Robust Network Computation

by

David Pritchard

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

Author .....
Department of Electrical Engineering and Computer Science
August 16, 2005

Certified by .....
Santosh S. Vempala
Associate Professor of Applied Mathematics
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Robust Network Computation

by

David Pritchard

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science

Abstract

In this thesis, we present various models of distributed computation and algorithms for these models. The underlying theme is to come up with fast algorithms that can tolerate faults in the underlying network. We begin with the classical message-passing model of computation, surveying many known results. We give a new, universally optimal, edge-biconnectivity algorithm for the classical model. We also give a near-optimal sub-linear algorithm for identifying bridges, when all nodes are activated simultaneously.

After discussing some ways in which the classical model is unrealistic, we survey known techniques for adapting the classical model to the real world. We describe a new *balancing* model of computation. The intent is that algorithms in this model should be automatically fault-tolerant. Existing algorithms that can be expressed in this model are discussed, including ones for clustering, maximum flow, and synchronization. We discuss the use of agents in our model, and give new agent-based algorithms for census and biconnectivity.

Inspired by the balancing model, we look at two problems in more depth. First, we give matching upper and lower bounds on the time complexity of the census algorithm, and we show how the census algorithm can be used to name nodes uniquely in a faulty network. Second, we consider using discrete harmonic functions as a computational tool. These functions are a natural exemplar of the balancing model. We prove new results concerning the stability and convergence of discrete harmonic functions, and describe a method which we call *Eulerization* for speeding up convergence.

Thesis Supervisor: Santosh S. Vempala
Title: Associate Professor of Applied Mathematics

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Computability, complexity, and algorithms are relatively new mathematical fields. The usefulness of computers in real life has helped stimulate a great deal of research in theoretical computer science in the past half-decade. Distributed algorithms, an even newer development, arise when we consider multiple computers that can communicate with one another. Distributed algorithms have applications in networking, supercomputing, and sensor networks. In this thesis, we aim to discuss distributed algorithms from both practical and theoretical standpoints. On the practical side, we include an introduction suitable for someone with no prior knowledge of distributed algorithms, we show simple examples using pseudocode, and we address the problems that distributed algorithms face when they are implemented in real networks. On the theoretical side, we survey current research for the best solutions to graph algorithms, we present several new results, and we describe a new model of distributed computation.

1.1 Summary

In Chapter 2, we give a general introduction to distributed algorithms, including some historical background. We introduce a clean model of distributed computation called the *message-passing model*. This is the most common model used by researchers to describe distributed computation. First, we discuss the fundamental problems of *leader election* and *spanning tree construction* in depth, including pseudocode solutions. We discuss why pre-existing unique node IDs are usually needed in deterministic algorithms. Then, we go on to survey the best known distributed solutions to many combinatorial graph problems. We pay special attention to algorithms whose running time is sublinear in n, the number of nodes.

In Chapter 3, we give a new distributed algorithm for computing the edge-biconnected components of a graph, which runs faster than any known algorithm for that problem. The algorithm constructs a short spanning tree of the network, and then proceeds to send messages up and down the tree. The algorithm depends on some nice properties of the tree's *pre-order* labeling. We show that the whole algorithm has complexity of $O(\mathsf{Diam})$ time and $O(m)$ messages on a synchronized network with a leader. Furthermore, we show that this performance is *universally optimal:* no correct algorithm can beat the performance of ours on any graph, except by a constant factor. We give a sublinear algorithm for the same problem, and also show that our technique can be used to find strongly-connected components in $O(n)$ time.

In Chapter 4, we discuss some unrealistic assumptions made by the clean model of Chapter 2, and discuss existing techniques by which some of these problems can be avoided. These techniques include message authentication, synchronizers, and clustering. We discuss the *self-stabilizing* model of computation, an alternative to the message-passing model. We then give a new model of computation, based on *balancing,* for use in faulty networks. Balancing algorithms for 2-coloring, routing, clustering, maximum flow, aggregation and synchronization are discussed. We also give *agent*-based algorithms for census and edge-biconnectivity that meet our model.

In Chapter 5 we discuss the agent-based census algorithm in more depth. We prove in Theorem 5.3.4 that the time complexity of this algorithm is $O(n \log n)$. Conversely, we show in Theorem 5.4.4 that this bound is tight on a family of graphs called *layered rings.* While the upper bound can be derived from earlier work [93] on approximate TSP solutions, the lower bound, to our knowledge, is new. As an application, we show how the census algorithm can be used to give unique names to the nodes of a faulty network with arbitrary topology, thus solving a problem for which no efficient algorithm seems known.

In Chapter 6 we discuss *discrete harmonic functions.* In our model, discrete harmonic functions arise when "balancing" is taken to mean "numerical averaging." We prove that, when the hitting time of a random walk on the network is small, the distributed computation of a discrete harmonic function converges quickly. The *stability factor* of a harmonic function describes how much the function's values may change when an edge is added to the underlying graph's topology; we express the stability factor in terms of a parameter called the commute time. We find that Eulerian graphs enjoy better performance than arbitrary weighted graphs, and give a distributed algorithm for making any directed graph Eulerian. We give applications of harmonic functions to planarity testing and fault-tolerant broadcast. In particular, we show that random perturbations of the edge weights allow us to use the fault-tolerant broadcast algorithm on any 2-vertex connected graph.

All of these chapters may be read independently, although a reader without any prior knowledge of distributed algorithms should begin with Chapter 2.

1.2 Conventions and Definitions

In the following chapters, we use a graph $\mathcal{G}$ to model a computation network. We denote the set of vertices by $V(\mathcal{G})$ and edges by $E(\mathcal{G})$. When the graph is clear from context, we simply write V and E. We use n to denote $|V|$, and m to denote $|E|$. We often write the vertices of $\mathcal{G}$ as $v_1, v_2, \ldots, v_n$. Sometimes we work with directed graphs; in such a case, the edge (u, v) goes *from* u *to* v. In an undirected graph, $(u, v) \in E$ if and only if $(v, u) \in E$.

Instead of directly working with edges, we often work with neighborhoods. The out-neighborhood of a vertex v, denoted $\Gamma(v)$, is defined by

$$\Gamma(v) := \{u \mid (v, u) \in E\}.$$

(Here, and in the rest of the thesis, the notation $A := B$ denotes an equation that defines the symbol A.) The in-neighborhood $\Gamma^{-1}(v)$ denotes the inverse relation. We use $\delta(v)$ to denote the out-degree of v, so $\delta(v) = |\Gamma(v)|$. The symbol Δ denotes the maximum degree of any node in the graph.

The distance between two vertices u and v in $\mathcal{G}$, denoted $d_{\mathcal{G}}(u, v)$, is the length of the shortest path from u to v. In the event that no such path exists, the distance is infinite. The *diameter* of a graph, denoted $\mathsf{Diam}(\mathcal{G})$, is the maximum distance between any two nodes in the graph,

$$\mathsf{Diam}(\mathcal{G}) := \max_{u,v \in V(\mathcal{G})} d_{\mathcal{G}}(u, v).$$

Let S be a subset of $V(\mathcal{G})$. Define the $\mathcal{G}$-diameter of S, denoted $\mathsf{Diam}_{\mathcal{G}}(S)$, to be the largest distance in $\mathcal{G}$ between any two elements of S,

$$\mathsf{Diam}_{\mathcal{G}}(S) := \max_{u,v \in S} d_{\mathcal{G}}(u, v).$$

The *subgraph* of $\mathcal{G}$ *induced* by S is the graph with vertex set S and edge set $\{(u, v) \in E(\mathcal{G}) \mid u, v \in S\}$.

This thesis contains several distributed algorithms, presented in pseudocode. The notation $x \leftarrow y$ indicates assignment of the value y to the variable x. Our pseudocode uses informal, but unambiguous, terms like "I, *me*, my" in order for a node to refer to itself.

Chapter 2

The Message-Passing Model

In this chapter, we give a short introduction to distributed algorithms, including some history and a model of distributed computation. This model is too simple in some ways; we discuss ways to make it more realistic and complicated in Chapter 4. In the latter half of the chapter, we survey the best known distributed solutions for many graph problems and constructions.

2.1 Motivation and History

A distributed computing environment consists of multiple computing agents that compute concurrently and that share information amongst each other. The basic principle is that the whole is greater than the sum of its parts: by sharing information, the group can accomplish a computation that no single agent could perform. We will describe four applications presently: interactivity, information sharing, parallel scientific computing, and sensor networks.

One advance in computer science which took place in the 1960s was the shift from *batch processing* systems to *interactive* systems. Whereas a batch processing system, like a punchcard machine, runs automatically until completion or an error occurs, an interactive system allows the user to provide new input (say, for debugging) during run-time. In an interactive system, the computing agents are different processes which handle user input, show output on a screen, et cetera. These agents communicate via shared sections of the computer's memory; coordination is needed to make sure that they do not overwrite each other's memory [32]. In a multi-user or time-sharing system, multiple users concurrently access the system, each with their own input and output device, and they share external resources such as printers.

In the 1960s, American universities and government produced a country-wide computer network

called ARPAnet in order to share electronic information among remote sites. This network is largely renowned as the predecessor of the Internet [31]. *Local area networks,* such as those used by businesses and households, also developed out of the need to share information. In these networks, algorithms are needed for fundamental tasks such as *routing* and *error detection*, in order that information can be shared efficiently and accurately.

Another motivating factor for distributed computing is that certain complex computational tasks involve inherent parallelizability, so that multiple computers working at the same time can perform the task faster than could a single computer. If possible, instead of having one computer work for 100 days, why not have 100 computers solve a problem in one day? *Supercomputers* combine processing hardware from multiple machines within a single motherboard; in *massively parallel processing* systems, heterogeneous computers are networked together in order to share their computing power. In 2005, the fastest supercomputer in the world consisted of 2^{32} separate processors [101]. Some projects like SETI@Home use the computing power of volunteers, organized via the Internet, to perform large computational tasks.

A more recent advance in technology is the advent of *sensor networks*. Typically, a sensor network is composed of small computers with wireless communication capability, batteries and/or solar power, a small central processor, and some physical sensors. These computers are manufactured in large quantities and are designed to be distributed about a geographical area for the purposes of monitoring that area. These networks are often used for environmental or military purposes, although a number of other applications such as disaster prevention and recovery are possible [42]. A natural extension of sensor networks is *amorphous computing*[1], wherein computing elements are small and numerous enough that they can be put in materials — like paint and concrete — by the millions.

2.2 A Formal Model

The basic model of distributed computation which we consider is the $\mathcal{CONGEST}$ model of [89]. Consider an undirected connected graph $\mathcal{G} = (V, E)$ with $|V| = n$. Each node represents a computing element, and each edge represents a reliable two-way communication channel between two nodes. At step i, the node v is allowed to look at the messages sent to it by its neighbours on step $i - 1$, perform a local computation that may change its state, and then send messages to some subset of its neighbours.

We model the messages sent along edges and the memory/state of the nodes as finite binary strings. Each node can be thought of as a Turing machine, with one tape for its memory, read-only

tapes to read incoming messages from each neighbor, and write-only tapes to send messages to each neighbor. When a node *activates,* the Turing machine is started, and when the Turing machine stops the messages for that round are sent to the node's neighbors.

We typically try to bound two different kinds of complexity in this model. First, the *time complexity* of an algorithm is the number of rounds that are necessary before the algorithm terminates. Second, the *message complexity* or *communication complexity* is the total number of messages that are sent during the course of the algorithm.

The communication model described above is *synchronous.* We may alternatively have an asynchronous setting, where nodes do not all activate at the same time. We may think of each edge (u, v) as a buffer that stores messages from u to v, to be received the next time v activates. We associate a delay with each message between the time it is sent and the earliest time at which it may be received, analogous to lag in networks. To measure time complexity in the asynchronous model, we assume the existence of a global clock, such that each node activates at least once per clock tick, and the message delay is at most one tick. Then, the asynchronous time complexity of an algorithm is the number of clock ticks elapsed before termination. Note that nodes cannot read the clock, it is only for the purpose of measuring complexity.

In our model, all messages must be $O(\log n)$ bits in size. This is roughly the amount of memory required to identify each node uniquely. This makes our algorithms practical for real applications. Furthermore, if unbounded-size messages are allowed, then any single-shot problem can be solved in $O(\mathsf{Diam})$ time, by collecting the entire network topology at a single node and then solving sequentially. We do not impose a bound on the storage space used by nodes, nor do we restrict the computational power which nodes may use for local computations.

Networks may be named or anonymous. In a *named network,* nodes are initialized containing unique integer identifiers (IDs). In an *anonymous network,* no IDs are given and so all nodes begin in the same state. Distributed algorithms may be either *deterministic* or *probabilistic/randomized*. We will see in Section 2.3 that randomization is usually necessary in anonymous systems. Even when node IDs are available, randomization sometimes gives the best known solution to a problem.

There are multiple ways in which a distributed algorithm can begin and end. Commonly, all nodes begin in a *sleep* state; a sleeping node does nothing until it receives its first message. The algorithm begins when one or more *initiators* enter a non-sleep state and begin computation. For termination, we usually have either *explicit termination*, where each node has to enter a specific "terminated" state in which no more messages can be sent, or *implicit termination*, where the states of the nodes stabilize in some permanent way even though further messages may be sent.

In the remaining sections of this chapter we discuss some classical problems of distributed com-

puting. We give explicit solutions for some and survey known results. All results pertain to the synchronized, deterministic, $O(\log n)$-bit message model unless otherwise noted.

2.3 Leader Election

Here is a formal statement of the *leader election* problem: each node has a variable *leader* initialized to the value *unknown*. All nodes are activated simultaneously, and when the algorithm terminates, exactly one node must have $leader = true$ and all others must have $leader = false$. The leader can then be used as the root of a spanning tree, or otherwise used to centralize some computation.

If the nodes are initially given unique integer identifiers, then we may elect the node with the lowest ID. In order to accomplish this, each node initially forwards its ID to each of its neighbours, and then continues forwarding the smallest ID that it knows about. After Diam rounds the smallest ID will have propagated over the entire network, and the unique node having that ID is elected. In Algorithm 2.1 we show pseudocode that describes how each individual node operates.

Algorithm 2.1 Leader election in a named network, when given a bound D on the diameter.

```
1: procedure BASIC-ELECTION
2:     Let minlabel ← ID
3:     for round ← 1 to D do
4:         if round > 1 then
5:             Let S be the set of values received from all neighbors
6:             minlabel ← min(S ∪ {minlabel})
7:         end if
8:         if round < D then
9:             Send minlabel to each neighbor
10:        end if
11:    end for
12:    If ID = minlabel, then this node is the leader, otherwise it is not the leader
13: end procedure
```

Algorithm 2.1 has three features which we might seek to eliminate. First, each node is required to have a unique ID. Second, the communication complexity mD is far from optimal. Third, the *a priori* bound D on the diameter seems unnecessary, and indeed it is. We now address these 3 issues in turn.

The need for IDs at each node is unavoidable, unless we use a randomized algorithm. As shown in [99, Theorem 9.5], there exists no deterministic algorithm for election in an anonymous network, even if each node knows the size n of the network. The proof that no such algorithm exists is based on symmetry; if we consider running a deterministic algorithm on a ring network, then the initial symmetry of the network is maintained at each step, and so it is never possible to single out one node as the leader. For this reason, in the remainder of the chapter, we always assume that nodes

have unique identifiers.

If we allow random choices in our algorithm, then the symmetry can be broken. Intuitively, a deterministic election algorithm for networks with IDs can be transformed to a randomized algorithm for anonymous networks as follows, provided we have an upper bound on n: each node selects a random label in $[1, n^3]$; run the deterministic election algorithm; if there are two nodes with the same label, pick new labels and restart. In expectation we need $O(1)$ restarts, since the probability of any two nodes generating the same ID is $O(1 - 1/n)$. The tricky part is discovering when a label conflict exists. A complete analysis of this strategy can be found in [34, p. 110].

There is a deterministic election algorithm of Awerbuch from [9] taking $O(n)$ time and $O(m + n \log n)$ messages, much better than the algorithm described here. This algorithm works in asynchronous networks, regardless of which nodes are initiators, and even if they start at different times. Under certain assumptions (asynchrony, comparison-based algorithms, or bounded-time algorithms) this algorithm is optimal, in that there exist three families of networks respectively requiring $\Omega(m)$ messages, $\Omega(n \log n)$ messages, and $\Omega(n)$ time [47].

Without being given the bound D on Diam, we can still elect a leader deterministically, as we describe shortly in Algorithm 2.4.

2.4 Spanning Trees and Applications

In this section, we consider several ways to construct rooted spanning trees of a network. A root is selected either with a leader election algorithm, or by initiating the algorithm at a single node. For the remainder of the section, we assume that the nodes of the network have unique IDs.

Having picked the root, a simple greedy algorithm can be used to construct a spanning tree of the network. First, we add the root to the tree; whenever a node joins the tree, it informs all of its neighbors of this fact. When a node that is not in the tree starts receiving messages, it picks one of the senders as its parent and joins the tree. By sending acknowledgement messages back up the tree, the root can determine when the algorithm has terminated. In Algorithm 2.2 we show pseudocode to this effect. The algorithm has optimal complexity, $O(\mathsf{Diam})$ time and $O(m)$ messages, even in an asynchronous network.

One application of a spanning tree is that it reduces the cost of a broadcast in the network; whereas m messages are necessary if we have no topological knowledge, only $n - 1$ messages are needed to broadcast along a spanning tree. Such a broadcast takes time proportional to the height of the tree.

Algorithm 2.2 Construction of a spanning tree, once the root has been chosen.

```
1: procedure SPANNING-TREE
2:     Let children ← ∅
3:     Let parent ← nil
4:     Let pending-replies ← 0
5:     if this node is the root then
6:         Send "join" to each neighbour
7:         pending-replies ← |Γ(me)|
8:     else
9:         Wait until one or more "join" messages are received
10:        Let S be the set of neighbours which sent me a join message
11:        parent ← an arbitrarily chosen element of S
12:        Send "accept" to parent
13:        Send "reject" to each node in S − {parent}
14:        Send "join" to each node in Γ(me) − S
15:        pending-replies ← |Γ(me) − S|
16:    end if
17:    while pending-replies > 0 do
18:        Wait until message action is received; let u be the sender
19:        If action ∈ {"accept," "reject"}, then pending-replies = pending-replies − 1
20:        If action = "accept," then children ← children ∪ {u}
21:        If action = "join," then reply to u with "reject"
22:    end while
23: end procedure
```

Another common application of rooted spanning trees is to compute aggregates in the network. Let f be a real-valued function defined at each node v_i. A rooted spanning tree can be used to compute $\sum_{i=1}^{n} f(v_i)$, as follows. Define the function $f^{\downarrow}$ by

$$f^{\downarrow}(v_i) := \sum_{v_j \in desc(v_i)} f(v_j),$$

where $desc(v_i)$ denotes the set of all descendants of v_i, including itself, in the spanning tree. In the protocol, each node v_i computes $f_i^{\downarrow}$ and sends this value to its parent, as shown in Algorithm 2.3. Generally, the technique of reporting information up a tree to the root is called a *convergecast* [89, Ch. 3]. The algorithm essentially boils down to the observation that

$$f^{\downarrow}(v_i) = f(v_i) + \sum_{v_j \in children(v_i)} f^{\downarrow}(v_j).$$

A convergecast takes time proportional to the height of the tree and uses $n - 1$ messages.

One neat application is that, when f is identically equal to 1, the root gets a count of the number of nodes in the network. A convergecast can similarly be used to compute any associative, symmetric function of the nodes, such as $\min, \Pi, \wedge, \gcd, \cup$, etc.

Spanning trees are used in the best known leader election algorithms [9]. In Algorithm 2.4 we

Algorithm 2.3 Simple aggregation using a convergecast on a spanning tree.

```
1: procedure AGGREGATE-VALUES
2:     tmp ← f(me)
3:     pending-replies ← |children|
4:     while pending-replies > 0 do
5:         Wait until message x is received          ▷ x = f↓(u), where u is the sender
6:         tmp ← tmp + x
7:     end while
8:     f↓(me) ← tmp
9:     if this node is not the root, then
10:        Send f↓(me) to parent
11:    else f↓(me) is the desired result
12:    end if
13: end procedure
```

present a simple protocol for election using spanning trees. The idea is that *every* node tries to build a spanning tree, but nodes prefer to join a tree with minimal-ID root. Eventually the entire network is subsumed by a tree rooted at the node with minimal ID. Unlike Algorithm 2.2, in the new algorithm the (elected) root node explicitly knows when the algorithm is over. This is accomplished via "done" messages indicating that the construction of a particular subtree is complete.

2.5 Efficient Spanning Tree Constructions

The first optimal algorithm for computing a spanning tree was given in [9], and its complexities are $O(n)$ time and $O(m + n \log n)$ messages. In fact, the algorithm can be used to compute a *minimum-weight spanning tree* (MST). Under one of several assumptions — asynchrony, time-bounded algorithms or event-driven algorithms — this bound is *existentially optimal,* that is, there are graphs for which $\Omega(n)$ time and $\Omega(m + n \log n)$ messages are provably necessary [99, Thm. 6.6] [78, §2.4.2]. As noted in [9] these upper and lower bounds also hold for computing any spanning tree, computing an MST, and electing a leader.

Despite the "optimality" of Awerbuch's result, we can actually compute an MST in $o(n)$ time on graphs with small diameter. A novel approach in [49] gives an MST algorithm with time complexity $O(\mathsf{Diam} + n^{0.614})$, which has sub-linear time on graphs with sub-linear diameter. Currently the best general protocol is from [70], with time complexity $O(\mathsf{Diam} + \sqrt{n} \log^* n)$. If all nodes initiate the algorithm simultaneously, an algorithm from [40] with running time $\tilde{O}(\mu(G, \omega) + \sqrt{n})$ can be used, where μ denotes a graph parameter than can be much smaller than Diam.

The best unconditional lower time bound on the MST problem is $\Omega\left(\mathsf{Diam} + \sqrt{\frac{n}{\log n}}\right)$. This lower bound comes from a bound [41] on the quality of approximation algorithms.

Breadth-first search (BFS) trees can be used to optimize tree-based algorithms, as the following

Algorithm 2.4 Election in a named network, using spanning trees.

```
1: Let root-ID and pending-replies be integer variables
2: Let parent be a variable referencing a neighbour of me
3: Let children be a variable referencing a subset of my neighbours
4: procedure JOIN(new-parent, new-root-ID)
5:     parent ← new-parent
6:     root-ID ← new-root-ID
7:     Let S ← Γ(me)
8:     If new-parent ≠ nil, then S ← S − {new-parent}
9:     Send ("join," root-ID) to each member of S
10:    pending-replies ← |S|
11:    children ← ∅
12: end procedure
13: procedure ELECT-WITH-TREES
14:    Call JOIN(nil, ID)
15:    loop
16:        Wait until a message (action, sender-root-ID) is received; let u be the sender
17:        if sender-root-ID > root-ID then
18:            Ignore this message
19:        else if action ="join" and sender-root-ID < root-ID then
20:            Call JOIN(u, sender-root-ID)
21:        else if action ="join" and sender-root-ID = root-ID then
22:            Send a (reject, sender-root-ID) message to u
23:        else                ▷ in this case sender-root-ID = root-ID and action ∈ {"done," "reject"}
24:            pending-replies ← pending-replies − 1
25:            If action = "done," then children ← children ∪ {u}
26:            if pending-replies = 0 then
27:                if parent ≠ nil then
28:                    Send ("done," root-ID) to parent
29:                else
30:                    Algorithm is complete, exit main loop
31:                end if
32:            end if
33:        end if
34:    end loop
35: end procedure
```

observation shows. The height of any spanning tree of $\mathcal{G}$ has height at least $\mathsf{Diam}(\mathcal{G})/2$, while a BFS tree has height at most $\mathsf{Diam}(\mathcal{G})$. Thus, a BFS tree optimizes the time to communicate up and down the tree, up to a constant factor. On a synchronous network, Algorithms 2.2 and 2.4 compute a BFS tree in optimal time. For an asynchronous network, the best known algorithm [16] takes $O(\mathsf{Diam}\log^3 n)$ time and $O(m+n\log^3 n)$ messages, and is based on applying an efficient synchronizer (see Section 4.2.1) to Algorithm 2.2. For graphs with small diameter, applying techniques of [86] to a result of [12] gives a slightly faster algorithm with time complexity $\mathsf{Diam}^{1+O(\log^{-1/4} n)}$.

A depth-first search (DFS) tree can also be efficiently constructed in the network. The best known algorithm, from [8], works as follows. A token performs a DFS of the network. Whenever v is visited for the first time, v announces to all of its neighbors that it has been visited, so that the token does not try to explore v more than once; it will pass through v exactly one more time, as it backtracks the DFS to the parent of v. This takes $O(n)$ time and $O(m)$ messages. In Section 3.9, we show how DFS trees can be used to distributively compute the strongly-connected components of a directed graph.

2.6 Optimization Problems

In this section we survey distributed solutions to some well-known optimization problems of theoretical and practical importance. We focus on algorithms with the best time complexity, as is typical in the literature; even so, generally speaking, the fastest algorithms are fairly communication-efficient as well.

2.6.1 Matching

Recall that a matching of $\mathcal{G}$ is a subset S of the edges of $\mathcal{G}$ such that each node is incident on at most one edge in S. A *maximum* matching is a matching of largest cardinality, and a deterministic algorithm in [94] computes a maximum matching, by using augmenting paths, in $O(n\log n)$ time. A matching is *maximal* if no other edges can be added to S without destroying the fact that it is a matching. The fastest known algorithm for computing a maximal matching [56] takes $O(\log^4 n)$ time. Recently, a lower bound of $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ time was proved [69] for maximal matching. Matchings are useful in load balancing [51].

Note that the former problem is inherently global (a matching is maximum if no augmenting paths exist) while the latter one is inherently local (a matching is maximal if every vertex is either matched, or has no unmatched neighbours). Global problems generally require $\Omega(\mathsf{Diam})$ time, while local ones may admit $o(\mathsf{Diam})$ solutions. Thus, in what follows, $O(\mathsf{Diam})+o(n)$ time complexity is be

called *sub-linear,* while $o(n)$ is be called *local.* Note that a local algorithm is necessarily initiated at more than one nodes, and usually it is assumed that *all* nodes initiate the algorithm, simultaneously.

2.6.2 Maximum s-t Flow

In the capacitated s-t maximum flow problem for a graph, we are given a directed graph, and a capacity c_{uv} for each edge (u, v). The task is to assign non-negative flow values to each edge so that

1. Each edge flow value is no more than its capacity.
2. For $v \notin \{s, t\}$, the in-flow sum of v is equal to the out-flow sum of v.
3. The out-flow sum at s, which equals the in-flow sum at t, is to be maximized.

One application of flows is as a modeling tool, such as for data transfers in computer networks, shipments of commodities across a country, and fluid networks.

The classic sequential solution for this problem was developed by Ford and Fulkerson [46], and augments the flow by finding paths from s to t in a "residual graph." The running time is improved by using augmenting paths of minimum length [39].

In [54], Goldberg and Tarjan give an algorithm based on the method of *preflows* which does not directly involve augmenting paths, but works on similar principles. Their algorithm is easily adapted into an asynchronous distributed algorithm, and the resulting time complexity is $O(n^2)$. In Section 4.6.4 we describe the algorithm in more detail.

There is also an elegant simulation algorithm [13][14] for computing $(1 + \epsilon)$-optimal solutions to *multicommodity* max-flow problems in $O(mn\epsilon^{-2} \log(m\epsilon^{-1}))$ rounds.

2.6.3 Maximal Independent Set, Vertex Coloring

An *independent set* is a subset S of V such that no two nodes in S are adjacent. The set S is a *maximal* independent set (MIS) if no proper superset of S is independent. A *vertex k-coloring* of G is a map from V to the colors $\{1, \ldots, k\}$ so that adjacent nodes are colored differently. Let Δ denote the maximum degree of any node of $\mathcal{G}$. It is easy to show, by a greedy argument, that $\mathcal{G}$ has a $(\Delta + 1)$-coloring. There is a connection between $(\Delta + 1)$-colorings and independent sets, as the following observations from [89, Cor. 8.2.2] and [71] show.

Theorem 2.6.1. *Suppose that a distributed algorithm exists which computes a $k(\mathcal{G})$-coloring of $\mathcal{G}$ in $T(\mathcal{G})$ time. Then there exists a distributed algorithm that, given a graph $\mathcal{G}$, computes a MIS of $\mathcal{G}$ in $k(\mathcal{G}) + T(\mathcal{G})$ time.*

Proof. First, we run the $k(\mathcal{G})$-coloring algorithm. The rest of the algorithm constructs a MIS S in $k(\mathcal{G})$ rounds. Initially, no nodes are in S; in round i, each node of color i joins S if possible, that is, if none of its neighbors have joined S. Since no two adjacent nodes can join in the same round, and every node tries to join at least once, S is a MIS. □

Theorem 2.6.2. *Suppose that a distributed algorithm exists which computes a MIS of $\mathcal{G}$ in $T(\mathcal{G})$ time. Then there exists a distributed algorithm for computing a $(\Delta+1)$-coloring of $\mathcal{G}$ in $T(\mathcal{G})$ time, using messages at most $(\Delta+1)$ times larger than those of the MIS algorithm.*

Proof. Denote by $K_{\Delta+1}$ the complete graph on $\Delta+1$ vertices. It is easy to see that there is a bijection between MIS's of $K_{\Delta+1} \times \mathcal{G}$ and $(\Delta+1)$-colorings of $\mathcal{G}$. We have each node of $\mathcal{G}$ simulate a $(\Delta+1)$-clique, and run the MIS algorithm on this virtual graph. The potential increase in message size is due to the fact that $\Delta+1$ edges join adjacent cliques in the virtual graph. □

Note that in practice, the message size increase can usually be avoided.

The best general deterministic result for $(\Delta+1)$ coloring and MIS was pioneered in [12], and was improved in [86] to run in $\exp(O(\sqrt{\log n}))$ time. For low-degree graphs, there are faster algorithms of times $O(\Delta \log n)$ and $O(\log^* n + \Delta^2)$ as described in [89, Lemma 7.4.1] and [71], respectively. Trees can be 3-colored in $O(\log^* n)$ time, provided that each node initially knows its parent [89, §7.3].

If we allow randomization, we can compute both a MIS and a $(\Delta+1)$-coloring in $O(\log n)$ time; the original algorithms are from [76] and [5], and good expositions are in [82, §12.3] and [89, §8.4]). In [86] it is shown that, with randomization, a $(\Delta+1)$-coloring can be converted into a Δ-coloring in $O(\log^3 n/\log\Delta)$ expected time, provided that a Δ-coloring exists.

The authors of [69] give a lower bound for MIS: namely, for each n, there exist graphs taking $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ time, and for each Δ, there exist graphs taking $\Omega\left(\frac{\log\Delta}{\log\log\Delta}\right)$ time. However, both bounds do not necessarily apply simultaneously to all graphs, since the $O(\log^* n + \Delta^2)$ algorithm of [71] beats the former bound on bounded-degree graphs. Linial proves in [71] that any $o(\mathsf{Diam})$ distributed algorithm for coloring a $d-$regular tree uses at least $\sqrt{d}/2$ colors. These lower bounds also apply to randomized algorithms.

2.6.4 Edge Coloring

Instead of coloring the nodes of a graph, we may color the edges, with the constraint that each node must have all of its incident edges labeled differently. It is easy to show that any graph can be $(2\Delta-1)$-edge-colored, and Vizing showed that any graph can be $(\Delta+1)$-edge-colored.

All of the known fast algorithms for edge coloring are randomized. The first local protocol, from [87], used $(1.6+\epsilon)\Delta + 0.4\log^{2+\delta} n$ colors, and ran in $O(\log n)$ rounds. Subsequent work [37] reduced this bound to $\Delta(1+\epsilon)$ colors and kept the $O(\log n)$ time bound, except on certain special cases. In [55], a a new algorithm for $\Delta(1+\epsilon)$ coloring is given that runs in $O(\log\log n)$ time, provided that certain constraints between Δ and n are met. These coloring algorithms have been used [38][3] to improve the efficiency of communication in networks, as each color class corresponds to a set of data transfers that can occur in parallel.

2.6.5 Small Dominating Sets and Ruling Sets

For $k \in \mathbb{Z}$, a k-dominating set of $\mathcal{G}$ is a subset of vertices such that all nodes in $\mathcal{G}$ are within k steps of one of these vertices. A dominating set is *small* if it contains at most $\max\left(\frac{n}{k+1}, 1\right)$ vertices; it is easy to show that k-dominating sets exist in every graph. In [49] it is shown that small 1-dominating sets on rooted trees can be computed in $O(1)$ time plus one $O(\log^* n)$ time call to MIS; this observation led to the first sub-linear MST algorithm. In [70] it is shown that a small k-dominating set can be distributively constructed in $O(k\log^* n)$ time, and this observation leads to an even faster MST algorithm.

A (j,k)-ruling set is a k-dominating set with the additional property that no two nodes are within a distance j of each other. An algorithm of [12] shows that a $(k, k\log n)$-ruling set can be constructed in $O(k\log n)$ time. In [86], an algorithm is given to construct (3, 4)-ruling sets in $\exp O(\sqrt{\log n})$ time.

2.7 Locality-Preserving Representations

In this section, we describe several ways to reduce a network into simpler components. These components have multiple nice properties — few edges, small clusters, low overlap, low chromatic number — so that we can simplify the network (say, by removing edges) without losing too much useful structure. We refer the reader to [89] for a comprehensive description.

2.7.1 Neighborhood Covers

A *sparse* (κ,ρ)*-neighborhood cover* of $\mathcal{G}$ is a collection of connected vertex sets ("clusters") such that

1. For each vertex v, the closed ρ-neighborhood of v is entirely contained in some cluster.
2. The subgraph of $\mathcal{G}$ induced by each cluster has diameter $O(\kappa\rho)$.

3. Each node belongs to $O(\kappa n^{1/\kappa})$ clusters. (Or, $O(\log n \cdot n^{1/\kappa})$ clusters in [40]).

Here κ and ρ are integer parameters. An excellent reference for these covers is [89, Ch. 21], where a deterministic protocol for constructing these covers is given with time complexity $O(n^{1+1/\kappa}\kappa^2)$ and communication complexity $O(m + n^{1+1/\kappa}\kappa^2)$. In [11] this algorithm was adapted to asynchronous networks, with a factor of about $\kappa \log n$ increase in both complexities. The full version of [40] gives a randomized *local* algorithm for computing sparse neighborhood covers which, with high probability, runs in $O(\rho\kappa^2 n^{1/\kappa} \log n)$ time and $O(m\kappa n^{1/\kappa} \log n)$ messages on a synchronous network. Neighborhood covers are used in the adaptation of near-optimal synchronizers to dynamic networks in [15]. The randomized protocol of [40] leads to a new sub-linear MST algorithm, and it also proves useful in Section 3.8.

2.7.2 Decompositions

A (b, c)-*cluster decomposition* of $\mathcal{G}$ is a partition of $V(\mathcal{G})$ such $d_G(u, v) \leq b$ whenever u and v are in the same component, and the components can be c-colored such that adjacent nodes of different components never have the same color. The best cluster decomposition algorithms are built using ruling sets, and in turn, decompositions give efficient MIS and coloring algorithms. An $(O(\log n), O(\log n))$-decomposition can be constructed in $\exp(O(\sqrt{\log n}))$ time deterministically [86], or in $O(\log^2 n)$ time probabilistically [73]. For other parameters (say $b = O(1)$ or $c = O(1)$) efficient distributed algorithms do not seem to be known, although a sequential algorithm of [89, §14.2] can construct a $O(\kappa), O(\kappa n^{1/\kappa})-$decomposition for any $\kappa \in \mathbb{Z}^+$.

2.7.3 Partitions

In [16] Awerbuch introduces a distributed algorithm for computing a *partition* of V, with parameter $\kappa \in \mathbb{Z}^+$, such that each component has diameter at most $2(\kappa - 1)$, and at most $n^{1+1/\kappa}$ edges have endpoints in different components. With improvements from [89, §20.6], this can be made to run with $O(n)$ time and $O(m)$ communication complexity. In [7] partitions are used to construct an efficient "γ" synchronizer. In [17] sparse partitions are applied to *regional matchings;* these matchings, in turn, gave the first near-optimal synchronizer [16].

2.7.4 Spanners

A κ-*spanner* of $\mathcal{G}$ is a spanning subgraph $\mathcal{G}'$ of $\mathcal{G}$, such that $d_{\mathcal{G}'}(v_1, v_2) \leq \kappa d_{\mathcal{G}}(v_1, v_2)$ for all $v_1, v_2 \in \mathcal{G}$. Spanners are most useful when they have few edges, since $\mathcal{G}$ is trivially a κ-spanner of itself. From a

κ-partition we can construct a κ-spanner with $O(n^{1+1/\kappa})$ edges: take the union of all intercomponent edges with a spanning tree for each component. Elkin notes in [40] that his randomized local protocol for a $(\log n, 1)$-neighbourhood cover can be used to construct a κ-spanner more efficiently: choose a spanning tree from each cluster, and take their union. Thus we can construct a $O(\log n)$-spanner using $O(n \log^3 n)$ edges in $O(\log^2 n)$ time and using $O(m \log n)$ messages.

2.8 Remarks

Some ideas from the distributed model also have applications in sequential and parallel computing. The preflow algorithm [54] led to a then-fastest sequential solution to the maximum flow problem. Locality-preserving representations have been used to improve sequential approximation algorithms [89, Ch. 28], including $\mathcal{NP}$-hard problems such as minimum-cut. The field of parallel algorithms often shares ideas and algorithms with the distributed world [76][5][98].

Many problems — in particular neighborhood covers, MIS/coloring, cluster decompositions, and spanners — the best randomized solutions are faster than the best deterministic ones. In anonymous networks randomness is provably necessary for elections and many other algorithms, but it seems that nobody has shown that randomness is essential to the fastest algorithms on named networks. It would be interesting to have a result to this effect, analogous to the sequential result of [82, §2.1] for the game tree evaluation problem.

In the $\mathcal{CONGEST}$ model which we consider, individual nodes are allowed unlimited storage and unlimited computational power. This is in some cases unrealistic[1]. One would imagine that low-memory protocols would be useful, especially in lieu of large, low-power sensor networks and amorphous computing environments. For many applications, and Algorithms 2.1–2.4, $\delta \log^{O(1)} n$ bits of memory per node suffice; but it is not clear what can be computed with this much memory. For example, no known sub-linear MST algorithm uses $\delta \log^{O(1)} n$ memory per node.

Local and sub-linear algorithms — such as those for maximal matching, coloring, and MST — will become more useful as the trend towards large sensor networks and amorphous computing continues. Ideally, huge networks should still be able to compute useful information locally, without need for global coordination. Two suspects that seem to be open to investigation are a sub-linear algorithm for DFS trees, and a local algorithm for partitions.

Some problems with well-known sequential solutions do not seem to have fast distributed solutions. This includes: single-source connectivity, vertex-connectivity, computing k-connected compo-

[1]Further, with unlimited memory, *any* single-shot problem can be solved centrally in $O(m)$ time using $O(m\mathsf{Diam})$ messages, by sending the entire network topology to a single node; however, it may be that some problems cannot be solved at all with low-memory protocols, which would be interesting from a complexity-theoretic standpoint.

nents for $k > 2$, planarity testing, and computing the diameter exactly.

One final remark: the "complexity classes" of sequential computation such as $\mathcal{P}, \mathcal{NP}, \mathcal{PSPACE}$ do not seem to have well-studied distributed counterparts. It would be nice to formalize observations of hardness equivalence (MIS with (Δ+1)-coloring, leader with MST, etc) into a complexity-theoretic framework. The main obstacle in doing so seems to be the large number of choices (synchrony, forms of activation, etc) in the distributed model, although there doesn't seem to be an inherent reason why these obstacles should not be overcome.

Chapter 3

An $O(\mathsf{Diam})$ Time Edge-Biconnectivity Algorithm

Let $\mathcal{G}$ be a connected graph. An edge (node) of $\mathcal{G}$ is said to be a *bridge* (*articulation point*) if its removal causes $\mathcal{G}$ to become disconnected. The *biconnected components* of $\mathcal{G}$ are the connected components of $\mathcal{G}$ after we delete all bridges. Equivalently, the biconnected components are the maximal induced subgraphs of $\mathcal{G}$ which remain connected even after an edge is deleted. The *edge-biconnectivity* problem is to identify all bridges and biconnected components of $\mathcal{G}$. One application is that, in designing a network, the bridges and articulation points highlight where more connections should be added in order to make the network more robust.

In this section, we present an optimal distributed algorithm for determining edge-biconnectivity. Given a leader, its complexities are $O(\mathsf{Diam})$ time and $O(m)$ messages, with a polylog(n) increase in the asynchronous setting. It meets the $\mathcal{CONGEST}$ model, and additionally has small memory requirements at each node. We also show how to modify this into a near-optimal *local* algorithm.

3.1 Overview

We use a tree $\mathcal{T}$ in our algorithm. Let $h(\mathcal{T})$ denote the height of $\mathcal{T}$, and $desc(v)$ denote the descendants of v in $\mathcal{T}$, including v itself. Let $\mathcal{C}$ denote the union of all simple cycles,

$$\mathcal{C} = \{e \in E \mid e \text{ lies within some simple cycle of } G\}.$$

The algorithm operates in 6 phases, as follows:

1. Construct a breadth-first search (BFS) tree $\mathcal{T}$ with root node v_0.
2. At each node v, compute $\#desc(v)$, the number of descendants of v in $\mathcal{T}$.
3. Compute a preorder labeling of $V(\mathcal{G})$ with respect to $\mathcal{T}$.
4. Identify all cross-edges (that is, all edges not belonging to $\mathcal{T}$).
5. By sending messages from cross-edges up to the root v_0, mark each edge in $\mathcal{C}$.
6. By broadcasting along the tree, label the nodes according to their biconnected components.

3.2 Preorder Labeling

To begin, we need to elect a leader in the network. As noted in Section 2.3 it is sufficient either that nodes have unique IDs, that nodes have access to a random number generator, or that the algorithm is initiated at a single node. Next, we compute a spanning tree $\mathcal{T}$, with the elected leader as the root. Recall the efficient constructions mentioned in Section 2.5. The algorithm runs fastest when $\mathcal{T}$ is a BFS tree, but for the purposes of correctness any tree will do.

The computation of $\#desc(v)$ at each node in Phase 2 can be accomplished in $2h(\mathcal{T})$ steps. First, the root node sends "Compute $\#desc$ of yourself" to each of its children, and this message is downcasted in $\mathcal{T}$. Then, we run the simple aggregation protocol (Algorithm 2.3) with $f(v) = 1$ at each node; it is easy to see that $f^{\downarrow}(v) = \#desc(v)$.

After this phase, a preorder labeling of $\mathcal{T}$ is computed by using another downcast, as follows:

- The root node sets its own *PreLabel* field to 1.
- Whenever a node v sets its *PreLabel* field to ℓ, it orders its children in $\mathcal{T}$ arbitrarily as $c_1, c_2, \ldots$. Then v sends the message "Set your *PreLabel* field to ℓ_i" to each c_i, where ℓ_i is computed by v as
$$\ell_i = \ell + 1 + \sum_{j<i} \#desc(c_j)$$

After $h(\mathcal{T})$ time steps, we will have computed a preordering of $\mathcal{T}$.

3.3 Least Common Ancestors

In order to simplify the presentation, we hereafter refer to nodes simply by their preorder labels. The preordering allows us to reduce congestion in Phase 5 of the algorithm, using the following properties.

Lemma 3.3.1. *The descendants of a node v in the tree $\mathcal{T}$ are precisely*

$$desc(v) = \{u \mid v \leq u < v + \#desc(v)\}.$$

Let $\mathrm{LCA}(u_1, u_2, \ldots)$ denote the lowest (by position, not value) common ancestor of nodes $u_1, u_2, \ldots$ in the tree $\mathcal{T}$.

Theorem 3.3.2. *If $v_1 \leq v_2 \leq v_3$, then $\mathrm{LCA}(v_1, v_3)$ is an ancestor of v_2.*

Proof. Let $a = \mathrm{LCA}(v_1, v_3)$. By Lemma 3.3.1, $a \leq v_1 \leq v_3 < a + \#desc(a)$. Thus $a \leq v_2 < a + \#desc(a)$, and by Lemma 3.3.1, v_2 must also be a descendant of a. □

Corollary 3.3.3. $\mathrm{LCA}(u_1, u_2, \ldots u_k) = \mathrm{LCA}(\min_i(u_i), \max_i(u_i))$.

Corollary 3.3.4. *If $u_i \leq v_i$ for all i, then*

$$\mathrm{LCA}(\mathrm{LCA}(u_1, v_1), \mathrm{LCA}(u_2, v_2), \ldots \mathrm{LCA}(u_k, v_k)) = \mathrm{LCA}(\min_i(u_i), \max_i(v_i)).$$

3.4 Marking Cycle Edges

The goal of Phases 4 and 5 is to mark the set $\mathcal{C}$, which consists of the union of all simple cycles.

When v' is an ancestor of v in $\mathcal{T}$, let $\mathrm{Chain}(v', v)$ denote the set of edges on the path from v' to v in $\mathcal{T}$. The edges appearing in $\mathcal{G} - \mathcal{T}$, commonly known as the *cross-edges,* permit a simple formula for $\mathcal{C}$:

Lemma 3.4.1.

$$\mathcal{C} = \bigcup_{(u,v) \in \mathcal{G} - \mathcal{T}} \{(u, v)\} \cup \mathrm{Chain}(\mathrm{LCA}(u, v), u) \cup \mathrm{Chain}(\mathrm{LCA}(u, v), v). \tag{3.1}$$

Proof. Note that each set $\{(u, v)\} \cup \mathrm{Chain}(\mathrm{LCA}(u, v), u) \cup \mathrm{Chain}(\mathrm{LCA}(u, v), v)$ is a simple cycle. It remains to show that this union formula contains *all* edges appearing in simple cycles. Suppose otherwise, that the above formula missed some edge (u, v) belonging to a simple cycle K of $\mathcal{G}$. Since Equation (3.1) includes all edges of $\mathcal{G} - \mathcal{T}$, we can assume that $(u, v) \in \mathcal{T}$, without loss of generality u the parent of v.

Let the cycle K contain, in order, the nodes $(k_0 = v, k_1, k_2, \ldots, k_{m-1} = u, k_m = v)$. If k_i is the first element of this list not in $desc(v)$, then (k_{i-1}, k_i) is a cross edge and

$$(u, v) \in \mathrm{Chain}(\mathrm{LCA}(k_{i-1}, k_i), k_{i-1}).$$ □

Thus, to mark the edges of $\mathcal{C}$, it suffices to just mark chains going up from each cross edge to its endpoints' LCA.

We could distributively mark the edges in $\text{Chain}(\text{LCA}(u, v), v)$ as follows:

- For each cross edge (u, v),

  Send a message from v to u which states "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent as being in $\mathcal{C}$."

  Send the same message from u to v.

We will abbreviate the message "If you are an ancestor of both u and v, ..." as "Join to $\text{LCA}(u, v)$". Without loss of generality, we will send our messages such that $u \leq v$.

Upcasting these messages naively leads to congestion. Namely, a parent will be faced with broadcasting many of these messages up simultaneously, which would either break the $O(\log(n))$ restriction on message sizes or our desired $h(\mathcal{T})$ time bound. The following Forwarding Rule fixes this congestion:

- If a node w simultaneously receives messages "Join to $\text{LCA}(u_i, v_i)$" for $i = 1 \dots k$, it should compute $u_{\min} = \min_i u_i$ and $v_{\max} = \max_i v_i$. If w is an ancestor of both $u_{\min}$ and $v_{\max}$, then no message is sent up. Otherwise, w should send "Join to $\text{LCA}(u_{\min}, v_{\max})$" to its parent, and mark the edge connecting w to its parent as being in $\mathcal{C}$.

Theorem 3.4.2. *The Forwarding Rule correctly marks* $\text{Chain}(\text{LCA}(u, v), v)$ *for each cross edge* (u, v).

Proof. Suppose that w, as described, is asked to propagate messages so that all edges in

$$\bigcup_i \text{Chain}(\text{LCA}(u_i, v_i), w)$$

become marked. They must all lie on the unique path between w and the root of $\mathcal{T}$, so we only need to mark the longest chain. The highest LCA is equal to $\text{LCA}(\text{LCA}(u_1, v_1), \text{LCA}(u_2, v_2), \dots)$ and by Corollary 2 this is $\text{LCA}(u_{\min}, v_{\max})$, so the propagated message (if any) is correct. □

3.5 Biconnected Decomposition

The following observations allow us to determine the articulation points and bridges.

Claim 3.5.1. *The bridges are precisely those tree edges which are not in $\mathcal{C}$.*

Proof. If an edge is in a cycle, then its deletion does not affect the connectedness of its two endpoints, and so it is not a bridge. Similarly, a bridge cannot be in a cycle. □

Note that only tree edges can be bridges; thus, if each non-root node stores a boolean variable indicating whether the edge to its parent is a bridge, then this suffices to identify all bridges.

To group the nodes according to their biconnected component, we need to broadcast an identifier along each component. The following claim means that a simple downcast along the edges of $\mathcal{T}$ will suffice.

Claim 3.5.2. *If we delete all bridges from $\mathcal{T}$, then the forest of resulting trees is a set of spanning trees for the biconnected components of $\mathcal{G}$.*

In Algorithm 3.1 we show a precise formulation of how this works.

3.6 Optimization and Analysis

The algorithm described above is nearly optimal. Assuming that we are given a spanning tree $\mathcal{T}$, the time complexity of each phase is $h(\mathcal{T})$, and, except for Phase 5, the message complexity is $O(m)$. However, the number of "Join to" messages (JMs) that Phase 5 sends may be as large as $\Omega(mh(\mathcal{T}))$. In order to reduce this, we modify the algorithm so that only a constant number of JMs are sent along any edge. First, without modification, exactly 2 JMs are sent along each cross edge. Second, we now insist that every node must wait until it hears from all of its non-parent neighbors before forwarding a JM to its parent, so that it reports to its parent at most once. We also add a "null" message which will be sent along bridges, so that every node reports to its parent exactly once. The full protocol is shown in Algorithm 3.1.

Note that this algorithm uses only $\Theta(\log n)$ bits of memory at each node. It is probably impossible to determine the bridges with only $o(\log n)$ bits of memory per node, but this does not seem easy to prove.

The best running time for our algorithm is obtained when $\mathcal{T}$ is a BFS tree, since this minimizes the height up to a constant factor. The exact time complexity depends on the model used. If we are in the synchronous setting, and a leader has already been chosen, then the entire algorithm runs in $O(\mathsf{Diam})$ time with $O(m)$ messages. If a leader has not been chosen, we need an additional $O(n \log n)$ messages to elect one. If we are in a fully asynchronous setting, then the construction of a BFS will take a further $O(n \log^3 n)$ messages and $O(\mathsf{Diam} \log^3 n)$ time [89, §5.3], as well as extra

Algorithm 3.1 Fast algorithm for edge-biconnected decomposition, given a spanning tree.

```
1: procedure EDGE-BICONNECTIVITY
2:     Let k be my preorder label, and desc, children, parent describe the spanning tree
3:     u_min ← k
4:     v_max ← k + |desc| − 1
5:     pending-replies ← |Γ(me)|
6:     If I am not the root, then pending-replies ← pending-replies − 1
7:     for each non-tree edge e incident on me do
8:         Send "Cross edge from k" along e
9:     end for
10:    for i ← 1 to pending-replies do
11:        Wait until a message is received
12:        if the message is "Join to LCA(u, v)" then
13:            u_min ← min(u_min, u)
14:            v_max ← max(v_max, v)
15:        else if the message is "Cross edge from ℓ" then
16:            u_min ← min(u_min, ℓ)
17:            v_max ← max(v_max, ℓ)
18:        end if
19:    end for
20:    if u_min ∈ desc and v_max ∈ desc then
21:        Send a null message to parent
22:        under-bridge ← true          ▷ The edge joining me to my parent is a bridge
23:    else
24:        Send "Join to (u_min, v_max)" to parent
25:        under-bridge ← false         ▷ The edge joining me to my parent is not a bridge
26:    end if
27:    if I am the root or under-bridge = true then
28:        Biconnected-component ← k
29:        Send "My biconnected component is k" to each node in children
30:        Ignore any remaining message that is received from parent
31:    else
32:        Wait until parent sends me the message "My biconnected component is c"
33:        Biconnected-component ← c
34:        Send "My biconnected component is c" to each node in children
35:    end if
36: end procedure
```

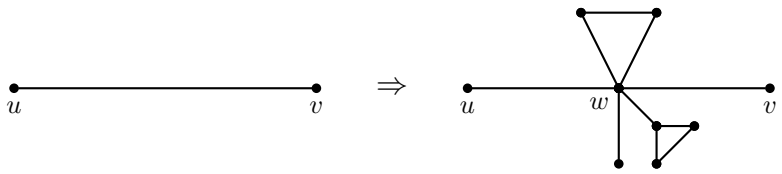


Figure 3-1: Modification of $\mathcal{G}$ into $\mathcal{G}'$, upon which a biconnectivity algorithm fails.

memory at each node. If we are in an moderately asynchronous setting where nodes satisfy the *s-archimedean assumption* [99, §12.5] — roughly speaking, that the fastest node runs s times as fast as the slowest node — then the naïve tree-building algorithm (Algorithm 2.2) makes $h(\mathcal{T})$ at most $O(s \cdot \mathsf{Diam})$, which may be better than using a synchronized BFS construction.

3.7 Optimality

We claim that the performance of this algorithm cannot be improved beyond constant factors, when the algorithm is initiated at a single node, and we work in the synchronous setting. To be precise, any singly-initiated protocol for bridge-finding must send at least m messages and take at least Diam time, or else the protocol will not work on all graphs. These lower bounds are similar in nature, and both depend on the fact that the whole network must be explored.

First we describe the lower bound for messages. Suppose that, when the protocol is executed on some graph $\mathcal{G}$, there is an edge (u, v) along which no messages are sent. Let $\mathcal{G}'$ be a graph obtained from $\mathcal{G}$ by adding a new node w and dividing (u, v) into two edges (u, w) and (w, v). We also attach some cycles and bridges to w, as shown in Figure 3-1. When we run the protocol on $\mathcal{G}'$, it must be that no messages ever reach w, nor do they reach the new nodes and edges. Consequently, the algorithm cannot correctly determine whether the new edges are bridges or not.

The lower bound on the time complexity of Diam rounds is similar. If an algorithm uses less than Diam steps on some graph $\mathcal{G}$, then there are parts of the graph which no messages reach. Consequently, we can modify $\mathcal{G}$ so that the algorithm operates incorrectly.

Note that these lower bounds apply to *all* graphs. In comparison, a $O(n)$-time algorithm of [26] was called "optimal" because *some* graphs require $O(n)$ time to find their bridges. The $O(n)$ algorithm is *existentially optimal,* since there exist some instances on which the protocol is optimal, and our $O(\mathsf{Diam})$ algorithm is *universally optimal,* since it has optimal running time on all instances. The different types of optimality were first observed by [88] and [10] in the context of leader election, and further discussion appears in [49] [40]. Universal optimality allows us to precisely state that the

inherent complexity of the biconnectivity problem is $\Theta(\mathsf{Diam})$ time and $\Theta(m)$ messages.

3.8 A Near-Optimal Local Algorithm

For now, let us forget the problem of labeling nodes according to their biconnected component, and only worry about identifying all of the bridges in a graph. We consider initiating all nodes at the same time, and want to know how long it will be before all edges are correctly identified as bridges or non-bridges. By removing the assumption of a single initiator, we can beat the lower time bound of Diam.

Suppose we remove the restriction on the message size. Then, each node can broadcast everything it knows about its local topology after each step, and so after t steps each node will know its own t-neighbourhood. Here is a local algorithm for bridge-finding. Initially, each edge is assumed to be a bridge; whenever a node learns of a cycle in its neighbourhood, it informs all of the edges in that cycle that they are not bridges. In this way we distributively determine $\mathcal{C}$, the union of all cycles in $\mathcal{G}$.

Now, let us determine the time before this algorithm has correctly identified the non-bridges. A non-bridge e will be identified as soon as a cycle containing e is known by a node; we call such a cycle a *witness* for e. We need each edge to be identified by a witness in order for the algorithm to be correct. Define the *cycle-witness radius* of $\mathcal{G}$, denoted $\Upsilon(\mathcal{G})$, by

$$\Upsilon(\mathcal{G}) = \max_{e \in \mathcal{C}} \min_{\substack{C \text{ a cycle} \\ C \ni e}} \min_{v \in V(\mathcal{G})} \max_{u \in C} d_{\mathcal{G}}(u, v).$$

Then the cycle-witness radius is the minimum time needed to identify all of the non-bridges (and, it will take another $\Upsilon(\mathcal{G})$ rounds to notify those edges). Further, $\Upsilon(\mathcal{G})$ is a lower bound on the number of rounds before all edges can be correctly identified, similar to the bounds of Section 3.7.

To get an algorithm that still runs under the $O(\log n)$ message size bound, we use a $(\log n, \Upsilon)$-neighbourhood cover, as defined in Section 2.7.1. Each edge will have a witness that is entirely contained within one cluster. Thus, we can run our Diam-time algorithm on each cluster in parallel, and all non-bridges will be witnessed in *some* cluster. Since each node may be in $O(\log n)$ clusters, there will be congestion; however, this will only increase the time of the biconnectivity algorithm by a factor of $O(\log n)$, since each node can rotate between participating in its containing clusters. The resulting local algorithm takes $O(\Upsilon \log^3 n)$ time and $O(m \log^2 n)$ messages to construct the clusters, then a further $O(\Upsilon \log n)$ time and $O(m \log n)$ messages to determine the non-bridges.

Finally, it is unlikely that Υ can be computed efficiently and/or locally. However, an algorithm

can successively "guess" $\Upsilon = 1, 2, 4, 8, \ldots$, and run the local algorithm for each value in turn. Once the guess is larger than the actual value of Υ, all edges will be correctly classified; this algorithm becomes correct within $O(\Upsilon \log^3 n)$ rounds and uses $O(m \log^2 n \log \Upsilon)$ messages. We note that this is essentially a *will-maintaining algorithm* as defined by Elkin in [40].

3.9 Other Extensions

With a small modification, the algorithm of this chapter can also be used to compute the strongly-connected components of a graph. We require that all directed edges function as 2-way communication channels. We compute a DFS tree $\mathcal{T}$ of the network that agrees with the direction of the edges. It is easy to show that an analog of Equation (3.1) holds in this case. The resulting algorithm takes $O(h(\mathcal{T}))$ time and $O(m)$ messages, identifies the edges that belong to cycles, and labels all nodes according to their strongly-connected component. Using Awerbuch's DFS algorithm from [8] gives a total of $O(n)$ time and $O(m)$ communication complexity, and this algorithm can be adapted without a synchronizer to an asynchronous network.

If a short DFS tree could be identified in sub-linear time, then we might be able to get a sub-linear algorithm for identifying strongly connected components.

Question 3.9.1. *Does there exist a $O(\mathsf{Diam})$-time distributed DFS tree construction algorithm, using messages of size $O(\log n)$?*

Computing a DFS tree is known to be hard in another sense — namely, that the problem of computing a lexicographic DFS tree is $\mathcal{P}$-complete [92] — so this sublinear algorithm may not exist. On the other hand, there is a divide-and-conquer algorithm [44] for strongly connected components, which may lend itself to distributed and parallel implementation.

As we mention in Section 6.7.2, 2-vertex connectivity is important for some applications, so it would be nice to determine whether a graph has any articulation points. It does not seem that our algorithm can be easily modified to determine articulation points. Articulation points can, in contrast, be computed by the $\Theta(n)$ time DFS-based biconnectivity algorithms of [4] and [58].

We might also try to determine the *triconnected* [59] components of a graph. There are efficient parallel algorithms for this problem [48]. We state now a lemma which may give a characterization of the triconnected components of a graph that can be computed distributively. See Section 4.7.3 for a simple way to prove this fact.

Claim 3.9.2. *Let $\mathcal{G}$ be a graph with no bridges. Define the relation $\sim_D$ on the edges of $\mathcal{G}$ by $x \sim_D y$ if the graph $\mathcal{G} - x - y$ is not connected. Then $\sim_D$ is an equivalence relation.*

3.10 Previous Work

There are a number of previous distributed bridge-finding algorithms. Several [4] [58] [26] use DFS trees à la Tarjan [96]. Another algorithm [66] uses an "ear decomposition" of the network. The best of these algorithms [4] has $O(n)$ time and $O(m)$ message complexity.

Some previous algorithms [26][25][23] use messages of size $\Omega(n)$ bits, including an incremental algorithm [95] and a self-stabilizing algorithm [64]. The protocol of [60] is similar to ours, but it uses $O(mn)$ messages. We also note a self-stabilizing algorithm of [27] using $O(n^2)$ time.

The first sub-linear algorithm was presented by Thurimella in [100]. That algorithm has $O(\mathsf{Diam}+n^{0.614})$ time complexity, although it is fairly complicated and uses $\tilde{\Omega}(n)$ bits of memory at some nodes.

In comparison to the DFS-based algorithms, the innovation of our algorithm is that *any* spanning tree can be used to efficiently compute the biconnected components. This seems to have been originally noticed in 1974 by Tarjan [97], using a post-order traversal. In [102] it was observed that a pre-order traversal would also work. The idea of using arbitrary trees has been exploited before in parallel [98][105] and distributed [60] settings.

The strongly-connected component algorithm of Section 3.9, with complexity of $O(m)$ messages and $O(n)$ time, seems to do as well as any other known distributed algorithm for that problem. However, the author has yet to research that problem in depth. It seems that there are few papers on this topic, perhaps because a sense of direction is conceptually incompatible with edges that allow 2-way communication.

Chapter 4

Faults and The Balancing Model

In Chapter 2, we introduced a model of distributed computation. In this chapter, we discuss how that model is not realistic. We survey some known techniques for adapting the simple model to real-world computation. We also present a new model of computation which we believe will help in the creation of new robust distributed algorithms. In brief, the only operation allowed is a *symmetric balancing operation* by which each node of the network adjusts its state based on the state of its (out-)neighborhood. The generic algorithm is the following: all points iterate this operation asynchronously, until the network reaches a stable state. In the remainder of the chapter, we present a number of algorithms for the balancing model.

In a *dynamic* graph, nodes/edges may leave or join the graph over time; these changes represent hardware failures, rebooting, and topology changes. We use $\overline{\mathcal{G}} = (\overline{V}, \overline{E})$ to represent the union of all vertices and edges that ever belong to the graph. In this chapter, we work with undirected *dynamic* graphs unless we specify otherwise.

4.1 Realistic Assumptions

It is usually convenient to assume that a distributed algorithm will execute on a network with fixed topology, and that the network is reliable, and this is reflected by the model of Chapter 2. However, in practice, that model is too simple. Here are some of the most common differences between the model and real life.

1. **Synchrony**: In practice, networks are rarely completely synchronous. Rather, the order in which nodes activate (check incoming buffers, perform a computation, send out messages) is impossible to determine a priori. An algorithm is called *fully asynchronous* if it does not

rely on the order in which the nodes activate (with the caveat that each node must activate infinitely many times). One intermediate possibility between synchrony and full asynchrony is *partial synchrony.* In partial synchrony, we allow the *s-archimedean assumption:* for some fixed constant s, any node can activate at most s times in between successive activations of any other node.

2. **Node Failures**: In the course of executing a distributed algorithm, the individual processors may operate incorrectly. Here is a hierarchy of common fault models from [99, p. 430], in increasing order of disruptiveness: initially dead processes; crash failures; omission errors; and *Byzantine* failures (arbitrary erroneous behaviour). Other types of failures such as timing errors and restart errors are also possible.

3. **Communication**: An error may occur in the course of transmitting a message between two nodes. The error may be one of the following types: the message may be lost; the message may be corrupted; the message may be delivered multiple times; messages may be received in a different order than the order in which they were sent; or messages may be delayed in the network.

4. **Unique Identifiers**: As mentioned in Section 2.2, networks can be named or anonymous. We can further break down anonymous networks into two classes. First, we may have *indirect addressing* [99, p. 69] (also called *local orientation* [45]), where each node has a consistent labeling of its neighbors. Alternatively, the network may be *ambiguous* [45]; that is to say, each node is aware of its neighbors only as an unordered multiset.

5. **Changing Topology**: The topology of the network may change while the algorithm is running.

Generally speaking, acknowledging these factors becomes more important as the network becomes physically larger: a multi-threaded computer is more reliable and synchronous than a local-area network, which is in turn more reliable than the Internet. In sensor networks, algorithms that work under these realistic assumptions are a necessity; it is also important that sensor network algorithms have low communication complexity, since the network lifetime decreases with each message sent.

4.2 Addressing Reality

Let us briefly discuss some of the issues mentioned in Section 4.1 before addressing the others in depth.

If there are node failures, we essentially need either randomization or partial/full synchrony. For example, deterministic leader election is impossible in a fully asynchronous setting, even if only one node fails [81]. In a complete network, asynchronous randomized algorithms can tolerate $\lceil n/2 \rceil - 1$ crash failures, or $\lceil n/3 \rceil - 1$ Byzantine failures. In partial or full synchrony, one can tolerate $\lceil n/3 \rceil - 1$ Byzantine failures deterministically, $n-1$ crash failures deterministically, and $n-1$ Byzantine failures probabilistically by using message authentication. We refer the reader to [99, §13.2.3] for more details about all of these problems.

Faulty communication channels can be improved with *handshaking* protocols [79, Ch. 22]. These protocols add "conversation IDs" to messages and use extra "acknowledgement" messages to improve reliability. The most important is the "5-packet handshake," which sends 4 extra messages for each message to be delivered, but ensures that messages are not lost or received out of order. Probabilistic message authentication is useful to detect messages that get corrupted or mistransmitted.

When randomness is available, most anonymous networks can be treated as named networks. If the graph is not dynamic, then we can elect a leader, construct a spanning tree, and then preorder the vertices to get unique node names. In a dynamic graph other approaches are possible, and we suggest one in Section 5.6.

4.2.1 Synchronizers

It is convenient, for the purposes of designing and analyzing algorithms, to model networks as being completely synchronous, even though this is usually unrealistic. A *synchronizer* allows one to convert an algorithm for a synchronous network into an equivalent algorithm for an asynchronous network.

Here is a simple synchronizer, called the α synchronizer in [7]. It enforces a local consistency constraint in the network, ensuring that no node runs too much faster than its neighbor. When a node v activates, depending on the state of v's neighbors, v may either perform a step of the synchronous algorithm, or it may do nothing.

For each node v, let a_v denote the number of steps of the synchronous algorithm that v has performed. Initially, $a_v = 0$ at each node. Whenever a node increases its counter a_v, it sends the new value a_v to each of its neighbors. Thus, each node v knows the values $\{a_w | w \in \Gamma(v)\}$. The consistency constraint mentioned above is that $a_w \geq a_v$ for all $w \in \Gamma(v)$. To reiterate, if this consistency constraint is met when v activates then v will 1) perform a step of the synchronous algorithm, 2) increase a_v, and 3) inform its neighbors of the new value a_v; otherwise v does nothing. Another way of looking at the consistency constraint is that $|a_v - a_w| \leq 1$ must be maintained whenever v and w are adjacent.

The main drawback of the α synchronizer is that it requires extra messages in order to communicate the values a_v; in total, $2m$ messages are sent in each round. Another simple synchronizer β discussed in [7] requires only $O(n)$ extra messages per phase, but slows down the speed of the algorithm by a factor of $\mathsf{Diam}(\mathcal{G})$. The γ synchronizer of [7] allows other time-communication tradeoffs between the α and β synchronizers. A near-optimal synchronizer for event-driven algorithms is introduced in [16] and adapted to a special class of dynamic networks in [15]. It requires only a polylog(n) increase in time complexity, message complexity, and storage at each node.

Tel sums up many general results about synchronizers in [99, p. 470]. Namely, we cannot have a 1-crash-robust deterministic synchronizer that works on fully asynchronous networks. However, with the archimedean assumption, randomization, or *bounded-delay networks with clocks,* synchronization is possible.

4.2.2 Clustering

One basic strategy to get around the impermanence of nodes in a dynamic graph is to group them together, identifying all nodes in a group as being part of the same object. In this way, even if an individual node fails, the state of the network as a whole is not disturbed. Such a strategy is explicitly formulated in [83], and the given clustering algorithm takes $O(\log n)$ rounds. The fact that this algorithm constructs a kind of partition on the nodes of $\mathcal{G}$ seem to indicate that fast partition/clustering algorithms like those discussed in Section 2.7 might be useful.

It is also possible to give clusters well-defined read and write primitives, so as to build a stable virtual graph out of the clusters. An implementation of such a structure is discussed in [19][20]. This algorithm assumes that the network is embedded in the plane. Each "persistent node" consists of a dartboard-like structure with three rings of vertices. The innermost ring is where data is read or written, with the same information replicated at each vertex. The second ring holds read-only data and routing information. The outermost ring knows only the identity of the node. This structure permits atomic reads and writes, and is robust against failures of individual vertices. We also note a similar approach in [36][35].

4.3 The Self-Stabilizing Model

Assume that we have a fixed network with reliable edges. A distributed algorithm is called *self-stabilizing* if, starting from any possible initial (corrupted) state of the network, the algorithm eventually brings the network to a "correct" state. The correct state usually corresponds to solving a particular problem.

The oldest (and perhaps most beautiful) example of a self-stabilizing algorithm comes from Dijkstra in [33]. He considers a ring of finite-state processors; the purpose of his algorithm is achieve *mutual exclusion* by passing a virtual token around the ring. There should be exactly one token, and each node should hold the token infinitely often. Since a self-stabilizing algorithm may be started in an arbitrary network state, there may initially be multiple tokens; however, after $O(n^2)$ steps, there will be exactly one token circling around the ring.

The precise algorithm is as follows. Let $K > n$. Each processor v_i has a state σ_i, and each state is an integer between 0 and $K - 1$. Each processor v_i can read its own state and the state of its predecessor v_{i-1}; the predecessor of v_1 is v_n. Each node runs a simple process which is shown in Algorithm 4.1. A node's state and its predecessor's state determine whether that node holds a token, as commented on lines 2 and 7. We refer the reader to Dijsktra's paper [33] or Tel's exposition [99, §17.1.3] for details and proof of the time bound.

Algorithm 4.1 Dijkstra's self-stabilizing mutual exclusion algorithm for a ring.

```
1:  procedure MUTUAL-EXCLUSION-FOR-v_1
2:      if σ_1 = σ_n then                          ▷ Right now, v_1 holds a token
3:          σ_1 ← (σ_n + 1)  mod K
4:      end if
5:  end procedure
6:  procedure MUTUAL-EXCLUSION-FOR-OTHER-NODES     ▷ Node v_i runs this algorithm
7:      if σ_i ≠ σ_{i-1} then                      ▷ Right now, v_i holds a token
8:          σ_i ← σ_{i-1}
9:      end if
10: end procedure
```

Whereas the previous sections discuss rather complicated ways to accommodate dynamic networks, self-stabilizing algorithms take as simple an approach as possible to fault-tolerance. The self-synchronized viewpoint is that, while arbitrary failures may occur in the network, useful computation can take place ones these failures stop.

Other self-stabilizing algorithms from [34] include the following. There exists a randomized self-stabilizing algorithm that, in expected $O(\mathsf{Diam})$ cycles, elects a leader in a symmetric network and gives all nodes unique names; there are self-stabilizing synchronizers; by using *stabilizers,* any distributed algorithm can be turned into a self-stabilizing one; a Turing machine can be simulated by a path graph of identical processors with finite state, such that the algorithm run by those processors is self-stabilizing.

In [34, p. 110], Dolev discusses a concept he calls *superstabilization*. An algorithm is called superstabilizing if it can react quickly to point failures. For example, he shows a self-stabilizing algorithm for colouring a path graph that takes $O(n)$ time to stabilize, but can correct the state of the graph after a single node failure within $O(1)$ time. An algorithm with small superstabilization

time is appealing for a large network, since multiple point failures would be fixed independently and isolated from one another.

4.4 Behaviorally Symmetric Algorithms

When an algorithm is run on an anonymous network, it is assumed that each node *initially* runs the same set of instructions. As we have discussed, leader election and many other problems cannot be computed deterministically in the anonymous model, but with randomization an anonymous network is as powerful as a named network. Anonymous networks demonstrate *initial symmetry.*

We want to discuss algorithms with a stronger form of symmetry, which we call *behavioral symmetry,* although it does not seem to be a formalizable quality. In a behaviorally symmetric algorithm, all nodes should really perform the same computation at all times.

For example, consider using Algorithm 2.3 (tree-based aggregation) to count the number of nodes in a network. The computation is centralized at the root, and the tree defines a structure on the edges that breaks the symmetry of the network. This highly un-symmetric algorithm is prone to failure on a dynamic graph: if a node v fails, then all of its descendants are unaccounted for, and there is no way to recover.

In order to get around this problem, we propose looking at algorithms which do not exhibit this reliance on delicate structures, and where no nodes are more "important" than any others. In effect, we will look at algorithms in which each node is really *doing the same thing* at each step. Thus behavioral symmetry is stronger than the initial symmetry of anonymous networks. Behavioral symmetry is related to the term *decentralized* as used in [57] and [68]. The resulting algorithms are reminiscent of physical processes, such as objects connected by springs which "compute" their positions based on balancing the local forces upon them.

4.5 A Balancing Model of Computation

In this section, we discuss a model for behaviorally symmetric distributed computation on dynamic graphs. The model is based on the following three principles:

1. **Global symmetry.** We can think of any distributed algorithm as an operation that "activates" at each node repeatedly. Its input is the current state of that node, and the states of its neighbors; the output is the new state of that node. The first principle of the model is that *at all times, each node performs the same operation.*

2. **Local symmetry.** As exemplified in Section 4.4, we wish to avoid algorithms which rely on unstable structures built within the graph. The algorithm should be largely insensitive to the state of the network at any point in time, and always work with whatever neighbors are available. For this reason, the second part of the model is that *the operation acts symmetrically on the neighbors of the activated node.*

3. **Steady state convergence.** The third principle relates to termination of the algorithm. Namely, *the network continually approaches a steady state, where all of the vertices are invariant under the operation.* The fixed point thus marks the termination of the algorithm.

We can succinctly combine all three of these ideas by describing our algorithms as performing some kind of **balancing.** Each node just ensures that a local balancing rule is satisfied when it activates; we are done when the whole graph is in equilibrium. In the dynamic setting, local failures cause a temporary loss of balance, and the operation restores balance in affected areas of the network.

Let us comment a bit on the first principle. In the terminology of Tel, we use the [99, p. 524] *read-all, state* model of communication, and the algorithm must be [99, p. 526] *uniform.* An argument for the insufficiency of the message-passing model in self-stabilizing algorithms appears in [99, §17.1.2], and the same basic idea applies to our model. Note that the traditional message-passing model can be simulated in the state-reading model by having each node store a message buffer for each neighbor. See Appendix A for more details.

The second principle requires some clarification. We want to say that each node's operation is a symmetric function of its neighbors. Equivalently, the network is *ambiguous,* in the terminology of [45]. We actually propose something a bit stronger, and it is motivated by the fact that two nodes of different degrees cannot be identical according to the "symmetric function" viewpoint, since they have different arity. When v activates, let its state σ determine an embedding function E_σ that maps its' neighbors states to some domain D, a combining operator $\star_\sigma : D \times D \to D$, and a selection function S_σ that maps an element of D back to a node state. Here is how we run a single step of the algorithm: E_σ is applied to each node in $\Gamma(v)$, the resulting values are combined arbitrarily using $\star_\sigma$, and S_σ is applied to the final result to determine the new state of v. If we insist that $\star_\sigma$ is commutative and associative, then we claim that the network is ambiguous. Further, using this formality, we get nodes of different degrees to perform the "same" computation. All algorithms we present here satisfy the additional restriction that D is a set of $O(\log n)$-bit strings, and the function $\star_\sigma$ is computable in $O(\log n)$ time and space. In Appendix A, we discuss what can be done when D is finite.

How does the third principle compare to the classical notion of explicit termination? So long as each node can determine its own invariance, termination can be globally determined in our model,

as follows. When a node's state changes, it broadcasts a "not done" message over the network, with maximum message lifetime set to Diam. Any node can determine that the algorithm has terminated when it hears no such messages for Diam steps. Note that the algorithm "broadcast to all nodes" fits our model, with corresponding symmetric operation "if you have received a message from any neighbor since last activating, then forward it to all neighbors."

Suppose that each node v_i stores a value f_i. Here are some example operations in our symmetric toolkit that can be applied when node v_i is activated:

- Select a random neighbor of v_i.
- Compute the indegree or outdegree of v_i.
- Compute the maximum/minimum of f on the neighbors of v_i.
- Compute the average of f over the neighbors of v_i.

Note that the results of these functions are independent of the order in which the neighbors are processed.

The usefulness of a balancing algorithm will depend on its rate of convergence and its resilience to faults. First, in a static graph, running the operation on all nodes should get us to a fixed point quickly (for our purposes, in poly(n) time). Second, if we start with a graph that is in equilibrium and then perturb it slightly by adding/removing an edge/node, the resulting graph should be "near" equilibrium, in that it will rebalance itself quickly.

4.6 Balancing Algorithms for Basic Problems

In this section, we give algorithms for several basic problems that fit the model. Each algorithm will be based on one (or a few) balancing operations performed asynchronously at all nodes of the network.

4.6.1 Bipartiteness

This algorithm determines whether a connected, undirected graph is bipartite, by attempting to compute a 2-coloring. Each node stores a color B, W or X, with all nodes initialized to X. We assume that there is a single initiator, which colors itself W. Here is the symmetric operation: at each step, any node with a W neighbor colors itself B, and vice-versa; a node with both a W and a B neighbor should broadcast "not bipartite" throughout the graph. Alternatively, in terms of a

balancing operation that takes place on edges, each edge balances itself so that it has one black endpoint and one white endpoint.

If the graph is bipartite then this generates a 2-coloring; otherwise, the nonbipartiteness message will be broadcast throughout the graph. This runs in $O(\mathsf{Diam})$ time. In dynamic graphs, the above algorithm has one-sided error: it correctly returns "bipartite" if $\overline{\mathcal{G}}$ is bipartite, but it may misclassify a nonbipartite graph if there are enough failures.

4.6.2 Shortest Paths

There is a symmetric local operation which allows us to find the shortest paths from all nodes to a fixed vertex t in a directed graph. Each node labels itself according to its distance from t. The corresponding operation works as follows: the node t has a fixed distance label $\ell(t)$ equal to 0. When any other node v activates, it sets its label to 1 more than the minimum of its outneighbors' labels:

$$\ell(v) \leftarrow 1 + \min_{u \in \Gamma(v)} \ell(u).$$

Consider a static graph. The label of v will stabilize at the minimum distance from v to t, provided that a path exists from v to t. The label of a vertex with no path to t will increase to without bound, but by capping labels at n we still get a fixed point, since $\mathsf{Diam} \leq n-1$. Convergence of this algorithm to the steady state takes $O(\mathsf{Diam})$ rounds. To route information to t using these labels, whenever a vertex activates, it sends any packets it holds to a neighbor with smaller label.

4.6.3 Clustering With Leaders

Next, in our directed network, consider replacing the single vertex t with multiple vertices T; again, each $t \in T$ has a fixed label 0, and all other vertices go to 1 more than their minimum neighboring label when they activate. This allows us to route information to a set of sinks in a network: for example, if the only permanent storage in a sensor network is held by a set of special nodes, then we could use this scheme to route important information to the nearest storage center. We note that this algorithm relates to the idea of Directed Diffusion from [61], in that a local gradient is used to direct information to the right place.

If we label each node with the member of T to which it is closest, we get a clustering algorithm. We assume that each element of T has a distinct cluster-label. Every node stores both a distance and a cluster-label. The corresponding operation is: identify the neighbor with minimal distance d, copy its cluster-label, and set your own distance to $d+1$. Without labels, individual nodes are not aware of their cluster leader, but the routing algorithm still maintains an implicit clustering which

routes packets to cluster leaders.

4.6.4 Maximum s-t Flow

Recall the definition of the maximum s-t flow problem from Section 2.6.2. One algorithm for determining the maximum flow is to repeatedly augment residual paths of shortest length.

We can adapt this to our balancing model as follows. We keep track of each node's shortest distance to t in the residual graph, using the algorithm of Section 4.6.2. The augmenting operation is: if some outneighbor of v_i in the residual graph has a smaller distance label to t than does v_i, and the incoming flow exceeds the outgoing flow, then augment the edge going to that neighbor as much as possible, and update the residual graph. Initially, we have an infinite excess of flow at s.

In fact, this presents an alternative (and more intuitive, we believe) description of the Goldberg-Tarjan [54] maximum flow algorithm, which has $O(n^2)$ time complexity. It can be made to work in a dynamic graph [53], by adding a third symmetric balancing operation which ensures that broken paths are removed.

4.6.5 Aggregation in Sensor Networks

As mentioned in Section 4.4, tree-based aggregation algorithms are unsuitable for dynamic networks. However, there are other possibilities. First, in the special case where our aggregate is an *infimum* [99, §6.1.5] or *semi-lattice* [84] function — a pairwise symmetric, associative, *idempotent* operation applied to all nodes — then repeated broadcast by every node can be used, which fits our model. This is similar to the leader election of Algorithm 2.1, for which the target infimum function is the minimum of all node labels.

Recently, an algorithm called *gossiping* has applied local balancing to the problem of computing an average. In terms of an edge-balancing operation, the idea [67][24] is that each edge (u, v) repeatedly replaces the values $f(u)$ and $f(v)$ by $\frac{f(u)+f(v)}{2}$. Repeating this operation at all nodes, we see that the value of every node converges to the mean. This idea is applied to larger groups of nodes in [28].

The authors of [84] give a neat idea for approximately computing n in a dynamic network of unknown size, adapting an older idea of [43]. Each node stores a k-bit boolean vector, where 2^k is an upper bound on the network size. Each node initially sets a single bit to 1, setting the ith bit with probability 2^{-i}, and setting no bits with probability 2^{-k}. By flooding (repeated broadcast), each node can robustly determine all bits chosen. Let *mex* denote the lowest index of all unset bits, then $1.3 \cdot 2^{mex}$ gives a good population estimate. This idea is extended in [84] to computing sums,

statistical moments, medians, and modes.

4.6.6 Synchronizer

The α synchronizer of Section 4.2.1 can be implemented in our model. Suppose we have an algorithm which we wish to run in a synchronized manner, with state space $\mathcal{S}$, initial state $S_0 \in \mathcal{S}$, and balancing operation $\mathcal{B}$. In Algorithm 4.2 we show how to run that algorithm on an asynchronous network, in a way which is consistent with our model. Essentially, the balancing operation for a node v is, "If $a_w \geq a_v$ for all $a_w \in \Gamma(a_v)$, then increase a_v and execute a step of the synchronous algorithm." Note that each node keeps two copies of its state, a "current" one and an "old" one.

Algorithm 4.2 Synchronization in the balancing model.

```
1: Let State ← S_0
2: Let OldState ← S_0
3: Let a_me ← 0
4: procedure BALANCING-SYNCHRONIZER
5:     if a_w ≥ a_me for all a_w ∈ Γ(me), then
6:         OldState ← State
7:         for each neighbor w_i ∈ Γ(me), with i ← 1 to δ(me), do
8:             if a_{w_i} = a_me then
9:                 Input_i ← State(w_i)
10:            else
11:                Input_i ← OldState(w_i)
12:            end if
13:        end for
14:        State ← B(State, Input_1, ..., Input_{δ(me)})
15:        a_me ← a_me + 1
16:    end if
17: end procedure
```

There is a practical problem in using this synchronizer on a network where nodes may crash or reboot: when a node v rejoins, the counter a_v will be 0, and thus the neighbors of v will not be able to do anything until a_v catches up to their counters. Alternatively, the issue is that the constraint $\forall (u, v) \in E : |a_u - a_w| < 1$ cannot be consistently enforced if the edge set of $\mathcal{G}$ changes. Tel's comment at the end of Section 4.2.1 actually implies this problem: no synchronizer can exist in a faulty network. However, if the algorithm to be synchronized is such that the details of "catching up" can be worked out, then this synchronizer is okay.

4.7 Agent Algorithms

An *agent* is a stateful entity that inhabits the nodes of a network. An agent traverses the network by following the graph's edges, as if it were passed around by messages. Agents are useful in the

balancing model as a simple conceptual tool. By having the agent traverse the entire network, one can simulate certain kinds of sequential computation.

One problem with any agent algorithm is that, if the node containing the agent dies, then the algorithm stops. Ghosh also notes this problem in the context of self-stabilizing algorithms [52]. A basic solution is to make the entire network aware of the agent, as follows. The agent can broadcast an "alive" message to the network at every step, with message lifetime n. If no such messages are heard for n steps, a new agent can be created and the algorithm re-started.

Creating an agent generally requires a leader, or else multiple agents would be created by multiple nodes. It is impossible [81] to deterministically elect a leader in a faulty network. We thus assume that the leader is chosen externally; even though this implies some stability in the network, we only require local stability at one node (the leader), whereas the classical model requires global stability. Note that the "random naming" trick gives a Monte Carlo algorithm for leader election; it would be interesting to come up with a Las Vegas election algorithm for the balancing model.

4.7.1 Census

We can use the shortest-paths algorithm of Section 4.6.2 to get a *census* algorithm. By this, we mean that the agent will eventually visit every node of the network. Each node keeps a boolean flag indicating whether it has been visited or not; thus, the agent can count the number of distinct nodes that it has visited, or compute an aggregate function of the nodes (as defined in Section 2.4). This algorithm applies to all strongly connected directed graphs.

The algorithm works as follows. Let T denote the set of unvisited nodes. Each node contains a flag indicating whether or not it belongs to T; initially, we set $T = V$. Concurrently with the agent's traversal, we run the T-shortest path and T-clustering algorithms of Sections 4.6.2 and 4.6.3. Here is the algorithm: the agent removes the leader of its current cluster from T, waits until a different cluster subsumes its position, travels along a shortest path to the leader of that cluster, and repeats.

Given n, it is easy to determine when this algorithm is done, since we can attach a counter c to the agent which counts how many nodes have been removed from T. Even if n is unknown, then we can use the agent to determine n, as the following observation shows.

Claim 4.7.1. *In a strongly connected static graph, the maximum T-distance label of any node is at most $|V - T|$.*

Proof. The T-distance label $\ell(v)$ of a node v represents the length of the shortest path from $\ell(v)$ to T. Now, all of the nodes on this shortest path must not be in T or else the path to such a node

would provide a shorter T-path than the label indicates. Thus there are at least $\ell(v)$ nodes not in T. □

Thus, when the distance label of the agent's current position exceeds c, we know that T is empty, and we terminate with the census count $n = c$. On a dynamic graph, it is possible for the distance labels to be incorrect; a good heuristic would be to pick some constant $\kappa > 1$ and terminate when the agent's position has distance label more than κc.

If we have already computed n, we also have a census algorithm for counting the number of nodes with a specific property. Namely, instead of initializing $T = V$ at the beginning of the algorithm, set T to be the set of nodes with this property. We run the same greedy walk, and terminate when the agent's position has T-distance κn.

The time complexity of the census algorithm can be broken down into two parts: the time that the agent spends traveling, plus the time that the agent waits for re-clustering to finish. The re-clustering time is at most n more than the travel time, since any wait period of k rounds (except the last) is followed by k rounds of agent movement. It is easy to see that the agent travels $O(n^2)$ distance, since it explores n nodes, each taking at most n steps to get to. We will show in Chapter 5 that the agent travels $O(n \log n)$ distance on an undirected graph. Consequently, the whole algorithm takes $O(n^2)$ or $O(n \log n)$ time, according to whether the graph is undirected or directed.

The need for node IDs is somewhat unsatisfactory, since our model was meant to be anonymous and even ambiguous. Node IDs can be removed without affecting the correctness of the algorithm. In the synchronous setting, the time complexity remains unchanged provided that, after removing a node from T, the agent waits until that node's distance label stabilizes for 3 rounds before moving to the next clusterhead. In an asynchronous setting we can apply a synchronizer to the network as described in Section 4.6.6 to get the same result.

4.7.2 Edge-Biconnectivity from a Random Walk

A random walk on a directed graph can be implemented in our model with the following operation: if a node contains the walker upon activation, it is considered unbalanced and it sends the walker to a random neighbor.

Using a random walk, we can determine the biconnected components of a connected undirected graph $\mathcal{G}$. Recall that an edge $e \in E$ is said to be a *bridge* of $\mathcal{G}$ if the removal of e causes $\mathcal{G}$ to be disconnected. This algorithm determines all bridges of $\mathcal{G}$.

Fix an orientation on each edge. Let each edge (v_j, v_k) have a counter c_{jk} which is initially set to 0. The idea is that we run a random walk on the graph; each time the walker traverses an edge, we

increase or decrease that edge's counter, according to whether it agrees with that edge's orientation.

Claim 4.7.2. *If (v_j, v_k) is a bridge of $\mathcal{G}$, then c_{jk} never exceeds 1 in absolute value.*

Proof. From the definition of a bridge, V can be partitioned into sets V_1 and V_2 such that $v_j \in V_1, v_k \in V_2$ and (v_j, v_k) is the only edge between V_1 and V_2. Successive traversals of (v_j, v_k) go in opposite directions, since that edge takes us alternately between V_1 and V_2. Consequently the values of c_{jk} as we traverse the edge are either $0, -1, 0, -1, \ldots$ or $0, 1, 0, 1, \ldots$. □

Claim 4.7.3. *Assume that $\mathcal{G}$ is non-dynamic. If (v_j, v_k) is not a bridge of $\mathcal{G}$, then the expected time before c_{jk} exceeds 1 in absolute value is $O(mn)$ steps.*

Proof. Let us construct a new graph. It has $3n+1$ vertices: three labeled v_i^{-1}, v_i^0, v_i^1 for each $v_i \in V$, plus the special vertex EXCEEDED. The idea is that v_i^r corresponds to the event that $c_{jk} = r$ and the walker is at node v_i. The node EXCEEDED corresponds to increasing the counter to ± 2. Thus, this new graph has $3m + 1$ undirected edges in total:

$$(v_i^r, v_{i'}^r) \text{ for each } r \in \{-1, 0, 1\} \text{ and each } (v_i, v_{i'}) \in E - \{(v_j, v_k)\}$$

as well as

$$(v_j^{-1}, v_k^0), (v_j^0, v_k^1), (v_j^1, \text{EXCEEDED}), (\text{EXCEEDED}, v_k^{-1}).$$

It is straightforward to show that a random walk on the new graph corresponds to the original process on the old graph.

Since (v_j, v_k) is not a bridge, we can reach any v_i^r from v_j^0: if $r \neq 0$ traverse a cycle containing (v_j, v_k) to set c_{jk} correctly; then, walk to v_i without using (v_j, v_k). Thus, the new graph is connected. By applying the hitting time bound for an undirected graph [82, p. 137], we expect to reach EXCEEDED in at most $2(3m + 1)(3n) = O(mn)$ steps. □

For the algorithm, we put counters on all of the edges, and update one each time the walker moves. If we wait for $O(cmn \log n)$ steps, then with probability $1 - n^{-c}$, all bridges of the graph will have been identified. If the graph contains no bridges, then it is biconnected; otherwise, we can straightforwardly determine the biconnected components by broadcasting component labels along the nonbridge edges.

Like the 2-coloring algorithm, on a dynamic graph, this algorithm has one-sided error: if e is determined to not be a bridge, then some cycle in $\overline{E}$ contains e.

4.7.3 Edge k-Connectivity

The algorithm of the previous section probabilistically identifies cuts of value 1 in the graph $\mathcal{G}$, making use of the fact that $|c_e| < 1$ for any bridge e of $\mathcal{G}$. Similarly, for *any* cut $(S, V - S)$ of $\mathcal{G}$ the values of the edges, properly oriented, have a sum which stays in $\{-1, 0, 1\}$. Precisely, let $e_1, \ldots, e_k$, be those edges with one endpoint in S and one endpoint in $V - S$. Define d_i to be -1 (resp. 1) if the orientation of e_i points from S to $V - S$ (resp. $V - S$ to S.) Then

$$\sum_{i=1}^{k} c_{e_i} d_i \in \{-1, 0, 1\}$$

by the same argument as in Claim 4.7.2.

For cuts of size 2, this algorithm seems feasible: we run a random walker on the network for a while, and in $O(mn)$ rounds we expect that edges whose counters remain equal up to ± 1 form a cut of value 2. It is then straightforward to determine the 3-edge-connected components of $\mathcal{G}$. However, it is not clear how to distributively check to see which pairs of edges are correlated.

4.8 Discussion

There are several other algorithms in the literature that exhibit the properties of our model. These include algorithms for end-to-end communication [2], approximate multicommodity flows [13][14], eigenvalue computation [68] and approximate linear programming [18].

The original motivation for the balancing model was sensor networks. However, the balancing model incurs a high communication complexity of $O(m)$ messages per time step. This makes it impractical for sensor networks, where every message sent shortens the network lifetime.

How does our new model of computation relate to the self-stabilizing model? Both models involve a model of state-based, read-all communication, and both models move toward a steady state. The symmetry of our model is not found in the self-stabilizing model, although uniform algorithms satisfy the weaker symmetry property that all nodes "run the same program."

The self-stabilizing model works with *any* kinds of failures, so long as, at some point in time, no more faults occur in the network. The reason that this unrealistic assumption is useful is that, if faults occur sufficiently infrequently, the system will usually be in a correct state. However, if the mean time between faults in the network is less than the stabilization time of the algorithm, then a self-stabilizing algorithm will never be correct. In comparison, some of the balancing algorithms (such as the agent-based ones) will operate correctly even when the network is never completely

stable for more than a single operation, so long as the failures don't disrupt the network topology too much.

The *cellular automata* [104] model of computation is somewhat related to our model. In it, each node has a finite number of possible states, and computation proceeds synchronously and uniformly. The topology of cellular automata seems to always be an infinite graph, and so this model is not of much practical use to a finite computer network.

We also point out the *input/output automaton* model of computation, defined in [77] and developed in [65] [50]. This model is a convenient way to express certain distributed algorithms [35] and proofs [80]. There are no symmetry constraints in this model.

Finally, let us point out that it is difficult to state, exactly, how robust is the balancing model. We have a general expectation that balancing algorithms will not be wrecked by single point failures. However, given enough properly placed node or edge failures, the graph becomes disconnected, and most algorithms cannot work. Even without total disconnection, a topology that changes quickly enough (e.g. $\mathcal{G}$ becomes a new random connected subgraph of K_n in each round) causes even the simplest algorithms to fail. In Chapter 6 we attempt to address some of these concerns as they apply to computation of discrete harmonic functions.

Chapter 5

The Greedy Tourist Algorithm

In this section, we flesh out the census algorithm described in Section 4.7.1. The main results are $\Theta(n \log n)$ upper and lower bounds on the number of steps required.

5.1 The Problem

We have just landed at the airport of a foreign island, and wish to visit every city. The catch is that we didn't buy a map, and we do not understand the road signs. We *do* recognize the alphabet, so we keep a map of the cities we've been to and the destinations of the outgoing roads from each. How can we visit all of the cities efficiently?

We model the cities and the roads as a graph $\mathcal{G}$ with n vertices. We call the entity which traverses the graph a *walker*. In each *step*, the walker moves from its current position v to an adjacent vertex $v' \in \Gamma(v)$. The walker must always determine its next step from incomplete, local information: the walker knows $\Gamma(v)$ for each node v that it has already visited, but the remainder of the graph is unknown to it. We require that $\mathcal{G}$ be strongly connected, so that we never get "stuck" without being able to visit more vertices.

5.2 The Algorithm

This chapter is concerned with a particular algorithm that visits all of the cities in few steps. Algorithm 5.1 depicts the *Greedy Tourist* (GT) algorithm for this task. It is essentially "while there are unvisited vertices, take a shortest path to the closest unvisited vertex."

Algorithm 5.1 The Greedy Tourist Algorithm, from the walker's perspective.

```
1: procedure WALKER-GT
2:     Let v_0 denote your initial position
3:     Let pos always denote your current position
4:     Let G be a graph initially containing v_0, Γ(v_0), and the outgoing edges of v_0
5:     visited ← {v_0}
6:     while visited ≠ V do
7:         Select dest ∈ (V − visited) such that d_G(pos, dest) is minimized
8:         Follow any shortest path from pos to dest (thus pos ← dest)
9:         Add Γ(pos) and the outgoing edges of pos to G
10:        visited = visited ∪ {pos}
11:    end while
12: end procedure
```

On an undirected graph, GT is not optimal: a simple depth-first search strategy will take at most $2n$ steps, whereas GT may incur $\Theta(n \log n)$ steps, as we show in Theorem 5.4.4. However, if (as in Chapter 4) we think of the graph as representing a faulty network, a depth-first search is unsuitable: a given node may become disconnected from its parent in the DFS tree during the traversal, causing the algorithm to fail. The greedy tourist algorithm is practical for traversing an entire faulty network, and we thus aim to prove its efficiency.

Note that line 7 of GT is non-deterministic, in that the walker is allowed to break ties arbitrarily. The following definition captures the allowable forms of nondeterminism.

Definition 5.2.1. *Fix a graph $\mathcal{G}$. The permutation $T = (t_1, t_2, \ldots, t_n)$ of V is said to be a* greedy traversal *of $\mathcal{G}$ if $d_{\mathcal{G}}(t_i, t_{i+1}) \leq d_{\mathcal{G}}(t_i, t_j)$ whenever $1 \leq i < j \leq n$.*

We say that a node is *explored* when the walker moves to that node for the first time.

Theorem 5.2.2. *Consider a walker that executes the GT algorithm starting from t_1. It is possible for the walker to explore the nodes in the order $t_1, t_2, \ldots$ if and only if $(t_1, \ldots, t_n)$ is a greedy traversal.*

Proof. A greedy traversal is easily seen to be the order of exploration of a *global* greedy search; namely for each i, we have that

$$d_{\mathcal{G}}(t_i, t_{i+1}) = \min_{j>i} d_{\mathcal{G}}(t_i, t_j).$$

To complete the theorem, we show that the walker's local greedy strategy is actually equivalent to a global greedy strategy. In other words, the closest unexplored node that the walker knows about is also the closest node that exists.

Claim 5.2.3. *If the walker has not yet visited all nodes, the globally closest unexplored node to the walker has at least one explored in-neighbor.*

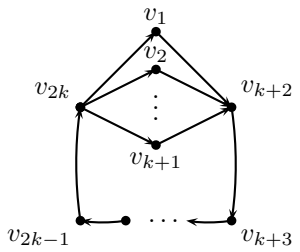


Figure 5-1: A graph taking $\Omega(n^2)$ steps to visit every node.

Proof. Suppose otherwise, that the walker is at *pos* and that the node *dest* minimizes $d_{\mathcal{G}}(pos, dest)$, but $\Gamma^{-1}(dest)$ is completely unexplored. Let v be the node preceding *dest* on a shortest path from *pos* to *dest*; since $v \in \Gamma^{-1}(dest)$, it must be that v is unexplored. We then have that $d_{\mathcal{G}}(pos, v) < d_{\mathcal{G}}(pos, dest)$, contradicting the minimality of *dest*. □

As a result of this claim, the globally closest node is always known to the walker, completing the proof of the theorem. □

Definition 5.2.4. *For a greedy traversal T of a graph $\mathcal{G}$, define its* cost $C(T)$ *by*

$$C(T) = \sum_{i=1}^{n-1} d_{\mathcal{G}}(t_i, t_{i+1}). \tag{5.1}$$

Thus, the cost measures the total number of edges that the walker must traverse before it has visited every vertex. In the remainder of this chapter, we will be concerned with bounding the cost in terms of n. Since the number of steps between successive explorations is at most $Diam(G) < n$, and there are $n-1$ nodes other than t_1 to explore, we know that $C(T) < nDiam(G) < n^2$. This bound is tight in the directed case, as the following theorem shows.

Claim 5.2.5. *Let $n = 2k$. The directed graph shown in Figure 5.2 takes at least $n^2/4$ steps to visit every node, no matter where you start, and no matter what strategy is used.*

Proof. Note that the nodes $v_1, v_2, \ldots, v_{k+1}$ are all at a distance k from each other. Any traversal of this graph must visit each of these $k+1$ nodes at least once. Since there must be at least k steps between successive explorations of $\{v_1, \ldots, v_{k+1}\}$, at least $k^2 = n^2/4$ steps take place in total. □

Hereafter, we concern ourselves *only with undirected graphs*. The greedy walk does much better in this setting.

5.3 An Upper Bound For Undirected Graphs

In this section, we show that $C(T) < 2n \ln n$. We introduce some terminology for convenience.

Definition 5.3.1. *Let T be a greedy traversal. Define the* leg of T from t_i *to be any shortest path from t_i to t_{i+1}.*

The proof of the upper bound is based on the following lemma which, given a low-diameter partition of the graph's vertices, bounds the number of long legs in a greedy traversal.

Lemma 5.3.2. *Let $\mathcal{G} = (V, E)$ be a graph and let $V_1, V_2, \ldots, V_k$ be a partition of V. Suppose the integer D is so large that $D \geq \mathsf{Diam}_{\mathcal{G}}(V_j)$ for each j. Then in a greedy traversal T of $\mathcal{G}$, the number of legs of length more than D is at most $k - 1$.*

Proof. Let t_i be a node from the walker's tour, and V_j be the component containing t_i. Suppose t_i is not the last node from V_j to be explored, say $t_{i'} \in V_j$ has $i' > i$. Then by Definition 5.2.1 and the small diameter of V_j,

$$d_{\mathcal{G}}(v_i, v_{i+1}) \leq d_{\mathcal{G}}(v_i, v_{i'}) \leq D.$$

Thus, for each of the k components, at most one leg leaving that component has length greater than D.

Let V_ℓ denote the component containing the last vertex to be explored. We see that no leg leaving V_ℓ has length greater than D. This completes the proof. □

Next, we show that every graph can be partitioned into a small number of subsets each having low $\mathcal{G}$-diameter. Note that this is one way to (sequentially) construct small $(t-1)$-dominating sets as defined in Section 2.6.5.

Lemma 5.3.3. *Let $\mathcal{G} = (V, E)$ be an undirected graph and let t be an integer. There exists a partition of V into at most $\lceil n/t \rceil$ components such that each component has $\mathcal{G}$-diameter at most $2t - 2$.*

Proof. First, we claim that it suffices to consider the case where $\mathcal{G}$ is a tree. This holds because the $\mathcal{G}$-diameter is non-increasing when we add an edge; precisely, if $\mathcal{H}'$ is obtained by adding one or more edges to $\mathcal{H}$, then $\mathsf{Diam}_{\mathcal{H}'}(S) \leq \mathsf{Diam}_{\mathcal{H}}(S)$. Thus, we consider a spanning tree of $\mathcal{G}$, apply the theorem to the tree to obtain the partitions, and then observe that adding back the non-tree edges does not increase the diameter of the partitions.

Let $\mathcal{T}$ denote our tree, with root r. Suppose, as a special case, that $\mathcal{T}$ has height $t - 1$ or less. Then for any two nodes $u, v \in \mathcal{T}$, we have $d_{\mathcal{T}}(u, v) \leq d_{\mathcal{T}}(u, r) + d_{\mathcal{T}}(r, v) \leq 2(t-1)$. Consequently

the trivial partition of V into a single component satisfies the requirements of the problem.

If T has height greater than $t-1$, then consider any subtree $\mathcal{S}$ of $\mathcal{T}$ with height exactly $t-1$. A similar argument shows that its vertex set $V(\mathcal{S})$ has $\mathcal{G}$-diameter at most $2(t-1)$. Since $h(\mathcal{S}) = t-1$, we have $|V(\mathcal{S})| \geq t$. Our construction is to make $V(\mathcal{S})$ into its own component, remove $\mathcal{S}$ from $\mathcal{T}$, and iteratively find more subtrees of height $t-1$ within $\mathcal{T}$ until only a single short tree is left.

Since each component except possibly the last one contains at least t vertices, we have no more than $\lceil n/t \rceil$ components in total. □

We finally obtain the upper bound by rewriting Equation (5.1) and then applying the previous two lemmas.

Theorem 5.3.4. *For any greedy traversal T of an undirected graph, $C(T) < 2n \ln n$.*

Proof. For $1 \leq i < n$, let ℓ_i denote the length of the path leaving v_i. Let λ_i denote the number of legs in T of length greater than or equal to i, for $i = 1$ to $n-1$.

It is straightforward to see that

$$C(T) = \sum_{i=1}^{n-1} \ell_i = \sum_{i=1}^{n-1} \lambda_i,$$

similar to the identity for a partition and its conjugate.

For an integer t, apply Lemma 5.3.3 to obtain a partition of V into $\lceil n/t \rceil$ subsets of $\mathcal{G}$-diameter at most $(2t-2)$, and then apply Lemma 5.3.2 with $D = (2t-2)$: as a result, we see that $\lambda_{2t-1} \leq \lceil n/t \rceil - 1 < n/t$. Also, note that $\lambda_{2t} \leq \lambda_{2t-1}$. Thus, we have

$$C(T) = \sum_{i=1}^{n-1} \lambda_i < n + n + n/2 + n/2 + \ldots \leq 2nH_{\lceil n/2 \rceil}.$$

The stated result now follows from the bound $H_k < \ln k + 0.578$, plus some checking to make sure that small cases are satisfied. □

5.4 Tightness of the Upper Bound

In order to show that the $\Theta(n \log n)$ bound for the greedy tourist algorithm is tight, we construct a family of labeled graphs called *layered rings*. The vertices of a layered ring are labeled with integers corresponding to the order of a greedy traversal.

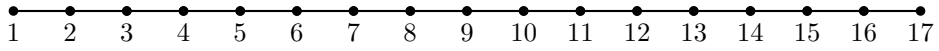


Figure 5-2: The layered ring $LR(2^4, 1)$.

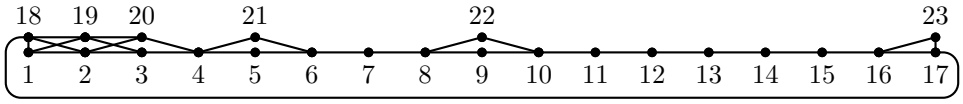


Figure 5-3: The layered ring $LR(2^4, 2)$.

We denote by $LR(2^m, k)$ the family of layered ring graphs, with parameters $m, k \in \mathbb{Z}^+$. Intuitively, $LR(2^m, k)$ can be thought of as a cycle of length $2^m + 1$, augmented with other nodes so that a greedy traversal can go around the cycle k times.

The vertices of each layered ring graph are partitioned into k *layers,* denoted $LR_i(2^m, k)$ for $i = 1, \ldots, k$. These layers are explored in ascending order. As shown in Figure 5-2, $LR(2^m, 1)$ is just a path of length 2^m. This path will be the first layer of $LR(2^m, k)$ for each k.

By adding just a few nodes, we can traverse around the ring a second time. Figure 5-3 shows $LR(2^m, 2)$ for $m = 4$. It is the path $LR(2^m, 1)$ augmented by a layer of $m + 2$ additional nodes that add legs of length $1, 1, 1, 2, 4, 8, \ldots, 2^{m-1}$ to the greedy traversal. The reader should verify that the labels in Figure 5-3 correspond to a greedy traversal.

We can iterate this augmentation process to make rings which are traversed k times, for any k. In $LR(2^m, k + 1)$, the first layer will be the 2^m-vertex path, the second layer will be new, and the remaining layers are taken from $LR(2^m, k)$ according to $LR_{i+1}(2^m, k + 1) = LR_i(2^m, k)$. Figure 5-4 shows $LR(2^m, 3)$ as an example: the first layer (1–17) is a path, the third layer (30–35) comes from $LR_2(2^m, 2)$, and the second layer (18–29) is new. The legs of $LR_3(2^m, k + 1)$ determine how many nodes we need to put in to $LR_2(2^m, k + 1)$. Precisely, for each leg of length 2^j in $LR_3(2^m, k + 1)$, we need $j + 1$ legs of length $1, 1, 2, 4, 8, \ldots, 2^{j-1}$ in $LR_2(2^m, k + 1)$. For example, the third-layer leg from 34 to 35 corresponds to the four second-layer legs from 25 to 29.

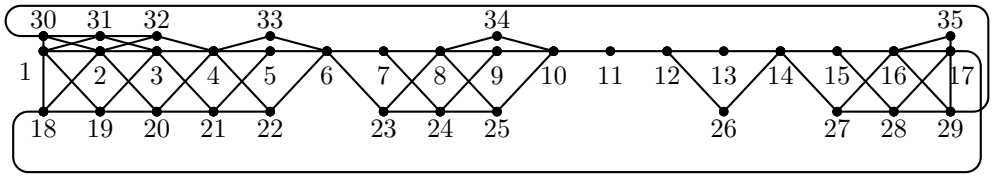


Figure 5-4: The layered ring $LR(2^4, 3)$.

5.4.1 Analysis

In the remainder of this chapter, we use the notation $F(n) \approx G(n)$ to mean that $\lim_{n\to\infty} \frac{F(n)}{G(n)} = 1$.

We see that, when we add a new layer, a leg of length 2^t decays into $t+1$ legs of lengths $1, 1, 2, 4, \ldots, 2^{t-1}$. Layer 2 of $LR(2^m, k)$ consists of $m+1$ legs of length $1, 1, 2, 4, \ldots, 2^{m-1}$. This gives us a recursive formula for counting legs, and therefore nodes, belonging to each layer.

Fix m and k, with $k > 1$.

Definition 5.4.1. *Let $L(t, j)$ denote the number of legs of length 2^t in the jth layer of $LR(2^m, k)$.*

The iterative construction of the graphs gives the following recurrence relation.

1. $L(0, k) = 2$, and $L(1, k) = L(2, k) = \cdots = L(m-1, k) = 1$.
2. For $t > 0$ and $1 < j < k$, we have $L(t, j) = \sum_{u>t} L(u, j+1)$.
3. For $t = 0$ and $1 < j < k$, we have $L(t, j) = L(t-1, j) + 2\sum_{u>t} L(u, j+1)$.

Lemma 5.4.2. *For $1 < j \le k$, the solution of this recurrence relation is given by*

$$L(t,j) = \begin{cases} \binom{m-1-t}{k-j}, & \text{if } t \ge 1; \\ 2\sum_{i=0}^{k-j} \binom{m-1}{i}, & \text{if } t = 0. \end{cases}$$

Now, let n denote the number of nodes in $LR(2^m, k)$. Clearly, the number of nodes in layer j is $1 + \sum_t L(t, j)$. We can simplify the part of the sum with $t \ge 1$, since

$$\sum_{t=1}^{m-1} L(t,j) = \sum_{t=1}^{m-1} \binom{m-1-t}{k-j} = \binom{m-1}{k-j+1}.$$

This observation leads to a simple formula for n.

$$\begin{aligned} n &= 2^m + 1 + \sum_{j=2}^{k} \left(1 + \sum_{t\ge 1} L(t,j) + L(0,j) \right) \\ &= 2^m + k + \sum_{j=2}^{k} \left(\binom{m-1}{k-j+1} + 2\sum_{i=0}^{k-j} \binom{m-1}{i} \right) \\ &= 2^m + k + (2k-2)\binom{m-1}{0} + \sum_{i=1}^{k-1} (2k-2i-1)\binom{m-1}{i}. \end{aligned}$$

Lemma 5.4.3. *Fix $\epsilon > 0$. For each m, let $k = (m-1)/(2+\epsilon)$. For the family of graphs $LR(2^m, k)$ we have*

$$\lim_{m\to\infty} \frac{n}{2^m} = 1.$$

Proof. Clearly this limit is at least 1, since $n > 2^m$. We have

$$\frac{n}{2^m} = \frac{2^m + 4k - 2 + \sum_{i=1}^{k-1}(2k-2i-1)\binom{m-1}{i}}{2^m} < 1 + k\frac{\sum_{i=0}^{k-1}\binom{m-1}{i}}{2^{m-1}}. \tag{5.2}$$

The fraction on the right-hand side is precisely the probability that $m-1$ flips of a fair coin will result in fewer than k heads. By applying a Chernoff bound [82, Thm. 4.2], this probability is seen to be at most $\exp\left(-\frac{(m-1)\epsilon^2}{4(2+\epsilon)^2}\right)$. Thus the quantity on the right-hand side of Equation (5.2) goes to 1 as m goes to infinity, completing the proof. □

The previous lemma allows us to finally prove our lower bound.

Theorem 5.4.4. *For any $\epsilon > 0$, and sufficiently large n, there exist greedy traversals of cost*

$$\frac{n \log_2 n}{2+\epsilon} - o(n \log n).$$

Proof. We use the family of graphs considered in Lemma 5.4.3, so that $n \approx 2^m$. The greedy traversal of $LR(2^m, k)$ that we are interested in takes $k2^m + (k-1) \approx kn$ steps, since it circles the ring k times. Since $k = (m-1)/(2+\epsilon) \approx \log_2 n/(2+\epsilon)$ the stated bound follows. □

5.5 Exact Results for Small Cases

As part of this project, we came up with an efficient algorithm for enumerating all possible traversals and identifying the most costly one. This algorithm was implemented in Java, and the results for $n \leq 10$ are shown in Table 5.1. For $n \leq 6$, the maximum value of $C(T)$ is $2n-3$ and is obtained by the star graph $K_{1,n-1}$; only the representative case $n = 6$ is shown.

5.6 Application: Fault-Tolerant Node Naming

The original motivation for the greedy tourist algorithm was to compute n in a faulty network, as mentioned in Sections 5.2 and 4.7.1. This protocol is easily extended to compute aggregates of a network; each time a new node is explored, the walker adds that node's value to the aggregate.

In an arbitrary-topology *dynamic* anonymous network, no efficient algorithm seems known for giving all nodes guaranteed unique identifiers. This can be accomplished by the greedy tourist algorithm as follows: label the nodes in the order that they are explored. As mentioned in Section 4.7, we would need to include timeouts in case the walker dies. We note that, on the complete graph,

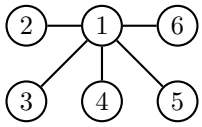
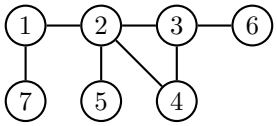
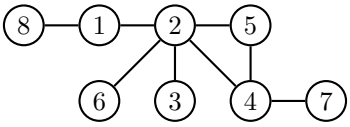
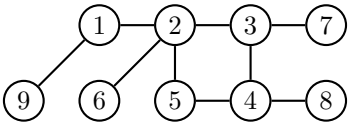
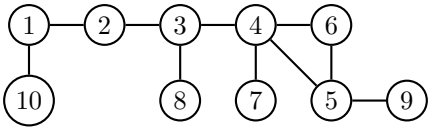
n	Maximum Cost	Representative
6	9	
7	12	
8	14	
9	17	
10	20	

Table 5.1: Greedy traversals of maximum cost for $n \leq 10$.

this is well-known as the *renaming problem* [6][85], but protocols for the renaming problem do not apply to arbitrary graphs.

5.7 Previous Results and Extensions

In [93], Rosenkrantz et al. consider the GT algorithm in the context of approximation algorithms for the Traveling Salesman Problem. They consider *weighted* graphs, where the length of a path is the sum of the edge weights along it. Let MinTSP denote the length of the shortest (not necessarily simple) cycle containing all nodes. They show, by a different method than used here, that for any weighted undirected graph on n vertices, the nearest neighbor algorithm produces a greedy traversal T such that

$$C(T) \leq \left(\frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2}\right) \text{MinTSP}.$$

Since a spanning tree traversal can be used to produce a cycle of length $2(n-1)$ in any graph, we can use their result to bound $C(T)$ by $n \log_2 n$.

The authors of [93] also give a lower bound, but only for the weighted case. They construct a graph on 2^m vertices with a Hamiltonian cycle of unit edges, but with a greedy traversal of cost $\Theta(m2^m)$. This traversal visits every node exactly once. In contrast, the expensive greedy traversal of the layered ring passes around the whole graph $\Theta(\log n)$ times. It does not seem possible to adapt their lower bound to the case where the edges are unweighted.

The upper bound of [93] improves the bound of Theorem 5.3.4 by a factor of $2 \ln 2 \approx 1.39$. Further, combining that upper bound with the lower bound of Theorem 5.4.4, the maximum number of steps in a greedy traversal is resolved within a factor of 2.

In [63], a greedy algorithm is given for solving approximate TSP instances on *planar* graphs, that produces traversals of cost $O(n)$. It is straightforward to make the graphs $LR(2^m, k)$ planar, which informs us that the greedy tourist algorithm does *not* enjoy a linear performance guarantee on planar graphs.

We had originally conjectured that, on unweighted graphs, the greedy tourist algorithm satisfied $C(T) = O(n)$, which we later discovered to be false. However, given that our lower bound construction is quite fragile, it is possible that a *randomized* nearest neighbor algorithm might have *expected* linear cost. For example, we could replace line 7 of Algorithm 5.1 with "select *dest* uniformly at random from the nearest unvisited nodes." An alternative which is more palatable for a distributed implementation is that, at each step, the walker randomly moves to a neighbor whose distance to an unexplored vertex is minimized. So far as we know, the performance of this algorithm is an open question.

Chapter 6

Discrete Harmonic Functions and Eulerian Graphs

In mathematical analysis, a function is said to be *harmonic* (or *analytic*) if its value at each point x is the average of the values of that function in a (suitably defined) neighborhood of x. This idea can be adapted to finite graphs: each vertex has value equal to the mean of its neighbors' values.

These functions are natural candidates for steady states of the model described in Chapter 4. In this chapter, we characterize the convergence and stability of harmonic functions, and discuss potential applications to distributed algorithms.

6.1 Basic Properties

In this chapter, $\mathcal{G} = (V, E)$ is a *directed,* finite graph, and each edge (v_i, v_j) has a positive weight w_{ij}. Define w_i to be the total weight leaving v_i and W to be the total weight,

$$w_i := \sum_{v_j \in \Gamma(v_i)} w_{ij} \qquad \text{and} \qquad W := \sum_{v_i \in V} w_i.$$

By convention, for a function $f : V \to X$, we write f_i for the value of f at v_i. In this chapter X will always be a $\mathbb{R}$-vector space, thus f may be interpreted as a $|V|$-dimensional vector over X.

A directed, weighted graph is *Eulerian* if every node has total in-weight equal to its total out-weight:

$$\mathcal{G} \text{ is Eulerian} \qquad \Leftrightarrow \qquad \sum_{j \in \Gamma(i)} w_{ij} = \sum_{i \in \Gamma(j)} w_{ji}, \text{ for all } v_i \in V.$$

Note that an undirected graph is always Eulerian, since $w_{ij} = w_{ji}$. The nicest results in this chapter hold only for Eulerian graphs, but for now we consider all graphs.

Let B denote a subset of V which we call the *boundary*. A vertex that is in $V - B$ is called an *interior vertex*. We say that f is *harmonic (on $\mathcal{G}$) with boundary B* if, for all interior vertices v_i,

$$f_i = \frac{\sum_{v_j \in \Gamma(v_i)} w_{ij} f_j}{w_i}. \tag{6.1}$$

In other words, the value of f at v is equal to the weighted average of its values at v's outneighbors.

Here is a simple characterization of all harmonic functions on a fixed graph with a fixed boundary.

Proposition 6.1.1. *For a fixed weighted directed graph and fixed boundary, the set of all harmonic functions is a vector subspace of X^V.*

Proof. Let f and g be harmonic with the specified boundary, and let $\alpha, \beta \in \mathbb{R}$. It follows easily from Equation (6.1) that $\alpha f + \beta g$ is a harmonic function. □

Now we look at some elementary properties of harmonic functions.

Proposition 6.1.2. *Let f be a real-valued harmonic function on $\mathcal{G}$ with boundary B. If v is an interior vertex, then either*

$$f(u) = f(v) \text{ for all } u \in \Gamma(v),$$

or there exist $u', u'' \in \Gamma(v)$ such that

$$f(u') < f(v) < f(u'').$$

Proof. This proposition follows immediately from Equation (6.1). □

Proposition 6.1.2 rules out strict local extrema except on the boundary. A boundary B is *proper* for a directed graph $\mathcal{G}$ if, for all $v \in V - B$, there exists a directed path from v to some boundary vertex $b \in B$. When the boundary is proper, it is straightforward to show [75, §3] that global extrema occur only on the boundary.

Proposition 6.1.3 (Maximum Principle)**.** *If a real-valued discrete harmonic function f has a proper boundary, the maximum and minimum values of f occur on the boundary.*

Another important consequence of having a proper boundary is that every set of boundary values extends uniquely to a harmonic function on all of V. Let A_{int} denote the weighted adjacency matrix of $\mathcal{G}$, modified by zeroing out rows corresponding to boundary vertices. Let D denote the diagonal matrix with $D_{ii} = w_i$.

Proposition 6.1.4. *Let $g \in \mathbb{R}^V$ be a vector containing values for boundary vertices, and zeroes for interior vertices. If the boundary is proper, then the matrix $D - A_{int}$ is invertible, and moreover the unique harmonic extension of g to all of V is the function f given by*

$$f = (D - A_{int})^{-1} Dg. \tag{6.2}$$

Proof. First, it is straightforward to see that the harmonic formula, Equation (6.1), is equivalent to

$$f = D^{-1} A_{int} f + g. \tag{6.3}$$

Equation (6.3) in turn can be rearranged to give Equation (6.2), provided that $(D - A_{int})$ is invertible, and so this invertibility is all that we need to show.

Let y be in the right nullspace of $(D - A_{int})$, so that $(D - A_{int})y = 0$. Rearrange to get $y = D^{-1} A_{int} y + 0$, which in comparison with Equation (6.3), implies that y is a harmonic function with all boundary values zero. By the Maximum Principle, all of y is zero. Thus $(D - A_{int})$ has a trivial right nullspace, implying that it is invertible. □

Note that this result holds even when f is a vector-valued function, by applying the same argument to each component of f.

Using Propositions 6.1.1 and 6.1.4, we get the following.

Corollary 6.1.5. *Regarded as a vector subspace of $\mathbb{R}^V$, the set of all real harmonic functions on a fixed graph with a fixed proper boundary B has dimension $|B|$. One basis is formed by the harmonic functions whose boundary values are 1 at one boundary point and 0 at all others.*

6.2 Interpretation

Real-valued harmonic functions represent the equilibria positions of points in spring networks [29] and the electric potentials of nodes in electrical networks [22]. Showing the equivalence of the harmonic formula (6.1) with these two systems requires just a little knowledge of physics.

There is another interpretation of harmonic functions which is very useful for our purposes, discussed previously in [22]. It allows us to succinctly describe the unique extension described by Proposition 6.1.4. For a weighted directed graph $\mathcal{G}$, we define the *(weighted) random walk* on $\mathcal{G}$ by specifying that the probability we step to v_j from v_i is w_{ij}/w_i. We remark for later that the stationary distribution for a weighted random walk on an Eulerian graph is $\frac{w_i}{W}$ at each node v_i, and this fact [30] is easy to verify.

Theorem 6.2.1. *Assign values $\{g_b \mid v_b \in B\}$ to the boundary nodes of a graph $\mathcal{G}$. For a node v_i, define the random variable $v_{\rho(i)}$ as follows. Start a random walker at v_i, and while the walker is not on the boundary, have it take a single step of the random walk on $\mathcal{G}$. Define $v_{\rho(i)}$ to be the position of the walker when it stops. Define f_i to be the expected value of g where the walker reaches the boundary,*

$$f_i = \mathsf{E}[g_{\rho(i)}].$$

The function f is harmonic and agrees with g on all boundary points.

Proof. It is clear that $f = g$ on the boundary, since $v_{\rho(i)} = v_i$ with probability 1. Since the random walk is a Markov chain, for any internal node v_i, the linearity of expectation allows us to write

$$\mathsf{E}[g_{\rho(i)}] = \sum_{v_j \in \Gamma(v_i)} \frac{w_{ij}}{w_i} \mathsf{E}[g_{\rho(j)}].$$

Thus, f satisfies the definition of a harmonic function. □

6.3 Stability of Harmonic Functions

In this section, we consider the following problem: if we have a graph $\mathcal{G}$ with proper boundary B, and we add a new edge to $\mathcal{G}$, how much can this change the values of the resulting harmonic function? This is motivated by the possibility of using a harmonic function as a tool in a faulty network. If we can show that a single edge doesn't change the function too much, we may gain some stability in the face of edge failures.

First, we show that, when adding an edge (v_i, v_j) of unit weight, the node whose value changes the most is v_i.

Proposition 6.3.1. *Let $\mathcal{G}$ be a graph with proper boundary B, let v_s be an interior vertex, and let $\mathcal{G}'$ be $\mathcal{G}$ augmented with the edge (v_s, v_t). Let f denote a harmonic function on $\mathcal{G}$, and let f' denote a harmonic function on $\mathcal{G}'$ with matching boundary values. Then for all i,*

$$|f_i - f'_i| \leq |f_s - f'_s|.$$

Proof. Let the boundary nodes be $v_{b1}, \ldots, v_{bk}$, and the boundary values of f be $f_{b1}, \ldots, f_{bk}$. Let X denote the vector space in which f takes its values. Let $\mathcal{G}''$ denote the graph obtained from $\mathcal{G}$ by turning v_s into a boundary vertex. Let $\mathfrak{f}^{Y:[p_{b1},\ldots,p_{bk},p_s]}$ denote the harmonic function on $\mathcal{G}''$ with values in the vector space Y, and boundary values $p_{b1}, \ldots, p_s \in Y$.

Let p be a parameter in X. The superposition principle of Proposition 6.1.1 shows that

$$\mathfrak{f}^{X:[f_{b1},\ldots,f_{bk},p]} = \mathfrak{f}^{X:[f_{b1},\ldots,f_{bk},0]} + \mathfrak{f}^{X:[0,\ldots,0,p]}.$$

Further, it is easy to see that $\mathfrak{f}^{X:[0,\ldots,0,p]} = p\mathfrak{f}^{\mathbb{R}:[0,\ldots,0,1]}$, so we have

$$\mathfrak{f}^{X:[f_{b1},\ldots,f_{bk},p]} = \mathfrak{f}^{X:[f_{b1},\ldots,f_{bk},0]} + p\mathfrak{f}^{\mathbb{R}:[0,\ldots,0,1]}. \tag{6.4}$$

In other words, the position of each v_i is a linear function of the position p of v_s.

Now, we have $f = \mathfrak{f}^{X:[f_{b1},\ldots,f_{bk},f_s]}$ and $f' = \mathfrak{f}^{X:[f_{b1},\ldots,f_{bk},f'_s]}$. By Equation (6.4),

$$\begin{aligned} f - f' &= (\mathfrak{f}^{X:[f_{b1},\ldots,f_{bk},0]} + f_s\mathfrak{f}^{\mathbb{R}:[0,\ldots,0,1]}) - (\mathfrak{f}^{X:[f_{b1},\ldots,f_{bk},0]} + f'_s\mathfrak{f}^{\mathbb{R}:[0,\ldots,0,1]}) \\ &= (f_s - f'_s)\mathfrak{f}^{\mathbb{R}:[0,\ldots,0,1]}. \end{aligned}$$

The maximum principle implies that $\mathfrak{f}_i^{\mathbb{R}:[0,\ldots,0,1]} \in [0,1]$ for each i, from which the proposition follows. □

A similar argument shows that, of all potential edges we could add which leave from v_s, the maximum change in f_s is incurred when the destination of the edge is on the boundary.

Proposition 6.3.2. *Let $\mathcal{G}$ be a graph with proper boundary B, with a fixed set of boundary values $\{g_b \mid v_b \in B\}$. Let $\mathcal{G}^{[i]}$ indicate $\mathcal{G}$ augmented with the unit edge (v_s, v_i), and let $f^{[i]}$ denote the harmonic extension of $\{g_b \mid v_b \in B\}$ on $\mathcal{G}^{[i]}$. Then for all i,*

$$f_s^{[i]} \in \mathsf{ConvexHull}\{f_s^{[b]} \mid v_b \in B\}.$$

Proof. Let $\mathcal{G}'$ be the graph $\mathcal{G}$ augmented with an additional boundary vertex v_{n+1} and an additional edge $\{v_s, v_{n+1}\}$. Let $\mathfrak{f}^p$ denote the harmonic function on $\mathcal{G}'$ with boundary values $\{g_b \mid v_b \in B\}$ and $g_{n+1} = p$. As in the proof of Proposition 6.3.1, we find that $\mathfrak{f}$ is a linear function of p, say $\mathfrak{f}^p = \mathfrak{a} + \mathfrak{b}p$, with $\mathfrak{b}$ a real vector and $\mathfrak{a}$ a vector taking its values in the same space as f.

Next, we claim that $f^{[i]} = \mathfrak{f}^{f_i^{[i]}}$. This holds because, fixing the destination of the new edge in $\mathcal{G}'$ at $f_i^{[i]}$, we obtain a harmonic function equivalent to $f^{[i]}$. By the maximum principle, $f_i^{[i]} \in \mathsf{ConvexHull}\{g_b \mid v_b \in B\}$. Thus

$$f_s^{[i]} = \mathfrak{f}_s^{f_i^{[i]}} = \mathfrak{a}_s + \mathfrak{b}_s f_i^{[i]} \in \mathfrak{a}_s + \mathfrak{b}_s\mathsf{ConvexHull}\{g_b \mid v_b \in B\}.$$

Since the ConvexHull operation commutes with a linear transformation, we conclude that

$$f_s^{[i]} \in \mathsf{ConvexHull}\{\mathfrak{a}_s + \mathfrak{b}_s g_b \mid v_b \in B\} = \mathsf{ConvexHull}\{f_s^{[b]} \mid v_b \in B\},$$

where the last equality comes from the fact that, when v_b is on the boundary, $f^{[b]} = \mathfrak{f}^{f_b^{[b]}} = \mathfrak{f}^{g_b}$. □

Now that we have determined how we can maximally change the graph by adding a single edge, we shall bound the value of this maximum change.

Proposition 6.3.3. *Let $\mathcal{G}$ and B be as in Proposition 6.3.2, and $\mathcal{G}'$ be $\mathcal{G}$ augmented with a new edge (v_s, v_t) of weight w_{new} where $v_s \notin B$ and $v_t \in B$. Let f denote a harmonic function on $\mathcal{G}$, and f' denote the harmonic function induced by the same boundary conditions on $\mathcal{G}'$. Let w_s denote the total weight of edges leaving v_s in $\mathcal{G}$, and p_s be the probability that a random walk in $\mathcal{G}$, starting from v_s, returns to v_s before hitting the boundary. Then*

$$f'_s - f_s = \frac{w_{new}}{w_{new} + w_s(1 - p_s)}(f_t - f_s).$$

Proof. Recall from Theorem 6.2.1 the interpretation of a harmonic function as the expected value of a random walk. Consider a random walk starting from v_s in $\mathcal{G}$, and let us stop this walk when it either hits a boundary node or when the walk returns to v_s. Let p_i denote the probability that this walk stops at v_i.

By linearity of expectation, and since the walk is a Markov chain, we have that

$$f_s = p_s f_s + \sum_{v_b \in B} p_b f_b, \qquad \text{and consequently} \qquad f_s = \left(\sum_{v_b \in B} p_b f_b\right)/(1 - p_s). \tag{6.5}$$

Consider performing the random walk on $\mathcal{G}'$ instead, and again stop when we hit a boundary node or v_s; the only difference from the walk on $\mathcal{G}$ is that, with probability $w_{new}/(w_{new} + w_s)$, the walk proceeds directly to v_t. This gives

$$f'_s = \frac{w_{new}}{w_{new} + w_s} f_t + \frac{w_s}{w_{new} + w_s}\left(p_s f'_s + \sum_{v_b \in B} p_b f_b\right), \qquad \text{thus} \qquad f'_s = \frac{w_{new} f_t + w_s\left(\sum_{v_b \in B} p_b f_b\right)}{w_{new} + w_s - w_s p_s}.$$

By subtracting the above two equations and cross-multiplying, we obtain

$$f'_s - f_s = \frac{w_{new}}{w_{new} + w_s(1 - p_s)}\left(f_t - \frac{\sum_{v_b \in B} p_b f_b}{1 - p_s}\right).$$

Finally, we substitute Equation (6.5) in the above equation to prove the proposition. □

Let $\mathsf{Diam}(f)$ denote the maximum Euclidean distance between any two values of f on the bound-

ary of . The term $(f_t - f_s)$ in Proposition 6.3.3 is roughly proportional to $\mathsf{Diam}(f)$ in the worst case; precisely, $|f_s - f_t| \leq \mathsf{Diam}(f)$ by the maximum principle, and for each s, there exists some t for which $|f_s - f_t| \geq \mathsf{Diam}(f)/2$.

From Propositions 6.3.1, 6.3.2, and 6.3.3, we have the following.

Theorem 6.3.4. *Let f be harmonic on $\mathcal{G}$ with proper boundary B. Let $\mathcal{G}'$ be constructed from $\mathcal{G}$ by adding a single edge (v_s, v_t) of weight w_{new}, and let f' be the harmonic extension of f's boundary values to $\mathcal{G}'$. Let $C(v_s)$ denote the maximum of $\max_{v_i \in V} |f_i - f'_i|$, over all choices of v_t. With w_s and p_s as in Proposition 6.3.2, we have*

$$C(v_s) \leq \frac{w_{new}\mathsf{Diam}(f)}{w_{new} + w_s(1 - p_s)} \leq 2C(v_s).$$

Thus, the stability of a harmonic function does not depend on the boundary values except for the scale factor $\mathsf{Diam}(f)$. While Theorem 6.3.4 gives an exact bound on the maximum change of a harmonic function, it is a bit unpalatable, and now we will make it a bit sweeter.

Let $\mathcal{G}^\star$ denote a modified version of $\mathcal{G}$ in which all of the boundary nodes are identified into a single node $v^\star$. Lovász proves in [74, Prop. 2.3] that

$$1 - p_s = \frac{1}{\kappa(v_s, v^\star)\pi(v_s)}$$

where the commute time $\kappa(v_s, v^\star)$ is the expected time for a random walker in $\mathcal{G}^\star$, starting at v_s, to go to $v^\star$ and back to v_s, and $\pi(v_s)$ is the steady-state distribution of v_s for the random walk on $\mathcal{G}^\star$. Note that this analysis is only valid when there is at least one path from the boundary to v_s.

To simplify, we assume for the remainder of the section that $\mathcal{G}$ is a unit Eulerian multigraph. Thus $W = m$, also $\pi(v_s) = \delta(v_s)/m$ and $w_s = \delta(v_s)$. With $C(v_s)$ as in Theorem 6.3.4, we get

$$C(v_s) = \frac{\Theta(\mathsf{Diam}(f))}{1 + m/\kappa(v_s, v^\star)}.$$

We thus call $m/\kappa(v_s, v^\star)$ the *harmonic stability factor* of v_s in $\mathcal{G}$, and we call $\min_s 2m/\kappa(v_s, v^\star)$ the harmonic stability factor of $\mathcal{G}$. Generally speaking, a stability factor less than 1 means that adding an edge can change a harmonic function by $\Theta(\mathsf{Diam}(f))$, while a stability factor $k > 1$ implies that this change is at most $O(\mathsf{Diam}(f)/k)$.

In undirected graphs, we have from [74] that $2m/\kappa(v_s, v^\star) = 1/R_{v_s, v^\star}$, where $R_{p,q}$ denotes the resistance between p and q when all edges are interpreted as unit resistors. By Raleigh's shortcut principle [74, Cor. 4.2] we get the following.

Proposition 6.3.5. *Adding a new edge to an undirected graph does not decrease its stability factor.*

Question 6.3.6. *Is is analogously true that adding a directed cycle of edges to an Eulerian graph does not decrease its stability?*

Again, in an undirected graph, let λ_2 denote the second eigenvalue of the transition matrix; then by [74, Cor. 3.3] we have

$$\frac{1-\lambda_2}{\delta(v_s)^{-1}+\delta(v^\star)^{-1}} \leq \frac{2m}{\kappa(v_s,v^\star)} \leq \frac{2}{\delta(v_s)^{-1}+\delta(v^\star)^{-1}}.$$

In particular the stability factor is at most $2\delta(v_s)$, so we would not expect bounded-degree graphs to be especially stable. On the other hand, an expander graph should be at roughly as stable as its minimum degree.

6.4 Convergence of Harmonic Functions

Now we discuss computing a harmonic function distributively. Let the weighted, directed graph $\mathcal{G}$ model a computer network. In our model, each v_i stores its value f_i, the node v_i can read f_j whenever $(v_i, v_j) \in E$, and v_i knows each outgoing weight w_{ij}.

Here is how we can compute the harmonic function: when an interior node v_i activates, set

$$f_i \leftarrow \frac{\sum_{v_j \in \Gamma(v_i)} w_{ij} f_j}{w_i}.$$

This will be called the *averaging operation,* and it fits the model of computation from Chapter 4. The fixed points of this operation are precisely harmonic functions. We will show that the convergence of this process is fast when the random walk associated with the weights hits the boundary in few expected steps. For the rest of this section, we consider a fixed directed, weighted graph $\mathcal{G}$ with fixed proper boundary B.

Definition 6.4.1. *Define the row-stochastic matrix T by*

$$T_{ij} = \begin{cases} 1, & \text{if } v_i \in B \text{ and } v_i = v_j; \\ w_{ij}/w_i, & \text{if } v_i \notin B \text{ and } v_j \in \Gamma(v_i); \\ 0, & \text{otherwise.} \end{cases}$$

Note that T is the transition matrix for a random walk on $\mathcal{G}$ that stays fixed once we hit the boundary. Also, note that a function f is harmonic if and only if $Tf = f$.

The next theorem shows how quickly the averaging process converges, under the ℓ^∞ norm.

Theorem 6.4.2. *Consider a network with a proper boundary. Let $\tau \in \mathbb{Z}$ be large enough that, starting from any internal node, the expected time for a random walk to hit the boundary is at most τ. Let f^0 denote the initial values stored at the nodes, and let f denote the harmonic function which agrees with f^0 on the boundary. Synchronously perform the averaging operation at all interior vertices, and let f^t denote the node values after t averaging steps. We have*

$$\max_i \left|f_i^{2\tau} - f_i\right| \leq \frac{1}{2} \max_i \left|f_i^0 - f_i\right|.$$

Proof. Recall that $f = Tf$ since f is harmonic. Our averaging process implies that, for any $t \in \mathbb{Z}^+$,

$$f^t = T^t f^0, \text{ and so } (f^t - f) = T^t (f^0 - f). \tag{6.6}$$

For any i, it follows from the above equation that

$$\left|f_i^t - f_i\right| = \left|\sum_j T_{ij}^t (f_j^0 - f_j)\right|.$$

Note that $f_j^0 - f_j = 0$ when v_j is a boundary node, so the above sum may be taken only over internal nodes. Furthermore, T^t is a row-stochastic matrix corresponding to t consecutive steps of the random walk. By Markov's inequality [82, Thm. 3.2] and the definition of τ, it follows that $\sum_{v_j \notin B} T_{ij}^{2\tau} \leq 1/2$. Combining these facts with the triangle inequality, we have

$$\left|f_i^{2\tau} - f_i\right| = \left|\sum_{v_j \notin B} T_{ij}^{2\tau} (f_j^0 - f_j)\right| \leq \sum_{v_j \notin B} T_{ij}^{2\tau} \max_j |f_j^0 - f_j| \leq \max_j |f_j^0 - f_j|/2.$$

Finally, the stated theorem holds by applying this argument to all i. □

By using Equation (6.6), one can characterize the long-term convergence speed in terms of the eigenvalue gap of T, but the $O(\tau)$ upper bound suffices for our purposes.

For some graphs, τ may be exponentially large, such as [30, Example 4]. It is well-known that $\tau = O(mn)$ for undirected graphs and this also holds for directed Eulerian graphs (we give a proof in Section 6.5). Assuming that f initially achieves its maximum and minimum on the boundary, the averaging process will converge to within $\varepsilon \mathsf{Diam}(f)$ of the true harmonic values in $O(mn \log \varepsilon^{-1})$ rounds. The assumption is easy to satisfy, for example pick any boundary value as the initial value for all internal nodes.

As an application, we can distributively compute the *exact* values of a harmonic function, provided that the edge weights and boundary values are integers. By Proposition 6.1.4, $f = (D - A_{int})^{-1} Dg$. Applying Hadamard's inequality [90], we have $\det(D - A_{int}) \leq \prod_{i=1}^n \sqrt{2} w_i \leq$

$2^{n/2}W^n$. By the matrix inversion formula, each value f_i is a rational number with denominator $O(\log n + n \log W)$ bits long. Therefore, in $O(\tau(\log n + n \log W + \log \mathsf{Diam}(f)))$ rounds, each node v_i will have a precise enough value to perform rational reconstruction of f_i via continued fractions. In the $\mathcal{CONGEST}$ model of computation, there is additional slowdown due to congestion, since the floating-point values exchanged between nodes are $\omega(\log n)$ bits long.

6.5 Hitting Time of Eulerian Graphs

The *hitting time* h_{ij} is the expected number of steps taken by a random walk to reach v_j starting from v_i. It is known [82, p. 134] that $h_{ij} \leq 2mn$ when $\mathcal{G}$ is undirected. We now show that this result also holds for Eulerian graphs.

Lemma 6.5.1. *Suppose we have a weighted directed Eulerian graph $\mathcal{G}$ and that $v_j \in \Gamma(v_i)$. Then*

$$h_{ij} \leq \frac{W}{w_{ij}}.$$

Proof. By the Fundamental Theorem of Markov Chains [82, p. 132], the expected time between successive visits to v_i on our walk is the reciprocal W/w_i of the stationary probability. Once we are at v_i, we expect to leave via the edge (v_i, v_j) once every w_i/w_{ij} attempts. So, in expectation we get to v_j within $W/w_i \cdot w_i/w_{ij} = W/w_{ij}$ steps. □

Theorem 6.5.2. *Let $\mathcal{G}$ be a weighted directed Eulerian graph and let w_{min} be the minimum weight of any edge. If $\mathcal{G}$ is strongly connected, then*

$$\max_{v_i, v_j \in V} h_{ij} \leq \mathsf{Diam}(\mathcal{G}) \frac{W}{w_{min}}.$$

Proof. Fix any particular v_i and v_j; then there is a path of length at most $\mathsf{Diam}(\mathcal{G})$ between them. We can apply the previous lemma to show that the expected time to advance on each step of the path is at most $\frac{W}{w_{min}}$. By the linearity of expectation, the expected time to traverse the total path is bounded by $\frac{W}{w_{min}}$ times the length of the path. □

In particular, this gives a bound of $O(mn)$ on the hitting time of a directed Eulerian multigraph. For some graphs like the "lollipop" graph [82, p. 133] this bound is tight.

6.6 A Distributed Eulerizing Algorithm

The idea of using a harmonic function as a computational tool seems to be more appealing for Eulerian graphs than for arbitrary graphs, as evidenced by the nice theorems of the last three sections. For the applications which we discuss in Section 6.7, the correctness is not affected if we change the edge weights. We will prove in this section that any graph can be *Eulerized* (made Eulerian), and give a distributed algorithm for Eulerization that fits the balancing model of Chapter 4.

Note that the Eulerian property is independent of the boundary, so even if the boundary of a graph changes with time, the graph will only have to be Eulerized once.

Proposition 6.6.1. *Let $\mathcal{G}$ be a strongly connected directed graph. It is possible to assign positive integer weights of size at most n^2 to the edges such that the resulting graph is Eulerian.*

Proof. Let us initially set all of the weights to 1. For each vertex v_i, define ϕ_i to be the sum of the weights leaving v_i minus the sum of the weights entering v_i:

$$\phi_i = \sum_{(i,j)\in E} w_{ij} - \sum_{(j,i)\in E} w_{ji}.$$

Note that $\sum \phi_i = 0$. Also, note that the graph is Eulerian if and only if each ϕ_i is zero. We define the discrepancy Φ by

$$\Phi = \sum_{\phi_i > 0} \phi_i = \frac{1}{2} \sum_{v_i \in V} |\phi_i|.$$

Initially, $|\phi_i| < n$ for each i, and so $\Phi < n^2/2$. Our strategy is to iteratively reduce Φ to 0, at which point the graph is Eulerian.

At each step, we find a vertex v_t for which $\phi_t > 0$ and a vertex v_u for which $\phi_u < 0$. Then, we pick any simple path from v_u to v_t, and add one to the weight of each edge along that path. As a result, ϕ_t decreases by 1, ϕ_u increases by 1, and the other ϕ_i are unchanged. Thus Φ decreases by 1 as well.

Since there are $O(n^2)$ iterations, and each edge weight is increased by at most 1 in each iteration, the final edge weights are $O(n^2)$. □

The above bound on the weights is tight. Consider a graph on $n = 2k+2$ nodes $v_0, \ldots, v_k, v'_0, \ldots, v'_k$,

with $k^2 + 2k + 1$ directed edges

$$(v_i, v_0) \text{ for } 1 \le i \le k,$$
$$(v_0, v_0'),$$
$$(v_0', v_i') \text{ for } 1 \le i \le k, \text{ and}$$
$$(v_i', v_j) \text{ for } 1 \le i, j \le k.$$

Since each v_i has indegree k and a sole outgoing edge (v_i, v_0), the weight of that edge must be at least k if the graph is made Eulerian. Thus, the total weight incoming on v_0 is at least k^2, and so (v_0, v_0') has weight at least $k^2 = \Theta(n^2)$.

We now describe how Eulerization can be done in our model via a modification of the preflow-push algorithm mentioned in Section 4.6.4. We start out with multiple sinks (resp. sources) at nodes v_i where ϕ_i is positive (resp. negative), with initial excesses of ϕ_i. Each node v_i maintains its estimate ℓ_i of the distance to a sink, and each source augments a shortest path to a sink. This is fleshed out in Algorithm 6.1.

Algorithm 6.1 The basic Eulerization algorithm.

```
1:  procedure DISTRIBUTED-EULERIZE
2:      Let v_i denote this node
3:      Compute φ_i = Σ_{v_j ∈ Γ(v_i)} w_ij − Σ_{v_i ∈ Γ(v_j)} w_ji
4:      if φ_i > 0 then
5:          ℓ_i ← 0
6:      else
7:          ℓ_i ← 1 + min_{v_j ∈ Γ(v_i)} ℓ_j
8:      end if
9:      if φ_i < 0 then
10:         Let v_j be the outneighbor of v_i for which ℓ_j is minimal
11:         Increase w_ij by 1
12:     end if
13: end procedure
```

In contrast to the preflow algorithm of Tarjan and Goldberg [54], the Eulerizing algorithm does not require a residual graph. Also, in line 11, the edge weight is augmented by only 1, whereas in [54] edges are augmented as much as possible. Nonetheless, the asynchronous $O(n^2)$-time bound of the preflow algorithm also applies to Eulerization.

Edge capacities may be useful in practice. Suppose, after the graph is made Eulerian, that a node/edge either joins or leaves the network. More augmenting will occur, increasing the weights. Thus, in a network where failures continue to occur, the individual edge weights will grow without bound. Since the bounds of Theorems 6.3.4 and 6.5.2 depend on the total weight of the network, bounding the edge weights — by using capacities and a residual graph — would improve performance. We would also probably use the additional stabilizing operation described in [53].

6.7 Applications

In this section, we expand on a couple of known algorithmic applications of harmonic functions. In addition to these applications, harmonic functions have been used to give sequential and parallel directed k-vertex connectivity algorithms [29][72].

6.7.1 Planar Embedding

In 1963, Tutte published a classic paper [103] in which harmonic functions played a central role. For a graph $\mathcal{G}$, any cycle C such that $\mathcal{G} - C$ is connected is called *peripheral.* He showed that every planar graph has exactly 2 peripheral cycles through each edge. The main result of Tutte's paper is that, if we use this cycle as the boundary of a harmonic function, and embed it as a convex polygon in $\mathbb{R}^2$, then the resulting harmonic function gives a planar, straight-line embedding of $\mathcal{G}$.

Now, if the graph is not planar, then the harmonic embedding will of course not be planar. This gives us a simple test for planarity: compute the harmonic embedding of a graph, and then check for intersecting edges.

Question 6.7.1. *Is there a simple polynomial-time distributed algorithm to determine if a graph planar?*

Two obstacles to the obvious approach are the identification of a peripheral cycle, and the location of intersecting edges. A possible solution to the latter problem might be walkers that try to traverse the faces of the embedding.

There is a related distributed algorithm [21] which can determine the genus of a graph in polynomial time, given an upper bound on the genus. Planar graphs are exactly those with genus zero, and so this gives an efficient test for planarity, by setting the upper bound to 1. However, the algorithm is quite complicated, and relies on observing a rather large subset of the graph.

6.7.2 Single-failure Tolerant Broadcast

Consider a real-valued harmonic function. Proposition 6.1.2 informs us that strict local minima appear only on the boundary. If we can somehow remove all local minima from the interior, then each interior node v_i has two outneighbors v_j and v_k such that

$$f_j < f_i < f_k. \tag{6.7}$$

A function f satisfying Equation (6.7) for each interior vertex v_i is called an *s-t labeling* of V.

If $\mathcal{G}$ has an articulation point v_i, then we claim that no s-t labeling can exist. Let $v_j \in \Gamma^{-1}(v_i)$; then it is straightforward to show that $f_j = f_k$ for all $v_k \in \Gamma(v_j)$. Consequently, v_j does not satisfy Equation (6.7), completing the claim.

If a graph has no articulation points, then it is called 2-vertex connected. We can obtain an s-t labeling of a 2-vertex connected graph by using a harmonic function with randomly perturbed edge weights. First, we need a lemma. All paths in what follows are assumed to be simple.

Lemma 6.7.2. *Let $\mathcal{G}$ be a 2-vertex connected, directed graph. Let u, w_1, w_2 be distinct vertices of $\mathcal{G}$. There exist paths P_1 and P_2 such that P_i goes from u to w_i, and the P_i are disjoint except for u.*

Proof. Since $\mathcal{G}$ is 2-connected, there exist 2 vertex-disjoint paths Q_1, Q_2 from u to w_1. Let R be any path from u to w_2. Let the last node of R that also lies on Q_1 or Q_2 be x; without loss of generality, $x \in Q_1$. Let R' be the part of R starting from x and going to w_2, and Q_1' be the part of Q_1 from u to x. Then the paths $Q_1' \circ R'$ (Q_1' followed by R') and Q_2 have the claimed properties. □

Theorem 6.7.3. *Let $\mathcal{G}$ be a directed, weighted graph, with proper boundary v_1, v_2. Assume that $\mathcal{G}$ is 2-vertex connected. For each edge e, let there be k distinct possible positive weights $w_e^1, \ldots, w_e^k$ for that edge. Select a weight for each edge uniformly and independently at random. Let f denote the harmonic function induced by the boundary values $v_1 = 0, v_2 = 1$. Then with probability at least $1 - n(n-2)/k$, the values of f form an s-t labeling of V.*

Proof. We will show that, for each interior vertex v_i, with probability at least $1 - n/k$, there are two nodes $\{v_a, v_b\} \subseteq \Gamma(v_i)$ having unequal values in f. It will follow in that case that Equation (6.7) is satisfied by some v_j and v_k in $\Gamma(v_i)$, and a union bound then proves the theorem.

Apply Lemma 6.7.2 to $\mathcal{G}$ with $u = v_i, w_1 = v_1$, and $w_2 = v_2$; let v_a and v_b be, respectively, the second nodes on P_1 and P_2. Let $\widehat{e}^j$ denote the unit vector with $\widehat{e}^j_j = 1$ and all other entries zero. By Proposition 6.1.4, the difference $f_a - f_b$ is $(\widehat{e}^a - \widehat{e}^b)^T (D - A_{int})^{-1} Dg$. Thus $f_a = f_b$ if and only if $F = 0$, where

$$F := (\widehat{e}^a - \widehat{e}^b)^T (D - A_{int})^{*T} Dg,$$

and $*$ denotes the adjoint operator. The quantity F is a polynomial of degree n in the edge weights. We will shortly show that F is not the zero polynomial. Then, by the Schwartz-Zippel Theorem [82, Thm. 7.2], when we randomly choose the edge weights, $F = 0$ with probability at most n/k. When $F \neq 0$ we have $f_a \neq f_b$ as needed.

Finally, we show that F cannot be identically zero, completing the proof. If F is identically zero, then $f_a = f_b$ for all edge weightings. However, as we increase the edge weights on P_1 and P_2 without bound, holding the other edge weights constant, we have $f_a \to 0$ and $f_b \to 1$. Thus F is not

identically zero, as needed. □

Theorem 6.7.3 may not be tight, and the author suspects that perturbations actually succeed with probability $1 - O(n/k)$.

In [62], an s-t labeling is used to perform efficient broadcasting that tolerates the failure of any one edge. In our terms, we set $f_1 = 0$ and $f_2 = 1$, and compute a harmonic function on the (directed) graph. Each node chooses a neighbor with a smaller value as its s-parent and one with a larger value as its t-parent, thereby defining two trees. Any message to be sent is forwarded up the trees to v_1 and to v_2, and then broadcast down both trees. At most $2n$ messages are sent in total.

6.8 Extensions and Related Work

To our knowledge, the bounds in this section for the convergence and stability of harmonic functions are new. The general idea of reweighting edges, exemplified by Eulerization, leads to a couple of other possibilities.

First, by re-weighting the edges of a graph, it is possible to reduce the *mixing time.* This parameter measures how quickly the distribution of a random walker approaches the steady-state distribution. It is possible [91] to re-weight the edges of an undirected graph so that the mixing time becomes $O(\mathsf{Diam}^2)$, although the method of [91] changes the steady-state distribution of the walk. The authors of [24] point out that the distributed spectral analysis algorithm of [68] can be used to minimize the mixing time among those weightings which do not change the steady-state distribution. This is accomplished by applying convex optimization to maximize the eigenvalue gap. Their algorithm does not seem very practical, and they give no concrete complexity bounds.

One other possible application of re-weighting is to space out the values of a harmonic function. An undirected harmonic function on n vertices with $\{0, 1\}$ boundary can have adjacent nodes whose values differ by exponentially small values. For example, the n-vertex harmonic function with boundary $v_1 = 0, v_2 = 1$ and graph edges $\{(v_{i-1}, v_i)\}_{i=3}^{n} \cup \{(v_1, v_i)\}_{i=3}^{n}$ has adjacent nodes that differ by $\Theta((3/2 + \sqrt{5}/2)^{-n})$. By modifying the weights of edges it should be possible to have adjacent distances at least $n^{-O(1)}$. This would make it possible to determine the relative order of all nodes quickly, and make it possible to distributively determine the order of nodes while using only $O(\log n)$ bits per node. This would potentially make the broadcast trees of Section 6.7.2 from [62] more stable. It would be interesting to investigate trade-offs between stability, convergence, and spacing.

Appendix A

Message Passing and a Balancing-FSA Random Walk

In this appendix, we show two results: first, how message-passing can be simulated in the balancing model, and second, how a network of probabilistic finite state automata (FSAs) can simulate a random walk on an undirected graph. We assume a non-faulty graph, so the topology stays fixed over all time.

We first address the basic problem of how message-passing can be simulated in this model, since this is how the walker travels around the graph. The simplest case is where the graph consists of two adjacent nodes v_1 and v_2. We simulate message passing as follows: each node v_i holds a message buffer m_i and a mod-3 counter c_i. The idea is that, modulo 3, the counters determines how many distinct messages have been sent back and forth. Each node reads incoming messages by checking the message buffer of the other node. Full pseudocode is shown in Algorithm A.1.

Note that the buffer is of a fixed size, as only one message can be in transit at a time. To turn the counted-buffer technique into a method of simulating message-passing on arbitrary topologies, each node needs a buffer for each of its neighbors. Furthermore, when a node u reads the state of $v \in \Gamma(u)$, some kind of labeling scheme is needed, in order that u can determine which of v's outgoing buffers holds messages for u. This can be accomplished with indirect addressing, although this destroys the property that our model is ambiguous. We ignore this problem for now.

Now we describe the FSA-based random walk. In our model, each automaton can read the states of all of its neighbors, but only change its own state. The automata run asynchronously. Each node is aware of its neighbors' states only as an unordered multiset (i.e., the network is ambiguous).

Algorithm A.1 Message passing using finite automata.

```
1: procedure MESSAGE-PASSING
2:     Let v_me denote this node and v_other denote its neighbor
3:     if c_other ≡ c_me + 1 (mod 3) then
4:         Read incoming message from m_other
5:         Write outgoing message to m_me
6:         c_me ← c_other + 1 (mod 3)
7:     else if c_me ≡ c_other + 1 (mod 3) then
8:         Do nothing, my neighbor has not activated since I last sent a message
9:     else
10:        Perform initialization
11:        Write outgoing message to m_me
12:        c_me ← c_other + 1 (mod 3)
13:    end if
14: end procedure
```

To simulate a random walk, the nodes clearly need access to some source of randomness. If the nodes have $O(\log n)$ bits of memory, then the node containing the walker can randomly pick an integer from 1 to n. However, a finite automaton has $O(1)$ memory. We propose a finite-automaton based walk using as little randomness as possible: only one state allows randomness, and it enters one of two states WON or LOST, each with probability 1/2. In Algorithm A.2 we give the formal description of the FSA.

There are nine states, which we denote by BLANK, WON, LOST, DEFEATED, CSENDING, CRECEIVED, WFLIP, WREFLIP, and WRESET. A node in a state prefixed by W contains the random walker. The walker travels by moving from a node in the CSENDING state to a node in the CRECEIVED state. After the old position of the walker is "cleaned up," the CRECEIVED node enters WFLIP.

The basic mechanism for picking a random outgoing neighbor is as follows. The node containing the walker enters WFLIP and asks all of its neighbors to flip a coin, thereby entering WON or LOST. If exactly one neighbor wins, then that neighbor will be passed the walker. However, if multiple neighbors win, then more flips must take place. The walker's node enters WREFLIP, which causes those nodes that win to become BLANK, and those that lose to become DEFEATED. Then the walker node enters WFLIP again and the non-defeated nodes flip coins.

In the event that there are no winners, then the coin flipping must be restarted from scratch. This is the purpose of the state WRESET. It waits until all neighbors enter BLANK, and then the walker node becomes WFLIP to begin flipping again.

When the walker node finally has a single neighbor in WON, it enters CSENDING and waits for the winner to become CRECEIVED. Then, the CRECEIVED node enters WFLIP and the next step takes place.

Algorithm A.2 Random walk using finite automata.

```
1: procedure Blank-Action
2:     if any neighbor is in state WFlip then
3:         Enter Won with probability 1/2 and Lost with probability 1/2
4:     end if
5: end procedure
6: procedure WReceived-Action
7:     if all neighbors are in state Blank then
8:         Enter WFlip
9:     end if
10: end procedure
11: procedure WFlip-Action
12:     if any neighbor is in state Blank then
13:         Do nothing
14:     else if exactly one neighbor is in state Won then
15:         Enter WSending
16:     else if no neighbors are in state Won then
17:         Enter WReset
18:     else
19:         Enter WReflip
20:     end if
21: end procedure
22: procedure WReflip-Action
23:     if all neighbors are in state Blank or Defeated then
24:         Enter WFlip
25:     end if
26: end procedure
27: procedure WReset-Action
28:     if all neighbors are in state Blank then
29:         Enter WFlip
30:     end if
31: end procedure
32: procedure Won-Action
33:     if one neighbor is in state WSending then
34:         Enter WReceived
35:     else if one neighbor is in state WReset or WReflip then
36:         Enter Blank
37:     end if
38: end procedure
39: procedure Lost-Action
40:     if one neighbor is in state WSending or WReset then
41:         Enter Blank
42:     else if one neighbor is in state WReflip then
43:         Enter Defeated
44:     end if
45: end procedure
46: procedure Defeated-Action
47:     if one neighbor is in state WSending or WReset then
48:         Enter Blank
49:     end if
50: end procedure
51: procedure WSending-Action
52:     if one neighbor is in state WReceived, and all other neighbors are in state Blank then
53:         Enter Blank
54:     end if
55: end procedure
```

Each step of the walker takes multiple rounds (where a round, in the asynchronous model, consists of every node activating at least once). It is straightforward to show that the time for the walker to step away from a node of degree δ is $O(\log \delta)$ rounds.

This rather complicated algorithm has been included in this appendix for two reasons. First, it demonstrates that, even with finite state, we can do some interesting computation. The 2-coloring algorithm of Section 4.6.1 can also be expressed using finite-state automata.

The second reason relates to how these automata determine their next state. All of the conditions in Algorithm A.2 for changing state can be expressed using the logical connectives "and," "or," and "not," together with basic expressions of the form "there are k or more neighbors in STATE." Recall the terminology of Section 4.5, and the commutative, associative operator $\star_\sigma$. It is easy to show that all of the logical connectives here can be expressed using $\star_\sigma$, with *finite* domain D. In other words, for each state σ of our automata, the order in which neighbors are "read" does not matter, and the computation of the next state can be performed in finite space.

Thus, the $\star_\sigma$ model may be interesting from a theoretic standpoint. We can actually show that *any* finite space-computable, commutative, associative operator must compute a boolean formula in the atoms "there are k or more neighbors in STATE" and "the number of neighbors in STATE, mod j, is equal to k." The converse result is also true: any such formula can be expressed using a finite space-computable, commutative, associative operation on the neighbors. From this equivalence we might derive some impossibility results in the balancing finite automaton model.

Bibliography

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Commun. ACM*, 43(5):74–82, 2000.

[2] Y. Afek, B. Awerbuch, E. Gafni, Y. Mansour, N. Shavit, and A. Rosén. Slide: The key to polynomial end-to-end communication. *J. Algorithms*, 22(1):158–186, 1997.

[3] G. Aggarwal, R. Motwaniand, D. Shah, and A. Zhu. Switch scheduling via randomized edge coloring. In *Proc. 44th Symp. Foundations of Computer Science*, page 502, 2003.

[4] M. Ahuja and Y. Zhu. An efficient distributed algorithm for finding articulation points, bridges, and biconnected components in asynchronous networks. In *Proc. 9th Conf. Foundations of Software Technology and Theoretical Computer Science*, pages 99–108, 1989.

[5] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583, 1986.

[6] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.

[7] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

[8] B. Awerbuch. A new distributed depth-first-search algorithm. *Inform. Process. Lett.*, 20:147–150, 1985.

[9] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proc. 19th Symp. Theory of Computing*, pages 230–240, 1987.

[10] B. Awerbuch. Distributed shortest paths algorithms. In *Proc. 21st Symp. Theory of Computing*, pages 490–500, 1989.

[11] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear cost sequential and distributed constructions of sparse neighborhood covers. In *Proc. 34th Symp. Foundations of Computer Science*, pages 638–647, 1993.

[12] B. Awerbuch, A. Goldberg, M. Luby, and S. Plotkin. Network decomposition and locality in distributed computation. In *Proc. 30th Symp. Foundations of Computer Science*, pages 364–369, 1989.

[13] B. Awerbuch and T. Leighton. A simple local-control approximation algorithm for multicommodity flow. In *Proc. 34th Symp. Foundations of Computer Science*, pages 459–468, 1993.

[14] B. Awerbuch and T. Leighton. Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks. In *Proc. 26th Symp. Theory of Computing*, pages 487–496, 1994.

[15] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Saks. Adapting to asynchronous dynamic networks. In *Proc. 24th Symp. Theory of Computing*, pages 557–570, 1992.

[16] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st Symp. Foundations of Computer Science*, pages 514–522, 1990.

[17] B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. 31st Symp. Foundations of Computer Science*, pages 503–513, 1990.

[18] Y. Bartal, J. W. Byers, and D. Raz. Global optimization using local information with applications to flow control. In *Proc. 38th Symp. Foundations of Computer Science*, pages 303–312, 1997.

[19] J. Beal. Persistent nodes for reliable memory in geographically local networks. Artificial Intelligence Lab Technical Report 2003-011, MIT, 2003.

[20] J. Beal. A robust amorphous hierarchy from persistent nodes. In *Proc. Communication Systems and Networks*, 2003.

[21] I. Benjamini and L. Lovász. Global information from local observation. In *Proc. 43rd Symp. Foundations of Computer Science*, pages 701–710, 2002.

[22] I. Benjamini and L. Lovász. Harmonic and analytic functions on graphs. *J. Geom.*, 76(1):3–15, 2003.

[23] J.-C. Bermond and J.-C. König. General and efficient decentralized consensus protocols II. In M. Cosnard et al., editor, *Parallel and Distributed Algorithms*, pages 199–210. North-Holland, 1988.

[24] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Gossip algorithms: design, analysis and applications. In *Proc. 24th IEEE Infocom, to appear*, 2005.

[25] E. J.-H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Softw. Eng.*, SE-8:391–401, 1982.

[26] P. Chaudhuri. An optimal distributed algorithm for computing bridge-connected components. *The Computer Journal*, 40(4):200–207, 1997.

[27] P. Chaudhuri. An $O(n^2)$ self-stabilizing algorithm for computing bridge-connected components. *Computing*, 62:55–67, 1999.

[28] J. Chen, G. Pandurangan, and D. Xu. Robust and distributed computation of aggregates in sensor networks. In *Proc. 4th International Conf. Information Processing in Sensor Networks*, 2005.

[29] J. Cheriyan and J. H. Reif. Directed s-t numberings, rubber bands, and testing digraph k-vertex-connectivity. *Combinatorica*, 14:435–451, 1994.

[30] F. Chung. Laplacians and the cheeger inequality for directed graphs. *Ann. Comb.*, 9:1–19, 2005.

[31] G. Couloris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 1994.

[32] J. M. Crichlow. *The Essence of Distributed Systems*. Prentice Hall, 2000.

[33] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[34] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[35] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. L. Welch. Virtual mobile nodes for mobile ad hoc networks. In *Proc. 18th International Conf. Distributed Computing*, pages 230–244, 2004.

[36] S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, and J. Welch. Geoquorums: Implementing atomic memory in mobile *ad hoc* networks. In *Proc. 17th International Conf. Distributed Computing*, 2003.

[37] D. Dubhashi, D. A. Grable, and A. Panconesi. Near-optimal, distributed edge colouring via the nibble method. *Theor. Comput. Sci.*, 203(2):225–251, 1998.

[38] D. Durand, R. Jain, and D. Tseytlin. Parallel I/O scheduling using randomized, distributed edge coloring algorithms. *J. Parallel Distrib. Comput.*, 63(6):611–618, 2003.

[39] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[40] M. Elkin. A faster distributed protocol for constructing a minimum spanning tree. In *Proc. 15th Symp. Discrete Algorithms*, pages 359–368, 2004. Full version at `http://www.cs.yale.edu/~elkin/mst.jour.ps`.

[41] M. Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *Proc. 36th Symp. Theory of Computing*, pages 331–340, 2004.

[42] J. Elson and D. Estrin. Sensor networks: A bridge to the physical world. In C. Raghavendra, K. Sivalingam, and T. Znati, editors, *Wireless Sensor Networks*, pages 3–20. Kluwer Academic Publishers, 2004.

[43] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[44] L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *Proc. 15th IPDPS Workshops on Parallel and Distributed Processing*, pages 505–511, 2000.

[45] P. Flocchini, A. Roncato, and N. Santoro. Backward consistency and sense of direction in advanced distributed systems. *SIAM J. Comput.*, 32(2):281–306, 2003.

[46] L. Ford, Jr. and D. Fulkerson. Maximal flow through a network. *Canad. J. Math.*, 8:399–404, 1956.

[47] G. N. Frederickson and N. A. Lynch. The impact of synchronous communication on the problem of electing a leader in a ring. In *Proc. 16th Symp. Theory of Computing*, pages 493–503, 1984.

[48] D. S. Fussell, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacement. *SIAM J. Comput.*, 22:587–616, 1993.

[49] J. A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27(1):302–316, 1998.

[50] S. J. Garland and N. Lynch. Using I/O automata for developing distributed systems. In G. Leavens and M. Sitaraman, editors, *Foundations of Component-based Systems*, pages 285–312. Cambridge University Press, New York, NY, USA, 2000.

[51] B. Ghosh and S. Muthukrishnan. Dynamic load balancing by random matchings. *J. Comput. Syst. Sci.*, 53(3):357–370, 1996.

[52] S. Ghosh. Cooperating mobile agents and stabilization. In *Proc. 5th International Workshop on Self-Stabilizing Systems*, pages 1–18, London, UK, 2001. Springer-Verlag.

[53] S. Ghosh, A. Gupta, and S. V. Pemmaraju. A self-stabilizing algorithm for the maximum flow problem. *Distrib. Comput.*, 10(4):167–180, 1997.

[54] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[55] D. Grable and A. Panconesi. Nearly optimal distributed edge colouring in $o(\log \log n)$ rounds. *Random Structures & Algorithms*, 10(3):385–405, 1997.

[56] M. Hańćkowiak, M. Karoński, and A. Panconesi. On the distributed complexity of computing maximal matchings. *SIAM J. Discrete Math.*, 15(1):41–57, 2001.

[57] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Commun. ACM*, 23(11):627–628, 1980.

[58] W. Hohberg. How to find biconnected components in distributed networks. *J. Parallel Distrib. Comput.*, 9(4):374–386, 1990.

[59] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[60] S. T. Huang. A new distributed algorithm for the biconnectivity problem. In *Proc. 1989 International Conf. Parallel Processing*, pages 106–113, 1989.

[61] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. 6th Conf. Mobile Computing and Networks*, pages 56–67, 2000.

[62] A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. *Inf. Comput.*, 79(1):43–59, 1988.

[63] B. Kalyanasundaram and K. R. Pruhs. Constructing competitive tours from local information. *Theoretical Comput. Sci.*, 130(1):125–138, 1994.

[64] M. H. Karaata. A stabilizing algorithm for finding biconnected components. *J. Parallel Distrib. Comput.*, 62(5):982–999, 2002.

[65] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. Technical Report MIT-LCS-TR-917a, MIT Laboratory for Computer Science, 2004. Available at `http://theory.lcs.mit.edu/tds/papers/Kirli/TIOA-synthesis.ps`.

[66] A. Kazmierczak and S. Radhakrishnan. An optimal distributed ear decomposition algorithm with applications to biconnectivity and outerplanarity testing. *IEEE Trans. Parallel Distrib. Syst.*, 11(2):110–118, 2000.

[67] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Proc. 44th Symp. Foundations of Computer Science*, pages 482–491, 2003.

[68] D. Kempe and F. McSherry. A decentralized algorithm for spectral analysis. In *Proc. 36th Symp. Theory of Computing*, pages 561–568, 2004.

[69] F. Kuhn, T. Moscibroda, and R. Wattenhofer. What cannot be computed locally! In *Proc. 23rd Symp. Principles of Distributed Computing*, pages 300–309, 2004.

[70] S. Kutten and D. Peleg. Fast distributed construction of k-dominating sets and applications. In *Proc. 14th Symp. Principles of Distributed Computing,*, pages 20–27, 1995.

[71] N. Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

[72] N. Linial, L. Lovász, and A. Wigderson. Rubber bands, convex embeddings and graph connectivity. *Combinatorica*, 8(1):91–102, 1988.

[73] N. Linial and M. Saks. Low diameter graph decomposition. *Combinatorica*, 13(4):441–454, 1993.

[74] L. Lovász. Random walks on graphs: A survey. In D. Miklós, V. T. Sós, and T. Szönyi, editors, *Combinatorics, Paul Erdös is Eighty*, volume 2, pages 353–398. János Bolyai Mathematical Society, 1996.

[75] L. Lovász. Discrete analytic functions: A survey. Available at `http://research.microsoft.com/users/lovasz/analytic.pdf`, 2000.

[76] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1055, 1986.

[77] N. Lynch. I/O automata: A model for discrete event systems. In *Proc. 22nd Conf. Information Sciences and Systems*, pages 29–38, 1988.

[78] N. Lynch. A hundred impossibility proofs for distributed computing. In *Proc. 8th Symp. Principles of Distributed Computing*, pages 1–28, New York, NY, USA, 1989. ACM Press.

[79] N. Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, 1996.

[80] N. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th Symp. Principles of Distributed Computing*, pages 137–151, 1987.

[81] S. Moran and Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Inform. Process. Lett.*, 26:145–151, 1987.

[82] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 2000.

[83] R. Nagpal and D. Coore. An algorithm for group formation and maximal independent set in an amorphous computer. Artificial Intelligence Lab Technical Report 1626, MIT, 1998.

[84] S. Nath, P. B. Gibbons, Z. Anderson, and S. Seshan. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. 2nd Conf. Embedded Networked Sensor Systems*, pages 250–262, 2004.

[85] A. Panconesi, M. Papatriantafilou, P. Tsigas, and P. Vitányi. Randomized naming using wait-free shared variables. *Distrib. Comput.*, 11(3):113–124, 1998.

[86] A. Panconesi and A. Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *Proc. 24th Symp. Theory of Computing*, pages 581–592, 1992.

[87] A. Panconesi and A. Srinivasan. Randomized distributed edge coloring via an extension of the Chernoff–Hoeffding bounds. *SIAM J. Comput.*, 26(2):350–368, 1997.

[88] D. Peleg. Time-optimal leader election in general networks. *J. Parallel Distrib. Comput.*, 8(1):96–99, 1990.

[89] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000.

[90] PlanetMath, "Hadamard's Inequality," `http://planetmath.org/encyclopedia/HadamardsInequality.html`, 2005.

[91] D. Pritchard. Getting lost efficiently. Final project for " 6:856: Randomized algorithms", MIT, May 2005. Available at `http://zeno.mit.edu/math/6856final.pdf`.

[92] J. H. Reif. Depth-first search is inherently sequential. *Inform. Process. Lett.*, 20:229–234, 1985.

[93] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6(5):563–581, 1977.

[94] B. Shieber and S. Moran. Slowing sequential algorithms for obtaining fast distributed and parallel algorithms: maximum matchings. In *Proc. 5th Symp. Principles of Distributed Computing*, pages 282–292, 1986.

[95] B. Swaminathan and K. J. Goldman. An incremental distributed algorithm for computing biconnected components in dynamic graphs. *Algorithmica*, 22:305–329, 1998.

[96] R. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[97] R. E. Tarjan. A note on finding the bridges of a graph. *Inform. Process. Lett.*, 2:160–161, 1974.

[98] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.

[99] G. Tel. *Introduction to Distibuted Algorithms*. Cambridge University Press, 2000.

[100] R. Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. In *Proc. 14th Symp. Principles of Distributed Computing*, pages 28–37, 1995.

[101] 25th TOP500 List, `http://www.top500.org/lists/2005/06/`, *2005 International Supercomputer Conf.,* 2005.

[102] Y. H. Tsin and F. Y. Chin. A general program scheme for finding bridges. *Inform. Process. Lett.*, 17(5):269–272, 1983.

[103] W. T. Tutte. How to draw a graph. *Proc. London Math. Soc.*, 13:743–768, 1963.

[104] S. Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

[105] J. Woo and S. Sahni. Computing biconnected components on a hypercube. *J. Supercomputing*, 5:73–87, 1991.