (d_i, d_{i+1})$ can be achieved by using the edge incident to u which leads towards d_S. In this case, a longest path is clearly the one that goes from u until d_S (any other path cannot be longer in this case because $d_S, \ldots, d_E$ is a diameter of T). Otherwise, there must exist a path in $T - (d_i, d_{i+1})$ starting in u which is longer than $d(u, d_S)$, and which does not use the edge incident to u leading towards d_S. In this case, the length of such a path is exactly $\mu_S(d_i, d_{i+1}) - d(u, d_S)$. Furthermore, in this case a longest path in $T - (d_i, d_{i+1})$ starting in d_S must consequently be longer than $2d(u, d_S)$, that is, $\mu_S(d_i, d_{i+1}) > 2d(u, d_S)$. Hence, in any case the correct length can be expressed as $d(r,u) + \max\{d(u,d_S), \mu_S(d_i,d_{i+1}) - d(u,d_S)\}$. □

4.5.1 Using Virtual Swap Edges

For any node v, let $\mathrm{nc}(v)$ be the node on the diameter which is closest to v (possibly, $\mathrm{nc}(v) = v$). According to Lemma 4.5, the value $|\mathcal{P}_{e/f}|$ of a particular swap edge $f = (u, v)$ for any failing edge e on the diameter is one of the following four terms (assuming that u lies in the same component of $T - e$ as d_S, and v lies in the same component of $T - e$ as d_E):

1. $d(u, d_S) + d(v, d_E) + l(f)$
2. $d(u, d_S) + \mu_E(e) - d(\mathrm{nc}(v), d_E) + d(v, \mathrm{nc}(v)) + l(f)$

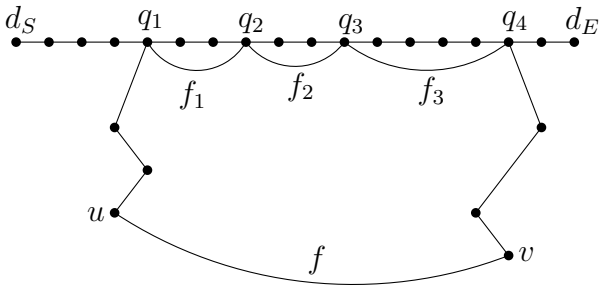

Figure 4.3: *Replacing a swap edge f by three virtual swap edges f_1, f_2, f_3.*

3. $\mu_S(e) - d(\mathrm{nc}(u), d_S) + d(u, \mathrm{nc}(u)) + d(v, d_E) + l(f)$
4. $\mu_S(e) - d(\mathrm{nc}(u), d_S) + d(u, \mathrm{nc}(u)) + \mu_E(e) - d(\mathrm{nc}(v), d_E) + d(v, \mathrm{nc}(v)) + l(f)$.

Note that in all above terms, the part depending on the failing edge e is independent of f. Thus, if two swap edges f' and f'' for edge e are such that their values $|\mathcal{P}_{e/f'}|$ and $|\mathcal{P}_{e/f''}|$ both correspond to the same of the four terms above, then we can omit the terms $\mu_S(e)$ and $\mu_E(e)$ when comparing their quality, without affecting the comparison.

Furthermore, note that since $\mu_S(d_i, d_{i+1})$ is monotonically increasing in i, all the failing edges (d_i, d_{i+1}) for which the equation

$$\max\{d(u, d_S), \mu_S(d_i, d_{i+1}) - d(u, d_S)\} = d(u, d_S)$$

holds form a connected path, as do all the failing edges for which $\max\{d(u, d_S), \mu_S(d_i, d_{i+1}) - d(u, d_S)\} = \mu_S(d_i, d_{i+1}) - d(u, d_S)$ (the same holds for d_E). Thus, for a given swap edge $f = (u, v)$, the set of diameter edges can be divided into at most three sets, each composing a path, such that for each set, f's value is defined by a specific one of the four terms above. We denote the endpoints of these paths by q_1, q_2, q_3, q_4. Note that $q_1 = \mathrm{nc}(u)$ and $q_4 = \mathrm{nc}(v)$.

The above observations lead to the idea of introducing *virtual swap edges* which replace the original swap edges, as follows (see also Figure 4.3):

A virtual swap edge f_i consists of its two endpoints, its *type*, and its *value*. The two endpoints define a path on the spanning tree T,

which is equal to the set of diameter edges for which f_i is a swap edge. The type of a virtual swap edge is a number in $1, 2, 3, 4$. By definition, two virtual swap edges can only be compared if they have the same type. The value of a virtual swap edge is a rational number. By construction, the quality of each virtual swap edge for a given failing edge e is identical to the quality of the original swap edge which it is replacing. Each swap edge f is replaced by at most three "virtual" swap edges f_1, f_2, f_3 in the following way:

- The endpoints are: $f_1 = (q_1, q_2)$, $f_2 = (q_2, q_3)$, $f_3 = (q_3, q_4)$.
- For every failing edge e on $\mathcal{D}(T)$ such that $f \in S(e)$, exactly one of f_1, f_2, f_3 is a swap edge.
- The value of each $f_i = (u, v)$ is one of the terms shown above, except for the part depending on e. Thus, it is either of

  1. $d(u, d_S) + d(v, d_E) + l(f)$,
  2. $d(u, d_S) - d(\mathrm{nc}(v), d_E) + d(v, \mathrm{nc}(v)) + l(f)$,
  3. $d(v, d_E) - d(\mathrm{nc}(u), d_S) + d(u, \mathrm{nc}(u)) + l(f)$,
  4. $d(u, \mathrm{nc}(u)) - d(\mathrm{nc}(u), d_S) + d(v, \mathrm{nc}(v)) - d(\mathrm{nc}(v), d_E) + l(f)$.

  The number of the term used to compute this value corresponds to the *type* of the virtual swap edge f_i.

Note that although there are four different types of virtual swap edges, each individual (original) swap edge is replaced by at most three different virtual swap edges, whose types are all different. In the following, we assume that a swap edge f is replaced by exactly three virtual swap edges; if fewer virtual swap edges are required, the adaptation of the method we describe is straightforward. Let us summarize.

Lemma 4.6. *The set of all swap edges can be replaced by at most three times as many virtual swap edges, each having one of four types and a value, such that the quality of two swap edges of the same type can be compared based solely on their values. Moreover, this transformation can be carried out using $O(m \log n)$ time and $O(m)$ space.*

Proof. For each swap edge f which may replace a failing edge on the diameter, we sequentially compute the virtual swap edges replacing it, as defined above. To obtain q_2, q_3, one needs to find the node $d_{i'}$ for which $\max\{d(u,d_S), \mu_S(d_{i'}, d_{i'+1}) - d(u,d_S)\} \neq \max\{d(u,d_S), \mu_S(d_{i'-1}, d_{i'}) - d(u,d_S)\}$ and the node $d_{i''}$ for which

$$\max\{d(v,d_E), \mu_E(d_{i''}, d_{i''+1}) - d(v,d_E)\} \neq \max\{d(v,d_E), \mu_E(d_{i''-1}, d_{i''}) - d(v,d_E)\}.$$

Using binary search on the sequence of diameter edges from (d_1, d_2) to (d_{k-1}, d_k), this can be easily accomplished in $O(\log n)$ time per swap edge, amounting to $O(m \log n)$ time in total. Then, q_2 is defined as the node among $d_{i'}, d_{i''}$ which is closer to d_S, and q_3 the other node among $d_{i'}, d_{i''}$. □

Using the virtual swap edges, we compute, for each of the four types $t \in \{1,2,3,4\}$ separately, the best virtual swap edge for all failing edges on the diameter in $O(m \log n)$ time and $O(m)$ space, with the following simple scanline algorithm:

1. Initialize an empty Heap H_t. The virtual swap edges which are later inserted into H_t are to be (heap-)ordered by their values.
2. Consider all failing edges $e_i = (d_i, d_{i+1})$ on the diameter sequentially, for $i = 1, 2, \ldots, k$. For each $e_i = (d_i, d_{i+1})$:
   - add to H_t all those virtual swap edges whose left endpoint (i.e., the one closer to d_1) is d_i.
   - remove from H_t all those virtual swap edges whose right endpoint is d_i.
   - store the current minimum of H_t as $best(e_i, t)$.

Then, for each e_i and each type t, replace the virtual swap edge $best(e_i, t)$ by its corresponding swap edge. This yields at most four potential best swap candidates for each diameter edge e_i, one of which is a best swap. The best swap edges are then found in time $O(n)$ by simply computing $|\mathcal{P}_{e/f}|$ explicitly (and in constant time) for each of these selected $O(n)$ candidates.

As we have shown above, replacing each and every swap edge by its virtual swap edges requires $O(m \log n)$ time and $O(m)$ space, and increases the number of swap edges by a factor of at most three. Summarizing, we have:

Theorem 4.7. *All best swap edges for failing edges on a chosen diameter can be computed in $O(m \log n)$ time and $O(m)$ space.*

4.6 Best Swap Edges for Failing Non-Diameter Edges

In this section, we describe an algorithm to compute the best swap edges for those tree edges which do not lie on the chosen diameter $\mathcal{D}(T)$ of the given MDST T. We will show that this algorithm runs in $O(m \log n)$ time and requires $O(m)$ space.

For our approach, we root T in an arbitrary node *on the diameter*, and label all edges by their occurrence in a postorder traversal.

To begin, let $f = (u, v)$ be an edge in $E \backslash E_T$ such that u is a descendant of v in T, and such that the path from u to v in T does not contain any edge of the diameter $\mathcal{D}(T)$. We call such an edge a *backedge*. For a backedge $f = (u, v)$, we call the endpoint u the *lower* endpoint of f, and v the *upper* endpoint of f. In the following, we assume for ease of exposition that all edges in $E \backslash E_T$ are backedges. In Section 4.6.6, we describe how to adapt our algorithm to work without this assumption.

Consider the sequence of (non-diameter) tree edges $e_1, \ldots, e_k$ in the path from u to v, starting with the edge incident to u: how does $|\mathcal{P}_{e_i/f}|$ depend on the e_i? Since the failing edge e_i is not on the diameter $\mathcal{D}(T)$, the connected component of $T - e_i$ containing v still contains $\mathcal{D}(T)$. According to Lemma 4.3, the longest path in $T - e_i$ starting in v is therefore the same for all edges $e_1, \ldots, e_k$, and thus $\mathcal{L}(T - e_i, v) = \mathcal{L}(T, v)$.

On the other hand, the longest path in $T - e_i$ starting in u may be different for different failing edges e_i. To characterize the structure of these paths, we introduce a new concept: The *central edge* of a tree's diameter is the edge on the diameter which contains the center of the diameter. More precisely, this is the diameter edge whose removal splits it into two parts whose difference in length is minimum (there could be two edges satisfying this definition; any of them can be chosen). Note that the position of the central edge determines in which direction a longest path starting in a particular node goes (again using Lemma 4.3): all longest paths which start on one side of the central

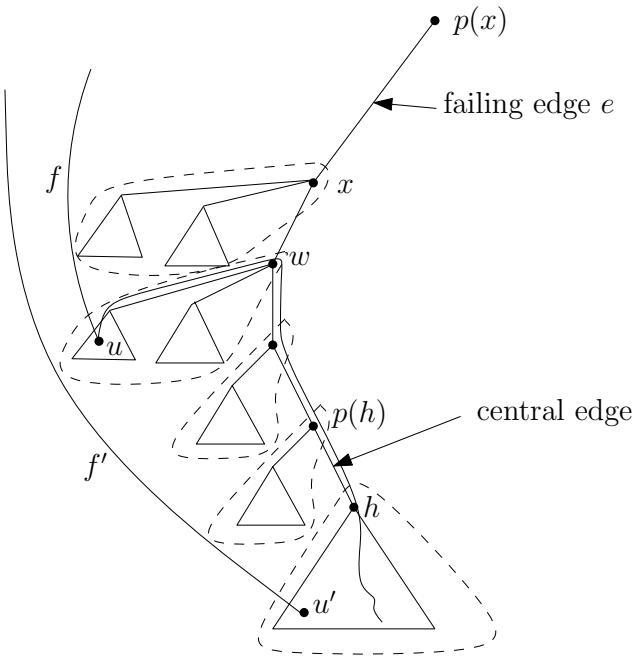

Figure 4.4: *Grouping of swap edges according to their endpoints in T_x. Sets of endpoints whose swap edges are grouped together are enclosed in dashed shapes.*

edge go to the opposite end of the diameter.

We now focus on a particular failing (non-diameter) edge $e = (x, p(x))$ for which the best swap is to be computed. Let $(h, p(h))$ be the central edge of T_x. By Lemma 4.3, the longest path starting in u inside T_x will contain $(h, p(h))$. This fact allows to partition the set $S(e)$ of swap edges for e into groups as follows (see Figure 4.4):

- All swap edges having their lower endpoint below the central edge $(h, p(h))$ will have a longest path going up towards this edge, and then further on to the furthest node in T_x (this furthest node is the endpoint of the diameter of T_x which lies outside of T_h, and which is precomputed). We call this the *lower* group, and denote it by $\mathcal{G}_{\text{lower}}(x)$. Formally, $\mathcal{G}_{\text{lower}}(x) := \{(u, v) \in E \backslash E_T | u \in T_h \ \wedge \ v \notin T_x\}$.

- All swap edges having their lower endpoint u above the central edge (i.e., not in T_h) will have a longest path which first leads to some node on the path from $p(h)$ to x, then continue down towards the central edge, and finally going into a deepest leaf in T_h (this node is the endpoint of T_x's diameter which lies inside T_h, which is also precomputed). We can partition these swap edges into groups distinguished by the node $w = \mathrm{nca}(u, h)$, the first node on the path from $p(h)$ to x contained in their longest path in $T - e$ starting in u. These groups are called the *upper* groups, and denoted by $\mathcal{G}_{\mathrm{upper}}(x, w)$. Formally: $\mathcal{G}_{\mathrm{upper}}(x, w) := \{(u, v) \in E \backslash E_T | u \in T_x \backslash T_h \ \wedge \ \mathrm{nca}(u, h) = w \ \wedge \ v \notin T_x\}$.

This grouping is helpful for computing best swap edges, due to the following fact:

Lemma 4.8. *In any group $\mathcal{G}_{\mathrm{upper}}(x, w)$ or $\mathcal{G}_{\mathrm{lower}}(x)$, a best swap candidate for the failing edge $e = (x, p(x))$ is a swap edge $f = (u, v)$ for which $\mathcal{L}(T, v) + l(f) + \mathrm{toRoot}(u)$ is minimum.*

Proof. For swap edges in the lower group, all "longest paths" in T_x are identical after crossing the central edge. For each upper group corresponding to some node w, all "longest paths" in T_x are identical after reaching w. □

In order to compare the best candidates of different upper groups, an additional offset has to be added to each candidate's value, such that the so-called *updated value* of a candidate f is exactly equal to $|\mathcal{P}_{e/f}|$. For $f = (u, v)$ with $\mathrm{nca}(u, h) = w$, this updated value is

$$\mathcal{L}(T, v) + l(f) + \mathrm{toRoot}(u) - \mathrm{toRoot}(w) + d(w, v_{deep}), \quad (4.4)$$

where v_{deep} is the endpoint of $\mathcal{D}(T_x)$ in T_h. For the following, it is useful to denote by $\mathcal{GR}(x)$ the union of all groups associated with a given edge $e = (x, p(x))$.

4.6.1 Relations between Groups for Different Non-Diameter Tree Edges

There is a close connection between the central edges of a subtree T_x rooted at a node x and the central edges of the subtrees of this node's children. Indeed, the following lemma is easy to prove:

Lemma 4.9 (Proof omitted)**.** *Consider a tree edge $e = (x, p(x))$ and the child $c \in C(x)$ for which the central edge of $\mathcal{D}(T_x)$ is either (c, x) or an edge in T_c. Then, the central edge of $\mathcal{D}(T_x)$ lies on the path from the central edge of $\mathcal{D}(T_c)$ to e (possibly, the central edges of $\mathcal{D}(T_c)$ and of $\mathcal{D}(T_x)$ are identical).*

Thus, the central edge only moves "upwards" when failing edges are visited in postorder: it never occurs that the central edge of T_x lies below the central edge of T_d for any descendant d of x. This implies that for any particular backedge $f = (u, v)$, if a longest path in $T - e_i$ starting in u does not visit any child of u, then nor will a longest path in any $T - e_j$, $j > i$.

Recall that we consider all (non-diameter) failing edges in a postorder. In the following, we show how the groups of swap edges for a non-diameter tree edge $e = (x, p(x))$ relate to the groups of previously considered failing edges. Later, we exploit these relations using a collection of suitable data structures.

The set of swap edges for edge $e = (x, p(x))$ can be expressed as

$$S(e) = \text{start-at}(x) \cup \left\{ \bigcup_{q \in C(x)} S((q, x)) \right\} \setminus \text{end-at}(x),$$

where start-at(x) is the set of swap edges whose lower endpoint is x, and end-at(x) is the set of swap edges whose upper endpoint is x. We now describe how $S(e)$ is partitioned into the lower group and all the upper groups of e.

Let $c \in C(x)$ be the child of x for which the central edge of $\mathcal{D}(T_x)$ is either (c, x) or an edge in T_c. Furthermore, let $(g, p(g))$ be the central edge of $\mathcal{D}(T_c)$ and let $(h, p(h))$ be the central edge of $\mathcal{D}(T_x)$ (see Figure 4.5).

Clearly, all swap edges which belong to the upper group of e associated with $w = x$ are

$$\mathcal{G}_{\text{upper}}(x, x) = \text{start-at}(x) \cup \left\{ \bigcup_{q \in C(x), q \neq c} S((q, x)) \right\} \setminus \text{end-at}(x)$$

$$= \text{start-at}(x) \cup \left\{ \bigcup_{q \in C(x), q \neq c} \mathcal{GR}(q) \right\} \setminus \text{end-at}(x).$$

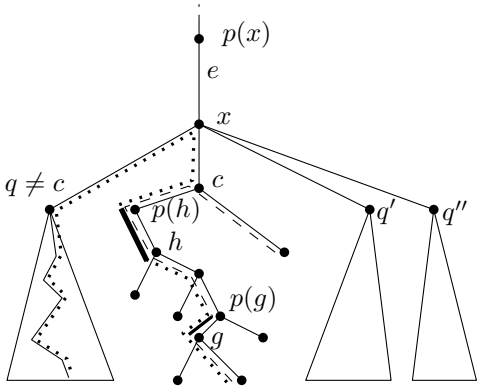

Figure 4.5: *A failing edge $e = (x, p(x))$, the central edge $(h, p(h))$ of $\mathcal{D}(T_x)$, the child $c \in C(x)$ whose subtree contains h, and the central edge $(g, p(g))$ of $\mathcal{D}(T_c)$.*

For any $w \neq x$ on the path from $p(h)$ to c, thanks to Lemma 4.9, we can express the set of swap edges in the upper group of e associated with w as

$$\mathcal{G}_{\text{upper}}(x, w) = \mathcal{G}_{\text{upper}}(c, w) \setminus \text{end-at}(x).$$

Finally, the swap edges belonging to the lower group of e are

$$\mathcal{G}_{\text{lower}}(x) = S((c, x)) \cap \left\{ \bigcup_{d \in T_h} \text{start-at}(d) \right\} \setminus \text{end-at}(x)$$

$$= \left(\mathcal{G}_{\text{lower}}(c) \cup \left\{ \bigcup_{d \in \langle p(g), \ldots, h \rangle} \mathcal{G}_{\text{upper}}(c, d) \right\} \right) \setminus \text{end-at}(x).$$

4.6.2 Our Data Structure

Our approach visits all non-diameter tree edges in postorder, and computes a best swap edge for each of them sequentially. In order to leverage our observations about connections between best swaps for different edges, we associate a compound data structure, denoted by $\text{GroupsDS}(x)$, with each considered edge $e = (x, p(x))$. This structure contains a representation of the group $\mathcal{G}_{\text{lower}}(x)$ and all groups

$\mathcal{G}_{\text{upper}}(x, w)$ for $w \in \langle p(h), \ldots, x\rangle$ (recall $(h, p(h))$ denotes the central edge), from which a best swap for e can be extracted in constant time. Moreover, it is designed such that $\text{GroupsDS}(x)$ can be efficiently composed of their counterparts of previously visited edges.

Lemma 4.9 implies that visiting all (non-diameter) failing edges in postorder results in a corresponding sequence of central edges that is also in postorder (although this may be only a subset of all tree edges). This is crucial for the correctness of our approach, because once we used a data structure $\text{GroupsDS}(x')$ associated with a previously visited edge $(x', p(x'))$ to compute $\text{GroupsDS}(x)$, the old data structure $\text{GroupsDS}(x')$ is no longer available (as it has been altered/merged into $\text{GroupsDS}(x)$).

In $\text{GroupsDS}(x)$, each group is represented by a Fibonacci-Heap (short *F-Heap* in the following). It is widely known that F-Heaps support all of the operations

- `make-heap`,
- `insert(·)`,
- `find-min`,
- `merge(·,·)`

in $O(1)$ amortized time, and the operations `delete(·)` as well as `delete-min` in $O(\log n)$ amortized time, where n is the number of elements in the F-Heap [32].

Each swap edge f contained in a group $\mathcal{G}$ is stored in the corresponding F-Heap, using the value $\mathcal{L}(T, v) + l(f) + \text{toRoot}(u)$, which we call the *invariant value* of f, as its key[1]. Note crucially, that this value is independent of the failing edge, and according to Lemma 4.8 the minimum element in the F-Heap corresponding to a group is the best swap (in this group) for the currently considered failing edge. Furthermore, even when this F-Heap is later altered, by inserting or deleting some swap edges, or by merging the F-Heap with another, the value associated with a given swap edge need never be changed. Specifically, $\text{GroupsDS}(x)$ contains:

[1] Technically speaking, the key must be made unique by using the tuple $(\mathcal{L}(T, v) + l(f) + \text{toRoot}(u), f)$ as key, where comparisons are based mainly on the first component, and the second is only used to break ties. We omit this detail in the main text for ease of exposition.

1. the diameter $\mathcal{D}(T_x)$ (given by the labels of its endpoints)

2. the central edge of $\mathcal{D}(T_x)$: $(h, p(h))$ (given by the labels of h and $p(h)$). If x is a leaf node in T, h is undefined, and in this case, it is assumed that $p(h) = x$.

3. the child $c \in C(x)$ which contains the central edge $(h, p(h))$. More precisely: either (c, x) is the new central edge, or the central edge is contained in the subtree T_c (see Figure 4.5).

4. a list $\mathrm{heaplist}(x)$ of F-Heaps, containing, for each node w on the path from $p(h)$ to x, an F-Heap $\mathrm{FH}_{\mathrm{upper}}(x, w)$ of all swap edges in $\mathcal{G}_{\mathrm{upper}}(x, w)$. The order of the F-Heaps in the list corresponds to the order of the respective nodes w (lowest node first).

5. an F-Heap $\mathrm{FH}_{\mathrm{lower}}(x)$ of all swap edges whose lower endpoint lies in T_h.

In principle, the best swap for e is found by choosing the candidate with minimum updated value among the best of each group. Doing this naively would require at least linear time in the number of groups, i.e., at least linear in the number of nodes between e and the central edge. To expedite this process during the induction, we use an ordinary minimum heap which contains the updated values of the best candidate from each upper group. The best swap edge is then either the minimum element in this heap, or the best candidate from the lower group. Thus, $\mathrm{GroupsDS}(x)$ additionally contains the following item:

6. an ordinary minimum heap $\mathrm{CandHeap}(x)$, containing for each F-Heap in $\mathrm{heaplist}(x)$ the best swap candidate, (heap-)ordered by their quality (i.e., their $\mathcal{P}_{e/f}$- lengths as defined in Equation 4.4).

From this information, a best swap edge for $(x, p(x))$ is found in constant time by comparing the best candidate in $\mathrm{CandHeap}(x)$ with the best candidate in $\mathrm{FH}_{\mathrm{lower}}(x)$, and taking the better of the two:

Lemma 4.10. *Given the data structure* $\mathrm{GroupsDS}(x)$ *associated with the non-diameter tree edge* $e = (x, p(x))$*, a best swap edge for* e *can be obtained in constant time.*

4.6.3 A Preprocessing Step

In this section, we explain how to compute the first three items of $\mathrm{GroupsDS}(x)$ for all x, i.e. the diameter $\mathcal{D}(T_x)$, the central edge of $\mathcal{D}(T_x)$ and the child $c \in C(x)$ containing it, in $O(n)$ time.

First, we traverse all nodes of the tree in postorder, to precompute the following information: In addition to $\mathrm{toRoot}(x)$, we associate with every node x the height of its subtree T_x, called $\mathrm{height}(x)$. Second, we compute $\mathcal{D}(T_x)$ for every node $x \in V$. This can be done in $O(n)$ time by traversing the tree in postorder, using induction: the diameter of the subtree rooted at a node x is either equal to the largest diameter found in any of its subtrees, or it is (roughly speaking) a path composed of the longest paths in those two of its subtrees which are deepest when first prolonged by the edge leading to x. Furthermore, it is straightforward to associate with each node x the child $c \in C(x)$ which contains the center of $\mathcal{D}(T_x)$.

During the same inductive computation, we associate with every node x the central edge of $\mathcal{D}(T_x)$. All of these central edges can be found in $O(n)$ time as follows: for every non-diameter edge $e_i = (x_i, p(x_i))$ in postorder, find the central edge $m_i = (h_i, p(h_i))$ by considering all non-diameter edges in the same order, and checking for each (in constant time) whether it is the central edge of $\mathcal{D}(T_{x_i})$. Due to Lemma 4.9, we know that either $m_i = m_{i-1}$, or m_i must come later than m_{i-1} in the given postorder. Thus, we consider each non-diameter edge at most once during this traversal.

4.6.4 Our Inductive Approach

Let us see how $\mathrm{GroupsDS}(x)$ can be constructed efficiently when visiting a new tree edge $e = (x, p(x))$ in the postorder traversal. First, we associate with each node $x \in V$ a list start-at(x) containing all swap edges whose lower endpoint is x, and a list end-at(x) containing all swap edges whose upper endpoint is x, in $O(m)$ total time.

Lemma 4.11. *In the base case of the induction, when x is a leaf, constructing $\mathrm{GroupsDS}(e)$ takes $O(1+|$start-at$(x)|)$ amortized time.*

Proof. Note that the central edge is undefined in this case, so we assume $p(h) = x$. Hence, $\mathrm{FH}_{\mathrm{lower}}(x)$ is the empty heap, which is constructed in constant time. $\mathrm{heaplist}(x)$ contains one entry, namely the

F-Heap $\mathrm{FH}_{\mathrm{upper}}(x, x)$, which is created by inserting into an empty F-Heap all edges in start-at(x), using their invariant values for the heap order. This step takes $O(1 + |\text{start-at}(x)|)$ amortized time. □

Lemma 4.12. *Consider any tree edge $e = (x, p(x))$, where x is not a leaf node of T. Provided that for all children $q \in C(x)$, $\mathrm{GroupsDS}(q)$ is available, the data structure $\mathrm{GroupsDS}(e)$ can be computed in $O(\log n + |C(x)| + (|\textit{start-at}(x)| + |\textit{end-at}(x)|) \cdot \log n + \alpha_x \cdot \log n)$ amortized time, where $\alpha_x = (\sum_{q \in C(x)} |\mathrm{heaplist}(q)|) - |\mathrm{heaplist}(x)|$.*

Proof. Let c be the child of x which contains the central edge of $\mathcal{D}(T_x)$, as computed in Section 4.6.3. We consider the path $\mathcal{Q} = \langle p(g), \ldots, h \rangle$ from the central edge of $\mathcal{D}(T_c)$ to the central edge of $\mathcal{D}(T_x)$. For the moment, ignore the fact that swap edges in end-at(x) must be removed. Using the relations from Section 4.6.1, we proceed as follows: $\mathrm{FH}_{\mathrm{upper}}(x, x)$ is obtained by merging all F-Heaps corresponding to a group in $\mathcal{GR}(q)$, for $q \in C(x), q \neq c$, and inserting all edges in start-at(x) into the resulting F-Heap, using their invariant values as keys. This step takes

$$O\left(\left(\text{start-at}(x) + \sum_{q \in C(x), q \neq c} |\mathrm{heaplist}(q)|\right) \log n\right)$$

amortized time. Similarly, $\mathrm{FH}_{\mathrm{lower}}(x)$ is obtained by merging all F-Heaps $\mathrm{FH}_{\mathrm{upper}}(c, d)$, for $d \in \langle p(g), \ldots, h \rangle$, and merging the resulting F-Heap with $\mathrm{FH}_{\mathrm{lower}}(c)$. This step takes $O((|\mathrm{heaplist}(c)| - |\mathrm{heaplist}(x)|) \log n)$ amortized time. Still ignoring the swap edges in end-at(x), note that for each $w \in \langle p(h), \ldots, c \rangle$, we have

$$\mathrm{FH}_{\mathrm{upper}}(x, w) = \mathrm{FH}_{\mathrm{upper}}(c, w).$$

It remains to delete all swap edges in end-at(x) from all these data structures. This can be achieved in $O(|\text{end-at}(x)| \log n)$ amortized time, as follows. For each edge $f = (u, v) \in \text{end-at}(x)$, we compute $w = \mathrm{nca}(u, h)$ in constant time, and distinguish two cases: (i) If $w \neq h$, we know by construction that f must be contained in $\mathrm{FH}_{\mathrm{upper}}(x, w)$ (and in no $\mathrm{FH}_{\mathrm{upper}}(x, w')$, $w \neq w'$). We simply remove the edge from this F-Heap[2]. (ii) If $w = h$, then by con-

[2]In order to find the element in the given heap in $O(\log n)$ time, a dictionary is required. For simplicity, we omit this detail in the main text.

struction f must lie in $\mathrm{FH}_{\mathrm{lower}}(x)$. In this case, we remove f from $\mathrm{FH}_{\mathrm{lower}}(x)$. In both of these cases, if the removed element was the current minimum element of the F-Heap, then the corresponding entry in $\mathrm{CandHeap}(x)$ must be deleted, and the new minimum element inserted (unless the F-Heap is now empty).

Due to Lemma 4.9, $\mathrm{heaplist}(x)$ can be obtained from $\mathrm{heaplist}(c)$, where all F-Heaps corresponding to nodes on $\mathcal{Q}$ are removed, and adding the new F-Heap $\mathrm{FH}_{\mathrm{upper}}(x, x)$. In $\mathrm{CandHeap}(x)$, the candidate corresponding to each of these removed F-Heaps must be removed as well.

Furthermore, the best candidate of the new F-Heap $\mathrm{FH}_{\mathrm{upper}}(x, x)$ must be inserted in $\mathrm{CandHeap}(x)$ (unless $\mathrm{FH}_{\mathrm{upper}}(x, x)$ is empty). The exact value for this candidate, say f, is

$$\mathcal{L}(T, v) + l(f) + \mathrm{toRoot}(u) - \mathrm{toRoot}(x) + d(x, v_{deep}),$$

where v_{deep} is the endpoint of $\mathcal{D}(T_x)$ which lies below the current central edge. Note that $d(x, v_{deep}) = l((x, c)) + \mathrm{height}(T_c)$. □

4.6.5 Analysis

Theorem 4.13. *All best swap edges for failing edges not lying on the chosen diameter can be computed in $O(m \log n)$ time and $O(m)$ space.*

Proof. We have already seen in Section 4.6.3 that the preprocessing step can be completed in $O(n)$ time. It remains to analyze the total cost of all the inductive computations during the postorder traversal.

By Lemmas 4.11 and 4.12, the amortized time for visiting a tree edge $(x, p(x))$ is bounded by $O(\log n + |C(x)| + (|\text{start-at}(x)| + |\text{end-at}(x)|) \cdot \log n + \alpha_x \cdot \log n)$ amortized time, where

$$\alpha_x = \left(\sum_{q \in C(x)} |\mathrm{heaplist}(q)| \right) - |\mathrm{heaplist}(x)|.$$

Furthermore, it is easy to see that

$$\sum_{x \in V'} |\text{end-at}(x)| \leq m, \sum_{x \in V'} |\text{start-at}(x)| \leq m, \text{ and } \sum_{x \in V'} |C(x)| \leq n,$$

where $V' \subset V$ is the set of nodes v for which $(v, p(v))$ does not lie on the diameter of T. Finally, we have

$$\begin{aligned}\sum_{x \in V'} \alpha_x &\leq \sum_{x \in V, x \neq root} \alpha_x \\ &= \sum_{x \in V, x \neq root} \left(\Big(\sum_{q \in C(x)} |\text{heaplist}(q)| \Big) - |\text{heaplist}(x)| \right) \\ &= \Big(\sum_{x \in V} |\text{heaplist}(x)| \Big) - \sum_{x \in V, x \neq root} |\text{heaplist}(x)|) \\ &= |\text{heaplist}(root)|,\end{aligned}$$

and since $|\text{heaplist}(r)| \leq n$ for any node r, $\sum_{x \in V'} \alpha_x \leq n$.

Hence, summing up the time for traversing all tree edges, we obtain $O(n \log n + n + m \log n + m \log n + n \log n) = O(m \log n)$. As to the space complexity, note that at any time during the inductive computation, each swap edge is contained in at most one F-Heap. Therefore, the total amount of space required for these heaps is $O(m)$ at all times. For the other data structures used in our algorithm, the $O(m)$ space bound is obvious. □

4.6.6 Transforming Non-Tree Edges to Backedges

So far, we have assumed that all swap edges are backedges. We now describe how to replace any edge f by at most two "virtual" backedges, whose lengths are defined in such a way that the best swap edge computed by our algorithm is correct. That is, if the computed best swap for a given failing edge e is a virtual backedge, then replacing the virtual backedge by the edge f it represents yields a (non-backedge) swap edge for e with the same quality. Formally, we replace $f = (u, v)$ by $f_1 := (u, v_1)$ and $f_2 := (v, u_2)$ (recall that these tuples are ordered), with lengths $l(f_1) := l(f) + d(v, v_1)$ and $l(f_2) := d(u, u_2)$. If the path from u to v in T uses one or more edges of $\mathcal{D}(T)$, we define $v_1 := \text{nc}(u)$ and $v_2 := \text{nc}(v)$. Otherwise, we define $v_1 := \text{nca}(u, v)$ and $u_2 := \text{nca}(u, v)$. If it happens that $u = v_1$, we omit f_1, and if $v = u_2$, we omit f_2. To see why this replacement works, note that in both cases f_1 represents f exactly for all failing edges on the path in T from u to v_1, and f_2 represents f exactly for all edges from v to u_2 (i.e., for each non-diameter tree edge, exactly

one of $\{f_1, f_2\}$ represents f). Furthermore, the lengths of f_1 and f_2 are defined exactly such that for any failing edge e for which f_i is a swap edge, it holds $|\mathcal{D}(T_{e/f_i})| = |\mathcal{D}(T_{e/f})|$. Summarizing, we have:

Theorem 4.14. *All best swap edges for failing edges not on a chosen diameter can be computed in $O(m \log n)$ time and $O(m)$ space.*

4.7 Improved Swap Edge Computation for Shortest Paths Trees

In this section, we briefly outline how the results presented so far can be adapted to the "$\{r, \max\}$-problem" as defined in Section 3.3.2, for which the best previous algorithm required $O(n\sqrt{m})$ time and $O(m)$ space [67]. We show that our results can be applied to this problem by giving a reduction: Suppose a graph G, a source node r and a shortest paths tree T are given. Choose a value q which is larger than the total length of all the edges in G. We add to this graph two new nodes v_1, v_2, and three new edges: (r, v_1) with length q, (r, v_2) with length $q/2$, and (v_1, v_2) with length $q/2$. We denote the thus obtained graph by G'. Clearly, G' is again 2-edge-connected. Moreover, the edge (r, v_1) is so long that for every failing edge e of T, and for every swap edge candidate f for e, every diameter of $T_{e/f}$ must either contain the edge (r, v_1) or, alternatively, the edges (r, v_2) and (v_1, v_2). Therefore, the resulting diameter of any such swap edge will be the distance from its endpoint in S_1 to v_1. Note that this is exactly the distance from its endpoint in S_1 to r plus q. Moreover, note that if we add to T the edges (r, v_1) and (v_1, v_2), we obtain a MDST of the extended graph G' (clearly, no other spanning tree of G' can have a smaller diameter). Since our algorithm for computing best swap edges in a MDST always evaluates a given swap f for e by the value $|\mathcal{P}_{e/f}|$ (and not by $|\mathcal{D}(T_{e/f})|$), applying it to G' will always yield a swap edge for which $|\mathcal{P}_{e/f}|$ is minimum. By construction, for any e and f, $|\mathcal{P}_{e/f}|$ is exactly q plus the maximum distance from r to any point in S_2 (measured in the swap tree $T_{e/f}$). Hence, for every edge e of T, a best swap for e in G' (measured by the resulting diameter) is also a best swap for e in G (measured as the maximum distance from r to any node in S_2). The above transformation clearly takes only linear time in m and n, and so we have:

Theorem 4.15. *Given a 2-edge-connected graph $G = (V, E)$ with*

$n = |V|$ nodes and $m = |E|$ edges, a root node $r \in V$ and a shortest paths tree T of G rooted in r, all best swap edges of T can be computed in $O(m \log n)$ time and $O(m)$ space.

Chapter 5

A Distributed Algorithm for Minimum Diameter Spanning Trees

5.1 Motivation

In this chapter, we solve the *distributed All-Best-Swaps* problem for minimum diameter spanning trees (MDSTs) in a time- and message-efficient fashion. The provided algorithm complements the centralized solution of the previous chapter. It is useful if the network at hand is not planned by a central authority, and no global view of the network is available. In such a distributed setting, it is crucial that all best swaps are precomputed before a failure occurs: once a link failure is present, the network is disconnected, and finding a best swap at this time would be very difficult, if not impossible. Our solution addresses this setting in an efficient way: the costs of computing all best swap edges with our method are easily subsumed by the costs of constructing a MDST distributively using the state-of-the-art distributed algorithm [12]. Thus, it is cheap to precompute all the best swaps in addition to constructing a MDST initially.

We assume for our solution that no failures occur during the distributed computation of all best swaps. This may seem unrealistic under the assumption that transient failures occur, for which we want

to use swap edges. However, note that the distributed algorithm we propose is rather fast (its running time is linear in the hop-diameter of the network), so it is unlikely that even a single failure occurs during this short period of time (if a link does fail, we can just run the computation again once it has been repaired). Moreover, the precomputation of best swaps will usually be performed soon after the network has been set up, when its components are still new and therefore less prone to errors. On the other hand, during the operational lifespan of the network, which naturally lasts much longer than the initial computation of best swaps, failures are bound to occur.

5.2 Summary of Results

We give an asynchronous distributed algorithm for computing all best swaps of a MDST using no more than $O(n^* + m)$ messages of size $O(1)$ each. Here, the *size of a message* denotes the number of atomic values that it contains, such as node labels, edge lengths, path lengths etc., and n^* is the size of the transitive closure of the MDST with edges directed towards the node of the tree which initiates the computation. Therefore, under the natural assumption that each such value can be represented using $O(\log n)$ bits, our algorithm complies with the restrictions of the $\mathcal{CONGEST}$ model. Both n^* and m are very natural bounds: When each subtree triggers as many messages as there are nodes in the subtree, the size of the transitive closure describes the total number of messages. Furthermore, it seems inevitable that each node receives some information from each of its neighbors in G, taking into account each potential swap edge, resulting in $\Omega(m)$ messages. Indeed, we show in Section 5.7 that $\Omega(n^* + m)$ messages are necessary under some natural assumption. Our algorithm runs in $O(\|\mathcal{D}\|)$ time (in the standard sense, as explained in Section 2.2.2), where $\|\mathcal{D}\|$ is the hop-length of the diameter path of G; note that this is asymptotically optimal. The message and time costs of our algorithm are significantly lower than the costs of computing a MDST, which requires $O(n)$ time complexity and uses $O(nm)$ messages [12].

The main structure of our algorithm is the same as in the related work discussed in Chapter 3.3.2. Just like the best swaps algorithms for shortest paths trees (see [28, 71, 30]), our algorithm (like many fundamental distributed algorithms) exploits the structure of the tree

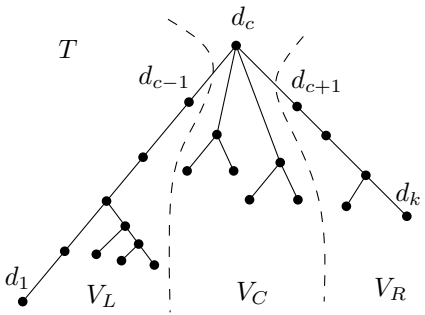

Figure 5.1: *A minimum diameter spanning tree T.*

under consideration. The minimum diameter spanning tree, however, is substantially different from shortest paths trees in that it requires a significantly more complex invariant to be maintained during the computation: We need to have just the right collection of pieces of information available so that on the one hand, these pieces can be maintained efficiently for changing failing edges, and on the other hand, they can be composed to reveal the diameter at the corresponding steps in the computation.

In Section 5.3, we introduce the terminology specific to this chapter. Section 5.4 states our assumptions about the distributed setting and explains the basic idea of our algorithm. In Section 5.5, we study the structure of diameter paths after swapping, and we propose an algorithm for finding best swaps. The algorithm uses information that is computed in a preprocessing phase, described in Section 5.6. In Section 5.7, we show that the message complexity of our algorithm is optimal under certain assumptions.

The results of this chapter were obtained in collaboration with Peter Widmayer and Nicola Santoro [36].

5.3 Terminology

Note that throughout this chapter, we measure distances in the given spanning tree T, not in the underlying graph G itself. The *hop-length* $\|\mathcal{P}\| := r - 1$ of a path $\mathcal{P} = \langle p_1, \ldots, p_r \rangle$ is the number of edges that $\mathcal{P}$ contains. Like in the previous chapter, given a spanning tree

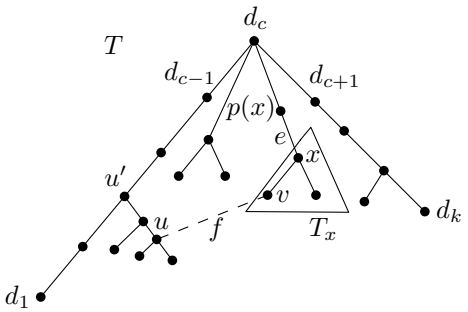

Figure 5.2: *A swap edge $f = (u, v)$ for $e = (x, p(x))$.*

$T = (V, E_T)$ of G, let $\mathcal{D}(T) := \langle d_1, d_2, \ldots, d_k \rangle$ denote a *diameter* of T, that is, a longest path in T (see Figure 5.1). Where no confusion arises, we abbreviate $\mathcal{D}(T)$ with $\mathcal{D}$. Furthermore, define the *center* d_c of $\mathcal{D}$ as a node such that the lengths of $\mathcal{D}_L := \langle d_1, d_2, \ldots, d_c \rangle$ and $\mathcal{D}_R := \langle d_c, d_{c+1}, \ldots, d_k \rangle$ satisfy $|\mathcal{D}_L| \geq |\mathcal{D}_R|$ and (under this condition) have the smallest possible difference $|\mathcal{D}_L| - |\mathcal{D}_R|$. We consider T to be rooted at d_c. Let V_L (L stands for "left") be the set of nodes in the subtree rooted at d_{c-1}, V_R the set of nodes in the subtree rooted at d_{c+1}, and V_C all other nodes.

A *local* swap edge of node z for some failing edge e is an edge in $S(e)$ incident to z. Throughout this chapter, let $e = (x, p(x))$ denote a failing edge and $f = (u, v)$ a swap edge, where v is a node inside T_x, and u a node in $T \backslash T_x$.

5.4 Algorithmic Setting and Basic Idea

In this chapter, we assume the asynchronous distributed model as described in Chapter 2.2.2. Each node knows its own neighbors in T and in G, and for each neighbor the length of the corresponding edge. At the end of the distributed computation, for every edge $e = (x, p(x))$ of T, the selected best swap edge f (if any exists) must be known to the nodes x and $p(x)$ (but not necessarily to any other nodes). Each message sent from some node to one of its neighbors arrives eventually (there is no message loss). Furthermore, nodes do not need to know the total number of nodes in the system (although it is easy to

count the nodes in T using a convergecast).

5.4.1 The Basic Idea

Our goal is to compute, for each edge of T, a best swap edge. A swap edge for a given failing edge $e = (x, p(x))$ must connect the subtree of T rooted at x to the part of the tree containing $p(x)$. Thus, a swap edge must be incident to some node inside T_x. If each node in T_x considers its own local swap edges for e, then in total all swap edges for e are considered. Therefore, each node inside T_x finds a best *local* swap edge, and then participates in a *minimum finding process* that computes a (globally) best swap edge for e. The computation of the best local swap edges is composed of three main phases: In a first (preprocessing) phase, a root of the MDST is chosen, and various pieces of information (explained later) are computed for each node. Then, in a second (top-down) phase each node computes and forwards some "enabling information" (explained later) for each node in its own subtree. This information is collected and merged in a third (bottom-up) phase, during which each node obtains its best local swap edge for each (potentially failing) edge on its path to the root. The efficiency of our algorithm will be due to our careful choice of the various pieces of information that we collect and use in these phases.

To give an overview, we now briefly sketch how each node computes a best local swap edge. First observe that after replacing edge e by f, the resulting diameter is longer than the previous diameter only if there is a path through f which is longer than the previous diameter, in which case the path through f is the new diameter. In this case, the length of the diameter equals the length of a longest path through f in the new tree. For a local swap edge $f = (u, v)$ connecting nodes $v \in V(T_x)$ and $u \in V \backslash V(T_x)$, such a path consists of

(i) a longest path inside $T \backslash T_x$ starting in u,

(ii) edge f, and

(iii) a longest path inside T_x starting in v.

Part (i) is computed in a preprocessing phase, as described in Section 5.6. Part (ii) is by assumption known to v, because f is incident

to v. Part (iii) is inductively computed by a process starting from the root x of T_x, and stopping in the leaves, as follows. A path starting in v and staying inside T_x either descends to a child of v (if any), or goes up to $p(v)$ (if $p(v)$ is still in T_x) and continues within $T_x \backslash T_v$. For the special case where $v = x$, node x needs to consider only the heights of the subtrees rooted at its children, where the height of a subtree denotes the maximum length of any path from the subtree root to a leaf in the subtree. All other nodes v in T_x additionally need to know the length of a longest path starting at $p(v)$ and staying inside $T_x \backslash T_v$. This additional *enabling information* will be computed by $p(v)$ and then be sent to v.

Once the best local swap edges are known, a best (global) swap edge is identified by a single minimum finding process that starts at the leaves of T_x and ends in node x. To compute all best swap edges of T, this procedure is executed separately for each edge of T. This approach will turn out to work with the desired efficiency:

Theorem 5.1. *All best swap edges of a MDST can be computed in an asynchronous distributed setting with $O(n^* + m)$ messages of constant size, and in $O(\|\mathcal{D}\|)$ time.*

Note that a naive algorithm, in which each node sends its topology information to the root, where the solution is computed and then broadcast to all nodes, would be a lot less efficient than the above bounds in several respects: Since the root must receive $\Theta(m)$ edge lengths in this naive algorithm, possibly through only a constant number of edges (e.g. if the MDST has constant degree), its running time would be $\Omega(m)$, possibly much higher than $O(\|D\|)$ (which is $O(n)$). Furthermore, sending information about the m edges to the root would require $\Omega(mn)$ constant size messages in some cases, as the information might have to travel through $\Omega(n)$ intermediate nodes. Reducing the number of messages to $O(n)$ would be possible by increasing the message size from constant to $O(m)$, which however does not seem practical.

We will prove the above theorem in the next sections, by proving that the preprocessing phase can be realized with $O(m)$ messages, and after that the computation of all best swap edges requires at most $O(n^*)$ additional messages.

Our algorithm requires that each node knows which of its neighbors are children and which neighbor is its parent in T. Although

this information is not known a priori, it can be easily computed in a preprocessing phase, during which a diameter and a root of T are selected.

5.5 How to Pick a Best Swap Edge

In our distributed algorithm, we compute for each (potentially) failing edge the resulting new diameter for each possible swap edge candidate. This approach can be made efficient by exploiting the structure of changes of the diameter path, as described in the following.

5.5.1 The Structure of Changes of the Diameter Path

For a given failing edge e, let $\mathcal{P}_{e/f}$ be a longest path in $T_{e/f}$ that goes through swap edge f for e. Then, recall Lemma 4.2 from Chapter 4: *The length of the diameter of $T_{e/f}$ is*

$$|\mathcal{D}(T_{e/f})| = \max\{|\mathcal{D}(T)|, |\mathcal{P}_{e/f}|\}.$$

That is, for computing the resulting diameter length for a given swap edge $f = (u, v)$ for e, we only need to compute the length of a longest path in $T_{e/f}$ that goes through f. For node v in the subtree T_x of T rooted in x, and u outside this subtree, such a path $\mathcal{P}_{e/f}$ consists of three parts. To describe these parts, let $\mathcal{L}(H, r)$ denote a longest path starting in node r and staying inside the graph H. The first part is a longest path $\mathcal{L}(T \backslash T_x, u)$ in $T \backslash T_x$ that starts in u. The second part is the edge f itself. The third part is a longest path $\mathcal{L}(T_x, v)$ starting in v and staying inside T_x. This determines the length of a longest path through f as $|\mathcal{P}_{e/f}| = |\mathcal{L}(T_x, v)| + l(f) + |\mathcal{L}(T \backslash T_x, u)|$.

5.5.2 Distributed Computation of $|\mathcal{L}(T_x, v)|$

For a given failing edge $e = (x, p(x))$, each node v in T_x needs its $|\mathcal{L}(T_x, v)|$ value to check for the new diameter when using a swap edge. This is achieved by a distributed computation, starting in x. As x knows the heights of the subtrees of all its children (from the preprocessing), it can locally compute the height of its own subtree

T_x as $|\mathcal{L}(T_x, x)| = \max_{q \in C(x)}\{l(x,q) + \text{height}(T_q)\}$, where $C(x)$ is the set of children of x. For a node v in the subtree rooted at x, a longest simple path either goes from v to its parent and hence has length $|\mathcal{L}(T_x \backslash T_v \cup \{(v, p(v))\}, v)|$, or goes into the subtree of one of its children and hence has length $|\mathcal{L}(T_v, v)|$ (see Figure 5.3). The latter term has just been described, and the former can be computed by induction by the parent r of v and can be sent to v. This inductive step is identical to the step just described, except that v itself is no candidate subtree for a path starting at r in the induction. In total, each node r computes, for each of its children $q \in C(r)$, the value of

$$|\mathcal{L}(T_x \backslash T_q \cup \{(q,r)\}, q)| = l(q,r) \; + \\ \max\Big\{ \quad |\mathcal{L}(T_x \backslash T_r \cup \{(r, p(r))\}, r)|, \\ \max_{s \in C(r), s \neq q} \{l(r,s) + \text{height}(T_s)\} \quad \Big\},$$

and sends it to q, where we assume that $|\mathcal{L}(T_x \backslash T_r \cup \{(r, p(r))\}, r)|$ was previously sent to r by $p(r)$.

A bird's eye view of the process shows that each node v first computes $|\mathcal{L}(T_x, v)|$, and then computes and sends $|\mathcal{L}(T_x \backslash T_q \cup \{(q,v)\}, q)|$ to each of its children $q \in C(v)$. Computation of the $|\mathcal{L}(T_x, v)|$ values finishes in T_x's leaves. Note that a second value will be added to the enabling information if $(x, p(x)) \in \mathcal{D}$, for reasons explained in the next section.

5.5.3 Distributed Computation of $|\mathcal{L}(T \backslash T_x, u)|$

In the following, we explain how v can compute $|\mathcal{L}(T \backslash T_x, u)|$ for a given swap edge $f = (u, v)$. In case the failing edge $e = (x, p(x)) \notin \mathcal{D}$, we show below that the information obtained in the preprocessing phase is sufficient.

For the sake of clarity, we analyze two cases separately, starting with the simpler case.

Case 1: The removed edge e is not on the diameter. For this case, we know from [65] that at least one of the longest paths in $T \backslash T_x$ starting from u contains d_c. If $u \in V_L$, we get a longest path from u through d_c by continuing on the diameter up to d_k, and hence we have $|\mathcal{L}(T \backslash T_x, u)| = d(u, d_c) + |\mathcal{D}_R|$. If u is in V_C or V_R, some longest path from u through d_c continues on the diameter up to d_1, yielding $|\mathcal{L}(T \backslash T_x, u)| = d(u, d_c) + |\mathcal{D}_L|$. Remarkably, in

this case $|\mathcal{L}(T\backslash T_x, u)|$ does not depend on the concrete failing edge $e = (x, p(x))$, apart from the fact that (u, v) must be a swap edge for e.

Case 2: The removed edge e is on the diameter. We analyze the case $e \in \mathcal{D}_L$, and omit the symmetric case $e \in \mathcal{D}_R$. If $u \in V_L$ or $u \in V_C$, we know from [65] that again, one of the longest paths in $T\backslash T_x$ starting at u contains d_c. Thus, for $u \in V_L$ we are in the same situation as for the failing edge not on the diameter, leading to $|\mathcal{L}(T\backslash T_x, u)| = d(u, d_c) + |\mathcal{D}_R|$. For $u \in V_C$, after d_c a longest path may continue either on $\mathcal{D}_R$, or continue to nodes in V_L[1]. In the latter case, the path now cannot continue on $\mathcal{D}_L$ until it reaches d_1, because edge e lies on $\mathcal{D}_L$. Instead, we are interested in the length of a longest path that starts at d_c, proceeds into V_L, but does not go below the parent $p(x)$ of x on $\mathcal{D}_L$; let us call this length $\lambda(p(x))$. As announced before, we include the $\lambda(p(x))$ value as a second value into the *enabling information* received by $p(x)$; then, we get $|\mathcal{L}(T\backslash T_x, u)| = d(u, d_c) + \max\{|\mathcal{D}_R|, \lambda(p(x))\}$. The remaining case is $u \in V_R$. For this case (see Figure 5.4), we know (from [65]) that at least one of the longest paths in $T\backslash T_x$ starting at u passes through the node u' closest to u on $\mathcal{D}(T)$. After u', this path may either

(i) continue on $\mathcal{D}_R$ up to d_k, or

(ii) continue through d_c going inside V_C, or

(iii) continue through d_c going inside V_L (without crossing $e = (x, p(x))$), or

(iv) continue towards d_c only up to some node d_i on $\mathcal{D}_R$, going further on non-diameter edges inside V_R.

Option (i) yields a length of $d(u, d_k) = d(u, u') + d(u', d_k) = d(u, u') + (|\mathcal{D}_R| - d(d_c, u'))$. Option (ii) requires the term γ, denoting the length of a longest path starting in d_c and consisting only of nodes in V_C. The length of the path using this option is then $d(u, d_c) + \gamma$. Option (iii) yields the length $d(u, d_c) + \lambda(p(x))$.

It remains to show how the length of a longest path of the last type (Option (iv)) can be found efficiently. We propose to combine

[1]The option of going back into V_C can be ignored because it cannot yield a path longer than $\mathcal{D}_R$.

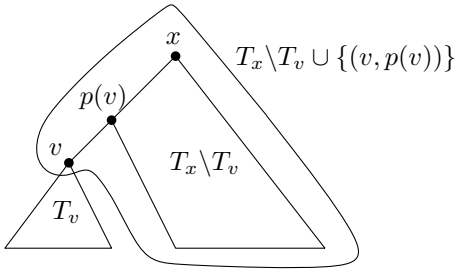

Figure 5.3: *Illustration of the tree* $T_x \backslash T_v \cup \{(v, p(v))\}$.

three lengths, in addition to the length of the path from u to u'. The first is the length of a longest path inside V_R that starts at d_k; let us call this length μ_R. In general, this path goes up the diameter path $\mathcal{D}_R$ for a while, and then turns down into a subtree of V_R, away from the diameter, at a diameter node that we call ρ_R (see Figure 5.4). Given μ_R, the distance from u' to ρ_R, and the distance from ρ_R to d_k, the desired path length of an upwards turning path inside V_R is $d(u, u') + d(u', \rho_R) + \mu_R - d(d_k, \rho_R)$. Note that while it may seem that ρ_R needs to lie above u' on $\mathcal{D}_R$, this is not really needed in our computation, because the term above will not be larger than Option (i) if ρ_R happens to be at u' or below. Furthermore, in this case Option (iv) cannot be better than Option (i) and thus need not be considered. In total, we get

$$\begin{aligned}|\mathcal{L}(T\backslash T_x, u)| = \max\Big\{ & d(u, d_k), \quad d(u, d_c) + \gamma, \\ & d(u, d_c) + \lambda(p(x)), \\ & d(u, u') + d\big(u', \rho_R\big) + \mu_R - d\big(d_k, \rho_R\big)\Big\}.\end{aligned}$$

All of these path length computations can be carried out locally with no message exchanges, if the constituents of these sums are available locally at a node. We will show in the next section how to achieve this in an efficient preprocessing phase.

5.5.4 The BESTDIAMSWAP Algorithm

For a given edge $e = (x, p(x))$ that may fail, each node v in the subtree T_x rooted at x executes the following steps:

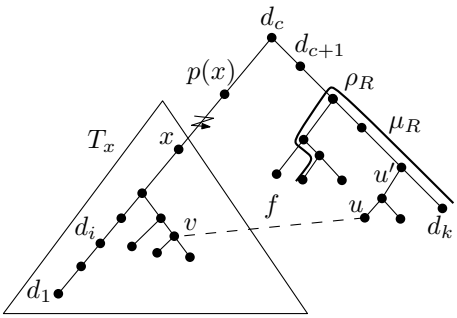

Figure 5.4: *Computing $|\mathcal{L}(T\backslash T_x, u)|$ if $e \in \mathcal{D}_L$, $v \in V_L$ and $u \in V_R$.*

(i) Wait for the enabling information from the parent (unless $x = v$), and then compute $|\mathcal{L}(T_x, v)|$. Compute the enabling information for all children and send it.

(ii) For each local swap edge $f = (u, v)$, compute $|\mathcal{L}(T\backslash T_x, u)|$ as described in Section 5.5.3.

(iii) For each local swap edge $f = (u, v)$, locally compute

$$|\mathcal{D}(T_{e/f})| = \max\Big\{|\mathcal{D}(T)|, \\ |\mathcal{L}(T_x, v)| + l(f) + |\mathcal{L}(T\backslash T_x, u)|\Big\}.$$

Among these, choose a best swap edge f^*_{local} and store the resulting new diameter as $|\mathcal{D}(T_{e/f^*_{local}})|$. If no local swap edge exists, then create a "dummy" candidate whose diameter length is ∞.

(iv) From each child $q \in C(v)$, receive the node label of a best swap edge candidate f^*_q and its resulting diameter $|\mathcal{D}(T_{e/f^*_q})|$. Pick a best swap edge candidate f^*_b among these, i.e., choose $b := \arg\min_{q\in C(v)} |\mathcal{D}(T_{e/f^*_q})|$. Compare the resulting diameter of f^*_b and f^*_{local}, and define f_{best} as the edge achieving the smaller diameter (or any of them if their length is equal), and its diameter as $|\mathcal{D}(T_{e/f_{best}})|$.

(v) Send the information f_{best}, $|\mathcal{D}(T_{e/f_{best}})|$ to the parent.

The above algorithm computes the best swap edge for one (potentially) failing edge e, based on the information available after the preprocessing phase. In order to compute all best swap edges of T, we execute this algorithm for each edge of T independently. A pseudocode description of algorithm BESTDIAMSWAP is given in Algorithm 1. BESTDIAMSWAP in turn uses algorithm LONGEST, which is described in Algorithm 2.

Analysis of the Algorithm

We now show that the proposed algorithm indeed meets our efficiency requirements:

Theorem 5.2. *After preprocessing, executing the* BESTDIAMSWAP *algorithm independently for each and every edge $e \in E_T$ costs at most $O(n^*)$ messages of size $O(1)$ each, and $O(\|\mathcal{D}\|)$ time.*

Proof. Correctness follows from the preceding discussion. Preprocessing ensures that all precomputed values defined for the other end u of a candidate swap edge are available locally at v (these values are required to compute, e.g., $|\mathcal{L}(T \backslash T_x, u)|$). As to the message complexity, consider the execution of the BESTDIAMSWAP algorithm for one particular edge $e = (x, p(x))$. Starting in node $x \in V \backslash \{d_c\}$, each node in T_x sends a message containing the "enabling information" (i.e., $\mathcal{L}(T_x \backslash T_q, q)$ and possibly $\lambda(p(x))$) containing $O(1)$ items to each of its children. Furthermore, each node in T_x (including finally x) sends another message with size $O(1)$ up to its parent in the minimum finding process. Hence, two messages of size $O(1)$ are sent across each edge of T_x, and one message is sent across e. Thus, the computation of a best swap for e requires $2 \cdot |E_{T_x}| + 1 = 2 \cdot |V(T_x)| - 1$ messages. The number of messages exchanged for computing a best swap edge for *each and every* edge $(x, p(x))$ where $x \in V \backslash \{d_c\}$ is $\sum_x (2 \cdot |V(T_x)| - 1) = 2n^* - (n-1)$.

As to the time complexity, note that the best swap computation of a *single* edge according to the BESTDIAMSWAP algorithm requires at most $O(\|\mathcal{D}\|)$ time. Now note that this algorithm can be executed independently (and thus concurrently) for each potential failing edge: In this fashion, each node x in T sends exactly one message to each node in T_x during the top-down phase. Symmetrically, in the bottom-up phase, each node z in T sends exactly one message to each node

Algorithm 1: BESTDIAMSWAP(x, v).

Describes how node v computes a best swap for $e = (x, p(x))$. BESTDIAMSWAP(x, v) is executed for each tree edge $e \in E_T$ separately and concurrently, using a farthest-first contention resolution policy.
if $v = x$ **then**
    $m := 0$
else
    $\{\ z \neq x\ \}$
    Wait until $m = |\mathcal{L}(T_x \backslash T_v \cup \{(v, p(v))\}, v)|$ is received from parent.
    if $x \in \mathcal{D}$ **then** wait for information $\lambda(p(x))$ from parent.
end
$|\mathcal{L}(T_x, v)| := \max\left\{m, \max_{q \in C(v)}\{l(v, q) + \text{height}(T_q)\}\right\}$
for *each local swap edge* $f = (v, u)$ **do**
    $|\mathcal{P}_{e/f}| := |\mathcal{L}(T_x, v)| + l(f) +$ LONGEST(e, f)
    $|\mathcal{D}(T_{e/f})| := \max\{|P_f|, |\mathcal{D}(T)|\}$
end
for *each child* $q \in C(v)$ **do**
    compute the enabling information:
    $|\mathcal{L}(T_x \backslash T_q \cup \{(q, v)\}, q)| :=$
    $l(v, q) + \max\left\{m, \max_{s \in C(v), s \neq q}\{l(v, s) + \text{height}(T_s)\}\right\}$
    if $x \in \mathcal{D}$ **then** append $\lambda(p(x))$ to the enabling information and send it to q.
end
wait until all children have sent back their best swap candidate
$currentbest :=$ a best among these and the local swap candidates
if *there is no swap candidate* **then** $\{$ all children have sent a "dummy" candidate, and there are no local swap candidates $\}$
    $currentbest :=$ a "dummy" candidate whose diameter length is ∞.
if $v = x$ **then**
    store $currentbest$ as the best swap edge for e
    inform $p(x)$ about the best swap edge $currentbest$.
else $\{\ v \neq x\ \}$
    send that swap edge to $p(v)$

Algorithm 2: LONGEST$(e = (x, p(x)), f = (u, v))$.

Input: an edge $e = (x, p(x))$ whose best swap edge shall be computed, and a local swap edge $f = (u, v)$.
Output: the length $|\mathcal{L}(T \backslash T_x, u)|$ of a longest path in $T \backslash T_x$ that starts in u.

if *e is on the diameter (i.e., $x \in \mathcal{D}$)* **then**
  if $x \in V_L$ **then** { $e \in V_L$ }
    if $u \in V_L$ **then**
      { one longest path containing f goes through d_c: }
      return $d(u, d_c) + |\mathcal{D}_R|$
    else if $u \in V_C$ **then**
      { one longest path containing f goes through d_c: }
      return $d(u, d_c) + \max\{|\mathcal{D}_R|, \lambda(p(x))\}$
    else if $u \in V_R$ **then**
      { Let u' be the nearest ancestor of u on the diameter. One longest path from u must go through u'. }
      $d(d_k, \rho_R) := |\mathcal{D}_R| - d(\rho_R, d_c)$
      $d(u', \rho_R) := |d(u', d_c) - d(\rho_R, d_c)|$
      $d(u', d_k) := |\mathcal{D}_R| - d(u', d_c)$
      $d(u, u') := d(u, d_c) - d(u', d_c)$
      $from\text{-}u' := \mu_R - d(d_k, \rho_R) + d(u', \rho_R)$
      return $d(u, u') + \max\{d(u', d_k), d(u', d_c) + \lambda(p(x)), from\text{-}u'\}$
    end
  else {$x \in V_R$: symmetric to $x \in V_L$ }
    { code omitted because it is completely symmetric to the above case }
else { e not on the diameter }
  if $u \in V_L$ **then**
    return $d(u, d_c) + |\mathcal{D}_R|$
  else { $u \in V_C$ or V_L, and $|\mathcal{D}_L| \geq |\mathcal{D}_R|$ }
    return $d(u, d_c) + |\mathcal{D}_L|$
  end

on its path to the root. The crucial point here is to avoid that some of these messages block others for some time (as only one message can traverse a link at a time). Indeed, one can ensure that each message reaches its destination in $O(\|\mathcal{D}\|)$ time as follows. A node z receiving a message with destination at distance d from z forwards it only after all messages of the protocol with a destination of distance more than d from z have been received and forwarded. By induction over the distance of a message from its destination, this "farthest-first" contention resolution policy (see also [51]) allows each message to traverse one link towards its destination after at most one time unit of waiting. Thus, the $O(\|\mathcal{D}\|)$ time complexity also holds for the entire algorithm. □

Instead of sending many small messages individually, we can choose to sequence the process of message sending so that messages for different failing edges are bundled before sending (see also [28, 30] for applications of this idea). This leads to an alternative with fewer but longer messages:

Corollary 5.3. *After preprocessing, the* distributed All-Best-Swaps *problem can be solved using $O(n)$ messages of size $O(n)$ each, and in $O(\|\mathcal{D}\|)$ time.*

5.6 The Preprocessing Phase

The preprocessing phase serves the purpose of making the needed terms in the sums described in the previous section available at the nodes of the tree. Details of this phase can be found in the pseudocode given as algorithm Preprocessing 1 in Algorithm 3, and algorithm Preprocessing 2 in Algorithms 5, 6 and 7.

5.6.1 Algorithms

In the preprocessing phase, a diameter $\mathcal{D}$ of T is chosen, and its two ends d_1 and d_k as well as its center d_c are identified. This can be done essentially by a convergecast, followed by a broadcast to distribute the result (see e.g. [82]); the details are standard and therefore omitted. After preprocessing exchanges $O(n)$ messages, each node knows the information that is requested in (A) and (C) below. It is crucial that

Algorithm 3: Preprocessing 1 for node z: Finding a diameter.

{ Let $\tilde{T}_s$ be the connected component of $T - \{(s, z)\}$ containing s. Each message $M_s = (deepestnode, height, source, diamLen, d_1, d_k)$ from neighbor s contains the identifier of a deepest node in $\tilde{T}_s$, $\mathrm{height}(\tilde{T}_s) + l(s, z)$, the neighbor s that sent the message, the length of a diameter of $\tilde{T}_s$, and its two endpoints d_1 and d_k. }
$Diams := \{\}$; $Heights := \{\}$
if *z is a leaf* **then**
    $last :=$ the only node in $\bar{N}_T(z)$
    Send $(z, l(z, last), z, 0, z, z)$ to $last$.
else { z not a leaf }
    Wait until at least $|\bar{N}_T(z)| - 1$ neighbors' messages have been received.
    $last :=$ node in $\bar{N}_T(z)$ whose message has not yet been received, or was received last.
    for *each message M_s from neighbor $s \in \bar{N}_T(z) \backslash \{last\}$* **do**
        $(deepestnode, height, s, diamLen, d_1, d_k) := M_s$
        Insert $(deepestnode, height, s)$ into $Heights$.
        Insert $(diamLen, d_1, d_k)$ into $Diams$.
    end
    $(diamLen^*, d_1^*, d_k^*) :=$ Update($Heights$, $Diams$)
    $(a, height_a, s_a) :=$ a tuple in $Heights$ with highest $height_a$.
    Send $(a, height_a + l(z, last), s_a, diamLen^*, d_1^*, d_k^*)$ to $last$.
end
Wait until a message (from $last$) is received.
if *the message is M_{last} from $last$ and $id(z) < id(last)$* **then**
    { Locally compute the *global* diameter of T: }
    Insert $(deepestnode, height)$ from M_{last} into $Heights$
    Insert $(diamLen, d_1, d_k)$ from M_{last} into $Diams$
    $(diamLen^*, d_1^*, d_k^*) :=$ Update($Heights$, $Diams$)
    Send "$\mathcal{D} = (d_1^*, d_k^*, diamLen^*)$" to all neighbors.
else if *the message is "$\mathcal{D} = (d_1, d_k, |\mathcal{D}|)$"* **then**
    Forward "$\mathcal{D} = (d_1, d_k, |\mathcal{D}|)$" to all other neighbors.
end
Start Preprocessing 2.

Procedure Update ($Heights$, $Diams$)

$(a, height_a, s_a) :=$ a tuple in $Heights$ with highest $height_a$.
$(b, height_b, s_b) :=$ a tuple in $Heights \backslash \{(a, height_a, s_a)\}$ with highest $height_b$.
Insert $(height_a + height_b, a, b)$ into $Diams$.
return $(diamLen^*, d_1^*, d_k^*) :=$ a triple in $Diams$ with highest $diamLen^*$.

Algorithm 5: Preprocessing 2 for node z: Computing information about the diameter.

{ Determine if z is itself on the diameter $\mathcal{D}$:
After Preprocessing 1, every node knows $\mathcal{D}$, d_1 and d_k. If no message containing d_1 (d_k) as the deepest node was received, then d_1 (d_k) must be in $\tilde{T}_{last}$: }
Find $(d_1, height_1, s_1)$ tuple in $Heights$, set $s_1 := last$ if none found.
Find $(d_k, height_k, s_k)$ tuple in $Heights$, set $s_k := last$ if none found.
if $(s_1 = s_k)$ **then** { z is not on $\mathcal{D}$ }
    Upon receiving M_* from neighbor q,
    set $(\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c),$
        $|\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c), u, d(u, d_c)) := M_*$, where $V_* \in \{V_L, V_C, V_R\}$; set $parent := q$; and send $M_* := (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c),$
        $|\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c) + d(z, r), u, d(u, d_c))$ to every neighbor $r \in \bar{N}_T(z) \backslash \{parent\}$.
    { Send the message M' across each non-tree edge: }
    Send $M' := (d(z, d_c), V_*, u', d(u', d_c))$ to all neighbors $z' \in \bar{N}_G(z) \backslash \bar{N}_T(z)$.
    return
end
{ z is on $\mathcal{D}$. In this case, z knows at least one of the distances $d(z, d_1) = \text{height}(\tilde{T}_{d_1})$ and $d(z, d_k) = \text{height}(\tilde{T}_{d_k})$. }
if $s_1 = last$ **then** $height_1 := |\mathcal{D}| - height_k$
if $s_k = last$ **then** $height_k := |\mathcal{D}| - height_1$
{ $height_1 = d(z, d_1)$ and $height_k = d(z, d_k)$. }
Continued on the next page.

Algorithm 6: Preprocessing 2, continued (1)

if $\big(height_1 \geq height_k\big) \wedge \big(height_1 - l(z, s_1) < height_k + l(z, s_1)\big)$ **then** { z is d_c }
  $\lambda(d_c) := 0$; $|\mathcal{D}_L| := height_1$; $|\mathcal{D}_R| := height_k$
  Send $\big(\lambda(d_c), l(s_1, z)\big)$ to s_1 and $\big(\lambda(d_c), l(s_k, z)\big)$ to s_k
  Receive $\big(\mu, \rho, d(\rho, d_c), d_1\big)$ from s_1 and $\big(\mu', \rho', d(\rho, d_c)', d_k'\big)$ from s_k, and set
  $(\mu_L, \rho_L, d(\rho_L, d_c) := \big(\mu, \rho, d(\rho, d_c), d_1\big)$,
  $(\mu_R, \rho_R, d(\rho_R, d_c) := \big(\mu, \rho, d(\rho, d_c), d_k\big)$.
  Forward $M_* := (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c),$
    $|\mathcal{D}_L|, |\mathcal{D}_R|, V_L, 0)$ to d_{c-1}.
  Forward $M_* := (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c),$
    $|\mathcal{D}_L|, |\mathcal{D}_R|, V_R, 0)$ to d_{c+1}.
  Forward $M_* := (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c),$
    $|\mathcal{D}_L|, |\mathcal{D}_R|, V_C, 0)$ to all neighbors in $N_T(d_c) \backslash \{d_{c-1}, d_{c+1}\}$.
Continued on the next page.

during preprocessing, each node obtains enough information to later carry out all computational steps to determine path components (i), (ii) and (iii). More precisely, each node gets the following global information (the same for all nodes):

(A) The endpoints d_1 and d_k of the diameter, the length $|\mathcal{D}|$ of the diameter, and the lengths $|\mathcal{D}_L|$ and $|\mathcal{D}_R|$.

(B) The length μ_R of a longest path starting in d_k that is fully inside $T_{d_{c+1}}$, together with the node ρ_R on $\mathcal{D}$ where this path leaves the diameter, and the distance $d(\rho_R, d_c)$. Figure 5.5 illustrates such a longest path μ_R. Symmetrically, the values μ_L, ρ_L and $d(\rho_L, d_c)$ are also required.

In addition, each node z obtains the following information that is specific for z:

(C) For each child $q \in C(z)$ of its children, the height of q's subtree T_q.

(D) Whether z is on the diameter $\mathcal{D}$ or not.

Algorithm 7: Preprocessing 2, continued (2)

else $\{\ z \neq d_c\ \}$
  Wait for the message containing λ and $d(z, d_c)$.
  $(i, height_i, s_i) :=$ a tuple in $Heights \backslash \{(s_1, \cdot, \cdot), (s_k, \cdot, \cdot)\}$ with largest $height_i$.
  Compute $\lambda(z) := \max\{\lambda, d(z, d_c) + height_i\}$.
  Send $\big(\lambda(z), d(z, d_c) + l(z, s_1)\big)$ to s_1, and $\big(\lambda(z), d(z, d_c) + l(z, s_k)\big)$ to s_k.
  if *z is d_1* **then**
    $\mu(d_1) := 0$
    Send $\big(\mu(d_1), d_1, d(d_1, d_c), d_1\big)$ to s_k
  else if *z is d_k* **then**
    $\mu(d_k) := 0$
    Send $\big(\mu(d_k), d_k, d(d_k, d_c), d_k\big)$ to s_1.
  else $\{\ z$ is on $\mathcal{D}$, but $z \notin \{d_1, d_c, d_k\}\ \}$
    Upon receiving $\big(\mu, \rho, d(\rho, d_c), d_*\big)$, where $d_* \in \{d_1, d_k\}$, compute $\mu(z) := \max\{\mu, d(d_*, z) + height_i\}$ and set $\rho(z) := z$ and $dist := d(z, d_c)$ if $\mu(z) > \mu$, and $\rho(z) := \rho$ and $dist := d(\rho, d_c)$ otherwise.
    Forward $\big(\mu(z), \rho(z), dist\big)$ along the diameter.
    Upon receiving M_* from one neighbor on the diameter, set $(\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c),$
      $|\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c)) := M_*$,
      where $V_* \in \{V_L, V_C, V_R\}$, send
    $M_* := (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c),$
      $|\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c) + d(z, q))$ to the other neighbor q on the diameter, and send $M_* := (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c),$ $|\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c) + d(z, r), u, d(u, d_c))$ to all other neighbors $r \in \bar{N}_T(z) \backslash \{q\}$.
Send $M' := (d(z, d_c), V_*, u', d(u', d_c))$ to all neighbors $z' \in \bar{N}_G(z) \backslash \bar{N}_T(z)$.

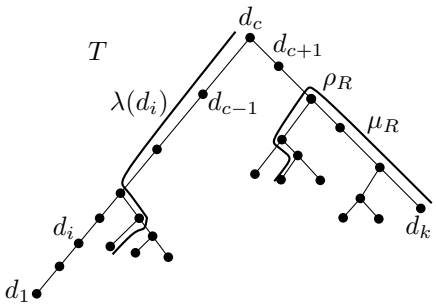

Figure 5.5: *Definition of $\lambda(d_i)$, μ_R and ρ_R.*

(E) The distance $d(z, d_c)$ from z to d_c.

(F) The identification of the parent $p(z)$ of z in T.

(G) To which of V_L, V_C and V_R does z belong.

(H) If $z \notin \mathcal{D}$, the closest ancestor u of z on the diameter; the distance $d(u, d_c)$ from u to d_c.

(I) If z is on the left (right) diameter $\mathcal{D}_L$ ($\mathcal{D}_R$), with $z = d_i$, the length $\lambda(d_i)$ of a longest path in T starting at d_c and neither containing the node d_{c+1} (d_{c-1}) nor the node d_{i-1} (d_{i+1}), nor any node from V_C (see Figure 5.5).

(J) For each of the neighbors z' of z in G, which of V_L, V_C and V_R contains z'; the distance $d(z', d_c)$ from z' to d_c; the nearest ancestor u' of z' on $\mathcal{D}$, the distance $d(u', d_c)$.

Computing the Additional Information

Recall that the first preprocessing part ends with a broadcast that informs all nodes about the information described in (A) and (C). The second part of the preprocessing phase follows.

A node z receiving the message about $\mathcal{D}$ can infer from the previous convergecast whether it belongs to $\mathcal{D}$ itself by just checking whether the paths from z to d_1 and d_k go through the same neighbor of z.

Information (E) is obtained by having the center node send a "distance from d_c" message to both neighbors d_{c+1} and d_{c-1} on $\mathcal{D}$, which is forwarded and updated on the diameter[2]. This information is used by the diameter nodes for computing $\lambda(d_i)$, required in (I). The center initiates the inductive computation of $\lambda(d_i)$:

- $\lambda(d_c) = 0$.
- For each d_j, $1 \leq j < c$,

  $$\lambda(d_j) = \max\{\lambda(d_{j+1}), d(d_c, d_j) + h_2(d_j)\},$$

  h_2 being the height of a highest subtree of d_j apart from the diameter subtree.
- For each d_j, $c < j \leq k$,

  $$\lambda(d_j) = \max\{\lambda(d_{j-1}), d(d_c, d_j) + h_2(d_j)\}.$$

In order to compute μ_L and μ_R as required in (B), we define $\mu(d_i)$ for each node d_i on $\mathcal{D}_L$ as the length of a longest path starting in d_1 that is fully inside T_{d_i}, together with the node $\rho(d_i)$ on $\mathcal{D}_L$ where such a path leaves the diameter. For d_i on $\mathcal{D}_R$, the definition is symmetric. We then have $\mu_L = \mu(d_{c-1})$ and $\mu_R = \mu(d_{c+1})$. The inductive computation of $\mu(d_i)$ is started by d_1 and d_k, and then propagated along the diameter:

- $\mu(d_1) = \mu(d_k) = 0$;
- for each d_j, $1 < j < c$,

  $$\mu(d_j) = \max\{\mu(d_{j-1}), d(d_1, d_j) + h_2(d_j)\};$$

- for each d_j, $c < j < k$,

  $$\mu(d_j) = \max\{\mu(d_{j+1}), d(d_k, d_j) + h_2(d_j)\}.$$

[2]The message is updated as follows: when forwarded through an edge on the diameter, the length of this edge is added to the forwarded distance. This ensures that each node which receives the message obtains its own distance from d_c. Details are described in Algorithm 6.

Along with $\mu(d_j)$, $\rho(d_j)$ and $d(\rho(d_j), d_c)$ can be computed as well. The computation stops in d_c, which receives the messages

$$(\mu(d_{c-1}), \rho(d_{c-1}), d(\rho(d_{c-1}), d_c)) = (\mu_L, \rho_L, d(\rho_L, d_c))$$

and

$$(\mu(d_{c+1}), \rho(d_{c+1}), d(\rho(d_{c+1}), d_c)) = (\mu_R, \rho_R, d(\rho_R, d_c)).$$

Altogether, this second preprocessing part operates along the diameter and takes $O(\|\mathcal{D}(T)\|) = O(n)$ messages.

Distributing the Information

When the computation of the two triples $(\mu_L, \rho_L, d(\rho_L, d_c))$ and $(\mu_R, \rho_R, d(\rho_R, d_c))$ completes in d_c, the center packs these values plus the values $|\mathcal{D}_L|$ and $|\mathcal{D}_R|$ into one message M_*. It adds the appropriate one of the labels "V_L","V_R" and "V_C" to M_*, before forwarding M_* to d_{c-1}, d_{c+1} and any other neighbor of d_c in T and then flooding the tree. Additionally, M_* contains the "distance from d_c" information which is updated on forwarding, such that all nodes know their distance to the center[3]. When M_* is forwarded from a node $u \in \mathcal{D}$ to a node not on $\mathcal{D}$, it is extended by the "distance from u" information, which is also updated on forwarding. In addition, $d(u, d_c)$ is appended to M_*. Finally, if node z receives M_* from node v, then z learns that v is its parent.

At the end of this second part of the preprocessing phase, each node z' sends a message M' to each of its neighbors z in $G \backslash T$. Note that this is the only point in our solution where messages need to be sent over edges in $G \backslash T$. M' contains $d(z', d_c)$ and exactly one of $\{$ "$z' \in V_L$", "$z' \in V_C$" , "$z' \in V_R$" $\}$, whichever applies. Furthermore, let u' be the nearest ancestor of z' on $\mathcal{D}$; the distance $d(u', d_c)$ is also appended to M'.

As a consequence, after each node has received its version of the message M_*, the information stated in (B), (E), (F), (G), (H) is known to each node. Furthermore, each node that has received M' from all its neighbors in G knows the information stated in (J). The distribution of this information requires $O(\|\mathcal{D}(T)\|)$ time and $O(m)$ messages. Let us summarize.

[3]The nodes on $\mathcal{D}$ already have that information at this point, but all other nodes still require it.

Lemma 5.4. *After the end of the two parts of the preprocessing phase, which requires $O(\|\mathcal{D}\|)$ time, all nodes know all information (A)–(J), and $O(m)$ messages have been exchanged.*

Recognizing Swap Edges Using Labels

A node $v \in T_x$ must be able to tell whether an incident edge $f = (v, w)$ is a swap edge for $e = (x, p(x))$ or not. We achieve this by the folklore method of attaching two labels to each node: The first label is the node's number in a preorder traversal, while the second is its number in a reverse preorder traversal. For any two nodes, a simple comparison of both respective labels tells whether one node lies in the subtree of the other node (see, e.g., [28, 30]). In Section 7.2, we describe this labeling in detail, and show how it can even be used for a compact routing scheme with swap capability.

5.7 Message Lower Bound for Best Swaps Computation

In this section, we show that any distributed algorithm requires to send $\Omega(n^* + m)$ messages in the worst case to solve the *distributed All-Best-Swaps* problem in a minimum diameter spanning tree. Thus, the distributed algorithm that was presented in this chapter is asymptotically optimal in terms of the number of messages. The lower bound exploits the fact that information about a best swap edge must travel from one of the best swap's endpoints to one endpoint of the corresponding failing edge. It requires the assumption that the identifiers of edges are atomic and hence incompressible. That is, if k edge identifiers are to be communicated through a link, then $\Omega(k)$ messages must be sent across that link.

We construct a family of graphs G on n nodes, and a subset $\widehat{E}$ of the edges of Gs MDST, such that no two different edges of $\widehat{E}$ have the same best swap. Then, for every failing edge $e \in \widehat{E}$ and its best swap f, we consider the shortest path in terms of hops (in G) that connects either endpoint of the failing edge with either endpoint of its (unique) best swap edge. Since the identifier of the best swap f (which is initially only available at the endpoints of f) must somehow reach the endpoints of e, we know that the sum of the hop-lengths

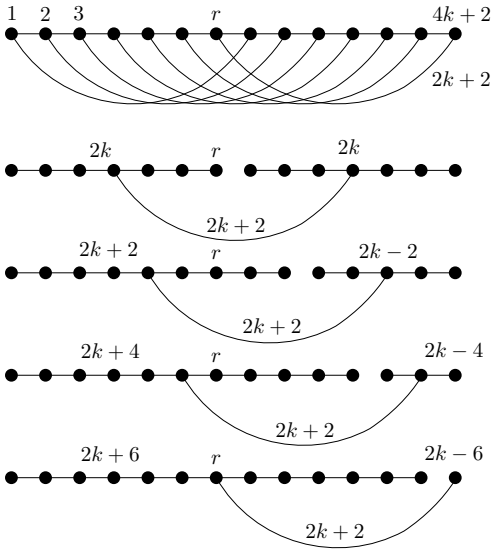

Figure 5.6: *An example of the graph used for the $\Omega(n^* + m)$ lower bound, and some of its best swap trees, for $k = 3$.*

of all these paths is an asymptotic lower bound for the number of messages sent by any algorithm which solves the *distributed All-Best-Swaps* problem. For our lower bound proof, we require the following lemma:

Lemma 5.5. *For two edges e, f of a graph Q, let $||e, f||_Q$ denote the minimum number of hops required to go from either endpoint of e to either endpoint of f in Q. For every n, there exists a graph G with at least n nodes, with root r, whose MDST T is unique, and there is a subset $\widehat{E}$ of T's edges, such that*

- *no non-tree edge f is a best swap for two different edges in $\widehat{E}$, and for every edge $e \in \widehat{E}$, there is exactly one best swap edge $bestswap(e)$, and*
- *it holds that*

$$8 \cdot \sum_{e \in \widehat{E}} ||e, bestswap(e)||_G \geq n_r^*,$$

where n_r^* is the size of the transitive closure of T, if all edges are directed towards r.

Proof. W.l.o.g., let n be such that $n = 4k + 2$ for some $k \in \mathbb{N}$. We define G as follows: $V := \{1, 2, \ldots, n\}$, and $E := \big\{(i, i + 1)|i \in \{1, 2, \ldots, n-1\}\big\} \cup \big\{(j, j+2k+1)|j \in \{1, 2, \ldots, 2k+1\}\big\}$. The edge lengths are $l(i, i + 1) := 1$ for $i \in \{1, 2, \ldots, n - 1\}$ and $l(j, j + 2k + 1) := 2k + 2$ for $j \in \{1, 2, \ldots, 2k + 1\}$. The root r is node $2k + 1$. Figure 5.6 shows an example for $k = 3$.

It is easy to verify that the unique MDST of G is the path consisting of the edges $\{(i, i + 1)|i \in \{1, 2, \ldots, n - 1\}$, which has diameter $n - 1 = 4k + 1$. We will focus on failing edges $\widehat{E} := \big\{(2h + 1, 2h + 2)|h \in \{k, k + 1, \ldots, 2k\}\big\}$. For every such edge $e = (2h + 1, 2h + 2)$, the unique best swap is $(h + 1, h + 2k + 2)$, whose endpoints bisect the two paths in $T - e$, leading to a diameter of $4k + 2$. Note that replacing e by any other swap edge would yield a longer diameter, and hence no two edges in $\widehat{E}$ have a common best swap edge.

In the following, let $H_Q(a, b)$ be the distance in hops from node a to node b in the graph Q. It remains to compute $||e, bestswap(e)||_G$ for every edge $e \in \widehat{E}$. To that end, we first prove the following claim: "For any two nodes a, b for which $H_T(a, b) \leq k$, it holds that $H_T(a, b) = H_G(a, b)$." We show this as follows: Let $x := H_T(a, b)$. Consider any path from a to b in G. When moving from a to b along edges of G, moving along an edge of T changes the number of the current node by 1 modulo $2k + 1$. However, moving along any other edge of G does not change the number of the current node modulo $2k + 1$. Therefore, the hop-length of any path from a to b in G is lower bounded by the hop-length of the shortest path from $\bar{a} := a \mod (2k + 1)$ to $\bar{b} := b \mod (2k + 1)$ in the cycle consisting of the nodes $0, 1, \ldots, 2k$, as shown in Figure 5.7. We need to determine the length of a shortest path from $\bar{a}$ to $\bar{b}$ in this cycle. The only two candidates are the two simple paths from $\bar{a}$ to $\bar{b}$. Since $H_T(a, b) = x$, there exists a path of length x from $\bar{a}$ to $\bar{b}$ in this cycle. It follows that the alternative path (which is disjoint from the former) has length $2k + 1 - x$. By assumption, $x \leq k$, so $2k + 1 - x \geq k + 1 \geq x$. Hence, it follows that $H_G(a, b) \geq x$, which proves our claim.

Returning to $||e, bestswap(e)||_G$, where $e = (2h+1, 2h+2)$ and $bestswap(e) = (h+1, h+2k+2)$, we require a bound on the length

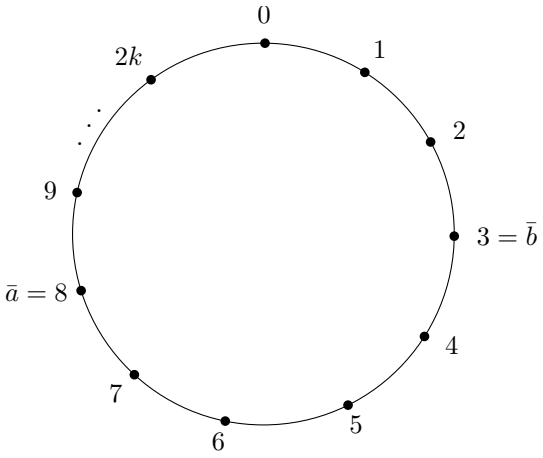

Figure 5.7: *The "modulo $2k + 1$ view" for bounding the length of a shortest path in terms of hops from a to b, showing the example $\bar{a} = 8$, $\bar{b} = 3$.*

of a shortest path connecting any of the four pairs $(2h + 1, h + 1)$, $(2h+1, h+2k+2)$, $(2h+2, h+1)$, $(2h+2, h+2k+2)$, because each of these connects one endpoint of the failing edge with one endpoint of its best swap edge. In the following, we only consider the pair $(2h + 2, h + 2k + 2)$ and bound $||e, bestswap(e)||_G$ by noting that the shortest path connecting any of the other three pairs can be at most two hops shorter.

By the claim proved above, the shortest path from $2h + 2$ to $h + 2k + 2$ in G is the path $\langle 2h + 2, \ldots, h + 2k + 2\rangle$ in T, because $H_T(2h + 2, h + 2k + 2) = 2k - h \leq k$.

We have $\sum_{e \in \widehat{E}} ||e, bestswap(e)||_G$

$$= \sum_{h \in \{k, k+1, \ldots, 2k\}} ||(2h + 1, 2h + 2), (h + 1, h + 2k + 2)||_G$$

$$\geq \sum_{h \in \{k, k+1, \ldots, 2k\}} 2k - h - 2 = \frac{1}{2}k^2 - \frac{3}{2}k - 2.$$

On the other hand,

$$\begin{aligned} n_r^* &= \sum_{v \in V \setminus \{r\}} H_T(v, r) \\ &= \left(\sum_{i=1}^{2k} 2k - i + 1 \right) + \left(\sum_{i=2k+2}^{4k+2} i - (2k+1) \right) \\ &= 4k^2 + 4k + 1. \end{aligned}$$

We have to show that $8 \cdot (\frac{1}{2}k^2 - \frac{3}{2}k - 2) \geq 4k^2 + 4k + 1$, which is equivalent to $(\frac{8}{2} - 4)k^2 + (\frac{3 \cdot 8}{2} - 4)k \geq 1 + 2 \cdot 8$. It is easy to verify that this condition is satisfied for any $k \geq 3$, from which the lemma follows. □

From the above lemma, we immediately have an $\Omega(n^*)$ lower bound for the message complexity of computing all best swaps in a MDST (if we let r be the node which initiates the computation):

Theorem 5.6. *Any distributed algorithm for solving the* distributed All-Best-Swaps *problem requires to send at least* $\Omega(n^*)$ *messages in the worst case.*

Since in the family of graphs used for the proof, the number of edges is $m = O(n)$, and $n^* \in \Omega(n)$, we also obtain a lower bound $\Omega(m)$ for the case $m = \Theta(n)$. Moreover, one can easily modify the above family of graphs by adding more edges (up to $\Theta(k^2) = \Theta(n^2)$) of sufficient length between the nodes $\{1, \ldots, k\}$, without changing the MDST nor the best swap edge for any of the edges in $\widehat{E}$ and thus without decreasing the number of required messages. Since the lower bound for the number of messages is $\Omega(n^2)$ in our construction, the above theorem in fact holds for any asymptotic growth of m.

Corollary 5.7. *Any distributed algorithm for solving the* distributed All-Best-Swaps *problem requires to send at least* $\Omega(n^* + m)$ *messages in the worst case.*

A similar comment applies to the asymptotic growth of n^*. In the proof above, $n^* = \Theta(n^2)$, but the construction can be modified easily to slow down the growth of n^* with respect to n down to any rate between $\Theta(n)$ and $\Theta(n^2)$. To that end, we use several copies of the constructed graph, each consisting of $o(n)$ nodes (the exact

number depends on the desired asymptotic growth of n^*), but place the root r at the node 1, and join these copies together by identifying the root node. Using very similar arguments as in Lemma 5.5, one can again show that the required number of messages is lower bounded by $\Omega(n^*)$.

Chapter 6

Algorithms for Minimum-Stretch Tree Spanners

6.1 Motivation

In the last two chapters, we were concerned with swap edge computation in minimum diameter spanning trees. While such trees minimize the longest distance any message may need to travel to reach its destination, MDSTs can be in some sense unfair: There might be a pair of nodes a, b whose distance in the underlying topology G is rather small, but whose distance in any MDST will be much larger, as shown in Figure 6.1. This calls for an alternative which tries to keep the increase in distance between every pair of nodes as small as possible. Indeed, such a spanning tree type has been proposed in the literature: so-called *tree spanners* [14]. A spanning tree T of G is called a t-spanner, for some $t \in \mathbb{N}$, if for every pair $a, b \in V$, it holds that $d_T(a, b) \leq t \cdot d_G(a, b)$. The smallest t for which T is a t-spanner of G is called its *stretch*. An *optimal tree spanner* for G is a spanning tree whose stretch is minimum. Hence, an optimal tree spanner minimizes the largest multiplicative increase in distance that any pair of nodes experiences when routing inside the tree instead of the entire network. Unfortunately, it is NP-complete to compute the

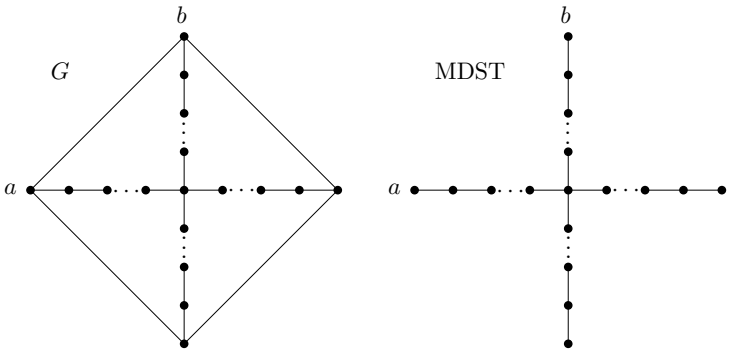

Figure 6.1: *An example where using a MDST leads to much longer paths between some pairs of nodes, compared to the underlying graph G.*

optimal tree spanner of a given graph [14].

In this chapter, we deal with the problem of computing best swaps in optimal tree spanners. Formally, we have

$$\widehat{obj}(T) := \max_{a,b \in V} d_T(a,b)/d_G(a,b)$$

and

$$obj := \widehat{obj},$$

so we use a definition for the best swap edge which is natural according to our framework. Note that we measure the stretch of a swap tree still with respect to the original graph, which includes the failed edge. One may consider to measure the stretch with respect to the graph without the failing edge. However, in weighted graphs such a stretch definition would be very unstable. Indeed, with such a definition it could happen that the swap tree used during an edge failure has a *lower* stretch than the stretch of the initial optimal tree spanner, which we consider rather unnatural[1]. Interestingly enough, by merely going for the best swap, for graphs with unit edge lengths we are guaranteed to find a tree that is not all that bad even in comparison with an entirely new optimal tree spanner: We show that the stretch of

[1]For example, let G be the unweighted cycle on n nodes: The optimal tree spanner of G has stretch $n-1$, but for any failing edge e, the (unique) swap tree, which is exactly the graph $G-e$, has stretch 1.

a swap tree T' obtained by adding a best swap edge is at most twice that of an optimal tree spanner of $G - e$ (measured w.r.t. distances in G).

This particular *All-Best-Swaps* problem appears to be considerably more difficult than the previously studied ones: In all of the latter, one can evaluate a tree obtained by replacing a failing edge with a swap edge in constant time, after some suitable preprocessing. However, for evaluating the stretch of a tree spanner, one needs to consider (at least implicitly) the stretch of each pair of nodes in the graph, which seems impossible to do in constant time, without an expensive precomputation. Furthermore, in the previously studied problems, the quality of a given swap edge could be described somewhat independently of the particular failing edge they are going to replace. Again, this is no longer possible when evaluating swap edges for tree spanners. For the above reasons, none of the techniques used in earlier studies are directly applicable for computing all best swaps of a tree spanner.

6.2 Related Work

The concept of graph spanners was introduced by Peleg and Ullman [77] who used it to construct good synchronizers for communication networks. Later, Peleg and Upfal [78] showed that spanners are useful as subnets for routing as they optimize both the route lengths and the space required for storing routing information. Sparse graph spanners, and in particular tree spanners, are useful in many applications such as designing communication networks, distributed systems, and parallel computers [14]. The problem of finding a tree spanner that minimizes the maximum stretch, called the MMST problem, is NP-hard [14]. There is a $O(\log n)$-approximation algorithm for the MMST in graphs with unit edge lengths [26].

6.3 Summary of Results

We first present and analyze a brute force algorithm for solving the *All-Best-Swaps* problem in Section 6.4. For a graph with n nodes and m edges, this algorithm requires $O(m^2 n)$ time and $O(m)$ space. In

Section 6.5, we describe a more efficient algorithm with time complexity $O(m^2 \log n)$ and space complexity $O(m)$. We also present an $O(n^3)$ time and $O(n^2)$ space solution for graphs with unit edge lengths, also called *unweighted graphs*, in Section 6.6. Finally, in Section 6.7 we compare our approach of using swap edges with the alternative of recomputing an optimal tree spanner for the graph excluding the faulty edge.

The results of this chapter were obtained in collaboration with Shantanu Das and Peter Widmayer [21].

6.4 Computing All Best Swaps

6.4.1 Some Definitions and Properties

We use the following definitions throughout this chapter.

Definition 6.1. *For any pair of nodes $a, b \in V(G)$, the* stretch *of (a, b) in a spanning tree T of G is the ratio given by*

$$Stretch_T(a, b) := d_T(a, b)/d_G(a, b)$$

When we replace an edge $e = (x, y)$ in the optimal tree spanner T by a swap edge f, the stretch of a pair (a, b) of nodes remains the same in the new tree $T_{e/f}$ if a and b are in the same connected component of $T - e$. The following properties further simplify the computation of the stretch of $T_{e/f}$.

Property 6.2 (Lemma 16.1.1 in [76])**.** *Let $G = (V, E)$ be any unweighted graph and let T be a spanning tree of G. For any pair of nodes $a, b \in V$ with $(a, b) \notin E$, there exists an edge $(a', b') \in E$ such that $Stretch_T(a', b') \geq Stretch_T(a, b)$.*

The above property also holds for graphs whose edges have arbitrary positive real lengths. Furthermore, we have:

Property 6.3 (Lemma 5.1 in [26])**.** *Let $G = (V, E)$ be an arbitrary graph and let T be a spanning tree G. For any edge $(a, b) \in E$, such that $l(a, b) > d_G(a, b)$ there exists an edge $(a', b') \in E$ such that $l(a', b') = d_G(a', b')$ and $Stretch_T(a', b') \geq Stretch_T(a, b)$.*

Based on the above observations, we define the concept of relevant stretch pairs, which will be used in the description of our algorithms:

Definition 6.4. *A pair of nodes a, b with $(a, b) \in E$ is called a* stretch pair. *A stretch pair $g = (a, b)$ is* relevant *for measuring the stretch of any swap edge replacing a given failing edge e if the cycle which g forms with T contains the edge e (in other words, if g is also a swap edge for e).*

6.4.2 Naive Approach

To compute the best swap edge for an edge $e \in T$, we need to compare the up to $\Theta(m)$ possible swap edges for the failing edge e. Unfortunately, there is no straightforward way of selecting the best among these candidates without evaluating each possible candidate. A simple trick such as choosing the swap edge minimizing the detour around the failure typically does not give an optimal solution. For instance, see the counterexample shown in Figure 6.2. This example can be generalized to obtain an arbitrary large difference between the stretch for the best swap f and the minimum detour edge g.

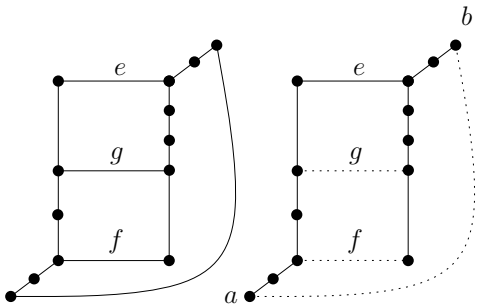

Figure 6.2: *An example showing that minimizing detour length does not minimize the stretch: On the left side, a 2-edge-connected graph G is shown, and on the right side the given tree spanner with stretch 8 is shown. Assuming that all edges have equal length, the swap edge f minimizes the stretch to the value 9. However, the swap edge minimizing the detour length is g, which yields a stretch of 10, attained by the stretch pair (a, b).*

We first consider the brute force method for solving the *All-Best-Swaps* problem in a tree spanner. For each edge e of T, we can simply consider each relevant swap edge f, compute the stretch of $T_{e/f}$ and select a swap edge with smallest stretch as the best swap edge for e. Notice that there could be $\Theta(m)$ relevant swap edges for each edge $e \in T$. Thus, the algorithm would iterate over $\Theta(nm)$ pairs (e, f) of failing edges and corresponding relevant swap edges. The running time of this approach clearly depends on how fast the stretch of a given tree $T_{e/f}$ can be computed. Due to Property 6.2, we know that for computing the stretch of $T_{e/f}$, it is sufficient to consider only those stretch pairs (a, b) where a and b are adjacent in G, as opposed to all $O(n^2)$ pairs of nodes in V. Thus the stretch of the tree $T_{e/f}$ can be computed in $O(m)$ time, if the stretch of each $(a, b) \in E$ can be computed in constant time. This can be done using some preprocessing as explained next.

Lemma 6.5. *After preprocessing in time $O(mn + n^2 \log n)$, for any failing edge $e \in E_T$, swap edge $f \in E \backslash E_T$, and relevant stretch pair (a, b), the stretch of (a, b) in $T_{e/f}$ can be computed in $O(1)$ time.*

Proof. First, we obtain distances between all pairs of nodes in G in time $O(nm + n^2 \log n)$, using the standard "all-pairs shortest paths" algorithm. Next, we root the tree T at an arbitrary node r and compute the "to-root" distance $d_T(r, v)$ for each node $v \in V$, with a single preorder traversal of T. Finally, we construct a data structure which provides the nearest common ancestor of any two given nodes in constant time. Such a data structure can be computed in $O(n)$ time, for example using the method described in [40].

After the preprocessing, we consider each relevant stretch pair (a, b), i.e. each $(a, b) \in E \backslash E_T$ where a and b lie on different sides of the failing edge[2]. For each such pair (a, b) we compute the distance in $T_{e/f}$ as

$$d_{T_{e/f}}(a, b) = d_T(a, u) + l(f) + d_T(v, b),$$

where $f = (u, v)$ and u lies on a's side of the cut induced by e. Further, $d_T(a, u)$ (and similarly $d_T(b, v)$) is computed as

$$d_T(a, u) = d_T(a, \mathrm{nca}(a, u)) + d_T(u, \mathrm{nca}(a, u)).$$

[2]This can be checked using a "preorder/inverted preorder" labeling. For details, see Section 7.2 of this thesis.

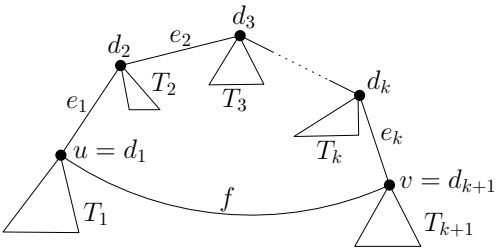

Figure 6.3: *The cycle that a non-tree edge f forms with the given spanning tree.*

Each of the five terms in the equations above can be computed in constant time. Note that the distance in the tree T between any node a and one of its ancestors, can be obtained as the absolute difference between the "to-root" distance of these two nodes (which can be looked up in the data structure that was precomputed). Finally, to obtain the stretch of (a, b), we divide $d_{T_{e/f}}(a, b)$ by $d_G(a, b)$, which has already been precomputed. □

To summarize, $d_{T_{e/f}}(a, b)$ and thus, the stretch of (a, b) can be computed in constant time, for each of the $O(m)$ relevant stretch pairs, for a particular e and f. This implies the following result:

Theorem 6.6. *The* All-Best-Swaps *problem in a tree spanner can be solved in $O(nm^2)$ time.*

In the next few sections, we present algorithms with better time complexity.

6.5 An $O(m^2 \log n)$ Time Solution for General Graphs

In the following, we describe an algorithm which computes all best swap edges of a tree spanner in $O(m^2 \log n)$ time and $O(m)$ space. The idea of the algorithm (called *BestSwaps*) is the following. We consider each potential swap edge $f \in E \backslash E_T$ separately, focusing on the cycle which $f = (u, v)$ forms with T (see Figure 6.3). This cycle consists of the edges $e_1, e_2, \ldots e_k$ which form a path in T. We use the

Algorithm 8: BestSwaps

```
1  for e ∈ T do
2      Current-Best(e) := ∞
3  end
4  for f = (u, v) ∈ E\T do
5      Assign indices to the nodes in G
6      Initialize Data Structure H
7      for e_i ∈ Path_T(u, v) do
8          Add to H stretch pairs in Start_i
9          Remove from H stretch pairs in End_i
10         St(e_i, f) := GetMax(H)
11         if St(e_i, f) < Current-Best(e_i) then
12             Update Current-Best(e_i)
13         end
14     end
15 end
```

nodes of V which lie on the path in T from $u = d_1$ to $v = d_{k+1}$ to partition T into subtrees as follows. With each node d_i on this path, we associate the subtree T_i, which consists of the connected component containing d_i in the graph $T \backslash \{e_1, e_2, \ldots, e_k\}$. For a given failing edge $e_i = (d_i, d_{i+1})$ for which f is a swap edge, the set of relevant stretch pairs contains all non-tree edges where one endpoint lies in some subtree $T_1, \ldots, T_i$, and the other endpoint lies in some subtree $T_{i+1}, \ldots, T_{k+1}$. We assign to each node the index i of the subtree T_i containing it. For any edge $(a, b) \in E \setminus E_T$ whose endpoints are a relevant stretch pair for f, this defines an order of the endpoints: if a's index is smaller than the index of b, we say that a is the *lower* endpoint, and that b is the *upper* endpoint. In order to evaluate f as a potential swap edge, we need to compute the stretch for every relevant stretch pair with respect to f and some failing edge e_i. Note that for $f = (u, v)$, the stretch of the pair a, b is given by $(d_T(a, u) + l(f) + d_T(b, v))/d_G(u, v)$, which is independent of the failing edge e_i.

We consider the potential failing edges $e_1, e_2, \ldots, e_k$, in that order and evaluate f as a potential best swap with respect to each e_i in turn. Observe the following: If $S(e_{i-1})$ is the set of relevant stretch pairs when considering f as a swap for e_{i-1}, then $S(e_i)$, the

set of relevant stretch pairs when considering f as a swap for e_i, is $S(e_i) = \big(S(e_{i-1}) \cup Start_i\big) \setminus End_i$, where $Start_i$ is the set of stretch pairs whose lower endpoint is d_i, and End_i is the set of stretch pairs whose upper endpoint is d_i. Therefore, we store the set $S(e_i)$ in a data structure H and update it as we move from e_i to e_{i+1}. To compute $S(e_i)$ from $S(e_{i-1})$, all stretch pairs that become relevant are added to H and all stretch pairs that become irrelevant are deleted from H. The data structure H we use to store the set $S(e_i)$ can be implemented as a heap where the priority of a stretch pair (a, b) is defined by the stretch value $Stretch_T(a, b)$. Notice that this stretch is independent of the failing edge e_i for a fixed swap edge f, and therefore the priority of a stretch pair stored in the heap need never be changed. The largest element in H yields the worst stretch pair for f replacing e_i. We simply check whether this value is smaller than the stretch of the current best swap edge for e_i (which we maintain in a separate data structure) and update the current swap edge for e_i if required. Once we have performed the above process for every edge $f \in E \setminus E_T$, we have obtained for each edge in T a best swap edge. Hence:

Theorem 6.7. *The algorithm* BestSwaps *computes all the best swap edges of a tree spanner in $O(m^2 \log n)$ time and using $O(m)$ space.*

Proof. We first show that the algorithm takes $O(m^2 \log n)$ time. For each swap edge $f = (u, v) \in E \setminus E_T$, the algorithm does the following. First, it assigns to each node z its index i, which denotes the subtree T_i in which it is contained, as described above. Assigning these indices takes $O(n)$ time. The non-tree edges of the graph G are partitioned into sets $Start_i$ and End_i, corresponding to nodes d_i in the path from u to v. This can be done in $O(m)$ time. Further, for each pair (f, e) of a swap edge f and a failing edge e, the algorithm performs a number of insertions and deletions[3] on the heap H. For a fixed swap edge f and all possible failing edges e_i, any stretch pair is inserted at most once and deleted at most once. Thus we require $O(m)$ heap operations, i.e. $O(m \log m)$ time for the process corresponding to each edge $f \in E \setminus E_T$. Note that we also need to compute the stretch of a stretch pair before inserting it, which however this takes only constant time using Lemma 6.5. Thus, in total the algorithm requires $O(m^2 \log m) = O(m^2 \log n)$ time.

[3]For the deletions, we assume that whenever an element is inserted into the heap, a pointer to its position in the heap is stored, such that the element can later be found in constant time and then removed in logarithmic time.

As for the space requirements, the heap data structure H requires $O(m)$ space for storing at most m elements. Storing the current best for each edge $e \in T$ requires $O(n)$ space. □

6.6 An $O(n^3)$ Time Solution for Unweighted Graphs

In this section, we consider a dynamic programming approach for computing all best swaps in an unweighted graph. We compute the best swap edge for each of the $n-1$ edges of T in a separate computation, each requiring $O(n^2)$ time and $O(n^2)$ space. For each failing edge $e = (l, r)$, we root the two trees T^l (for "left") and T^r (for "right") of $T - e$ at the nodes l and r, respectively. Recall that the stretch of a swap edge $f = (u, v)$ is obtained at some stretch pair a, b, whose stretch is $d_{T_{e/f}}(a, b)/d_G(a, b)$. In unweighted graphs, $d_G(a, b) = 1$ and hence the maximum stretch is obtained by the stretch pair a, b for which $d_{T_{e/f}}(a, b)$ is maximum. Furthermore, for $u, a \in T^l$ and $v, b \in T^r$ we have $d_{T_{e/f}}(a, b) = d_{T_{e/f}}(a, u)+l(u, v)+ d_{T_{e/f}}(v, b) = d_T(a, u) + l(a, b) + d_T(v, b)$. Therefore, the stretch of a swap edge $f = (u, v)$ is equal to the length of a longest simple path from u to v in G, using only edges of $T - e$ and either exactly one non-tree edge $(a, b) \in E \backslash E_T$, or the edge e [4]. In the following, we call paths of this nature the *stretch paths* of the node pair u, v. In our approach, we compute the length of a longest stretch path for each of the $O(n^2)$ node pairs u, v, even for those which are not linked by an edge in G. It turns out that by partitioning the set of all stretch paths into nine different types, and by computing the length of the longest stretch paths of a particular type for each node pair u, v in a suitable order, all these lengths can be computed in $O(n^2)$ time by dynamic programming. In the following, we describe this approach in detail.

The *type* of a stretch path $\mathcal{P}$ depends on which of the edges incident to $u \in T^l$ and $v \in T^r$ it includes. If $\mathcal{P}$ contains the edge $(u, p(u))$, we say it goes *up* on the left side. If $\mathcal{P}$ contains an edge (u, q) for some $q \in C(u)$, we say it goes *down* on the left side. Furthermore, if $\mathcal{P}$ uses a non-tree edge incident to u (and hence does

[4] We have to include e here because the stretch is measured with respect to G, not with respect to $G - e$.

not contain any other edge from T^l), we say it *stays* at u. The corresponding definitions hold for the right side of stretch paths. Hence, we have the following nine types of paths (where the first word corresponds to the left side of the path, and the second to the right side): `Stay-Stay`, `Stay-Down`, `Down-Stay`, `Stay-Up`, `Up-Stay`, `Down-Down`, `Down-Up`, `Up-Down`, `Up-Up`.

For each `TypeA-TypeB` combination and each node pair u, v, we use $\texttt{TypeA-TypeB}(u, v)$ to denote the length of a longest stretch path from u to v of type `TypeA-TypeB`. If no stretch path from u to v of type `TypeA-TypeB` exists, then we define

$$\texttt{TypeA-TypeB}(u, v) := -\infty.$$

We compute the longest path of each type with an inductive computation (dynamic programming) requiring $O(n^2)$ time. To that end, we first explain the necessary recursive equations. We start with `Stay-Stay` paths: For a given node pair u, v, the only possible path of that type is composed of the edge (u, v) (if present). Thus, we have

$$\texttt{Stay-Stay}(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \backslash E_T \cup \{e\} \\ -\infty & \text{otherwise.} \end{cases}$$

Clearly, $\texttt{Stay-Stay}(u, v)$ for all $u, v \in V$ can be obtained in $O(n^2)$ time. It is easy to see that the length of a longest stretch path of type `Stay-Down` satisfies the following recursion:

$$\texttt{Stay-Down}(u, v) = \\ 1 + \max_{q \in C(v)} \Big\{ \max\{\texttt{Stay-Stay}(u, q), \texttt{Stay-Down}(u, q)\} \Big\}.$$

Naturally, the symmetric equation holds for $\texttt{Down-Stay}(u, v)$. Note that this recursion can be translated into a dynamic program: since $\texttt{Stay-Stay}(u, q)$ for any $u, q \in V$ is already available from the previous computation, we only need to ensure that $\texttt{Stay-Down}(u, q)$ is available for all $q \in C(v)$ when $\texttt{Stay-Down}(u, v)$ is computed. This is guaranteed if we consider the pairs u, v in an order in which the v's occur in postorder. Thus, the entries $\texttt{Stay-Down}(u, v)$ and $\texttt{Down-Stay}(u, v)$, for all $u, v \in V$, can be computed in $O(n^2)$ time.

To compute all the `Stay-Up` paths, we need information on paths of type `Stay-Stay` as well as of type `Stay-Down`. More pre-

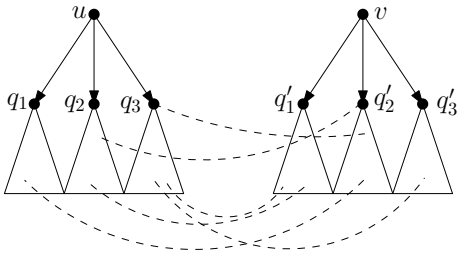

Figure 6.4: *The possible stretch paths of type* `Down-Down` *for the node pair* u, v.

cisely:

$$\texttt{Stay-Up}(u,v) = \max\Big\{ \\ 1 + \texttt{Stay-Stay}(u, p(v)), \quad 1 + \texttt{Stay-Up}(u, p(v)), \\ 2 + \max_{q \in C(p(v)), q \neq v} \{\texttt{Stay-Down}(u,q), \texttt{Stay-Stay}(u,q)\} \quad \Big\}.$$

A symmetric equation holds for $\texttt{Up-Stay}(u,v)$. Assuming that the values `Stay-Stay` and `Stay-Down` have been previously computed for all pairs of nodes, we just need to guarantee that the value $\texttt{Stay-Up}(u, p(v))$ is available when computing $\texttt{Stay-Up}(u,v)$. So, in our dynamic programming algorithm, we consider the pairs u, v in an order where the v's occur in preorder. In this way, $\texttt{Stay-Up}(u,v)$ and $\texttt{Up-Stay}(u,v)$, for all $u, v \in V$, can be computed in $O(n^2)$ time. Consider now a `Down-Down` stretch path from u to v (see Figure 6.4). We have:

$$\texttt{Down-Down}(u,v) = \\ 1 + \max\Big\{ \max_{q \in C(u)} \{\texttt{Stay-Down}(q,v), \texttt{Down-Down}(q,v)\}, \\ \max_{q' \in C(v)} \{\texttt{Down-Stay}(u,q'), \texttt{Down-Down}(u,q')\} \Big\}.$$

In order to write a dynamic program corresponding to this recursion, the node pairs u, v must be considered in an order such that all children of a node are considered before the node itself (i.e. both the trees T_l and T_r are traversed in postorder). In this way, $\texttt{Down-Down}(u,v)$

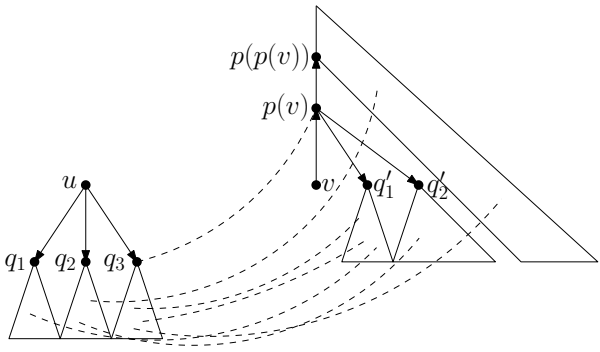

Figure 6.5: *The possible stretch paths of type* `Down-Up` *for the node pair* u, v.

for all $u \in V_{T_l}$, $v \in V_{T_r}$ can be computed in $O(n^2)$ time. Next, let us focus on the `Down-Up` paths (see Figure 6.5). Here, we have

$$\begin{aligned}\texttt{Down-Up}(u,v) = \max\Big\{ & \\ & 1 + \texttt{Down-Stay}(u, p(v)), \quad 1 + \texttt{Down-Up}(u, p(v)), \\ & 2 + \max_{q' \in C(p(v)), q' \neq v} \{\texttt{Down-Down}(u, q'), \texttt{Down-Stay}(u, q')\} \Big\}.\end{aligned}$$

We omit the equation for $\texttt{Up-Down}(u, v)$, which is completely symmetric. By considering all pairs $u, v \in V$ such that the v's occur in preorder, $\texttt{Down-Up}(u, v)$ and $\texttt{Up-Down}(u, v)$ could be computed in $O(n^2)$ time. Finally, the length of a longest `Up-Up` stretch path for u, v (see Figure 6.6) can be expressed as

$$\begin{aligned}\texttt{Up-Up}(u,v) = \max\Big\{ & \\ & 1 + \texttt{Up-Stay}(u, p(v)), \quad 1 + \texttt{Up-Up}(u, p(v)), \\ & 1 + \texttt{Stay-Up}(p(u), v), \quad 1 + \texttt{Up-Up}(p(u), v), \\ & 2 + \max_{q' \in C(p(v)), q' \neq v} \{\texttt{Up-Down}(u, q'), \texttt{Up-Stay}(u, q')\}, \\ & 2 + \max_{q \in C(p(u)), q \neq u} \{\texttt{Down-Up}(q, v), \texttt{Stay-Up}(q, v)\} \Big\}.\end{aligned}$$

To obtain $\texttt{Up-Up}(u, v)$ for all $u, v \in V$ in $O(n^2)$ time, the pairs

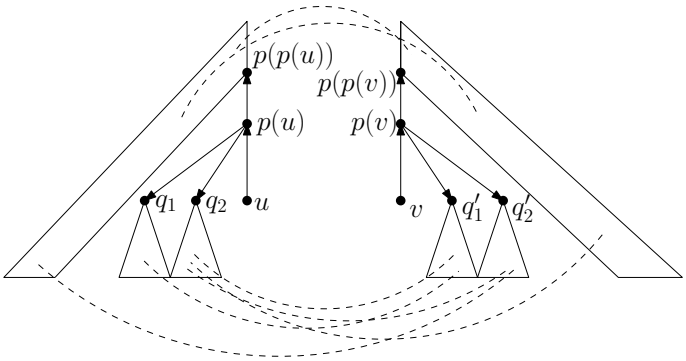

Figure 6.6: *The possible stretch paths of type* `Up-Up` *for the node pair* u, v.

are considered in an order in which both the u's and the v's occur in preorder.

Each of these dynamic programs fills an $(n \times n)$-matrix, and thus needs $O(n^2)$ space. As mentioned in the beginning, we repeat these computations for each of the $n-1$ edges $e \in E_T$. Then, the algorithm computes, for each non-tree edge $f = (u, v)$, the stretch of $T_{e/f}$ as

$$\max \Big\{ \begin{array}{lll} \texttt{Stay-Stay}(u,v), & \texttt{Stay-Down}(u,v), & \\ \texttt{Down-Stay}(u,v), & \texttt{Stay-Up}(u,v), & \texttt{Up-Stay}(u,v), \\ \texttt{Down-Down}(u,v), & \texttt{Down-Up}(u,v), & \texttt{Up-Down}(u,v), \\ \texttt{Up-Up}(u,v) & & \end{array} \Big\},$$

in constant time. After each such computation, we can delete the computed matrix from memory, only storing the best swap edge found for the considered failing edge e. Thus, the total space complexity of our approach is $O(n^2)$. In short, we have the following:

Theorem 6.8. *In unweighted graphs, all best swap edges of a tree spanner can be computed in $O(n^3)$ time and $O(n^2)$ space.*

6.7 Best Swap Tree versus Recomputed Tree Spanner

In this section, we investigate how a best swap tree compares with an optimal tree spanner of $G - e$, with respect to the maximum stretch. We show that at least for unweighted graphs, the stretch is at most twice as large in the swap tree as in the tree spanner.

Lemma 6.9. *For any failing edge e in an optimal tree spanner of an* unweighted *graph G, the maximum stretch of the* swap tree*, measured w.r.t. distances in G, is at most two times larger than the stretch of an optimal tree spanner of $G - e$. The bound of two is tight.*

Proof. Let T be an optimal tree spanner of G, let k be the stretch of T, and let T' be a best swap tree when $e = (x, y)$ fails. Let (a, b) be a stretch pair for which the stretch with respect to T' is maximum, i.e.

$$(a, b) = \arg \max_{(i,j)\in E} \frac{d_{T'}(i, j)}{d_G(i, j)}.$$

Further, let (u, v) be a best swap edge for e. We have

$$\begin{aligned}
\frac{d_{T'}(a, b)}{d_G(a, b)} &\leq d_T(a, x) + d_T(x, u) + l(u, v) + d_T(v, y) + d_T(y, b) \\
&\leq d_T(a, b) - l(x, y) + d_T(u, v) - l(x, y) + l(u, v) \\
&\leq \frac{d_T(a, b) + d_T(u, v) - 1}{d_G(a, b)} \\
&\leq k + \frac{d_T(u, v)}{d_G(a, b)} \leq 2k.
\end{aligned}$$

For any other spanning tree of G (including the optimal spanner of $G - e$), the stretch must be at least k, and hence the result follows. An example that achieves the bound of two is shown in Figure 6.7. □

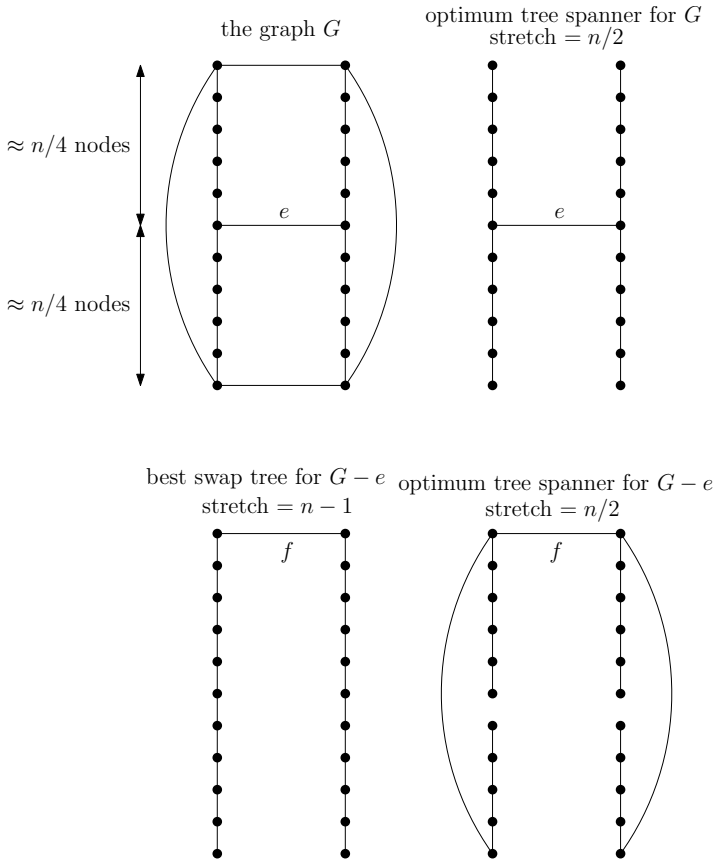

Figure 6.7: *An example of a graph G where the stretch of any best swap tree is two times worse than an optimum tree spanner of $G - e$.*

Chapter 7

Routing Issues

A natural question arises concerning routing in the presence of a failure: After replacing the failing edge e by a best swap edge f, how do we adjust our routing mechanism in order to guide messages to their destination in the swap tree $T_{e/f}$? And how is routing changed back again after the failing edge has been repaired? Clearly, it is desirable that the adaptation of the routing mechanism is as fast and inexpensive as possible. The use of swap edges in a communication network only makes sense if the routing scheme can cope with the necessary changes required by a swap. To the best of our knowledge, no such routing scheme has been described in the literature prior to our work.

7.1 Summary of Results

In this chapter, we propose a compact routing scheme for trees which can quickly and inexpensively adapt routing when a failing edge is replaced by a best swap edge. Notably, our scheme does not require an additional full backup table, but assigns a label of $c \log n$ bits to each node (for some small constant c); a node of degree δ stores its own label and the labels of all its neighbors in the tree, which amounts to $\delta c \log n$ bits per node, or $2nc \log n$ bits in total. We will show how, given this labeling, knowledge of the labels of both incident nodes of a failing edge and the labels of both incident nodes of its swap edge is sufficient to adjust routing.

Our routing scheme is presented in Section 7.2. Motivated by this routing scheme, we further consider a different variant of the swap edge computation problem, where instead of optimizing the quality of the resulting tree, we minimize the time required for the routing adaptation. This is useful whenever recovery of a failed edge is so quick that the speed of adjusting the routing tables takes priority. This boils down to replacing each failing edge by a swap edge whose endpoints are close to the endpoints of the failing edge. For two different variants of this problem (depending on whether an edge failure is detected at both of its endpoints or only at one), we give distributed algorithms with running time $O(\|\mathcal{D}\|)$ and message complexity $O(n^* + m)$ in Section 7.3. We also give centralized algorithms for both variants, using $O(m\alpha(m, n))$ time and $O(m)$ space for the first variant, and $O(m \log n)$ time and $O(m)$ space for the second, in Section 7.4.

The results in Sections 7.2.2 and 7.3 of this chapter were obtained in collaboration with Peter Widmayer and Nicola Santoro [36]. The additional results in Section 7.4 were obtained by the author of this thesis, and have not been published before.

7.2 Compact Routing with Swap Edges

7.2.1 Existing Approaches

The simplest routing scheme uses a routing table of n entries at each node, which contains, for each possible destination node, the link that should be chosen for forwarding. But this approach is not well-suited for adapting to swaps: In order to achieve shortest-path routing in the tree, a single swap may require changes in the routing tables of most nodes, which creates a memory issue as n different routing tables must be stored at each node.

A partial solution is to use two edge-disjoint spanning trees [43]. With this approach, routing can be performed with two routing tables for the two trees, where a bit in the header of each message indicates which tree should be used. However, since for this approach the route of a message during a failure must be disjoint for the initial tree, it might be much longer than the corresponding path in a best swap tree. In [44], a somewhat better solution is proposed for shortest paths trees with only one destination: The basic idea is to use a routing table

for the fault-free routing, and to have only one backup table which is used for the path to go from the point of failure to the swap. In addition, one bit in the message header is required in order to indicate whether the standard or the backup routing table must be used for this message (depending on whether the message has already crossed the swap edge or not). However, this solution does not guarantee that each message is routed on the shortest possible path in the backup network, because the swap that is chosen is not always a best swap. This solution requires that each node with degree δ stores two entries (one for the default routing table and one for the backup table) of $O(\log \delta)$ bits for routing to one destination.

Another partial solution is to store the best swap for each failing edge only at the two endpoints of the failing edge [28]. Note however that this approach does not always route messages along shortest paths in the swap tree: instead, a message will first travel along its regular path until it reaches the failing edge, and then typically has to backtrack from there to use the swap edge.

In the following, we propose to use a *compact* routing scheme for arbitrary tree types (shortest paths, minimum diameter, or any other), which is capable of always routing messages along the shortest path in the swap tree. It requires only δ entries, i.e. $\delta c \log n$ bits, at a node of degree δ, thus n entries or $2mc \log n$ bits in total, which is the same amount of space that the *interval routing* scheme of [83] requires. The header of a message requires $c \log n$ bits to describe its destination.

7.2.2 Our Routing Scheme

Throughout this chapter, we assume that the tree T in which messages are routed has a designated root node r. Our routing scheme for trees is based on the labeling $\gamma : V \to \{1, \ldots, n\}^2$ defined as follows: Assign a label $\gamma(v) = (a, b)$ to each node v, where a is the number of v in a preorder traversal of T, and b is the number of v in the inverted preorder traversal of T, where children are visited in reversed order. We define a partial order $\geq$ on γ: Consider two nodes v and w, and let $(v_1, v_2) := \gamma(v)$ and $(w_1, w_2) := \gamma(w)$. Then, we define $\gamma(v) \geq \gamma(w) \Leftrightarrow (v_1 \geq w_1) \wedge (v_2 \geq w_2)$. It is folklore that this partial order satisfies

$$\gamma(w) \leq \gamma(v) \Leftrightarrow w \in T_v,$$

and thus allows to decide in constant time, for any two given nodes a and b, whether a is in the subtree of b. This property is useful for determining swap edges, since clearly, (v, w) with $v \in T_x$ is a swap edge for edge $e = (x, p(x))$ if (and only if) w is not a descendant of x (and $w \neq x$). Moreover, the same property can be used to route messages in a tree, as follows:

Basic Routing Algorithm:
A node s routes message M with destination d as follows:

(i) If $d = s$, M has arrived at its destination.

(ii) If $d \notin T_s$, s sends M to $p(s)$.

(iii) Otherwise, s sends M to the child $q \in C(s)$ for which $d \in T_q$.

This algorithm clearly routes each message directly on its (unique) path in T from s to d. Before describing the adaptation in the presence of a swap, observe that a node s which receives a message M with destination d can locally decide whether M traverses a given edge $e = (x, p(x))$: edge e is used by M if and only if exactly one of s and d is in the subtree T_x of x, i.e., if $(s \in T_x) \neq (d \in T_x)$. Thus, to adapt routing, it is sufficient to inform all nodes about the failure of an edge (and later the repair) by two broadcasts starting at its two incident nodes (the points of failure). However, the following lemma shows that optimal rerouting is guaranteed even if only those nodes which lie on the two paths between the points of failure and the swap edge's endpoints are informed about the failure.

Lemma 7.1. *Let $e = (x, p(x))$ be a failing edge, and $f = (u, v)$ a best swap for e, where v is in T_x and u in $T \backslash T_x$, as shown in Figure 7.1. If all nodes on the path from x to v know that e is unavailable and that $f = (u, v)$ is a best swap edge, then any message originating in $s \in T_x$ will be routed on the direct path in $T_{e/f}$ from s to its destination d. Symmetrically, if all nodes on the path from $p(x)$ to u know about e and f, then any message originating in $s' \in T \backslash T_x$ will be routed on the direct path from s' to its destination d'.*

Proof. Let M be any message with source $s \in T_x$. If $d \in T_x$, then trivially M will be routed on its direct path, because it does not require edge e. If $d \in T \backslash T_x$, consider the path $\mathcal{P}_T$ from s to d in T, and

the path $\mathcal{P}_{T_{e/f}}$ from s to d in $T_{e/f}$. Consider the last common node i of $\mathcal{P}_T$ and $\mathcal{P}_{T_{e/f}}$ in T_x. The path composed of the paths $\langle x, \ldots, i\rangle$, $\langle i, \ldots, v\rangle$ is exactly the unique path in T from x to v, so node i lies on that path. Obviously, M will be routed on the direct path towards

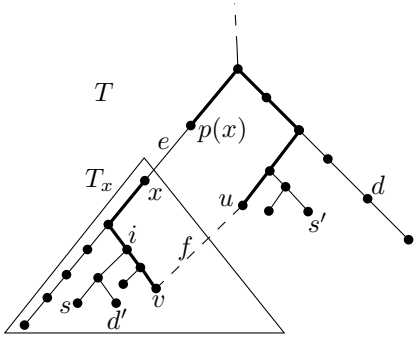

Figure 7.1: *Only some nodes need to know about failure of edge* $e = (x, p(x))$.

d up to i. As i lies on the path from x to v, it knows about the failure and the swap, and will route M towards v. Because each node on the path from i to v also knows about the swap, M will proceed on the direct path to v. At v, M will be routed over the swap edge f, and from u onwards, M is forwarded on the direct path from u to d. □

Given Lemma 7.1, we propose the following "lazy update" procedure for informing nodes about an edge failure:

> **Algorithm** LAZYSWAP**:**
> If an edge fails, no action is taken as long as no message needs to cross the failed edge. As soon as a message M which should be routed over the failing edge arrives at the point of failure, information about the failure and its best swap is attached to message M, and M is routed towards the swap edge. On its way, all nodes which receive M route it further towards the swap edge, and remember for themselves the information about the swap.

Observation (Adaptivity). *After one message M has been rerouted from the point of failure to the swap edge, all messages originating in the same side of T as M (with respect to the failing edge) will be*

routed to their destination on the direct path in the tree (i.e., without any detour via the point of failure).

If a failing edge has been replaced by a swap edge, then all nodes which know about that swap must be informed when the failure has been repaired. Therefore, a message is sent from the point of failure to the swap edge (on both sides if necessary), to inform these nodes, and to deactivate the swap edge.

Of course, one could also decide to always inform the nodes as specified in Lemma 7.1 about a failure as soon as it occurs. Whether such a proactive dissemination of failure information is better than the LAZYSWAP approach depends on the frequency and repair time of faults in the network at hand.

Remark

The above routing scheme has the disadvantage that each node must know the labels of all its neighbors. Thus, an individual node is potentially required to store a lot more than $O(\log n)$ bits. This drawback can be removed by combining the above scheme with a compact routing scheme for the designer-port model, see e.g. [94]: Such a routing scheme assigns a label of $O(\log n)$ bits to every node, such that the correct forwarding port for a given destination can be computed solely on the basis of the labels of the current position and the destination. The labels we introduced in our scheme are then only used to determine whether a message needs to be rerouted (because it would otherwise attempt to use the failing edge). As this is possible solely on the basis of the labels of the message's current position and its destination, this combination of labels yields a compact routing scheme which can efficiently adapt to swaps.

7.3 Distributed Computation of Close Swaps

In the routing scheme described in Section 7.2, the time required to adapt to an edge failure by activating a swap edge depends on the lengths of the paths between the two points of failure and the corresponding two endpoints of the swap edges. Two different possible models of failure detection seem reasonable:

1. The failure of an edge is detected at both of its endpoints concurrently.

2. The failure of an edge is detected at one of its endpoints only.

Of course, in some systems it might be unknown in advance whether one or both endpoints will detect a given failure. In such a system, the latter of the above variants would minimize the worst-case time to adapt routing.

If the prime goal is to reconnect the network quickly after an edge failure, and the quality of the resulting tree is less important, then one should precompute swap edges which are "closest" to the failing edge. In the following, we present two efficient distributed algorithms for computing such swap edges in both models of failure detection. Both of these algorithms employ the same basic principle as the algorithm BESTDIAMSWAP described in Section 5.5.4: For each failing edge $e = (x, p(x))$, first all nodes in the subtree T_x compute their locally best swap edge, and then a minimum finding process finds a globally best swap edge for e. The difference lies only in the means of computing the quality of a given local swap edge candidate f, given a failing edge e. Therefore, in the following sections, we describe only how to evaluate quality efficiently, and do not again describe the actual swap edge computation algorithms.

7.3.1 An Edge Failure is Detected at Both Endpoints

If an edge failure is detected at both endpoints of the failing edge, then the fastest way to inform all nodes which need to be informed in order to readjust routing, as identified by Lemma 7.1, is to send two messages, each starting at one endpoint of the failing edge $e = (x, p(x))$, to the two respective endpoints of the swap edge $f = (u, v)$. Thus, the time until this message reaches both destinations is the maximum length of the two paths connecting f's endpoints with e's endpoints, i.e., $\max\{d(x, v), d(p(x), u)\}$ (see Figure 7.2).

In order to evaluate a local swap edge candidate $f = (u, v)$ for a given failing edge e, a node $v \in T_x$ must compute

$$\max\{d(x, v), d(p(x), u)\}.$$

For a given edge $f = (u, v)$, let nca_f denote the nearest common ancestor (in T) of its endpoints u and v. It is easy to see that $d(p(x), u) =$

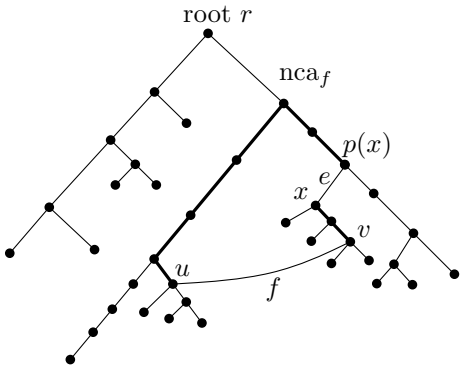

Figure 7.2: *Illustration of the distances $d(x, v)$ and $d(p(x), u)$.*

$d(p(x), \mathrm{nca}_f) + d(u, \mathrm{nca}_f)$. We propose to use again the folklore labeling described in Section 7.2.2. Using this labeling, there is a simple solution for providing u and v with all the information required to compute the terms

$$d(u, \mathrm{nca}_f),\ d(p(x), \mathrm{nca}_f) \text{ and } d(x, v), \tag{7.1}$$

respectively: The root r sends its own label to all nodes in T. Each other node z sends the two labels of z and $p(z)$ and the distance $d(z, p(z))$ to all nodes in its subtree T_z. In total, each node in the tree then knows its entire path to the root, including the labels of all nodes and all distances. This process clearly terminates in $O(\|\mathcal{D}\|)$ time and requires $O(n^*)$ messages. Given this information, a node can compute distances between two arbitrary ancestors of itself. Hence, node v can compute $d(x, v)$, and if nca_f were known to both u and v, then in the same way the other terms listed in (7.1) could also be computed: v computes $d(p(x), \mathrm{nca}_f)$, and u computes $d(u, \mathrm{nca}_f)$.

It remains to show how nca_f can be computed locally. This is where we make use of the labeling scheme once again: As u knows the label of $p(x)$, it can determine in constant time whether a given node t is an ancestor of $p(x)$. Since u knows all nodes on the path from u to the root (and all their labels), one of which is nca_f, it just needs to find the lowest node on its path to the root which is still an ancestor of $p(x)$. Similarly, v can also compute nca_f locally.

As a second step, each node u incident to some swap edge candidate $f = (u, v)$ sends $d(u, \mathrm{nca}_f)$ to node v across the edge f

(note that nca_f depends only on the swap edge, not on the failing edge). This step requires two messages per non-tree edge, which amounts to $O(m)$ additional messages in total. Then, node v knows all the terms in (7.1), and can evaluate the quality of swap edge $f = (u, v)$ for the given failing edge $e = (x, p(x))$ as $\max\{d(u, \mathrm{nca}_f) + d(p(x), \mathrm{nca}_f), d(v, x)\}$.

Let us summarize. As in Chapter 5, by a *message of constant size*, we mean a message which only contains a constant number of atomic items, such as node identifiers, edge lengths, numbers, etc.

Theorem 7.2. *In a network where the failure of an edge is detected at both endpoints concurrently, all closest swap edges of a spanning tree can be computed in an asynchronous distributed setting with $O(n^* + m)$ messages of constant size, and in $O(\|\mathcal{D}\|)$ time.*

7.3.2 An Edge Failure is Detected at One Endpoint Only

If the failure of an edge is detected at one of its endpoints only, then the fastest way of informing all nodes as identified by Lemma 7.1 is to send one message from the point where the failure is detected to the endpoint of its swap edge on the corresponding side. The message then must cross the swap edge and continue towards the other endpoint of the failing edge. Thus, the time used by the message to inform all required nodes is proportional to $d(x, v) + l(f) + d(u, p(x))$, irrespective of which side of the failing edge detects the failure. Note that an algorithm for computing the distances $d(x, v)$ and $d(u, p(x)) = d(u, \mathrm{nca}_f) + d(p(x), \mathrm{nca}_f)$ has already been described in Section 7.3.1. A slight modification of this algorithm thus computes all best swap edges in this scenario, using $O(n^* + m)$ messages and $O(||\mathcal{D}||)$ time.

Theorem 7.3. *In a network where the failure of an edge is only detected at one of its endpoints, all closest swap edges of a spanning tree can be computed in an asynchronous distributed setting with $O(n^* + m)$ messages of constant size, and in $O(\|\mathcal{D}\|)$ time.*

7.4 Centralized Computation of Close Swaps

If the network at hand is planned in a centralized way, then an algorithm for computing closest swaps is useful. In this section, we therefore provide efficient algorithms for the centralized model of computation.

7.4.1 An Edge Failure is Detected at One Endpoint Only

This problem can be solved using the transmuter approach as described in Section 3.3.1. More precisely: Recall that the quality of a swap edge $f = (u, v)$ for $e = (x, p(x))$, where $v \in T_x$, is defined here as

$$obj(T_{e/f}) := d_T(v, x) + d_T(u, p(x)).$$

This value depends on the failing edge, and is hence not directly suitable for the transmuter approach. However, a simple trick, which is similar to those in [67, 9], solves this problem: We add to the quality of edge f the length $l(e)$ of the failing edge. We define

$$\begin{aligned} obj'(f) &:= obj(T_{e/f}) + l(e) \\ &= d_T(v, x) + d_T(u, p(x)) + l(x, p(x)) = d_T(u, v), \end{aligned}$$

which is independent of the failing edge e. Moreover, adding $l(e)$ to the quality of all swap edges for e clearly has no effect on the ranking of swap edges, that is, a best swap measured with respect to obj' is also a best swap with respect to obj. Hence, we can assign the value $obj'(f) = d_T(u, v)$ to every non-tree edge $f = (u, v)$, and then use the transmuter approach to obtain all best swap edges. Note that $d_T(u, v)$ is easily computed in constant time for each non-tree edge (u, v), as was already explained in Section 7.2.2. Therefore:

Theorem 7.4. *In a network where the failure of an edge is detected at both endpoints concurrently, all closest swap edges of a spanning tree can be computed in the centralized setting in $O(m\alpha(m, n))$ time and $O(m)$ space.*

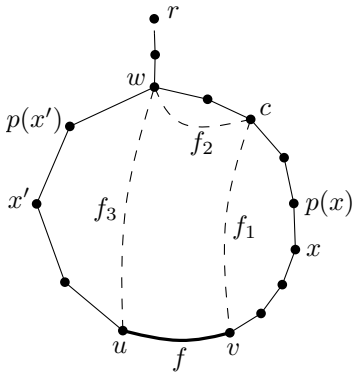

Figure 7.3: *Illustration of the replacement of a swap f by at most three virtual swap edges.*

7.4.2 An Edge Failure is Detected at Both Endpoints

If an edge failure is detected at both endpoints of the failing edge, a centralized algorithm can again be obtained using a transmuter. However, since in this case the quality of a swap edge $f = (u, v)$ for $e = (x, p(x))$, where $v \in T_x$, is defined as

$$obj(T_{e/f}) := \max\big\{d_T(v, x), d_T(u, p(x))\big\},$$

it is somewhat more difficult to assign values to non-tree edges which describe their quality independent of the failing edge. In fact, it is necessary to replace the present non-tree edges by virtual swap edges (similar to Section 4.5.1). Let c be the center node on the path from u to v in T, and assume w.l.o.g. that $v \in T_c$ (it is clear that either $u \in T_c$ or $v \in T_c$). Furthermore, let w be the nearest common ancestor of u and v in T (see Figure 7.3). Note that w can be computed in $O(1)$ time after linear time preprocessing [40]. Below, we show how to compute the center node c for all non-tree edges in $O(m \log n)$ time. For any failing edge on the path in T from v to c, we have $obj(T_{e/f}) = d_T(u, p(x))$, and for any failing edge on the path in T from c to u, we have $obj(T_{e/f}) = d_T(v, x)$ if x lies on the path from c to w in T, or $obj(T_{e/f}) = d_T(v, p(x))$ if x lies on the path from w to u in T. Therefore, a first virtual swap edge which we introduce for representing f is $f_1 := (c, v)$. We call this a virtual swap of *type 1*. The quality of f_1 is defined as $obj(T_{f_1/e}) := d_T(u, p(x))$. Since this

value depends on the failing edge e, we have to modify it: We subtract the term $d_T(r, p(x))$ from the quality (note that the subtracted term only depends on the failing edge e, and will hence be the same for all swap edges for e). This yields

$$\begin{aligned} d_T(u, p(x)) - d_T(r, p(x)) &= d_T(u, w) + d_T(w, p(x)) - d_T(r, p(x)) \\ &= d_T(u, w) - d(r, w), \end{aligned}$$

which assigns a value to f_1 that is independent of e, and which correctly reflects the order of virtual swap edges of type 1 by quality. Thus, a best swap of type 1 for each failing edge can be determined using a transmuter.

We need additional virtual swap edges for representing f for the case when the failing edge lies somewhere on the path in T from c to u. These are defined as follows: $f_2 := (c, w)$, and $f_3 := (w, u)$. The quality of f_2 is $d_T(v, x)$, which depends on e. To make it independent, we add $d_T(x, r)$ to this term, which yields $d_T(v, x) + d_T(x, r) = d_T(v, r)$. Hence, the transmuter approach can be used to compute best virtual swap edges of type 2 as well. For f_3, the quality is $d_T(v, p(x))$, which again depends on e. To make it independent, we add $d_T(r, p(x))$: $d_T(v, p(x)) + d_T(r, p(x)) = d_T(v, r)$, which is suitable for applying the transmuter approach.

Hence, we obtain in $O(m\alpha(m, n))$ time, for every tree edge e, at most three virtual best swap edges (at most one of each type). Among these, we pick the edge which is best for e, to obtain the best real swap edge. Since the quality of each swap tree $T_{e/f}$ can be evaluated in constant time, this last step only costs $O(n)$ time.

It remains to show how we can obtain, for every non-tree edge $f = (u, v)$, the center node c of the path from u to v in T. We first compute $w = \mathrm{nca}(u, v)$ in constant time. The idea is to find c by performing two binary searches, one on the path from u to w in T, and one on the path from v to w in T. Note that we can check in constant time whether a given node c' is the center node or not. Yet, for the binary search we must also be able to jump from a node x in the tree to an ancestor of x whose depth is roughly half the depth of x. To that end, we build a data structure in a preprocessing phase consisting of $O(\log n)$ steps. In the first step, we build a tree containing each node of T, where each node has a pointer to its parent in T. In the second step, we add to every node a pointer to its nearest ancestor with even depth. In the third step, we add to every node a pointer to its nearest

ancestor whose depth is a multiple of 4, and so on. Basically, we are building a deterministic skip list for every path from a leaf of T to the root. Building this structure requires $O(n \log n)$ time and $O(n)$ space. With the help of this structure, a center node c can be found in $O(\log n)$ time for any path $\langle u, \ldots v \rangle$ in T. Hence, the center nodes corresponding to all non-tree edges can be computed in $O(m \log n)$ time in total. We have:

Theorem 7.5. *In a network where the failure of an edge is detected at only one endpoint, all closest swap edges of a spanning tree can be computed in the centralized setting in $O(m \log n)$ time and $O(m)$ space.*

Chapter 8

Discussion

In Part I, we have presented improved algorithms for computing all best swap edges in minimum diameter spanning trees, in shortest paths trees and in tree spanners. While our distributed algorithm for minimum diameter spanning trees is optimal in both time and message complexity, it remains open whether the running times of our centralized algorithms could be further improved. Although we think that the considered problems are more difficult than computing all best swap edges in a minimum spanning tree, which can be done in almost linear time, proving a lower bound remains a challenge.

In the context of swapping, the subject of node failures is still wide open, except for minimum spanning trees. In principle, precomputing (sets of) swap edges for node failures is still reasonable, since the total number of swap edges that have to be stored (and prepared for use during a failure) in total for coping with the failure of any node is not larger than the number of edges in the tree. However, it seems that efficiently computing these sets is more difficult than computing best swaps for failing edges.

It might also be interesting to investigate a variant of the swapping approach where the initial tree is chosen such that the resulting swap trees are best possible. For example, for a given graph one could compute a spanning tree for which the maximum diameter of any swap tree is minimized. It seems likely, however, that computing such a tree is harder than computing a minimum diameter spanning tree, possibly even NP-hard.

Part II

Interlude

Chapter 9

Lower Bounds for Synchronous Consensus

9.1 Motivation

In the first part of the thesis, we have considered networks where failures are transient but rare. In the second part, we look at less reliable networks, where the number of transiently failing links can be much larger. If there are too many failures, then clearly any distributed computation becomes impossible. But where does this threshold lie? To answer this question, we look at one of the most fundamental problems in distributed computing, the *consensus* problem, in synchronous networks. In the consensus problem, each processor initially has a binary input value, and has to negotiate a binary value with the other processors, which is to be output by all processors within finite time. The trivial solution of always choosing the same fixed output bit is forbidden by an additional *validity* condition, which enforces that the output cannot be independent of the input values (there are different variants of this condition, some weaker than others). The consensus problem is a cornerstone of fault-tolerant distributed computing, and has been studied in various different models [60, 27, 75].

Nevertheless, the picture is still far from complete: for example, most of the work assumes that the network topology is a complete graph. Moreover, the predominant model for failures is the *com-*

ponent failure model, where it is assumed that a component which becomes faulty at some point remains faulty throughout the computation. In contrast, we consider *transient* (or dynamic) faults of links: Although the number of concurrent faults is bounded in our model, we allow the faults to occur at any link in the network. Hence, by the time the distributed computation finishes, *every* link in the network may have failed once. This model allows more precise statements about the possibility of achieving consensus, compared to the component failure model: In the latter, each link which fails once has to be declared faulty for the remainder of the computation.

The consensus problem for transient faults in synchronous networks was initially studied for complete graphs [84], and later also for general graphs [85]. In the former setting, the bounds on the number of tolerable failures were tight for all types of faults, but not in the setting of general graphs: For these, the bounds were not tight when the allowed faults were either omissions, a combination of additions and corruptions, or Byzantine faults. More specifically, the lower bounds were expressed in terms of the maximum degree of the network, while the upper bounds were expressed in terms of the edge-connectivity of the network. Recently, another variant of synchronous consensus in complete networks was considered, where the number of faults on both incoming and outgoing transmissions at every processor is bounded in each round [88].

9.2 Summary of Results

We study the consensus problem in synchronous networks with arbitrary topology, where communication links experience dynamic failures. This means that *any* link in the network may change arbitrarily between being operational or faulty, as long as at most f links are faulty in any given round. We give tight lower bounds for the number of faults which can render consensus impossible in this setting, for only omission failures, addition and corruption failures, or Byzantine failures: Consensus in a general graph G with edge-connectivity $c(G)$ is impossible for $c(G)$ omission faults, up to $c(G)$ addition and corruption faults, or up to $\lceil c(G)/2 \rceil$ Byzantine faults. These bounds are tight, since it has been shown that consensus is possible for up to $c(G) - 1$ omission faults, up to $c(G) - 1$ addition and corruption faults, and up to $\lceil c(G)/2 \rceil - 1$ Byzantine faults [85].

We actually study a natural generalization of the consensus problem, called k-agreement, in which at least k (but not necessarily all) processors need to decide for the same output bit. The consensus problem is a special case of the k-agreement problem where k is equal to the number of processors. For general k-agreement, where $k > \lceil n/2 \rceil$ processes must agree, our bounds are not tight, but improve upon the previously known bounds in [85] for graphs whose edge-connectivity is lower than their maximum degree.

Technically, our results are obtained by generalizing the main theorem of [85]. Our proof is similar in structure to its counterpart in [85], but has two crucial differences, which we point out in the next section. The results in this chapter are the sole work of the author of this dissertation. They have not been published prior to this dissertation.

9.3 Problem Definition and Terminology

The results of this chapter hold for the synchronous model of distributed computation. Although we have briefly described this model in Chapter 2.2.2, we need to specify it in more detail in order to prove our lower bounds. Most of our notation is taken from [85], some of which originates from [27]. We consider synchronous systems composed of a set V of n *processors* $p_1, p_2, \ldots, p_n$, where all processors start executing their program in the same round. Each processor p_i has a one-bit input register with input value $x_i \in \{0, 1\}$, a one-bit output register initially containing the null value $\perp \notin \{0, 1\}$, and an unbounded amount of local storage. In particular, it has a message register m_{ij} for each of its neighbors p_j, holding a *message* $\in M \cup \{\Psi\}$ to be sent to p_j in the next round. Here, M is a fixed and possibly infinite message universe, and Ψ is the null element indicating that no message should be sent over that link. The values of the registers and of the global clock, together with the program counters and the internal storage, comprise the *internal state* of each processor.

The *initial state* of each processor defines starting values for all registers except the input register. Each processor acts deterministically; it can never change the input register nor the clock, and can change the value of its output register only once, from the null value $\perp$ to $b \in \{0, 1\}$. A processor p_i has *decided* for $b \in \{0, 1\}$ if its

output register contains b.

A *configuration* of the system consists of the internal state of all processors at a given time. In an *initial* configuration, all processors are in an initial state. Starting from an initial configuration, the system evolves as follows: In each round, the system is in some configuration C. The set of all messages sent in this round is represented by the $n \times n$ *message array* $\Lambda(C)$: If p_i and p_j are neighbors, then the entry $\Lambda(C)[i,j]$ contains the message sent by p_i to p_j (possibly Ψ); otherwise, we define $\Lambda(C)[i,j] = *$, where $* \notin M$ is a distinguished symbol.

Due to communication failures, the set of messages received may differ from $\Lambda(C)$. A *communication* is a pair (α, β) of messages, where α is the message sent and β is the message received. If $\alpha \neq \beta$, then the communication is *faulty*, otherwise *non-faulty*. Let Φ denote the set of all possible communications (i.e., $\Phi := M \times M$). We distinguish between the following three types of communication faults:

- *omissions:* $\mathcal{O} = \{(\alpha, \beta) \in \Phi \mid \alpha \neq \Psi = \beta\}$
- *additions:* $\mathcal{A} = \{(\alpha, \beta) \in \Phi \mid \alpha = \Psi \neq \beta\}$
- *corruptions:* $\mathcal{C} = \{(\alpha, \beta) \in \Phi \mid \Psi \neq \alpha \neq \beta \neq \Psi\}$

We represent the set of received messages as an $n \times n$ *transmission matrix* τ for $\Lambda(C)$, defined as follows: if p_i and p_j are neighbors, then $\tau[i,j]$ contains the communication (α, β), where $\alpha = \Lambda(C)[i,j]$ is what p_i sent and β is what p_j receives; for non-neighboring p_i, p_j, we define $\tau[i,j] = (*, *)$.

After the communication specified by τ has occurred, the global clock is incremented by one time unit; depending on its internal state, on the current clock and the received messages, each processor prepares a new message for each neighbor, and enters a new internal state. The system thus enters a new configuration, denoted $\tau(C)$, which is entirely determined by the previous configuration and the transmission matrix τ, because processors act deterministically. We call τ an *event* and the transition from one configuration to the next a *step*.

We will distinguish sets of events by the maximum number f of faulty communications. A set S of events is *f-admissible*[1], $f > 0$, if

[1]Note that this definition differs slightly from the one in [85]. In particular, our

1. for each message array Λ, there is a nonempty set of events $S(\Lambda) \subseteq S$ for Λ, and

2. no event in S contains more than f faulty communications.

Informally, this means that for every message array, at least one possible transmission matrix defining the received messages is contained in S, in which at most f faulty communications occur.

Let $R_S^1(C) = \{\tau(C) | \tau \in S(\Lambda(C))$ be the set of all possible configurations resulting from C in one step, provided that only events from S may occur. Every configuration $C' \in R_S^1(C)$ is called a *succeeding* configuration of C. More generally, let $R_S^t(C)$ be the set of all possible configurations resulting from C in $t > 0$ steps, and $R_S^+(C) = \{C' | \exists t > 0, C' \in R_S^t(C)\}$ the set of configurations reachable from C. A configuration reachable from some initial configuration is said to be *accessible*.

Our results are derived from sets of events (i.e., transmission matrices) which preserve similarities of configurations. Let us stress that our definitions of adjacency and adjacency-preservation, which follow below, differ crucially from those in [85].

Two configurations C' and C'' are called *f-cut-adjacent* with respect to $w \in V$ and a cut F^* of size at most f in G, if for each processor p that is not on w's side of the cut (i.e., all processors that are not in the connected component of $G - F^*$ which contains w), p's state is the same in C' as in C''. If two configurations C' and C'' differ in the state of at most one processor, we say that they are *neighboring*.

We call a set S of events *f-cut-adjacency-preserving* if for any two configurations C' and C'' that are f-cut-adjacent with respect to some node w and some cut F^*, there exist in S two events τ' and τ'' for $\Lambda(C')$ and $\Lambda(C'')$, respectively, such that $\tau'(C')$ and $\tau''(C'')$ are f-cut-adjacent with respect to the same cut F^* and the same node w.

A set S of events is *continuous* if for any configuration C and for any two events $\tau', \tau'' \in S$ for $\Lambda(C)$, there exists a finite sequence $\tau_0, \ldots, \tau_m$ of events in S for $\Lambda(C)$ such that $\tau_0 = \tau'$,$\tau_m = \tau''$, and $\tau_i(C)$ and $\tau_{i+1}(C)$ are neighboring, $0 \leq i < m$.

---

definition does not require that some event in S contains exactly f faulty communications.

A k-*agreement protocol* P, for $k > \lceil n/2 \rceil$, is a protocol defining the actions of all processors. It is *correct* if it satisfies the following conditions (in every possible execution):

1. **Termination.** Every processor eventually chooses its output bit.
2. **Agreement.** At least k processors choose the same output bit b.
3. **Validity.** If all processors have the same input bit b', then at least k processors must choose the output bit b'.

In the special case where k equals the number of processors ($k = n$), a k-agreement protocol is called *consensus protocol*. Note that for $k \leq \lceil n/2 \rceil$, k processors can trivially agree without communication: each processor simply writes its input bit into the output register. Therefore, we are only interested in k-agreement for $k > \lceil n/2 \rceil$.

For a fixed k (which denotes the required level of agreement) and $b \in \{0, 1\}$, we call a configuration C *b-valent* if there exists a $t > 0$ such that in each $C' \in R_S^t(C)$, at least k processors have decided for b. A configuration C is *bivalent* if $R_S^+(C)$ contains both a 0-valent and a 1-valent configuration.

9.4 Abstract Impossibility Bounds

In this section, we obtain our main theorem, which defines sufficient conditions to render k-agreement impossible. These conditions are somewhat abstract; we will use them in Section 9.5 to derive several concrete impossibility results.

Let $\mathcal{C}(P, S)$ be the set of all initial and accessible configurations when executing the protocol P and the events are those in S.

Theorem 9.1. *Consider a graph $G = (V, E)$ on n nodes, in which for every node $v \in V$, there exists a cut of size at most $f > 0$ in G, such that on v's side of the cut, there are less than $(2k-n)$ nodes. Let P be a k-agreement protocol, $k > \lceil n/2 \rceil$, for G. Let S be continuous, f-cut-adjacency-preserving, and f'-admissible. If $\mathcal{C}(P, S)$ contains two accessible neighboring configurations, where one is 0-valent and the other 1-valent, then P is not correct in spite of f' communication faults.*

Proof. Assume in contradiction that P is a k-agreement protocol for G that is correct in spite of f communication faults when the message system returns only events in S; and let A and B be two neighboring accessible configurations in $\mathcal{C}(P, S)$ that are 0-valent and 1-valent, respectively. Note that there must be exactly one processor, say w, whose state in A is different from its state in B. By the theorem's assumptions, there exists an f-cut in G, say F^*, which separates w from more than $2n - 2k$ nodes in the graph (i.e., the connected component of $G - f$ containing w consists of less than $2k - n$ nodes). Thus, the configurations A and B are f-cut-adjacent with respect to the cut F^* and the node w. Let V_w be the set of processors on w's side of the cut, and $v_w := |V_w|$. Note that $v_w < 2k - n$.

Since S is f-cut-adjacency-preserving, there exist in S two events, π for $\Lambda(A)$ and ρ for $\Lambda(B)$, such that the resulting configurations $\pi(A)$ and $\rho(B)$ are f-cut-adjacent with respect to F^* and w. Similarly, there exist two events, π' for $\Lambda(\pi(A))$ and ρ' for $\Lambda(\rho(B))$, such that the resulting configurations $\pi'(\pi(A))$ and $\rho'(\rho(B))$ are still f-cut-adjacent with respect to F^* and w. For $t > 0$, let $\pi^t(A)$ and $\rho^t(B)$ be the resulting f-cut-adjacent configurations of iterating this procedure t times. Since P is correct, there exists a $t^* \geq 1$ such that in both $\pi^{t^*}(A)$ and $\rho^{t^*}(B)$, every processor has written its output register.

Clearly, all processors in $V \backslash V_w$ must take the same decision in $\pi^{t^*}(A)$ as in $\pi^{t^*}(B)$. Let d_0 and d_1 be the number of processors in $V \backslash V_w$ which, at time t^*, have decided for 0 and 1, respectively. We have $d_0 + d_1 = n - v_w$.

As A is 0-valent, in $\pi^{t^*}(A)$ at least k processors in V have decided for 0; similarly, as B is 1-valent, at least k processors have decided for 1 in $\pi^{t^*}(B)$. This implies $v_w + d_0 \geq k$ and $v_w + d_1 \geq k$. Inserting $d_1 = n - v_w - d_0$ into the latter inequality, and then adding it to the second last inequality, we get $v_w \geq 2k - n$. However, this contradicts our earlier conclusion that $v_w < 2k - n$. □

We can now prove the main theorem.

Theorem 9.2. *Let $G = (V, E)$ be a graph on n nodes in which for every node $v \in V$, there exists a cut of size at most $f > 0$, such that on v's side of the cut, there are less than $(2k - n)$ nodes. Let S be continuous, f-cut-adjacency-preserving, and f'-admissible, $f' > 0$.*

Then no k-agreement protocol for G is correct in spite of f' communication faults in S for $k > \lceil n/2 \rceil$.

Proof. Assume P is a correct k-agreement protocol. The following two lemmas show that $\mathcal{C}(P, S)$ has an initial bivalent configuration, and that every bivalent configuration in $\mathcal{C}(P, S)$ has a succeeding bivalent configuration. Thus, there exists an execution of P in which some processor never decides, which violates the termination condition. □

Lemma 9.3. *$\mathcal{C}(P, S)$ of Theorem 9.1 has an initial bivalent configuration.*

Proof. By contradiction, let every initial configuration be b-valent for $b \in \{0, 1\}$ and let P be correct. The validity condition implies that if all input bits are 0, then the initial configuration must be 0-valent, and vice versa for 1. Thus, if there is no bivalent initial configuration, there is at least a 0-valent initial configuration A and a 1-valent initial configuration B. Consider now a sequence of initial configurations, starting with all input bits being 0, and by sequentially changing each input bit to 1, arriving at the initial configuration with all input bits being 1. As each configuration of this sequence is either 0-valent or 1-valent, the first being 0-valent and the last being 1-valent, there must be a 0-valent initial configuration and a 1-valent initial configuration which differ only in the input bit of one processor, and hence are neighboring. Hence, it follows from Theorem 9.1 that P is not correct. □

Lemma 9.4. *Every bivalent configuration in $\mathcal{C}(P, S)$ of Theorem 9.1 has a succeeding bivalent configuration.*

Proof. Let C be a bivalent configuration in $\mathcal{C}(P, S)$. If C has no succeeding bivalent configuration, then C has at least one 0-valent and at least one 1-valent succeeding configuration, say A and B. Let $\tau', \tau'' \in S$ be such that $\tau'(C) = A$ and $\tau''(C) = B$. Since S is continuous, there exists a sequence $\tau_0, \ldots, \tau_m$ of events in S for $\Lambda(C)$ such that $\tau_0 = \tau'$, $\tau_m = \tau''$, and $\tau_i(C)$ and $\tau_{i+1}(C)$ are neighboring, for $0 \leq i < m$. Consider now the corresponding sequence of configurations: $A = \tau'(C) = \tau_0(C), \tau_1(C), \tau_2(C), \ldots, \tau_m(C) = \tau''(C) = B$. Since the sequence starts with a 0-valent configuration

and ends with a 1-valent configuration, it contains a 0-valent configuration neighboring a 1-valent one. Therefore, by Theorem 9.1, P is not correct — we have a contradiction. □

9.5 Concrete Impossibility Bounds

In this section, we apply Theorem 9.1 to prove impossibility of consensus and k-agreement under different conditions. More precisely, we consider the following combinations of faults:

- omissions
- additions and corruptions
- Byzantine failures (omissions, additions and corruptions)

Note that we do not consider the cases of only additions, or only corruptions, because for these it was shown already that even consensus is possible regardless of the number of faults per round [85].

The following lemma shows continuity for a general class of sets of events, which we can later apply to each of the different sets of events we consider.

Lemma 9.5. *Let $\mathcal{F}$ be the set of allowable communication faults. Further, let S be the set of all events in which at most k faulty communications from the set $\mathcal{F}$ occur, and all other communications are non-faulty. Then the set S is continuous.*

Proof. Starting from the non-faulty event (i.e., the event where all communications are non-faulty), there exists a sequence of length at most k to any event in S, in which any two consecutive events differ in only one transmission. It follows that in the corresponding sequence of configurations, any two consecutive configurations are neighboring. Thus, any two events of S can be connected by a sequence of at most $2k + 1$ events (with the non-faulty event in the middle). □

9.5.1 Impossibility: Omission Faults

We can now use Theorem 9.2 to show how many omission faults per round any (correct) k-agreement protocol can tolerate in a given graph G.

Informally, the set of events $\mathcal{O}$ is the set of all events where at most $c(G)$ omissions occur, where $c(G)$ is the edge-connectivity of the graph G. For message array $\Lambda = (\alpha_{ij})$, let $\mathcal{O}(\Lambda)$ be the set of all events τ for Λ defined as follows: for at most $c(G)$ pairs $(i,j) \in E$, $\tau[i,j] = (\alpha_{ij}, \Psi)$, and for all other pairs $(i,j) \in E$, $\tau[i,j] = (\alpha_{ij}, \alpha_{ij})$. Then, $\mathcal{O} := \bigcup_{\Lambda} \mathcal{O}(\Lambda)$ is the set of all events containing at most $c(G)$ omission faults.

Lemma 9.6. *The set $\mathcal{O}$ is $c(G)$-admissible, continuous and $c(G)$-cut-adjacency-preserving.*

Proof. $\mathcal{O}$ is $c(G)$-admissible by construction. From Lemma 9.5, it follows that $\mathcal{O}$ is also continuous.
To see that $\mathcal{O}$ is $c(G)$-cut-adjacency-preserving, consider two configurations A and B which are $c(G)$-cut-adjacent with respect to the cut F^* and the node w. Let $\mathcal{P}_w$ be the set of processors of the side of the cut where the states may differ. Let the events τ_A and τ_B be those corresponding to configurations A and B, respectively, where each message which is sent from any $p \in \mathcal{P}_w$ over the cut F^* is hit by an omission fault, and all other messages arrive unaltered. Clearly, τ_A and τ_B are $c(G)$-admissible and hence in $\mathcal{O}$, and the configurations $\tau_A(A)$ and $\tau_B(B)$ are $c(G)$-cut-adjacent with respect to F^*. Hence $\mathcal{O}$ is $c(G)$-cut-adjacency-preserving. □

From combining Lemma 9.6 with Theorem 9.2, and setting $k = n$, it follows that no consensus protocol is correct in spite of $c(G)$ omission faults. For general k-agreement, the statement we obtain is somewhat more involved: Let x be the number of possible omission faults per round. In a graph G with n nodes, k-agreement is impossible if for every node v in G, there is a cut of size at most x such that there are less than $2k - n$ nodes on v's side of the cut. Intuitively, this means that if there is no sufficiently large set of nodes (i.e., at least $2k - n$ elements) in the graph which induces an f-edge-connected subgraph, then agreement is impossible if we allow up to f omission faults per round.

For a concrete example of what the above implies, consider a graph whose topology is a line. In such a graph, it is possible to cut away any node v using a cut of size one, such that the number of nodes on v's side of the cut is at most $\lceil n/2 \rceil$. Therefore, if we allow only one omission fault per round, the smallest k for which Theorem 9.2 is still applicable is roughly $k = \frac{3}{4}n$, implying that for any

k between $\frac{3}{4}n$ and n, k-agreement is impossible in such a graph. For lower values of k, for example $k = \lceil n/2 \rceil + 1$ ("strong majority"), however, our theorem does not apply anymore, unless we allow two or more omission faults. Note that for the same example, the bounds given in [85] only yield that k-agreement for any $k > \lceil n/2 \rceil$ is impossible with two or more omission faults. With only one omission fault, these bounds do not show impossibility of k-agreement even for $k = n$ in this example graph.

9.5.2 Impossibility: Addition and Corruption Faults

As in the previous section, we can use Theorem 9.2 to show how many faults per round any (correct) k-agreement protocol can tolerate in a given graph G, if faults can be any combination of additions and corruptions.

Informally, the set of events $\mathcal{AC}$ is the set of all events where at most $c(G)$ additions or corruptions occur, where $c(G)$ is the edge-connectivity of the graph G. For message array $\Lambda = (\alpha_{ij})$, let $\mathcal{AC}(\Lambda)$ be the set of all events τ for Λ defined as follows: for at most $c(G)$ pairs $(i,j) \in E$, $\tau[i,j] = (\alpha_{ij}, \beta_{ij})$, $\alpha_{ij} \neq \beta_{ij}$, where $\beta_{ij} \neq \Psi$ if $\alpha_{ij} \neq \Psi$, and for all other pairs $(i,j) \in E$, $\tau[i,j] = (\alpha_{ij}, \alpha_{ij})$. Then $\mathcal{AC} := \bigcup_{\Lambda} \mathcal{AC}(\Lambda)$ is the set of all events containing at most $c(G)$ additions and corruption faults.

Lemma 9.7. *The set $\mathcal{AC}$ is $c(G)$-admissible, continuous and $c(G)$-cut-adjacency-preserving.*

Proof. $\mathcal{AC}$ is $c(G)$-admissible by construction. By Lemma 9.5, $\mathcal{AC}$ is continuous.
To prove that $\mathcal{AC}$ is $c(G)$-cut-adjacency-preserving, consider two configurations A and B which are $c(G)$-cut-adjacent with respect to the cut F^* and the node w. Let $\mathcal{P}_w$ be the set of processors on that side of the cut where the states may differ. Consider the events π for $\Lambda(A) = \{\alpha_{ij}\}$ and ρ for $\Lambda(B) = \{\gamma_{ij}\}$, where for all $(i,j) \in E$

$$\pi[i,j] := \begin{cases} (\alpha_{ij}, \gamma_{ij}) & \text{if } (i,j) \in F^*,\ i \in \mathcal{P}_w \text{ and } \alpha_{ij} = \Psi \\ (\alpha_{ij}, \alpha_{ij}) & \text{otherwise,} \end{cases}$$

and

$$\rho[i,j] := \begin{cases} (\gamma_{ij}, \alpha_{ij}) & \text{if } (i,j) \in F^*,\ i \in \mathcal{P}_w \text{ and } \alpha_{ij} \neq \Psi \\ (\gamma_{ij}, \gamma_{ij}) & \text{otherwise.} \end{cases}$$

It is not difficult to verify that $\pi, \rho \in \mathcal{AC}$ and the configurations $\pi(A)$ and $\rho(B)$ are $c(G)$-cut-adjacent with respect to the cut F^*. □

From combining Lemma 9.7 with Theorem 9.2, and setting $k = n$, it follows immediately that no consensus protocol is correct in spite of $c(G)$ addition and corruption faults. For lower k, again the statement is somewhat more involved, but very similar to the case of only omissions discussed in the previous section.

9.5.3 Impossibility: Byzantine Faults

In this section, we consider Byzantine faults, which can be omissions, additions and corruptions. Again, we use Theorem 9.2 to prove impossibility bounds.

For a message array $\Lambda = (\alpha_{ij})$, let $\mathcal{ACO} := \bigcup_\Lambda \mathcal{ACO}(\Lambda)$ be the set of all events τ for Λ containing at most $\lceil c(G)/2 \rceil$ Byzantine communication faults, i.e. an arbitrary combination of omission, corruption or addition faults.

Lemma 9.8. *The set $\mathcal{ACO}$ is $\lceil c(G)/2 \rceil$-admissible, continuous and $c(G)$-cut-adjacency-preserving.*

Proof. By definition, $\mathcal{ACO}$ is $\lceil c(G)/2 \rceil$-admissible. It follows from Lemma 9.5 that $\mathcal{ACO}$ is continuous.
To prove that $\mathcal{ACO}$ is $c(G)$-cut-adjacency-preserving, consider two configurations A and B which are $c(G)$-cut-adjacent with respect to the cut F^* and the node w. Let $\mathcal{P}_w$ be the set of processors on that side of the cut where the states may differ. We split the edges of E which cross the cut F^* into two sets F_1^* and F_2^*, each containing at most $\lceil c(G)/2 \rceil$ edges. Now, consider the events π for $\Lambda(A) = \{\alpha_{ij}\}$ and ρ for $\Lambda(B) = \{\gamma_{ij}\}$, where for all $(i,j) \in E$

$$\pi[i,j] := \begin{cases} (\alpha_{ij}, \gamma_{ij}) & \text{if } (i,j) \in F_1^* \text{ and } i \in P_w \\ (\alpha_{ij}, \alpha_{ij}) & \text{otherwise,} \end{cases}$$

and

$$\rho[i,j] := \begin{cases} (\gamma_{ij}, \alpha_{ij}) & \text{if } (i,j) \in F_2^* \text{ and } i \in \mathcal{P}_w \\ (\gamma_{ij}, \gamma_{ij}) & \text{otherwise.} \end{cases}$$

It is not difficult to verify that $\pi, \rho \in \mathcal{ACO}$ and the configurations $\pi(A)$ and $\rho(B)$ are $c(G)$-cut-adjacent with respect to the cut F^* and

Fault type	Impossibility Bound	Possibility Bound
$\mathcal{A}$	$-$	∞
$\mathcal{C}$	$-$	∞
$\mathcal{O}$	$c(G)$ $(\star)$	$c(G)-1$
$\mathcal{AC}$	$c(G)$ $(\star)$	$c(G)-1$
$\mathcal{AO}$	$c(G)$ $(\star)$	$c(G)-1$
$\mathcal{CO}$	$c(G)$ $(\star)$	$c(G)-1$
$\mathcal{ACO}$	$\lceil c(G)/2 \rceil$ $(\star)$	$\lceil c(G)/2 \rceil - 1$

Table 9.1: *Summary of the known bounds for synchronous consensus with transient link faults. New results are marked with* $(\star)$.

the node w. Furthermore, since each of the cuts F_1^* and F_2^* contains at most $\lceil c(G)/2 \rceil$ edges, both π and ρ contain at most $\lceil c(G)/2 \rceil$ communication faults. Hence, $\pi, \rho \in \mathcal{ACO}$, and thus $\mathcal{ACO}$ is $c(G)$-cut-adjacency-preserving. □

Combining Lemma 9.8 with Theorem 9.2 and setting $k = n$ shows that no consensus protocol in a graph G is correct in spite of $\lceil c(G)/2 \rceil$ Byzantine faults. Again, the implications of our results for general k-agreement are very similar to those mentioned at the end of Section 9.5.1.

9.6 Discussion

The bounds we have shown strictly improve on those in [85]. In the case of consensus, that is $k = n$, Theorem 9.2 states that agreement is impossible in the cases "only omissions" and "additions and corruptions" if the number of faults is above the edge-connectivity $c(G)$ of the graph. It follows trivially that consensus remains impossible with $c(G)$ faults if the type of faults are "additions and omissions" or "corruptions and omissions". For Byzantine faults, the bound is $\lceil c(G)/2 \rceil$. An overview of these results together with the previous possibility bounds is given in Table 9.1. For consensus, our impossibility bounds are tight, since there exist algorithms in each setting which achieve consensus if the number of faults is below these bounds.

For general k-agreement, the bounds are not necessarily tight: for

n processors in a graph of n nodes forming a path, our results show that agreement of more than roughly $\frac{3}{4}n$ is impossible with one omission fault per round, but it is not clear whether agreement is possible for smaller values of k. Yet, currently not even a $\lceil n/2 \rceil + 1$-agreement protocol that can tolerate one omission fault per round is known. We conjecture that no such protocol exists, that in fact on the path with only one omission failure per round, even $\lceil n/2 \rceil + 1$-agreement (strong majority) is impossible. Proving tight lower bounds for k-agreement, however, seems to require a method different from the bivalency arguments that we used for consensus: Using our methods, one can prove that no fixed set of processors can achieve unanimity in their decision values in this setting. Yet, in the execution of an agreement protocol, the subset of processors whose decision value will agree in the end need not be predetermined by its initial state, but may depend on the failures that occur. It is this freedom which makes it difficult to prove tight lower bounds for general k-agreement.

Part III

Local Algorithms for Wireless Networks

Chapter 10

Introduction

10.1 Motivation

In this part of the thesis, we consider algorithms for wireless ad hoc networks. A *wireless ad hoc network* typically consists of a large number of nodes, each equipped with wireless communication capability, modest computational power, and a battery with limited capacity. Two nodes can communicate directly if they are within mutual communication range; to talk to distant nodes, a node can forward messages via intermediate nodes. *Wireless sensor networks* are used in an increasing number of current applied research projects, in a variety of applications ranging from agricultural management targeted at Indian farmers [72] to body area networks for human activity recognition [45]. To assist these developments, it is becoming ever more important to find solutions to the fundamental problems in employing such networks.

Wireless networks are much more dynamic than wired ones, for at least two different reasons: First, communication via radio is much more error-prone than using a wire, because there may be numerous external sources causing interference, which are beyond the control of the network operator. Second, devices which have wireless communication capability are often mobile. As they move, some devices may come within communication range, while others may become out of range. These constant changes make it unreasonable to precompute the actions to take in case a link becomes unavailable. An

additional difficulty in highly dynamic networks is that a node in the network may not be able to gather information about the entire network topology, since any information that is received from faraway nodes will be outdated when it reaches the node. Therefore, algorithms for wireless networks must be *local*, which means that their running time is so short that a node can only gather information about the part of the network in its close neighborhood. Local algorithms have the additional advantage that they can be designed without explicitly taking link faults into account, since there exists a general method to transform any local algorithm for a static network into a local algorithm for dynamic networks with negligible overhead [6]. In this part, we give efficient local algorithms for two fundamental structures in wireless ad hoc networks: Maximal independent sets, which are a basic building block for many distributed algorithms, since they can be used to coordinate the actions of the involved processors, and minimum connected dominating sets, which can serve as an energy-efficient "virtual backbone" for transmitting messages in a wireless sensor network.

To model the particular connectivity structure inherent to wireless networks, usually a restricted class of graphs (as opposed to general graphs) is considered in the literature. By far the most prominent such class are *unit disk graphs* (UDGs). However, for various reasons UDGs are a very idealized model of real wireless networks, and many generalizations have been proposed [70,87]. We adopt the class of growth-bounded graphs (defined below), which seems relatively general since it contains UDGs and many other models as special cases.

10.2 Terminology and Network Model

Terminology

Consider an unweighted graph $G = (V, E)$. The *neighborhood* $N(v)$ of a node $v \in V$ is defined as the set of all nodes $w \in V$ with $d_G(v, w) \leq 1$ (including v, in contrast to $\bar{N}(v)$). The neighborhood of a set $S \subseteq V$ is defined as $N(S) := \bigcup_{s \in S} N(s)$. Likewise, the *$i$-neighborhood* of a set of nodes S is defined recursively as $N_0(S) := S$, and $N_i(S) := N_{i-1}(N(S))$ for $i \geq 1$.

A *maximal independent set* (MIS) M in a given graph $G = (V, E)$ is a subset $M \subseteq V$ such that for every $v, w \in M$ we have $v \notin N(w)$ (*independence*) and furthermore no proper superset $M' \supset M$ with the latter property exists (*maximality*). A *maximum independent set* (MaxIS) is a MIS of largest cardinality. For a subset $S \subseteq V$, MaxIS(S) is a maximum independent set on the induced subgraph $G[S]$. A *dominating set* (DS) D for a given graph $G = (V, E)$ is a subset $D \subseteq V$ such that for every $v \in V$ there is a $w \in D$ such that $v \in N(w)$. Note that a MIS is always a DS. A *connected dominating set* (CDS) for a given graph $G = (V, E)$ is a dominating set M with the additional requirement that the nodes in M induce a connected subgraph of G. A *minimum* (or *optimal*) *connected dominating set* (MCDS) is a CDS of smallest possible cardinality, and the *minimum connected dominating set problem* is the problem of providing such a MCDS for a given graph. We are interested in a special class of graphs, called *growth-bounded graphs*.

Definition 10.1. *A class of graphs is called* growth-bounded *if there is a polynomial* bounding function $f(r)$ *such that for any graph* $G = (V, E)$ *of this class, the size of any MIS in the neighborhood* $N_r(v)$ *of any node* $v \in V$ *is at most* $f(r)$, $\forall r \geq 0$*. Furthermore, we say that a graph* G *is* growth-bounded *if it belongs to a growth-bounded class of graphs.*

An example of a growth-bounded graph class are *unit disk graphs*, which are often used to model wireless communication networks. A graph $G = (V, E)$ is a unit disk graph if it can be represented by placing a point p_v for each node $v \in V$ on the Euclidean plane $\mathbb{R}^2$, such that an edge $(u, v) \in E$ exists if and only if the distance $\|p_v - p_u\|_2$ is at most 1. Furthermore, all edges in a unit disk graph have length one by definition (they are "unweighted").

Unit disk graphs are growth-bounded with a bounding function $f(r) \in O(r^2)$, which is easily proved as follows: Consider any node v and any independent set $I \subseteq N_r(v)$, for some $r \geq 0$. The corresponding points of two (independent) nodes in I must have distance greater than one. Thus each point corresponding to a node in I exclusively occupies a disk of radius $1/2$ with an area of $(1/2)^2\pi$. As all these disks must lie inside a circle of radius $r + 1/2$, it follows that $|I| \leq \frac{(r+1/2)^2\pi}{(1/2)^2\pi} = 4r^2 + 4r + 1$. In addition to unit disk graphs, most other graph classes used to model wireless ad-hoc networks such as

quasi unit disk graphs, cover area graphs and other intersection graphs are growth-bounded [50, 70].

When discussing the asymptotic running time of our algorithms, we tacitly assume that each input graph G belongs to the same class $\mathcal{C}$ (for example all unit disk graphs).

Network Model

All algorithms in this part of the thesis use the synchronous model of distributed computation as described in Section 2.2.2. Each node has a unique identifier and knows which nodes are within its transmission range. When we say that a distributed algorithm computes a CDS (or, respectively, a MIS), we mean that each node knows after executing its code whether it is part of the CDS (MIS) or not. Assuming that a MAC layer for direct communication of neighbors has already been established, we do not consider collisions.

Chapter 11

Maximal Independent Sets

11.1 Motivation and Related Work

The efficient distributed construction of a maximal independent set (MIS) of a graph is of fundamental importance and has been studied for many years [18, 54, 5, 53]. This problem asks for a subset S of the nodes of a graph with the property that no two nodes of S share an edge and such that there is no strict superset of S with the same property. Its importance stems from the fact that it serves as a basic building block for many distributed algorithms, and that it captures the essence of symmetry breaking. Furthermore, it is a nice example of a graph-theoretical problem that admits a trivial greedy solution in a centralized setting, but is nevertheless a challenging research topic for the distributed computation community.

With the advent of wireless ad hoc and sensor networks, the study of distributed MIS computation has received further attention [59,74]. This is because algorithms for topology control and routing in such networks often construct a MIS in an initial phase [34,19,2,79]. Typically, the time required for this phase dominates the total running time of the algorithms. For example, constructing a $(1+\varepsilon)$-approximate minimum dominating set is possible in time $O(T_{\mathsf{MIS}} + \log^* n/\varepsilon^{O(1)})$

in growth-bounded graphs[1], where T_{MIS} is the time for MIS computation [47]. Furthermore, as we show in Chapter 12, a $(1+\varepsilon)$-approximate minimum connected dominating set, which can be used as a "virtual backbone" for routing, can be obtained in the same time, again in growth-bounded graphs.

Various approaches for distributively computing a MIS in UDGs are available. However, many of them have a worst-case running time of $O(n)$, e.g. [2]. As an interesting variant, [49] considers the distributed computation of a MIS in a setting where nodes are embedded in a metric space, and some geometric information about the embedding is available to the nodes. It is shown that if each node knows the distance to each of its neighbors, one can compute a MIS in $O(\log^* n)$ rounds. If nodes are located in the Euclidean plane, and each node knows its position, a MIS can even be computed in constant time. Thus, very fast running times can be achieved under the above assumptions. However, it seems debatable whether these features are implementable in a practical scenario. It may be that the energy they require is unacceptable for the wireless devices at hand, or these features (e.g. measuring distance) may simply not be available, due to hardware constraints or the type of deployment of the sensors. In any case, a distance measuring device might increase the hardware costs considerably.

In this respect, one might want to compute a MIS without using these features, yet being similarly fast. We therefore propose to use randomization instead of distance measuring. This seems like a natural choice, as intuitively, randomization should ease symmetry-breaking. Moreover, randomization has proved successful in computing a MIS in general graphs: While the fastest known deterministic algorithm for general graphs due to Panconesi and Srinivasan [73] runs in $O(n^{d\sqrt{1/\log n}})$ (where d is a constant), Luby proposed an almost exponentially faster $O(\log n)$ randomized algorithm [54]. This is close to the $\Omega(\sqrt{\log n/\log\log n})$ lower bound given in [48]. The question whether a polylogarithmic deterministic algorithm for the MIS problem exists is still open.

In contrast, for growth-bounded graphs a deterministic MIS algorithm with running time $O(\log\Delta\log^* n)$ exists [46] (where Δ is the maximum degree of the graph), while the best known lower bound for

[1] $\log^* n$ is the minimum integer t such that t times iterating the logarithm on n yields a value smaller than 1.

this class is $\Omega(\log^* n)$ (even for randomized algorithms) [53,61]. This raises the question whether randomization enables a running time improvement of the same magnitude as in general graphs. Our results, summarized in the next section, show that using randomization, one can indeed compute a MIS in growth-bounded graphs almost as quickly as with the help of distance information.

	Determ. Alg.	Rand. Alg.	Lower Bound
(a)	$O\left(n^{d\sqrt{1/\log n}}\right)$ [5,73]	$O(\log n)$ [54]	$\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ [48]
(b)	$O(\log\Delta \log^* n)$ [46]	$O(\log\log n \log^* n)$ [this chapter]	$\Omega(\log^* n)$ [53,61]
(c)	$O(\log^* n)$ [18]	-	$\Omega(\log^* n)$ [53,61]
(d)	$O(\log^* n)$ [49]	-	$\Omega(\log^* n)$ [53,61]

Table 11.1: *Summary of known results (prior to our work) on the distributed computation of a MIS in different graph classes: (a) General Graphs, (b) Growth-Bounded Graphs, (c) Constant-Degree Graphs, (d) UDGs with distance information. The lower bounds hold also for randomized algorithms.*

11.2 Summary of Results

Our main contribution is a synchronous randomized distributed algorithm for computing a MIS in growth-bounded graphs with n nodes running in $O(\log\log n \log^* n)$ rounds with high probability. By high probability, we mean that the probability is $1-O(1/n^k)$ for any fixed $k > 3$ (k affects the constant factor hidden in the asymptotic notation). Note that besides the running time bound which is probabilistic, the correctness of our algorithm is guaranteed, i.e., it is a "Las Vegas" algorithm. The nodes require only *connectivity* information about the graph, and all messages are of size $O(\log n)$, that is, our algorithm runs in the $\mathcal{CONGEST}$ model of computation. In view of the $\Omega(\log^* n)$ lower bound for randomized algorithms [61, 53], our solution is close to optimal. One year after our work, a determin-

istic algorithm for computing a MIS in $O(\log^* n)$ rounds in growth-bounded graphs was found, which is asymptotically optimal [89].

The main technical novelty introduced in this chapter is a new randomized algorithm to find a $2t$-ruling set with low induced maximum degree in $O(t)$ rounds. Even in general graphs, our approach finds an $O(\log\log\Delta)$-ruling set with induced degree $O(\log^5 n)$ in $O(\log\log\Delta)$ rounds (with high probability), which might be of independent interest. In each step of this algorithm, a subset of the nodes is selected. This subset consists of three different sets, which are designed such that their combination guarantees the desired ruling-property and a rapidly decreasing maximum degree. Two of these sets are chosen deterministically, and the third set is chosen randomly using an approach resembling Luby's algorithm [54], but with a different choice of probabilities.

The results of this chapter were obtained in joint work with Elias Vicari [37], and also appeared in his Ph.D. thesis [95].

11.3 Terminology

The *size* d_v of a node $v \in V$ is the size of $N(v)$, that is, the degree of v plus one[2]. Let Δ denote the maximum degree of any node in G. For convenience, we define $\overline{\Delta} := \Delta + 1$. A set $T \subseteq V$ is a *k-ruling set* if every node of G is within distance k from some node of T. Note that a MIS is a 1-ruling set. Throughout this chapter, we denote the probability of an event $\mathcal{E}$ by $P[\mathcal{E}]$, and the expected value of a random variable X by $E[X]$.

11.4 Distributed MIS Algorithm

In this section, we present the Algorithm MAXINDEPSET, which computes a MIS in growth-bounded graphs within $O(\log\log n \log^* n)$ rounds with very high probability. First, we outline the structure of the algorithm.

In the first phase of Algorithm MAXINDEPSET, a subset $T \subseteq V$ of G's nodes is selected, such that

[2]This $+1$ increment leads to simpler terms in our analysis, but is not of particular importance.

- T is a $O(\log \log n)$-ruling set of G, and
- the subgraph induced by T has a maximum degree of $O(\log^5 n)$.

This set is obtained by repeatedly applying Algorithm RANDSTEP (see below), using V as the initial set. Then in a second phase, the set T is further thinned out, such that the remaining set of nodes T' is an *independent* $O(\log \log n)$-ruling set of G. As in the first phase, this is achieved by repeatedly selecting particular subsets of the previous set (starting with T). Finally, in a third phase, this sparse independent set T' is extended into a maximal independent set.

The main novelty of our approach lies in the first phase of the algorithm, where the *logarithm* of the maximum degree induced by the remaining nodes decreases *geometrically* in each step, using randomization. For the second and the third phase, we use two deterministic algorithms from [46].

Note that the transition from the first phase to the second is triggered by a threshold of $O(\log^5 n)$ for the maximum degree of the remaining graph. Not knowing n, the nodes cannot know within reasonable time (i.e., locally) when the maximum degree has been reduced enough. This is why we "interleave" the executions of the algorithms for the first two phases. We will argue that this does not harm the effectiveness of either phase.

The next sections describe the building blocks of our algorithm in more detail. We will show that with high probability, each phase requires at most $O(\log \log n \log^* n)$ rounds (Phase 1 uses $O(\log \log \Delta)$ rounds with high probability), leading to the following result.

Theorem 11.1. *There exists a randomized algorithm that computes a MIS for any growth-bounded graph G within $O(\log \log n \log^* n)$ rounds of synchronous distributed computation with probability at least $1 - O(1/n^k)$ (for any $k > 3$) in the message-passing model. All messages are of size $O(\log n)$ bits.*

11.4.1 Phase 1: An $O(\log \log n)$-Ruling Set with Small Induced Degree

This phase consists of repeated executions of the Algorithm RANDSTEP: Let T_1 be the set of nodes returned by the first execution of RANDSTEP. This set induces a graph $G[T_1]$, on which RANDSTEP

is applied again, yielding T_2 and a new graph $G[T_2]$. In this way, we obtain a sequence of sets T_i which are increasingly sparse, while at the same time remaining $O(\log \log n)$-ruling.

The central idea behind Algorithm RANDSTEP is to choose a random subset of all nodes, such that the maximum degree of the induced subgraph decreases rapidly, i.e., from Δ to Δ^c with $c < 1$, but assuring that the remaining set of nodes is a 2-ruling set of the previous graph. The key question then is how to choose the probabilities such that nodes with large degree will have small degree afterwards: If all nodes have the same degree d, and each node decides independently with probability $p = 1/d^{1/4}$ whether to stay in the set, then the expected degree is $d^{3/4}$. This achieves that the maximum degree (or, correspondingly, size) decreases from Δ to Δ^c with high probability, and furthermore the probability that a node *and all* of its neighbors leave the set is rather low. Clearly, when nodes have different degrees, they will use different probabilities, so the above reasoning does not work. As we will see however, if the degrees of neighboring nodes do not differ too much, then one can still obtain a similarly fast decreasing maximum degree (with high probability).

Algorithm 9: RANDSTEP

Input: A growth-bounded graph $G = (V, E)$.
Output: A 2-ruling set $T \subseteq V$ of G.

1 $\mathcal{R} := \emptyset$
2 $\mathcal{S} := \{u \in V \mid d_u^2 < d_v \text{ for some } v \in N(u)\}$
3 $\mathcal{U} := N(\mathcal{S}) \backslash \mathcal{S}$
4 $\mathcal{B} := V \backslash (\mathcal{U} \cup \mathcal{S})$ ($\mathcal{B}$ is the set of black nodes)
5 Each $u \in \mathcal{B}$ independently joins $\mathcal{R}$ with probability $p = \frac{1}{d_u^{1/4}}$. ($\mathcal{R}$ is the set of red nodes)
6 $\mathcal{G} := \mathcal{B} \backslash (N(\mathcal{R}) \cup N(\mathcal{U}))$ ($\mathcal{G}$ is the set of green nodes)
7 **return** $T := \mathcal{S} \cup \mathcal{R} \cup \mathcal{G}$

Algorithm RANDSTEP works as follows (see Figure 11.1): First, nodes which have a neighbor with size much larger than their own know that they have a relatively low size, so they can simply stay in the graph for the next round (this is the set $\mathcal{S}$ in the algorithm). Additionally, their neighbors (set $\mathcal{U}$ in the algorithm) can safely leave the graph because they are sure to have a neighbor who stays. All other

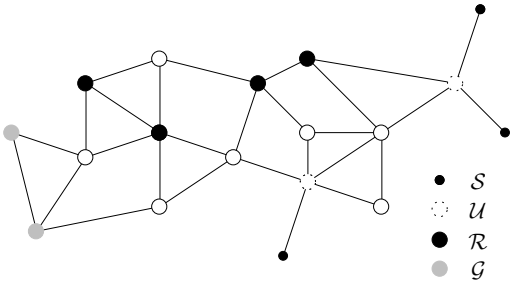

Figure 11.1: *Illustration of Algorithm* RANDSTEP. *The sets $\mathcal{S}$, $\mathcal{R}$ and $\mathcal{G}$, which are drawn as filled disks, remain in the graph. Their union $\mathcal{S} \cup \mathcal{R} \cup \mathcal{G}$ is a 2-ruling set of G.*

nodes then have only neighbors with similar sizes. We refer to these nodes as black nodes and denote them by the set $\mathcal{B}$. A black node u becomes red (and stays in the graph) independently with probability $p = 1/d_u^{1/4}$. The respective set is called $\mathcal{R}$ in Algorithm RANDSTEP. Finally, with low probability, there may be black nodes for which no node within distance two stays in the graph. In order to guarantee the 2-ruling property (in contrast to merely knowing a high probability bound for it), we add these nodes to the set of green nodes $\mathcal{G}$ which also stays in the graph[3].

For convenience, the algorithm is formulated in a global fashion, but a distributed version, where each node can execute these steps by communicating only with its direct neighbors, is immediate.

Analysis

In the following, we prove that with high probability, iterating Algorithm RANDSTEP for $O(\log \log n)$ times yields a $O(\log \log n)$-ruling set whose induced subgraph has a maximum degree of $O(\log^5 n)$.

First, we examine the ruling property of Algorithm RANDSTEP when applied repeatedly.

Lemma 11.2. *After iterating Algorithm* RANDSTEP *t times, the ob-*

[3]For ensuring the 2-ruling property, it would suffice to define $\mathcal{G} := \mathcal{B} \backslash (N_2(\mathcal{R}) \cup N(\mathcal{U}))$, but replacing $N_2(\mathcal{R})$ by $N(\mathcal{R})$ simplifies the analysis.

tained set T_t is a $2t$-ruling set.

Proof. Consider an iteration from T_i to T_{i+1}. Note that after every execution i of Algorithm RANDSTEP, the sets $\mathcal{S}$, $\mathcal{U}$ and $\mathcal{B}$ form a partition of T_i. Any node in $\mathcal{S}$ stays in the set. Any node in $\mathcal{U}$ is dominated by some node in $\mathcal{S}$. $\mathcal{R} \cup \mathcal{G}$ is by construction a dominating set of $\mathcal{B} \backslash N(\mathcal{U})$. Thus, the only nodes in T_i that may not be dominated by $\mathcal{S} \cup \mathcal{R} \cup \mathcal{G}$ are those in $N(\mathcal{U}) \backslash \mathcal{U}$. These nodes by definition have a 2-hop neighbor in $\mathcal{S}$. Thus, after each iteration, $\mathcal{S} \cup \mathcal{R} \cup \mathcal{G}$ is a 2-ruling set of T_i. The claim now follows by induction over i (as initially, $T_0 = V$ is a 0-ruling set). □

For the further analysis, we consider only one execution of Algorithm RANDSTEP on the graph $G = (V, E)$. During the exposition and the analysis of the algorithms, the sizes and neighbors of a node are to be understood with respect to the current graph. In the analysis we choose the constants quite arbitrarily to yield simple terms. A more careful selection might lead to smaller constant factors in the running time.

The first lemma states that the sizes of neighboring nodes in $\mathcal{B}$ do not differ too much.

Lemma 11.3. *For any $u \in \mathcal{B}$, and for each $v \in N(u)$, we have*

$$(d_v)^{1/2} \leq d_u \leq (d_v)^2.$$

Proof. By the lemma's assumption we have $u \notin \mathcal{S}$, and thus $d_u^2 \geq \max_{w \in N(u)} d_w \geq d_v$, which implies the left inequality. From $u \in \mathcal{B}$ we further have $u \notin \mathcal{U}$ and hence $v \notin \mathcal{S}$. So $d_v^2 \geq \max_{w \in N(v)} d_w \geq d_u$ and thus the right inequality follows. □

Lemma 11.4. *The probability that a node $u \in \mathcal{B}$ with $d := d_u \geq k^2 \ln^2 n$ becomes green is at most $\frac{1}{n^k}$, for $k > 1$.*

Proof. Let $u \in \mathcal{B}$ be a node with the property that $N(u)$ consists of only black nodes. Only such nodes may potentially become green. If, after the algorithm's execution, one of the nodes in $N(u)$ becomes red then u does not become green.

As the $d - 1$ neighbors are all black, their sizes are all at most d^2 by Lemma 11.3. Thus, each of them becomes red with probability

at least $d^{-1/2}$. The probability that neither u nor any of its $d-1$ neighbors become red is therefore at most

$$\left(1-d^{-1/2}\right)^d \leq e^{-d^{1/2}},$$

using the basic inequality $1-x \leq e^{-x}$. So for $d \geq k^2 \ln^2 n$, we have $P[u \text{ becomes green}] \leq \frac{1}{n^k}$. □

As we show in the following Lemmas, as long as the maximum degree Δ of G is at least $\Omega(\log^5 n)$, one iteration of RANDSTEP almost surely decreases $\overline{\Delta}$ of the graph to $2\overline{\Delta}^{7/8}$ or less.

Lemma 11.5. *For any $k > 1$, the probability that a black node $u \in \mathcal{B}$ with $d \geq 9k^2 \ln^2 n$ has more than $2d^{7/8}$ red neighbors (including itself) is at most $\frac{1}{n^k}$.*

Proof. Recall that each black node v becomes red with probability $\frac{1}{d_v^{1/4}}$, independent of the choices of its neighbors.

Let X be the number of red neighbors of u, plus one if u is red, i.e., $X := |N(u) \cap \mathcal{R}|$ (this possibly includes u itself). X is a random variable which is the number of successes in d independent *Poisson trials* (see e.g. [58]), where a "success" means that some neighbor of u (or u itself) joins $\mathcal{R}$. Each of these trials (of joining $\mathcal{R}$) may have a different success probability, but by Lemma 11.3 all neighbors of u (and u itself) have size at least $d^{1/2}$, so each neighbor of u (and u too) joins with probability at most $\frac{1}{d^{1/8}}$. Clearly, $E[X] \leq d \cdot \frac{1}{d^{1/8}} = d^{7/8}$. Using the Chernoff bound [16]

$$P[X \geq (1+\delta)E[X]] \leq e^{-E[X]\delta^2/3} \quad \text{for} \quad 0 < \delta \leq 1,$$

with $\delta = 1$ and $d \geq 9k^2 \ln^2 n$, we obtain

$$P\left[X \geq 2d^{7/8}\right] \leq e^{-\frac{1}{3}d^{7/8}} \leq e^{-(k \ln n)} \leq \frac{1}{n^k}.$$

□

Lemma 11.6. *The probability that a node $u \in \mathcal{B}$ with $d := d_u \geq 9k^4 \ln^4 n$ has more than $2d^{7/8}$ neighbors (including itself) in $\mathcal{R} \cup \mathcal{G}$ is at most $\frac{2}{n^{k-1}}$, $k > 1$.*

Proof. Let A be the event that u has more than $2d^{7/8}$ red neighbors, and let B be the event that u has any green neighbor. By Lemma 11.3, all neighbors of u have at least size $3k^2 \ln^2 n$, so by Lemma 11.4 every neighbor of u becomes green with probability at most $\frac{1}{n^k}$. From the union bound[4], it follows that the probability of u having any green neighbor is at most $\frac{1}{n^{k-1}}$, because u can have at most n neighbors.

Using this fact and Lemma 11.5 for the second inequality, we have

$$P[A \cup B] \leq P[A] + P[B] \leq \frac{1}{n^k} + \frac{1}{n^{k-1}} \leq \frac{2}{n^{k-1}}.$$

□

Lemma 11.7. *Let Δ be the maximum degree of the subgraph induced by T_i after $i \geq 0$ iterations of Algorithm* RANDSTEP. *Assume that $\overline{\Delta} \geq 6k^5 \ln^5 n$, $k > 2$ (recall $\overline{\Delta} = \Delta + 1$). After one more iteration, the maximum size of the subgraph induced by T_{i+1} is at most $2\overline{\Delta}^{7/8}$ with probability at least $1 - \frac{2}{n^{k-2}}$.*

Proof. Consider any node $u \in T_i$ with $d_u \geq 2\overline{\Delta}^{7/8}$. This node will only be in T_{i+1} after the next iteration if it is in $\mathcal{S} \cup \mathcal{R} \cup \mathcal{G}$. However, clearly u cannot be in $\mathcal{S}$ (because otherwise there would exist a node v with $d_v > d_u^2 \geq 4\overline{\Delta}^{14/8} \geq \overline{\Delta}$, a contradiction.). So u is either in $\mathcal{R}$ or in $\mathcal{G}$. Moreover, by construction any node in $\mathcal{B}$ has no neighbors in $\mathcal{S}$, so its size is the number of neighbors in $\mathcal{R} \cup \mathcal{G}$. As $d_u \geq 2\overline{\Delta}^{7/8} \geq 2(6k^5 \ln^5 n)^{7/8} \geq 9k^4 \ln^4 n$, Lemma 11.6 can be applied, so the size d'_u of u after the iteration satisfies $d'_u \leq 2\overline{\Delta}^{7/8}$ with probability at least $1 - \frac{2}{n^{k-1}}$. Hence the probability that any of the nodes with size $d \geq 6k^5 \ln^5 n$ has a size $\geq 2\overline{\Delta}^{7/8}$ is at most $\frac{2}{n^{k-2}}$, again using the union bound. □

The above bounds together yield a high-probability bound for the number of RANDSTEP iterations required to reduce the maximum degree to polylogarithmic size. Note that this result holds for arbitrary graphs, not only for growth-bounded graphs.

[4]The *union bound* upper bounds the probability that any of the events $E_1, E_2, \ldots, E_n$ occurs:
$P\left[\bigcup_{i=1}^n E_i\right] \leq \sum_{i=1}^n P[E_i]$, even if these events are not independent (see e.g. [58]).

Theorem 11.8. *For any constant $k > 3$, and some suitable constant $c = c(k)$, when* RANDSTEP *is run on any graph $G = (V, E)$ with maximum degree Δ, then with probability at least $1 - 2c/n^{k-3}$ the maximum degree in the subgraph of G induced by T_i is at most $6k^5 \ln^5 n$ after no more than $c \ln \ln \overline{\Delta}$ iterations.*

Proof. We call an iteration (of RANDSTEP) *successful* if the current maximum size of any node in the subgraph induced by T_i is reduced from $\overline{\Delta}$ to $2\overline{\Delta}^{7/8}$. Thus, after m successful iterations, the degree is at most

$$\begin{aligned} 2^{1+7/8+(7/8)^2+\ldots+(7/8)^{m-1}} \cdot \overline{\Delta}^{(7/8)^m} &= 2^8 2^{1-(7/8)^m} \overline{\Delta}^{(7/8)^m} \\ &\leq 512 \overline{\Delta}^{(7/8)^m}. \end{aligned}$$

In order to reduce the degree to below $6k^5 \ln^5 n$, m must satisfy

$$512 \overline{\Delta}^{(7/8)^m} \leq 6k^5 \ln^5 n,$$

which holds for $m \geq c \ln \ln \overline{\Delta}$ for some constant $c = c(k)$.

By Lemma 11.7, each iteration is successful with probability at least $1 - \frac{2}{n^{k-2}}$ as long as the degree is at least $6k^5 \ln^5 n$. Since we require at most $c \ln \ln \overline{\Delta}$ successes and each iteration succeeds with probability at least $1 - \frac{2}{n^{k-2}}$, as long as the maximum degree is big enough (Lemma 11.7), the probability that *all* rounds are successful is high. Formally, let A_i be the event that round i is not successful. There are at most $c \ln \ln \overline{\Delta}$ iterations that could potentially fail. Hence, the probability that at least one of these iterations fails is at most

$$P\left[\bigcup A_i\right] \leq \sum_i P\left[A_i\right] \leq \frac{2c \ln \ln \overline{\Delta}}{n^{k-2}} \leq \frac{2c}{n^{k-3}}.$$

By taking into account that $\overline{\Delta}$ might reach the threshold $6k^5 \ln^5 n$ earlier in the algorithm's execution, the probability that it ends within $c \ln \ln \overline{\Delta}$ iterations only grows. Thus, after $c \ln \ln \overline{\Delta}$ iterations, $\overline{\Delta} \leq 6k^5 \ln^5 n$ and hence the maximum degree of the graph is at most $6k^5 \ln^5 n$ with high probability. □

11.4.2 Phase 2: A Deterministic Algorithm for an $O(\log \log n)$-Ruling Set

As the maximum degree of the remaining graph becomes smaller, RANDSTEP becomes less effective in reducing it. This is because a node u in $\mathcal{B}$ with small d_u will only be removed from the graph with a small probability (and its neighbors, too). On the other hand, the deterministic algorithm SPARSIFY from [46] (called Algorithm 1 in that paper), which computes a MIS in growth-bounded graphs in $O(\log \Delta \log^* n)$ rounds, guarantees to halve the maximum degree of a growth-bounded graph in a constant number of steps. Thus, the latter algorithm is slower than ours while Δ is large, but is faster than RANDSTEP once Δ is only polylogarithmic. The algorithm

Algorithm 10: SPARSIFYSTEP

Input: A growth-bounded graph $G = (V, E)$.
Output: A 2-ruling set $T \subseteq V$ of G

1 Send an invitation to one arbitrarily chosen neighbor.
2 Accept one invitation, if any was received.
3 $E' := \big\{(u, v) \in E \,\big|\, u$ accepts the invitation of v or vice versa$\big\}$
4 Compute a MIS T on the edge-induced subgraph $\overline{G} := G(V, E')$.
5 **return** T

SPARSIFY is roughly described as follows: In each iteration, called SPARSIFYSTEP here, a subgraph $\overline{G}$ of G is selected such that every node in $\overline{G}$ has degree 1 or 2, and each node of G has a neighbor in $\overline{G}$. Due to the bounded degree of $\overline{G}$, a MIS of $\overline{G}$ can be computed in time $O(\log^* n)$, using the algorithm from [18]. Only nodes in this MIS stay in the graph for the next iteration. This subset is a 2-ruling set of the nodes in G.

It can be seen from the proof of Lemma 5 in [46] that this algorithm halves the degree of the graph after a constant number of iterations. Furthermore, inspection of this proof also shows that adding interleaved iterations of RANDSTEP does not harm the analysis. Hence:

Lemma 11.9. *For a growth-bounded graph,* SPARSIFY *reduces the maximum degree of the graph from Δ to $\Delta/2$ in $h = O(1)$ iterations.*

Each iteration needs $O(\log^* n)$ rounds, and all messages are of size $O(\log n)$ bits.

11.4.3 Combining Phase 1 and Phase 2: The Independent Ruling-Set Algorithm

The key observation for obtaining Algorithm RULINGSET which is fast in both of these phases is that the two algorithms can be "interleaved": after executing one call of RANDSTEP, we execute one call of SPARSIFYSTEP. This is possible because for both algorithms, one iteration takes a growth-bounded graph as input, and returns as output a subset of its nodes which is a 2-ruling set of the input graph. Hence, after t iterations of the combined algorithm, the remaining

Algorithm 11: RULINGSET

Input: A growth-bounded graph $G = (V, E)$
Output: An independent ruling-set I of G

```
1 I := V
2 while I is not independent do
3     I := RandStep(G[I])
4     if I independent then exit while-loop
5     I := SparsifyStep(G[I])
6 end
7 return I
```

set of nodes is a $4t$-ruling set of the original graph. Furthermore, Theorem 11.8 still holds for the combined algorithm, as the inserted iterations of SPARSIFYSTEP never increase the degree of any node. Once the degree of the remaining subgraph is at most $\Delta \leq 6k^5 \ln^5 n$, by Lemma 11.9 the deterministic steps guarantee that the degree is quickly decreased to zero (i.e., all remaining nodes are independent) within $O(\log(6k^5 \ln^5 n)) = O(\log \log n)$ iterations of the combined algorithm, as also the iterations of the RANDSTEP algorithm never increase any node's degree.

Of course, interleaving one single step of the deterministic algorithm (and not the required h steps to guarantee the bisection of the degree) deteriorates the constant factor of the running time, but allows on the other hand the execution of the algorithm without knowledge

of h. Summarizing, the following theorem holds.

Theorem 11.10. *For any $k > 1$ and any growth-bounded graph G, Algorithm* RULINGSET *computes an $O(\log\log n)$-ruling independent set of G in $O(\log\log n \log^* n)$ rounds of synchronous distributed computation, with probability at least $1 - O(1/n^k)$. All messages are of size $O(\log n)$ bits.*

11.4.4 Phase 3: Obtaining the Maximal Independent Set

At this point the $O(\log\log n)$-ruling independent set computed by Algorithm RULINGSET is *condensed* to yield a maximal independent set. We invoke the condensing algorithm from [46] (Algorithm 2), which extends any t-ruling independent set to a MIS in $O(t \cdot \log^* n)$ rounds in growth-bounded graphs, using only messages of $O(\log n)$ bits. Thus, we get a MIS in $O(\log\log n \log^* n)$ rounds. For the sake of completeness we describe shortly how this algorithm works. It is, in turn, split into two stages: the first stage condenses the existing ruling set until the resulting set is 3-ruling. This is achieved by letting every active node (i.e., a node in current independent set) compute an independent set in its 4-neighborhood under the constraint that every node in the 3-neighborhood is dominated by an active node. The resulting new set might be no longer independent, but the growth-bounded property of the graph permits to efficiently compute an independent set out of this denser set of active nodes. This process is repeated until the final set is 3-ruling and takes $O(\log\log n \log^* n)$ rounds.

For the final stage, i.e. the task of producing the desired MIS from the 3-ruling set, we define a clustergraph that enables the local completion of the independent set achieved so far. A coloring of the clustergraph defines an order of precedence on the active nodes that, in turn, ensures that the final set is independent. Again, the efficiency of the algorithm relies on the bounded growth property of the original graph, which leads to the final MIS in $O(\log^* n)$ rounds.

Combined, Theorem 2 and Lemma 8 from [46] imply the following:

Theorem 11.11. *For a growth-bounded graph G, Phase 3 transforms a $O(\log\log n)$-ruling independent set of G into a MIS of G*

in $O(\log\log n \log^* n)$ rounds of synchronous computation. All messages are of size $O(\log n)$ bits.

11.4.5 Removing Global Coordination

This section deals with issues that arise when the algorithms which we formulated in a global way are to be transformed into local distributed algorithms. We assume these techniques to be well-known, but include some explanations for the sake of completeness.

In the descriptions of our algorithms so far, we have used global criteria at some stages. For example, in Algorithm 11 (RULINGSET) we sequentially call two different distributed algorithms. The call to RANDSTEP is easy to implement locally, because its execution requires the same (constant) number of rounds at each node. The call to SPARSIFYSTEP, however, is more subtle: As a part of its execution, a MIS is computed with the algorithm from [18], whose global termination time is not available to the nodes. The third global criteria used in Algorithm RULINGSET is the termination of the while loop (which determines when Phase 3 of our algorithm is started). If this condition had to be checked globally as stated, then the running time of the algorithm would be at least linear in the network diameter. Fortunately, this is not required: One can replace the global termination condition by a local termination condition for each node, as follows: Each node u terminates Algorithm RULINGSET as soon as it has either joined the independent set I or has been removed from G without joining I (in both cases, it is determined whether node u will be part of the independent set when Algorithm RULINGSET terminates globally). Thus, some nodes may terminate Algorithm RULINGSET earlier than others.

If fact, all the global termination conditions we have just mentioned have the following two crucial properties: There is a local termination condition for each node, such that the global termination condition is true if and only if all local termination conditions are true. Furthermore, no local termination condition can become false again once it has become true in some round.

In the following, we describe how an algorithm which is split into several phases, where the termination condition of each phase has this structure, can be implemented without global coordination. The idea is to delay the execution of nodes which are ahead of their neighbors,

thus locally maintaining synchrony. More precisely, a node u which has completed Phase i starts Phase $i+1$ only when all its neighbors have also completed Phase i. If some neighbor of node u has not yet completed Phase i, node u enters a "waiting" state and sends a "pause $i+1$" message to all those neighbors which have already completed Phase i. Upon receiving a "pause $i+1$" message, a node forwards it to all neighbors which have not yet completed Phase $i+1$. Then, if it has already completed Phase i, it enters a "waiting" state itself. Otherwise, it continues its computation, and enters the "waiting" state only after completing Phase i.

Whenever a node u has completed Phase i, it informs all its neighbors. This may allow some nodes (those which now have only neighbors that have completed Phase i) to start with Phase $i+1$, thus they in turn inform all neighbors to which they have sent a "pause $i+1$" message with a "continue $i+1$" message. A node in waiting state in Phase $i+1$ continues its execution as soon as it has received a "continue $i+1$" message from all neighbors which have previously sent a "pause $i+1$" message, and in turn sends a "continue $i+1$" message to all neighbors waiting for it.

It should be clear that by this procedure, the execution is equivalent to one in which the beginning of a new phase is globally coordinated. Note that the mechanism just described is related to the synchronizers from [4], viewing the different phases of the algorithm as rounds in the synchronized model which must be ensured to be in synchrony in the asynchronous setting.

Next, we argue that the asymptotic running time of the algorithm is not increased by replacing the global termination conditions by local conditions. To that end, consider any *critical* node, i.e., a node whose computation has advanced the least. Such a node does not need to wait for any neighbor. If it is in the waiting state when it becomes critical, then it will receive a "continue" message in the next round. Thus, any critical node will continue its execution at most one round after becoming critical. Hence it follows that the worst-case time for completing each phase is at most doubled, and hence the asymptotic running time of the algorithm is not increased by replacing the global termination conditions by local conditions.

In addition, note that using the α-synchronizer of [4], our algorithm also terminates in $O(\log\log n \log^* n)$ time in an asynchronous setting, at the cost of a somewhat increased message complexity.

Chapter 12

Connected Dominating Sets

12.1 Motivation

In contrast to wired networks, where usually a dedicated *backbone* infrastructure with high-throughput capabilities is available for long-distance routing, *ad hoc* networks do not have any *a priori* means to manage the routing and scheduling of messages. Another specialty of wireless ad hoc networks is mobility of the nodes, which may continuously cause changes in the topology.

In the absence of an organized routing scheme, simple flooding (i.e., the first time a message is received from any neighbor, it is forwarded to all other neighbors) could be used to transmit messages. However, this is very wasteful in terms of energy and causes interference problems if many nodes transmit a message at the same time. To organize routing more cleverly, a *virtual backbone* can be computed, i.e., a subset of the nodes which participate in multi-hop routing. If messages are then forwarded by flooding them only within this virtual backbone, the energy savings are significant.

When modeling the network as a graph, the most widely used concept for defining a backbone is the *connected dominating set* (CDS). The energy savings are higher if the number of nodes in the CDS is small. However, computing a *minimum* CDS (MCDS) is NP-hard

even on unit disk graphs [17], and would require global information[1]. Considering the highly dynamic nature of ad hoc networks, it is important that the CDS can be computed locally within a short time; a linear running time in the diameter of the network (as required to obtain global information) is clearly inappropriate. Moreover, a MCDS lacks some desirable properties which we demand from an efficient backbone. For instance, routing a message from a node v to node a w should not need many more intermediate hops than a shortest path in the original network. The maximum ratio over all node pairs between these two hop-distances is called the *stretch factor*. The stretch factor of a MCDS can be as bad as linear in the number of nodes (for instance, consider a ring network), and one would like to prevent this effect. Another desirable property is that nodes have constant degree in the CDS-induced graph, which might help to address interference issues. For these reasons, a natural trade-off is to find a CDS with only near-minimum size but fulfilling the aforementioned properties.

12.2 Related Work

The concept of a *virtual backbone* (in analogy to backbones in wired networks) was introduced in [20]. Since then, the construction of small connected dominating sets in unit disk graphs (UDGs) has been intensively studied. A recent overview can be found in [11]. For the centralized setting with given coordinates of the nodes (which are embedded in the Euclidean plane), a *polynomial-time approximation scheme* (PTAS) was proposed in [15]. The approach of [19] (as well as our own approach) yields a PTAS that does not require coordinate information about the nodes.

However, many of the early distributed algorithms either did not guarantee a good approximation ratio in the worst case, or had a linear running time (see [11]). The first approach achieving a constant approximation ratio in polylogarithmic time was [74]. Alzoubi et al. [1] were the first to provide an algorithm for computing a CDS with low stretch and low degree, which was named *well-connected* CDS in [74]. Recently, Czygrinow et al. [19] presented a distributed algorithm with running time $O(1/\varepsilon^6 \cdot \log(1/\varepsilon) \cdot \log^3 n)$ achieving the

[1] To see this, consider a ring topology: since any optimal solution must contain exactly all but two adjacent nodes, computing a minimum CDS in a ring is as hard as electing a leader.

approximation ratio $1 + \varepsilon$ for any $\varepsilon > 0$, but without proving any stretch or degree bounds. Related to the minimum connected dominating set problem is the minimum dominating set problem, in which the desired set need not be connected. For this problem, a centralized approximation scheme for UDGs is given in [41]. In UDGs, the minimum dominating set problem even admits a distributed approximation scheme [47].

In general graphs, the best known distributed algorithm for the MCDS problem has polylogarithmic running time and achieves an approximation ratio of $O(\log \Delta)$, where Δ is the maximum degree of the network [25].

12.3 Summary of Results

In this chapter we present a distributed approximation scheme for the problem of finding a minimum connected dominating set in the class of growth-bounded graphs. An important feature of our algorithm is that the only information required by the nodes is the set of their direct neighbors. Distance estimation between them or even coordinate information are not required. Our algorithm complies with the $\mathcal{LOCAL}$ model of computation, but not with the stricter $\mathcal{CONGEST}$ model.

The algorithm computes a well-connected $(1+\varepsilon)$-approximation of a minimum connected dominating set, for any $\varepsilon > 0$. This takes $O(T_{\mathsf{MIS}} + 1/\varepsilon^{O(1)} \cdot \log^* n)$ rounds of synchronous computation, where T_{MIS} is the number of rounds needed to compute a maximal independent set (MIS). For growth-bounded graphs, a deterministic distributed algorithm for computing a MIS in $O(\log^* n)$ time has recently been found [89]. Prior to this result, the fastest known MIS algorithm was randomized and required $O(\log \log n \cdot \log^* n)$ time [37].

Thus, we improve on the running time for computing a $(1+\varepsilon)$-approximate MCDS, while adding the guarantee that the computed CDS has constant stretch, constant degree, and therefore a linear number of edges. Moreover, a recent lower bound implies that a constant approximation of a MCDS in unit disk graphs requires $\Omega(\log^* n)$ time [52], showing that the running time of our algorithm is asymptotically optimal.

The algorithm we propose builds substantially on the approach

of Nieberg et al. [69] for computing a $(1+\varepsilon)$-approximate minimum dominating set (for growth-bounded graphs): In a nutshell, their solution partitions the graph into clusters of appropriate radius, computes an optimal DS on each of these, and takes the union of these sets to yield a DS for the graph. Employing the same idea, we cluster the graph and compute an optimal CDS for each cluster. However, we let the clusters overlap such that the union of the small CDS solutions forms a connected DS of the graph. We prove the approximation ratio of the CDS by adapting the proof from [69], which requires an additional non-trivial step, i.e., Lemma 12.7. In the proof of Lemma 12.7, ideas related to [19] are used. In addition, we prove the (well-)connectedness of the computed set, which has no equivalent in [69, 19]. In order to turn our centralized approximation scheme for CDS into a distributed algorithm, we follow the lines of [47].

This combination and modification of known techniques yields a distributed approximation scheme which runs substantially faster than the previously known solution [19], and additionally guarantees well-connectedness.

The results of this chapter were obtained in joint work with Elias Vicari [38], and also appeared in his Ph.D. thesis [95].

12.4 Preliminaries

We require a few definitions which are specific to this chapter: Two sets $S, T \subseteq V$ are called *2-separated*[2] if and only if $d(S,T) \geq 3$. For any two points p, q in the plane, we denote their Euclidean distance by $\|p-q\|_2$. For a graph $G = (V,E)$ and a subset $V' \subseteq V$, the *reduced* neighborhood $\Gamma_j(v, V')$ of a node $v \in V'$ is defined for all $j \geq 0$ as $N_j(v)$ on the graph induced by V'.

We define the function $\mathcal{C}(A)$, where $A \subseteq V$ is a subset of all nodes, as follows: $\mathcal{C}(A) \subseteq N(A)$ is a minimum connected set of nodes that dominates all nodes in A, under the condition that only nodes in $N(A)$ are used. It is crucial that this set may contain nodes from $V \backslash A$. To end this section, we introduce two properties of minimum connected dominating sets in growth-bounded graphs, which we use in our approach.

[2] We use the term 2-separation because any shortest path between S and T contains at least two nodes outside $S \cup T$.

Lemma 12.1. *For any growth-bounded graph $G = (V, E)$ from a class with bounding function f, there is a polynomial $p(r) \leq 3 \cdot f(r)$ such that for any $v \in V$, it holds $|\mathcal{C}(N_r(v))| \leq p(r), \forall r > 0$.*

Proof. Let $f(r)$ be the polynomial bounding function of the growth-bounded graph G. For any node $v \in V$, consider a maximal independent set I of $N_r(v)$ (for a fixed $r \geq 0$) and set $Q := I$. In the following, we extend Q to a connected dominating set of $N_r(v)$. Let $k = |I|$ be the number of components of $G[I]$. We proceed by induction over k. If $k = 1$, $G[Q]$ is connected and the claim follows. Otherwise, since I is also a dominating set of $N_r(v)$, we can find two connected components $A, B \subseteq Q$ such that $d(A, B) \leq 3$. By adding to Q the nodes of a shortest path between them, we decrease the number of components by at least one and we increase $|Q|$ by at most two. We proceed inductively until Q induces a connected graph. Since $k = |I|$, we get $|\mathcal{C}(N_r(v))| \leq |Q| \leq |I| + 2|I| \leq 3f(r)$. As this holds for every $r > 0$, the claim is proved. □

Lemma 12.2. *Let $G = (V, E)$ be a growth-bounded graph with bounding function f, and choose a $S \subseteq V$ that induces a connected subgraph. Then for any MIS M of $G[S]$, it holds: $|M| \leq f(1) \cdot |\mathcal{C}(S)|$.*

Proof. As $\mathcal{C}(S)$ dominates S, each node in M must have a neighbor in $\mathcal{C}(S)$ (or be in $\mathcal{C}(S)$ itself). But by definition, at most $f(1)$ nodes in M can have the same neighbor in $\mathcal{C}(S)$, so the claim follows. □

12.5 Finding a Small Connected Dominating Set

In the following, we describe a (sequential) procedure to construct for each $\varepsilon > 0$ a connected dominating set of size at most $(1 + \varepsilon)$ times the minimum. This procedure can be executed efficiently in a centralized way, and thus leads to a PTAS. Moreover, in Section 12.6, we show that the same procedure can be implemented efficiently in a distributed way, using the same technique as in [47].

The CDS is constructed by computing optimal connected dominating sets for small parts of the graph, and taking the union of these CDSs. We will construct the small CDSs such that their union leads to a connected set, as required. Each small CDS is an optimal solution

of a small cluster specified as follows: In the subgraph $G[V']$, where $V' \subseteq V$, choose any node $v \in V'$, and consider the r-neighborhood of v for increasing values of $r = 0, 1, 2, \ldots$ until we find a large enough r^* such that

$$\left|\mathsf{MaxIS}\big(N(\Gamma_{r^*}(v, V'))\backslash\Gamma_{r^*}(v, V')\big)\right| \leq \varepsilon \cdot \left|\mathsf{MaxIS}\big(\Gamma_{r^*}(v, V')\big)\right| \tag{12.1}$$

holds. We call this operation an *expansion* of v. Initially, we set $V' = V$. As we show in Lemma 12.4 below, r^* is bounded by a function in $O(1/\varepsilon \cdot \log(1/\varepsilon))$ depending solely on ε for any class of growth-bounded graphs.

Our algorithm for finding a CDS for G proceeds as follows: starting with an empty set D, it chooses any node $v_1 \in V$ of G, finds a corresponding r_1^* such that Inequality 12.1 holds, and adds the set $\mathcal{C}(\Gamma_{r_1^*+4}(v_1, V))$ to the current solution D. After that, it removes all nodes in $\Gamma_{r_1^*+2}(v_1, V)$ from the graph G and we denote the set of remaining nodes by V'. Note that here we do not remove all nodes that are dominated by the current solution D from the graph. This is an important difference to the approach in [69]. As we will show, this modification guarantees that the final solution will be a *connected* dominating set.

In the reduced graph, the algorithm chooses another node $v_2 \in V'$, considers growing neighborhoods of v_2, until a r_2^* satisfying Inequality 12.1 is found. Note that the bounding function f of the original graph is still valid for the reduced graph, because any set which is independent in the reduced graph is also independent in the original graph. Furthermore, recall that $\mathcal{C}(\Gamma_{r_2^*}(v_2, V'))$ and $\mathcal{C}(\Gamma_{r_2^*+4}(v_2, V'))$ may contain some nodes from the original graph G as *dominators* which are outside V' because they are already dominated. Then, $\mathcal{C}(\Gamma_{r_2^*+4}(v_2, V'))$ is added to the current solution D, and all nodes in $\Gamma_{r_2^*+2}(v_2, V')$ are removed from the graph, just as before. Then, this procedure is repeated until all nodes have been removed from the graph.

The algorithm is described formally in Algorithm 12. Since the set of remaining nodes should always be clear from the context, we omit the second argument of $\Gamma(\cdot, \cdot)$ in the rest of the chapter.

For proving the correctness of our algorithm, we need the following two Lemmas.

Lemma 12.3. *Consider any class of growth-bounded graphs with*

Algorithm 12: Computes a $\big(1 + O(\varepsilon)\big)$-approximate MCDS

Input: A growth-bounded graph $G = (V, E)$, $\varepsilon > 0$
Output: An $\big(1 + O(\varepsilon)\big)$-approximate MCDS of G

```
1  D := {}
2  { For analysis: i := 1 }
3  while V ≠ {} do
4      Choose any v ∈ V
5      r := 0
6      while |MaxIS(N(Γ_r(v))\Γ_r(v))| > ε · |MaxIS(Γ_r(v))| do
7          r := r + 1
8      end
9      D := D ∪ C(Γ_{r+4}(v))
10     V := V\Γ_{r+2}(v)
11     { For analysis: S_i := Γ_r(v); T_i := Γ_{r+4}(v); i := i + 1 }
12 end
13 return D
```

bounding function f, and a graph $G = (V, E)$ of this class. Then for any $r \geq 1$ and any $v \in V$ it holds:

$$\big|\mathsf{MaxIS}\big(N(\Gamma_r(v))\backslash\Gamma_r(v)\big)\big| \leq f(2) \cdot \big|\mathsf{MaxIS}\big(\Gamma_r(v)\backslash\Gamma_{r-1}(v)\big)\big|$$

Proof. Clearly, each node in $\mathsf{MaxIS}(N(\Gamma_r(v))\backslash\Gamma_r(v))$ has a neighbor in $\Gamma_r(v)$ in G. As $\mathsf{MaxIS}(\Gamma_r(v)\backslash\Gamma_{r-1}(v))$ is a dominating set of $\Gamma_r(v)\backslash\Gamma_{r-1}(v)$, each node in $\mathsf{MaxIS}(N(\Gamma_r(v))\backslash\Gamma_r(v))$ has a node in $\mathsf{MaxIS}(\Gamma_r(v)\backslash\Gamma_{r-1}(v))$ within distance at most two. However, the number of nodes in $\mathsf{MaxIS}(N(\Gamma_r(v)\backslash\Gamma_r(v))$ that lie within two hops of the same node in $\mathsf{MaxIS}(\Gamma_r(v)\backslash\Gamma_{r-1}(v))$ can be at most $f(2)$, as otherwise these nodes could not be mutually independent. □

Lemma 12.4. *Consider any class of growth-bounded graphs with bounding function f. Then, for any $\varepsilon > 0$, there is a $R^*_f(\varepsilon) = O\big(1/\varepsilon \cdot \log(1/\varepsilon)\big)$ such that for each graph G of this class, and each node v of G, it holds*

$$\big|\mathsf{MaxIS}\big(N(\Gamma_{r^*}(v))\backslash\Gamma_{r^*}(v)\big)\big| \leq \varepsilon \cdot \big|\mathsf{MaxIS}\big(\Gamma_{r^*}(v)\big)\big|$$

for some $r^* \leq R^*_f$.

Proof. Fix an $\varepsilon > 0$ and assume in contradiction that no such R_f^* exists. This implies that for arbitrarily large values r', there is a graph in the class such that for some node v, $|\mathsf{MaxIS}(N(\Gamma_{r'}(v))\backslash\Gamma_{r'}(v))| > \varepsilon \cdot |\mathsf{MaxIS}(\Gamma_{r'}(v))|$ holds for all $0 \leq r \leq r'$. Consider such a value $r' \geq 2$. From

$$\begin{aligned}\big|\mathsf{MaxIS}\big(N(\Gamma_{r'}(v))\backslash\Gamma'_r(v)\big)\big| &> \varepsilon \cdot \big|\mathsf{MaxIS}\big(\Gamma_{r'}(v)\big)\big| \\ &\geq \varepsilon \cdot \big|\mathsf{MaxIS}\big(\Gamma_{r'-2}(v)\big)\big|\end{aligned}$$

and Lemma 12.3, we have

$$\big|\mathsf{MaxIS}\big(\Gamma_{r'}(v)\backslash\Gamma_{r'-1}(v)\big)\big| > \bar{\varepsilon} \cdot \big|\mathsf{MaxIS}\big(\Gamma_{r'-2}(v)\big)\big|,$$

for $\bar{\varepsilon} = \varepsilon/f(2)$. Hence, for all r: $2 \leq r \leq r'$ we have

$$\begin{aligned}\big|\mathsf{MaxIS}\big(\Gamma_r(v)\big)\big| &\geq \big|\mathsf{MaxIS}\big(\Gamma_r(v)\backslash\Gamma_{r-1}(v)\big)\big| + \big|\mathsf{MaxIS}\big(\Gamma_{r-2}(v)\big)\big| \\ &> (1+\bar{\varepsilon}) \cdot \big|\mathsf{MaxIS}\big(\Gamma_{r-2}(v)\big)\big|.\end{aligned}$$

Assume for the moment that r' is an even number. Then we have

$$\begin{aligned}\big|\mathsf{MaxIS}\big(\Gamma_{r'}(v)\big)\big| &> (1+\bar{\varepsilon}) \cdot \big|\mathsf{MaxIS}\big(\Gamma_{r'-2}(v)\big)\big| \\ &> (1+\bar{\varepsilon})^2 \cdot \big|\mathsf{MaxIS}\big(\Gamma_{r'-4}(v)\big)\big| \\ &> \dots \\ &> (1+\bar{\varepsilon})^{\frac{r'}{2}} \cdot \big|\mathsf{MaxIS}\big(\Gamma_0(v)\big)\big| = (1+\bar{\varepsilon})^{\frac{r'}{2}}.\end{aligned}$$

Since $|\mathsf{MaxIS}(\Gamma_{r'}(v))|$ grows only polynomially in r', but the term $(1+\bar{\varepsilon})^{\frac{r'}{2}}$ grows exponentially in r' (note that $(1+\bar{\varepsilon})^{1/2} > 1$), the above inequality will be violated for some large enough r', which is a contradiction. If r' is odd, the same reasoning can be applied.
The claimed bound on R_f^* follows easily from the inequality $(1+\bar{\varepsilon})^{1/\bar{\varepsilon}} > e - 1$, for $\bar{\varepsilon}$ small enough. □

It is clear that Algorithm 12 terminates, and that D then contains a dominating set, because only dominated nodes are removed from the graph. Let $S_1, S_2, \dots, S_k$ and $T_1, T_2, \dots, T_k$ be the sets $\Gamma_r(v_i)$ and $\Gamma_{r+4}(v_i)$ respectively as chosen in each iteration of the outer while-loop of Algorithm 12. We now show that the computed solution D, which consists of the union of the $\mathcal{C}(T_i)$, forms a *connected* subgraph of G.

Lemma 12.5. *The union* $D := \bigcup_{i=1}^{k} \mathcal{C}(T_i)$ *induces a* connected *subgraph of* G.

Proof. First, we show that any two nodes $a, b \in D$ with distance $d(a, b) = 2$ in G are part of the same connected component in (the subgraph induced by) D. To that end, consider any node $w \in V$ and its neighbors $N(w)$ (see the left part of Figure 12.1). Let s be the *first* node among $N(w)$ (including w) that is removed in line 10 of Algorithm 12. When s is removed in the i-th iteration, it holds that $s \in \Gamma_{r_i^*+2}(v_i)$ and so all nodes in $N(w)$ are in $T_i = \Gamma_{r_i^*+4}(v_i)$, whereby $v_1, v_2, \ldots$ represent the centers of the expansions. Hence they are dominated by $\mathcal{C}(\Gamma_{r_i^*+4}(v_i))$. Therefore, any pair of nodes in $D \cap N(w)$ is connected by a path of length at most $p(R^* + 4) + 1$ hops consisting only of nodes in D (recall p from Lemma 12.1).

Second, consider any pair $u, v \notin D$ of nodes adjacent in G (see the right part of Figure 12.1). We show that there must exist nodes $u' \in N(u) \cap D$ and $v' \in N(v) \cap D$ such that u' and v' are connected by a path of length at most $p(R^* + 4) + 1$ consisting only of nodes in D. To that end, assume w.l.o.g. that u is removed (line 10) before or at the same time as v. When u is removed in the j-th iteration (i.e., $u \in \Gamma_{r_j^*+2}(v_j)$), also v is dominated by $\mathcal{C}(\Gamma_{r_j^*+4}(v_j))$. The second claim follows. Combining this with the first claim, we have that any two nodes $a, b \in D$ with distance $d(a, b) = 3$ in G are connected by a path of length at most $3p(R^* + 4) + 1$ consisting only of nodes in D.

These two facts together imply that D induces a connected subgraph. Indeed, if D were disconnected, then the shortest path in G between two closest components of D would consist solely of nodes in $V \setminus D$. However, such a path cannot be of length two or three by the above facts, and not longer either, because then D would not dominate all the nodes. □

Now that we have shown that the set D computed by Algorithm 12 is a connected dominating set, we prove that its size is at most $1 + \varepsilon$ times larger than the optimum. To this end, we need two lemmas.

Lemma 12.6. *Let* $\varepsilon > 0$ *and* $r > 0$ *be such that the following inequality is satisfied:*

$$\left|\mathsf{MaxIS}\big(N(\Gamma_r(v)) \setminus \Gamma_r(v)\big)\right| \leq \varepsilon \cdot \left|\mathsf{MaxIS}\big(\Gamma_r(v)\big)\right|.$$

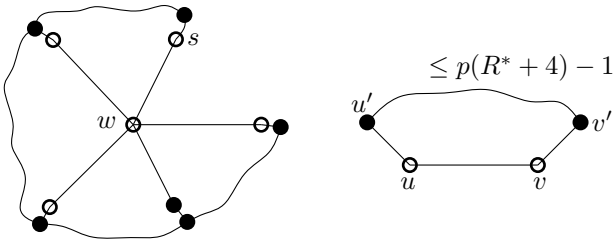

Figure 12.1: *Illustration of Lemma 12.5. The filled dots represent nodes that have already joined D. The other nodes are represented by empty dots.*

Then, for $\varepsilon' := \varepsilon \cdot (3f(4)+3) \cdot f(1)$, it follows:

$$\left|\mathcal{C}(\Gamma_{r+4}(v))\right| \leq (1+\varepsilon') \cdot \left|\mathcal{C}(\Gamma_r(v))\right|.$$

Proof. We show how to extend $\mathcal{C}(\Gamma_r(v))$ to a connected dominating set of the graph induced by $\Gamma_{r+4}(v)$ by adding only relatively few nodes so that the claim follows. Let M be a maximal independent set of the graph induced by $\Gamma_{r+1}(v)\backslash\Gamma_r(v)$. Any node in $\Gamma_{r+4}(v)$ lies within 3 hops of some node in $\Gamma_{r+1}(v)\backslash\Gamma_r(v)$, and thus within 4 hops of some node in M. Thus, all nodes in $\Gamma_{r+4}(v)$ are dominated if we add to our solution the set $\mathcal{C}(\Gamma_4(w))$ for each $w \in M$ (note that $|\mathcal{C}(\Gamma_4(w))| \leq p(4) \leq 3f(4)$). In order to connect these sets to $\mathcal{C}(\Gamma_r(v))$, we need to add at most 3 additional nodes for each $w \in M$. Thus in total, we obtain a connected dominating set of $\Gamma_{r+4}(v)$ of size at most

$$\begin{aligned}
&|\mathcal{C}(\Gamma_r(v))| + (3f(4)+3) \cdot |\mathsf{MaxIS}(\Gamma_{r+1}(v)\backslash\Gamma_r(v))| \\
&\leq |\mathcal{C}(\Gamma_r(v))| + (3f(4)+3) \cdot |\mathsf{MaxIS}(N(\Gamma_r(v))\backslash\Gamma_r(v))| \\
&\leq |\mathcal{C}(\Gamma_r(v))| + \varepsilon \cdot (3f(4)+3) \cdot |\mathsf{MaxIS}(\Gamma_r(v))| \\
&\leq |\mathcal{C}(\Gamma_r(v))| + \varepsilon \cdot (3f(4)+3) \cdot f(1) \cdot |\mathcal{C}(\Gamma_r(v))|,
\end{aligned}$$

using Lemma 12.2 for the last inequality. □

Let V^* be the set of nodes chosen as centers v for the growing neighborhoods in the algorithm. As for each $v \in V^*$, $\Gamma_{r+2}(v)$ is removed from the graph before choosing a new node, the collection $\{S_1, S_2, \ldots, S_k\}$ consists of 2-separated sets. We have the following lower bound for the size of an optimal CDS for G.

Lemma 12.7. *Let $S_1, S_2, \ldots, S_k$ be the collection of 2-separated sets in $G = (V, E)$ computed by Algorithm 12. Then,*

$$(1 + \varepsilon'') \cdot \big|\mathcal{C}(V)\big| \geq \big|\bigcup_{i=1}^{k} \mathcal{C}(S_i)\big|,$$

for $\varepsilon'' := 4f(1)\varepsilon$ and $\varepsilon \leq \frac{1}{4f(1)}$.

The latter assumption is without loss of generality: If $\varepsilon > \frac{1}{4f(1)}$, we can set $\varepsilon = \frac{1}{4f(1)}$, which only improves the approximation ratio.

Proof. Since the S_i are 2-separated, the sets $N(S_i)$ are pairwise disjoint, so the sets $\mathcal{C}(V) \cap N(S_i)$ are pairwise disjoint, too. Furthermore, as $\mathcal{C}(V)$ must dominate all nodes of G, including those in S_i, the set $\mathcal{C}(V) \cap N(S_i)$ must dominate all nodes in S_i. To complete the proof, we now show that $|\mathcal{C}(S_i)| \leq (1 + \varepsilon'') \cdot |\mathcal{C}(V) \cap N(S_i)|$ for all i, and thus, $|\mathcal{C}(V)| \geq \sum_{i=1}^{k} |\mathcal{C}(V) \cap N(S_i)| \geq \frac{1}{1+\varepsilon''} \cdot \sum_{i=1}^{k} |\mathcal{C}(S_i)|$.

To that end, we add some nodes to $\mathcal{C}(V) \cap N(S_i)$ in order to obtain a *connected* set which dominates S_i. Let x be the number of connected components in $\mathcal{C}(V) \cap N(S_i)$ and suppose that $x \geq 2$, otherwise $\mathcal{C}(V) \cap N(S_i)$ is connected and hence $|\mathcal{C}(S_i)| \leq (1 + \varepsilon'') \cdot |\mathcal{C}(V) \cap N(S_i)|$ is trivial. Then the individual connected components of $\mathcal{C}(V) \cap N(S_i)$ can be connected by adding at most $2x$ nodes (see the proof of Lemma 12.1). Note that each connected component of $\mathcal{C}(V) \cap N(S_i)$ must contain one node from $N(\Gamma_r(v)) \backslash \Gamma_r(v)$ to ensure the global connectivity of the solution. Thus, by choosing one such node for each connected component of $\mathcal{C}(V) \cap N(S_i)$, we obtain an independent set of size x. Therefore, we have $x \leq |\mathsf{MaxIS}(N(S_i) \backslash S_i)|$. By construction of the S_i, $|\mathsf{MaxIS}(N(S_i) \backslash S_i)| \leq \varepsilon \cdot |\mathsf{MaxIS}(S_i)|$, and by Lemma 12.2, $|\mathcal{C}(S_i)| \geq |\mathsf{MaxIS}(S_i)|/f(1)$. This shows that

$$|\mathcal{C}(S_i)| \leq |\mathcal{C}(V) \cap N(S_i)| + 2x \leq |\mathcal{C}(V) \cap N(S_i)| + 2\varepsilon f(1)|\mathcal{C}(S_i)|$$

and hence

$$|\mathcal{C}(V) \cap N(S_i)| \geq \big(1 - 2\varepsilon f(1)\big)|\mathcal{C}(S_i)|.$$

The claim now follows by choosing $\varepsilon'' = \frac{2f(1)\varepsilon}{1-2f(1)\varepsilon} \leq 4f(1) \cdot \varepsilon$, if $\varepsilon \leq \frac{1}{4f(1)}$. □

Theorem 12.8. *The set D computed by Algorithm 12 is a $\big(1+O(\varepsilon)\big)$-approximation for the connected dominating set problem.*

Proof. Let $\{S_1, S_2, \ldots, S_k\}$ and $\{T_1, T_2, \ldots, T_k\}$ be as defined in Algorithm 12. By Lemma 12.6, it holds that $|\mathcal{C}(T_i)| \leq (1+\varepsilon') \cdot |\mathcal{C}(S_i)|$ for all $i = 1, \ldots, k$, and $D = \bigcup_{i=1}^{k} \mathcal{C}(T_i)$. Hence we have

$$\begin{aligned} |D| &= \Big|\bigcup_{i=1}^{k} \mathcal{C}(T_i)\Big| \leq \sum_{i=1}^{k} \big|\mathcal{C}(T_i)\big| \leq (1+\varepsilon') \cdot \sum_{i=1}^{k} |\mathcal{C}(S_i)| \\ &= (1+\varepsilon') \cdot \Big|\bigcup_{i=1}^{k} \mathcal{C}(S_i)\Big| \leq (1+\varepsilon')(1+\varepsilon'') \cdot |\mathcal{C}(V)|, \end{aligned}$$

where the last inequality follows from Lemma 12.7. □

We now shortly discuss how Algorithm 12 can be implemented in a centralized fashion to obtain a PTAS. Most steps of the algorithm can be trivially computed efficiently. The crucial part is the computation of the *maximum* independent sets $\mathsf{MaxIS}(N(\Gamma_r(v)) \setminus \Gamma_r(v))$ and $\mathsf{MaxIS}(\Gamma_r(v))$, and of $\mathcal{C}(\Gamma_{r^*+4}(v))$. Note that r is bounded by the constant R^* for any fixed ε, so from the growth-bounded property we know that the size of the MaxIS is bounded by a constant. Thus, by enumerating all node subsets of cardinality at most $f(r)$, and selecting the largest of those which is both independent and maximal, a *maximum* independent set is found in polynomial time. Since the considered subsets have only constant size, independence and maximality of the subsets can be checked in constant time. The same arguments apply to the computation of $\mathcal{C}(\Gamma_{r^*+4}(v))$, due to Lemma 12.1. Hence, Algorithm 12 has polynomial time complexity for any fixed $\varepsilon > 0$, but exponential time complexity in $1/\varepsilon$.

12.6 A Distributed Approximation Scheme

The main goal of this chapter is to provide a fast distributed algorithm that computes a $(1+\varepsilon)$-approximation for the MCDS problem in growth-bounded graphs. The algorithm we describe here is an adaptation from [47], adjusted to computing a CDS instead of a DS.

A naive distributed implementation would be that in each round all nodes which have the highest ID within their $2R^* + 9$-hop neighborhood are expanded concurrently. However, this approach requires a linear number of rounds in the worst case, because there can be a linear waiting chain of nodes. The observation that every expansion affects only neighbors within a small radius leads to a more efficient algorithm: expansions of nodes with sufficient mutual distances can be scheduled concurrently. Roughly speaking, this can be achieved by computing a MIS of G, and then coloring this MIS with few colors such that two nodes with the same color are distant enough. The coloring is achieved using a *clustergraph* $\bar{G}$ of radius $c = c(\varepsilon)$ with centers $W \subseteq V$: $\bar{G} = (\bar{V}, \bar{E})$, where $\bar{V} := W$ and for every $u, v \in W$, we let $(u, v) \in \bar{E}$ if and only if $d_G(u, v) \leq c$. Note that if W is an independent set and G is growth-bounded, $\bar{G}$ has a maximum degree of $\Delta_{\bar{G}} = O(f(c))$. Hence using a MIS of G to construct a clustergraph $\bar{G}$ of radius c, $\bar{G}$ can be colored with $O(\Delta_{\bar{G}}^2)$ colors in $O(c \cdot \log^* n)$ time [53]. Note that a communication round in the clustergraph costs $O(c)$ rounds in the original graph. The coloring is then used to schedule the expansion of neighbors of MIS-nodes. We choose $c = 2R^* + 11$ for reasons that will become apparent in the proof of Lemma 12.10. A more detailed description is given in Algorithm 13.

Lemma 12.9. *Algorithm 13 terminates in $O(T_{\mathsf{MIS}} + 1/\varepsilon^{O(1)} \cdot \log^* n)$ time.*

Proof. Computing a MIS of G takes time T_{MIS}. Then, the clustergraph can be constructed in constant time, as its edges only span at most distance $2R^* + 11 = O(1/\varepsilon \cdot \log(1/\varepsilon))$. Furthermore, $\bar{G}$ can be colored with $O(\Delta_{\bar{G}}^2) = O\big(f^2(R^*)\big)$ colors in $O(R^* \cdot \log^* n)$ time using the algorithm of [53].

The outer for-loop is executed $O(\Delta_{\bar{G}}^2)$ times. Inside the for-loop, the number of different MaxIS that each node u (as in line 8) must compute is $2r^* = O(R^*) = O(1/\varepsilon \cdot \log(1/\varepsilon))$. For computing each MaxIS and MCDS for a neighborhood of radius r, u collects all information about $\Gamma_r(u)$ and then computes the set locally. As $r \leq R^*$, all steps in lines 8 to 12 can be executed in $O(1/\varepsilon \cdot \log(1/\varepsilon))$ time. □

Lemma 12.10. *The set D computed by Algorithm 13 is a $\big(1+O(\varepsilon)\big)$-approximate minimum connected dominating set.*

Algorithm 13: Computes a $\big(1+O(\varepsilon)\big)$-approximate MCDS distributively

Input: A growth-bounded graph G, $\varepsilon > 0$, R^* (according to Lemma 12.4)
Output: An $\big(1 + O(\varepsilon)\big)$-approximate MCDS of G

```
1  Compute a MIS I of G;
2  Construct the clustergraph Ḡ of I using radius 2R* + 11;
3  Color Ḡ with γ = O(Δ²_Ḡ) colors;
4  D := {};
5  for k := 1 to γ do
6      for every v ∈ I with color k do in parallel
7          if N(v) ∩ V ≠ {} then
8              For some u ∈ N(v) ∩ V, find the smallest r* such
               that
               |MaxIS(N(Γ_{r*}(v)) \ Γ_{r*}(v))| ≤ ε·|MaxIS(Γ_{r*}(v))|;
9              Compute C(Γ_{r*+4}(u));
10             Inform Γ_{r*+4}(u) about r* and C(Γ_{r*+4}(u));
11             D := D ∪ C(Γ_{r*+4}(u));
12             V := V \ Γ_{r*+2}(u);
13         end
14     end
15 end
16 return D
```

Proof. By construction, any two nodes that are concurrently used for an expansion have distance at least $2R^* + 9$, because they are respective neighbors of two MIS nodes of distance at least $2R^* + 11$. The radius used by either expansion is at most R^*, and since each expansion only involves the nodes within a radius of at most $R^* + 5$, all concurrent expansions would have the same result if they were executed sequentially. Therefore, there exists an execution of the sequential Algorithm 12 which computes the same set D as Algorithm 13. It follows that Algorithm 13 achieves the same approximation ratio as Algorithm 12. □

These two lemmas lead to our main theorem.

Theorem 12.11. *For any $\varepsilon > 0$, Algorithm 13 computes a $\big(1 + O(\varepsilon)\big)$-approximate minimum connected dominating set in $O(T_{\mathsf{MIS}}+$*

$1/\varepsilon^{O(1)} \cdot \log^* n)$ *time.*

12.7 Well-Connectedness

The connected dominating set computed by our Algorithm 12 is not only a $(1+\varepsilon)$-approximation of a minimum CDS, but has additional properties which are desirable for its usage as a backbone in a wireless network. Let $G' = (V', E')$ be the graph induced by the CDS of the (growth-bounded) graph $G = (V, E)$ computed by Algorithm 12. Then, the following statements hold for any $\varepsilon > 0$:

1. The backbone graph G' has maximum degree $O(1/\varepsilon^{O(1)})$, and therefore it has only $O(1/\varepsilon^{O(1)} \cdot |V'|)$ edges.

2. Using G' as a routing backbone guarantees stretch $O(1/\varepsilon^{O(1)})$.

We assume in the following that source s and destination d of a routing request are both members of the CDS. If this is not true for either or both of them, then we can easily choose a neighbor inside the CDS as a representative. This will add at most two hops to the routing path, so if the stretch is low for any pair s, d inside the CDS, the stretch of any pair s, d inside G is also low.

To make the second statement precise, define

$$\lambda := \max_{u,v \in V'} \frac{d_{G'}(u,v)}{d_G(u,v)}$$

as the *hop-stretch* of G'. Furthermore, if G is a UDG and if $D_G(u,v)$ denotes the geometric length of a shortest path in G, then the *geometric stretch* of G' is $\mu := \max_{u,v \in V'} \frac{D_{G'}(u,v)}{D_G(u,v)}$.

Lemma 12.12. *Let $\mathsf{CDS}_\mathcal{A}$ be the CDS computed by Algorithm 12. The subgraph G' of G induced by the nodes in $\mathsf{CDS}_\mathcal{A}$ has maximum degree $O(1/\varepsilon^{O(1)})$.*

Proof. First, note that each partial CDS $\mathcal{C}(T_i)$ computed by the algorithm covers a subgraph with diameter $O(R^*)$, so according to Lemma 12.1, $|\mathcal{C}(T_i)| \in O(f(R^*))$. Therefore the maximum degree in the graph induced by $\mathcal{C}(T_i)$ is also at most $O(f(R^*))$. Second,

each node u of G can only be contained in $O(f(R^*))$ many different $\mathcal{C}(T_i)$, because any expansion that leads to some $\mathcal{C}(T_j)$ containing u must have as its center a MIS-node in distance at most $R^* + 4 = O(1/\varepsilon \cdot \log(1/\varepsilon))$ from u (and there are only $O(f(R^*))$ MIS nodes within distance $O(R^*)$ of v). □

Lemma 12.13. *For any $\varepsilon > 0$, the hop-stretch γ of G' is $O(f(R^*)) = O(1/\varepsilon^{O(1)})$. Further, if G is a UDG, then the geometric stretch λ of G' is also $O(1/\varepsilon^{O(1)})$.*

Proof. Let D be the CDS computed by Algorithm 12. Consider any source s and destination d in V'. Let $\mathcal{P} = \langle p_1, p_2, \ldots, p_k \rangle$ be the sequence of nodes in a shortest path in G from $s = p_1$ to $d = p_k$. We define a new path $\mathcal{Q}$ going through intermediate nodes $q_1, q_2, \ldots, q_k$ as follows: $q_i := p_i$ if $p_i \in D$. Otherwise, let q_i be any node in $D \cap N(p_i)$ (such a node exists because D is dominating). Note that $q_i \in D$, $\forall i : 1 \leq i \leq k$. From the proof of Lemma 12.5, we can conclude that between any pair (q_i, q_{i+1}), $\forall i : 1 \leq i \leq k-1$, there is a path in D of length at most $3p(R^* + 4) + 1 = O(f(R^*))$. Hence there is a path $\mathcal{Q}$ of length $\leq k(3p(R^* + 4) + 1) = k \cdot O(f(R^*)) = k \cdot O(1/\varepsilon^{O(1)})$ from q_1 to q_k solely consisting of nodes in D.
For the geometric stretch in UDGs, note that in the path $\mathcal{R} := \langle t_1 = s, t_2, \ldots t_k = d \rangle$ of shortest geometric length, the outer two of any three consecutive nodes t_i, t_{i+1}, t_{i+2} must have distance at least 1: $\|t_i - t_{i+2}\|_2 \geq 1$, $\forall i \in \{1, \ldots, k-2\}$. So $\mathcal{R}$ with k hops has length at least $(k-1)/2$. On the other hand, the path with the fewest number of hops (at most k) has length at most k. Since we have shown just before that G' includes a path with hop-stretch $\gamma = O(1/\varepsilon^{O(1)})$ between any pair of nodes, it follows that the geometric stretch λ of G' is at most $2k \cdot \gamma/(k-1) \leq 4\gamma = O(1/\varepsilon^{O(1)})$, $\forall k \geq 2$. For $k = 1$, the path with fewest hops has length at most $\gamma = O(1/\varepsilon^{O(1)})$, which completes the claim. □

Summarizing, we have the following.

Theorem 12.14. *The CDS computed by Algorithm 12 / Algorithm 13 is well-connected.*

Chapter 13

Discussion

Concerning Part III, there seem to be no immediate questions that remain open: one year after the publication of our randomized algorithm for computing a MIS, an asymptotically optimal algorithm was discovered [89]. As for the MCDS problem, the running time our distributed approximation scheme was shown to be asymptotically optimal by later work [52]. Nevertheless, many open questions remain in a more general context. Especially intriguing is the role of randomness for computing a MIS in general graphs: While a simple randomized distributed algorithm requires only $O(\log n)$ time with high probability [54], the best known deterministic algorithm is much slower and a lot more complicated [73]. Is there a deterministic algorithm for the maximal independent set problem in general graphs with polylogarithmic running time?

Bibliography

[1] Khaled M. Alzoubi. Connected Dominating Set and its Induced Position-less Sparse Spanner For Mobile Ad Hoc Networks. In *8th IEEE International Symposium on Computers and Communications (ISCC)*, pages 209–216, 2003.

[2] Khaled M. Alzoubi, Peng-Jun Wan, and Ophir Frieder. Message-optimal connected dominating sets in mobile ad hoc networks. In *3rd ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc)*, pages 157–164. ACM Press, 2002.

[3] Takao Asano, Tetsuo Asano, Leonidas J. Guibas, John Hershberger, and Hiroshi Imai. Visibility of Disjoint Polygons. *Algorithmica*, 1(1):49–63, 1986.

[4] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985.

[5] Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network Decomposition and Locality in Distributed Computation. In *30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–369, 1989.

[6] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *29th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 206–219, 1988.

[7] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, Martin Vetterli, Olivier Couach, and Marc Parlange. SensorScope: Out-of-the-Box Environmental Monitoring. In *7th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 332–343. IEEE Computer Society, 2008.

[8] Amir M. Ben-Amram. What is a "pointer machine"? *ACM SIGACT News*, 26(2):88–95, 1995.

[9] Amit M. Bhosle and Teofilo F. Gonzalez. Algorithms for Single Link Failure Recovery and Related Problems. *Journal of Graph Algorithms and Applications*, 8(2):275–294, 2004.

[10] Davide Bilò, Luciano Gualà, and Guido Proietti. Finding all the best swap edges of a routing tree: a faster algorithm and an effectiveness analysis. Manuscript, 2008.

[11] Jeremy Blum, Min Ding, Andrew Thaeler, and Xiuzhen Cheng. *Handbook of Combinatorial Optimization*, chapter *Connected dominating set in sensor networks and MANETs*, pages 329–369. Kluwer Academic Publishers, 2004.

[12] Marc Bui, Franck Butelle, and Christian Lavault. A Distributed Algorithm for Constructing a Minimum Diameter Spanning Tree. *Journal of Parallel and Distributed Computing*, 64:571–577, 2004.

[13] Zack J. Butler, Peter I. Corke, Ronald A. Peterson, and Daniela Rus. Virtual Fences for Controlling Cows. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4429–4436. IEEE, 2004.

[14] Leizhen Cai and Derek G. Corneil. Tree Spanners. *SIAM Journal on Discrete Mathematics*, 8(3):359–387, 1995.

[15] Xiuzhen Cheng, Xiao Huang, Deying Li, Weili Wu, and Ding-Zhu Du. A polynomial-time approximation scheme for the minimum-connected dominating set in ad hoc wireless networks. *Networks*, 42(4):202–208, 2003.

[16] Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507, December 1952.

[17] Brent N. Clark, Charles J. Colbourn, and David S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86(1-3):165–177, 1990.

[18] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[19] Andrzej Czygrinow and Michał Hańćkowiak. Distributed approximation algorithms in unit-disk graphs. In *20th International Symposium on Distributed Computing (DISC)*, volume 4167 of *LNCS*, pages 385–398. Springer, 2006.

[20] Bevan Das, Raghupathy Sivakumar, and Vaduvur Bharghavan. Routing in ad-hoc networks using a virtual backbone. In *6th International Conference on Computer Communications and Networks (IC3N)*, pages 1–20, 1997.

[21] Shantanu Das, Beat Gfeller, and Peter Widmayer. Computing Best Swaps in Optimal Tree Spanners. In *19th International Symposium on Algorithms and Computation (ISAAC)*, volume 5369 of *LNCS*, pages 716–727. Springer, 2008.

[22] Aleksej Di Salvo and Guido Proietti. Swapping a Failing Edge of a Shortest Paths Tree by Minimizing the Average Stretch Factor. *Theoretical Computer Science*, 383(1):23–33, 2007.

[23] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, second edition, 2000.

[24] Brandon Dixon, Monika Rauch, and Robert E. Tarjan. Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[25] Devdatt P. Dubhashi, Alessandro Mei, Alessandro Panconesi, Jaikumar Radhakrishnan, and Aravind Srinivasan. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. *Journal of Computer and System Sciences*, 71(4):467–479, 2005.

[26] Yuval Emek and David Peleg. Approximating Minimum Max-Stretch spanning Trees on unweighted graphs. In *15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 261–270. Society for Industrial and Applied Mathematics, 2004.

[27] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[28] Paola Flocchini, Antonio M. Enriques, Linda Pagli, Giuseppe Prencipe, and Nicola Santoro. Point-of-failure Shortest-path Rerouting: Computing the Optimal Swap Edges Distributively. *IEICE Transactions on Information and Systems*, E89-D(2):700–708, 2006.

[29] Paola Flocchini, Antonio M. Enriquez, Linda Pagli, Giuseppe Prencipe, and Nicola Santoro. Distributed Computation of All Node Replacements of a Minimum Spanning Tree. In *13th International Conference on Parallel Processing (Euro-Par)*, pages 598–607, 2007.

[30] Paola Flocchini, Linda Pagli, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Computing All the Best Swap Edges Distributively. *Journal of Parallel and Distributed Computing*, 68(7):976–983, 2008.

[31] Pierre Fraigniaud and Cyril Gavoille. Routing in Trees. In *28th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of *LNCS*, pages 757–772. Springer, 2001.

[32] Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[33] Carlo Gaibisso, Guido Proietti, and Richard B. Tan. Optimal MST Maintenance for Transient Deletion of Every Node in Planar Graphs. In *9th Annual International Computing and Combinatorics Conference (COCOON)*, pages 404–414, 2003.

[34] Ece Gelal, Gentian Jakllari, Srikanth V. Krishnamurthy, and Neal E. Young. Topology Control to Simultaneously Achieve Near-Optimal Node Degree and Low Path Stretch in Ad hoc Networks. In *3rd Annual IEEE Communications Society on Sensor and Ad Hoc Communications and Networks (SECON)*, volume 2, pages 431–439, 2006.

[35] Beat Gfeller. Faster Swap Edge Computation in Minimum Diameter Spanning Trees. In *16th Annual European Symposium on Algorithms (ESA)*, volume 5193 of *LNCS*, pages 454–465. Springer, 2008.

[36] Beat Gfeller, Nicola Santoro, and Peter Widmayer. A Distributed Algorithm for Finding All Best Swap Edges of a Minimum Diameter Spanning Tree. *IEEE Transactions on Dependable and Secure Computing*, 2009. To appear.

[37] Beat Gfeller and Elias Vicari. A Randomized Distributed Algorithm for the Maximal Independent Set Problem in Growth-Bounded Graphs. In *26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–60, 2007.

[38] Beat Gfeller and Elias Vicari. A Faster Distributed Approximation Scheme for the Connected Dominating Set Problem for Growth-Bounded Graphs. *Ad Hoc & Sensor Wireless Networks*, 6(3–4):197–213, 2008.

[39] Martin Grötschel, Clyde L. Monma, and Mechthild Stoer. Design of Survivable Networks. *Handbooks in Operations Research and Management Science*, 7:617–672, 1995.

[40] Dov Harel and Robert E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[41] Harry B. Hunt III, Madhav V. Marathe, Venkatesh Radhakrishnan, S. S. Ravi, Daniel J. Rosenkrantz, and Richard Edwin Stearns. NC-approximation schemes for NP- and PSPACE-hard problems for geometric graphs. *Journal of Algorithms*, 26(2):238–274, 1998.

[42] Gianluca Iannaccone, Chen-Nee Chuah, Richard Mortier, Supratik Bhattacharyya, and Christophe Diot. Analysis of link failures in an IP backbone. In *2nd ACM SIGCOMM Internet Measurement Workshop (IMW)*, pages 237–242. ACM, 2002.

[43] Alon Itai and Michael Rodeh. The multi-tree approach to reliability in distributed networks. *Information and Computation*, 79(1):43–59, 1988.

[44] Hiro Ito, Kazuo Iwama, Yasuo Okabe, and Takuya Yoshihiro. Single Backup Table Schemes for Shortest-path Routing. *Theoretical Computer Science*, 333(3):347–353, 2005.

[45] Holger Junker, Mathias Stäger, Gerhard Tröster, Dimitri Blättler, and Olivier Salama. Wireless networks in context aware wearable systems. In *1st European Workshop on Wireless Sensor Networks (EWSN)*, volume 2920 of *LNCS*, pages 37–40. Springer, 2004.

[46] Fabian Kuhn, Thomas Moscibroda, Tim Nieberg, and Roger Wattenhofer. Fast Deterministic Distributed Maximal Independent Set Computation on Growth-Bounded Graphs. In *19th International Symposium on Distributed Computing (DISC)*, volume 3724 of *LNCS*, pages 273–287. Springer, 2005.

[47] Fabian Kuhn, Thomas Moscibroda, Tim Nieberg, and Roger Wattenhofer. Local Approximation Schemes for Ad Hoc and Sensor Networks. In *3rd ACM Joint Workshop on Foundations of Mobile Computing (DIALM-POMC)*, pages 97–103. ACM, 2005.

[48] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In *23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 300–309. ACM, 2004.

[49] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. On the Locality of Bounded Growth. In *24th ACM Symposium on the Principles of Distributed Computing (PODC)*, pages 60–68. ACM, 2005.

[50] Fabian Kuhn, Roger Wattenhofer, and Aaron Zollinger. Ad Hoc Networks Beyond Unit Disk Graphs. *Wireless Networks*, 14(5):715–729, 2008.

[51] Frank T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1992.

[52] Christoph Lenzen and Roger Wattenhofer. Leveraging Linial's Locality Limit. In *22nd International Symposium on Distributed Computing (DISC)*, volume 5218 of *LNCS*, pages 394 – 407. Springer, 2008.

[53] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

[54] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1055, 1986.

[55] K. Malik, A. K. Mittal, and S.K. Gupta. The k most vital arcs in the shortest path problem. *Operations Research Letters*, 8(4):223–227, 1989.

[56] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of Failures in an Operational IP Backbone Network. *IEEE/ACM Transactions on Networking*, 16(4):749–762, 2008.

[57] Alain Mayer, Moni Naor, and Larry Stockmeryer. Local computations on static and dynamic graphs. In *Third Israel Symposium on the Theory of Computing and Systems*, pages 268–278, Jan 1995.

[58] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, January 2005.

[59] Thomas Moscibroda and Roger Wattenhofer. Maximal independent sets in radio networks. In *24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 148–157. ACM Press, 2005.

[60] Yoram Moses and Sergio Rajsbaum. A layered analysis of consensus. *SIAM Journal on Computing*, 31(4):989–1021, 2002.

[61] Moni Naor. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM Journal on Discrete Mathematics*, 4(3):409–412, 1991.

[62] Moni Naor and Larry Stockmeyer. What can be computed locally? In *25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 184–193. ACM, 1993.

[63] Enrico Nardelli, Guido Proietti, and Peter Widmayer. Finding the Detour-Critical Edge of a Shortest Path between two Nodes. *Information Processing Letters*, 67(1):51–54, 1998.

[64] Enrico Nardelli, Guido Proietti, and Peter Widmayer. A faster computation of the most vital edge of a shortest path. *Information Processing Letters*, 79(2):81–85, 2001.

[65] Enrico Nardelli, Guido Proietti, and Peter Widmayer. Finding All the Best Swaps of a Minimum Diameter Spanning Tree Under Transient Edge Failures. *Journal of Graph Algorithms and Applications*, 5(5):39–57, 2001.

[66] Enrico Nardelli, Guido Proietti, and Peter Widmayer. Finding the Most Vital Node of a Shortest Path. *Theoretical Computer Science*, 296(1):167–177, 2003.

[67] Enrico Nardelli, Guido Proietti, and Peter Widmayer. Swapping a Failing Edge of a Single Source Shortest Paths Tree Is Good and Fast. *Algorithmica*, 35(1):56–74, 2003.

[68] Enrico Nardelli, Guido Proietti, and Peter Widmayer. Nearly Linear Time Minimum Spanning Tree Maintenance for Transient Node Failures. *Algorithmica*, 40(2):119–132, 2004.

[69] Tim Nieberg and Johann L. Hurink. A PTAS for the minimum dominating set problem in unit disk graphs. Memorandum 1732, University of Twente, Enschede, 2004.

[70] Tim Nieberg and Johann L. Hurink. Wireless Communication Graphs. In *DEST International Workshop on Signal Processing for Sensor Networks (ISSNIP)*, pages 367–372, 2004.

[71] Linda Pagli, Giuseppe Prencipe, and Tranos Zuva. Distributed Computation for Swapping a Failing Edge. In *6th International Workshop on Distributed Computing (IWDC)*, volume 3326 of *LNCS*, pages 28–39. Springer, 2004.

[72] Jacques Panchard, Seshagiri Rao, T.V. Prabhakar, H.S. Jamadagni, and Jean-Pierre Hubaux. COMMON-Sense Net: Improved Water Management for Resource-Poor Farmers via Sensor Networks. In *International Conference on*

Communication and Information Technologies and Development (ICTD), pages 22–33, 2006.

[73] Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *24th Annual ACM Symposium on Theory of Computing (STOC)*, pages 581–592. ACM Press, 1992.

[74] Srinivasan Parthasarathy and Rajiv Gandhi. Distributed algorithms for coloring and domination in wireless ad hoc networks. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 447–459, 2004.

[75] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[76] David Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[77] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. In *6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 77–85. ACM, 1987.

[78] David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36(3):510–530, 1989.

[79] Sriram V. Pemmaraju and Imran A. Pirwani. Energy conservation via domatic partitions. In *7th ACM international Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 143–154. ACM Press, 2006.

[80] Seth Pettie. Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time. In *Proceedings 16th International Symposium on Algorithms and Computation (ISAAC)*, volume 3827 of *LNCS*, pages 964–973. Springer, 2005.

[81] Guido Proietti. Dynamic Maintenance Versus Swapping: An Experimental Study on Shortest Paths Trees. In *4th International Workshop on Algorithm Engineering (WAE)*, volume 1982 of *LNCS*, pages 207–217. Springer, 2000.

[82] Nicola Santoro. *Design and Analysis of Distributed Algorithms*. Wiley Series on Parallel and Distributed Computing. Wiley, 2007.

[83] Nicola Santoro and Ramez Khatib. Labelling and Implicit Routing in Networks. *The Computer Journal*, 28(1):5–8, 1985.

[84] Nicola Santoro and Peter Widmayer. Time is not a healer. In *6th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 349 of *LNCS*, pages 304–313. Springer, 1989.

[85] Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theoretical Compututer Science*, 384(2-3):232–249, 2007.

[86] Baruch Schieber, Amotz Bar-Noy, and Samir Khuller. The complexity of finding most vital arcs and nodes. Technical Report UMIACS-TR-95-96, University of Maryland at College Park, 1995.

[87] Stefan Schmid and Roger Wattenhofer. Algorithmic Models for Sensor Networks. In *14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2006.

[88] Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility Results and Lower Bounds for Consensus under Link Failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.

[89] Johannes Schneider and Roger Wattenhofer. A Log-Star Distributed Maximal Independent Set Algorithm for Growth-Bounded Graphs. In *27th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 35–44. ACM, 2008.

[90] Bing Su, Qingchuan Xu, and Peng Xiao. Finding the anti-block vital edge of a shortest path between two nodes. *Journal of Combinatorial Optimization*, 16(2):173–181, August 2008.

[91] Robert E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[92] Robert E. Tarjan. Applications of Path Compression on Balanced Trees. *Journal of the ACM*, 26(4):690–715, 1979.

[93] Robert E. Tarjan. Sensitivity Analysis of Minimum Spanning Trees and Shortest Path Trees. *Information Processing Letters*, 14(1):30–33, 1982. Corrigendum in Volume 23, Issue 4, 1986, page 219.

[94] Mikkel Thorup and Uri Zwick. Compact Routing Schemes. In *13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10. ACM Press, 2001.

[95] Elias Vicari. *On Locality and Related Problems: Communicating, Computing, Exploring*. PhD thesis, ETH Zurich, Diss. No. 17937, 2008.

[96] Bang Ye Wu, Chih-Yuan Hsiao, and Kun-Mao Chao. The Swap Edges of a Multiple-Sources Routing Tree. *Algorithmica*, 50(3):299–311, 2008.

[97] Fu-Long Yeh, Shyue-Ming Tang, Yue-Li Wang, and Ting-Yem Ho. The tree longest detour problem in a biconnected graph. *European Journal of Operational Research*, 155(3):631–637, 2004.

Publications

The following lists all publications written during the three and a half years of my being a Ph.D. student at ETH Zurich.

Journals

1. Beat Gfeller and Elias Vicari. A Faster Distributed Approximation Scheme for the Connected Dominating Set Problem for Growth-Bounded Graphs. *Ad Hoc & Sensor Wireless Networks*, 6(3–4):197–213, 2008.
2. Beat Gfeller, Leon Peeters, Birgitta Weber, and Peter Widmayer. Single Machine Batch Scheduling with Release Times. *Journal of Combinatorial Optimization*, 17(3):323–338, April 2009.
3. Beat Gfeller, Nicola Santoro, and Peter Widmayer. A Distributed Algorithm for Finding All Best Swap Edges of a Minimum Diameter Spanning Tree. *IEEE Transactions on Dependable and Secure Computing*, 2009. To appear.

Conferences

1. Beat Gfeller, Leon Peeters, Birgitta Weber, and Peter Widmayer. Online Single Machine Batch Scheduling. In *31st International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 4162 of *Lecture Notes in Computer Science*, pages 424–435. Springer, 2006.
2. Beat Gfeller, Matúš Mihalák, Subhash Suri, Elias Vicari, and Peter Widmayer. Counting Targets with Mobile Sensors in an Unknown Environment. In *3rd International Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS)*, volume 4837 of *Lecture Notes in Computer Science*, pages 32–45. Springer, 2007.
3. Beat Gfeller and Elias Vicari. A Randomized Distributed Algorithm for the Maximal Independent Set Problem in Growth-Bounded Graphs. In *26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–60. ACM Press, 2007.
4. Beat Gfeller, Nicola Santoro, and Peter Widmayer. A Distributed Algorithm for Finding All Best Swap Edges of a Minimum Diameter Spanning Tree. In *21th International Symposium on Distributed Computing (DISC)*, volume 4731 of *Lecture Notes in Computer Science*, pages 268–282. Springer, 2007.
5. Beat Gfeller and Elias Vicari.
   A Faster Distributed Approximation Scheme for the Connected Dominating Set Problem for Growth-Bounded Graphs. In *6th International Conference on Ad-Hoc, Mobile, and Wireless Networks (ADHOC-NOW)*, volume 4686 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2007.
6. Beat Gfeller, Matúš Mihalák, Subhash Suri, Elias Vicari, and Peter Widmayer. Angle Optimization in Target Tracking. In *11th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 5124 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2008.

7. Beat Gfeller. Faster Swap Edge Computation in Minimum Diameter Spanning Trees. In *16th Annual European Symposium on Algorithms (ESA)*, volume 5193 of *Lecture Notes in Computer Science*, pages 454–465. Springer, 2008.

8. Shantanu Das, Beat Gfeller, and Peter Widmayer. Computing Best Swaps in Optimal Tree Spanners. In *19th International Symposium on Algorithms and Computation (ISAAC)*, volume 5369 of *Lecture Notes in Computer Science*, pages 716–727. Springer, 2008.

9. Beat Gfeller and Peter Sanders. Towards optimal range medians. In *36th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5555 of *Lecture Notes in Computer Science*, pages 475–486. Springer, 2009. Also invited to a special issue.