Compact Forbidden-set Routing

Andrew David Twigg

King's College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

September 2006

Abstract

Routing on the Internet is policy-based, meaning that each node is free to decide how to assign costs to paths. This freedom is important since the nodes are autonomous, competing organizations whose path preferences may be dictated by external factors (such as economic or political) rather than simply by path length. Although shortest-path routing is well-understood, little is known about the complexity of policy routing. The only known algorithms for policy routing use routing trees – for each destination, construct a routing tree and forward packets along it.

A negative result of Griffin et al. shows that routing tree-based algorithms (including the Internet routing algorithm, BGP) may not converge when arbitrary policies are used, and deciding whether they will is NP-complete. Yet there are no better algorithms known for policy routing; one possible reason is that the problem is much harder than shortest-path routing.

We study the complexity of policy routing with *forbidden-set* policies – each node specifies a set of forbidden nodes and wants to route on paths that avoid them. We begin by proving some new intractability results and reviewing known ones about routing tree-based algorithms. We show that routing trees are both impractical (they may not exist) and intractable (deciding if they exist is NP-complete) for forbidden-set policies on tree-like networks. We also prove the first communication complexity results for deciding if stable routing trees exist – for general policies, we show that communication exponential in the network size is needed. This implies that routing trees are a bad choice, even for some simple policy routing problems.

We describe the first compact forbidden-set routing schemes that do not suffer from non-convergence. For degree-d n-node graphs of treewidth t, our schemes use space $\tilde{O}(t^2 d)$ bits per node; a trivial scheme uses $O(n^2)$ and routing trees use $\tilde{O}(n)$ per node[1]. We also show how to do forbidden-set routing on planar graphs between nodes whose distance is less than a parameter l. We prove a lower bound on the space requirements of forbidden-set routing for general graphs, and show that the problem is related to constructing an efficient distributed representation of all the separators of an undirected graph. Finally, we consider routing while taking into account path costs of intermediate nodes and show that this requires large routing labels. We also study a novel way of approximating forbidden-set routing using quotient graphs of low treewidth.

[1]These results have since been improved and extended [CT07]

Acknowledgements

This dissertation would not have been possible without the help of many people. I must thank my supervisor, Ken Moody for supporting almost everything I have wanted do (including far too much rowing) and for his advice when I got stuck or confused.

Thanks to my friends - Athena, John, Jason and James for supporting me and for the numerous – and often humorous – coffee and lunch breaks. Also thanks to everyone else at the Computer Laboratory – Dave Eyers, Tim Moreton, Brian Shand, Sid Chau, David Ingram, Eiko Yoneki, Lauri Pesonen, Tim Griffin and others, for always being ready with encouraging advice, reminding me to eat lunch and letting me test my strange ideas on them.

I once read that you should always try to surround yourself with people who are smarter than you. During my stay at BRICS in Aarhus, this was annoyingly easy – I want to thank Mogens Nielsen for his inspiring enthusiasm, and also Karl Krukow and Marco Carbone. I also want to thank everyone that I have had useful discussions with, especially Rahul Sami, Peter Bro Miltersen, Bruno Courcelle, Andrew Thomason, Elan Pavlov, Mike Paterson, Tim Griffin, Sid Chau, Rob Ennals (from whom this LaTeX thesis style is inherited) and many more.

I wish to thank the Engineering and Physical Sciences Research Council (EPSRC), British Telecommunications Research (BTexact), The Marie Curie Foundation, King's College Cambridge and the Cambridge Philosophical Society for providing generous financial support.

Finally, and most of all I thank my parents and younger siblings Nicky and Stephanie.

Contents

1 Introduction **1**

1.1 Introduction 1
1.2 Chapter 3: Routing trees 2
1.3 Chapter 4: Towards compact forbidden-set routing 5
1.4 Chapter 5: Handling intermediate nodes 7
1.5 Chapter 6: Approximating forbidden-set routing 8

2 Preliminaries **10**

2.1 Graph theory 10
2.2 Labeling schemes 12
2.3 Communication complexity 14
2.4 Boolean circuits 16

3 Routing Trees **18**

3.1 The stable paths problem 18
3.2 Routing algebras 22
3.3 Forbidden-set routing 28
3.4 Communication complexity of solvability 32
3.5 Proof labeling schemes 44
3.6 Discussion 50

4 Towards Compact Routing **52**

4.1 Introduction 52
4.2 Motivation 53
4.3 Preliminaries 54
4.4 Deciding if there exists a zero-cost path 56
4.5 A forbidden-set routing scheme 58

4.6 Distance separator labels 62
4.7 A partial forbidden-set routing scheme for planar graphs 82
4.8 Decremental graph connectivity 85

5 Handling Intermediate Nodes 100
5.1 An $\Omega(\sqrt{n})$ lower bound 101
5.2 A $\tilde{O}(\sqrt{kn})$ upper bound 103
5.3 A 2-approximate scheme 112
5.4 Bounded-distance forbidden sets 114
5.5 Compact routing on good paths 117
5.6 Nondistributed data structures 119
5.7 Dynamic labeling schemes 120
5.8 Discussion 123

6 Approximating Forbidden-set Routing 125
6.1 Compact routing with a small number of objective costs 126
6.2 Approximate separator labels 130

7 Discussion 133
7.1 SPPs and routing trees 134
7.2 Compact routing schemes 134

CHAPTER 1

Introduction

1.1 Introduction

A fundamental task for any communications network is routing – the process of discovering paths between nodes in the network and using them for communication. Without a path connecting two nodes, they cannot send packets to each other and so the problem of deciding reachability is crucial to any routing algorithm. The basic job of any routing algorithm is to allow nodes to route on paths having low cost – but what do we mean by a low-cost path? The usual view of routing is to assign weights to edges, and define the cost of a path as the sum of its edge costs. This is known as the shortest-path routing problem.

We study a problem motivated by routing in networks having the following properties:

1. There is no centralized control, so all decisions should be made using *local information*.
2. The nodes are *autonomous*, meaning they are free to make their own independent decisions (we shall explain later what these decisions are).
3. The network is large, so nodes cannot store a piece of information for every other node, i.e. we want $o(n)$ space per node.

The Internet is an example of such a network – each node may be an independent organization with its own economic aims, possible competing with other organizations to provide

connectivity in the network. We shall not be interested in the economics of Internet routing, but we shall be interested in the routing problems that such a situation creates. We begin by describing at a high level, how Internet routing is done. Later on, we shall use this as an abstract model for routing in which we can prove various results.

The main protocol used for routing on the Internet is known as the Border Gateway Protocol (BGP). It works roughly as follows: a node advertises to its neighbours the route it currently uses to each destination. A node with many neighbours will thus learn about many routes to some destination j. It then selects one of these routes as the route that it will use to send data to j; subsequently, it can advertise this route to all of its neighbours. This process repeats until the set of routes stabilises (and so the protocol converges), and each node discovers at least one route to every destination (if such a route exists). Since there may be many possible routes to choose from, a crucial decision a node must make is route selection: given all the currently available routes to a destination j, which one should it choose? In the early days of the Internet (when it was known as ARPANET), each node simply chose the *shortest* available path [MW77]. In this case, the algorithm we have just described can be seen as a distributed algorithm for solving the shortest-path problem, and indeed it is possible to show that this algorithm will converge in finite time and every node will discover the shortest path to every destination.

However, today's Internet no longer consists of machines owned and run by a single organisation; instead it consists of independent competing organisations whose routing preferences are influenced by external factors other than path length, such as commercial relationships with other organizations in the network. For this reason, shortest-path routing is often not appropriate or desirable (the shortest path from u may go via another organization that u wishes to avoid). BGP allows nodes complete freedom to pick routes according to local *routing policies*, and this leads us to the *policy routing* problem – each node has a policy that defines how it assigns costs to paths, and each node wants to route on paths that are of low cost to itself. Very little is known about the complexity of policy routing, in contrast to the problem of shortest-path routing, which is very well-understood. Our aim is to develop an understanding of policy routing, and how to design good algorithms for routing with specific classes of policies.

1.2 Chapter 3: Routing trees

We begin by discussing a simple but widely-used method of routing, that we shall refer to as a 'routing tree'-based scheme. Imagine that we want to send packets on shortest paths between

nodes. A *routing tree* for a node v is simply a spanning tree rooted at v. Imagine that we construct a forest of routing trees, one for each node in the network. Then to route to a destination v, we find its routing tree and forward the packet along the edges of the tree until it reaches the root. The set of shortest paths to each node v is a routing tree for v. Therefore we can construct the forest by running an efficient distributed shortest-path algorithm, and storing at each node the parent node for each routing tree. This scheme uses space $O(n)$ at each node for a network of n nodes (from now on, we shall use n to denote the number of nodes).

For routing in autonomous networks with no centralized control, we need our routing trees to satisfy an additional property, called *stability*. A routing tree is *stable* if no node in the tree can switch to a lower cost path without creating a cycle in the tree. Throughout the thesis, we shall assume that stable routing trees are the only ones that we can use for routing. The reason for this is that if the nodes are autonomous, then given the choice between a path of high cost and a path of low cost to a destination, we have to assume that they will pick the low cost path for routing to that destination. The collection of shortest paths to some node has a useful property, sometimes referred to as the principle of optimality: *any subpath of a shortest path is a shortest subpath*. This property implies that a shortest-path routing tree will always be stable, if all nodes prefer shorter paths over longer paths. In modelling policy routing, we shall assume that each node u has a policy c_u that assigns a nonnegative cost $c_u(P)$ to each path P. Clearly, it is possible to construct a set of policies so that the principle of optimality no longer holds. In this case, it is natural to ask if we can still construct stable routing trees.

Griffin et al. [GSW02] answered this question in the negative. Furthermore, they showed that given a set of policies (encoded in a particular way) and a network, it is NP-complete to decide whether there exists a stable routing tree to some fixed destination. This intractability result is particularly surprising because it models how routing actually happens on the Internet – not only is it possible that the BGP algorithm may not converge to a solution (i.e. a set of stable routing trees), but it is NP-complete to decide if it will do so. Why then do we still use BGP? The main answer is that it still works ‘in practice’, but as the Internet grows and we become more reliant on it as a means of communication, this answer will eventually not be a good enough one.

In light of this hardness result, there are two natural ways in which we might hope to attack the problem, if we wish to construct useful algorithms for the policy routing problem. One direction is to restrict the class of policies allowed in the hope that the reduction in expressiveness will permit an efficient algorithm. Another direction is to restrict the class of networks.

Feigenbaum et al. [FKMS05] investigated the first direction. They studied the simple class of *next-hop preferences* where the cost of a path depends only on its next-hop. They showed that a stable routing tree always exists and so deciding solvability is trivial. Gao and Rexford [GR00] suggested that next-hop preferences capture the effect of ASes having different commercial relationships with neighbouring ASes. However, there are many desirable classes of policies that cannot be expressed in terms of next-hop preferences. For example, the government of country X may want to avoid any path that goes through some other country Y, perhaps because X is afraid that Y may do bad things to its packets, or because it does not want Y to know who it is communicating with. This motivates the *forbidden-set routing* problem, introduced by Feigenbaum et al. [FKMS05]: each node u has a forbidden set $S(u) \subseteq V(G)$ of nodes, and the cost $c_u(P)$ of a path P from u is the number of nodes it contains from $S(u)$, i.e. $c_u(P) = |S(u) \cap P|$. In addition to being a relevant and interesting class of routing policies, the problem is interesting from a graph theory point of view, since there is no path P from u to v with $c_u(P) = 0$ iff $S(u)$ separates u and v in G.

In Chapter 3, we further the study of routing trees for the policy routing problem. Our main results are following.

- We show that deciding if there exists a stable routing tree where the nodes use forbidden-set preferences is NP-complete, even on bounded treewidth graphs. This shows that even if we severely restrict both the class of policy *and* the class of networks, deciding solvability is still intractable. This rules out the possibility of using a single routing tree for policy-based routing, even in simple cases.

- We show that a small change in policy can give a huge change in complexity of deciding solvability – it is trivial for next-hop preferences but NP-complete for two-hop preferences. We conjecture that there exists a dichotomy theorem for SPP solvability, i.e. for a given class of policy, it is either NP-complete or trivial.

- We prove the first communication complexity results for solvability; in particular, any distributed algorithm must communicate $2^{\Omega(n)}$ bits over at least $\Omega(n)$ edges in the worst-case. We also prove lower bounds for the class of forbidden-set preferences.

- Finally, we consider labeling the nodes so that they can verifiably and locally check if the current path assignment is a stable routing tree. We show an $\Omega(n)$ lower bound on the

proof size and give a proof labeling scheme of size $O(n)$, hence this is tight in the worst case.

These results suggest that routing trees are not a good idea for policy routing, even for simple networks (bounded treewidth) and simple policies (forbidden-set).

1.3 Chapter 4: Towards compact forbidden-set routing

In Chapter 4 we forget about using routing trees, and try to develop a model that will allow us to construct efficient routing schemes for the forbidden-set routing problem. The results of Griffin et al. [GSW02] show that for policy routing, it may be impossible to construct a stable routing tree and so it is not always possible to route on lowest-cost paths using this method.

Consider the following simple (non-tree-based) scheme for policy routing. Each node w stores a table where the entry (u, v) specifies the next hop from w on the path from u to v of lowest cost to u. When a node u wants to send a packet to destination v, it writes into the header of the packet the string $\langle u, v\rangle$. Now when some node w receives this packet, it looks up the entry $\langle u, v\rangle$ to find the next link for this packet. This way, each node can route on its lowest-cost path to each destination. However, the downside is that each router now stores $O(n^2)$ entries in its local routing table, which is too demanding in a large network. With a routing tree, all the sources whose paths pass through the same node w to the same destination v must agree to use the same path from w and therefore each node can store $O(n)$ entries. However, we know that stable routing trees are not guaranteed to exist, so we cannot always route on lowest-cost paths (even though the path clearly exists in the network!).

The above scheme can be seen as a simple instance of the following model of routing. Each node is assigned a data structure (called its routing table) and a label, which identifies the node to other nodes. Routing is then done as follows: if node u wants to route to v it writes v's label into the packet header. Nodes can then use their routing tables and v's label to decide how to forward the packet through the network. This model is known as *compact routing* and was introduced in a series of papers by Peleg and Upfal [PU89] who showed how to do stretch-k routing using $n^{O(1/k)}$ bits per label. Cowen [Cow99] showed how to route on stretch-3 shortest paths using $\tilde{O}(n^{2/3})$ bits per node[1]. These are both substantial improvements on the $O(n)$ space required by simply using routing trees. Indeed, a routing scheme is said to be *compact* if the

[1] $f(n) = \tilde{O}(g(n))$ if $\exists c \geq 0$ such that $f(n) = O(g(n)\log^c n)$

space requirement at each node is $o(n)$, i.e. sublinear in the number of nodes in the network. For more details about localized and compact data structures for shortest-path routing, we refer the reader to the excellent survey paper by Gavoille and Peleg [GP03]. In fact, it is known that compact routing is almost-optimal for approximate shortest-path routing: Thorup and Zwick [TZ01b] have given a scheme that routes on paths of stretch three (a path has stretch k if its length is within a factor k of optimal) using routing tables of size $\tilde{O}(n^{1/2})$ and $O(\log n)$-bit labels. By a proven conjecture of Erdos, related to the girth of a graph, this space requirement is optimal to within logarithmic factors.

The question we wish to answer is the following: for the special case of forbidden-set routing, can we do better than space $O(n^2)$ per node, while still being able to route on *all* forbidden-set-avoiding paths? We answer this in the positive by constructing a compact routing scheme that routes on the shortest path $u - v$ that avoids $S(u)$. Our main results are the following. Let k be the size of the largest forbidden set, i.e. $k = \max_u |S(u)|$, and let d be the degree of G.

- We show how to do forbidden-set routing on trees using $O(k \log n)$ bits per node. However, the problem on trees is simple since a set S is a separator of u, v in T iff at least one element of S lies on the unique path between u, v.

- For the class of bounded cliquewidth graphs, we can construct a forbidden-set routing scheme using $O(dk \log^2 n)$ bits per node and labels of size $O(\log n)$ bits. However, the hidden constant may be a tower of exponentials in the cliquewidth, making the scheme quite impractical, but nevertheless hinting at the existence of more efficient schemes.

- For graphs of treewidth t, we give a forbidden-set routing scheme using $O(t^2 dk \log^2 n)$-bit labels.

- We give a space lower bound of $\Omega(n)$ bits per node for any forbidden-set routing scheme in general graphs.

We argue that for policy-based routing on the Internet, compact routing schemes are better than using routing trees. Since no routing trees are ever constructed, our routing schemes can send packets on all lowest-cost paths between nodes, whenever they are reachable in the network. In contrast, packets can be sent only if a stable routing tree exists where the source node is not assigned the empty path (and deciding if such a tree exists is NP-complete even with forbidden-set policies on bounded treewidth graphs). So far nothing is known about the

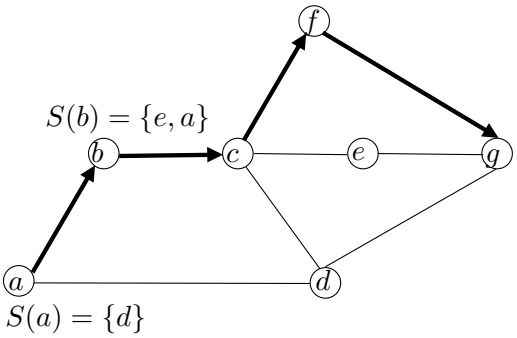


Figure 1.1: The only good path from a to g is marked in bold. The same path in reverse is not a good path from g to a

viability of compact routing schemes for policy routing. In particular, it may be that the space requirements are higher than $\Omega(n)$ per node. The idea of using compact routing on the Internet has been suggested elsewhere. For example, Krioukov et al. [KFY04] suggest that existing compact routing schemes perform excellently for Internet-like topologies. However, this ignores the freedom offered by policy routing, which is the main attraction of BGP. Until there exists a scheme that can handle policy routing (even for restricted policies such as forbidden-set), there will remain no viable alternative to BGP. We believe that our algorithms take us an important step closer towards this goal, and also provide some interesting and difficult questions along the way.

1.4 Chapter 5: Handling intermediate nodes

In the previous chapter, we constructed forbidden-set routing schemes under the assumption that intermediate nodes will always forward packets, even if the path on which they are forwarding the packets is costly to them. We call a path *good* if all its subpaths have zero cost. An example of this is shown in Figure 1.1. Note that for shortest-path routing this is not an important concern, because every subpath of a shortest path is itself a shortest path. The problem of routing on good paths can be seen to model a common situation in BGP routing: if nodes only advertise paths of zero cost (to themselves), then no node will ever discover a non-good path. It is important to note that this 'goodness' property is implicit in stable routing trees – if u has a lower cost path available than its current one then it will choose it, regardless of the preferences of other nodes that may need to route through u (although this may be restricted by

the requirement that the new path does not create a cycle). Since we are interested in compact routing schemes, we ask ourselves the following question: what difference does routing on only good paths make to the complexity of routing schemes for the forbidden-set routing problem?

In Chapter 5, we show that the answer is somewhat negative – taking into account the costs of intermediate nodes makes the problem much more difficult. Let k be the size of the largest forbidden set, i.e. $k = \max_k |S(u)|$. Our main results are the following.

- We show that if the forbidden sets are of size at most k then on trees, labels of size $\Omega(\sqrt{n} + k \log n)$ are required to decide if there is a good path between two nodes. This should be compared with the $O(\log n)$ bound shown in Chapter 4 for simply routing on zero-cost paths to the source node.
- We prove an almost-optimal $\tilde{O}(\sqrt{kn})$ upper bound and show various time-space tradeoffs for centralized versions of the problem.
- We also show that routing can be done using $\tilde{O}(k)$-bit routing tables and labels, but a packet may traverse $\Omega(n)$ edges before being returned if a good path does not exist.

Our results imply that it may not be practical to construct forbidden-set routing schemes that take into account costs incurred by intermediate nodes, unless we are willing to sacrifice features such as the ability to decide if there exists a good path before sending the packet.

1.5 Chapter 6: Approximating forbidden-set routing

We finish by considering an approach to *approximating* forbidden-set routing. We partition the network into connected clusters and instead of choosing arbitrary subsets of nodes, the forbidden sets are a subset of these clusters. This has the effect of avoiding whole clusters rather than individual nodes. We define the problem of obtaining a cluster graph with good graph-theoretic properties, and motivate the problem of obtaining a cluster graph with bounded treewidth. We show that if we can construct a cluster graph having small treewidth, then we can apply our forbidden-set routing schemes from Chapter 4 to it. This may be of interest when the network lends itself naturally to clustering.

We begin by considering an approach inspired by the work of Feigenbaum et al. [FKMS05] – they considered a relaxed version of shortest-path routing where each link has a number of objective values associated with it, representing for example its delay, its bandwidth and other

metrics. All nodes agree on these values, in the same way that all nodes agree on the weights of edges for shortest-path routing. Each node has an individual cost function, which is a convex combination of the objective values assigned to edges (for example, one node may be interested in paths minimizing the sum of delays, while another may be interested in paths minimizing another metric). They showed that using a small number of routing trees (instead of a single routing tree) is sufficient for all nodes to route on almost-optimal paths. Their scheme does not immediately imply a space-efficient routing scheme, though. We shall show how to use their construction to build a space-efficient compact routing scheme with a small increase in the approximation factor. We can then observe that this multiple objective cost problem can be seen as a special case of clustering the network and assigning costs to clusters. Since we are interested in forbidden-set routing, it is natural to ask if we can cluster the graph so as to obtain efficient forbidden-set routing algorithms for it.

CHAPTER 2

Preliminaries

In this chapter, we give some useful preliminary definitions and background to areas and basic results that we shall frequently refer to.

2.1 Graph theory

We assume familiarity with basic concepts in graph theory; see [Wil86] for a good reference text on graph theory. We shall model the network by an undirected simple graph $G = (V, E)$ having n nodes and m edges. The size of a graph is the number of nodes in the graph. Given a graph G, its node set is denoted $V(G)$ and its edge set $E(G)$. The *degree* of a node u in G is denoted by $deg_G(u)$ and the *maximal degree* of a node of G is denoted by $\Delta(G)$. The *neighbourhood* of a node $u \in V(G)$ is denoted by $N_G(u) = \{v \in V(G) : \{u, v\} \in E(G)\}$ and the neighbourhood of a set of nodes $S \subseteq V(G)$ is denoted by $N(S) = \bigcup_{s \in S} N(s) \setminus S$. We shall drop the subscripts when it is clear which graph we are referring to. The *transitive closure*, or *reachability graph* of a graph G is denoted by G^*.

A *path* is a sequence of nodes such that from each of its nodes there is an edge to the next node in the path, and no nodes are repeated. The *length* of a path is the number of edges contained in the path. A *cycle* is a path, except that the start and end nodes are the same. A graph is *acyclic* iff it contains no cycles of length > 1. We shall denote the empty path by ϵ.

If $P = v_1, \ldots, v_k$ and $Q = v_k, \ldots, v_r$ then $PQ = v_1, \ldots, v_r$ is the concatenation of P and Q ($\epsilon P = P = P\epsilon$). A *Hamiltonian* path is one that visits each node of the graph exactly once. A graph that contains a Hamiltonian path is called Hamiltonian. The distance from u to v in G is the length of the shortest path from u to v and is denoted by $d_G(u, v)$.

A *separator* (*cut*) is a set of nodes (edges) whose removal disconnects the graph into connected subgraphs. A cut is denoted by (X, Y) where $X, Y \subseteq V(G)$ and the *value* (or *size*) of the cut is the number of edges needed to partition the graph into (X, Y). A graph is *k-connected (k-edge-connected)* iff it remains connected after removing any $k - 1$ nodes (edges). A graph is k-connected iff it contains k node-disjoint paths between any two nodes. The *connectivity* $\kappa(G)$ of a graph G is the minimum number of nodes needed to disconnect G. By convention, K_n has connectivity $n - 1$ and a disconnected graph has connectivity 0.

2.1.1 Graph layouts

A *linear layout*, or layout, of an undirected graph $G = (V, E)$ with n nodes is a bijective function $\phi : V \to \{1, \ldots, n\}$. Given a layout ϕ of a graph G and an integer i, we define the set $L(i, \phi, G) = \{u \in V | \phi(u) \leq i\}$ and the set $R(i, \phi, G) = \{u \in V | \phi(u) > i\}$. We shall use $L(i)$ and $R(i)$ when ϕ and G are obvious. The *edge cut* at position i of ϕ is defined as

$$\Theta(i, \phi, G) = |\{\{u, v\} \in E | u \in L(i) \wedge v \in R(i)\}|.$$

A common way to represent a layout is to align the nodes horizontally, mapping each node u to its position $\phi(u)$. The *cutwidth* of a layout is $\max_{i \in \{1,\ldots,n\}} \Theta(i, \phi, G)$ and the *cutwidth* of a graph is the minimum cutwidth over all possible layouts of G, denoted by $cw(G)$.

2.1.2 Treewidth

Many problems have efficient algorithms when restricted to trees. The notion of treewidth, introduced by Robertson and Seymour [RS86] as part of their work on graph minors, captures the idea that a graph may be 'tree-like'. It is often possible to construct efficient algorithms for difficult problems, when restricted to small treewidth graphs. We shall make frequent reference to the concept of treewidth, so we define it here for reference. A *tree decomposition* of a graph $G = (V, E)$ is a pair (X, T) where $T = (I, F)$ is a tree and each node of i of T is associated with a subset $X_i \subseteq V$ with the following properties.

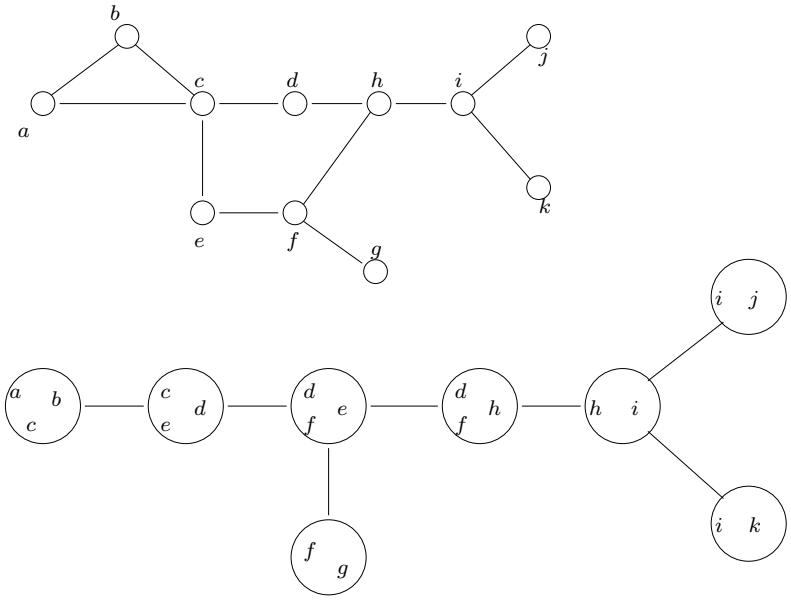


Figure 2.1: An example of a graph and a tree decomposition of width 2

1. The X_i's cover the nodes of G, i.e. $\bigcup_{i \in I} X_i = V$;
2. For every edge $\{v, w\} \in E$, there is some node $i \in T$ where $v, w \in X_i$;
3. For every $v \in V$, the set $\{i \in I | v \in X_i\}$ is a connected subtree of T.

The *width* of a tree decomposition (X, T) is defined as $\max_{i \in I} |X_i| - 1$. The *treewidth* of a graph G is the minimum width over all tree decompositions of G, and we shall denote it by $tw(G)$. For example, trees have treewidth one and cliques have unbounded treewidth. Figure 2.1 shows an example of a graph and a tree decomposition. In some cases, T will be considered to be a rooted tree, in which case a specific node of T shall be its root. A tree decomposition with T a rooted tree is called a rooted tree decomposition. For a node $i \in I$, we call the set X_i the *bag* of i. More details about the history and uses of treewidth can be found in the paper by Bodlaender [Bod93b].

2.2 Labeling schemes

Implicit in any distributed algorithm is a representation of the network, and many network representations are inherently global; for example each node is assumed to know the entire

network. A common distributed representation of a graph is to assign nodes unique identifiers from $\{1, \ldots, n\}$ and then store at each node the identifiers of its neighbours. In such a scheme, answering a query such as 'what is the distance between u and v?' may require access to data distributed across the entire network, e.g. by running a shortest path algorithm. Another idea is to use a completely local representation of the network, for example having each node know the entire graph. The problem is that both these representations are inefficient: the first has high query time and second has high space requirements at each node.

Labeling schemes were introduced in [KNR92]. Assume that $P(x_1, \ldots, x_k)$ is some graph property on nodes $x_1, \ldots, x_k$ that we want to answer, e.g. $P(x_1, x_2) = d(x_1, x_2)$ or $P(x_1, \ldots, x_k) = 1$ iff the subgraph induced by $x_1, \ldots, x_k$ is a clique. A *P-labeling scheme* (L, f) consists of two things:

1. A *marker algorithm* that takes as input the graph and assigns a label $L(v)$ to each node v (L is called a *labeling* of the nodes);

2. A *decoder algorithm* f such that $f(L(x_1), \ldots, L(x_k)) = P(x_1, \ldots, x_k)$.

The *size* of a labeling is the maximum size of a label given to some node. For a family of graphs $\mathcal{G}$ we denote a labeling scheme by (L, f), where L is the labeling computed by the marker on the particular graph $G \in \mathcal{G}$ and f is the decoder algorithm (that depends only on the marker algorithm, not the particular $G \in \mathcal{G}$).

The labeling L can be viewed as a distributed data structure, with the decoder as a distributed algorithm that answers $P(x_1, \ldots, x_k)$ using data only stored at $x_1, \ldots, x_k$. If the labels are short then they can be given as part of the query, by using them in place of the traditional $\lg n$-bit node identifier (e.g. in packet headers). We shall be interested in the *maximum* label size rather than the *total* label size since the graph given to the marker algorithm is unknown and so any node could be assigned a label of the maximum size, which would require that each node has sufficient memory to store it. Clearly, a good bound on the individual label size gives a good bound on the total size but not the other way around. It is also clear that labels of unrestricted size can be used to encode any desired property (by storing the entire graph). For a labeling scheme to be useful the labels should be short (say of length polylogarithmic in the number of nodes), and the time to answer a query given the labels be small (also polylogarithmic).

2.2.1 An example – adjacency labeling

We now give an example of adjacency labeling in trees, to give a better understanding of the local nature of these schemes. Adjacency labeling schemes were first introduced in [KNR92] for the implicit representation of graphs. In particular, a labeling scheme using $2\log n$-bit labels for the class of trees was given, which we now describe. Given an n-node tree T, labels are assigned to nodes as follows. Choose a root and associate a unique identifier $\mathrm{ID}(v) \in \{1 \dots n\}$ with each node $v \in T$, then assign a node v with parent w the label $(\mathrm{ID}(v), \mathrm{ID(w)})$ (the root r is assigned the label $(\mathrm{ID}(r), 0)$). Now, given two labels $(\mathrm{ID}(v), \mathrm{ID}(w))$ and $(\mathrm{ID}(v'), \mathrm{ID}(w'))$, the nodes v, v' are neighbours iff either $\mathrm{ID}(v) = \mathrm{ID}(w')$ or $\mathrm{ID}(v') = \mathrm{ID}(w)$. The scheme can be extended to families of graphs having separators of bounded size such as c-decomposable graphs (e.g. bounded genus graphs and bounded treewidth graphs).

Another basic result in the area of graph labeling concerns distance labeling schemes. A distance labeling scheme is a labeling scheme (L, f) where $f(u, v)$ is the distance between two nodes u, v in the graph. It has been shown [KNR92] that a class of $2^{\Omega(n^{1+\epsilon})}$ n-node graphs must use adjacency labels (and thus distance labels) whose total combined length is $\Omega(n^{1+\epsilon})$ bits. Hence, at least one label must be of $\Omega(n^{\epsilon})$ bits. More specifically, for the class of all unweighted graphs, any adjacency (and hence distance) labeling scheme must assign some node a label of size $\Omega(n)$ bits.

Given the $\Omega(n)$ lower bound for general graphs, a large amount of research has tackled the problem of constructing *approximate* distance labeling schemes. Thorup and Zwick [TZ01a] give a distance labeling scheme with approximation factor $2k-1$ using $\tilde{O}(kn^{1/k})$ bits per label, which is essentially optimal by a 1963 girth conjecture of Erdos that has been proven for certain small values of k including $k = 2$.

2.3 Communication complexity

In Chapters 3 and 4 we shall make use of results from communication complexity. Therefore, we give some basic concepts here but for further information and details of proofs, we refer the reader to the excellent and interesting book [KN97] by Kushilevitz and Nisan.

Let X, Y, Z be arbitrary finite sets and let $f : X \times Y \to Z$ be an arbitrary function. There are two players, Alice and Bob, who wish to evaluate $f(x, y)$ for some inputs $x \in X$ and $y \in Y$. The difficulty is that Alice only knows x and Bob only knows y. Thus, to evaluate the

function, they will need to communicate with each other. The communication will be carried out according to some fixed protocol P (which depends only on the function f). The protocol consists of the players sending bits to each other in turn, until the value of $f(x, y)$ can be determined. We are usually only interested in the amount of communication between Alice and Bob, so we ignore the internal computations each of them makes. Thus, Alice and Bob are assumed to both have unlimited computational power. The *cost* of a protocol P on input (x, y) is the number of bits communicated on that input. The cost of a protocol P is the worst case cost of P over all inputs (x, y). The *communication complexity* of f is the minimum cost of a protocol that computes f.

Although we assume that the players have unlimited computational power, the way that the choices are made at each step of the protocol can have an impact on the amount of communication required. The *deterministic* communication complexity of a function f, denoted $D(f)$, is the minimum cost of any deterministic protocol that computes f, i.e. one that makes no random choices and computes the answer deterministically. The *randomized* communication complexity is defined similarly, except that the protocols used by the players are randomized, and the result must be known with a sufficiently high probability. The book [KN97] gives several examples of functions whose deterministic complexity is $\Omega(n)$, yet there exist randomized protocols that use only $O(\log n)$ bits of communication (for example, the set equality function).

We shall also be interested in nondeterministic communication protocols. In a nondeterministic protocol, we can imagine the presence of an all-powerful prover who knows the inputs of both players (and hence the answer). Therefore, the communication required is equivalent to that needed to verify a nondeterministic guess of the answer. For example, consider the disjointness function $DISJ$ on n-bit strings where $DISJ(P, Q) = 1$ iff $P \cap Q = \emptyset$. If $DISJ(P, Q) = 1$ then there exists an index i such that $P_i = Q_i$. The prover can tell both players i and they can verify that $P_i = Q_i$ with $O(\log n)$ bits of communication. We define the *nondeterministic* communication complexity of a function f is the minimum cost of any nondeterministic protocol that verifies that $f(x, y) = 0$ for any $x \in X, y \in Y$, and is denoted $N^0(f)$. Similarly, the *co-nondeterministic* communication of a function f is the minimum cost of any nondeterministic protocol that verifies that $f(x, y) = 1$ for any $x \in X, y \in Y$, and is denoted $N^1(f)$.

2.4 Boolean circuits

In Chapter 3 we make use of boolean circuits and boolean functions in proving some of our lower bound results for the complexity of finding stable routing trees. Results on the complexity of boolean functions also underly some of our results in the last section of Chapter 3. Therefore we now give a brief overview of the main (mostly simple) concepts that we shall use; for more details and related results in this deep and interesting area, we refer the reader to the book by Clote and Kranakis [CK02].

A *boolean circuit* is a directed acyclic graph with labeled nodes as follows:

- *Input* nodes have fan-in 1 and are labeled with a variable x_i or a constant in $\{0, 1\}$.
- *Gate* nodes have fan-in $k > 0$ and are labeled with a boolean function $\wedge$(AND), $\vee$(OR), $\neg$(NOT) on the k inputs. In the case that the label is $\neg$, the fan-in is restricted to be 1.
- *Output* nodes have fan-out 0.

A *boolean formula* is a boolean circuit having only one output gate. The edges of a circuit are called *wires*. The *depth* of a circuit is the maximum distance from an input to an output gate. It is important to note that any circuit can be modified, by using de Morgan's laws, to push all the negations to the input gates, without changing the depth of the circuit. Therefore, we will assume wlog that all gates are one of $\wedge, \vee$, and that the negated versions of each variable are available as inputs, i.e. x_1 and $\overline{x}_1$. Similarly, adjacent gates of the same type can be combined together, so we assume wlog that a circuit contains levels alternating between $\vee$ and $\wedge$ gates. A Π_d^k formula is a boolean formula of depth d where the top level gate (the output) is a $\wedge$ gate, and with fan-in bounded by k. A Σ_d^k formula is defined similarly, except that the top level gate is a $\vee$ gate. Π_1 and Σ_1 formulae are single literals and correspond to input gates in the circuit. Figure 2.2 shows an example of a Π_3 formula.

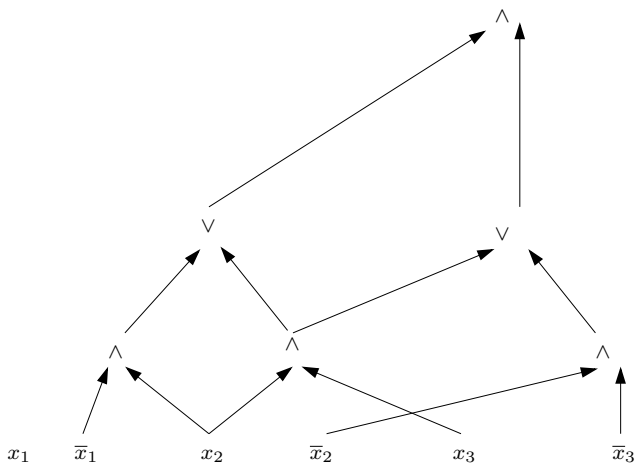


Figure 2.2: A Π_3 formula $((\overline{x}_1 \wedge x_2) \vee (x_2 \wedge x_3)) \wedge ((x_2 \wedge x_3) \vee (\overline{x}_2 \wedge \overline{x}_3))$

CHAPTER 3

Routing Trees

In this chapter we consider policy-based routing using routing trees. The chapter is structured in two parts. In the first part, we use the formalism of routing algebras and the stable paths problem to show hardness results on the computational complexity of policy-based routing using routing trees, even for simple policies. These extend other results due to Feigenbaum et al. [FKMS05] and Griffin et al. [GSW02].

In the second part of the chapter, we consider the stable paths problem as a problem in distributed computing and prove the first communication complexity lower bounds for it. In the final section, we describe the notion of proof labeling schemes, which provide a distributed representation of a solution that is locally verifiable. We prove a lemma that lets us use our communication lower bounds to give lower bounds on the proof size of deciding solvability of a stable paths problem.

The aim of this chapter is to convince the reader that routing trees are not practical for policy-based routing, even for seemingly simple policies such as forbidden-set routing.

3.1 The stable paths problem

That BGP is not guaranteed to converge was first observed by Varadhan et al. [VGE96]. More recently, Griffin et al. [GSW02] introduced the *stable paths problem* (SPP) as a tool to model the instabilities that can arise from using routing tree-based algorithms such as BGP. They use

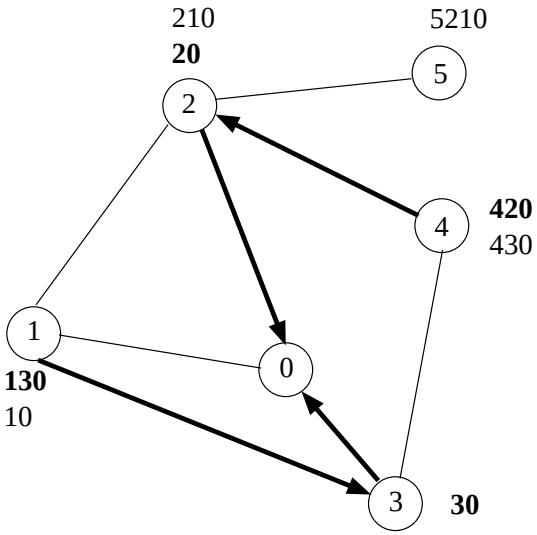


Figure 3.1: An instance of the stable paths problem. The nodes' path preferences are ordered from most preferred to least preferred, and the routing tree representing the solution is marked with bold edges.

the following motivating analogy: *If Dijkstra's algorithm solves the shortest path problem then BGP solves the stable paths problem.*

We now describe the stable paths problem. Let $G = (V, E)$ be an undirected rooted graph, with the root having identifier 0. All nodes wish to establish a path to the root. For each node $v \in V$, the set of *permitted paths* from v to the root is denoted by Σ_v. Each node has a total order $(\Sigma_v, \sqsubseteq_v)$ over its permitted paths. We assume that for all v, Σ_v contains the empty path ϵ, and $\epsilon \sqsubseteq_v \sigma$ for all $\sigma \in \Sigma_v$, i.e. any nonempty path is preferable to the empty path. A *stable paths instance* is written $S = (G, \Sigma, \sqsubseteq)$ where $\Sigma = \{\Sigma_v\}$ and $\sqsubseteq = \{\sqsubseteq_v\}$.

A *path assignment* π is a function that assigns a permitted path $\pi(v) \in \Sigma_v$ to each node $v \in V$ (the root's assigned path is $\pi(0) = \epsilon 0$). We say that a path $P = v_1 \dots v_r 0$ assigned to $v = v_1$ is *consistent* with a path assignment π if for all $v_i \in P, v_i \neq v$ implies that the path $\pi(v_i)$ is a subpath of P. A path assignment π is *valid* if for all v, the path $\pi(v)$ is consistent with π. Intuitively, the assigned paths of a valid path assignment are confluent, i.e. they form a tree rooted at node 0.

Valid assignments are important as this is how routing takes place over the Internet; routers examine the destination of incoming packets and simply forward them to the next hop on the route to that destination, which is the parent in the tree rooted at the destination. The problem

is that even though π may be valid, some node v might prefer (perhaps for economic reasons) another path $P' \in \Sigma_v$ to its assigned path $P = \pi(v)$. Therefore, as long as P' is also consistent with π, v may (at its own will) switch to using P'. We say that an assignment π is *stable at node* v if there is no other permitted path in Σ_v consistent with π that v prefers over $\pi(v)$. A path assignment π is *stable*, or is a *solution to* S, iff it is stable at every node. Stable routing trees are important since we assume that the only routing trees that we can use for routing are those corresponding to stable path assignments. A stable paths instance $S = (G, \Sigma, \sqsubseteq)$ is *solvable* if there exists a solution to S, and unsolvable otherwise. We can now define the problem SPP-SOLVABILITY:

Problem SPP-SOLVABILITY
Input: A stable paths instance $S = (G, \Sigma, \sqsubseteq)$.
Output: Is S solvable?

The main difference between the stable paths problem and the shortest paths problem is that the latter always has a unique solution, while the former may have one, none or many solutions. As an example, consider the network in Figure 3.1. A stable solution is indicated with bold edges. Note that node 5 prefers path 5210 to the empty path, but the path 210 is not part of this solution, so node 5 is assigned the empty path. It is easy to see that there is no stable solution where node 5 is assigned a nonempty path – node 3 will always prefer (and be able to switch to) path 30 and so node 1 will always be able to choose path 130 over 10. Therefore node 2 will have to choose path 20 and so node 5 will be assigned the empty path. Also, note that although node 2 prefers the path 210 to the path 20, it will never be able to use this path in a stable solution because node 3 will always be able to choose path 30 and so node 1 will always be able to choose path 130.

3.1.1 Results of this chapter

Compared to the shortest paths problem, very little is known about the complexity of the stable paths problem. Griffin et al. [GSW02] showed that deciding SPP-SOLVABILITY is NP-complete for general graphs. Given this result, there are two natural ways that we could hope to reduce the complexity. We could restrict the class of policies allowed, in the hope that the reduction in expressiveness will permit efficient algorithms. Another direction is to restrict the class of networks allowed, in the hope that this will allow more efficient algorithms.

Feigenbaum et al. [FKMS05] investigated restricting the policy. They studied the class of *next-hop preferences* where the cost of a path can depend only on the next-hop on the path. They showed that deciding solvability is trivial since a stable routing tree always exists. Gao and Rexford [GR00] suggest that next-hop preferences capture the effect of ASes having different commercial relationships with neighbouring ASes, in the sense that the cost of a path depends only on whether the next-hop is a customer or provider etc. (they do not capture *transit policies* where the next-hop depends on the previous hop). There are many useful policies that cannot be expressed in terms of next-hop preferences. For example, a node u may wish to avoid any route that goes through node v, perhaps because v is a competitor who may drop u's data or due to some economic agreement between them. This leads to the *forbidden-set routing* problem: each node u has a forbidden set $S(u) \subseteq V$ of nodes where the cost to u of a path is the number of nodes it contains from $S(u)$. Forbidden-set preferences capture a fundamental yet expressive class of routing policies, so showing that we can handle them efficiently would be an important positive result. Our main results are the following:

- We show (by a simple extension of a result of [FKMS05]) a strongly negative result – deciding solvability for forbidden-set preferences on bounded treewidth graphs is NP-complete. Thus, even if we severely restrict both the class of policy *and* the class of graphs, the problem of deciding solvability is still intractable. This almost certainly rules out the possibility of using a single routing tree for policy-based routing.

- We show that a small change in policy can give a huge change in complexity of deciding solvability – deciding solvability of an SPP is trivial for next-hop preferences but NP-complete for the class of two-hop preferences. We conjecture that there exists a dichotomy theorem for the problem of deciding solvability, i.e. for a given routing algebra it is either NP-complete or trivial.

- We prove that any distributed algorithm that decides if there is a set of stable paths must communicate $2^{\Omega(n)}$ bits across each of at least $\Omega(n)$ edges in the worst-case. We also prove lower bounds for the communication complexity of solvability using forbidden-set preferences.

- Finally, we consider labeling the nodes so that they can verifiably and locally decide whether the current routing tree is stable, and prove an $\Omega(n)$ lower bound on the label

size. We show that this is tight in the general case by giving a proof labeling scheme using $O(n)$-bit labels.

3.2 Routing algebras

In this section we describe the formalism of *routing algebras* introduced by Sobrinho [Sob03]. We then describe how they naturally generate instances of the stable paths problem. This allows us to completely separate the complexity of the policy from the complexity of the graph used. Note that the path preferences $\sqsubseteq$ over Σ implicitly encode information about the graph *and* the policy. The advantage of using the routing algebra formalism is that it separates policy and network. This will enable us to understand what makes certain policy classes hard by studying their algebraic properties, independent from the class of graphs used. In the work presented here, we do not study the link between algebraic features of policies and the complexity of the SPPs that they generate. Our main use for routing algebras is to succinctly and accurately describe the policy classes that were are interested in studying. As related work, Chau et al. [kCGG06] have investigated how the algebraic features of policies affect the convergence properties of Bellman-Ford-style iterative algorithms. However, the general problem of understanding how the algebraic properties of routing algebras relate to the convergence properties of algorithms and their complexity is still an open problem.

We shall now introduce routing algebras. Routing algebras can be thought of as generalising shortest-path routing in the following way. Consider Figure 3.2(a): there is a path from v to w of weight m and node u has an edge to node v of weight n, hence u has a path to w of weight at most $n + m$. Now let us generalise this as in Figure 3.2(b). Each edge has a *label* $l \in L$, and each path is described by a *signature* $\sigma \in \Sigma$. We assume that there is a special 'zero' signature $\epsilon \in \Sigma$ (similar to the zero element of a group) that denotes the empty path. Paths are composed using the binary operator $\oplus : L \times \Sigma \to \Sigma$ (paths are assumed to begin at the root, and are extended towards the source node). Finally, there is a totally-ordered set of *weights* $(W, \leq)$ and a *cost function* $f : \Sigma \to W$. We can now define a *routing algebra* $\mathcal{A}$ as the tuple $A = (L, \Sigma, \epsilon, \oplus, (W, \leq), f)$.

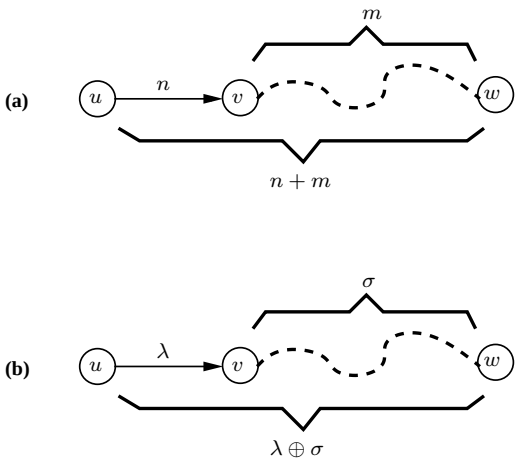


Figure 3.2: How path lengths are computed in the shortest path setting (a), and how path signatures are computed in the routing algebra setting (b).

3.2.1 Generating stable paths problems

Routing algebras naturally generate instances of the stable paths problem. Given a routing algebra $\mathcal{A}$ and a rooted graph G with edges labelled from L, we say that $\mathcal{A}$ *generates an instance* $\mathcal{A}(G)$ *of a stable paths problem* as follows. For every path from a node in G to the root node, its signature is generated by recursively applying $\oplus$ to the labels along the path. Every node then ranks its paths to the root using the cost function f, and a node prefers a path P with signature σ to a path P' with signature σ' iff $f(\sigma) \leq f(\sigma')$. We can now define the problem of solvability, restricted to SPP instances that are generated by some routing algebra. Let $\mathcal{A} = (L, \Sigma, \epsilon, \oplus, (W, \leq), f)$ be a routing algebra.

Problem $\mathcal{A}$-SPP-SOLVABILITY
Input: An undirected rooted graph G with edges labeled from L
Output: Is the stable paths problem instance $\mathcal{A}(G)$ solvable?

3.2.2 Next-hop routing

We now present a routing algebra for the next-hop policy routing problem. In this case, the cost of a path can depend only on the next hop on the path. Feigenbaum et al. [FKMS05] studied the class of next-hop preferences and showed that deciding solvability is trivial since a

stable routing tree always exists. The reason for this is that we can take the labeled rooted graph and build a minimum weight spanning tree rooted at the root of the graph. Such a tree always exists, and by the optimality property of minimum spanning trees (that every subpath of an optimal path is also an optimal subpath), this tree includes the minimum weight edge adjacent to each node. The class of next-hop preferences capture the effect of ASes having different commercial relationships with neighbouring ASes, and this model was suggested by Gao and Rexford [GR00] as a policy class for BGP routing where convergence would be guaranteed. The figure below shows a routing algebra $\mathcal{NH}$ for next-hop preferences. The algebra takes weights assigned to edges (representing the next-hop preferences) and computes the cost of a path from a node to the destination by setting the cost of the path to be equal to the cost of the first edge on the path. This operation is implemented by the composition operator $\oplus$, as described in Figure 3.2.

$$\begin{aligned} L &= \mathbb{N} \\ \Sigma &= \mathbb{N} \\ W &= (\mathbb{N}, \leq) \\ f(c) &= c \\ l \oplus c &= l \end{aligned}$$

Figure 3.3: A routing algebra $\mathcal{NH}$ for the next-hop preferences routing problem

Any SPP instance generated by a next-hop preferences algebra is always solvable [FSS04], thus the complexity of deciding if there exists a stable routing tree, i.e. $\mathcal{NH}$-SPP-SOLVABILITY, is trivial.

3.2.3 Two-hop routing

We now consider a class of policies that we call two-hop preferences. Here, each node can rank paths based only on the first two hops on each path. It might seem natural that this provides a small degree of extra expressiveness over next-hop preferences, but here we prove the surprising result that deciding solvability of two-hop preference SPPs is NP-complete. This shows that there is a complete change in the character of the problem in going from next-hop to two-hop preferences, and so there must be some important property of the algebra that permits this.

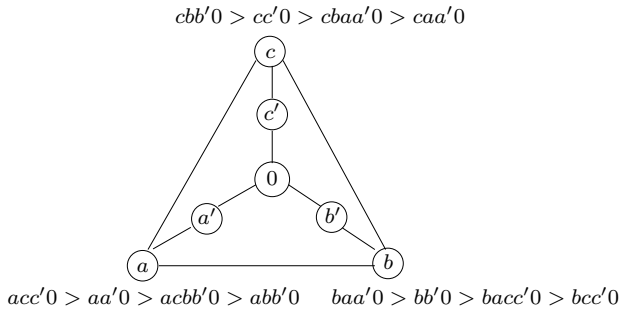


Figure 3.4: The bad triangle gadget. There is no stable set of paths that form a tree rooted at the center node, due to the cyclic preferences of the outer nodes.

To prove this, we make use of the following important construction of [FKMS05] known as 'bad triangle', which is a variation of the bad gadget construction introduced by Griffin et al. [GSW02]. The bad triangle is shown in Figure 3.4. It is not difficult to see that this network has no stable solution, and that the preferences can be ordered using the first two hops. Consider any routing tree that has the center node r as its root, for example $aa'0, cbb'0, bb'0$ (we can assume that the inner nodes all go directly to the center). Since node b prefers path $baa'0$ to $bb'0$, it will switch to $baa'0$ without creating a cycle. But now c takes the path via a, which is less preferred than the path $cc'0$, so c will switch to this path. But now a will prefer to switch to the path $acc'0$, which will force b to switch to $bb'0$, which will in turn force c to switch back to $cbb'0$. This process clearly continues for ever, and for any possible routing tree. Therefore the bad triangle has no stable routing tree to its center. However, it is important to note that if we are using the bad triangle as part of a larger network and at least one of a, b, c has an alternative path to the centre (for example, using some external path) that is more preferred than any path in the bad triangle then we say that this 'breaks' the bad triangle. It can be seen that this allows *all* other nodes in the triangle to have paths to the centre node using edges only from the bad triangle.

We can now prove our main result for two-hop routing preferences.

Theorem 3.2.1 *Let $\mathcal{A}$ be a routing algebra for two-hop preferences. Then $\mathcal{A}$-SPP-SOLVABILITY is NP-complete.*

Proof. The proof is by a reduction from the 3-SAT problem, which is known to be NP-complete [GJ90]. Given a 3CNF formula, we shall take the clauses and encode them into bad triangle gadgets, one for each clause (for this reduction we shall ignore the nodes a', b', c' in the bad

triangle). We shall then take these gadgets and connect them to the destination via a long chain of nodes. The network is shown in Figure 3.5. The clauses are encoded by the edges between the bad triangles and the y_i nodes as follows. For each node v in a bad triangle, there is an edge (v, y_i) iff v represents either of the literals x_i or $\overline{x}_i$. Assume v represents the literal x_i. Then v prefers any path through the bad triangle to any path containing $\overline{x}_i$, and vice-versa if v represents the literal $\overline{x}_i$. We claim that there is a solution iff the 3-CNF formula f is satisfiable.

Let (v_1, v_2, v_3, c) be a bad triangle, with node c at the centre (the bad triangle can be constructed using 2-hop preferences). Assume that v_1 represents the literal x_i, v_2 is x_j and v_3 is x_k. Then v_3's path preferences are as follows (remembering that we can only order paths based on the first two hops):

$$v_3\ y_k\ x_k > v_3\ v_2\ c > v_3\ c\ \cdot > v_3\ y_k\ \overline{x}_k > v_3\ v_1\ y_i > v_3\ v_2\ y_j$$

and similarly for v_1, v_2 (using the bad triangle preferences). The order expresses that v_3 prefers to route through x_k than to route through the bad triangle (the second and third items), which are preferred to routing through $\overline{x}_k$. The last item says that v_3 prefers to go through the bad triangle than to escape through the other y_i nodes.

Claim 1: Any satisfying assignment for f gives a solution to the SPP instance.

Proof. Assign the y_i nodes paths consistent with the satisfying assignment, as in [FKMS05]. Then each bad triangle will have at least one node that has its most preferred path through the y_i's, so the other nodes can then route through their bad triangles. □

Claim 2: Every solution to the SPP instance gives a satisfying assignment for f.

Proof. We will prove that if there are no satisfying assignments then there is no solution. If f has no satisfying assignment then there is no assignment of paths to the y_i nodes such that every bad triangle has at least one node with its top preference path available (since f is unsatisfiable). Assume now that in some bad triangle, some node v_3 breaks up the triangle by going via its y_i node. Then the other two nodes v_1, v_2 will prefer to route via the bad triangle rather than escape via v_3 and y_i. But then v_3 would now prefer to go via the bad triangle. Hence no bad triangle will be broken up and there is no solution. □

The above claims establish that f is satisfiable iff the SPP instance is solvable. □

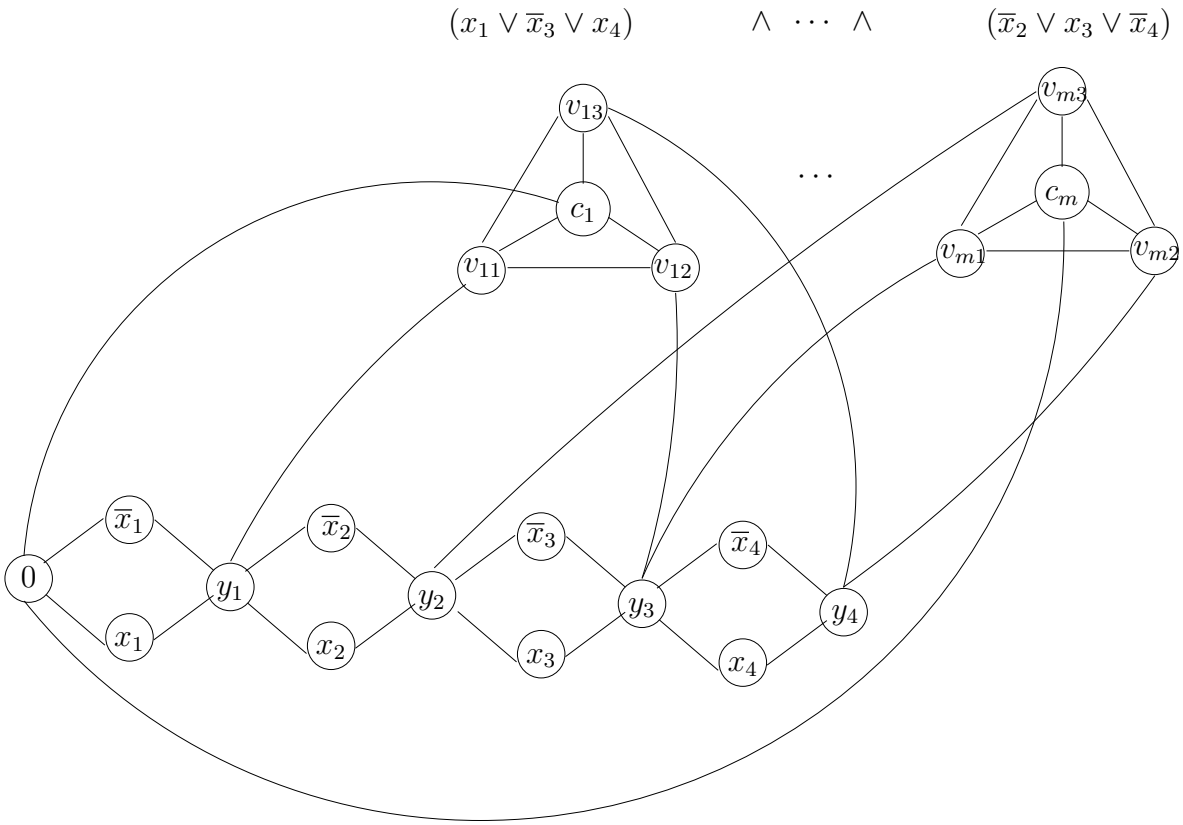


Figure 3.5: Encoding a 3-SAT instance into an SPP instance using 2-hop preferences. The SPP is solvable iff the formula is satisfiable.

Remark. The original 3-SAT reduction in [GSW02, Theorem V.1] also only uses two-hop preferences.

The above result raises several interesting questions: *why* does changing from next-hop to two-hop preferences result in a completely different character of problem, and in general, what makes deciding solvability of some stable paths problems hard and others easy? This is an open problem, and something that is outside the scope of this thesis.

3.3 Forbidden-set routing

There are many natural and desirable classes of policy that cannot be expressed using next-hop preferences, or even using two-hop preferences. For example, the government of country X may want to avoid any path that goes through some other country Y, perhaps because X is afraid that Y may do bad things to its packets, or because it does not want Y to know who it is communicating with. Another example is that nodes may have economic agreements to not forward traffic for each other, and so they should avoid paths passing through each other's networks. This motivates the *forbidden-set routing* problem, introduced by Feigenbaum et al. [FKMS05]: each node u has a forbidden set $S(u) \subseteq V(G)$ of nodes, and the cost $c_u(P)$ of a path P from u is the number of nodes it contains from $S(u)$, i.e. $c_u(P) = |S(u) \cap P|$. In addition to being an interesting class of routing policies, the problem is interesting from a graph theory point of view, since there is no path P from u to v with $c_u(P) = 0$ iff $S(u)$ separates u and v in G.

We shall begin by presenting a routing algebra for the class of forbidden-set preferences, in Figure 3.6. Applying the algebra to a graph G can be described as follows – for a directed edge $e = (x, y)$, let $c_u(e) = 1$ iff $v \in S(u)$, and 0 otherwise. A label $l \in L$ assigned to an edge e contains two things – the first endpoint of the edge and a vector describing the cost $c_u(e)$ of the edge e to each node u in the network. We shall use this vector to add up the cost of a path to each individual node, then finally project out the component that we are interested in. A signature $\sigma \in \Sigma$ contains the first node on the path described by σ, and the cost of the path to each node. The operator $\oplus$ accumulates the costs by doing component-wise addition on the cost vectors, and sets the new first node on the path. Finally, the function f projects out the element of the cost vector corresponding to the current first node on the path.

In their paper, Feigenbaum et al. [FKMS05] considered a similar problem but using costs taken from $\{0, 1, 2\}$. They showed that deciding solvability of the resulting SPP is NP-complete.

$$\begin{aligned} L &= \{1,\ldots,n\} \times \{0,1\}^n \\ W &= (\mathbb{Z}, \leq) \\ \Sigma &= \{1,\ldots,n\} \times \mathbb{Z}^n \\ (u,c) \oplus (v,d) &= (u, c+d) \\ f((v,d)) &= d_v \end{aligned}$$

Figure 3.6: A routing algebra $\mathcal{FS}$ for the forbidden-set routing problem

We now show how a simple extension to this result shows that $\mathcal{FS}$-SPP-SOLVABILITY is also NP-complete, and hence constructing stable routing trees for forbidden-set preferences is an intractable problem for general graphs.

3.3.1 Forbidden-set solvability is NP-complete

In this subsection we show that $\mathcal{FS}$-SPP-SOLVABILITY is NP-complete by a reduction from Π_2-SAT. In the second part of the chapter we use this reduction to characterise the communication complexity of deciding solvability by proving a lower bound on the communication complexity of deciding Π_2-SAT. Recall that for each family of boolean circuits, there is an associated satisfiability decision problem:

Problem Π_2-SATISFIABILITY
Input: A Π_2 formula f, given as a circuit.
Output: Is f satisfiable?

The above problem is known to be NP-complete [GJ90]. We now show how to encode Π_2-SAT into forbidden set routing preferences. We make use of the bad triangle presented earlier for the two-hop preferences reduction. Feigenbaum et al. [FKMS05] show how the bad triangle can be expressed using fs-preferences: set $S(a) = \{a', b, b'\},\ S(b) = \{b', c, c'\}$ and $S(c) = \{c', a, a'\}$. It can be verified that this corresponds to the bad triangle constructs for the case of two-hop preferences earlier, so it follows that this small network also has no stable solution.

Theorem 3.3.1 *$\mathcal{FC}$-SPP-SOLVABILITY is NP-complete.*

Proof. The proof is by reduction from 3-SAT. The proof is essentially that of Feigenbaum et al., except that we show that we only need costs in $\{0,1\}$ instead of $\{0,1,2\}$. We feel that the

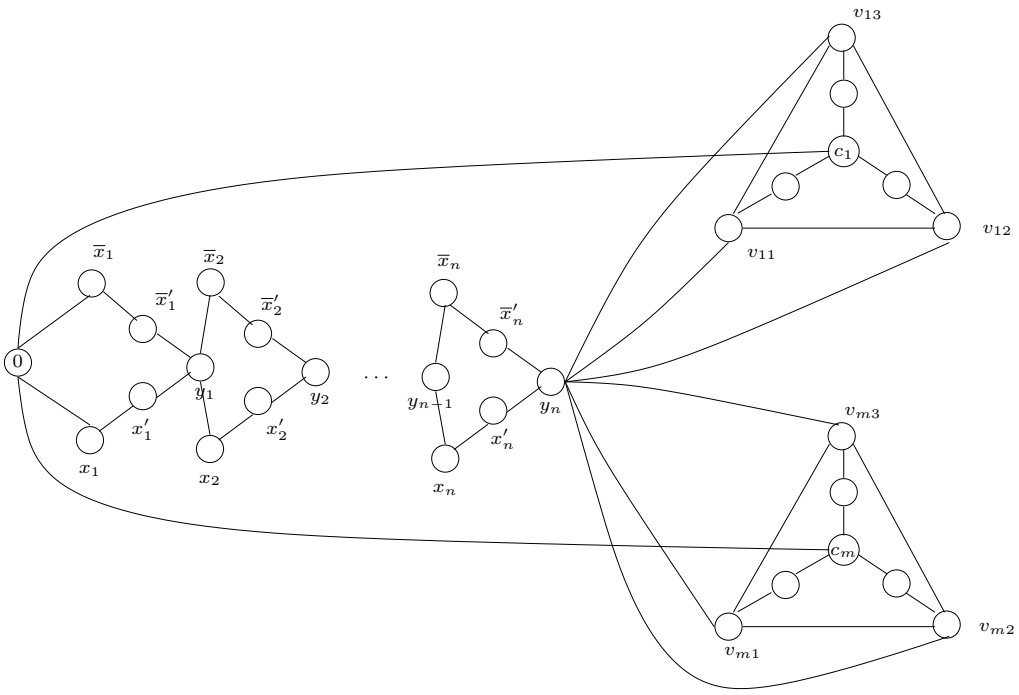


Figure 3.7: The reduction from 3-SAT to $\mathcal{FS}$-SPP-SOLVABILITY

proof is quite important, therefore we state it in full. Given a set of variables $\{x_1, \ldots, x_n\}$ and a set of clauses $\{C_1, \ldots, C_m\}$, where clause C_i contains three literals x_{i1}, x_{i2}, x_{i3}, we construct a stable paths instance with fs-preferences, which is solvable iff the 3-CNF formula $\phi = (x_{11} \vee x_{12} \vee x_{13}) \wedge (x_{21} \vee x_{22} \vee x_{23}) \wedge \ldots \wedge (x_{m1} \vee x_{m2} \vee x_{m3})$ is satisfiable.

The network is constructed as in Figure 3.7 with the destination node 0. Each clause C_i is represented by a bad triangle as described above, with the three outer nodes v_{i1}, v_{i2}, v_{i3} representing the three literals of the clause. If v_{ij} corresponds to the literal x_k, then $\{\overline{x}_k, \overline{x}'_k\}$ is in v_{ij}'s forbidden set (this corresponds to setting v_{ij}'s subjective cost of $\overline{x}_k$ to 2 in [FKMS05]). If it corresponds to $\overline{x}_k$, then $\{x_k, x'_k\}$ is in v_{ij}'s forbidden set.

We now show that any stable solution gives a satisfying assignment to the formula. In this case, the assignment shall consist of all the literals on the path from y_n to 0 in the stable solution. Since no path can contain both x_i and $\overline{x_i}$, and every stable solution contains a path through the y_i nodes, the assignment constructed in this way is valid. Now we show that it is also a satisfying assignment. Since each bad triangle is unsolvable, there must be at least one node in each bad triangle that has a route to 0 through the y_i nodes. Each node v_{jk} could always

route to 0 through its centre c_j, using a path of cost 1, and so it only routes through the y_i if this is a path of zero cost, which is the case only if the literal corresponding to v_{jk} is true. Since there exists such a node for every bad triangle, every clause is satisfied and so the assignment is a satisfying assignment.

We now show that any satisfying assignment to the formula gives a stable solution. We can find this stable solution by constructing a path using the true literals of the assignment, from y_n to 0. We then assign to all nodes v_{jk} corresponding to a true literal the long route through y_n. This route has zero cost to v_{jk}, and so it cannot strictly prefer any other route. Since the satisfying assignment has at least one true literal in each clause, at least one node in every bad triangle has a path of zero cost from y_n to 0, and so each bad triangle is 'broken up', leaving the remaining nodes to route to r on stable paths through their bad triangle centres.

□

3.3.2 NP-completeness on bounded treewidth graphs

We now show that deciding $\mathcal{FS}$-SPP-SOLVABILITY is NP-complete on graphs of bounded treewidth. This implies that forbidden-set routing using routing trees is almost certainly impractical even for very restricted classes of graphs that appear in practice (for example, even if we wanted to do fs-routing only on the Internet backbone and if the backbone was strongly tree-like). This shows that the problem has a very difficult core; for comparison, many NP-complete problems such as maximum independent set are solvable in linear time on bounded treewidth graphs.

Theorem 3.3.2 *$\mathcal{FS}$-SPP-SOLVABILITY is NP-complete on graphs of treewidth at least 7.*

Proof. Figure 3.8 shows a suitable tree decomposition of the graph used in the reduction of Theorem 3.3.1 with treewidth 7, and this completes the proof (the definition of treewidth can be found in the preliminaries in Chapter 2). □

We now pause to consider the results of this chapter so far. The main message is that even with simple tree-like networks (treewidth at most 7) and simple policies (forbidden-set), the problem of deciding if there exists a stable routing tree remains NP-complete.

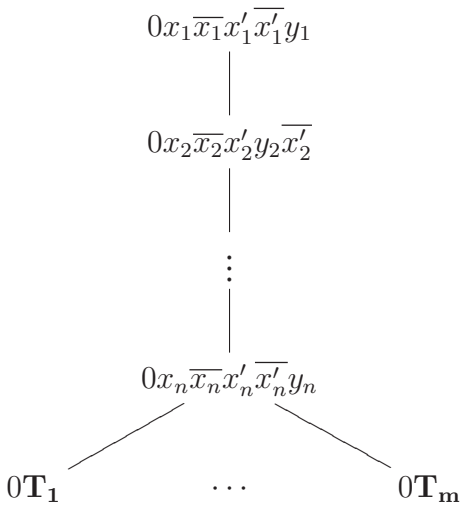


Figure 3.8: A tree decomposition of the forbidden set reduction graph. The $\mathbf{T_i}$ represents the nodes of the ith bad triangle, each containing 7 nodes. Since each bag contains at most 8 nodes from G, the graph has treewidth at most 7.

3.4 Communication complexity of solvability

In this second part of the chapter, we shall consider the stable paths problem as a problem in distributed computing, and prove the first communication complexity lower bounds for it. We then describe the notion of proof labeling schemes, which provide a distributed representation of a solution that is locally verifiable. We prove a lemma that lets us use our communication lower bounds to give lower bounds on the proof size of deciding solvability of a stable paths problem.

We begin by proving a communication complexity lower bound for SPP-SOLVABILITY. The lower bound relies on a construction that gives a large set of long stable paths, and this is based on a recursive construction of the DISAGREE gadget that was used by Griffin et al. [GSW02] in their original reduction from 3-SAT to SPP-SOLVABILITY. The idea of the construction is as follows. The DISAGREE gadget has nodes $x, \overline{x}$ and the root 0. Both x and $\overline{x}$ prefer to go through each other to reach the root, but are also happy to go direct to the root. Hence there are exactly two stable states. We say that the gadget is in the configuration x if all paths to 0 pass through x, and in the configuration $\overline{x}$ if all paths to 0 pass through $\overline{x}$, as in Figure 3.9.

The gadget j-DISAGREE is constructed by joining together j DISAGREE gadgets as fol-

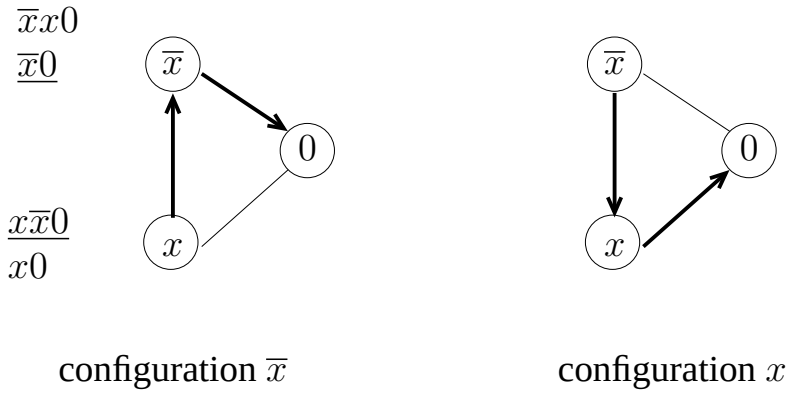


Figure 3.9: The two stable states of the DISAGREE gadget.

lows. Let 1-DISAGREE be equal to DISAGREE, and form j-DISAGREE by adding a copy of DISAGREE on $\{x_j, \overline{x}_j\}$ to $(j-1)$-DISAGREE and using the node x_{j-1} in place of the root node. We will say that the nodes $x_1, x_2, \ldots, x_j$ form the *spine* of j-DISAGREE. Figure 3.10 shows 3-DISAGREE.

Now we define the policies of the nodes in j-DISAGREE. Let $\Sigma_{x_1} = (x_1\overline{x}_1 0, x_1 0)$ and $\Sigma_{\overline{x}_1} = (\overline{x}_1 x_1 0, \overline{x}_1 0)$, as for DISAGREE. Recursively define Σ_{x_i} and $\Sigma_{\overline{x}_i}$ as follows:

$$\Sigma_{x_i} = \left(x_i\overline{x}_i\Sigma_{x_{i-1}}, x_i\Sigma_{x_{i-1}}\right) \quad \Sigma_{\overline{x}_i} = \left(\overline{x}_i x_i\Sigma_{x_{i-1}}, \overline{x}_i\Sigma_{x_{i-1}}\right)$$

where $x_i\Sigma_{x_{i-1}}$ is the order obtained by prefixing all the paths in $\Sigma_{x_{i-1}}$ by x_i and then adding them in their original order. The construction is a recursive extension of DISAGREE, where x_i prefers all paths to 0 that pass through $\overline{x}_i$ (in the order that x_{i-1} prefers them) to those not passing through $\overline{x}_i$ and vice-versa. The next lemma proves the main property of this construction.

Lemma 3.4.1 *The SPP defined by n-DISAGREE has 2^n distinct stable states where each path has length n.*

Proof. We show an injection between the powerset of $\{1, \ldots, n\}$ and the set of stable path assignments to nodes in T_n (the function is actually a bijection but we do not require this). For a set $X \subseteq \{1, \ldots, n\}$, if $i \in X$ then assign the ith DISAGREE gadget of the construction the configuration x, otherwise assign it the configuration $\overline{x}$ (see Figure 3.9). It is clear that this path assignment forms a tree rooted at the node 0, that each path through the structure has length at least n, and that there are 2^n distinct assignments (the set of configurations of the gadgets).

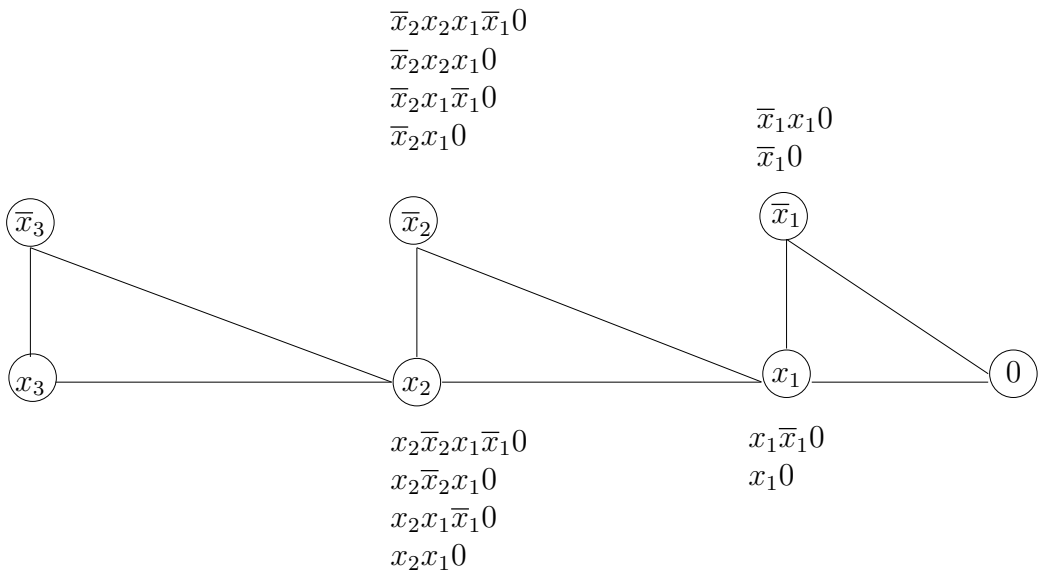


Figure 3.10: The construction 3-DISAGREE. The path preferences are recursively constructed from right to left, with the most preferred path at the top of each list.

We show that every such assignment is stable by induction on the length of the spine. The assignment to 1-DISAGREE is stable since it is just the gadget DISAGREE. Assume the assignment π to j-DISAGREE specified by the injection is stable. Assume that the (j+1)th DISAGREE gadget is placed in the configuration x (the case $\overline{x}$ is similar). Then node x_{j+1} is assigned the path $x_{j+1}\pi(x_j)$ and node $\overline{x}_{j+1}$ is assigned the path $\overline{x}_{j+1}x_{j+1}\pi(x_j)$. Since $\pi(x_j)$ is part of a stable assignment, the only thing that could make the new assignment unstable is for the (j+1)th gadget to switch to configuration $\overline{x}$, but this cannot happen as the assigned configuration x is already stable. □

We are now ready to prove the lower bound by an approximability-preserving reduction from set-disjointness. We will make use of the bad gadget [GSW02], as shown in Figure 3.11. The useful property of bad gadget is that it has no stable path assignment, and hence no solutions to the stable paths problem on it.

Figure 3.12 shows the network used for the lower bound. It is built by taking the bad gadget and adding a path so that if the sets are not disjoint then the bad gadget can be broken up, and then SPP becomes solvable.

We shall encode a large set into the path preferences Σ_C of node C as follows. For some set $\mathcal{X} \in 2^{\{1,\ldots,n\}}$ and an element $X \in \mathcal{X}$, define the path P_X as $P_X(i) = x_i$ if $i \in X$ and

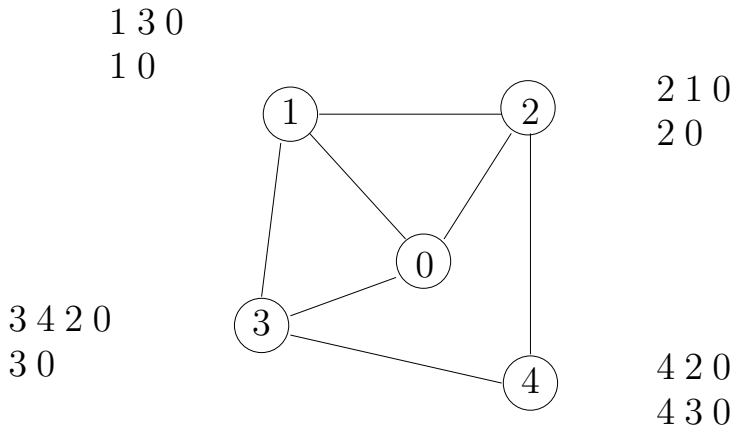


Figure 3.11: The construction bad gadget

$P_X(i) = x_i\overline{x}_i$ otherwise[1]. We encode $\mathcal{X}$ by adding for every $X \in \mathcal{X}$ the path $C(P_X 0)$ to Σ_C, in any order. The set of paths at C includes all the paths going to the root node 0, including those not in the set X. However, C prefers the path $C0$ (avoiding n-DISAGREE) over any path that passes through n-DISAGREE but represents an element not in X.

The node y_3 encodes a set $\mathcal{Y}$ in a similar way – for every $Y \in \mathcal{Y}$ we add the path $y_3 C P_Y 0$. In addition, we add to Σ_{y_3} the bad gadget paths $y_3y_4y_2y_00$, y_3y_00, y_3C0 and the empty path ϵ in that order, so that the paths corresponding to $\mathcal{Y}$ are preferred to any path that goes through the center y_0 of the bad gadget and y_1 prefers to go through the bad gadget than taking the shortcut path $C0$.

The idea is that if there is no valid assignment π that is stable at node y_3 then the system reduces to bad gadget, hence there is no assignment that is stable at y_3, hence no assignment is stable. The next lemma completes the proof by giving the reduction from the set-disjointness problem.

Lemma 3.4.2 *Consider the reduction (as in Figure 3.12), for two arbitrary sets $X, Y \subseteq 2^{\{1,\ldots,n\}}$. The sets X, Y are not disjoint iff there exists a stable path assignment.*

Proof. We first prove the $\Leftarrow$ direction: every solution gives a counterexample to disjointness. Let π be a solution, i.e. a stable assignment where $\pi(u)$ is the path assigned to node u. Then $\pi(y_3)$ must pass through the DISAGREE gadget (otherwise π would not be a solution since one side of the network would reduce to bad gadget). Therefore, C must have a path through the

[1]We use 2^S to denote the powerset of S

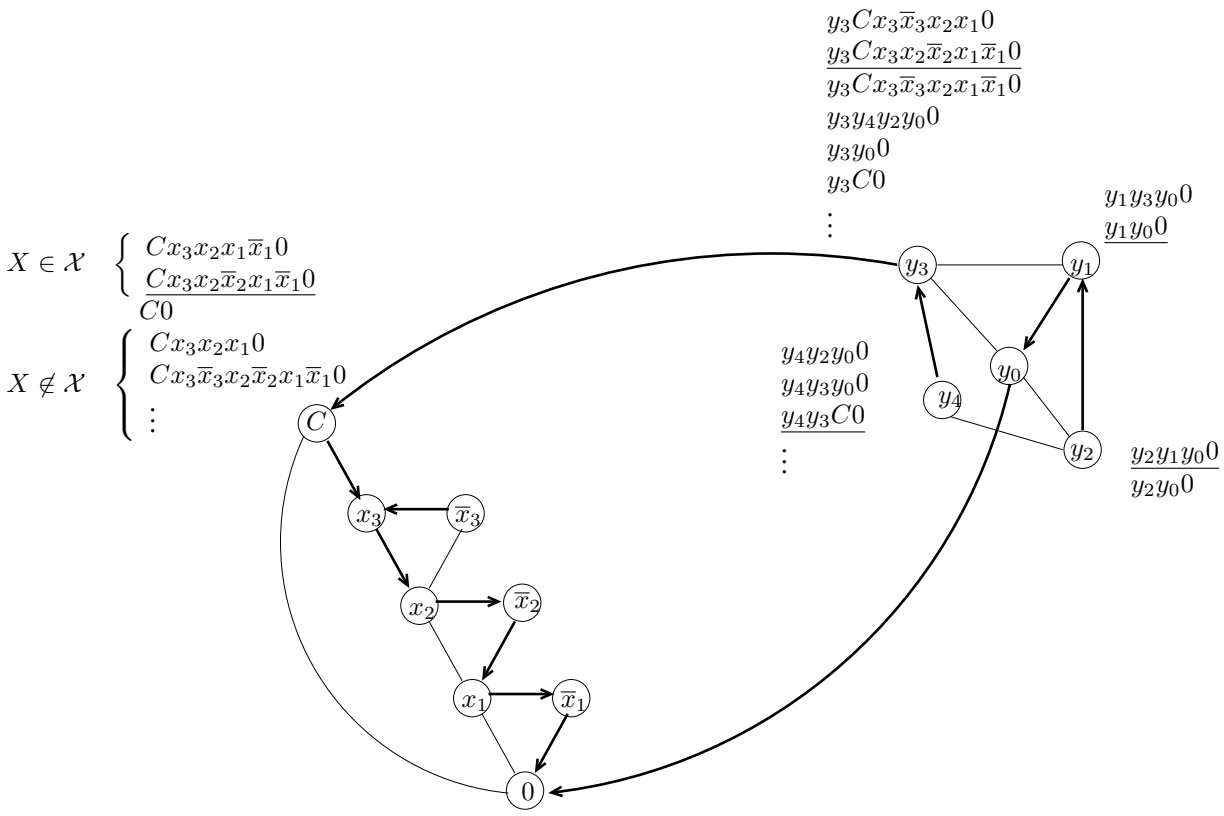


Figure 3.12: The reduction from set-disjointness to SPP-SOLVABILITY, for $n = 3$ and the sets $X = \{\{2,3\},\{3\}\}, Y = \{\{1,2\},\{3\},\{2\}\} \subseteq 2^{\{1,2,3\}}$. A stable solution is shown by the bold edges.

DISAGREE gadget that it prefers to the path $C0$. Hence the paths $y_3\pi(C)$ and $\pi(y_3)$ must be equal, and this represents an element in the intersection of the two sets.

Now consider the $\Rightarrow$ direction: not-disjoint implies there exists a solution. If the sets are not disjoint then there must exist some (not necessarily stable) path assignment such that $y_3\pi(C)$ and $\pi(y_3)$ are equal. By Lemma 3.4.1, there exists a path assignment that is stable at all the nodes in n-DISAGREE that is consistent with $\pi(C)$. Hence a stable solution π exists. □

We say that a distributed algorithm *solves* a problem if it terminates in finite time with at least one node knowing the answer. We do not need to consider the communication model in our lower bounds (and so they hold for both synchronous and asynchronous models); all we require is that some node knows the answer. In this sense, our bounds are purely information-theoretic. In the following proof, we require that all the nodes in one part (corresponding to either Alice or Bob) know the answer, but it also holds for the case when a single node knows the result, since it can broadcast the result using only $O(n)$ bits of communication. We can use Lemma 3.4.2 and the communication complexity of set-disjointness [KN97] to prove the following.

Theorem 3.4.3 *Any distributed algorithm that computes SPP-SOLVABILITY with probability at least $2/3$ must send at least $\Omega(2^{n/2})$ bits across $O(1)$ links in the worst case.*

Proof. Let there be two players Alice and Bob where Alice knows only the set P and Bob knows only the set Q. Partition the network into two parts by cutting the edges $\{C, y_1\}, \{y_0, 0\}$. Now Alice and Bob can respectively construct their parts of the network knowing only their set. Now assume that Alice and Bob run a protocol that decides SPP-SOLVABILITY with probability p. We show that they can solve set-disjointness on sets of size 2^n, with the same probability p. It is known that any protocol that solves disjointness on sets of size r with probability at least $2/3$ must use $\Omega(r)$ bits [KN97]. Since the construction of Figure 3.12 contains $2n + O(1)$ nodes, this gives a lower bound of $\Omega(2^{n/2})$ for networks of size n. □

Remarks. The theorem implies that any distributed algorithm must incur high congestion, since an exponential number of bits must be sent over a constant number of edges. Assuming that it takes one unit of time to send one bit, we also get a strong lower bound on the time to solve the problem, since the cut between the two parts of the network is of constant size (even assuming that all nodes are computationally unbounded). Note that if messages are of unbounded size the number of rounds required is constant, since each node can simply send its entire list of preferred paths.

3.4.1 Communication complexity of $\mathcal{FS}$-SPP-SOLVABILITY

In this subsection we prove a lower bound on the communication complexity of deciding solvability when forbidden-set preferences are used. We first extend the reduction of Theorem 3.3.1 to encode satisfiability of Π_3 formulae. The original construction only encoded Π_2 formulae, but it can be easily verified that if we use forbidden sets of size $2k$, then this corresponds to adding a level of AND gates at the bottom level of the circuit, i.e. Π_3^k formulae. Without affecting the solvability of the construction, we can partition its nodes into two sides A and B, with the chain of x_i's and y_i's in A, and half the bad triangles in A and the other half in B. We add extra nodes $y'_n, 0'$ in B and edges $\{y'_n, y_n\}, \{0', 0\}$, with all the bad triangles in B connecting to y'_n instead of y_n. This ensures that the cut (A, B) contains at most two edges.

Now, any balanced partition of the clauses between two players Alice and Bob corresponds to a balanced partition of the bad triangles as above. Therefore, any distributed algorithm that computes $\mathcal{FS}$-SPP-SOLVABILITY must communicate at least $\Omega(g(n))$ bits across the cut (A, B), where $g(n)$ is the communication complexity of Π_3^k-SAT.

Finally, we can replace the two edges in the cut (A, B) by two lines each of $n/2$ edges without affecting either the solvability of the construction, or the forbidden sets required. Dietzfelbinger [Die97] proves a version of the linear array conjecture implying that asymptotically, no distributed algorithm can do any better than to simply use these new nodes as relays, and therefore must send at least $\Omega(g(n))$ bits over at least $\Omega(n)$ edges. We shall show that $g(n) = \Omega(n \log n)$ in the deterministic case, which implies the following lower bound.

Lemma 3.4.4 *Any deterministic distributed algorithm that computes $\mathcal{FS}$-SPP-SOLVABILITY must communicate at least $\Omega(n \log n)$ bits over $\Omega(n)$ edges in the worst case.*

To prove this lemma, we now prove a lower bound on the communication complexity of deciding Π_3^k-SAT. Consider some Π_3^k formula $f = f_1 \wedge f_2$ on n variables with at most n clauses, and give Alice f_1 and Bob f_2. Nothing appears to be known about the communication complexity of deciding satisfiability of f.

In fact, we conjecture that in the deterministic case, nothing can beat the simple protocol of Bob sending his whole formula f_2 to Alice. Since each clause of a Π_2^k formula can be described with $\log \binom{2n}{k} = O(k \log(n/k))$ bits, this simple protocol uses at most $O(nk \log(n/k))$ bits.More precisely, we conjecture that the following holds.

Conjecture 3.4.5 *Let P be any two-party deterministic protocol that decides satisfiability of any Π_3^k formula on n variables with at most n gates, where the top gate is an AND, and the gates have unbounded fan-in except for the bottom level, which have fan-in at most k. Then there exists a formula $f = f_1 \wedge f_2$ such that P communicates at least $\Omega(nk \log(n/k))$ bits on the input (f_1, f_2).*

Conjecture 3.4.6 *Let P be as above but for Π_2^k formulae on n variables with at most n gates, i.e. in k-CNF. Then the same lower bound as above holds.*

We can prove the conjectures in the case that f is a Π_2^2 formula with n variables and n clauses. For larger $k > 2$, the problem Π_2^k-SAT is at least as hard as Π_2^2-SAT and so the same lower bound applies. Since the trivial protocol uses communication linear in k, this means that the trivial protocol is asymptotically optimal for deciding Π_2^k-SAT with constant k.

We now state the lower bound in terms of the number of gates and wires of a boolean function as this leads to an appealing way of describing the current bounds. Any boolean function with m wires and n gates can be described using at most $cm \log n$ bits for some constant c, yet we can only show a lower bound of $\Omega(n \log n)$ bits. Therefore, this gap is due to some gates being connected to many wires (which occurs when the fan-in is large). Note that Π_2^k formulae with large values of k have a large number of wires.

Lemma 3.4.7 *Let P be a deterministic two party protocol for deciding satisfiability of a formula with m wires and n gates. Then there exists a formula $f = f_1 \wedge f_2$ such that P communicates at least $\Omega(n \log n)$ bits on the input (f_1, f_2).*

Proof. We give two proofs of the lemma; the first one can only prove a lower bound of $\Omega(n \log n)$ bits but is simpler to state, and the second proof is more general and may help to prove the conjecture in more general cases.

For the first proof, we appeal to the $\Omega(n \log n)$ bits communication lower bound for deciding st-connectivity, which was proved in [HMT88]. We can reduce from st-connectivity to 2-SAT as follows:

- Make one variable for each node in the graph.
- If the edge (u, v) is in the graph, include the clause $(u \Rightarrow v)$, i.e. $(v \vee \overline{u})$.
- Also, include the clauses $(s) \wedge (\overline{t})$.

Then the formula is satisfiable if and only if there is no path from s to t in the graph. The lower bound in [HMT88] holds for sparse graphs (where the number of edges is linear in the number of nodes). It can be seen that the reduction described above produces formulae with a number of clauses linear in the number of edges in the graph. It follows from the result of Hajnal et al. [HMT88] that we can find a constant c such that the communication complexity of 2-SAT is at least $cn \log n$, where the formulae have at most n variables and at most n clauses (and so they have at most n gates and $c'n$ wires, for some constant c').

The above lower bound is for 2-SAT, and we want to get a stronger bound for k-SAT, where the number of wires is k times the number of gates. With this in mind, we give a reduction from a partition problem that we now describe.

A *partition* of a set S is a set $\{S_1, \ldots S_k\}$ of disjoint subsets, called *blocks*, of S whose union is S. We say that a partition P *refines* a partition Q iff every block of P is contained in some block of Q, i.e. $\forall P_i \in P$, $\exists Q_j \in Q$ such that $P_i \subseteq Q_j$. The *join* of two partitions P, Q is denoted $P \vee Q$, and is the finest partition R such that both P and Q are refinements of R (i.e. R refines every partition R' that is also refined by both P and Q). The problem PARTITION is as follows:

Problem PARTITION:
Input: Two partitions P, Q of $\{1, \ldots, s\}$
Output: Are elements 1,2 in the same block in the partition $P \vee Q$?

We can construct a reduction to Π_2^2-SAT as follows. Alice has a partition P and Bob has a partition Q. Given a partition $P = \{P_1, \ldots, P_k\}$ of $\{1, \ldots, n\}$, order its blocks (the order can be chosen arbitrarily) and let $P(i) = j$ iff $i \in P_j$. Alice then constructs a formula f_1 as follows. For each $i \in \{1, \ldots, n\}$ with $P(i) = j$ then add the two clauses $(\overline{y}_i \vee x_j) \wedge (\overline{x}_j \vee y_i)$ to f_1. Bob constructs a formula f_2 in the same manner with the partition Q, but uses variables z_i in place of the x_i. Let $f = f_1 \wedge f_2$, then f is a 2-CNF formula and f_1, f_2 can each be constructed with no communication between Alice and Bob. Now, the elements 1,2 are in the same block in $P \vee Q$ iff the formula $(f \wedge y_1 \wedge \overline{y}_2)$ is unsatisfiable.

The idea of the construction can be easily explained with an example. Consider the two partitions $P = \{\{1, 3\}, \{2, 4\}\}$ and $Q = \{\{1, 4\}, \{2\}, \{3\}\}$. We construct the formulae as

follows:

$$\begin{aligned} f_1 &= (\overline{y}_1 \vee x_1) \wedge (\overline{x}_1 \vee y_1) \wedge (\overline{y}_3 \vee x_1) \wedge (\overline{x}_1 \vee y_3) \\ &\quad \wedge\ (\overline{y}_2 \vee x_2) \wedge (\overline{x}_2 \vee y_2) \wedge (\overline{y}_4 \vee x_2) \wedge (\overline{x}_2 \vee y_4) \\ f_2 &= (\overline{y}_1 \vee z_1) \wedge (\overline{z}_1 \vee y_1) \wedge (\overline{y}_4 \vee z_1) \wedge (\overline{z}_1 \vee y_4) \\ &\quad \wedge\ (\overline{y}_2 \vee z_2) \wedge (\overline{z}_2 \vee y_2) \wedge (\overline{y}_3 \vee z_3) \wedge (\overline{z}_3 \vee y_3). \end{aligned}$$

Now we can test satisfiability of $f_1 \wedge f_2 \wedge y_1 \wedge \overline{y}_2$ by deciding whether 1,2 are in the same block in $P \vee Q$. The intuition is that a 'path' from element 1 to element 2 will force the variables y_1, y_2 to take the same value in any satisfying assignment of the formula. Since y_1 is true, this forces x_1 to be true, which forces y_3 to be true (since 1,3 are in the same block in P). Since y_3 is now true, this forces z_1 to be true (since the first block of Q also contains the element 1). This in turn forces y_4 to be true (since Q contains the block $\{1, 4\}$), which forces x_2 to be true. Finally, y_2 becomes true, which contradicts the clause $\overline{y}_2$ we already had, and so the formula cannot be satisfiable. It is useful to think of the x_i as communicating within the same formula f_i, and the y_i as communicating between the two different formulae.

It is known [HMT88, Reference [8]] that the deterministic communication complexity of deciding PARTITION on sets of size s is $\log(B_s - B_{s-1}) = \Omega(s \log s)$ where B_s is the sth Bell number. It follows that any two-party deterministic protocol that decides satisfiability of a 2-CNF formula on n variables and with $O(n)$ clauses, must communicate at least $\Omega(n \log n)$ bits. □

Combined with the reduction described above, this proves Lemma 3.4.4.

Remarks. Since a Π_2 formula is a special case of a Π_3 formula, with fan-in 1, the previous lower bounds immediately apply to Π_3^1 formulae. However, there is hope that we can prove the conjecture for Π_3 formulae, as they are more expressive than Π_2 formulae. We have tried to prove the conjecture for larger values of k but without much success.

The following idea may be helpful. With Π_3 formulae, we can express each block of a partition of $\{1, \ldots, n\}$ with a single clause, rather than using 1 clause per element (as in the Π_2-SAT reduction above). Therefore, a reduction from PARTITION to Π_3-SAT can be obtained as follows. Recall that the jth block of a partition P is denoted P_j. For each block $P_j \in P$, Alice

adds the following two clauses to f_1:

$$\left(x_j \vee \bigwedge_{k \in P_j} \overline{y}_k \right) \wedge \left(\overline{x}_j \vee \bigwedge_{k \in P_j} y_k \right).$$

Bob does the same for Q, using variables z_i in place of the y_i. The same claim holds as for the original reduction, except that we now only use two clauses per block of each partition, whereas the reduction to Π_2-SAT uses two clauses per *element*.

We can consider the nondeterministic communication complexity of the problem, which will be useful later. It is known [KN97] that the deterministic and nondeterministic communication complexities are related by $D(f) = O(N^0(f)N^1(f))$ (recall that D, N^0, N^1 are the deterministic, nondeterministic and co-nondeterministic communication complexities). Lemma 3.4.7 proves that $D(\Pi_2\text{-SAT}) = \Omega(n \log n)$, where the formula has at most n clauses and n variables. Observe that $N^1 = O(n)$ (the complexity of verifying that a formula is satisfiable) since a satisfying assignment (if one exists) can be described with $O(n)$ bits. Therefore, $N^0 = \Omega(n \log n)/N^1 = \Omega(\log n)$ bits.

3.4.2 A randomized lower bound

We can show an $\Omega(n)$ randomized lower bound for Π_2^k-SAT by a reduction from the set disjointness problem.

Lemma 3.4.8 *Let P be a two party randomized protocol for deciding Π_2^k-SAT. There exists an input such that P communicates at least $\Omega(n)$ bits on this input.*

Proof. Assume that $m = 2^l$ is a power of two. The proof is by reduction from set-disjointness on sets of size m. Assume that Alice has a set $P \subseteq \{0, \ldots, m{-}1\}$ and Bob has a set $Q \subseteq \{0, \ldots, m{-}1\}$, and there exists a randomized protocol P that computes $f(\phi_1, \phi_2) = 1$ if $\phi_1 \wedge \phi_2$ is satisfiable and 0 otherwise, where ϕ_1, ϕ_2 are Π_2^k formulae. Assume that the protocol has error probability p.

We can associate a subset $Y \subseteq \{0, \ldots, m{-}1\}$ with its characteristic Boolean function $f_Y : \{0,1\}^l \to \{0,1\}$ by setting $f(y) = 1$ iff $y \in Y$. Alice constructs the CNF formula ϕ_P for the truth table of f_P corresponding to the characteristic function of her set P by making a clause for every 0-entry in the table, i.e. at most $2^l = m$ clauses each of size l. This is done as follows: for an assignment of values to the y_i's such that $f(y_1, \ldots, y_l) = 0$, add a clause containing $\overline{x}_i$

if $y_i = 1$, and x_i if $y_i = 0$ (each clause is a disjunction of literals). Therefore, the clause is not satisfied iff the element does not appear in Y. So, for a setting of variables corresponding to an element in Y, all the clauses are satisfied. Therefore the formula $\phi_P \wedge \phi_Q$ is satisfiable iff there is an element common to both P and Q, i.e. they are not disjoint. Similarly, Bob constructs the CNF formula ϕ_Q corresponding to the characteristic function of his set Q.

This gives a protocol to decide disjointness with the same error probability p. The lemma follows since the randomized communication complexity of disjointness on sets of size n is $\Omega(n)$ bits [KN97]. □

3.4.3 Communication complexity of $(\Sigma_2^k \wedge \Sigma_2^k)$-SAT

As an interesting aside we can show that the communication complexity of deciding satisfiability of $(\Sigma_2^k \wedge \Sigma_2^k)$ formulae, i.e. the conjunction of two k-DNF formulae, is exponentially lower than for k-CNF formulae when $k = O(1)$. This is quite surprising.

We can decide satisfiability of a formula $f_1 \vee f_2$ (where Alice has f_1 and Bob has f_2) using a single bit, regardless of the complexity of f_1, f_2. It might be tempting to blame the high communication complexity on that fact that the formula f is split at a conjunction rather than a disjunction. We now show that this is not the case, by giving an efficient protocol for deciding satisfiability of $f_1 \wedge f_2$, where f_1, f_2 are each k-DNF formulae. The problem is as follows:

Problem $(\Sigma_2^k \wedge \Sigma_2^k)$**-SAT**
Alice's Input: A k-DNF formula f_1 on n variables $\{x_1, \ldots, x_n\}$ having $\leq n$ clauses.
Bob's Input: A k-DNF formula f_2 on n variables $\{x_1, \ldots, x_n\}$ having $\leq n$ clauses.
Output: Is $f_1 \wedge f_2$ satisfiable?

Lemma 3.4.9 *The deterministic communication complexity of* $(\Sigma_2^k \wedge \Sigma_2^k)$*-SAT is* $O(\log n)$*, for fixed* k*.*

Proof. We give a recursive protocol for the conjunction of two k-DNF formulae, for any constant k. We prove the existence of our protocol by inductively constructing a protocol for deciding satisfiability of the conjunction of two k-DNF formulae, by assuming that we have a protocol for satisfiability of the conjunction of two $(k-1)$-DNFs, which (inductively) satisfies our time bound. This will give a communication bound that depends exponentially on k.

A k-DNF formula f is *bad* iff there is a set of at most k variables so that every term of f contains as a literal one of these variables (appearing either negated or unnegated). If f is not bad, we call it *good*. We make use of the following lemma.

Lemma 3.4.10 *If f_1 is a good k-DNF and f_2 is a non-empty k-DNF, then $f_1 \wedge f_2$ is satisfiable.*

Proof. Pick an arbitrary term t of f_2. If $f_1 \wedge f_2$ is not satisfiable then every term u of f_1 must contain a variable from t (occurring negated in u iff it occurs unnegated in t). But then f_1 is bad. □

The protocol is as follows. We assume that Alice and Bob both remove any inconsistent terms in their formulae before beginning the protocol. Alice first checks if her formula is good. If it is, $f_1 \wedge f_2$ is satisfiable unless f_2 is empty, in which case it is unsatisfiable. They use $O(1)$ bits of communication to discuss this. We then do the corresponding check with Alice and Bob switching roles. So we can henceforth assume that both f_1 and f_2 are bad, so by the lemma there is a set S_1 of at most k variables, occurring in every term of Alice's formula and a set S_2 of at most k variables, occurring in every term of Bob's formula. Alice now sends S_1 to Bob and Bob sends S_2 to Alice, using $O(k \log n)$ bits of communication in total. They now run the protocol for $(k-1)$-DNFs on the 2^{2k} subproblems corresponding to fixing the variables in the set $S_1 \cup S_2$, trying all possible truth assignments. They output "satisfiable" if and only if one of these runs says "satisfiable".

For the communication complexity bound, let $C(n,k) = O(k \log n) + 2^{2k} C(n, k-1)$, which gives $C(n,k) = k^2 4^{k^2} \log n$. Therefore for fixed k the protocol uses $O(\log n)$ bits. □

3.5 Proof labeling schemes

Solvability of an SPP is a *global* property of the network, yet in a large network we would like to be able to verify that the assigned routing tree is indeed a solution, by using only *local* information. For example, if each node i is assigned a path $\pi(i)$, we would like to construct a distributed representation of π in order that we can *locally* and *verifiably* check if the path assignment π is stable. This is the idea of proof labeling schemes, which were introduced by Korman et al. [KKP05].

Imagine that there is some graph property P that we want to verify (e.g. can the current graph be coloured with k colours?) and that we have a candidate solution (e.g. a colouring of the nodes) that is encoded by giving each node a local state and a label. We assume that the decoder

algorithm, when run at a node v, can observe the state of v and the labels of v's neighbours. The decoder must be able to verifiably check if the property holds, i.e. the neighbours cannot convince a node that the property holds if in actual fact it does not.

3.5.1 Definition

We now define proof labeling schemes as in [KKP05]. A *marker algorithm* M is an algorithm that given a graph G, assigns a label $L(v)$ to each node v. For a marker algorithm and a node, let $Adj'_L(v)$ be a set of fields, one field per neighbour. Each field corresponding to an edge $e = (v, u)$ contains the label $L(u)$. Let $Adj_L(v) = \langle s_v, L(v), Adj'_L(v)\rangle$. Informally, $Adj_L(v)$ contains the labels given to all of v's neighbours, along with the edges connecting v to them. It also contains v's state and label $L(v)$.

A *decoder algorithm* D is an algorithm that can be run separately at each node. When D is run at a node v, its input is $Adj_L(v)$ and its output is denoted by $D(v, L)$. The idea is that the decoder algorithm, when run at a node v, can see v's state in addition to the labels for v and all its neighbours.

Let f be some boolean function over a family of graphs $\mathcal{G}$. A *proof labeling scheme* $\pi = (M, D)$ for f over $\mathcal{G}$ consists of a marker algorithm M and a decoder algorithm D, such that the following properties hold:

1. For every graph $G \in \mathcal{G}$, if $f(G) = 1$ then $D(v, L(M, G)) = 1$ for every node $v \in G$, where $L(M, G)$ is the labeling produced by M on G.

2. For every graph $G \in \mathcal{G}$, if $f(G) = 0$ then for *any* labeling L there exists a node $v \in G$ such that $D(v, L) = 0$, i.e. the property cannot be verified at some point in the network.

The *size* of a proof labeling scheme π is the maximum number of bits assigned to some label over all graphs $G \in \mathcal{G}$ and nodes $v \in G$. For a family $\mathcal{G}$ of graphs and a function f, the *proof size* of f on $\mathcal{G}$ is the smallest size of any proof labeling scheme for f on $\mathcal{G}$.

3.5.2 Proof size and communication complexity

We now prove a lemma that relates the size of any proof labeling scheme for a problem to the communication complexity of any protocol for the same problem, when played between two players. We combine this with our communication complexity results for the problem of

deciding solvability of a stable paths instance (SPP-SOLVABILITY) to obtain lower bounds for the size of proof labeling schemes for SPP-SOLVABILITY.

Let f be a boolean graph property on graphs $G \in \mathcal{G}$ for some family $\mathcal{G}$ (in our case later on, f will be the property 'is a particular routing tree on G stable?'). We shall partition the nodes of G between two players Alice and Bob, in order to construct the two-party communication problem associated with f and G. Let $(X, V \setminus X)$ be a (not necessarily balanced) partition of the nodes of G. Denote by $N_{(X,V\setminus X)}(f, G)$ the nondeterministic communication complexity of the best protocol for f on the family $\mathcal{G}$, when run on the graph G and this partition of nodes. To avoid confusion, we use $Adj(X)$ for the neighbours of X, and $\widetilde{A}dj(X) = Adj(X) \cap (V \setminus X)$ for the set of neighbours of nodes in X that are in the other side of the partition. We can now prove the main result of this section.

Lemma 3.5.1 *Let f be a graph property on a family of graphs $\mathcal{G}$. The proof size of f is at least*

$$\max_{G\in\mathcal{G}} \max_{X\subseteq V} \frac{N_{(X,V\setminus X)}(f, G) - O(1)}{|\widetilde{A}dj(X) \cup \widetilde{A}dj(V \setminus X)|},$$

and the total label size is at least $\max_{G\in\mathcal{G}} \max_{X\subseteq V} N_{(X,V\setminus X)}(f) - O(1)$.

Proof. Let $G \in \mathcal{G}$ be a graph. Let $L(v)$ be the label assigned to node v by the marker algorithm M, and let D be the corresponding decoder algorithm. Given a labeling of the nodes and a partition $(X, V \setminus X)$ of nodes, we construct a reduction showing how we can use a proof labeling scheme for f on G to construct a two party nondeterministic protocol to solve f on G, when the players are given nodes X and $V \setminus X$.

Alice is given the nodes X and Bob is given $V \setminus X$ as in Figure 3.13, and they each nondeterministically guess a labeling for their nodes. Note that Alice and Bob can independently run the decoder algorithm on the nodes in their part of the graph that have no neighbours in the other side. Therefore we can assume that $D(v, L) = 1$ for all the nodes in $(X \setminus \widetilde{A}dj(V \setminus X))$ and $(V \setminus X) \setminus \widetilde{A}dj(X)$, since Alice and Bob can discuss this using $O(1)$ bits of communication. Now they just need to run the decoder on the remaining nodes, as follows. Bob sends the labels $L(\widetilde{A}dj(X))$ to Alice who runs the decoder on the remaining nodes in X, and then Alice sends to Bob $L(\widetilde{A}dj(V \setminus X))$ who runs the decoder on the remaining nodes of $V \setminus X$. They can then discuss with $O(1)$ bits whether the decoder failed on any node of G, and hence compute $f(G)$.

The protocol that is described above has nondeterministic communication complexity at most $|L(\widetilde{A}dj(X))|+|L(\widetilde{A}dj(V\setminus X))|+O(1)$ on the graph G, since the labels are communicated

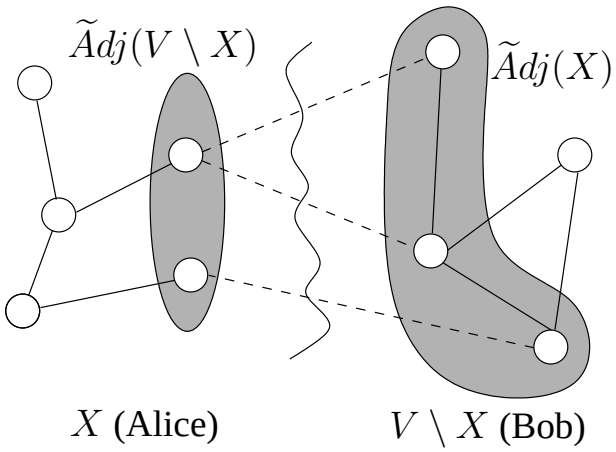


Figure 3.13: Illustrating Lemma 3.5.1

by simply sending their bit string representations. It follows that for any graph $G \in \mathcal{G}$ that is partitioned into $(X, V \setminus X)$,

$$\sum_{v \in V} |L(v)| + O(1) \geq |L(\widetilde{A}dj(X))| + |L(\widetilde{A}dj(V \setminus X))| + O(1) \geq N_{(X,V\setminus X)}(f, G),$$

which gives the lower bound on the total label size. For the individual label size, we can divide the above inequality by $|\widetilde{A}dj(X) \cup \widetilde{A}dj(V \setminus X)|$. Therefore at least one node in the set $\widetilde{A}dj(X) \cup \widetilde{A}dj(V \setminus X)$ must be assigned a label of size at least

$$\max_{X \subseteq V} \frac{N_{(X,V\setminus X)}(f, G) - O(1)}{|\widetilde{A}dj(X) \cup \widetilde{A}dj(V \setminus X)|}$$

bits. Finally, we take the maximum of this quantity over all graphs $G \in \mathcal{G}$, since the size of a proof labeling scheme is the maxmimum label size over all graphs $G \in \mathcal{G}$. □

Remarks. The bound on individual size can be improved if a smaller number of labels are sent, since the middle term in the inequality concerns fewer nodes and so the denominator can be made smaller. A better strategy may be possible in e.g. bounded treewidth graphs.

Intuitively, since the marker algorithm can examine the whole graph in order to construct the proof labeling, the marker algorithm can be thought of as playing the role of the 'all-powerful prover' in nondeterministic complexity [KN97]. In this way the bound on the label size follows naturally from the bound on the communication required for Alice and Bob to verify a

nondeterministic guess of a solution to $f(G)$.

The lemma implies that in order to get a good lower bound on the proof size of f, we should look for a partition involving few nodes having edges in both sides of the partition, yet where the function f still has high nondeterministic communication complexity on this partition.

3.5.3 Proof labeling schemes for solvability

We now apply the above lemma to some of our communication complexity lower bounds, to obtain lower bounds on the proof size for deciding solvability in networks.

Lemma 3.5.2 *The proof size of SPP-SOLVABILITY is $\Omega(n)$ bits.*

Proof. Theorem 3.4.3 shows that we can solve disjointness on $\Omega(2^{n/2})$-element sets by a reduction to SPP-SOLVABILITY. A closer look at the reduction shows that the SPP is solvable iff the two sets are not disjoint. Since the decoder algorithm can only decide locally if the graph property f does *not* hold (since the property holds globally iff there is no node where it does not hold locally, i.e. there is no node that has a proof of non-solvability), we need to consider the communication complexity of proving that two r-element sets are disjoint, i.e. the co-nondeterministic complexity $N^1(DISJ) = \Omega(\log r)$ bits [KN97].

Since the reduction in Theorem 3.4.3 is from sets of size $\Omega(2^{n/2})$, it follows from Lemma 3.5.1 that the total label size is at least $\Omega(n)$ bits. For the proof size, the network partition used in the reduction has at most $O(1)$ nodes having edges in both sides and so the proof size is also $\Omega(n)$. □

The lower bound is almost tight since we can construct a proof labeling scheme using $O(n \log n)$ bits per label. Assume that $T = \{\pi_i\}$ is a stable routing tree where the path π_v is assigned to node v. Let $p(v)$ be the parent of v in T ($p(v) = \epsilon$ iff v is the root or $\pi_v = \epsilon$, i.e. v is assigned the empty path). We construct the labels $L(v) = \pi(v)$, and the state $s_v = (c_v, \pi(v), p(v))$, i.e. v's cost function, v's path and the parent of v in T.

The decoder algorithm is given $Adj_L(v) = \langle s_v, L(v), Adj'_L(v) \rangle$ and outputs $D(v, L) = 1$ iff all of the following hold, and 0 otherwise.

1. $p(v)$ is a neighbour of v in G (for the root, assume that ϵ is a neighbour of v);

2. $v \circ L(p(v)) = \pi(v)$ (the paths form a confluent routing tree);

3. For all neighbours u of v in G such that $v \notin L(u)$, we have $c_v(L(v)) \leq c_v(v \circ L(u))$ (the path assignments are stable and switching cannot create a cycle).

Note that the function $c_v(\cdot)$ can be computed by consulting v's local state. We now claim that the scheme satisfies both properties of a proof labeling scheme:

Claim. Let π be a stable solution and let L be the labeling computed as above for π. Then there is no node v where $D(v, L) = 0$.

Proof. Assume that π is a stable solution and that there is some node v such that at least one of the three properties above fails to hold. Then we show that L, which equals π, is not a solution – a contradiction.

If the first property fails then the set of paths π does not exist in G and so π is not a solution. If the second property fails then the set of paths of π, which are the same as L, do not form a confluent routing tree. If the third property fails then there exists a neighbour u of v such that v would prefer the path via u, and switching would not create a cycle. Therefore if any of these properties fail to hold then the labeling L is not a solution. □

Claim. If the SPP is unsolvable then for any labeling L there exists a node v with $D(v, L) = 0$.

Proof. If the SPP is unsolvable then by definition there does not exist a stable solution π. Now consider any labeling L. We show that there is a node v such that $D(v, L) = 0$.

We can assume wlog that the labeling is a valid path assignment, i.e. the paths form a confluent rooted tree, rooted at 0. If this were not the case then some node would clearly fail at properties 1 or 2. Therefore we can assume that both properties 1 and 2 hold at all nodes. It remains to show that property 3 fails at some node v. Assume, to the contrary, that property 3 holds at all nodes under the labeling L. But then all nodes have a path (possibly the empty path) that they would not switch from. By definition, this is exactly a stable path, and so there must be some node v where $D(v, L) = 0$. □

It is important to note that the above lower bounds are independent of the distributed representation of a solution.

Proof size of unsolvability

We can also consider the problem of deciding *unsolvability* of an SPP, i.e. the function SPP-UNSOLVABILITY, which is the negation of the function for SPP-SOLVABILITY.

Lemma 3.5.3 *The proof size of SPP-UNSOLVABILITY is* $\Omega(2^{n/2})$ *bits.*

Proof. We use the same construction as for the previous lemma, but now the function is true iff the SPP is unsolvable iff the two sets are not disjoint. Therefore, the decoder outputs true iff no node has a proof of solvability. In this case, we need to consider the complexity of proving that two sets are not disjoint, i.e. the nondeterministic complexity $N^0(DISJ)$. It is known that $N^0(DISJ) = \Omega(r)$ [KN97] for r-element sets. Applying Lemma 3.5.1 as in the previous proof shows that the proof size is $\Omega(2^{n/2})$ bits. □

Intuitively, the proof size of unsolvability is so high because a node must be able to reject a false proof, which is at least as hard as verifying a proof of solvability of the SPP.

3.5.4 Forbidden-set preferences

For the forbidden-set routing algebras, Theorem 3.3.1 gives a reduction from Π_3-SAT where the formula is satisfiable iff the network is solvable. The discussion following Lemma 3.4.7 implies that $N^0(\Pi_3\text{-SAT}) = \Omega(\log n)$ bits. A similar argument to Lemma 3.5.2 shows that the proof size of $\mathcal{FS}$-SPP-SOLVABILITY is $\Omega(\log n)$ bits (the reduction in Theorem 3.3.1 can be modified to have $O(1)$ nodes having edges in both sides by adding two extra nodes $0'$ connected to 0, and y'_n connected to y_n, where $0, y_n$ are in one side and $0', y'_n$ are in the other side and all previous connections from the bad triangles to $0, y_n$ are now connected to $0', y'_n$ instead. This modification does not affect the solvability of the construction.) We have been unable to improve the $O(n)$ upper bound of the general protocol. Therefore it is open as to whether there exists a more compact distributed representation of a solution (and hence a better proof labeling scheme) for $\mathcal{FS}$-SPP-SOLVABILITY.

3.6 Discussion

At this point it is worth discussing the results for the three routing algebras we have considered: forbidden-set, two-hop and next-hop. First let us consider the complexity of deciding solvability – of all these, only next-hop routing does not give an NP-complete problem, but it is trivial as there always exists a solution. This gives the following open problem.

- Fully characterise the relationship between the algebraic properties of A and the computational (or communication) complexity of A-SPP-SOLVABILITY. The results of this chapter show only that some algebras generate hard instances (NP-complete, or exponen-

tial communication) and others generate easy instances (trivial).As yet, there is no known algebra A where A-SPP-SOLVABILITY is neither trivial nor NP-complete.

We conjecture that for any non-trivial routing algebra (an algebra is non-trivial if it can generate both solvable and unsolvable SPP instances), the problem of deciding solvability is NP-complete. In addition to the strong negative results in this chapter, this would be a strong argument against the use of stable routing trees for policy-based routing. Feamster et al.[FJB05] have also considered the additional problem of verifying if an iterative algorithm will converge on a collection of policies from *any* initial state. They call this property *safety* and show that any SPP that is both solvable and safe must have policies that are essentially equivalent to ranking based on path lengths.

Now we consider the communication complexity of deciding solvability. Let us call an SPP instance generated using the forbidden-set algebra $\mathcal{FS}$ a *sparse* instance if the size of each forbidden set is bounded, and *dense* otherwise. In the case of sparse forbidden-set instances, Lemma 3.4.4 implies that no deterministic distributed algorithm for deciding solvability can do better than sending all the forbidden sets to a central node, using a spanning tree of the network. For dense instances, it is open as to whether the policies can be compressed to save communication (and hence space in a proof-labeling scheme), by utilising redundancy in the policies. Also, it is an interesting open question as to whether one can do better (in both the sparse and dense cases) by using randomization.

However, for next-hop algebras, constructing a solution (with minimum cost) reduces to constructing a (minimum cost) spanning tree and therefore it is possible to do useful intermediate computation in the network in order to save communication. For two-hop algebras, there is still an $\Omega(n \log n)$ bits communication bound on the associated two-party game, but the stretching trick that we used for the forbidden-set case fails, because it would require policies that could distinguish between paths, based on nodes at a distance $\Omega(n)$ from the source node.

CHAPTER 4

Towards Compact Routing

4.1 Introduction

In the previous chapter, we showed several intractability results for the problem of constructing and verifying stable routing trees for forbidden-set routing. In this chapter, we shall forget about using routing trees and try to construct alternative routing schemes. To do this, we study the model of compact routing, for which good schemes are known for shortest-path routing. We then show how to construct compact routing schemes to solve for the forbidden-set routing problem for various classes of graphs.

Routing tree-based schemes construct a forest of routing trees, one for each destination, and forward packets along the tree for each destination. Therefore, each node stores one port number for each destination, i.e. $O(n \log n)$ bits. A routing tree is *stable* iff no node can switch to a strictly lower-cost path without creating a cycle. The difficulty with routing tree-based schemes is that since nodes are free to choose paths, we have to assume that stable routing trees are the only ones that can exist. Unfortunately, stable trees may not exist and deciding if they do is intractable, both in communication and computation. Our goal is to route on *all* lowest-cost paths while still having low space requirements, preferably sublinear in n (we will show that this is impossible for general graphs, but possibly achievable for some restricted classes of graphs).

4.2 Motivation

Shortest-path routing can be done by storing at each node w a table that lists for every destination v, the next-hop on the shortest path from w to v. It is easy to see that the paths used for routing in this way form a forest of shortest-path trees rooted at each destination. Since every subpath of a shortest path is also a shortest path, a shortest-path routing tree is always stable. The results of Griffin et al. [GSW02] show that for policy-based routing it may be impossible to construct a stable routing tree, so it is not always possible to route on lowest-cost paths using this method. Consider the following simple scheme for policy-based routing. Each node w stores a table where the entry (u, v) specifies the next hop from w on the path from u to v of lowest cost to u. When a node u wants to send a packet to destination v, it writes into the header of the packet the string $\langle u, v\rangle$. Now when some node w receives this packet, it looks up the entry $\langle u, v\rangle$ to find the next link for this packet. This way, each node can route on its lowest-cost path to each destination. However, the downside is that each router now stores $O(n^2)$ entries in its local routing table, which is too demanding in a large network. With a routing tree, all the sources whose paths pass through the same node w to the same destination v must agree to use the same path from w and therefore each node can store at most $O(n)$ entries.

The central question we want to answer is this: *can we reduce the space to below $O(n^2)$ per node, while still being able to route on all lowest-cost paths?* Consider the case of shortest-path routing; it is known [GPPR04] that $\Omega(n)$ bits are required if we wish to route on exactly shortest paths, but this can be reduced if we are willing to accept approximately-shortest paths. We say that a path has stretch k if it has cost within a factor k of the optimal path. Proven cases of a conjecture of Erdos imply that any scheme that routes on paths of stretch three must use space $\Omega(n^{1/2})$ at some node (as remarked in [TZ01b]).

A promising direction is to make use of a compact and localized representation of the graph – each node is assigned a data structure (called its routing table) and a label, which identifies the node to other nodes. Routing is then done as follows: if node u wants to route to v it writes v's label into the packet header. Nodes can then use their routing tables and v's label to decide how to forward the packet through the network. This is known as *compact routing* and was first introduced by Peleg and Upfal [PU89]. More details about localized data structures for routing can be found in the excellent survey paper by Gavoille and Peleg [GP03]. Compact routing has been extremely successful for approximate shortest-path routing: Thorup and Zwick [TZ01b] gave an almost-optimal stretch-3 scheme using routing tables of size $\tilde{O}(n^{1/2})$ and $O(\log n)$-bit

labels[1] where each routing decision takes just constant time.

We argue that for policy-based routing on the Internet, compact routing schemes are better than using routing trees. Since no routing trees are constructed, compact routing schemes can send packets whenever two nodes are reachable. In contrast, packets can be sent only if a stable routing tree exists where the source node is not assigned the empty path (and deciding if such a tree exists is NP-complete even with forbidden-set policies on bounded treewidth graphs). So far nothing is known about the viability of compact routing schemes for policy routing. In particular, it may be that the space requirements are higher than $\Omega(n)$ per node.

The idea of using compact routing on the Internet has been suggested elsewhere, for example [KFY04]. However, the suggestion is to make use of the schemes for approximate shortest-path routing. The freedom offered by policy routing is important and therefore until a scheme exists that can handle policy routing (even for restricted policies such as forbidden-set), there will remain no viable alternative to BGP.

4.3 Preliminaries

We now introduce the model of routing that we will use for this and subsequent chapters. Readers familiar with compact routing may wish to skip this section. A *routing scheme* is a distributed algorithm for delivering packets between processors in a network. Assume that a labeling of the nodes of the network has been given. Each packet has a *header* that contains the label of the destination of the packet and perhaps some additional information that can be used to guide the routing of the packet. Each edge adjacent to a processor is identified by its *port* number. Each processor stores a local data structure called the *routing table*. When a processor receives a packet on an incoming port, it uses its routing table, the incoming port number and packet header to decide whether the packet has reached its destination or, if not, which outgoing port the packet is to be sent on and what the new header should be.

Let G be a graph representing a communication network. We shall assume that G is connected, undirected and unweighted. Each node of G has an identifier $\mathrm{ID}(u) \in \{1, \ldots, n\}$. However, the routing scheme uses a routing label $L(u)$ to identify u. The difference between the identifiers and labels is that the labels may be used to encode additional information about nodes, which may enable more efficient routing strategies to be used. Given a graph with labels

[1] $f(n) = \tilde{O}(g(n))$ if $\exists c \geq 0$ such that $f(n) = O(g(n) \log^c n)$

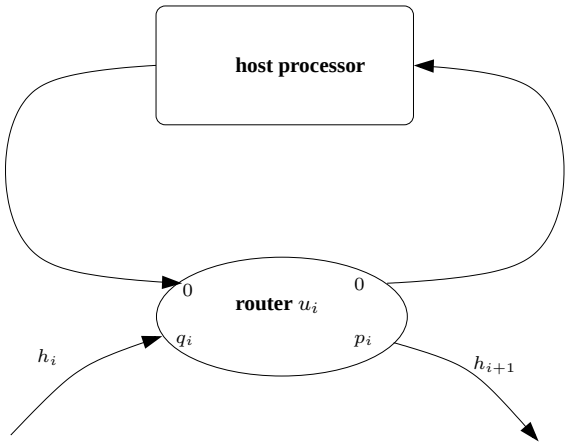


Figure 4.1: A model of a router

$L(\cdot)$, a *routing function* R on G is a distributed algorithm for routing on G. The algorithm builds a path from the source to the destination by selecting, at each intermediate node, the next link on which to forward the packet. More precisely, $R = (P, H)$ where P is the *port function* and H is the *header function*. For any two distinct nodes u, v, R computes a *route* $u = u_0 u_1 \dots u_r = v$, a sequence $h_0 h_1 \dots h_r$ of headers, and a sequence $p_0 p_1 \dots p_r$ of output port numbers. The port numbers identify the links connected to a given node, and may be particular to that node; for example a link connecting x to y may have a different port number in x to its port number in y. The port numbers at a node u are uniquely chosen from $\{1, \dots, \deg(u)\}$. The restriction of R to u is called u's *local routing function*, and this is what we shall refer to as u's local data structure.

Figure 4.1 shows the model of a router that we use. A message arriving at a node u_i through an input port q_i is given a new header $h_{i+1} = H(u_i, q_i, h_i)$ and is forwarded on the output port $p_i = P(u_i, q_i, h_i)$. We require that $q_0 = p_r = 0$, and $h_0 = L(v)$, i.e. the initial header provided by the source is the label for the destination node.

A *routing strategy* is an algorithm that computes, for a graph G, a routing function R on G. Therefore, the strategy consists of a preprocessing stage that assigns labels to nodes and constructs the distributed data structures necessary. A routing scheme can be thought of as an implementation of a function. An *oblivious* routing function is a routing function that only depends on the header, and not the input port. A *direct* routing function is an oblivious routing function that only depends on the destination, and therefore $h_0 = h_1 = \dots = h_r = L(v)$. The

routing functions we shall consider are all oblivious, and some of them are also direct. Direct routing has the advantage that it is usually faster, as routing decisions can be made quickly. A routing scheme is said to be *compact* if the local data structures are sublinear in size, i.e. $o(n)$ bits, and the packet headers are all polylogarithmic in size, i.e. at most $\log^c n$ bits for some constant c.

4.4 Deciding if there exists a zero-cost path

We begin by considering a simpler problem than routing. The *path cost labeling* problem is as follows: given a graph G and costs $\{c_u(v)\}$, assign labels $L(v)$ to nodes so that given only $L(u), L(v)$, we can compute the minimum cost $c_u(P)$ of a path P from u to v. We shall call such a label a *path cost label* for the costs $\{c_u\}$. Throughout the chapter, we shall assume that the costs represent forbidden-set policies, i.e. $c_u(P) = |S(u) \cap P|$ for a path P from u to v, so we will refer to the forbidden sets $S(u)$ instead of the costs. In the *zero-cost* path problem, we are only interested in whether there is a path P of zero cost from u to v. Let $d_G(u, v)$ represent the (unweighted) distance between u, v in the graph G. Then the zero-cost path problem is equivalent to deciding whether $S(u)$ is a separator of u, v in G. This relationship implies that our problem may also be of interest from a graph theory perspective – constructing small zero-cost labels is equivalent to constructing an efficient distributed localized representation of all the separators of a graph.

The motivation for this problem is that any routing scheme that can route on approximately lowest-cost paths must be able to distinguish between the case where there exists a path of zero cost and when there is no zero-cost path. The problem is analogous to deciding reachability in graphs (labels of size $O(\log n)$ bits suffice to decide reachability in undirected graphs – simply label each node with the identity of its connected component). We can now state our first result. Let k be an upper bound on the size of a forbidden set, i.e. $k \geq \max_u |S(u)|$.

Theorem 4.4.1 *Let nodes have forbidden-set policies of maximum size k, with cost as defined above. Then any undirected graph G has zero-cost path labels of size $O(k\Delta(T)\log n)$ bits, where $\Delta(T)$ is the degree of a minimum degree spanning tree of G. Given the labels, we can decide whether there is a zero-cost path in time $O(\log k\Delta(T))$.*

We shall prove Theorem 4.4.1 by first proving a similar result for the case where the sets $S(u)$ are sets of edges rather than nodes. In this case, we are interested in deciding if a set $S(u)$

is a cut between u, v in G. First we shall assume that the sets $S(u)$ are edges instead of nodes, so we are interested in the problem of detecting cuts. Assume also that $|S(u)| \leq k$. An Euler tour of a graph is a cycle that traverses each edge exactly once, although it may visit a node more than once. Euler's theorem says that a graph has an Euler tour iff every node has even degree. We can assume that G is Eulerian by doubling up each edge into its two directed edges, so deleting an edge from G corresponds to deleting two edges from its Euler tour. For each node u, partition the tour into at most $2k - 1$ intervals corresponding to deleting the set $S(u)$. Now build an auxiliary graph $H(u)$ on the intervals where two intervals are adjacent iff they both have an occurrence of the same node. Now we consider reachability in the graph $H(u)$. Let $R(u)$ be the set of nodes of $H(u)$ that can be reached from an interval containing an occurrence of node u. (It is easy to see that all the nodes in $H(u)$ whose intervals contain an occurrence of the same node v, form a clique in $H(u)$. Therefore for a node v in G, we can arbitrarily pick a node of $H(u)$ containing an interval of v in order to determine reachability in $H(u)$.)

The label for node u contains two things:

1. The set of intervals $R(u)$ (using $O(k \log n)$ bits);

2. The position $P(u)$ on the Euler tour of some (arbitrarily-chosen) occurrence of node u (using $O(\log n)$ bits).

Now, given two labels $L(u), L(v)$, we check whether $P(v)$ is contained in an interval of $R(u)$. If not, then $S(u)$ is a uv-cut in G. The intervals $R(u)$ can be stored in an interval search tree (a binary search tree where the key for an interval is its lower limit) so that we can make the decision in time $O(\log k)$ (assuming that $O(\log n)$-bit comparisons take constant time).

Proof of Theorem 4.4.1. If $S(u)$ is a set of nodes, things are harder (it seems). We could delete all adjacent edges to a node, but this would incur a factor of $\Delta(G)$, the maximum degree of a node in G. Alternatively, if G has a Hamiltonian cycle, we can use the Hamiltonian cycle in place of the Euler tour, but cutting nodes instead of edges. For more general graphs, we can do the following. Construct a minimum-degree spanning tree T of G (i.e. a tree whose maximal degree is smallest), and then construct the Euler tour of the tree T (this tour contains at most $2(n-1)$ edges after doubling up the edges of T). Now, we can build the auxiliary graph $H(u)$ where two intervals are adjacent iff at least one of the following holds:

1. both intervals both have an occurrence of the same node;

2. one interval contains an occurrence of u and the other an occurrence of v, and the edge (u, v) is in $E(G) \setminus E(T)$.

The set $R(u)$ and the labels are constructed as for the edge case. Now, each node appears at most $\Delta(T)$ times in the tour, so $R(u)$ contains at most $O(k\Delta(T))$ intervals. Therefore the labels are of size at most $O(k\Delta(T)\log n)$ bits. As for the edge case, we can store the set $R(u)$ in an interval search tree, so the decoder takes time $O(\log k\Delta(T)) = O(\log n)$. □

Note that finding a minimum-degree spanning tree is NP-hard – G has a Hamiltonian cycle iff it has a spanning tree with degree two. So for the ring, we can use the above construction to find labels of size $O(k \log n)$. Graphs of bounded independence number, $1/O(1)$-tough graphs and almost all r-regular graphs (for fixed $r \geq 3$) have spanning trees of bounded degree, so they have labels of $O(k \log n)$ bits. There are many cases where the bound is far from tight; for example, the n-star has $O(1)$-bit labels (we just need to store whether each forbidden set contains the center node) but any spanning tree has degree $\Omega(n)$. We believe that it is possible to improve the space bound (possibly at the expense of a higher running time), but we have been unable to do so.

4.5 A forbidden-set routing scheme

We now consider the forbidden-set routing problem, i.e. routing packets on paths that avoid the source node's forbidden set. Imagine how we might use the cost labels constructed in the previous section to guide the routing of packets through the network. We begin with an example – consider the network of Figure 4.2 where node u wants to send a packet to v on a path that avoids nodes b, e. Should u first send the packet to a or c? Either node is not in the forbidden set of u, so assume that u sends it to c. Now c has to decide where next to send it. We can write the set $\{b, e\}$ into the packet header so that c knows not to forward it to b. Perhaps we can also write into the packet header the route that the packet has taken so far (although this will violate our requirement of small packet headers). Then c knows that the packet just came from u, so the only alternative is to send to d. But then what should it do? Should the packet go to a or to f? The problem is that d does not know that a cannot reach v while avoiding b, e, without returning to d. One possible solution is to construct a *flooding protocol*, where each node sends the packet to all its neighbours except those in the set $S(u)$. Although this would ensure that the packet reaches the destination if there exists a zero-cost path, it is extremely inefficient in terms

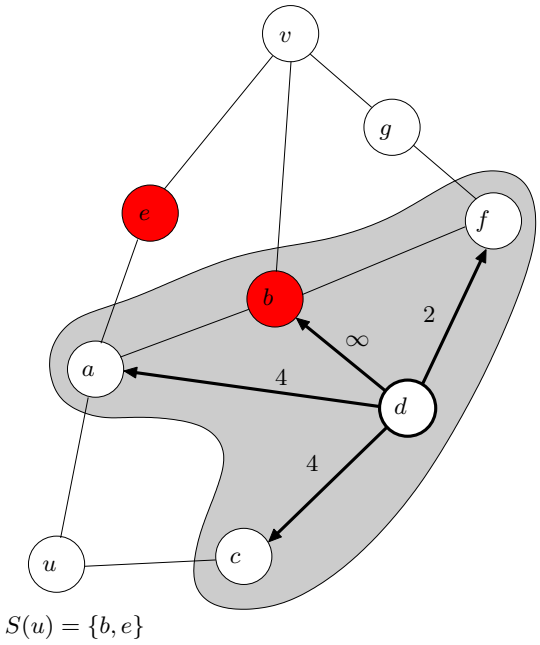


Figure 4.2: Motivating distance separator labels. Consider a packet from u that arrives at d with destination v, and forbidden set $\{b, e\}$. Where should d forward the packet? The distance separator labels allow d to compute the distances to v in $G \setminus \{b, e\}$ from each of its neighbours. It can then forward the packet to the neighbour that minimizes this distance.

of communication complexity and congestion. We want a *forwarding protocol* where packets are only forwarded, not replicated.

4.5.1 Distance separator labels

This motivates the following approach: what if we could construct labels in a different way, so that intermediate nodes w can decide which of their neighbours cannot reach the destination without returning to the current path? We could then guarantee that the packet always makes progress towards the destination, i.e. the distance in $G \setminus S$ to the destination always decreases.

Definition 4.5.1 (Distance separator label) *A* distance separator label *is a label $L(v)$ such that given $L(u), L(v)$ and $L(s_1), \ldots, L(s_k)$, we can compute $d_{G \setminus S}(u, v)$, where $S = \{s_1, \ldots, s_k\}$.*

If S is a separator of u, v in G then by definition, $d_{G \setminus S}(u, v) = \infty$, and we shall define $d_{G \setminus S}(u, s) = \infty$ for all $s \in S$. We shall call such labels *distance separator labels* since they

measure the distance in a graph with a given set of nodes removed. Finally, note that we are interested in measuring the unweighted distances in $G \setminus S$, not the path costs using the $c_u(v)$. The distances in $G \setminus S$ will be used to guide the packet through the network.

Imagine that we have assigned distance separator labels to nodes in an undirected graph G, and each node knows its forbidden set $S(u) \subseteq V(G)$ (the separator labels are constructed without any knowledge of these sets). Also assume that each node u knows the distance separator label for each of its neighbours $u_1 \dots u_d$, with $|L(u_i)| \leq l$ for all u_i. We can then route on a shortest zero-cost path from u to v in G as follows: the source node u writes the labels $L(v)$ and $L(s)$ for each $s \in S(u)$ into the packet header. It then sends the packet to its neighbour u_i that minimizes the distance $d_{G \setminus S(u)}(u_i, v)$. This is done by consulting the distance separator labels (if all the distances are ∞ then u declares that there is no zero-cost path). Each intermediate node w does a similar thing – it examines the incoming packet and forwards it to the neighbour w_i that minimizes the distance $d_{G \setminus S(u)}(w_i, v)$ (without changing the packet header).

If there exists a path of zero-cost from u to v, this scheme always routes packets on the shortest zero-cost path, i.e. a path not containing any element of $S(u)$ and having length equal to $d_{G \setminus S(u)}(u, v)$. This is because a packet is always forwarded to a node that is closer to the destination in $G \setminus S(u)$ than the current node. If u, v are not connected in $G \setminus S(u)$ then u can detect this since $d_{G \setminus S(u)}(u_i, v) = \infty$ for all its neighbours u_i. Since each node u stores the label for itself, the labels for each of its neighbours and each element of $S(u)$ (using $O(lk\Delta(G))$ bits) and the packet headers contain the label of the destination and the labels of $S(u)$ (using $O(lk)$ bits), we have shown the following.

Lemma 4.5.2 *Assume that a family of graphs $\mathcal{G}$ have distance separator labels of size l bits and forbidden sets of size at most k. Then every graph $G \in \mathcal{G}$ has a forbidden-set routing scheme using $O(lk\Delta(G))$-bit routing tables and $O(lk)$-bit packet headers. Packets are sent on shortest paths in $G \setminus S(u)$.*

Remarks. It is important to note that we need *exact* distances; approximate distances will not suffice. If the labels only return c-approximate distances, i.e. a distance $\hat{d}_{G \setminus S}(u, v)$ where

$$(1/c)\hat{d}_{G \setminus S}(u, v) \leq d_{G \setminus S}(u, v) \leq c\hat{d}_{G \setminus S}(u, v),$$

then the scheme may create routing loops (since the packet header does not store the route). Figure 4.3 gives a simple example of this. With exact distances, the distance labels ensure that

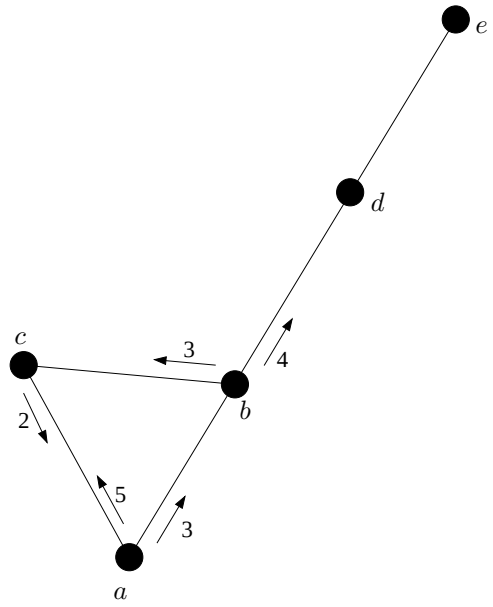


Figure 4.3: How routing loops can occur using approximate instead of exact distances. The distances shown are 2-approximate distances to the destination e, but there is a routing loop $a, b, c, \ldots$

the scheme does not create loops in the routing. It is not clear if we can make use of approximate distance labels for routing, without using large packet headers to store the path taken so far.

Our job is now to find efficient distance separator labels for various graph families, in order to construct forbidden-set routing schemes using the above lemma.

4.5.2 A lower bound

Before continuing, we give a lower bound on the size of labels required by any forbidden-set routing algorithm. The lower bound is for the problem of deciding if there exists a zero-cost path between two nodes, but this is also a lower bound for any forbidden-set routing algorithm that can decide if there exists a zero-cost path before sending the packet. The lower bound is approximately linear in the size of the forbidden sets, and also holds for trees. Therefore, for small k, say $O(\log n)$, it might be possible to construct routing schemes with sublinear space requirements.

Theorem 4.5.3 *Any forbidden-set routing algorithm on n-vertex trees with forbidden sets of*

size at most k must assign labels of size at least $\Omega(k \log n/k)$ bits in the worst case.

Proof. Consider a tree with root u and n children $u_1 \dots u_n$. Each u_i can independently select as its forbidden set an arbitrary subset of size k of $\{u_1, \dots, u_n\}$. For every distinct collection of forbidden sets $\{S(u_1), \dots, S(u_n)\}$, it can be seen that there exists a distinct pair of nodes u_i, u_j whose reachability has changed. Each of these requires a different set of labels to be assigned to the graph, so at least $\Omega\left(\left(\sum_{i=0}^{k} \binom{n}{i}\right)^n\right) = \Omega((n/k)^{nk})$ distinct labelings of the graph are required. Taking logs and dividing by n, it follows that at least one of these nodes must receive a label of size $\Omega(k \log n/k)$ bits. We can also prove a bound for the case where $S(\cdot)$ is a set of edges – let each $S(u_i)$ independently choose a subset of the n edges and then the same argument also applies. □

4.6 Distance separator labels

In the previous section we motivated the construction of efficient distance separator labels, by showing how they can be used to construct a reasonably efficient distributed forbidden-set routing scheme. In this section we try to construct efficient (by which we mean of size polylogarithmic in the number of nodes in the graph) distance separator labels.

4.6.1 Trees

We shall show how to exploit the very simple structure of trees, namely that there is a unique path between each pair of nodes, to construct efficient separator labels for them. Assume that we have a rooted tree (the root can be chosen arbitrarily, since it is not important for deciding if a node is a separator of two nodes – the unique path is the same regardless of the root). It is well-known that we can assign labels of size $2\lceil \log n \rceil$ bits to nodes, so that given the labels $L(x), L(y)$ we can decide in constant time if x is an ancestor of y [KNR92]. This is done as follows. Do a depth-first traversal from the root, labeling each node v with its identifier $ID(v)$ in the depth-first traversal. For each node w, let f_w be the descendant of w with the largest identifier. The *ancestor label* for v is defined as $L(v) = (ID(v), f_v)$. A node v is a descendant of w iff $ID(v) \in [ID(w), f_w]$.

We also need the related concept of the least common ancestor. A node w is the *least common ancestor* (LCA) of two nodes u, v iff w is the unique node that is furthest from the root and on both the paths from u and v to the root. Peleg [Pel00] showed that we can assign

$O(\log^2 n)$-bit labels such that given $L(u), L(v)$ we can deduce the LCA of u, v. We can also deduce its ancestor label by storing a mapping from the identifiers of ancestors stored in a label to their ancestor labels with a constant increase in label size. More recently, Alstrup et al. [AGKR02] showed the following theorem.

Theorem 4.6.1 ([AGKR02]) *There is a linear-time algorithm that labels the nodes of a rooted tree T with distinct labels of length $O(\log n)$ bits such that from the labels of $u, v \in T$ we can compute in constant time the label of* $\mathrm{LCA}(u, v)$.

Remarks. It should be noted that the difference between the schemes of Alstrup and Peleg is that Alstrup's scheme only computes the *LCA label* for the LCA, whereas Peleg's scheme computes the *identifier* of the LCA. Peleg has also shown a lower bound of $\Omega(\log^2 n)$ for any labeling scheme that computes the identifier of the LCA node. We shall be able to make use of Alstrup's scheme, and therefore use only $O(\log n)$-bit labels.

Our scheme relies on the following observation: a node w is a uv-separator in T iff $\mathrm{LCA}(u, v)$ is an ancestor of w and w is an ancestor of either u or v. A set $S = s_1, \ldots, s_k$ is a uv-separator in T iff at least one member of S is a uv-separator in T, so to see if S is a separator we can test each member of S independently. We can solve this using Alstrup's LCA labeling scheme as follows. Let $\mathcal{J}$ be the LCA label for node u. Then we compute the LCA label for the node $\mathrm{LCA}(u, w)$ and check if it equals $\mathcal{J}$. If so, u is an ancestor of w. Therefore Alstrup's LCA labels suffice to decide the ancestor relation.

The separator label for u is simply Alstrup's LCA label for u in T. Given $L(u), L(v)$ and $L(s_1), \ldots, L(s_k)$, we compute the LCA label for $\mathrm{LCA}(u, v)$ using the scheme of Alstrup et al. [AGKR02] and then test if each s_i is a uv-separator of T using the observation described above. Therefore the label has $O(\log n)$ bits. To construct distance separator labels for T, we can combine any distance labeling scheme [GPPR04] for trees with the separator labels. To compute the distance $d_{T \setminus S}(u, v)$, use the separator labels to decide whether $d_{T \setminus S}(u, v) = 0$; if not, then $d_{T \setminus S}(u, v) = d_T(u, v)$, so we can use the distance labeling scheme. Peleg et al. showed that unweighted trees have distance labeling schemes using $O(\log^2 n)$ bits per label (and this is known to be tight). By combining these two scheme, we get distance separator labels using $O(\log^2 n)$ bits. This is asymptotically optimal, since there is a lower bound of $\Omega(\log^2 n)$ bits for distance labeling in trees [GPPR04].

Interestingly, the size of the distance separator label is dominated by the size of the distance label, not the separator label. It is an interesting question whether we can do better if we accept

approximate distances in distance separator labels. In general, there are an exponential number of possible separators, so at this stage it is not obvious whether the scheme for trees can offer much insight into how to deal with more general graphs.

4.6.2 Bounded cliquewidth graphs

We now consider the class of bounded cliquewidth graphs. The *cliquewidth* of a graph is a measure of its complexity, closely related to treewidth but more powerful since every graph having bounded treewidth has bounded cliquewidth but the converse is not true (cliques have cliquewidth two but unbounded treewidth). For a given graph its cliquewidth is defined as the minimum number of distinct labels required to construct the graph by only using the following operations:

- create a node with a given label;
- $p \rightarrow q$: relabel all the nodes having some label p to another label q;
- $p \times q$: connect every node having label p to all the nodes having label q.

We can therefore represent a graph by its algebraic expression, or term tree where the leaves are labelled nodes of the graph and the interior nodes of the tree represent either a relabelling operation $p \rightarrow q$ or a join operation $p \times q$.

Many graph problems can be formulated in monadic second-order logic (MS), by using logical operations $(\wedge, \vee, \neg)$, quantification $(\forall, \exists)$, membership tests $(\in, \subseteq)$ and adjacency tests $(\{u, v\} \in E)$ over subsets $(X_1, X_2, \ldots)$ of nodes of a graph. As an example, consider the following graph property.

"is the subgraph of G induced by Z connected?"

$$\begin{aligned}
&\mathsf{Partition}(U, V, Z) && \equiv (Z = U \cup V) \wedge (U \cap V = \emptyset) \wedge (U \neq Z) \wedge (V \neq Z) \\
&\mathsf{Adjacent}(U, V) && \equiv \exists u, v (u \in U \wedge v \in V \wedge (\{u, v\} \in E(G))) \\
&\mathsf{Connected}(Z) && \equiv \forall U, V (\mathsf{Partition}(U, V, Z) \implies \mathsf{Adjacent}(U, V))
\end{aligned}$$

A property P is said to be *MS-definable* if it can be expressed in MS logic. A *graph property* is a property where the variables denote the nodes of the graph under consideration. Courcelle and Vanicat showed that for any MS-definable graph property, we can construct small labels that can be used to efficiently decide the property:

Lemma 4.6.2 ([CV03]) *Let q be an integer and $P(x_1, \ldots, x_n)$ an MS-definable graph property. Let G a graph with n nodes cliquewidth at most q. Then we can assign to nodes of G labels of size $O(\log n)$ bits so that given only $L(x_1), \ldots, L(x_k)$, we can decide $P(x_1, \ldots, x_k)$ in worst-case time $O(k \log n)$. The constants in the big-oh notation depend only on q and P.*

We now show that the property required by distance separator labels is MS-definable, which will enable us to appeal to the above result.

Lemma 4.6.3 *Bounded cliquewidth graphs have separator labels of size $O(\log n)$ bits, and a decoder with worst-case time complexity $O(k \log n)$.*

Proof. We first show that the graph property "a set of nodes is a uv-separator" is MS-definable. We can use the property Connected to construct the following property.

"is the subgraph induced by Z connected and $x, y \in Z$?"
$\mathsf{Connected}(x, y, Z) \quad \equiv (x \in Z \wedge y \in Z \wedge \mathsf{Connected}(Z))$

Now we can express our desired property of deciding if there is a zero-cost path from x to y. We can do this by testing if there exists a set of nodes S such that S does not contain a forbidden set and there exists a path through only the nodes of S from x to y.

"is there a path x to y that avoids nodes in Z?"
$\mathsf{Path}(x, y, Z) \quad \equiv \exists S((Z \cap S = \emptyset) \wedge \mathsf{Connected}(x, y, S))$

It is clear that the property $\mathsf{Path}(x, y, Z)$ is MS-definable, and it is not difficult to check that $\mathsf{Path}(x, y, Z)$ holds iff Z is not a uv-separator in G. □

In the same paper, Courcelle and Vanicat defined an optimization version $\min(\varphi)$ of an MS-definable property φ. Here, there is a free variable that denotes a set of nodes and the cardinality of the set denoted by the free variable is minimized. Using this, we can compute distance separator labels by appealing to the following theorem.

Lemma 4.6.4 ([CV03]) *Let q be an integer and $P(x_1, \ldots, x_n)$ an MS-definable graph property. Let G a graph on n nodes with cliquewidth at most q. Then we can assign to nodes of G labels of size $O(\log^2 n)$ bits so that given only $L(x_1), \ldots, L(x_k)$, we can compute the value $\min P(x_1, \ldots, x_k)$ in worst-case time $O(k \log^2 n)$. The constants in the big-oh notation depend only on q and P.*

Combining this with the routing scheme in Lemma 4.5.2 gives the following result.

Theorem 4.6.5 *Bounded cliquewidth graphs have a forbidden-set routing scheme, with routing tables of size $O(\Delta(G)k\log^2 n)$ bits and packet headers of size $O(k\log^2 n)$ bits.*

Proof. We just need to show how to define the property of distance separator labels as a monadic second-order logic optimization property over the graph nodes. This is easily done using the following:

"what is $d_G(u,v)$?"
$d_G(u,v)+1 = \min(\varphi)$ where $\varphi(u,v,Z) \equiv \mathsf{Connected}(Z) \wedge (u \in Z) \wedge (v \in Z)$

"what is $d_{G\setminus S}(u,v)$?"
$d_{G\setminus S}(u,v)+1 = \min(\varphi)$ where
$\varphi(u,v,S,Z) \equiv \mathsf{Connected}(Z) \wedge (u \in Z) \wedge (v \in Z) \wedge (S \cap Z = \emptyset)$

In both cases, the cardinality of the set Z is one greater than the length of the path. Combining the distance separator labels with the routing scheme in Lemma 4.5.2 gives the stated result. □

Remarks

The bounded-clique width scheme suffers from two major problems:

1. A clique decomposition of the graph needs to be given to the algorithm. Unfortunately, it is known that given a graph G and a positive integer q, the problem of deciding if G has clique width at most q is NP-complete [FRRS06], for arbitrary values of q. However, a result of Oum [iO05] gives, for fixed q, a cubic algorithm that computes a clique decomposition of width $2^{O(q)}$, which is enough if we are only interested in graphs having clique width bounded by some fixed integer.

2. The hidden constant in the label size is huge – if h is the number of quantifier alternations in the formula for P, then the constant is a tower of exponentials in the clique width, having height $O(h)$:

$$2^{2^{2\cdots 2^q}}$$

The second problem is due to the tree automaton approach used by Courcelle and Vanicat to construct the labels. Since their result is for general MS-definable properties, a result of Grohe and Frick [FG04] implies that (unless P=NP), this tower of exponentials is unavoidable. Therefore, although the general scheme is somewhat impractical for our specific problem of distance separator labels, it is important since it shows that labels whose size is only polylogarithmically-dependent on n are possible. Our aim will now be to try to reduce the dependence on the cliquewidth (or other graph parameter) to polynomial or even linear for various, often more specific, families of graphs. We do this by exploiting the simple structure offered by separators in graphs.

4.6.3 Cographs

Before tackling graphs with small tree width, we warm up by considering *cographs*, which are graphs having clique width at most two. The family of cographs can be defined using two operations:

- *disjoint union*: for graphs G, H on disjoint vertices, $G + H$ is the graph formed by the union of the edges and vertices of G and H.
- *complete product*: $G \times H$ is the graph formed by taking the union of G and H and adding edges $\{u, v\}$ for all $u \in G, v \in H$.

We can write the algebraic expression for a cograph as a term tree as in Figure 4.4. Each node is labeled by its *access path* from the root, which describes how to reach the node in the term tree. In the figure, we get $L(u) = +2 \times 2 + 2 \times 1 + 1$ where the numbers 1,2 indicate which child to take at each level in the path. These labels are of size $O(h)$ bits where h is the height of the term tree.

Let T be a term tree defining a cograph G. Given the labels for two nodes u, v and k nodes $S = \{s_1, ..., s_k\}$, the set S is a uv-separator iff

- The least common ancestor $w = \mathrm{LCA}(u, v)$ is labeled with '$+$', and;
- For every ancestor z of w labeled '$\times$', with z_i the child of z whose subtree contains w, all the descendants of z_{3-i} must be in S.

From $L(u), L(v)$ we can find w, z and hence check the first property. We can then examine the access paths for $s_1, \ldots, s_k$ and check the second property by seeing if these paths contain the

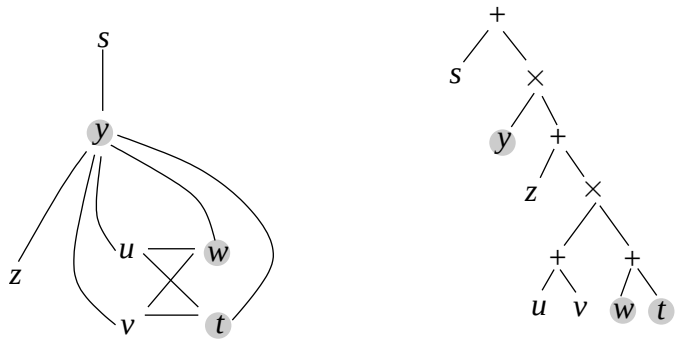


Figure 4.4: The cograph generated by the term $+(s, \times(y, +(z, \times(+(u, v), +(w, t)))))$ and its term tree. The set $\{y, w, t\}$ is a uv-separator.

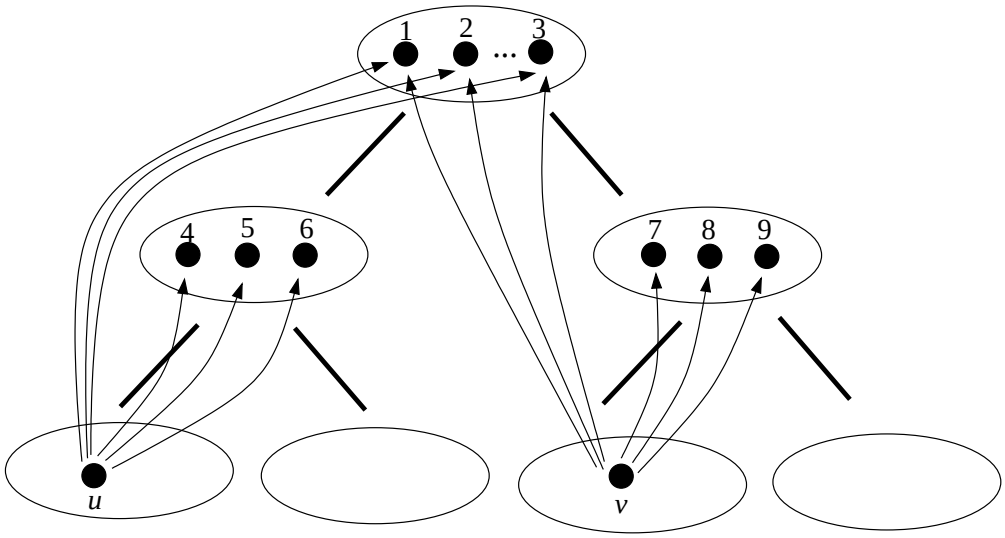
entire subtree rooted at z_i (since we know that the term tree is a binary tree). As an example, consider Figure 4.4 and deciding whether $S = \{y, w, t\}$ is a separator of u, v. The LCA of u, v is labeled with '$+$', and for the highest node labeled '$\times$', we have $y \in S$. For the other ancestor labeled '$\times$', we also have that $w, t \in S$. Therefore S is a uv-separator.

Remarks

The diameter of a cograph is at most two, so we only considered the problem of deciding whether a given set is a separator (rather than computing the distance around this set). This will be enough to introduce the more involved scheme for treewidth k graphs in the next section. We may hope that we could arrange for the term tree to be of height $O(\log n)$; however this is not possible using the algebra here – for a counterexample, take the cograph with nodes $1, \ldots, n$ where node i links to every node $j < i$ if i is even, and is not linked to any node $j < i$ if i is odd. The cograph expression is unique and of height n, hence we would assign some nodes labels of size $\Omega(n)$ bits.

4.6.4 Treewidth k graphs

A graph having treewidth k can be expressed as the nondisjoint union of graphs of size $k + 1$, arranged as nodes in a tree such that the set of tree nodes containing some graph node forms a connected subtree of the tree. Small treewidth graphs are an interesting class of graphs to study for several reasons: firstly, they capture a common class of networks (those having a tree-like structure, for example the Internet backbone) and secondly, the concept of treewidth is weaker



$$L(u) = \langle (4, d(u,4)), (5, d(u,5)), (6, d(u,6)), (1, (d(u,1)), (2, d(u,2)), (3, d(u,3)) \rangle$$

Figure 4.5: A decomposition tree for a graph with small recursive separators. Each node of the tree is a separator in the graph, and the distance label for a node u stores its distance to each of the nodes in its ancestor separator nodes. To find the distance from u to v, the decoder returns the minimum value of $d(u,w) + d(w,v)$ over all nodes w in the least common ancestor node of u, v.

than cliquewidth – any graph with treewidth k has cliquewidth at most $2.2^k - 1$ [CR05]. For this reason, we can use the result for bounded cliquewidth graphs to handle bounded treewidth graphs, but if k is nonconstant then we immediately get a huge blowup in label size. Additionally, if we can show how to construct distance separator labels for treewidth k graphs then it might give us insight into how to handle cliquewidth k graphs.

We will show, as our main technical result, how to construct distance separator labels for treewidth k graphs using $O(k^2 \log^2 n)$ bits (Theorem 4.6.10). For comparison, the best known distance labeling scheme for these graphs uses labels of size $O(k \log^2 n)$ bits, so we will have paid an additional factor k to encode distances under node deletions. For graphs of small treewidth, eg $k = O(\log n)$, this is a small penalty, which means that compact (i.e. $O(n)$-bit labels) forbidden-set routing may be possible.

Background: distance labeling for small treewidth graphs

Before tackling distance separator labels, we shall review some distance labeling schemes for treewidth k graphs. We will make use of the following definition. A graph G has a 1/3-balanced separator of size $r(n)$ if there is a set of $r(n)$ vertices whose removal breaks the graph into two connected components of size at least $n/3$. The graph G has a *recursive* 1/3-balanced separator of size $r(n)$ if it has a 1/3-balanced separator of size $r(n)$, and both the components obtained by removing the separator also have recursive 1/3-balanced separators. Therefore, the graph can be recursively decomposed until we reach singletons, giving a binary decomposition tree of height $O(\log n)$. It is known that treewidth k graphs have recursive 1/3-balanced (or simply balanced) separators of size k.

Peleg et al. [GPPR04] showed how to easily extend a distance labeling scheme for trees to one for graphs with small recursive separators. In a tree, the label for u stores the distance $d(u, w)$ to every ancestor w of u in the decomposition tree. One can extend this by storing the distance $d(u, w)$, for every node w that is in an ancestor node in the decomposition tree. Then to compute the distance between u and v, it suffices to compute their least common ancestor S in the decomposition tree and then to compute $d(u, v) = \min_{w \in S}(d(u, w) + d(w, v))$. This works because every path between u, v must go through some node $w \in S$. Since each separator is of size at most c and the tree is of height $O(\log n)$, each label stores at most $O(c \log n)$ distances using $O(c \log^2 n)$ bits. Figure 4.5 illustrates this technique for distance labeling. More precisely, they showed the following result.

Theorem 4.6.6 ([GPPR04]) *Let $R(n) = \sum_{i \leq \log_{3/2} n} r\left(\left(\frac{2}{3}\right)^i n\right) \leq r(n) \log n$. For a family of graphs $\mathcal{G}$ having recursive balanced separators of size $r(n)$, every graph in $\mathcal{G}$ has distance labels of size at most $O(R(n) \log n + \log^2 n)$ bits. Moreover, the distance can be computed in time $O(\log n)$ given two labels.*

The above result immediately implies that treewidth k graphs have distance labels of size $O(k \log^2 n)$ bits. Peleg [Pel99] describes an alternative method for constructing approximate distance labels, based on a hierarchy of tree covers. A *tree cover* of a graph G is a family $\mathcal{F} = \{T_1, \ldots, T_k\}$ of trees with the following two properties:

1. Each tree dilates distances, i.e. $d_{T_j}(u, v) \geq d_G(u, v)$ for all u, v.

2. For any pair of nodes u, v, there exists a tree T_i such that $d_{T_i}(u, v) = d_G(u, v)$.

If a graph has a tree cover of size k then it has distance labels of size $O(k \log n)$ bits. This follows since we can compute distance labels of size $O(\log n)$ bits for each tree, and then simply pick the tree that minimizes the distance. In fact, we can use the tree cover for routing; we use an extra $\log k$ bits in the packet header to specify the tree to route on, and each node maintains information for routing on k trees. This can be seen as a natural extension of using a single routing tree, which was the model we used to prove our negative results in Chapter 3.

It is easy to observe that all graphs have linear-sized tree covers, consisting of the n shortest path trees ending at each node. In the worst-case, this is tight since the complete graph K_n does not have a tree cover of size $n/2-1$: the union of the trees must cover all the edges of the graph, otherwise the scheme would not be able to report that the endpoints of some edge are adjacent (and at distance one). Since each tree has at most $n-1$ edges and the undirected clique has $\binom{n}{2}$ edges, any tree cover of it must use at least $\binom{n}{2}/(n-1) > n/2-1$ trees.

The scheme of Peleg uses a tree cover construction that exhibits a tradeoff between the number of trees each node appears in (the overlap) and the depth of each tree. By constructing trees with depth 2^i for $i = 1 \ldots \log n$, each node appears in a small number of trees and for every pair of nodes, there exists a tree that contains them both. Using $o(\log^3 n)$ bit labels, the scheme can provide distance estimates accurate up to a factor of $\sqrt{2 \log n}$, for arbitrary undirected unweighted graphs.

Now we consider graphs of treewidth at most k. As described above, the schemes described by Peleg et al. [GPPR04] construct distance labels for these graphs by building a decomposition tree using the property of small recursive separators. It is not clear if it is possible to use decomposition trees to construct separator labels (since we need to encode *all* separators), nor is it clear that the tree decomposition associated with treewidth graphs can also be used efficiently. Therefore, we shall take a different approach and make use of an alternative representation of small treewidth graphs, based on algebraic expressions.

Algebraic expressions for treewidth k graphs

Every graph of treewidth k can be represented by an algebraic expression (or term) over some domain of sources $\{1, \ldots, k+1\}$. A j-source graph is a graph with at most j distinguished nodes called *sources*, each tagged with one of j distinct labels. Courcelle [Cou07, ACPS93] shows that a graph has treewidth k if and only if it is the value of some term tree whose leaves are $(k+1)$-source graphs and where every non-leaf node is labeled with one of the following

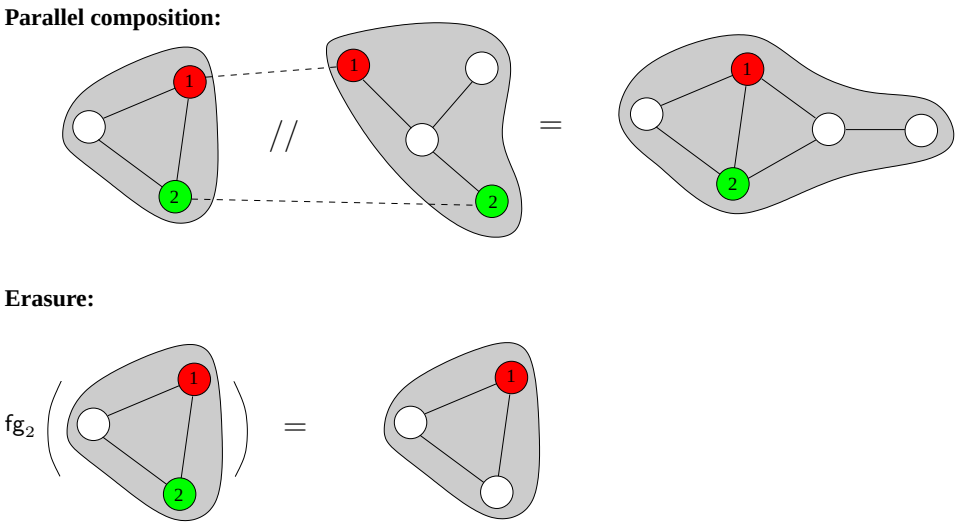


Figure 4.6: The parallel composition and erasure operations for constructing graphs of treewidth k

operations, illustrated in Figure 4.6.

- *Parallel composition*: The graph $G \,//\, H$ is obtained from the disjoint union of graphs G and H where sources having the same label are fused together into a single node.
- *Erasure*: Let a be a label. Then the unary operation $\text{fg}_a(G)$ erases the label a and the corresponding source in G is no longer a source in G.

The term tree can be constructed given a tree decomposition of the graph – Corollary 2.1.1 of [Cou07] shows that given a tree decomposition of width k of a graph, it is possible to construct in linear time a term tree using at most $k + 1$ source labels. The nodes of the term tree are the bags of the tree decomposition; hence the height and degree are unchanged. We now give a brief sketch of how to construct the term tree. Let us assume that we have a tree decomposition (T, X) of width k of a graph G, i.e. every bag contains at most $k + 1$ nodes. Then it is possible to colour the nodes of G using at most $k + 1$ colours so that no two nodes in the same bag of the tree decomposition have the same colour. We can construct a term tree recursively: if T is a single node, then the term tree is the graph with sources being the nodes in T. Otherwise, let r be the root of T and let $T_1, \ldots, T_p$ be its subtrees. For each subtree T_i, associate its source graph G_i on at most k sources, where node x is the jth source iff x is assigned the colour j in the colouring of G. Recursively compute the terms $t_1, \ldots, t_p$. For every i, let A_i be the set of

sources that are in G_i but not the root of T. Then we can represent G by the term

$$\mathsf{fg}_{A_1}(t_1) \mathbin{//} \ldots \mathbin{//} \mathsf{fg}_{A_p}(t_p) \mathbin{//} X(r)$$

where $\mathsf{fg}_{A_i}(t_i)$ is the graph obtained from t_i by forgetting the sources A_i and $X(r)$ is the set of edges between nodes of G in the bag r of T. The details of this construction are not important for the presentation of our algorithm, since we shall simply assume that we are given some term tree that evaluates to the graph G.

We shall assume that the term tree has height $O(\log n)$, since the following result of Bodlaender shows that we can always convert a tree decomposition of some graph into a binary one of logarithmic height with only a constant increase in the width.

Lemma 4.6.7 ([Bod89]) *Given a tree decomposition of width k and a graph G with n nodes, one can compute a binary tree decomposition of G of depth at most $2\log_{5/4}(2n)$ and width at most $3k+2$ in time $O(n)$.*

Therefore, we can always assume that if we are given a tree decomposition of width k of an n-node graph then we can construct a balanced term tree for the graph on $O(k)$ labels and having height $O(\log n)$. From now on, we shall use $k' = 3k+3 = O(k)$ to denote the number of distinct source labels in the (balanced) term tree.

We assume that there are no sources remaining after evaluating the term tree, i.e. all sources have been erased below the root. Therefore the nodes of the graph can be put in bijection with erasure operations; we shall use v to refer to both the node in G and its unique corresponding erasure operation in the term tree. For a node u, we shall use $G(u)$ to denote the graph that results from evaluating the subtree of the term tree rooted at u.

Each node u shall have a *state* $q(u)$ assigned to it. This is a $k' \times k'$ matrix describing the reachability of sources in $G(u)$ – in this matrix, the entry (p,q) is 1 iff the source labeled p can reach the source labeled q in the graph $G(u)$, and 0 otherwise. For convenience, we shall use the equivalence relation $p \sim q$ to denote reachability of source p, q in some source graph. By definition of parallel composition, if $p \sim q$ in $G(u)$ and $q \sim r$ in $G(v)$, then $p \sim r$ in $G(u) \mathbin{//} G(v)$.

It can sometimes be confusing when there are several sources with the same label on an access path (due to erasing then introducing a new source with the same label using $//$). To make things simpler, we shall add subscripts to the source labels to uniquely identify them; for

example, instead of p appearing several times on a path, we may have $p_1, p_2, \ldots$. This does not affect the correctness of the term tree; since we never have two sources with the same label in the same graph $G(u)$, we shall never have two sources p_i, p_j appearing at the same time. From now on, we assume that our source labels are subscripted in this way. This assumption will be helpful when we try to construct separator labels.

Finally, it will be easier to deal with binary term trees than ones having a mixture of unary and binary operations, so we compress a parallel composition operation followed by a sequence of erase operations into a single binary operation as in [CV03]; the sequence $\mathsf{fg}_a(\mathsf{fg}_b(\cdots(G \mathbin{//} H)))$ becomes $G \mathbin{//}_{\mathsf{fg}_a,\mathsf{fg}_b\ldots} H$. All nodes of the graph associated with an erase node u are now associated with the compressed operation containing the erasure operation u. In particular, we shall associate u with the graph $G(u)$ obtained by applying the parallel composition operation but *not* the sequence of erasure operations. Therefore the source associated with u still exists in $G(u)$. We write 'the access path for u' to denote the unique path from the root to u in the term tree, *excluding* the node u, as this simplifies the exposition. As a result, the set of nodes adjacent to the access path for u always contains u.

A connectivity labeling scheme

We begin by constructing a labeling scheme that allows us to determine if two nodes are connected in G, and then extend it to compute connectivity when nodes are removed. As in [CV03], we shall store in the label $L(u)$ a string describing the access path for u and the state for every node adjacent to the access path. In addition, the label contains the source label of the node u in $G(u)$ (recall that u is always a source node in $G(u)$). If u has the source label s_u then the string is of the form

$$Q(u) = (f_1, i_1)q(s_{3-i_1}(u_1))\ (f_2, i_2)q(s_{3-i_2}(u_2))\ \ldots\ (f_h, i_h)q(s_{3-i_h}(u_h))\ s_u$$

where h is the height of the term tree, $f_1 \ldots f_h$ are the operations on the path (e.g. $\mathbin{//}_{\mathsf{fg}_a\ldots}$), $i_1 \ldots i_h \in \{1, 2\}$ indicate which child to take and $s_1(u)$ (respectively $s_2(u)$) denote the left (respectively right) child of u. The states $q(s_{3-i_1}(u_1))q(s_{3-i_1}(u_2)) \ldots q(s_{3-i_1}(u_h))$ are the states of nodes adjacent to the access path for u. Since each set of at most k' erasure operations can be identified with k' bits and the term tree has height $O(\log n)$, the access path can be described using $O(k' \log n) = O(k \log n)$ bits. The reachability matrices adjacent to the path are stored using $(2 \log n)k'^2 = O(k^2 \log n)$ bits, so the labels have size $O(k^2 \log n)$ bits. Figure 4.8 gives

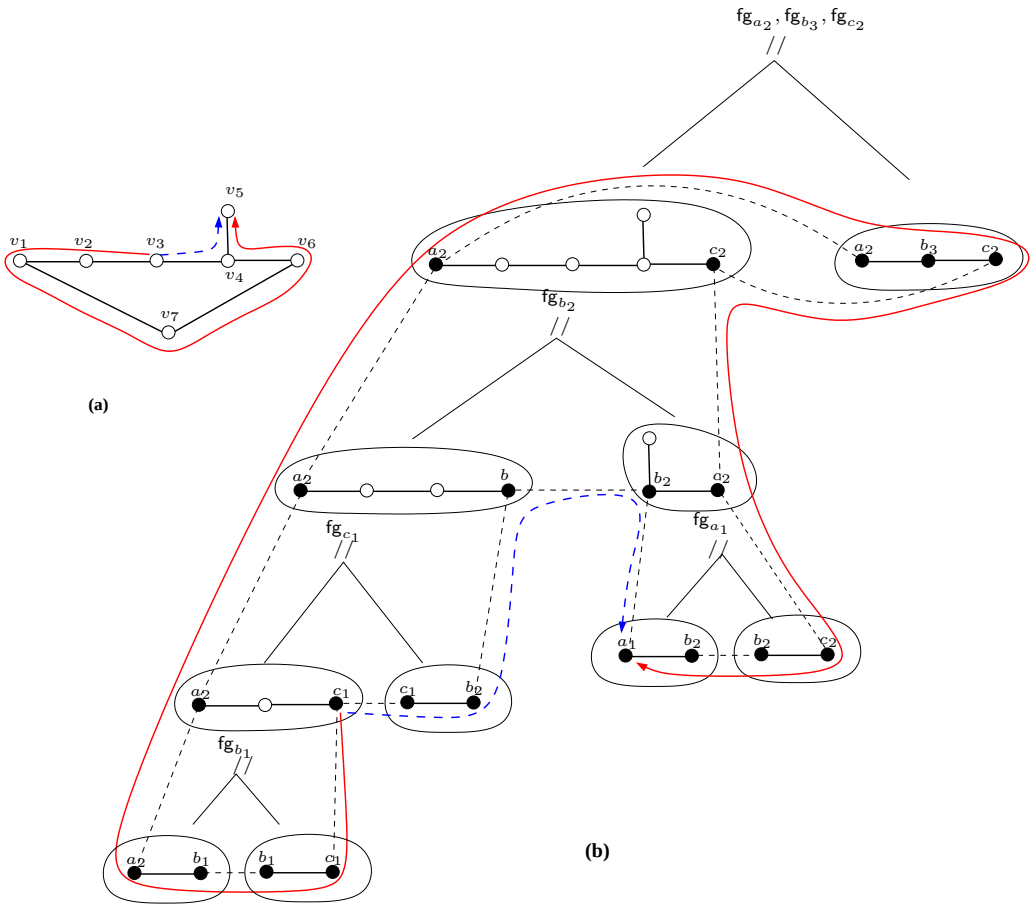


Figure 4.7: (a) A graph and (b) a term tree that evaluates to the graph. The two paths from v_3 to v_5 are drawn in red and blue (dashed). Deleting v_1 removes the source node a_2, so would remove the red path.

CONSTRUCT-LABELS(G)
1 $(X, I) \leftarrow$ a tree decomposition of G of width k
2 Convert (X, I) into a binary tree decomposition of width $3k + 2$ and height $O(\log n)$
3 Compute a binary balanced term tree T from (X, I) with $k' = 3k + 3$ labels [Cou07]
4 **for** each node $u \in T$
5 **do** $s_u \leftarrow$ the source label of u in $G(u)$
6 $Q(u) \leftarrow$ the access path for u and the states of nodes adjacent to the access path
7 $L(u) \leftarrow (s_u, Q(u))$

Figure 4.8: An algorithm to construct connectivity labels $L(u)$ for treewidth k graphs

an algorithm to construct these labels.

The decoder algorithm works as follows. Let the root of the tree be r, and let nodes u, v have source labels s_u and s_v in the graphs $G(u), G(v)$. The following lemma shows how to use the reachability matrices for nodes adjacent to the access paths for u, v to construct a path from u, v in G, if one exists. The idea is to construct a set of paths involving only the source nodes of the graphs $G(w)$ for nodes w adjacent to the access paths for u, v. Sequences of non-source nodes on each path (i.e. source nodes that have been erased below either u, v in the tree) are contracted into a single edge in these paths. Joining these paths together will give a path in G.

Lemma 4.6.8 *Let s_u, s_v be the source labels of u, v in $G(u), G(v)$. Then u, v are connected in G iff we can find a source $p \in G(x)$ for some ancestor x of LCA(u, v), and sequences of parallel compositions establishing the following:*

1. *$(s_u \sim p)$ using the states of nodes adjacent to the access path for u;*

2. *$(s_v \sim p)$ using the states of nodes adjacent to the access path for v;*

Proof. First consider the '$\rightarrow$' direction. At each parallel composition $G(u) \,//\, G(v)$, the connectivity of the sources in the resulting graph is completely determined by the connectivity of the sources in $G(u)$ and the connectivity of the sources in $G(v)$. In particular, $p \sim r$ in $G(u) \,//\, G(v)$ iff $p \sim q$ in $G(u)$ and $r \sim q$ in $G(v)$, for some q in both $G(u), G(v)$. Therefore, any sequence of parallel composition operations as in the claim corresponds to a path connecting u, v in $G(\text{root}) = G$.

Now we consider the other direction. Recall our assumption that all sources are eventually erased. If we cannot find sequences of parallel compositions as in the statement of the lemma, then this implies that there is a node x in the term tree where x is an ancestor of LCA(u, v), the graph $G(x)$ has no sources, and $s_u \not\sim s_v$ in $G(x)$. Since at each parallel composition, the connectivity of the resulting graph is completely determined by the connectivity of the sources of the child graphs, it is impossible to find a sequence that establishes $s_u \sim s_v$ by any sequence of compositions involving $G(x)$. Therefore u, v remain unconnected in the graph corresponding to any ancestor of $x \in T$; in particular this holds for $G(\text{root}) = G$. □

The proof of the above lemma implies that if u, v are connected in G, we can find a path by examining only the connectivity of sources in the graphs $G(u), G(v)$ and the graphs associated with nodes that are adjacent to the access paths of u, v from the root. Replacing the entry i, j in the reachability matrix for $G(u)$ by the distance from s_i to s_j in $G(u)$ allows us to find the length of every path from u to v. Just as the sources are the only nodes that determine connectivity under parallel composition, they also completely determine the distance, i.e.

$$d_{G(u) \;//\; G(v)}(p, r) = \min_q d_{G(u)}(p, q) + d_{G(v)}(q, r).$$

This modification gives distance labels of size $O(k^2 \log^2 n)$ bits for treewidth k graphs. However, it is already known that treewidth k graphs have $O(k \log^2 n)$-bit distance labels [GPPR04], so this bound is larger by a factor of k. We now show that these larger labels capture more structure of the graph, in particular they capture the structure of separators that will allow us to construct distance separator labels using the same label size.

As an example, Figure 4.7 shows a term tree and the graph that it evaluates to. In the figure, two paths between nodes v_3 and v_5 are drawn, and sources that fuse together are joined with dashed lines (in the final graph, these are a single node). We can apply Lemma 4.6.8 by tracing subpaths in the leaf graphs and joining them together using the parallel composition operations. For example, we can take p to be the source labeled c_2, then $c_1 \sim c_2$ using the label $L(v_3)$ and $a_1 \sim c_2$ using the label $L(v_5)$.

Constructing separator labels

We now show how to decide if two nodes are connected in $G \setminus S$ for some set S of nodes. We first give a somewhat inefficient scheme, then we show how to reduce the space requirement later. For each node u and each subset S of source labels (where $S \subseteq \{1, \ldots, k'\}$), we store a

CONNECTED$(L(u), L(v), L(s_1), \ldots, L(s_k))$
▷ Let $S = \{s_1, \ldots, s_k\}$
▷ Returns TRUE iff u, v are connected in $G \setminus S$
1 **for** each $s_i \in S$
2 **do** Recompute the state $q(x)$ for every node x on the access path for s_i using the source graph $G(x) \setminus S$
3 Let the sources for u, v be s_u, s_v
4 Decide whether s_u, s_v are connected, as in Lemma 4.6.8

Figure 4.9: A decoder algorithm for separator labels on treewidth k graphs

reachability matrix for sources in $G(u) \setminus S$ (each node $u \in G$ corresponds to a unique source in $G(u)$). Then we use the reachability matrices for $G(x) \setminus S$ for nodes x on the access paths for $s_1, \ldots, s_k$ to compute new reachability matrices $G_S(x)$ for nodes in $Q(u), Q(v)$ (where the access paths for $s_1, \ldots, s_k$ become adjacent to the access paths for u, v).

We now proceed in a similar way to Lemma 4.6.8; we use the new information $G_S(x)$ for $x \in Q(u) \cup Q(v)$ to construct subpaths, then join them together using the parallel composition operations in the term tree. However, we place a restriction on the sources that we are allowed to use in our subpaths – if s_i is the source corresponding to a node $s \in S$ (recall that we add subscripts to make identification easier) then we are not allowed to use the source s_i in the subpaths. More precisely, for each node x in the term tree we construct subpaths using $G_S(x)$. Since there are at most $2^{k'}O(\log n)$ reachability matrices adjacent to each access path, the labels are of size $O(2^{k'}k'^2 \log n) = O(2^{3k}k^2 \log n)$ bits. The decoder algorithm is shown in Figure 4.9.

We can now employ the same argument as before to turn our separator labels into distance separator labels, with an additional $O(\log n)$ factor in the label size. We do this as before: replace each of the $2^{k'}$ reachability matrices by a matrix storing the distance between sources in the graph where some set S of sources have been removed. All the distances used involve paths that avoid the sources representing nodes in the forbidden set S, and therefore the paths that remain are exactly those that do not intersect S.

CONSTRUCT-SOURCE-CONNECTIVITY-GRAPH(G)

```
1  while (G contains a non-source node)
2      do Let u be any non-source node in G
3         Add edges {x, y} between all neighbours x, y of u
4         Remove u from G
```

Figure 4.10: The procedure to construct the source connectivity graph

Reducing the space requirements using source connectivity graphs

For each node v and its graph $G(v)$, we can avoid storing $2^{k'}$ matrices by constructing a graph $G'(v)$ on the k' sources, which we call the *source connectivity graph*. This graph will have the property that for any set S of source nodes, the reachability of sources in $G'(v) \setminus S$ equals the reachability of sources in $G(v) \setminus S$. Therefore, we can replace the $2^{k'} = O(2^{3k})$ reachability matrices by a single graph on k' nodes.

Constructing the graph is easy – for any path between two sources, we contract all its subpaths containing only non-source nodes into a single edge. More precisely, we want to solve the following problem. Given a graph G on n nodes and having k distinguished source nodes, construct a graph G' on the k source nodes so that the following holds: for any set $S \subseteq \{1, \ldots, k\}$ of sources and two sources s_i, s_j, we want that $1s_i$ and s_j are connected in $G \setminus S$ iff they are connected in $G' \setminus S$. We can construct G' using the procedure shown in Figure 4.10.

For each non-source node u, the procedure turns the neighbourhood of u into a clique then removes u; if u has only one neighbour then this does nothing. The graph remaining at the end of the procedure is the desired graph G'. It is easy to check that the nodes of G' are exactly the source nodes in G (they are the only nodes never contracted). We now show that the graph it computes has the property described above, i.e. it captures the connectivity of sources when only sources are removed. An example is shown in Figure 4.11.

Lemma 4.6.9 *The graph computed by the above procedure has the desired property, i.e for any set $S \subseteq \{1, \ldots, k\}$ of source nodes and two sources $s_i, s_j \notin S$, we have that s_i and s_j are connected in $G \setminus S$ iff they are connected in $G' \setminus S$.*

Proof. We begin by making two claims about the graph computed by the procedure.
Claim 1: The graph G' contains no paths not in G. Every path in G' corresponds to some

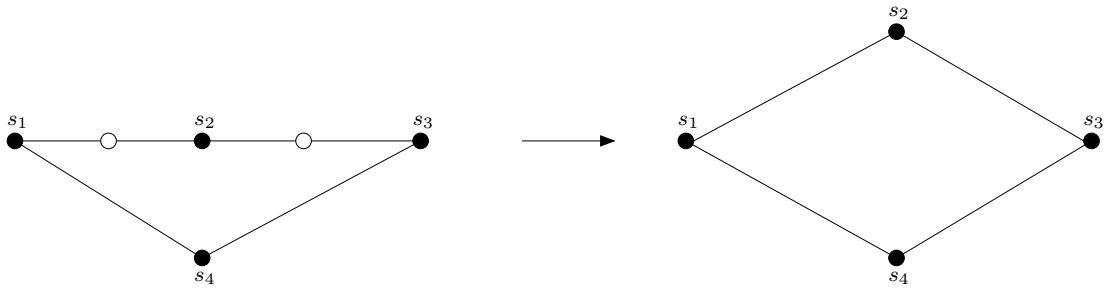


Figure 4.11: A graph and its source connectivity graph. The source connectivity graph preserves reachability between sources under deletion of sources. The source nodes are filled and the non-source nodes are drawn unfilled.

(possibly non-unique) path in G. Consider a path $s_{i_1}s_{i_2}\ldots s_{i_r}$ in G'. Then it is easy to see that there must exist a path $s_{i_1}P_{i_1i_2}s_{i_2}P_{i_2i_3}\ldots P_{i_{r-1}i_r}s_{i_r}$ in G where the P_{ij} are paths in G containing only non-source nodes.

Claim 2: The source connectivity graph G' contains all paths between sources in G. Since all non-source paths P_{ij} in G between s_i, s_j are contracted into a single edge in G', every path of the form $s_{i_1}P_{i_1i_2}s_{i_2}P_{i_2i_3}\ldots P_{i_{r-1}i_r}s_{i_r}$ in G corresponds to the unique path $s_{i_1}s_{i_2}\ldots s_{i_r}$ in G'.

The lemma now follows from the claims. If two sources are connected in $G \setminus S$ then there exists a path between them that avoids the sources in S, and by claim 2 there also exists a path in G' that avoids S. If two sources are connected in $G' \setminus S$ then there exists a path between them that avoids S, and by claim 1 there also exists a path between them in G that avoids the sources in S, and possibly uses some non-source nodes. Since the set S only contains source nodes, these paths P_{ij} still exist in $G \setminus S$. □

Using the above construction gives the following labeling scheme: construct the labels as before, except that we store the 'source connectivity graph' for $G(u)$ in place of the $2^{k'}$ reachability matrices for sources in $G(u)$. Note that for each node u and set of sources S, we can take the source connectivity graph for $G(u)$ and compute the reachability matrix for sources in $G(u) \setminus S$. Therefore, the decoder algorithm can simulate the decoder algorithm using the reachability matrices. This reachability matrix can be computed in time $O(k'^2 \log k') = O(k^2 \log k)$ by running an all-pairs shortest path algorithm on the source connectivity graph and ignoring nodes in S.

For distance separator labels, we shall show how to assign weights to the edges of the source connectivity graph such that the minimum weight path in the source connectivity graph between

CONSTRUCT-SOURCE-DISTANCE-GRAPH(G)

1 Set $w(u,v) = w(v,u) = 1$ for all edges $\{u,v\}$ of G, and ∞ otherwise
2 **while** (G contains a non-source node)
3 **do** Let u be any non-source node in G
4 For each pair of neighbours x, y of u
5 Set $w(x,y) = w(y,x) = \min\{w(x,u) + w(u,y), w(x,y)\}$
6 Remove u from G (also set $w(v,u) = w(u,v) = \infty$ for all v)

Figure 4.12: The procedure to construct the source distance graph

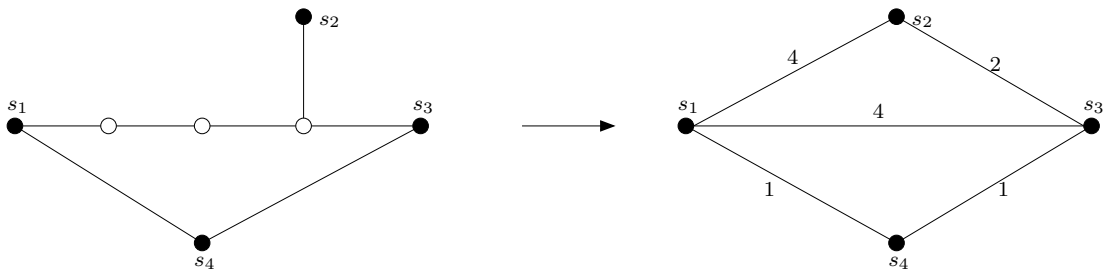


Figure 4.13: An example of a graph and its source distance graph. The distances between sources are preserved under deletion of source nodes. Note that deleting the source node s_4 increases the distance between s_1, s_3.

s_i, s_j equals the length of the shortest path between s_i, s_j in G that only uses non-source nodes. Furthermore, we show that this property is preserved under deletion of sources. We call this graph the source distance graph, and the algorithm to construct it is given below (as before, the input graph G has a distinguished set of k source nodes).

The graph G' constructed by the above procedure has the following property: for any set S of source nodes and two sources $s_i, s_j \notin S$, $d_{G' \setminus S}(s_i, s_j) = d_{G \setminus S}(s_i, s_j)$. The reason that we only contract edges connected to non-source nodes is that otherwise there may be an edge in G' that represents a path containing a source node s_j, and then setting $s_j \in S$ would give an incorrect distance using this edge. An example of the construction is given in Figure 4.13.

For each graph $G(u)$ in the term tree, we can use its source connectivity graph to reconstruct the distance matrix for sources in $G(u) \setminus S$ for any set S of sources. This allows us to simulate the scheme where we explicitly construct the $2^{k'}$ distance matrices, so we can use the argument in that case to construct separator distance labels and to argue for its correctness. Since the edge

weights in the source connectivity graph may be in the range $[1, n]$, the graph can be represented using $O(k'^2 \log n)$ bits. Therefore the distance separator labels are of size $O(k^2 \log^2 n)$ bits. We have now proved the following result.

Theorem 4.6.10 *The family of treewidth k graphs has separator labels of size $O(k^2 \log n)$ bits, and distance separator labels of size $O(k^2 \log^2 n)$ bits.*

Remarks

Unfortunately, the problem of determining whether the treewidth of a given graph is at most a given integer k is NP-complete [ACP87] (although, for constant k there exist linear-time algorithms [Bod93a]). Our algorithm works even when the tree decomposition given to it is not optimal – the only cost we pay is that the k in the label size becomes the treewidth of the decomposition given to the algorithm. There exist polynomial-time approximation algorithms that compute tree decompositions with treewidth a factor $O(\sqrt{\log k})$ of optimal, where k is the optimal treewidth of the graph [Ami02, FHL05]. Therefore for graphs having non-constant treewidth k, we can use an $O(\rho)$-factor approximation algorithm to obtain distance separator labels of size $O(\rho^2 k^2 \log^2 n) = O(k^2 \log^2 k \log^2 n)$ bits in polynomial time.

4.7 A partial forbidden-set routing scheme for planar graphs

We now show how to utilise some of our results for small treewidth graphs to obtain results for some special types of planar graphs. Eppstein [Epp00], improving on a previous result of Baker [Bak94], showed the following connection between bounded-genus graphs and treewidth, known as the 'diameter-treewidth property'.

Lemma 4.7.1 (Eppstein [Epp00]) *Let G be a graph with genus g and diameter D. Then G has treewidth $O(gD)$.*

It follows immediately from Theorem 4.6.10 on treewidth k graphs that genus-g graphs with diameter D have distance separator labels of size $O(g^2 D^2 \log^2 n)$ bits.

Eppstein [Epp95] considered the following problem on planar graphs: given a nonegative integer l, construct a centralized data structure so that given two nodes, we can decide if their distance is at most l, and if so, construct a path between them. We now show how to obtain

a similar result for the case of forbidden-set routing, but using a distributed data structure (the labels). Fix some nonnegative integer l. We shall assign labels to nodes so that given the labels for u, v and the nodes of $S \subseteq V(G)$, we can either return the distance $d_{G\setminus S}(u, v)$ or determine that it is greater than l. Once we can do this, we can use the labels with the routing scheme of Section 4.5 to route on a shortest path in $G \setminus S$ if $d_{G\setminus S}(u, v) \leq l$. Since these labels represent a restricted version of distance separator labels, we shall call them 'distance-l separator labels'. We will make use of the following planar graph covering result of Eppstein.

Lemma 4.7.2 (Eppstein [Epp95]) *Let G be a planar graph and l a nonnegative integer. Then in time $O(n)$ we can find a collection of subgraphs G_i with the following properties:*

1. *For every node v of G, the l-neighbourhood[2] of v is contained in one of the subgraphs G_i;*
2. *Every node of G is included in at most two subgraphs G_i;*
3. *Every subgraph G_i has treewidth $O(l)$.*

By applying Theorem 4.6.10 separately to each subgraph G_i we can construct distance-l separator labels of size $O(l^2 \log^2 n)$ bits. If $d_G(u, v) \leq l$ for two nodes u, v then the only way that $d_{G\setminus S}(u, v) > d_G(u, v)$ for some set S is if some nodes of S are within a distance l from u in G. Therefore, if the distance in $G \setminus S$ is at most l then it suffices to consider only those elements of S that lie within a distance l of u. The above lemma guarantees that we shall only have to consider a single subgraph to do this. Therefore, we have the following result.

Theorem 4.7.3 *Let G be a planar graph and $S(u) \subseteq V(G)$ the forbidden set of node u, with $k \leq \max_u |S(u)|$ for all u. Let l be a nonnegative integer such that $d_G(u, S(u)) \leq l$ for all u. Then we can construct a distributed forbidden-set routing scheme such that for any u, v, we can route on the shortest path that avoids $S(u)$, or declare that their distance in $G \setminus S(u)$ is greater than l. The routing tables have $O(k\Delta(G)l^2 \log^2 n)$ bits and the labels $O(kl^2 \log^2 n)$ bits.*

Remarks. Ideally, we would like to have a scheme that can route between *all* pairs of nodes, still with the restriction that $d(u, S(u)) \leq l$ for all u. However, the problem is that even if $d_{G\setminus S(u)}(u, v) > l$, we could still have that $d_{G\setminus S(u)}(u, v) > d_G(u, v)$. We would need to be able to know which node x on the 'fringe' of the subgraph G_i containing u that we should

[2]The l-neighbourhood of a node v is the set of nodes at distance at most l from v.

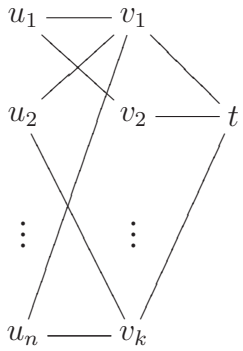


Figure 4.14: Illustrating the lower bound for separator labels (Proposition 4.7.4). Nodes u_i, v_j are not adjacent iff the set $V \setminus \{v_j\}$ is a separator of u_i and t.

route to, in order to reach v on the shortest $S(u)$-avoiding path. We could then route to x using the forbidden-set routing scheme in G_i, and then from x to v using any shortest-path routing scheme.

4.7.1 Lower bounds for distance separator labels

We now prove an easy lower bound on the size of separator labels by a reduction from adjacency labeling. We shall parametrise our lower bound by k, the maximum size of a separator that we are interested in detecting. The motivation for this is that k would correspond to the maximum size of any forbidden set in a forbidden-set routing scheme.

Proposition 4.7.4 *Assume that we are only interested in detecting separators of size at most $k \leq n$. At least one node must be assigned a separator label of size $\Omega(k)$ bits on n-node graphs in the worst case.*

Proof. Let $G = ((U, V), E)$ be an undirected bipartite graph on the node sets U, V where $U = \{u_1, \ldots, u_n\}, V = \{v_1, \ldots, v_k\}$. Construct G' by adding a node t connected to each node of V as in Figure 4.14. Now consider two nodes $u \in U$ and $v \in V$. It is clear that u is not adjacent to v in G iff $V \setminus \{v\}$ is a separator between u and t in G' (we can use the separator labels for the set $V \setminus \{v\}$ and u, v to decide whether this is the case). There are $\Omega(2^{nk/2})$ distinct bipartite graphs G, and for any two such graphs G_1, G_2 there are two nodes u, v where u, v are adjacent in G_1 but not in G_2. The corresponding graphs G'_1, G'_2 must also have two nodes u, v where $V \setminus \{v\}$ does not separate u, t in G'_1 but does in G'_2. Therefore the sum of all the labels

assigned to nodes must be at least $\Omega(\log 2^{nk/2}) = \Omega(nk)$ bits, so some node must be assigned a label of size $\Omega(k)$ bits. □

Remarks. Even when k is unrestricted, the lower bound does not appear to be tight. By storing the entire graph in each label, we can get a trivial $O(n^2)$ upper bound. Unfortunately, nothing better is known for general graphs and so the $\Omega(n)$ lower bound leaves a large gap. It is worth examining the lower bound to see that it is most likely far from tight. In particular, it does not seem to make good use of the combinatorial nature of the problem since each node v is implicitly associated with its witness set $V \setminus \{v\}$. For this reason, we expect that the lower bound can be strengthened to $\Omega(n^{3/2})$ or even $\Omega(n^2)$ but we have been unable to do so. Most likely, the current construction will not suffice and some more insight into how the structure of the forbidden sets affects the connectivity of the graph will be needed. The interesting (but seemingly difficult) case is when k is small, say $O(\log n)$. In this case, it would be very interesting to show that we can construct sublinear-sized separator labels for general graphs.

For the case of distance separator labels, the situation is somewhat different – if the set S is empty then it reduces to the problem of distance labeling in undirected graphs. It is known that there are graphs on n nodes and m edges where some node must be assigned a label of size $\Omega(m/n \log(n^2/m))$ bits [CHKZ02]. Therefore, distance separator labels must be of size $\Omega(n)$, regardless of the size of the set S allowed. This immediately gives the following lower bound.

Proposition 4.7.5 *There are n-node graphs where some node must be assigned a distance separator label of size $\Omega(n)$ bits, regardless of the size of the set S.*

Since distance separator labels are more general than separator labels, the remarks made above for separator labels also apply here; in particular, we do not expect that $\Omega(n)$ is a tight lower bound for distance separator labels in general graphs. Unfortunately, we have not been able to prove anything stronger and improving this is a completely open problem.

4.8 Decremental graph connectivity

In this final section, we use some of the techniques for constructing separator labels, combined with a novel reduction to orthogonal range searching, to show how to solve dynamic graph connectivity with good worst-case query time. We then show how to use this technique to construct efficient algorithms for solving the problem of k-edge witness.

Definition 4.8.1 (k-vertex witness) *Given a graph G, the k-vertex witness problem is to preprocess G so that given k nodes S, we can quickly decide whether S is a separator of u, v in G, i.e. whether S is a witness to the fact that u, v are not k-connected in G.*

Separator labels can be seen as a distributed version of the k-vertex witness problem. The k-edge witness problem is defined similarly but we want to know whether u and v are k-edge-connected. These two problems are closely related to the well-studied problem of *decremental graph connectivity*. Here, we wish to construct a data structure that efficiently supports the following operations on a graph: DELETE(u, v), which deletes edge $\{u, v\}$ from the current graph, CONNECTED(u, v), which returns TRUE iff u, v are still connected. The node version of the problem is similar, except that we instead support DELETE(u), which deletes a single node and all its adjacent edges. It is easy to see that k-vertex witness can be solved using a decremental connectivity algorithm – delete the set S of nodes, test connectivity of u, v and then reinsert the nodes deleted. In fact, the best known bounds for k-vertex witness (and k-edge witness) are obtained in this way.

We are interested in worst-case bounds because the problems we are trying to solve are fundamental network problems and therefore are most likely to be used as subroutines in higher-level applications. Without a good worst-case bound on the performance of the underlying algorithms, it is difficult for algorithms that use them to provide good performance guarantees of their own. Despite much work, the best known worst-case time for DELETE(u, v) is $O(\sqrt{n})$ due to Eppstein et al. [EGIN97] who improved the result of Frederickson [Fre83] from $O(\sqrt{m})$ to $O(\sqrt{n})$ per update using the sparsification technique.

All known algorithms that have better update time have *amortized* time bounds. The first algorithm with polylogarithmic update time was given by Henzinger, King and Thorup [HK99, HT97]. They gave a fully dynamic algorithm (supporting both insertions and deletions) such that for a sequence of $\Omega(m_0)$ update operations (where m_0 is the number of edges in the initial graph), an update takes expected amortized time $O(\log^2 n)$ and a connectivity query takes time $O(\log n / \log\log n)$. This gives an algorithm for k-edge witness with update time $O(\log^2 n)$ and amortized expected query time $O(k \log^2 n)$. This query time is amortized over the updates made, so this is not a worst-case bound for a single k-edge witness query.

Holm et al. [HdLT01] obtained a deterministic version of the algorithm with $O(\log^2 n)$ amortized time per update. However, as before, this time bound is amortized over a large sequence of edge insertions followed by deletions. Therefore, there is no guarantee that the cost

of a deletion will be small when taken over some isolated set of k edge deletions.

We improve the worst-case bound for k-edge witness in general graphs when the number of deletions is fairly small, i.e. $k = O(\sqrt{n})$. Our algorithms are simple, and reduce the problem of maintaining decremental connectivity to maintaining fully dynamic connectivity (supporting both insertions and deletions) on some auxiliary graph H, which usually has size linear in the number of deletions already performed. We can then use known algorithms to maintain connectivity on H, and answer queries on G by quickly translating them to queries on H. An artifact of our approach is that the time for a deletion depends on the number of nodes or edges already deleted from the original graph, which explains why it only works for small numbers of deletions. Our main results for this are the following:

- We give an algorithm for decremental connectivity that handles the kth edge deletion in worst-case time $O(k \log n)$ and answers connectivity queries in time $O(k^2)$. The downside is that it may use space $O(n\Delta(G)^2)$.

- Using the above algorithm, we solve k-edge witness in general graphs with worst-case query time $O(k^2 \log n)$ and space $O(k^2 n^2)$. This improves the $O(k\sqrt{n})$ bound of Eppstein [EGIN97] for $k = O(\sqrt{n})$. Our algorithm uses a novel reduction from orthogonal range searching.

- Let T be a spanning tree of G with degree $\Delta(T)$. We give a decremental connectivity algorithm using space $O(n^2 \log n / \log\log n)$, handling the kth deletion in time $O(\Delta(T)^2 \log n + k \log n)$ and connectivity queries in $O(k^2)$. For Hamiltonian graphs, graphs with bounded independence number, $1/O(1)$-tough graphs and almost all r-regular graphs (for fixed $r \geq 3$), this gives $O(k \log n)$ time for deletions.

4.8.1 Preliminaries

We begin by describing the algorithm of Henzinger and King (HK) [HK99], since our algorithm works in a conceptually similar way. They achieve both polylogarithmic update and query time but this bound is expected and amortized over $\Omega(m_0)$ updates, where m_0 is the number of edges in the initial graph. We remove this amortization but at the cost of additional space and an update time that depends linearly on the the number of edges deleted thus far.

They maintain a spanning forest of the graph, starting with some arbitrary spanning tree T (we assume that the graph is initially connected). When a tree edge e is removed from T it

breaks T into two subtrees T_1, T_2; and a replacement edge e' for e needs to be found (if one exists) to reconnect T_1, T_2 into another spanning tree $T' = T_1 \cup T_2 \cup \{e'\}$. To do this, they maintain a partition of the edges into $O(\log n)$ levels; to find a replacement for e, the nontree edges in a particular level are randomly sampled. If one of them connects T_1 to T_2, then the trees are reconnected using this edge. Otherwise, all the nontree edges adjacent to nodes of T_1 are searched exhaustively. By carefully managing this partitioning and sampling, they obtain good amortized bounds on the update times.

They also employ a technique to efficiently represent the trees in a linear form, which allows trees to be efficiently spliced or reconnected at a given edge. This data structure is known as the *Euler tour tree*. Since we also make use of it, we shall now describe it.

Euler tours

An Euler tour of a graph is a path that traverses every edge exactly once in each direction. Henzinger and King [HK99] use an Euler tour of a spanning tree T of G, constructed by calling the following procedure with the root node.

```
ET(v)
    ▷ Constructs an Euler tour of the tree
1   visit v
2   for each child u of v
3       do ET(u)
4          visit u
```

Figure 4.15: Constructing an Euler tour of a tree

Each edge is visited twice (traversed once in each direction) and every degree-d node d times. Each time any node u is encountered in the tour, we call this an *occurrence* of u and denote the set of occurrences of u by $O(u)$. We shall refer to a particular occurrence by its unique position in the tour. If the sequence $ET(T)$ is stored in a balanced binary search tree, then one may insert an interval or splice out an interval (delete an edge of the tour) in time $O(\log n)$, while maintaining the balance of the tree.

Some of our algorithms use an Euler tour $ET(G)$ of the entire graph G instead of a spanning tree. In this case, we can use the well-known theorem of Euler that states that a graph has an

Euler tour iff every node has even degree. Therefore a simple trick to ensure that G has an Euler tour is to 'double up' each undirected edge so that it gets traversed once in each direction.

Sparse connectivity certificates

The concept of a sparse k-connectivity certificate is important for some of our algorithms. A *sparse k-connectivity certificate* for a graph G is a subgraph G' of G, containing at most kn edges, such that any cut of value at most k in G has the same value in the certificate. The idea of using such a certificate is that if we are only interested in detecting cuts (or separators, if we are in the node case) of size at most k, then without any penalty we can work on the sparse graph G' instead of the (possibly dense) graph G.

Nagamochi and Ibaraki [NI92] show how to construct a sparse k-connectivity certificate in linear-time. The problem is also known to be in NC [NH98] and can be solved using a distributed algorithm [Thu95, JM96].

4.8.2 The algorithm

We now present our algorithm for solving k-edge witness for general graphs using a centralized algorithm. As described earlier, Henzinger and King [HK99] use an Euler tour data structure to represent a spanning tree of G. In contrast, we shall construct an Euler tour of the *entire graph*.

The algorithm can be explained as follows. We maintain an auxiliary undirected graph H where we associate with each node of H a connected interval of $ET(G)$ (i.e. a connected subpath of the Euler tour) and the nodes of H form a disjoint partition of the subpaths of the tour. There is an edge between two nodes of H iff there is some node $u \in V(G)$ with an occurrence in both intervals. We denote by $h(u)$ a node of H whose interval in $ET(G)$ contains an occurrence of u (if there is more than one, choose one arbitrarily). For an integer i corresponding to an occurrence of a node, we denote by $h(i)$ the (unique) node of H whose interval on the tour contains i (the version used will be clear from the context).

For a node $u \in V(G)$, let H_u be the subgraph of H induced by the nodes whose intervals contain an occurrence of u. Let h_1, h_2 be any two nodes of H_u, then there must be an edge $\{h_1, h_2\}$. It follows that the subgraph H_u is a clique, for all u. We shall represent the graph H by storing a balanced binary search tree (e.g. a 2-3 tree) on the intervals associated with nodes of H. This allows us to find the node $h(u)$ in worst-case time $O(\log |H|)$. The following lemma states a simple property of H.

Lemma 4.8.2 *Nodes $u, v \in V(G)$ are connected in G iff $h(u), h(v)$ are connected in H.*

Proof. First, note that since H_u, H_v are cliques, we can choose to compute reachability between any pair of nodes $a \in H_u, b \in H_v$. The lemma now follows from the definition of H – every path from u to v in G corresponds to a set of paths from $h(u)$ to $h(v)$ in H, and every path from $h(u)$ to $h(v)$ in H corresponds to a collection of paths from u to v in G. □

Now we can describe our algorithm. H starts as a singleton representing the entire tour $ET(G)$. To delete an edge $\{u, v\} \in E(G)$ the Euler tour is spliced at this edge in time $O(\log n)$. This corresponds to splitting exactly one node h of H into two new nodes h_1, h_2 with $N(h) = N(h_1) \cup N(h_2)$. Therefore to construct the new edges of h_1 and h_2, we do not need to test for edges between all the nodes of H – it suffices to test only the old edges of h to see if they are also edges of h_1 or h_2. By definition of H, there is an edge between two nodes of H iff their corresponding intervals in the Euler tour both contain an occurrence of some node $u \in V(G)$. We shall show that this 'edge test' can be done in worst-case time $O(\log m) = O(\log n)$ for each edge by making use of orthogonal range trees.

To maintain connectivity on H under node insertions and both edge deletions and insertions we can use any fully dynamic connectivity algorithm. A simple method is to store the adjacency list representation of H; each edge insertion and deletion then takes time $O(1)$, and connectivity queries can be answered by running a depth-first search in time $O(|V(H)| + |E(H)|) = O(|H|^2)$. An alternative is to use the fully-dynamic algorithm of Eppstein et al. [EGIN97], which handles edge insertions and deletions in time $O(\sqrt{|H|})$ and answers connectivity queries in time $O(1)$.

To answer connectivity queries, we use the fact that the subgraph H_u is a clique, so one node of H_u can reach some node h of H iff all of H_u can reach h. A connectivity query for u, v is then handled by finding $h(u), h(v)$ and then calling CONNECTED$_H(h(u), h(v))$. To handle the query CONNECTED we simply call CONNECTED$_H$, since G is connected iff H is connected.

Reduction from orthogonal range searching

The crucial part of our algorithm is the ability to test for an edge in the auxiliary graph H. We do this by using a reduction to two-dimensional orthogonal range searching as follows. A *range tree* is a data structure that supports two operations on a two-dimensional space: INSERT(x, y), which inserts a point (x, y), and BOX-EMPTY$((x_1, y_1), (x_2, y_2))$, which returns true iff the box with corners (x_1, y_1) and (x_2, y_2) does not contain any points (sometimes we shall use

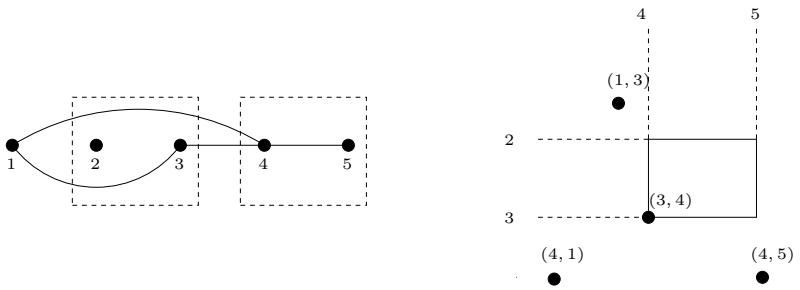


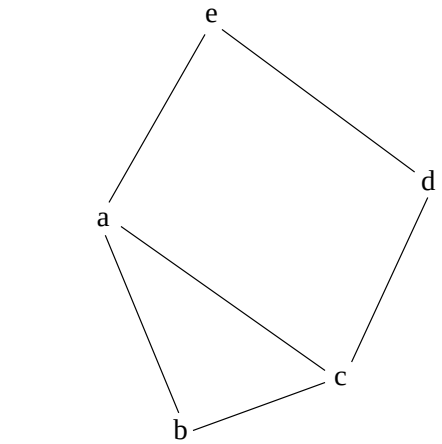
Figure 4.16: The box $(2, 4) \times (3, 5)$ is empty iff there are no edges between the intervals $[2\,..\,3]$ and $[4\,..\,5]$ on the graph

BOX-NOT-EMPTY for the negation of this function).

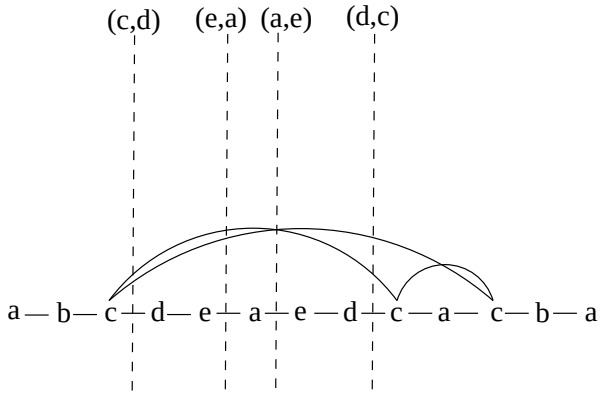
Given an undirected graph G and a unique identifier $I(u) \in [1\,..\,n]$ for each node u, each undirected edge $\{u, v\}$ of G is mapped to two points $(I(u), I(v))$ and $(I(v), I(u))$ on an $n \times n$ grid. Then there is an edge in G with endpoints having identifiers in both the intervals $[a\,..\,b]$ and $[c\,..\,d]$ iff the box $(a, c) \times (b, d)$ is nonempty (for $a \leq b, c \leq d$) as illustrated in Figure 4.16. Orthogonal range searching has been extensively studied during the last thirty years, with many applications to databases and computational geometry. There are several dynamic algorithms having efficient worst-case update and query times. This is the first time we know of that they have been used for graph connectivity. The transformation described above may also be of independent interest.

The transformation described above is used as follows, and illustrated in Figure 4.17. For each ordered pair of occurrences u_i, u_j of node u, add a point (u_i, u_j) to the space and associate with each node of H a unique interval $[i\,..\,j]$ (with $i \leq j$) on the Euler tour. Then the two nodes of H associated with the intervals $[a\,..\,b]$ and $[c\,..\,d]$ are adjacent in H iff there exists some node $u \in V(G)$ with occurrences in $[a\,..\,b]$ and $[c\,..\,d]$ in the tour, which occurs iff the box $(a, c) \times (b, d)$ is nonempty. The algorithm of this section does not need to remove points (splitting nodes of H keeps track of the deleted edges of G), so a static range tree algorithm will suffice (in contrast, the algorithm of the next section requires a dynamic range tree). Chazelle [Cha88] has given an algorithm for the static case that stores r points with space $O(r)$ and answers emptiness box queries in worst-case time $O(\log r)$.

The decremental connectivity algorithm is given in full in Figure 4.18, and the procedure for k-edge witness in Figure 4.19.



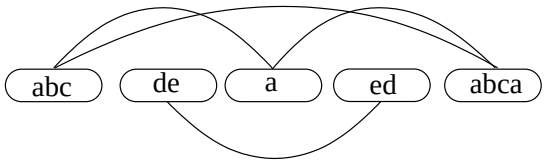
(a) The graph G



(b) An Euler tour of G

(only edges corresponding to occurrences of node c are drawn)

Deleting edges {a,e} and {c,d} cut the tour at the points indicated by the dashed vertical lines



(c) The graph H corresponding to deleting {a,e} and {c,d}.

The edges of H indicate regions of the tour containing occurrences of the same node

Figure 4.17: A graph G and an Euler tour of G with the edges between occurrences of the same node marked. The dashed lines represent the splicing of the tour from deleting edges $\{a, e\}$ and $\{c, d\}$. The auxiliary graph H at this point is shown below. There is no path between any nodes of H containing occurrences of e and b, therefore $\{a, e\}, \{c, d\}$ is a cut between e, b in G.

INITIALISE(G)

1 Double up each edge of G
2 $ET(G) \leftarrow$ an Euler tour of G
3 **for** each node u
4 **do** $O(u) \leftarrow$ the set of occurrences of u in $ET(G)$
5 **for** each pair $u_i, u_j \in O(u)$
6 **do** RANGE-TREE-INSERT(u_i, u_j)
7 Let H be a graph with a single node $[1\,..\,2m]$.

DELETE$(\{u, v\})$

1 **for** each appearance $\{i, i+1\}$ of edge $\{u, v\}$ in $ET(G)$
2 **do** EULER-TOUR-DELETE$(\{i, i+1\})$ $\triangleright\ O(\log n)$
$\triangleright$ Split h into two new nodes h_1, h_2
3 $h \leftarrow [a\,..\,b] = h(i)$ $\triangleright\ O(\log |H|)$
4 $h_1 \leftarrow$ INSERT-NODE$_H([a\,..\,i])$ $\triangleright\ O(1)$
5 $h_2 \leftarrow$ INSERT-NODE$_H([i+1\,..\,b])$
$\triangleright$ Check for an edge between h_1 and h_2
6 **if** BOX-NOT-EMPTY$((a, i+1), (i, b))$ $\triangleright\ O(\log n)$
7 **then** INSERT-EDGE$_H(h_1, h_2)$ $\triangleright\ O(1)$
$\triangleright$ Check for edges between $N(h)$ and $N(h_1), N(h_2)$
8 **for** each neighbour $h' = [c\,..\,d]$ of h in H
9 **do if** BOX-NOT-EMPTY$((a, c), (i, d))$ $\triangleright\ O(\log n)$
10 **then** INSERT-EDGE$_H(h_1, h')$ $\triangleright\ O(1)$
11 **if** BOX-NOT-EMPTY$((i+1, c), (b, d))$
12 **then** INSERT-EDGE$_H(h_2, h')$
13 DELETE-NODE$_H(h)$ $\triangleright\ O(\Delta(H))$

CONNECTED(u, v)

$\triangleright$ Returns true iff u, v are connected in G
1 **return** CONNECTED$_H(h(u), h(v))$ $\triangleright\ O(|H|^2)$

CONNECTED

$\triangleright$ Returns true iff G is connected
1 **return** CONNECTED$_H$ $\triangleright\ O(|H|^2)$

Figure 4.18: The decremental connectivity algorithm

k-EDGE-WITNESS$(u, v, \{x_1, y_1\}, \ldots, \{x_k, y_k\})$
1 $G' \leftarrow$ a sparse k-connectivity certificate of G
2 INITIALISE(G')
3 Let H contain a single node $[1\,..\,2m]$
4 **for** each $i \in \{1, \ldots, k\}$
5 **do** DELETE$(\{x_i, y_i\})$
6 **return** CONNECTED(u, v)

Figure 4.19: The algorithm for k-edge witness

Complexity

Let DELETE$[k]$ denote the worst-case time taken by DELETE for any sequence of k edges. Assume that k edges have been deleted from G. Then the graph H has at most $2k + 1$ nodes, since each edge of G appears twice in the Euler tour of G. Also, it takes time $O(\log n)$ to splice out an interval of the Euler tour, and each range tree query takes time $O(\log n)$. If we use the adjacency list representation of H, then the times for each line are as shown in the code above. The loop at line 6 of DELETE is repeated $O(\Delta(H)) = O(k)$ times and so DELETE$[k] = O(k \log n)$. Therefore the procedure k-EDGE-WITNESS takes worst-case time $O(k^2 \log n)$.

Alternatively, using the fully-dynamic connectivity algorithm of Eppstein et al. makes lines 4,5,8,9 take time $O(\sqrt{|H|})$ and line 11 takes time $O(\Delta(H)\sqrt{|H|})$, so DELETE$[k] = O(k^{3/2} + k \log n)$, but CONNECTED$(u, v)$ and CONNECTED now take $O(\log k)$ time. This might be more useful if the algorithm was being used for decremental connectivity. However, for solving k-edge witness, using the adjacency representation of H gives the best time bound.

The space requirement is dominated by the cost of storing the points representing the node occurrences in the range tree. Since each node $v \in V(G)$ appears $d_G(v)$ times in the tour, there are $d_G(v)^2$ points in the range tree corresponding to v. Using the range tree of Chazelle [Cha88] gives a data structure using $O(\sum_{v \in G} d_G(v)^2) = O(n\Delta(G)^2)$ bits of space.

For k-edge witness we are only interested in cuts of size at most k, and so we can reduce the space requirement by using sparse k-connectivity certificates. The modified algorithm is the same except that we replace G with its sparse k-connectivity certificate G' in INITIALISE. The correctness follows from the connectivity properties of the connectivity certificate. The space

complexity is reduced to that needed to store the node occurrences in the range tree for the Euler tour of the certificate, i.e. $O(\sum_{v \in V} d_{G'}(v)^2) = O(n^2 k)$ bits in the worst case that we have a complete bipartite graph on $2k$ nodes. The query time is unchanged, as it only depended on the set of edges being deleted.

Note that this idea can be applied to any decremental connectivity algorithm when we are only interested in cuts of small value. Since there are algorithms [Tho00] that use space $O(m)$, this would transform them to use space $O(kn)$ for solving k-edge witness.

4.8.3 An more space-efficient algorithm for tree-like graphs

In this section we reduce the space requirement of the previous algorithm but increase the query time for general graphs. For graphs having a spanning tree with small maximum degree, however, we shall show how to maintain a small query time.

Let T be a spanning tree of G having degree $\Delta(T)$. We give a decremental connectivity algorithm that uses space $O(n^2 \log n / \log\log n)$ and handles the kth deletion in time $O(\Delta(T)^2 \log n + k \log n)$. It answers connectivity queries in time $O(k^2)$ and therefore gives an $O(k\Delta(T)^2 \log n + k^2 \log n)$-time algorithm for k-edge witness, which is $O(k^2 \log n)$ for Hamiltonian graphs, graphs of bounded independence number, $1/O(1)$-tough graphs and almost all r-regular graphs (for fixed $r \geq 3$). As before we can use a different algorithm to maintain connectivity on the auxiliary graph, and this gives $O(\Delta(T)^2 \log n + k^{3/2} \log n)$ time for deletions but with $O(\log k)$ query time.

The algorithm is more similar to that of Henzinger and King [HK99] than the one of the previous section, in that we use a spanning forest rather than the Euler tour of the entire graph. The main difference is that instead of *maintaining* a spanning forest of G, we do not bother to replace tree edges when they are deleted. Instead we keep track of the fragmented parts of the forest as tree edges are deleted, and use this to answer connectivity queries efficiently.

Initialisation

The algorithm works as follows. Consider an undirected connected graph $G = (V, E)$ and a spanning tree $T = (V, F)$ of G. We construct the Euler tour $ET(T)$ of T (note that in the previous section we used the Euler tour of G), and maintain an undirected graph H whose nodes represent intervals on the Euler tour of T. As before, we build a range tree with a point for each

pair of occurrences of the same node u, i.e. the set $O(u) \times O(u)$. Therefore there are at most

$$\sum_{v \in V} deg_T(v)^2 \leq \left(\sum_{v \in V} deg_T(v) \right)^2 \leq (2n-2)^2$$

points since in any tree T we have $\sum_{v \in T} deg_T(v) = (n-1)$.

We must also handle the nontree edges of G. Assume that there are m' such nontree edges. We do this by adding to the range tree for each nontree edge $\{u, v\} \in G$, a point (u_i, v_j) for each pair of occurrences in $O(u) \times O(v)$. Since each node v appears in the Euler tour of T $deg_T(v)$ times, the number of nontree edge points is

$$\sum_{\{u,v\} \in E \setminus F} deg_T(u) deg_T(v) \leq \sum_{\{u,v\} \in E} deg_T(u) deg_T(v) = \left(\sum_{v \in V} deg_T(v) \right)^2 \leq 4n^2$$

for any spanning tree T. If the number of nontree edges is small (e.g. G is tree-like) then it makes sense to bound this by $\min(m' \Delta(T)^2, 4n^2)$ (by subtracting the sum containing the tree edges). Combining the space requirements for tree and nontree edges, the algorithm uses at most $\min(8n^2, (m'+n)\Delta(T)^2)$ points in total.

Deleting an edge

Assume that we represent the auxiliary graph H with its adjacency list (so that edge operations take $O(1)$ time). Deleting a tree edge is handled as before: we delete the two appearances of the edge from the Euler tour of T, each of which splits some node of H into two nodes. We then test for edges adjacent to the new nodes of H using emptiness queries on the range tree. This takes total time $O(k \log n)$ after k edges have been deleted (since the graph H will contain at most $O(k)$ nodes).

Deleting a nontree edge $\{u, v\}$ has no effect on the Euler tour of T, but now we must delete all the points $(u_i, v_j) \in O(u) \times O(v)$ from the range tree, corresponding to the edge $\{u, v\}$ and the occurrences of u and v in T. After deleting each point (u_i, v_j) we do a range query to check that an edge still exists in H between the nodes whose intervals contain u_i, v_j. If this check fails then we delete the corresponding edge from H. In total, this requires $O(\Delta(T)^2)$ emptiness queries in the worst-case.

Answering connectivity queries

Connectivity queries are answered exactly as before: to answer CONNECTED(u, v), we find $h(u)$ and $h(v)$ and check whether $h(u)$ can reach $h(v)$ in H. As before, the correctness of this follows from Lemma 4.8.2 and that the subgraphs H_u are cliques in H.

Complexity

Mortensen [Mor03] has given a dynamic range tree data structure that handles emptiness queries and deletions in worst-case time $O(\log r)$, and uses space $O(r \log r / \log\log r)$ to store r points. If we use this algorithm then we obtain a decremental connectivity algorithm that handles the kth deletion in worst-case time $O(\Delta(T)^2 \log n + k \log n)$ and uses space $O(r \log r / \log\log r)$, where $r = \min(8n^2, (m' + n)\Delta(T)^2)$.

It is clear that our algorithm relies on constructing a spanning tree of the graph having small maximum degree. In fact, it actually requires a spanning tree such that if $\{u, v\}$ is a nontree edge of G, then the product $deg_T(u)deg_T(v)$ should be small. There are several results known about graphs with spanning trees of small degree. Let $\Delta^*(G, T)$ be the smallest integer d such that G has a spanning tree of maximum degree d. Determining $\Delta^*(G, T)$ exactly is NP-hard, since $\Delta^*(G, T) = 2$ iff G has a Hamiltonian path, which is NP-complete [GJ90]. On the other hand, Furer and Raghavachari [FR94] give a polynomial-time approximation algorithm that outputs a spanning tree T with degree at most $\Delta^*(G, T) + 1$. A theorem of Dirac [Dir52] says that if each node of G has degree at least $n/2$, then G contains a Hamiltonian cycle, and therefore a spanning tree of degree 2. It is known that almost all r-regular graphs (for $r \geq 3$) have a Hamiltonian cycle.

An alternative characterisation of $\Delta^*(G, T)$ is in terms of the *toughness* of the input graph. A graph $G = (V, E)$ is *t-tough* if the number of connected components of $G \setminus S$ is at most $|S|/t$ for every separator $S \subseteq V$. In 1989, Win proved the following theorem.

Theorem 4.8.3 ([Win89]) *Let t be a positive integer. Every $1/t$-tough graph G has a spanning tree of degree $t + 2$, i.e. $\Delta^*(G, T) = O(t)$.*

Combining the above theorem with the algorithm of this section gives the following result.

Theorem 4.8.4 *Let G be $1/t$-tough. Then we can solve decremental connectivity on G, handling the kth deletion in time $O(t^2 \log n + k \log n)$ and answering connectivity queries in time*

$O(k^2)$. *The algorithm uses space* $O(n^2 \log n / \log\log n)$ *bits. Furthermore, only polynomial preprocessing time is required.*

The above theorem immediately implies that we can also solve k-edge witness in worst-case time $O(kt^2 \log n + k^2 \log n)$ on $1/t$-tough graphs.

Remarks. If the number of edges in $G \setminus T$ is zero (i.e. G is a tree) then we can ignore the analysis in the case of the nontree edges and so the kth deletion takes time $O(k \log n)$. Similarly, if the number of nontree edges is small (for example, G is 'tree-like'), it may be possible to do better. For example, a good randomized bound may be possible by considering the probability of deleting a nontree edge at each step. We leave it as an open problem to obtain such bounds.

```
INITIALISE
 1  T ← a minimum degree spanning tree of G
 2  ET(T) ← the Euler tour of T
 3  for each node u ∈ T
 4       do O(u) ← the set of occurrences of u in ET(T)
 5          for each pair of occurrences u_i, u_j ∈ O(u) × O(u)
 6               do RANGE-TREE-INSERT(u_i, u_j)
 7  for each nontree edge {u, v} ∈ G
 8       do for each pair of occurrences u_i, v_j ∈ O(u) × O(v)
 9               do RANGE-TREE-INSERT(u_i, v_j)
10  H ← a graph with a single node representing the interval [1 . . n].

DELETE({u, v})
    ▷ Assume v is the parent of u
 1  if {u, v} is a tree edge
 2    then for each appearance {i, i + 1} of edge {u, v} in ET(G)
 3              do EULER-TOUR-DELETE({i, i + 1})                    ▷ O(log n)
                 ▷ Split h into two new nodes h_1, h_2
 4                 h ← [a . . b] = h(i)                             ▷ O(log |H|)
 5                 h_1 ← INSERT-NODE_H([a . . i])                   ▷ O(1)
 6                 h_2 ← INSERT-NODE_H([i + 1 . . b])
                   ▷ Check for an edge between h_1 and h_2
 7                 if BOX-NOT-EMPTY((a, i + 1), (i, b))             ▷ O(log n)
 8                   then INSERT-EDGE_H(h_1, h_2)                   ▷ O(1)
                   ▷ Check for edges between N(h) and N(h_1), N(h_2)
 9                 for each neighbour h' = [c . . d] of h in H
10                     do if BOX-NOT-EMPTY((a, c), (i, d))          ▷ O(log n)
11                          then INSERT-EDGE_H(h_1, h')             ▷ O(1)
12                        if BOX-NOT-EMPTY((i + 1, c), (b, d))
13                          then INSERT-EDGE_H(h_2, h')
14                 DELETE-NODE_H(h)                                 ▷ O(Δ(H))
15    else  ▷ {u, v} is not a tree edge
16              do for each pair of occurrences u_i, v_j ∈ O(u) × O(v)
17                    do RANGE-TREE-DELETE(u_i, v_j)
                         ▷ Check for an edge between h(u_i) and h(v_j)
18                       h_1 ← [a . . b] = h(u_i)                   ▷ O(log |H|)
19                       h_2 ← [c . . d] = h(v_j)
20                       if BOX-EMPTY((a, c), (b, d))               ▷ O(log n)
21                         then DELETE-EDGE_H(h_1, h_2)             ▷ O(1)
```

Figure 4.20: The decremental connectivity algorithm for graphs having a spanning tree with low degree

CHAPTER 5

Handling Intermediate Nodes

The previous chapter considered the problem of routing from a source u to a destination v, using the lowest-cost path to u. However, the routing process relies on intermediate nodes forwarding packets towards the destination, possibly along paths that are of high cost to them. If we assume that the nodes are autonomous, competing organisations (such as the autonomous systems on the Internet), then these intermediate nodes may simply drop these packets. This type of behaviour is can be seen in the BGP Internet routing algorithm – nodes may choose to only advertise routes that are of low cost to them, so nodes can only discover routes where every subpath is also of low cost to its source. This problem does not arise in shortest-path routing, since every subpath of a shortest path is also a shortest subpath, and therefore the intermediate nodes will always route on shortest subpaths.

In this chapter, we extend the routing model of the previous chapter to take into account the costs incurred by intermediate nodes, when we use forbidden-set policies. We shall call a path P_{uv} a good path if all of it subpaths P_{wv} have zero cost, i.e. $c_w(P_{wv}) = 0$. We shall also assume that k is an upper bound on the size of a forbidden set, i.e. $k \leq \max_v |S(u)|$. Our main results are the following.

- Taking into account intermediate nodes is hard. Consider any routing scheme that can decide if there exists a good path *before* sending the packet. We show that such a scheme must assign labels to nodes (not just routing tables) of size $\Omega(\sqrt{n} + k \log n/k)$ bits. This

lower bound holds even for trees, and stands in contrast to the $O(k \log n)$ bound shown in the previous chapter for distance labels in trees. Since the labels are placed in the packet header, this makes routing on 'good' paths infeasible.

- We show that the lower bound is almost tight by giving a scheme that can decide if there is a good path in trees. Our scheme uses labels of size $\tilde{\Omega}(\sqrt{kn})$ bits and makes routing decisions in time $O(\log kn)$. We also show how to extend the scheme to compute a 2-approximation to the sum of costs along the path using $\tilde{O}(\sqrt{kn})$ bits per label.

- We show that it is possible to avoid the lower bound by not checking if there is a good path before sending the packet, and instead letting the packet return if it cannot be routed on a good path. In this case, we show a simple routing scheme that works in general graphs, and uses $\tilde{O}(k)$-bit labels. The price is that the packet headers may become large, and a packet may travel the diameter of the graph before being returned.

- Finally, we show how to construct centralized algorithms for the problem, with various time-space tradeoffs.

Recall that each node assigns a non-negative cost to every other node $c_u(v) = 1$ iff $v \in S(u)$ and 0 otherwise. Define the *cost to* u of a path P_{uv} as $c_u(P_{uv}) = \sum_{w \in P_{uv}} c_u(w)$, and the full cost (or simply cost) of P_{uv} as

$$c(P_{uv}) = \sum_{w \in P_{uv}} c_w(P_{wv}) \tag{5.1}$$

where P_{wv} is the subpath of P_{uv} from w to v. A good path is then a path with zero cost, and a 'forbidden-set-avoiding' (fs-avoiding) path is a path with cost zero to the source node. If there exists a good path from u to v then we say that u can reach v, and we call the problem of deciding if there is a good path *fs-reachability* for short.

5.1 An $\Omega(\sqrt{n})$ lower bound

Section 4.6.1 showed that trees enjoy $O(\log n)$-bit separator labels. Let us now consider the problem of deciding if there is a good path from u to v. The most obvious idea might be for $L(u)$ to store the separator labels for the forbidden sets of each of its ancestors w, and then to use these to check that none of them is a uv-separator. Unfortunately this scheme would require

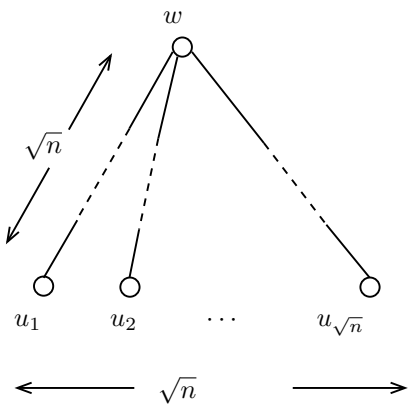


Figure 5.1: The $\sqrt{n}$ lower bound of Theorem 5.1.1

$\Omega(n)$ bit labels in the case of a line. We now show that labels of size $\Omega(\sqrt{n})$ bits are required to decide if the path is good in a tree.

Theorem 5.1.1 *Any labeling scheme for fs-reachability on n-node trees with forbidden sets of size k must assign to some node a label of size at least $\sqrt{n} + \Omega(k \log n/k)$ bits.*

Proof. Consider the tree as in Figure 5.1 with a set U of $\sqrt{n}$ leaves $u_1, ..., u_{\sqrt{n}}$ each linked to the root node w by node-disjoint paths of length $\sqrt{n}$. The forbidden sets are either empty or contain a single element from the set U of leaves. Now, u_i can reach u_j (for $i \neq j$) iff there is no node on the path u_i to w whose forbidden set contains u_j. By assigning forbidden sets in this way to the nodes on the paths, the nodes on each path from u_i to w can collectively select one of $\sum_{j=0}^{\sqrt{n}} \binom{\sqrt{n}}{j} = 2^{\sqrt{n}}$ subsets of U, and for each distinct choice, a distinct labeling of the set $U \cup \{u_i\} = U$ is required.

Since the paths from the $\{u_i\}$ to the children of w are node-disjoint, the labelings required are independent for each path (since the forbidden sets chosen on node-disjoint paths will not interfere with the output of the decoder) and so we can apply the above argument independently to each path – for each of the $2^{\sqrt{n}}$ distinct choices of nodes on the path from u_1 to w, there are $2^{\sqrt{n}}$ distinct choices of nodes on the path from u_2 to w (and each of these requires a distinct labeling) and so on. Hence there are at least $\underbrace{2^{\sqrt{n}} 2^{\sqrt{n}} \ldots 2^{\sqrt{n}}}_{\sqrt{n}} = 2^n$ distinct labelings of the nodes of U. As $|U| = \sqrt{n}$, it follows that at least one node in U must be assigned a label of size $\frac{1}{\sqrt{n}} \log 2^n = \sqrt{n}$ bits.

This bound holds for $k = 1$ and we have been unable to extend it to depend on k. However, we can combine it with the lower bound of Theorem 4.5.3 to obtain an $\Omega(\sqrt{n} + k \log n/k)$ bound, i.e. when k is approximately greater than $\sqrt{n}$ the lower bound grows linearly with k. □

This lower bound rules out the possibility of extending the separator label scheme of the previous chapter with only a small (polylogarithmic) increase in size. For the tree in Figure 5.1, it is quite easy to achieve a matching upper bound: for each node $u \in U$, the label $L(u)$ stores (using $O(\sqrt{n})$ bits) for every other node $v \in U$, whether u has a good path to v. Unfortunately, this strategy is doomed to fail for the complete binary tree, where u would end up storing reachability separately for $\Omega(n)$ leaves.

5.2 A $\tilde{O}(\sqrt{kn})$ upper bound

In this section we show an almost optimal upper bound on the label size for trees. Define $f_T(u, v) = 1$ iff there is a good path from u to v in T. First we look at two simple cases from which we derive the scheme for general trees.

Line scheme. On the line, labels of size $3\lceil \log n \rceil$ bits suffice: number the nodes from left to right, then store in $L(v)$ the position of v and the positions of the two closest nodes left(v) and right(v) that cannot reach v, from each side of v. Given $L(u)$ and $L(v)$, the decoder declares that u can reach v iff u lies between left(v) and right(v). Notice that the label size is independent of k, the size of the forbidden sets.

Tree scheme. Next, consider a complete binary tree on n nodes – each of the $n/2$ leaves may be independently unreachable from u so listing these regions as for the line will use $\Omega(n)$ bits. However, $O(kh \log n)$ bit labels suffice for a tree of height h: the label for u stores, for every ancestor w of u, $\langle f(u, w), f(w, u), L_{SEP}(S(w))\rangle$, where $L_{SEP}(S(w))$ are the separator labels for $S(w)$ in the tree. Given $L(u)$ and $L(v)$, the decoder finds the least common ancestor w of u and v and checks that $f(u, w) = 1, f(w, v) = 1$ and that none of the forbidden sets on the path u to w are uv-separators in the tree. This scheme is clearly inefficient for a long path.

The above discussion shows that, while lines and complete trees have efficient schemes, each fails on the other case. We now show how to tradeoff between the two schemes to obtain a scheme with labels of size at most $\tilde{O}(\sqrt{kn})$ bits. We first need some preliminary definitions.

A *separator* of a rooted tree T is a node w whose removal partitions the tree into connected components, each with at most $n/2$ nodes. In 1869, Jordan proved that such a node always

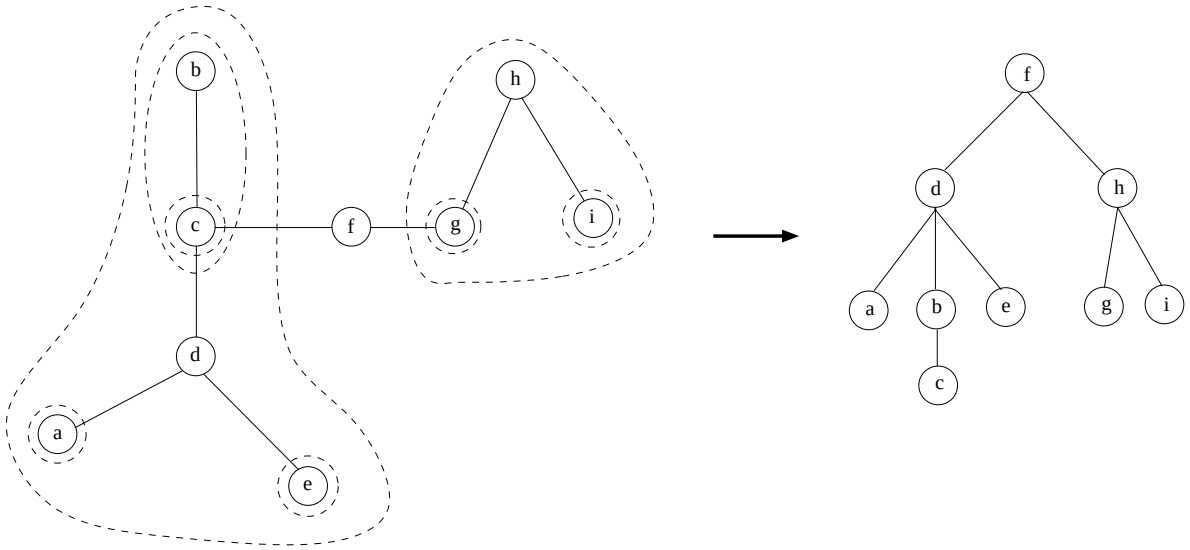


Figure 5.2: An example of a separator tree.

exists and can be obtained in the following way. Pick an arbitrary node u from the tree. If v partitions the tree into components of size at most $n/2$ then we are done. Otherwise, there exists a component with more than $n/2$ nodes – let u be the node adjacent to v in that component, change v to u and repeat the process. Since there are a finite number of nodes in the tree, and each node is visited at most once, this process terminates in linear time and finds a separator w.

The separators can be used to build a *separator tree* T' for T as follows. First, find a separator w of T and make it the root of T'. Then, recursively construct the separator trees of the components of $T \setminus \{w\}$ and make the roots of these trees the children of w in T'. It is clear that the separator tree has depth $O(\log n)$ and can be constructed in time $O(n \log n)$ since we can find the separator nodes in time linear in the size of the subtrees. An example of a separator tree is illustrated in Figure 5.2. We also need the notion of a painting. A *painting* χ of a tree T partitions the nodes of T into disjoint subsets $\mathsf{Shallow}(\chi)$ and $\mathsf{Deep}(\chi)$. An α*-painting* is a painting with the following properties:

1. The shallow nodes induce a connected subtree of T with at most n/α leaves;

2. For every node $v \in T$, there are at most α deep nodes on the path from v to the root.

The following 'painting lemma' is key to the labeling scheme and shows how we can tradeoff the space required between the deep and shallow nodes.

Lemma 5.2.1 *For any n-node tree T and any integer $1 \leq \alpha \leq n$, there exists an α-painting of T, computable in linear time.*

Proof. Assume that the tree is rooted; if not then choose a root node arbitrarily. We claim that the following procedure is suitable: for all subtrees of T having depth exactly α, paint all their nodes deep[1]. All remaining nodes (above these subtrees) of T are painted shallow.

Condition (1) of the definition of a painting implies that all the shallow nodes must be above all the deep nodes. Now we just have to look for the frontier between the shallow nodes (above) and the deep nodes (below). Consider all the nodes whose subtrees have depth exactly α. Each such subtree has at least α nodes in it, so there may be at most n/α such subtrees in the tree. If we paint all these subtrees deep and everything above them shallow, then (1) is clearly satisfied and the subtree induced by the shallow nodes contains at most n/α leaves.

It is easy to see how to do the painting in time $O(n)$: do a depth-first search from the root of the tree, keeping track of the height of nodes on the current path. On the way back up, if the height of a node is at most α, then paint it deep, otherwise paint it shallow[2]. This takes time $O(|V| + |E|) = O(n)$. □

For some set $X \subseteq V$ of nodes, we define $X^* \subseteq X$ as the *ancestor-free subset* of X. X^* is the unique maximal subset of X where no distinct pair of nodes in X^* are ancestors of each other. We can define it as $X^* = \{x \in X | \nexists y \in X : y \neq x$ and y is an ancestor of $x\}$. For example, if X is a set of leaves then $X^* = X$, and if X is a path then X^* is the node of X closest to the root.

In the remainder, we shall use the following notation: $T[w]$ is the tree T rerooted at w, $id_T(w)$ is the identifier of node w in a depth-first search of a rooted tree T, and $\mathcal{A}_T(X)$ is the set of ancestor labels for a set $X \subseteq V$ in a tree T.

The labeling scheme

We now describe the labeling scheme for trees. The idea is to first compute the separator tree for T. Then for every ancestor w of u in the separator tree, we reroot T at w and do the following to $T[w]$: for all the deep ancestors of u, we apply the strategy for the binary tree (i.e. $L(u)$ contains the separator labels for their forbidden sets); for the remaining shallow nodes, we apply the strategy for the line to each path of shallow nodes from the root to a shallow leaf.

[1] The depth of a subtree is the length of its longest path from the root to a leaf.

[2] The height of a node is the minimum distance from it to a leaf.

MARKER(T):
Let $n = |V(T)|$ and $k = \max_{v \in T} |S(v)|$.

1. If T contains a single node w, set $L(w) \leftarrow (1, \{\}, \{\}, f_T(w,w), f_T(w,w), 0)$ and return.
2. Find and remove a separator w of T, breaking T into subtrees $\{T_i\}$ of size $\leq |T|/2$.
3. Recursively call $\mathbf{M}(T_i)$; let each node $v \in T_i$ be given the label $L_i(v)$.
4. Let χ be a $\sqrt{n/k}$-painting of $T[w]$ i.e. paint the nodes of T as if it were rooted at w.
5. For each node $v \in T_i$ do the following:
   (a) Let $\mathsf{Deep}(v) = S(\mathsf{Deep}(\chi) \cap P_{vw})$ be the forbidden nodes of the deep nodes on the path from v to w.
   (b) Let $\mathcal{S} = \{u \in (\mathsf{Shallow}(\chi) \setminus T_i) \mid f_T(u,v) = 0\}$ be the shallow nodes that cannot reach v and let $\mathsf{Shallow}(v) = \mathcal{S}^* \subseteq \mathcal{S}$ be its ancestor-free subset, using the ancestor relation from $T[w]$.
   (c) Construct the sublabel $\mathcal{J}(v)$ for node v as follows:
   $$\mathcal{J}(v) \leftarrow (id_T(v), \mathcal{A}_{T[w]}(\mathsf{Deep}(v)), \mathcal{A}_{T[w]}(\mathsf{Shallow}(v)), f_T(v,w), f_T(w,v), i)$$
   (d) Append this to v's label in the component T_i by doing $L(v) \leftarrow \mathcal{J}(v) \circ L_i(v)$

Figure 5.3: The marker algorithm

The marker algorithm. The marker algorithm works as follows. Consider some tree T rooted at w, a painting χ of T, and a node u with w an ancestor in the separator tree T'. The deep nodes on the path P_{uw} are stored in $L(u)$, and the ancestor-free subset of shallow nodes below w that cannot reach v (have no fs-avoiding path to v) is stored in $L(v)$. This is done for each ancestor w of u in the separator tree. The marker algorithm is given in Figure 5.3 and uses a recursive procedure, initially called with the entire tree T. It is clear that the marking is done in polynomial time. Note that each tree is rerooted at its separator before painting it. The reason for this is that if we used the painting of T rather than $T[w]$, then the shallow/deep tradeoff given by the painting lemma would not carry through to the size of the labels. This is because the path from u to v must be 'split' at the same node used as the root in the painting, i.e. the separator w.

The decoder algorithm. The decoder algorithm is given in Figure 5.4. Given $L(u), L(v)$,

DECODER$(L(u), L(v))$:
Assume that $L(u), L(v)$ are of the form

$$\begin{aligned} L(u) &= \mathcal{J}_1(u) \circ \ldots \circ \mathcal{J}_p(u) \\ L(v) &= \mathcal{J}_1(v) \circ \ldots \circ \mathcal{J}_q(v) \end{aligned}$$

1. If $q = 1$ then v is the separator of T, so return the value of $f(u, v) \in \mathcal{J}_1(u)$. If $p = 1$ then return the value of $f(u, v) \in \mathcal{J}_1(v)$. For $p, q > 1$ let

$$\begin{aligned} \mathcal{J}_1(u) &= (id(u), \mathsf{Deep}(u), \mathsf{Shallow}(u), f(u, w), f(w, u), i) \\ \mathcal{J}_1(v) &= (id(v), \mathsf{Deep}(v), \mathsf{Shallow}(v), f(v, w), f(w, v), j) \end{aligned}$$

2. If $i \neq j$ then u, v are in different subtrees and w is the least common ancestor of u, v in the separator tree. Do the following:

   (a) Check that u can reach w and that w can reach v via forbidden-set avoiding paths (by checking $f(u, w) \wedge f(w, v)$) It remains to decide whether any of the forbidden sets of nodes on the path P_{uw} appear on the path P_{wv}.

   (b) Using $L(u)$, check that none of the forbidden nodes in $\mathsf{Deep}(u)$ are ancestors of v in $T[w]$, by checking that $id(v) \notin [i, j]$ for all $[i, j] \in \mathsf{Deep}(u)$.

   (c) Using $L(v)$, check that none of the unreachable nodes in $\mathsf{Shallow}(v)$ are ancestors of u in $T[w]$, by checking that $id(u) \notin [i, j]$ for all $[i, j] \in \mathsf{Shallow}(v)$.

   Return that $f(u, v) = 1$ iff all the above are satisfied.

3. If $i = j$ then u, v are in the same subtree. In this case, discard the sublabels $\mathcal{J}_1(u)$ and $\mathcal{J}_1(v)$ and invoke the decoder recursively on the labels

$$\begin{aligned} L'(u) &= \mathcal{J}_2(u) \circ \ldots \circ \mathcal{J}_p(u) \\ L'(v) &= \mathcal{J}_2(v) \circ \ldots \circ \mathcal{J}_q(v), \end{aligned}$$

   returning the value of $\mathbf{D}(L'(u), L'(v))$.

Figure 5.4: The decoder algorithm

its first computes the least common ancestor w of u, v in the separator tree for T, and then checks (1) u can reach w (using $L(u)$), (2) w can reach v (using $L(v)$) and (3) $S(P_{uw}) \cap P_{vw} = \emptyset$, where $S(P_{uw})$ is the union of forbidden sets of the nodes on P_{uw}. The third check is conducted in two parts. In the first part, the decoder uses the label $L(u)$ to examine the forbidden sets of deep nodes on P_{uw}. In the second part, it uses $L(v)$ to examine the subtrees of $T[w]$ containing nodes not having a good path to v.

5.2.1 Proof of correctness

Lemma 5.2.2 *The labeling scheme $\langle \mathbf{M}, \mathbf{D} \rangle$ is correct, i.e. $D(L(u), L(v)) = f_T(u, v)$.*

Proof. For $L(u), L(v)$, let $w = \mathrm{LCA}(u, v)$ and consider $T[w]$. We know that u can reach v iff (1) u can reach w, (2) w can reach v and (3) none of the forbidden sets of nodes on the path P_{uw} appear on the path P_{wv}. Conditions (1) and (2) are handled by just looking at $L(u)$ and $L(v)$ independently.

To see that the decoder correctly decides (3), note that every node on the path P_{uw} is either painted deep or shallow. The forbidden sets of the deep nodes on this path are stored in $L(u)$, and only those that are in the same subtree of $T[w]$ are stored in the label. Hence if one of them is an ancestor of v, then it must be on the path P_{wv}. It remains to check that none of the shallow nodes on the path P_{uw} have forbidden sets on the path P_{wv}.

Imagine that there is some shallow node y on the path P_{uw} where an element of $S(y)$ is on the path P_{wv}, so that v is unreachable from y. Some ancestor of y in $T[w]$ must be in the set of unreachable shallow nodes stored in $L(v)$. Finally, if no shallow node on the path P_{uw} has a forbidden set that intersects the path P_{wv}, then no unreachable shallow node stored in $L(v)$ is an ancestor of u. □

5.2.2 Complexity

The efficiency of the labeling scheme relies on the observation that it is possible to paint the nodes of the tree so that there are not too many deep nodes on each path (and hence $L(u)$ does not need to store too many forbidden sets), and so that the subtrees containing shallow nodes that cannot reach u can be described with a small amount of space.

Our initial idea was to use the painting as described in the LCA labeling scheme of [Pel00]: a node is painted light if its subtree contains at most half the nodes of the subtree of its parent. This guarantees that each node has at most one heavy child and that every node has at most

$O(\log n)$ light ancestors. However, this is not what we need as we can construct instances where the forest induced by the heavy nodes would have $\Omega(n)$ leaves and therefore require lots of space. The following lemma gives the main result of this section.

Theorem 5.2.3 *The labeling computed by* $\mathbf{M}(T)$ *has labels of size* $\tilde{O}(\sqrt{kn})$ *bits and the decoder algorithm answers queries in time* $O(\log kn)$ *on this labeling.*

Proof. Assume that the painting χ used by the marker algorithm on each subtree is an α-painting, for some integer $1 \leq \alpha \leq n$. Lemma 5.2.1 implies that in a subtree with n nodes and forbidden sets of size at most k, the set $\mathsf{Shallow}(v)$ will contain at most n/α nodes and the set $\mathsf{Deep}(v)$ will contain at most $k\alpha$ nodes.

Since each ancestor label requires $2\log n$ bits to store the interval of identifiers of its descendants, the sublabels $\mathcal{J}(v)$ are each of size at most $(k\alpha + n/\alpha)(2\log n) + 3\log n + 2 \leq 3(k\alpha + n/\alpha)\log n$ bits (for large enough n such that $k\alpha + n/\alpha \geq 3$). Since the separator tree has depth at most $\log n$, the recursion has $\leq \log n$ levels, so each label is composed of at most this number of sublabels.

It follows that there is a scheme with labels of at most $3(k\alpha + n/\alpha)\log^2 n$ bits for any choice $1 \leq \alpha \leq n$. Minimizing the quantity $k\alpha + n/\alpha$ gives $\alpha = \sqrt{n/k}$, and using this choice of α for each subtree of size n and with forbidden sets of size at most k (note that the value of α is recomputed for each subtree) gives a labeling scheme using labels of size at most $3(k\sqrt{n/k} + n/\sqrt{n/k})\log^2 n = 6\sqrt{kn}\log^2 n$ bits.

The time complexity of the decoder is dominated by step (2), which is executed exactly once per query. Step (2a) takes time $O(1)$. For steps (2b) and (2c), consider the following related problem: there is a set S of intervals $\{[l_i, r_i]\}$ (where $l_i, r_i \in \{1, \ldots, n\}$) and we want to decide if some integer $x \in [1, n]$ is contained in any of the intervals. This can be done in worst-case time $O(\log n)$ using an *interval tree*, as described in [PS85]. The tree uses space $O(|S|)$, where $|S|$ is the number of intervals stored. Using this method to store the ancestor labels of the forbidden sets, step (2b) takes time $O(\log |\mathsf{Deep}(v)|) = O(\log k\alpha)$ and step (2c) takes time $O(\log |\mathsf{Shallow}(v)|) = O(\log n/\alpha)$. The decoder may iterate h times before executing step (2), where h is the height of the separator tree. Since the separator tree has depth $\log n$, the decoder takes total time $O(\log n + \log(k\alpha) + \log(n/\alpha)) = O(\log kn)$. □

Remarks. Note that the constant factors involved are small— for $k = 1$ the lower bound is $\sqrt{n}$ and the upper bound is $6\sqrt{n}\log^2 n$.

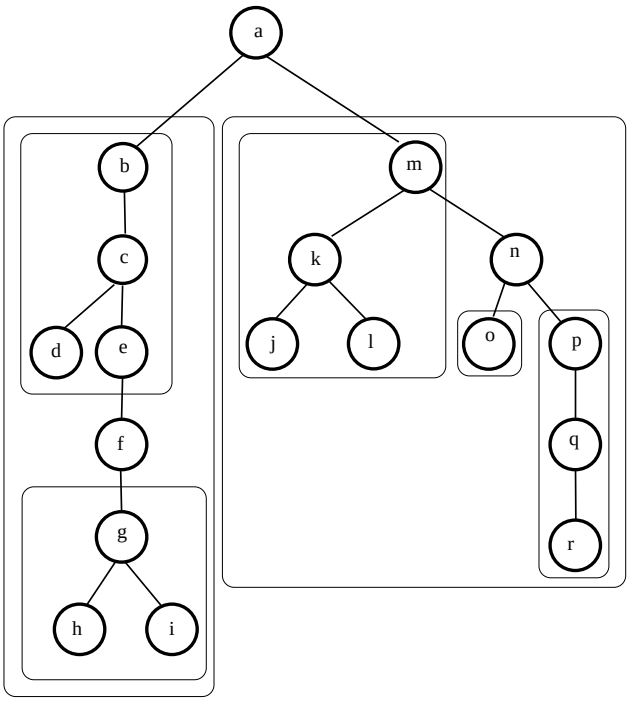


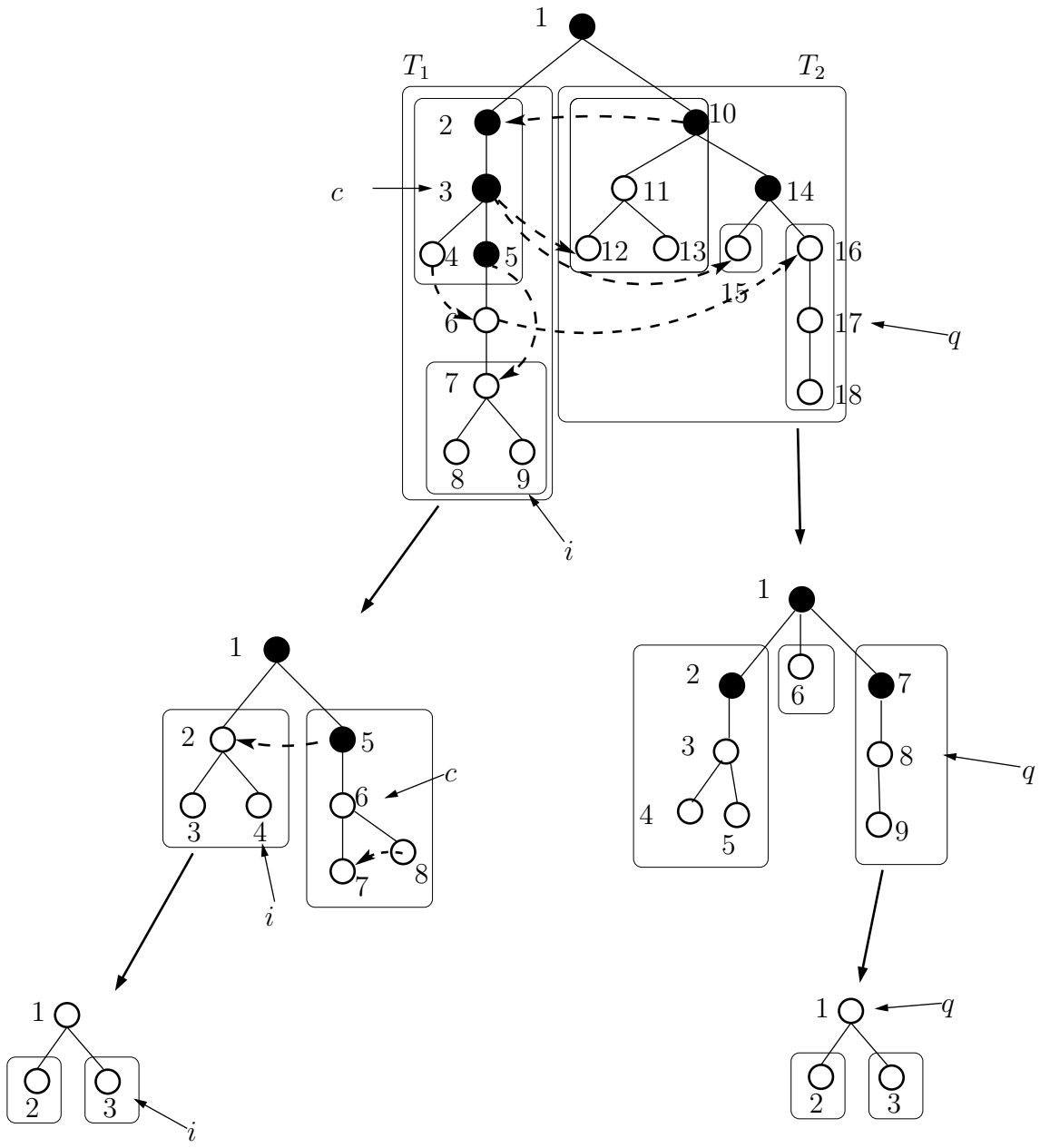
Figure 5.5: The tree used for the example and the first two levels of its recursive partitioning.

5.2.3 An example

Figure 5.5 shows a tree T that we will use to illustrate the labeling scheme. The same tree is drawn in Figure 5.2.3 with the shallow nodes filled and the deep nodes unfilled. Each node is drawn with its identifier from a depth-first search of the tree (note that the identifiers are recomputed for each subtree in the recursion). The forbidden sets are indicated by the dashed edges in the figure:

$$S(c) = \{j, o\}, S(d) = \{f\}, S(e) = \{g\}, S(f) = \{p\}, S(m) = \{b\}$$

We first root T at its separator w. To compute the sublabel for node i we first find the union of the forbidden sets of its deep ancestors. The forbidden set of the node f contains p, whose ancestor label in T is $[16, 18]$. Next we find the set of shallow nodes in $T \setminus T_1$ that i cannot be reached from (nodes in the tree T_1 containing i are not considered as this part of the label is only used for nodes in different subtrees). This is the set $\{a, m, n\}$, whose ancestor-free subset is a. The ancestor label for a is $[10, 18]$. Finally, we check if i can reach the root of T and if



$$
\begin{aligned}
L(c) &= (3,\{\},\{[10,18]\},1,1,1)\circ(6,\{\},\{\},1,1,2)\circ(1,\{\},\{\},1,1,0)\\
L(i) &= (9,\{[16,18]\},\{[10,18]\},1,0,1)\circ(4,\{\},\{[5,8]\},1,1,2)\circ\\
&\quad\circ(3,\{\},\{\},1,1,2)\circ(1,\{\},\{\},1,1,0)\\
L(q) &= (17,\{\},\{\},1,1,2)\circ(8,\{\},\{\},1,1,3)\circ(1,\{\},\{\},1,1,0)
\end{aligned}
$$

Figure 5.6: Illustrating the marker algorithm. Shallow nodes are filled and deep nodes are unfilled, and a dashed edge (i,j) means that $j \in S(i)$

the root can reach i (which it cannot, due to the forbidden set of e). This gives the sublabel $(9, \{[16, 18]\}, \{10, 18\}, 1, 0, 1)$.

The next sublabel for i is computed by repeating this process on the subtree T_1, as shown in the figure. Note that at each level the subtrees are repainted and the identifiers are recomputed, hence a node has an identifier for each subtree. The final labels are shown in the figure. As an example of the non-symmetry of the fs-reachability relation, it can be seen that $f(c, q) = 1$ but $f(q, c) = 0$.

5.3 A 2-approximate scheme

For general graphs, our aim is to efficiently route on good paths. Since the path is unique in a tree, this reduces to deciding if the path is good or not. However, it may be acceptable to use paths of low cost (recall that cost of a path is $c(P_{uv}) = \sum_{w \in P_{uv}} |S(w) \cap P_{wv}|$). We have been unable to construct an efficient labeling scheme to compute the exact cost of a path, but we can give a 2-approximate scheme with a logarithmic increase in label size. Let k be the maximum size of a forbidden set.

Theorem 5.3.1 *There exists a 2-approximate labeling scheme for trees for the cost $c(P_{uv})$ using labels of size $\tilde{O}(\sqrt{kn})$ bits and answering queries in time $O(\log kn)$.*

Proof. We augment the labeling scheme of Section 5.2 with a technique for approximately counting the number of forbidden elements intersected. First consider a line as in the top of Figure 5.7, where a dashed directed edge (u, v) means that $v \in S(u)$. The key observation is that for $i \leq j$, the cost $c(P_{ij})$ equals the number of crossing edges going from left to right that have both endpoints in $[i, j]$, and for fixed i this number is monotone increasing with j. Therefore the label for the root node in the figure stores the positions of the $\lceil \lg kn \rceil = O(\log kn)$ intervals to the right of it, which have cost $1, 2, 4, \ldots, kn$. It is easy to see that this indeed gives a 2-approximation to the actual cost.

This extends naturally to a 2-approximate scheme for trees – we apply the scheme for the line down every path of the tree. A region is now a subtree, identified by the root of the subtree, i.e. its ancestor label. The tree at the bottom of Figure 5.7 illustrates this. We do this for the subtree induced by the shallow nodes. There are at most n nodes and hence at most $\lceil \lg kn \rceil$ intervals on each path. By Lemma 5.2.1, the subtree induced by the shallow nodes has at most n/α leaves. Therefore, the marker algorithm can be modified so that each sublabel $\mathcal{J}(u)$ stores

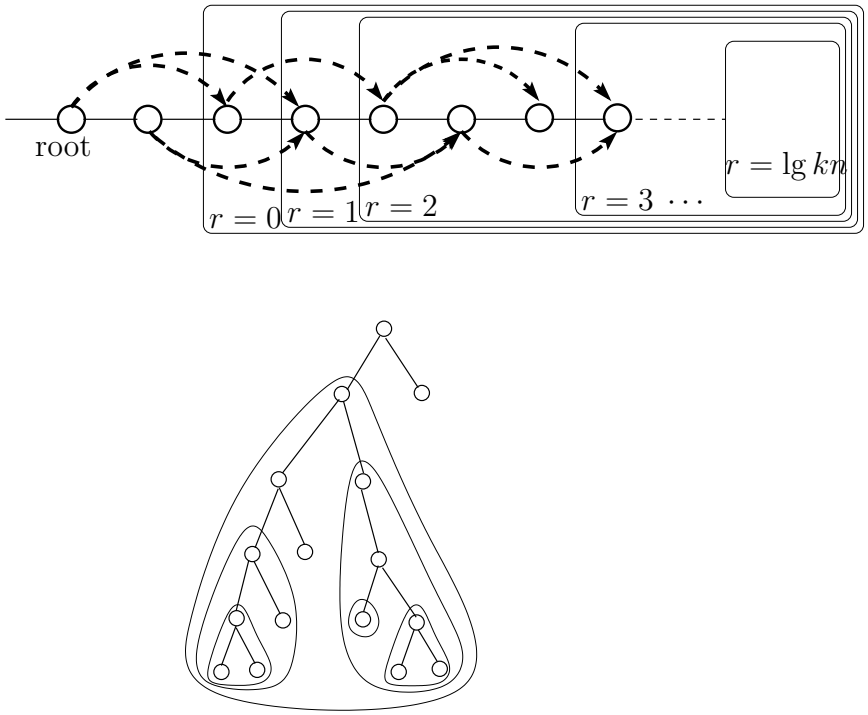


Figure 5.7: Illustrating the 2-approximation of $c(P_{ij})$. The cost of a path from the root to a node j is the number of crossing edges with both endpoints on the path. For a node j in region r, the algorithm returns 2^r.

at most $(n/\alpha)O(\log kn)$ intervals of shallow nodes, so this gives only a logarithmic increase in the label size.

The forbidden sets of the deep ancestors of u can be handled in a similar way. Instead of storing only the ancestor labels of forbidden sets of deep ancestors of u, we store ancestor labels denoting each of the $\log kn$ intervals. We construct (for each ancestor w of u in the separator tree) $O(\log kn)$ intervals (representing subtrees), where the identifier of v is contained in r intervals iff the path from w to v intersects between 2^r and 2^{r+1} elements of the forbidden sets of u's deep ancestors . In addition, $L(u)$ can store the *number* of forbidden nodes intersected on the paths to and from w in place of the boolean value $f(u, w)$. This increases the label size by a constant factor (since we already pay $O(\log n)$ to store the separator labels).

For the time complexity, the intervals in the ancestor labels can be stored in an interval tree [PS85] such that in worst-case time $O(k + \lg n)$ we can list all k intervals that contain a given integer. Since there are at most $O(\log kn)$ intervals that contain an identifier (by construction in both the shallow and deep cases), we can compute the number of intervals r containing an identifier in time $O(\log kn)$, hence a 2-approximation 2^r to the cost in each case. This is the same time complexity as in the original decoder. Since we have a 2-approximation to the cost of intersecting the forbidden sets of the deep ancestors of u and the cost of intersecting the shallow nodes of $T[w]$ (in addition to the exact cost of the paths P_{uw} and P_{wv}), adding them together gives a 2-approximation to the cost of P_{uv}. □

5.4 Bounded-distance forbidden sets

One of the difficulties that the algorithm must handle is that the forbidden set $S(u)$ may contain nodes that are far away from u, as large as the diameter of G. In practical scenarios, we expect that the forbidden set of u will contain nodes that are 'near' to u, perhaps within its own cluster. Another factor is the following: in graphs with good connectivity, it is likely that the minimum size of a uv-separator will increase with the distance from $d(u, v)$, and therefore the forbidden sets will only interfere with routing to nodes far away, if the forbidden set contains a large number of nodes.

Based on this observation, we consider a restricted policy where the set $S(u)$ may only contain nodes within some bounded distance δ of u. As before, assume that the forbidden sets are of size at most k. Define the δ-fs-reachability problem to be fs-reachability except that for every node u, every node of $S(u)$ lies within a distance δ of u, i.e. $d(u, S(u)) \leq \delta$. We shall

give a lower bound on label size for the problem of δ-fs-reachability.

Note that 1-fs-reachability is no harder than routing with next-hop preferences (where the cost $c_u(P_{uv})$ depends only on the next hop). In this case, we can make use of the fact that the cost $c(P_{uv})$ is equal to the cost of the same path in the directed graph G' where the edge (u, v) has weight $c_u(v)$ (we assume that $c_u(v)$ is finite). We can therefore make direct use of the distance labeling schemes from e.g. Gavoille et al. [GPPR04]. Note that only the weights are directed in G', i.e. there is an edge (i, j) iff there is an edge (j, i). Hence reachability is undirected and can be done with $O(\log n)$-bit labels, but the distances are not symmetric. Together with the distance labeling schemes of Gavoille et al. [GPPR04], this gives a next-hop cost labeling scheme using $O(\log^2 n)$-bit labels for trees. It can most likely be extended to other classes of graphs supporting an efficient distance labeling scheme (with directed edge weights).

5.4.1 Lower bound for trees

We start by extending the lower bound of Theorem 5.1.1 to δ-fs-reachability. The idea is simple – for small δ we flatten the tree, creating a large number of short paths.

Lemma 5.4.1 *Any δ-fs-reachability labeling scheme with $\delta \leq 2\sqrt{n}$ must assign some node a label of size $\Omega(\delta \log n/\delta^2)$ bits on n-node trees.*

Proof. The argument is similar to the lower bound of Theorem 5.1.1. Consider the tree having $2n/\delta$ node-disjoint paths each of length $\delta/2$. Each path to the root can independently choose a set of leaves of size at least $\binom{2n/\delta}{\delta/2}$ and by a similar argument to Theorem 5.1.1 the label size is bounded below by

$$\begin{aligned} \frac{1}{2n/\delta} \log \binom{2n/\delta}{\delta/2}^{2n/\delta} &\approx (\delta/2)\log(2n/\delta - \delta/2) - (\delta/2)\log(\delta/2) \\ &= (\delta/2)\log(4n/\delta^2 - 1) \end{aligned}$$

and this holds for $\delta/2 \leq 2n/\delta$, i.e. $\delta \leq 2\sqrt{n}$. □

5.4.2 Lower bound for general graphs

When the forbidden set $S(u)$ contains only neighbours of u, we can prove an $\Omega(n)$ lower bound on the label size for general graphs.

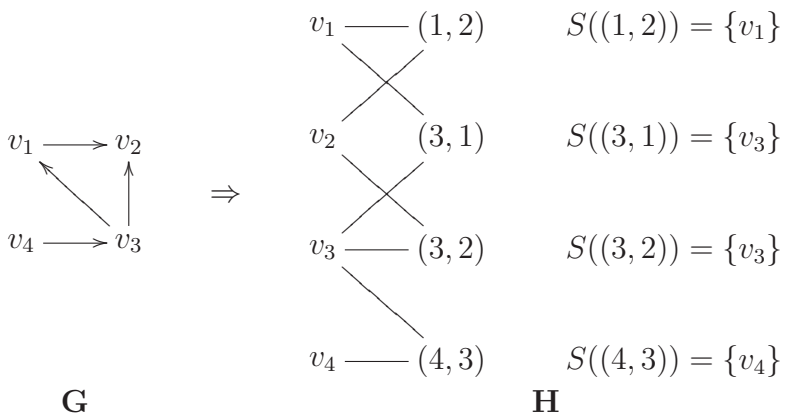


Figure 5.8: The reduction in Lemma 5.4.2

Lemma 5.4.2 *Any* 1*-fs-reachability labeling scheme for* $k = 1$ *must use labels of size* $\Omega(\sqrt{n})$ *bits.*

Proof. The proof is by reduction from reachability on directed graphs. Given a directed graph G on n nodes $v_1 \ldots v_n$, construct the undirected bipartite graph H on node sets V_1 and V_2 as follows. In V_1 there are n nodes $v_1 \ldots v_n$ and the set V_2 contains a node (i, j) for each edge (v_i, v_j) of G. Now for each directed edge (v_i, v_j) of G, add the undirected edges $(v_i, (i, j))$ and $(v_j, (i, j))$ to H. Finally assign the nodes in V_2 the forbidden sets $S((i, j)) = \{v_i\}$ (the nodes in V_1 all have empty forbidden sets). Figure 5.4.2 illustrates this construction.

We claim that v_i can reach v_j in G iff there exists a good path from v_i to v_j in H. The "$\Rightarrow$" direction is clear – if there is a path $P = v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ with no repeated nodes in G, then the path $P' = v_{i_1}, (i_1, i_2), v_{i_2}, \ldots, (i_{k-1}, i_k), v_{i_k}$ is a good path in H. For the other direction, assume that P' is a good path in H from v_{i_1} to v_{i_k}. We claim that the path P corresponds to a path in G. Each node in V_2 has degree exactly two, and the forbidden sets ensure that in any good path of H containing $\ldots, v_i, (i, j), v_j, \ldots$, the edge (v_i, v_j) exists in G.

Given a 1-fs-reachability scheme for H using r bits per label we can construct a directed reachability labeling scheme for G using r bits per label by setting $l_G(v_i) = l_H(v_i)$. Since there exist n-node directed graphs that require reachability labels of size $\Omega(n)$ bits [CHKZ02], the construction gives a family of $O(n^2)$-node graphs requiring $\Omega(n)$ bits per label. □

For the case where the size of the forbidden sets is unbounded, we can show an $\Omega(n)$ lower bound, which is clearly optimal in the worst-case.

Lemma 5.4.3 *Any* 1*-fs-reachability labeling scheme must use labels of size* $\Omega(n)$ *bits when the forbidden sets are unbounded in size.*

Proof. We can prove the lemma by reduction from adjacency in directed bipartite graphs. Given a bipartite graph $((V_1, V_2), E)$ on node sets V_1, V_2 with edges directed from V_1 to V_2, we construct the same graph $((V_1, V_2), E')$ but with undirected edges, and for $v \in V_1, S(v) = V_2 \setminus N(v)$ ($N(v)$ is the set of neighbours of v). It is clear that there is a good path from $v_1 \in V_1$ to $v_2 \in V_2$ iff $(v_1, v_2) \in E$. It follows (by a similar argument to [CHKZ02]) that there are $2^{\Omega(n^2)}$ distinct labelings of $O(n)$ nodes, and so at least one node must be assigned a label of size $\Omega(n)$ bits. □

5.5 Compact routing on good paths

The lower bound of Theorem 5.1.1 implies that any scheme that decides if there exists a good path between u, v by consulting only $L(u), L(v)$ must use labels of size $\Omega(\sqrt{n})$ bits. This is much too large to place in the headers of packets. In this section we show that it is possible to reduce this space by using a compact routing scheme. We describe a scheme that uses $O(k \log n)$ bits of storage at each node and uses labels of size $O(\log n)$ bits in each packet header. However, this comes at a price – a packet may travel over $\Omega(n)$ edges before the algorithm realises that there does not exist a good path.

5.5.1 Overview of compact routing on trees

Our scheme makes use of any compact routing scheme for trees, so we begin by giving an overview of compact routing on trees. The standard 'interval routing' technique due to Santoro and Khatib [SK85] is as follows. We root the tree arbitrarily and do a depth-first traversal, labeling each node with its identifier in the depth-first traversal. This is known as the ancestor label for the node. For each node w, let f_w be the descendant of w with the largest identifier. By the property of the ancestor labels, a node v is a descendant of w iff $v \in [w, f_w]$. A packet destined for v that arrives to w is routed as follows: if $w = v$ then the packet has reached its destination. If $v \notin [w, f_w]$, the packet is sent to the parent of w, using the parent pointer of w. Otherwise, a search among the children $w_1, \ldots, w_d$ of w is performed and the packet is forwarded to the last child w_i whose identifier is smaller than or equal to v's.

The packet headers are only of size $O(\log n)$ bits, but the routing table at a node w is of size $O(\deg(w) \log n)$ bits, making this inefficient for large degree nodes. Furthermore, routing decisions at these large degree nodes take time $O(\deg(w))$. This scheme has been improved; the most space-efficient scheme for trees is due to Thorup and Zwick [TZ01b] (and independently in [FG01]). Their scheme uses routing tables and labels of size $(1 + o(1)) \lg n$ bits, and each routing decision takes constant time.

Interestingly, it is known that an address space larger than $\lg n$ bits is needed for efficient routing on trees – if the address space is $\{1, \ldots, n\}$ and the packet header only contains the destination address then [EGP03] implies that no loop-free routing strategy can guarantee a local memory space better than $O(\sqrt{n})$ bits on every family of graphs, including trees.

5.5.2 A scheme for routing on good paths

Our scheme is simple and makes use of any compact routing scheme for trees such as that of Thorup and Zwick (TZ) [TZ01b] (or the scheme in [FG01]). The routing table for a node u stores the separator labels for u and the nodes in $S(u)$ in addition to the requirements of the TZ routing scheme. To send a packet from u to v, the label $L(v)$ that is placed in the packet header by the TZ routing scheme is augmented with the separator label for v. Therefore the packets have headers of size $O(\log n) + (1 + o(1)) \lg n = O(\log n)$ bits, and the routing tables are of size $O(k \log n)$ bits.

Now consider a packet that originated at u and arrives at w, destined for v. Using the separator labels, w checks whether the set $S(w)$ is a wv-separator in T. If so, the packet is returned to u (using the routing scheme in place). Otherwise it is forwarded to the next node using the TZ routing scheme. Since each TZ routing decision takes constant time and deciding if a set of size k a uv-separator takes time $O(k)$, each routing decision takes time $O(k)$.

The reason that this scheme breaks the $\Omega(\sqrt{n})$ space lower bound is that reachability is not decided locally at u; in the worst-case, the packet may traverse $\Omega(n)$ links before being returned, and so the worst-case time complexity of this scheme can be $\Omega(n)$. In many cases, such a delay is not acceptable and moreover, u cannot know that it cannot reach v until it tries sending the packet. On the other hand, if u is to be able to decide reachability with only local information, then it must use space $\Omega(\sqrt{n})$ and receive a label of size $\Omega(\sqrt{n})$ from v.

Instead of returning the packet if there is no path of zero cost, we can consider the following operation: route a packet from u to v iff there is a path P_{uv} with $c(P_{uv}) \leq r$. Given the separator

Algorithm	Space	Time	Space × Time
Table method	$O(n^2)$	$O(1)$	$O(n^2)$
Theorem 5.2.3	$\tilde{O}(\sqrt{k}n^{3/2})$	$O(\log kn)$	$\tilde{O}(\sqrt{k}n^{3/2})$
Theorem 5.6.1 ($1 \leq \alpha \leq n$)	$\tilde{O}(n^2/\alpha + kn)$	$O(k\alpha + \log \frac{n}{\alpha})$	$\tilde{O}(\frac{n^2}{\alpha} \log \frac{n}{\alpha} + kn^2 + \alpha k^2 n)$

Figure 5.9: Summarising the space-time tradeoffs for deciding whether there exists a good path in forbidden-set routing.

labels for u, v, S, we can determine the size of the intersection $|P_{uv} \cap S|$ by counting the number of nodes of S that are a uv-separator. By using an extra field of $\lg r \leq \lg n$ bits into the packet header, the nodes can keep track of the cost of the path so far and return the packet if the cost exceeds r at any point. The routing table and header sizes are still $O(k \log n)$ and $O(\log n)$ bits.

5.6 Nondistributed data structures

Any labeling scheme using s bits per label on some family $\mathcal{F}$ of graphs can be converted into a non-distributed data structure on $\mathcal{F}$ using $O(ns)$ bits of space and supporting queries with the same time complexity as the decoder. Therefore, Theorem 5.2.3 implies a non-distributed data structure for fs-reachability using $\tilde{O}(\sqrt{k}n^{3/2})$ bits space and having query time complexity $O(\log kn)$.

There are of course many non-distributed data structures for fs-reachability. One could build a table that lists for each pair of nodes u, v whether there is a good path from u to v, using $O(n^2)$ space and having $O(1)$ time complexity (and of course this would work for general graphs).

For trees, we can achieve a tradeoff between query time and space. For $1 \leq \alpha \leq n$ the scheme of Section 5.2.2 has labels of size $\hat{s} = \tilde{O}(n/\alpha + k\alpha)$ and time $t = O(\log kn)$, so the label size is minimized by choosing $\alpha = \sqrt{n/k}$. This gives a non-distributed data structure that may use space $n\hat{s} = \tilde{O}(\sqrt{k}n^{3/2})$ in the worst case. We now show how to construct data structures using space between $\tilde{O}(kn)$ and $\tilde{O}(n^2)$ but at the expense of increased query time.

Theorem 5.6.1 *For every $1 \leq \alpha \leq n$, there is a non-distributed data structure for forbidden-set reachability on n-node trees using space $\tilde{O}(n^2/\alpha + kn)$ and answering queries in time $O(k\alpha + \log(n/\alpha))$.*

Proof. We will show how with a small modification we can reduce the space required to $\tilde{O}(kn)$ but at the expense of an increased query time. Instead of storing the set of deep ancestors (and their forbidden sets) of each node using $\tilde{O}(kn\alpha)$ space as in the distributed labeling scheme of Section 5.2, in a centralized data structure all this can be stored once using $\tilde{O}(kn)$ space. This gives a data structure having space $s = \tilde{O}(n^2/\alpha + kn)$ instead of $\tilde{O}(n^2/\alpha + kn\alpha)$. Note that the strategy for shallow nodes is unchanged.

However, the search tree method used in Theorem 5.2.3 to store the ancestor label intervals cannot be used, since it constructs a different binary search tree for every node. The best alternative we can find is the following: by the painting lemma (Lemma 5.2.1), the deep nodes induce a forest of height[3] at most α below the shallow subtree. Hence for any node u, its deep ancestors can be found in time α and the ancestor intervals for their forbidden sets can be checked to see if they contain the destination v in time $O(|\mathsf{Deep}(u)|) = O(k\alpha)$. The shallow nodes are handled as before (using a balanced binary search tree) in time $O(\log(n/\alpha))$. This gives total time $O(k\alpha + \log(n/\alpha))$. □

Table 5.9 gives a summary of the space-time tradeoffs obtained in this section. For $\alpha = n/k$, Theorem 5.6.1 gives a data structure with space $\tilde{O}(kn)$ yet having query time $O(n + \log k) = O(n)$. These results show that, even on a simple family of graphs (trees), the problem still allows for some non-trivial algorithms.

5.7 Dynamic labeling schemes

So far we have considered only static labeling schemes, where the network and the forbidden sets are fixed in advance. These schemes rely on a centralized marker algorithm that is given an entire description of the network and uses this to output the entire set of labels. Therefore while the labels allow the problem to be solved using local information, the process of generating the labels has been centralized. In a dynamic network where nodes may join or leave and policies are updated, it is desirable to update the distributed representation offered by the labels in an efficient and distributed fashion. A centralized marker algorithm clearly limits the applicability of such labeling schemes in real dynamic networks.

Korman et al. [KPR02] describe a general method for converting a static labeling scheme on n-node trees to a fully dynamic one with only a $\log n$ factor increase in the label size. Since

[3]The height of a forest is the maximum height of any tree in the forest

the new marker algorithm is now a distributed algorithm, its communication complexity (to recompute the labels after a change) is an important property. Korman et al. show that if the static scheme has a *distributed* marker algorithm that computes the labels in the static setting and sends $\mathcal{MC}$ messages (of size $O(\log n)$) then it can be converted into a distributed marker algorithm for updating the labels in the dynamic setting with amortized message complexity $O((\log n)\mathcal{MC})$.[4]

We shall show that (assuming we use the notion of shallow sets) we can do no better asymptotically than to recompute from scratch when there is a change. To transform our static scheme into a dynamic scheme we need to convert the sequential marker algorithm into an efficient distributed one. Let us consider a distributed marker algorithm having three distinct phases:

1. **Painting.** As in the proof of Lemma 5.2.1, nodes of depth at most α are painted deep, and the rest are painted shallow. This can be done efficiently by a distributed depth-first search of the tree T.

2. **Deep nodes.** For each node v, the algorithm computes the part of the label that contains the forbidden sets of the deep ancestors of v. This can be done by propagating the forbidden sets of the deep nodes down to each of their deep descendants. Since all descendants of a deep node are also deep, each path is no longer than α and there may be $O(n)$ deep leaves. Therefore, the total number of forbidden elements sent over edges is $O(nk\alpha) = O(\sqrt{k}n^{3/2})$ (by setting $\alpha = \sqrt{n/k}$).

3. **Shallow nodes.** The final step is to inform each node v about its set $\mathsf{Shallow}(v)$, i.e. the shallow nodes that cannot reach v. However, this step appears to be costly – the following lemma shows that *any* distributed algorithm that computes these sets must have high communication complexity.

Lemma 5.7.1 *Any distributed algorithm that terminates with every node v knowing $\mathsf{Shallow}(v)$ must communicate $\Omega(n \log n)$ bits over $\Omega(n)$ edges in an n-node tree, even for $k = 1$.*

Proof. Assume that the parameter $1 \leq \alpha \leq n$ is given. Now construct the following tree, as in Figure 5.7 – there is a root r, a path of $(n + \alpha)$ nodes $h_1, \ldots, h_{n+\alpha}$ hanging from r and n nodes $v_1, \ldots, v_n$, each being a child of r. Consider some permutation σ of $\{1, \ldots, n\}$ where $\sigma(i)$ is

[4] In fact, the value of n depends on the size of the network at a particular time, but we assume for simplicity that it never grows by more than a polynomial factor.

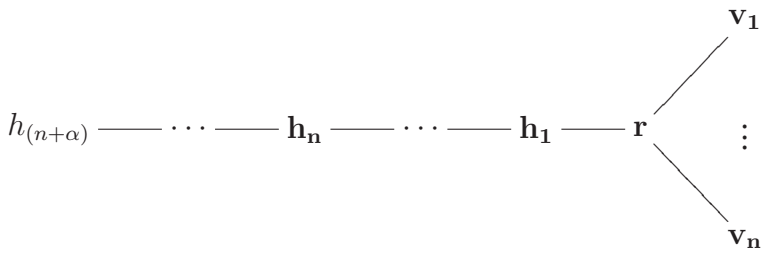


Figure 5.10: The tree used in the proof of Lemma 5.7.1. The forbidden sets of nodes $h_1 \dots h_n$ choose a permutation of $\{v_1 \dots v_n\}$. The nodes $h_1 \dots h_n$ are all painted shallow.

the ith element of the permutation. For $i = 1, \dots, n$, set $S(h_i) = \{v_{\sigma(i)}\}$ as h_i's forbidden set. The set $\{S(h_1) \dots S(h_n)\}$ is a permutation over the nodes $\{v_1, \dots, v_n\}$.

Now we show how to reduce from the two-party communication problem of deciding set-disjointness. Partition the nodes between the two players Alice and Bob as follows: give Alice the long path (and implicitly σ) and Bob the nodes $v_1, \dots, v_n$ and r. By construction, all the nodes $h_1, \dots, h_n$ will be painted shallow and therefore $\mathsf{Shallow}(v_j) = \{h_i\}$ iff $\sigma(i) = j$. Therefore, given the sets $\{\mathsf{Shallow}(v_i)\}$, Bob can know the permutation σ.

Define the unique *rank encoding* of a permutation by replacing each symbol in the permutation by its rank among the remaining symbols. For example, the rank encoding of 341562 is 331221. The rank encoding of a permutation can be expressed as a binary string by replacing each symbol by its unique binary expansion. There are $(n - i)$ possible values for the ith symbol in the rank encoding of a permutation of $\{1 \dots n\}$ and so every binary string of $\sum_{i=0}^{n-1} \log i = \lg(n!)$ bits corresponds to a unique rank encoding and hence a unique permutation of $\{1, \dots, n\}$.

Now for the reduction – given sets $P, Q \subseteq \{1, \dots, n\}$, Alice receives P and Bob receives Q. Alice computes the unique permutation σ corresponding to P and uses this to construct the forbidden sets in her side of the tree. Then they run the protocol to compute the shallow sets. From this, Bob can determine σ and thus also P. He can then locally decide if P, Q are disjoint.

Since the randomized communication complexity of disjointness on sets of size r is $\Omega(r)$ bits, at least $\Omega(\lg(n!)) = \Omega(n \log n)$ bits must cross the cut between Alice and Bob, which consists of a single edge. We can replace this edge by a path of n edges. It is known [Die97] that asymptotically, these nodes can no better than to act as relays, and so each must have $\Omega(n \log n)$ bits communicated across it. □

The lemma implies that for $k = O(1)$ the following algorithm is asymptotically optimal for computing the shallow sets: broadcast the entire tree (including the forbidden sets) to all nodes using $\tilde{O}(kn^2)$ bits then let each node locally compute its label using the centralized marker algorithm.

It is worth noting that this lower bound does not exclude the possibility of a labeling scheme with both $O(\sqrt{n})$-bit labels and low communication complexity, but such a scheme would have to avoid using the shallow sets as defined here.

5.8 Discussion

The most interesting open issue here is to investigate fs-reachability on other families of graphs such as those of small treewidth (although we have been unable to show good bounds for these families). Knowing the complexity of fs-reachability on more general graphs would be interesting as reachability is a fundamental problem for any routing scheme.

We have been unable to prove a stronger lower bound for general graphs than in Section 5.1. In fact, we conjecture that it does not get any harder than for trees:

Conjecture 5.8.1 *For any graph family $\mathcal{F}$, there is a labeling scheme for fs-reachability on $\mathcal{F}$ (with $k = O(1)$) having $\tilde{\Theta}(\sqrt{n})$-bit labels.*

Roughly, the intuition behind the conjecture is that to obtain a good lower bound, one should find a large set A of nodes where, for each distinct pair of nodes $u, v \in A$, there is a path P_{uv} that contains a 'large' number c of nodes disjoint from any other path P_{wx} where at least one of w, x is not u, v. The tree construction in Section 5.1 has this property with $c = \Omega(\sqrt{n})$ and attempting to increase this forces the paths to be non-disjoint. However, it is not clear how to efficiently encode the reachability information for a large number of paths between any pair of nodes.

Classical reachability on undirected graphs has a scheme with $\lg n$-bit labels: label each node with the identity of its connected component. On the other hand, it is known that directed reachability requires labels of total size $\Omega(m \log n^2/m)$ bits on some n-node m-edge graph [CHKZ02]. Forbidden-set reachability is at least as hard as undirected reachability: just set all the forbidden sets to be the empty set. Below is a simple reduction showing that it is no harder than directed reachability. Hence for forbidden sets of size $O(1)$, the label size for fs-reachability lies somewhere between undirected and directed reachability.

Lemma 5.8.2 *Forbidden-set reachability on undirected graphs is no harder than directed-graph reachability.*

Proof. Given an undirected graph G on n nodes $v_1 \ldots v_n$ and a forbidden set $S(v)$ for each node v, construct the bipartite graph H on $2n$ nodes $x_1 \ldots x_n$ and $y_1 \ldots y_n$ where there is an edge (x_i, y_j) iff there is a good path from v_i to v_j in G. It is easy to see that in H, x_i can reach y_j iff they are neighbours, and so the adjacency relation in H represents the fs-reachability relation in G. □

CHAPTER 6

Approximating Forbidden-set Routing

In Chapter 4 we showed that there is an $\Omega(k \log n/k)$ lower bound on the space requirements (per node) for any forbidden-set routing scheme, where k is an upper bound on the size of a forbidden set (Theorem 4.5.3). For small k, this means that good routing schemes may be possible. However, we believe that for general graphs there is a lower bound of $\Omega(n)$ bits. To avoid this bound, it is natural to think about ways of relaxing the problem, for example are we happy with an approximate solution? The difficulty with such an idea is that the problem of deciding if there exists a path of zero cost between two nodes is a decision problem not an optimization problem, so there is no natural notion of approximation.

In this chapter we consider one such approach to *approximating* the forbidden-set routing problem. We partition the network into connected clusters and instead of choosing arbitrary subsets of nodes, the forbidden sets must choose a subset of these clusters. This has the effect of grouping nodes together and treating them as the same node for the purposes of forbidden-set routing. We define the problem of obtaining a cluster graph that has good graph-theoretic properties, and motivate the problem of obtaining a cluster graph with bounded treewidth (deciding if there is a cluster graph with treewidth at most k may be an NP-complete problem, even though deciding if a given graph has treewidth at most k can be done in linear time). We show that if we can construct a cluster graph having small treewidth, then we can apply our forbidden-set routing schemes from Chapter 4 to it.

We begin, however, by considering an approach inspired by the work of Feigenbaum et al. [FKMS05] – they considered a relaxed version of shortest-path routing where each link has a number of objective values associated with it, for example delay, packet loss, bandwidth and so on. All nodes agree on these values, in the same way that all nodes agree on the weights of edges for shortest-path routing. Each node has an individual cost function, which is a convex combination of the objective values assigned to edges (for example, one node may be interested in paths minimizing the sum of delays, while another may be interested in paths minimizing another metric). They showed that a small number of routing trees (instead of a single routing tree) is sufficient for all nodes to route on almost-optimal paths. Their scheme does not immediately imply a space-efficient routing scheme, since each node would store a small number of trees for each destination, giving super-linear $\omega(n)$ routing table sizes.. We shall show how to use their construction to build a space-efficient compact routing scheme with a small increase in the approximation factor. We can then observe that this multiple objective cost problem can be seen as a special case of clustering the network and assigning costs to clusters. Since we are interested in forbidden-set routing, it is natural to ask if we can cluster the graph so as to obtain efficient forbidden-set routing algorithms for it.

6.1 Compact routing with a small number of objective costs

In this section we consider a variation on the forbidden-set routing model introduced by Feigenbaum et al. [FKMS05]. They considered a restricted model where each node w is assigned d *objective* costs $\langle l_1(w), \ldots, l_d(w) \rangle$, which are assumed to be integers bounded by a polynomial, i.e. less than n^c for some constant c. Then the policy of a node u is a probability distribution over d local variables $0 \leq \lambda_i(u) \leq 1$ for $i = 1, \ldots, d$ such that $\sum_{i=1}^{d} \lambda_i(u) = 1$. The policy can be interpreted as defining the cost to u of routing through w as a convex combination of the objective costs assigned to w, i.e.

$$c_u(w) = \sum_{i=1}^{d} \lambda_i(u) l_i(w).$$

The motivation for this cost model is that the costs may represent objective measurements such as latency and packet loss, but nodes may assign different opinions to their relative importance. They showed that for the case of $d = 2$, a small number of routing trees suffices to route on

approximately-optimal paths. This is a promising result, since Feigenbaum et al. showed that finding a single minimal-cost tree in the case of 2 metrics is APX-hard.

Theorem 6.1.1 ([FKMS05]) *Assume that the costs $l_1(\cdot), \ldots, l_d(\cdot)$ are at most n^c. Fix some destination node v. Given any $\epsilon > 0$, there is a set of routing trees $T_1, \ldots, T_r$ with $r = O(\frac{1}{\epsilon}(\log n + \log \frac{1}{\epsilon}))$ such that the following holds – for each node u, there exists a tree T_{t_u} such that $c_u(T_{t_u}) \leq (1+\epsilon)c_u(P^*_{uv})$, where P^*_{uv} is the path from u to v minimizing $c_u(\cdot)$ (i.e. of minimum cost to u).*

Proof. Let $\alpha = (1+\epsilon)$. Each tree T_t in the collection is the shortest-path tree for a specific convex combination of the two metrics $l_1(\cdot), l_2(\cdot)$. We name the trees after the metrics they optimize:

$T_\infty : l_1(\cdot)$, with ties broken by minimum $l_2(\cdot)$.

$T_{-\infty} : l_2(\cdot)$, with ties broken by minimum $l_1(\cdot)$.

$T_t : l_t(\cdot) = \frac{\alpha^t}{1+\alpha^t} l_1(\cdot) + \frac{1}{1+\alpha^t} l_2(\cdot)$ for $t \in \{-k, -(k-1), \ldots, -1, 0, 1, \ldots, k\}$ where $k = \lceil \log_\alpha (2\epsilon^{-1} n^{c+1}) \rceil$.

Thus, there are a total of $r = 2k + 3 = O(\log n)$ trees. These trees can be constructed with r shortest-path computations on node-weighted graphs (using e.g. Dijkstra's algorithm) and hence can be done in polynomial time. The proof goes on to show that the collection of trees do indeed achieve the desired approximation factor. □

They also showed that for general $d > 2$, approximately $O(4^d \log^d n)$ routing trees are sufficient (depending on the parameters ϵ and the size n^c of the costs $l_i(\cdot)$). Using their result we can construct a routing scheme by applying the theorem separately to each node as a destination: construct a function $t(u,v) = argmin_i d_{T_i}(u,v)$ where $t(u,v) = k$ means that the tree T_k contains the lowest cost path from u to v out of all the trees constructed. Each node also stores the port number for the edge to its parent in each tree. Then u can route to v by sending a packet to its parent in $T_{t(u,v))}$, writing $t(u,v)$ in the packet header to indicate which tree the packet should be sent on. The problem with using this as a routing scheme is that each node stores a routing table of size $O(n \log r)$ bits to identify the tree used for each destination.

Before presenting the scheme, we need to deal with the fact that the above result is for node-weighted graphs, but the compact shortest-path routing schemes we will use require edge-weighted graphs. We therefore apply a simple transformation as follows. Given an undirected node-weighted graph $G = (V, E)$ on n nodes where v has weight $l(v)$, we compute the edge-weighted dual $G' = (V', E')$ on w with $2n$ nodes as follows (the construction is shown in

Figure 6.1). The node set $V' = V \cup \{v' | v \in V\}$ contains the nodes of V and a node v' for every node v of V. The edge set $E' = E \cup \{\{v, v'\} | v \in V\}$ contains the edges of E and an edge $\{v, v'\}$ for every node v of V. For an edge $\{u, v\}$ where $u, v \in V$, assign it the weight $l(u, v) = l(v, u) = l(u) + l(v)$. For an edge $\{v, v'\}$, assign it the weight $l(v)$. It is easy to see that $2d_G(u, v) = d_{G'}(u', v')$ for $u, v \in V(G)$, and that if $P_{u'v'}$ is a path from u' to v' in G' then the subpath P_{uv} obtained by removing the first and last edges of $P_{u'v'}$ is a path in G. The following simple lemma shows that the edge-weighted graph has the same lowest cost paths as in the original graph, so we can apply any routing scheme to G by using the corresponding node $v \in G'$.

Lemma 6.1.2 *If P_{uv} is the lowest-cost uv-path (lcp) in the dual graph G' then P_{uv} is also the lcp in G.*

Proof. Assume for the sake of contradiction that P_{uv} is the lcp in G' and some other path P'_{uv} is the lcp in G. Let $c_G(P)$ denote the cost in G of the path P and $c_G(v)$ denote the weight of a node v (similarly, $c_{G'}(\{u, v\})$ denotes the cost of an edge $\{u, v\}$ in the dual G'). Therefore, $c_G(P_{uv}) > c_G(P'_{uv})$. Now consider the cost of the path P'_{uv} in G'. We will show that $c_{G'}(P_{uv}) > c_{G'}(P'_{uv})$, contradicting the assumption that P_{uv} is the lcp in G'. We know that for a path P_{uv} from u to v, $c_G(P_{uv}) = \frac{1}{2}\left(c_{G'}(P_{uv}) + c_G(u) + c_G(v)\right)$. It follows that

$$\begin{aligned} c_{G'}(P_{uv}) &= 2c_G(P_{uv}) - c_G(u) - c_G(v) \\ &> 2c_G(P'_{uv}) - c_G(u) - c_G(v) \\ &= c_{G'}(P'_{uv}), \end{aligned}$$

so P_{uv} cannot be the lcp in G'. □

We now show how to construct a routing scheme based on Theorem 6.1.1 by combining it with a compact approximate shortest-path routing scheme $\mathcal{R}$ such as Thorup-Zwick (TZ) [TZ01b]. We also assume that we have access to distance labels giving the lengths of the paths used by the routing scheme. Our modification is quite simple – let $G = (V, E)$ be an undirected unweighted graph, and denote by $G_t = (V, E)$ the node-weighted graph with node weights given by $l_t(\cdot)$. For each $t = 1, \ldots, r$ we compute the edge-weighted dual G'_t of G_t and then run the routing scheme $\mathcal{R}$ on G'_t. This gives for each node a sequence of labels $L(v) = L_1(v), \ldots, L_r(v)$ and routing tables $RT(v) = RT_1(v), \ldots, RT_r(v)$ where $L_t(v), RT_t(v)$ are the label and routing table given to v in G'_t by $\mathcal{R}$ (we assume that the labels $L_t(v)$ also contain

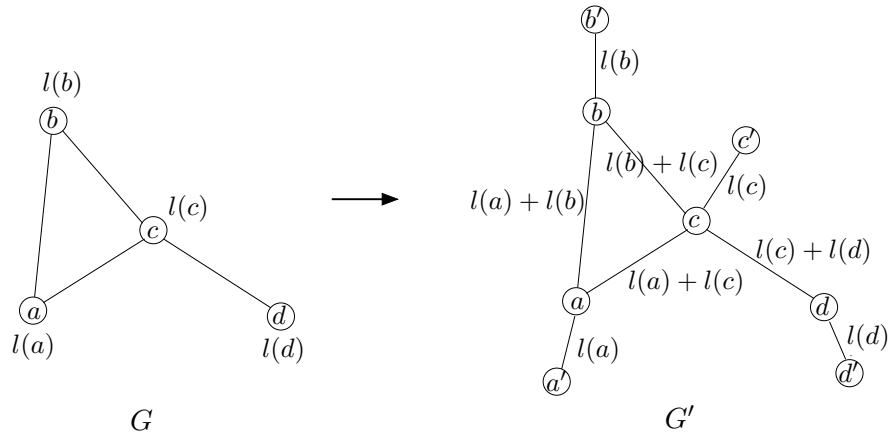


Figure 6.1: Constructing the edge-weighted dual G' from the node-weighted graph G.

information for the distance labeling scheme).

Routing from u to v in G is done as follows. Node u is given the labels $L(v), L(v')$ for the destination $v \in G$ and runs the distance decoder on $L_i(u')$ and $L_i(v')$ in G' to compute the approximate distances $\hat{d}_{G'_i}(u', v')$ for $i = 1, \ldots, r$. Let t_u be the value of i for which the reported distance $\hat{d}_{G'_i}(u', v')$ is smallest. Node u then uses the routing table $RT_{t_u}(u)$ and label $L_{t_u}(v) \in L(v)$ to compute the port number of the outgoing edge for the next hop w on the path in G. It uses the routing scheme $\mathcal{R}$ to construct the packet P to be sent to w and adds the identifier t_u and the label $L_{t_u}(v)$ to the header of P.

When a packet containing $L_{t_u}(v)$ and t_u is received by an intermediate node w, it uses the routing table $RT_{t_u}(v)$ and $L_{t_u}(v)$ to compute the port number for its next outgoing edge, computes the next packet (keeping $L_{t_u}(v)$ and t_u in the header) and forwards it (we assume that the routing scheme R is able to decide when the packet has reached its destination).

Lemma 6.1.3 *If the routing scheme $\mathcal{R}$ routes on paths of stretch s using routing tables of size $\mathcal{RT}$ and distance labels of size $\mathcal{L}$, then the routing scheme described above uses routing tables of size $r.\mathcal{RT}$, labels of size $r.\mathcal{L}$ and routes from u to v on a path P_{uv} where $c_u(P_{uv}) \leq s(1 + \epsilon)c_u(P^*_{uv})$, where P^*_{uv} is the path from u to v minimizing $c_u(\cdot)$.*

Proof. The main observation is the following: for any destination v, since each of the trees T_i is a shortest-path routing tree in the graph G_i, any stretch-s shortest-path routing scheme for G_i will produce a path P_{uv} from u to v of length within a factor s of $d_{G_i}(u, v)$. Since the path in T_i

is within a factor $(1+\epsilon)$ of optimal with respect to the costs c_i in G, it follows that the path P_{uv} in G_i is also within a factor $s(1+\epsilon)$ of optimal with respect to the costs c_i in G. □

The compact routing scheme TZ [TZ01b] has stretch 3, uses routing tables and distance labels of size $\tilde{O}(n^{1/2})$. This gives the following result.

Theorem 6.1.4 *Assume that $d = 2$. Given any $\epsilon > 0$, there is a compact routing scheme that routes packets from u to v on a path P_{uv} satisfying $c_u(P_{uv}) \leq 3(1+\epsilon)c_u(P^*_{uv})$. The scheme uses routing tables and labels of size $\tilde{O}(\frac{n^{1/2}}{\epsilon}(\log n + \log \frac{1}{\epsilon}))$.*

6.2 Approximate separator labels

We now show that the model described in the previous section leads to a natural notion of approximation for constructing separator labels. If we set all the costs $l_i(v)$ to be binary values we can consider the clusters $V_i = \{v \mid l_i(v) = 1\}$, and can require that they form a disjoint partition of G into connected components. If we also require that the nonzero λ_i all take the same value, then we have a simple structure where each node chooses a set of clusters.

Define the *quotient graph* corresponding to the partition $\Pi = \{V_1, \ldots, V_d\}$ as the undirected graph $G_{|\Pi} = (V', E')$ where $V' = \{V_1, \ldots, V_d\}$ (i.e. each cluster of the partition is contracted into a single node) and $E' = \{\{V_i, V_j\} \mid \exists x \in V_i, y \in V_j$ where $\{x, y\} \in E(G)\}$ (i.e. if there is an edge between two nodes in different clusters then there is an edge in the quotient graph between the nodes representing the clusters). The set $\{V_i \mid \lambda_i \neq 0\}$ then corresponds to choosing a set of nodes in the quotient graph to avoid, which corresponds to a set of nodes in G.

For forbidden-set routing (where the sets are arbitrary subsets of nodes), we would need to have n objective costs per node (dimensions), so Theorem 6.1.1 would construct $\Omega((\log n)^n)$ routing trees. However, it is clear that n trees suffice to allow choosing the exact lowest-cost path for every node, regardless of the policies. Therefore, it would be good if we could make use of the simple structure described above to improve these bounds. We now show that if we can construct quotient graphs with good properties then we can efficiently approximate forbidden-set routing. It is an approximation in the sense that forbidden sets are now arbitrary subsets of the set of clusters in the partition rather than arbitrary subsets of nodes, so nodes are grouped together in the forbidden sets by the partition chosen.

We will apply the compact forbidden-set routing scheme developed in Chapter 4 directly on

the quotient graph. Given any labeling L for the quotient graph $G_{|\Pi}$, we construct a labeling L' for G by assigning to all the nodes in a cluster V_i the label $L(V_i)$ for V_i in $G_{|\Pi}$. Given a set of labels for nodes in G, the decoder for L' simply runs the decoder for L on the labels. If the partition is chosen so that $G_{|\Pi}$ has good properties (e.g. bounded cliquewidth) then the labeling L' will be as if G has the same good properties but the price we pay is that the scheme treats all nodes of G in the same cluster as being the same node. Note that this is an approximation since if there is a fs-avoiding path in the quotient graph then there is also a fs-avoiding path in the original graph (if the query set now contains all the nodes contained in the forbidden clusters).

Routing is done as follows on G - for each edge $\{X, Y\}$ in the quotient graph, we store two nodes $x \in X, y \in Y$ where $\{x, y\} \in G$. Then we route on the quotient graph using the compact forbidden-set routing scheme, and route in connected components between the corresponding nodes in G using a separate shortest-path routing scheme (eg Thorup-Zwick). We now define the problem of constructing a quotient graph with small cliquewidth since we have separator labels with $O(\log n)$ bits on bounded cliquewidth graphs.

Problem CLIQUEWIDTH-k QUOTIENT GRAPH

Input:	A connected graph G.
Output:	A partition Π of $V(G)$ into connected components, such that $G_{\|\Pi}$ has cliquewidth at most k.
Objective:	Maximize the size of the quotient graph, i.e. $\|\Pi\|$

Remarks. Note that the parameter k is not part of the input. Also note that the problem of asking for a quotient graph with minimum cliquewidth is trivial, since taking the partition $\Pi = \{\{V(G)\}\}$ always gives a solution with cliquewidth 1 (since $G_{|\Pi}$ is the graph having one node). If k is part of the input then the problem is NP-complete, since Fellows et al. [FRRS06] have recently shown that given a graph G and an integer k, deciding whether the cliquewidth of G is at most k is NP-complete, by asking whether $|\Pi| = n$. However, the recognition problem (for fixed k, is the cliquewidth of G at most k?) is still open, and since our problem is closely-related to (and at least as hard as) the recognition problem, we leave it as an intriguing open problem.

We can also consider the similar problem TREEWIDTH-k QUOTIENT GRAPH. Since there is a linear-time algorithm for deciding if the treewidth of a graph is at most k [Bod93a], this may be a more tractable problem (if k is part of the input then it is NP-complete [ACP87]). For the case $k = 1$, the problem is asking for the largest integer s such that there is a partition

of the nodes of G whose quotient graph is a tree of s nodes. This can can be solved in linear time by contracting all cycles of G using the linear-time biconnected components algorithm of Tarjan [TV84]. One idea for solving this in practice may be to use similar heuristics to those for treewidth, such as the minimum-degree heuristic [Bod05], as shown below. This procedure

```
TREEWIDTH-k QUOTIENT GRAPH
1  G' ← G
2  repeat
3        if treewidth(G') ≤ k                    ▷ O(n) time [Bod93a]
4          then return G'
5          else choose a node v ∈ G of minimum degree
6               contract v with all its neighbours
```

Figure 6.2: Illustrating the minimum-degree heuristic for TREEWIDTH-k QUOTIENT GRAPH

is guaranteed to terminate since G is connected and we eventually reach the singleton graph, which has treewidth 0. We leave it as an open problem to construct efficient solutions for larger values of k.

CHAPTER 7

Discussion

In this thesis, we have studied the problem of routing in large distributed networks where nodes are free to define their own routing policies. In particular, we focused on the case where each node is free to specify a set of nodes that it wishes to avoid – this gives rise to the forbidden-set routing problem. Although we have succeeded in answering some basic questions about the complexity of forbidden-set routing – some in the negative (such as can we efficiently use routing trees), and some in the positive (for example, our forbidden-set routing algorithms for small treewidth and bounded cliquewidth graphs), we feel that our work represents a small step toward understanding how to deal with the additional complexity of policy-based routing. From a practical point of view, we believe that developing efficient and reliable algorithms for policy routing is important – as the number of nodes using policy routing increases, the intractability and space problems associated with routing-tree based schemes such as BGP will only become amplified. We propose the model of compact routing as the way forward for policy routing algorithms, but at the moment very little is known about the problem, even for the simple case of forbidden-set policies. From a theoretical point of view, the forbidden-set routing problem has fundamental open questions related to graph theory and other areas that are likely to be of independent interest (for example, how to construct an efficient distributed representation of all the separators in a graph). In this final chapter we try to suggest interesting directions and summarise some of the open questions arising from our work.

7.1 SPPs and routing trees

In Chapter 3 we presented some negative results about using routing trees for forbidden-set (and more generally, policy-based) routing. We used this as an argument against the use of routing trees. Although the NP-completeness results rule out even the simple case of forbidden-set preferences, it may be that there is a class of policies that are tractable yet more expressive than next-hop preferences. It would be very interesting to understand how the algebraic properties of a routing algebra relate to the complexity of solving the SPP instances that it generates – for example, what makes deciding solvability of SPPs with two-hop preferences NP-complete, while those with next-hop preferences are always solvable? The discussion in Section 3.6 contains more details about these open problems.

Another interesting problem (perhaps of independent interest) is to consider the k-SAT communication complexity conjecture – Lemma 3.4.7 proves it for $k = O(1)$ and only holds in the deterministic case. It would be interesting to prove or disprove the conjecture for larger values of k. We have only been able to prove a weaker randomized lower bound – whether we can do better with randomization is open.

7.2 Compact routing schemes

Since this is the first time that the problem of forbidden-set routing (and in general, compact algorithms for policy routing) have been studied, there are naturally many fundamental open problems remaining. Here we list what we consider to be some of the most important problems associated with the work in this chapter.

Open problem 1: Construct distance separator labels for other graph families.

One direction is to attempt to construct distance separator labels for cliquewidth graphs using labels of size polynomial, or even linear, in the cliquewidth $cw(G)$. Alternatively, it would be interesting to prove a lower bound for the label size involving the cliquewidth. One difficulty in extending the treewidth scheme to handle small cliquewidth graphs may be in constructing a binary term tree of small height that represents the graph. For treewidth k graphs, we relied on a result that enables us to convert any tree decomposition of width k into a balanced one with height $O(\log n)$ and width $O(k)$. For cliquewidth graphs, no such result is known; if we want a *balanced* term tree (having height $O(\log n)$), it is not known if we can avoid suffering an exponential increase in the cliquewidth. In the case of treewidth, we made use of the fact

that there is only a linear increase in treewidth. It is an open problem to reduce this cliquewidth blowup to even a polynomial factor increase [CV03].

Open problem 2: Tighten the gap between the upper and lower bounds for the size of distance separator labels for general graphs.

We showed an $\Omega(n)$ bits lower bound, but there is nothing better than the trivial $O(n^2)$ bound for general graphs (store at each node a copy of the entire graph). Can this be improved to $O(n^{3/2})$ or $O(n^{1+\epsilon})$ bits, or is it optimal? One way of attacking this may be to try to prove that treewidth k graphs have separator labels of size $\tilde{O}(k)$ bits. If this were true, then since n is an upper bound of the treewidth of any graph with n nodes (construct a tree decomposition having a single bag), a $\tilde{O}(n)$ upper bound for general graphs would follow. We believe that this problem has deep connections with many other areas of graph theory.

Open problem 3: Construct a more efficient routing scheme by removing the dependency on the degree of G.

We showed how to construct forbidden-set routing schemes by using distance separator labels and a simple routing scheme using these distances. However, the simple routing scheme involves storing the neighbours for each node, so we pay a factor $O(\Delta(G))$ in addition to the size of the distance separator labels and the size of the forbidden sets. We believe that it is possible to apply similar ideas to those of Cowen [Cow99] and Thorup and Zwick [TZ01b] by using carefully-selected 'landmarks' in the graph to reduce the space requirements. This would immediately improve many of our results in Chapter 4.

Open problem 4: Investigate randomization and approximation as a means of reducing the complexity.

The motivation for approximate shortest-path routing is primarily due to the $\Omega(n)$ lower bound in the case of exact shortest-path routing in general graphs. Since we also have a high $\Omega(n)$ lower bound for the problem of deciding if a given set of nodes is a separator of two nodes, we would like to investigate if we can circumvent this, but obviously at the cost of something else. One interesting idea, pioneered by Karger [Kar94] in his work on network cuts, is to use randomization. Is it possible to construct smaller separator labels that give a correct answer with high probability? If, in addition we are interested in distance separator labels then it might be possible to consider approximate distance separator labels – given labels for u, v, S, we would like to compute an approximation to the distance $d_{G\setminus S}(u, v)$. If we can achieve sublinear label sizes in this case then that would be a good result.

Bibliography

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[ACPS93] Stefan Arnborg, Bruno Courcelle, Andrzej Proskurowski, and Detlef Seese. An algebraic theory of graph reduction. *J. ACM*, 40(5):1134–1164, 1993.

[AGKR02] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–264, New York, NY, USA, 2002. ACM Press.

[Ami02] Eyal Amir. Approximating treewidth. Submitted for publication, 2002.

[Bak94] Brenda S. Baker. Approximation algorithms for np-complete problems on planar graphs. *J. ACM*, 41(1):153–180, 1994.

[Bod89] H. L. Bodlaender. NC-algorithms for graphs with small treewidth. In *Proc. 14th Workshop Graph-Theoretic Concepts in Computer Science WG'88*, pages 1–10. Springer-Verlag, Lecture Notes in Computer Science 344, 1989.

[Bod93a] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 226–234, New York, NY, USA, 1993. ACM Press.

[Bod93b] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22, 1993.

[Bod05] Hans L. Bodlaender. Discovering treewidth. In Peter Vojtás, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *SOFSEM*, volume 3381 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[Cha88] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

[CHKZ02] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[CK02] Peter Clote and Evangelos Kranakis. *Boolean Functions and Computation Models*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[Cou07] Bruno Courcelle. *Thorie des graphes*, chapter Dcompositions arborescentes. 2007. In preparation.

[Cow99] Lenore J. Cowen. Compact routing with minimum stretch. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 255–260, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[CR05] Derek G. Corneil and Udi Rotics. On the relationship between clique-width and treewidth. *SIAM J. Comput.*, 34(4):825–847, 2005.

[CT07] Bruno Courcelle and Andrew Twigg. Compact forbidden-set routing. In *STACS '07: Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (to appear)*, Lecture Notes in Computer Science. Springer, 2007.

[CV03] Bruno Courcelle and R. Vanicat. Query efficient implementation of graphs of bounded clique-width. *Discrete Applied Mathematics*, 131(1):129–150, 2003.

[Die97] Martin Dietzfelbinger. The linear-array problem in communication complexity resolved. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 373–382. ACM Press, 1997.

[Dir52] G. A. Dirac. Some theorems on abstract graphs. *Proceedings of the London Mathematical Society*, 2:69–81, 1952.

[EGIN97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification — A technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, September 1997.

[EGP03] Tamar Eilam, Cyril Gavoille, and David Peleg. Compact routing schemes with low stretch factor. *J. Algorithms*, 46(2):97–114, 2003.

[Epp95] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 632–640, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[Epp00] David Eppstein. Diameter and treewidth in minor-closed graph families. *Algorithmica*, 27(3):275–291, 2000.

[FG01] Pierre Fraigniaud and Cyril Gavoille. Routing in trees. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 757–772, London, UK, 2001. Springer-Verlag.

[FG04] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Logic*, 130(1-3):3–31, 2004.

[FHL05] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. Improved approximation algorithms for minimum-weight vertex separators. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 563–572, New York, NY, USA, 2005. ACM Press.

[FJB05] Nick Feamster, Ramesh Johari, and Hari Balakrishnan. Implications of autonomy for the expressiveness of policy routing. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 25–36, New York, NY, USA, 2005. ACM Press.

[FKMS05] Joan Feigenbaum, David Karger, Vahab Mirrokni, and Rahul Sami. Subjective-cost policy routing. In *Lecture Notes in Computer Science*, volume 3828, pages 174–183, 2005.

[FR94] M. Furer and B. Raghavachari. Approximating the minimum-degree steiner tree to within one of optimal. *Journal of Algorithms*, 17:409–423, 1994.

[Fre83] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 252–257, New York, NY, USA, 1983. ACM Press.

[FRRS06] Michael R. Fellows, Frances A. Rosamond, Udi Rotics, and Stefan Szeider. Clique-width minimization is np-hard. In *STOC 2006: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, New York, NY, USA, 2006. ACM Press.

[FSS04] Joan Feigenbaum, Rahul Sami, and Scott Shenker. Mechanism design for policy routing. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 11–20, New York, NY, USA, 2004. ACM Press.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GP03] Cyril Gavoille and David Peleg. Compact and localized distributed data structures. *Distrib. Comput.*, 16(2-3):111–120, 2003.

[GPPR04] Cyril Gavoille, David Peleg, Stephane Perennes, and Ran Raz. Distance labeling in graphs. *J. Algorithms*, 53(1):85–112, 2004.

[GR00] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. *SIGMETRICS Perform. Eval. Rev.*, 28(1):307–317, 2000.

[GSW02] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, 2002.

[HdLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[HK99] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

[HMT88] A Hajnal, W Maass, and G Turan. On the communication complexity of graph properties. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 186–191. ACM Press, 1988.

[HT97] Monika R. Henzinger and Mikkel Thorup. Sampling to provide or to bound: with applications to fully dynamic graph algorithms. In *Proceedings of the workshop on Randomized algorithms and computation*, pages 369–379, New York, NY, USA, 1997. John Wiley & Sons, Inc.

[iO05] Sang il Oum. Approximating rank-width and clique-width quickly. In Dieter Kratsch, editor, *WG*, volume 3787 of *Lecture Notes in Computer Science*, pages 49–58. Springer, 2005.

[JM96] Esther Jennings and Lenka Motyckova. Distributed algorithms for sparse k-connectivity certificates. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, page 180, New York, NY, USA, 1996. ACM Press.

[Kar94] David R. Karger. Random sampling in cut, flow, and network design problems. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 648–657, New York, NY, USA, 1994. ACM Press.

[kCGG06] Chi kin Chau, Richard Gibbens, and Timothy G.Griffin. Towards a unified theory of policy-based routing. In *Proc. IEEE INFOCOM*, April 2006.

[KFY04] D. Krioukov, K. Fall, and X. Yang. Compact routing on internet-like graphs. In *Proc. IEEE INFOCOM*, 2004.

[KKP05] Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 9–18, New York, NY, USA, 2005. ACM Press.

[KN97] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, UK, 1997.

[KNR92] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM J. Discret. Math.*, 5(4):596–603, 1992.

[KPR02] Amos Korman, David Peleg, and Yoav Rodeh. Labeling schemes for dynamic tree networks. In *STACS '02: Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science*, pages 76–87, London, UK, 2002. Springer-Verlag.

[Mor03] Christian Worm Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 618–627, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[MW77] John M. McQuillan and David C. Walden. The arpa network design decisions. *Computer Networks*, 1:243–289, 1977.

[NH98] Hiroshi Nagamochi and Toru Hasunuma. An efficient nc algorithm for a sparse k-edge-connectivity certificate. In *ISAAC '98: Proceedings of the 9th International Symposium on Algorithms and Computation*, pages 447–456, London, UK, 1998. Springer-Verlag.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992.

[Pel99] David Peleg. Proximity-preserving labeling schemes and their applications. In *WG '99: Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 30–41, London, UK, 1999. Springer-Verlag.

[Pel00] David Peleg. Informative labeling schemes for graphs. In *MFCS '00: Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*, pages 579–588, London, UK, 2000. Springer-Verlag.

[PS85] Franco P. Preparata and Michael I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[PU89] David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, 1989.

[RS86] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[SK85] Nicola Santoro and Ramez Khatib. Labelling and implicit routing in networks. *Comput. J.*, 28(1):5–8, 1985.

[Sob03] Joao Luis Sobrinho. Network routing with path vector protocols: theory and applications. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 49–60. ACM Press, 2003.

[Tho00] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 343–350, New York, NY, USA, 2000. ACM Press.

[Thu95] Ramakrishna Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 28–37, New York, NY, USA, 1995. ACM Press.

[TV84] Robert Endre Tarjan and Uzi Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time (extended summary). In *FOCS*, pages 12–20. IEEE, 1984.

[TZ01a] Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 183–192, New York, NY, USA, 2001. ACM Press.

[TZ01b] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 1–10, New York, NY, USA, 2001. ACM Press.

[VGE96] Kannan Varadhan, Ramesh Govindan, and Deborah Estrin. Persistent route oscillations in inter-domain routing. Technical report, USC/ISI, 1996.

[Wil86] Robin J Wilson. *Introduction to graph theory*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Win89] S. Win. On a connection between the existence of the k-trees and the toughness of a graph. *Graphs and Combinatorics*, 7:201–205, 1989.