Time-Space Tradeoffs for Graph s-t Connectivity

Gregory Barnes

Technical Report 92-10-02
October, 1992

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Time-Space Tradeoffs for Graph s-t Connectivity

by

Gregory Barnes

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1992

Approved by_____

(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree_____

Date_____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, Michigan 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microfilm and/or (b) printed copies of the manuscript made from microfilm."

Signature_____

Date_____

University of Washington

Abstract

Time-Space Tradeoffs for Graph s-t Connectivity

by Gregory Barnes

Chairperson of Supervisory Committee: *Walter L. Ruzzo*
Department of Computer Science and Engineering

The problem of graph s–t connectivity, determining whether two vertices s and t in a graph are in the same connected component, is a fundamental problem in computational complexity theory. Determining the space complexity of s-t connectivity either for directed graphs (STCON) or for undirected graphs (USTCON) would tell us a great deal about the relationships among deterministic, nondeterministic, and probabilistic logarithmic space bounded complexity classes.

A fruitful intermediate step to determining the space complexity of STCON and USTCON is to explore time-space tradeoffs for the problems: the simultaneous time and space requirements of algorithms for connectivity. Prior to this work, all deterministic connectivity algorithms that used less than linear space (the space bound for well-known algorithms such as breadth-first and depth-first search) required super-polynomial time (*e.g.* Savitch's algorithm, which uses $O(\log^2 n)$ space and $n^{O(\log n)}$ time).

In this thesis, we explore various techniques for creating small-space, but polynomial-time algorithms for undirected and directed graph connectivity. We present sublinear space, polynomial time deterministic algorithms for both problems. No previous such algorithms were known for these problems; in fact, there was some evidence that such algorithms could not exist for directed graphs. We present a subproblem, the *short paths problem*, that could separate STCON and USTCON: we

present evidence that finding the length of a simple path is intrinsic to solving STCON, but not USTCON, in small space.

Table of Contents

List of Figures iii

List of Tables iv

Chapter 1: Introduction 1

1.1 Motivation 1

1.1.1 s-t Connectivity in Complexity Theory 1

1.2 Overview of This Thesis 4

1.3 Previous Algorithms 6

1.3.1 Randomized Algorithms for Connectivity 7

1.3.2 Nonuniform Algorithms for Connectivity 8

1.4 Lower Bounds for s-t Connectivity 8

1.4.1 Graph Encodings 9

1.4.2 Time Lower Bounds 10

1.4.3 Space Lower Bounds 13

1.5 Logspace Reducibility 16

Chapter 2: Time-Space Tradeoffs for Undirected s-t Connectivity 18

2.1 Introduction 18

2.2 The Simple Algorithm 19

2.2.1 Analysis of the Simple Algorithm 22

2.3 The Recursive Algorithm 24

2.3.1 Analysis of the Recursive Algorithm 30
2.4 The Batched Algorithm 35
2.4.1 Analysis of the Batched Algorithm 38
2.5 Conclusions 42

Chapter 3: Time-Space Tradeoffs for Directed s-t Connectivity 43

3.1 Introduction 43
3.2 The Bounded Queue Tradeoff 44
3.3 The Breadth-First Search Tradeoff 46
3.4 The Short Path Tradeoff 49
3.4.1 Notes on the Algorithm 53
3.5 Combining the Two Algorithms 54

Chapter 4: Conclusions 57

4.1 Introduction 57
4.2 Other Work on Undirected s-t Connectivity 57
4.2.1 A Variation of the $O(\log^{1.5} n)$ Space Algorithm 58
4.3 Undirected vs. Directed s-t Connectivity 59
4.3.1 The Importance of Finding Distances 61
4.4 Future Work 69

Bibliography 71

List of Figures

2.1 The main routine of the simple algorithm 23

2.2 The cl function of the simple algorithm 24

2.3 A graph where the search might fail 28

2.4 The main routine of the recursive algorithm 31

2.5 The cl function of the recursive algorithm 32

2.6 The ri and nv functions of the recursive algorithm 33

2.7 The batched algorithm's data structure 37

2.8 Code to find landmarks in the batched algorithm. 39

2.9 Code to find the "closest landmarks" in the batched algorithm. 40

2.10 The main routine of the batched algorithm 41

3.1 Details of the bounded queue algorithm 45

3.2 Details of the breadth-first search algorithm 47

3.3 Details of the short paths algorithm 51

3.4 Details of the recursive short paths algorithm 52

3.5 Combining the two algorithms efficiently. 55

List of Tables

1.1 The current knowledge about STCON and USTCON with respect to four complexity classes. 3

Acknowledgments

This thesis owes much of its existence to two people: A few years ago, in a meeting of a group of theory students at the University of Washington, Simon Kahan asked if it was possible to search a graph in linear space. Together we devised the algorithm that appears in Section 3.2. When we took the algorithm to my advisor, Larry Ruzzo, he pointed us toward what he claimed was a more interesting problem, finding a sublinear space, polynomial time algorithm for undirected s-t connectivity. Unlike most of the other problems Larry has thrown my way in the past few years, I was able to solve this one. Together we devised the algorithms that appear in Chapter 2. Most of this chapter appeared in the 1991 ACM Symposium on Theory of Computing [BR91]. Later, with the help of Jonathan Buss and Baruch Schieber, Larry and I came up with the remainder of Chapter 3, which appeared in the 1992 Structure in Complexity Theory Conference [BBRS92].

Throughout my graduate student career, Larry has been a good advisor, a good listener, and a good friend. Without his direction and assistance, this thesis would not have been written. I am also indebted to Paul Beame and Martin Tompa, the other two members of my reading committee, who have always been willing to listen to my ideas and answer my questions. Martin helped formulate the crossing sequences argument in Section 1.4, and the padding arguments of Section 4.3.

I also owe thanks to Uri Feige, who helped discover the minimum space bound of the algorithm in Section 3.5, and Allan Borodin, who helped point me toward the importance of finding simple paths in solving directed s-t connectivity, the subject of most of Chapter 4.

In graduate school, personal support is as important as technical support. Many people helped make my last five years as enjoyable as it was. I'd especially like to thank Donald Chinn, Eric Koldinger, Amy Martindale, Lisa Munro, Naomi Nishimura, Raj Vaswani, and Elizabeth Walkup.

Portions of this research were supported by NSF Grants CCR-8703196 and CCR-9002891. Portions of this thesis were written on the Crayfish X1 Steam-Powered Alternating Turing Machine.

Dedication

For my family, Owen, Gwen, Steve and Tim.

Chapter 1

INTRODUCTION

We are preoccupied with time. If we could learn to love space as deeply as we are now obsessed with time ... Edward Abbey *[Abb68]*

1.1 Motivation

In the graph s-t connectivity problem, we are given a graph, $G = (V, E)$, and two distinguished vertices, s and t, and must determine whether there is a path from s to t. s-t connectivity is an important problem in general, since it is the abstraction of many general search problems. For example, many artificial intelligence problems are simply searches of a large state space, where one wishes to know whether a goal state is reachable from a start state. s-t connectivity is a basic problem in graph theory and an important building block when constructing graph algorithms. Finally, s-t connectivity is particularly important in complexity theory, since it provides complete and hard problems for many small-space complexity classes. This thesis concentrates on the importance of s-t connectivity in complexity theory.

1.1.1 s-t Connectivity in Complexity Theory

The fundamental open problem in computational complexity theory is the power of nondeterminism: is a nondeterministic computing machine significantly more powerful than a deterministic one? This is the motivation underlying the $P \stackrel{?}{=} NP$ question — can one do more in nondeterministic polynomial time than deterministic polynomial time? The P vs. NP problem is more than twenty years old [Coo71], and remains a very difficult open problem.

Consider the corresponding question for nondeterministic space — can one do more with nondeterministic logarithmic space (NL) than deterministic logarithmic space (DL)?[1] This is also a difficult open problem, but we seem to know more about the power of nondeterministic space than nondeterministic time. For example, it is known that NL is closed under complementation [Imm88, Sze88], while the corresponding question for NP is still open. Savitch [Sav70] shows that nondeterministic space S is contained in deterministic space S^2 for any constructible $S = \Omega(\log n)$ (note: throughout this thesis, $\log x$ will be used to denote the logarithm base 2 of x). No such polynomial relationship is known for nondeterministic and deterministic time. The power of nondeterministic space seems to be a more easily solvable question than the power of nondeterministic time, while remaining at the core of the nondeterminism question. If we could solve the question of DL vs. NL, we could be well on the way to solving the question of the power of nondeterminism.

The complexity of s-t connectivity is central to this question. Both the s-t connectivity problem for directed graphs (STCON) and the s-t connectivity problem for undirected graphs (USTCON) are DL-hard under NC^1 reducibility. STCON is the prototypical NL-complete problem [Sav70], just as SATISFIABILITY is the prototypical NP-complete problem. Settling the deterministic space complexity of STCON, then, would solve the DL vs. NL question — if STCON $\notin DL$, then $NL \neq DL$, and nondeterministic machines are more powerful than deterministic machines. If, on the other hand, STCON $\in DL$, then any problem solvable nondeterministically in log space is solvable deterministically in log space. In addition, proving STCON $\in DL$ would, by a padding argument, show that NSPACE($f(n)$) = DSPACE($f(n)$) for all space-constructible $f(n) = \Omega(\log n)$ [Sav70, Theorem 3].

USTCON is complete for symmetric logarithmic space (SL) [LP82], a seemingly weaker variant of NL where a configuration A derives a configuration B if and only if B derives A. USTCON is known to be solvable by randomized logarithmic space, polynomial expected time algorithms, even errorless ones [AKL+79, BCD+89]. De-

[1] The choice of logarithmic here is somewhat arbitrary. The directly analogous classes of deterministic and nondeterministic polynomial space are very powerful, and it is not difficult to show they are equivalent. Logarithmic space is the most restrictive space class that remains interesting and robust — machines in space classes below log space cannot write down their position in the input string, for example.

Table 1.1: The current knowledge about STCON and USTCON with respect to four complexity classes. Key: `complete` = complete for this class under NC^1 reducibility, `hard` = hard for this class under NC^1 reducibility, `member` = a member of this class.

class	USTCON	STCON
DL	`hard`	`hard`
SL	`complete`	`hard`
ZPLP	`member`	`hard`
NL	`member`	`complete`

termining the deterministic space complexity of USTCON would help in settling the relations between *DL* and these intermediate classes, particularly if, as has been conjectured, STCON $\notin$ *DL*.

Table 1.1 summarizes the current knowledge about the membership, completeness and hardness of STCON and USTCON for four complexity classes. *ZPLP* is the class of languages solvable by randomized algorithms that run in logarithmic space and polynomial expected time with zero-sided error. See Borodin *et al.* [BCD$^+$89] for a detailed breakdown of the various deterministic, randomized, and nondeterministic log space complexity classes.

The classes in Table 1.1 are arranged from weakest to strongest, with the strongest at the bottom. That is,

$$DL \subseteq SL \subseteq ZPLP \subseteq NL.$$

Determining the space complexity of STCON and USTCON is the key to showing whether any of these containments is proper; it is not known whether any pair of these four classes is separate or distinct. Showing whether USTCON and STCON are equivalent or distinct would also be a significant result — STCON $\leq_{\log}$ USTCON $\Longleftrightarrow$ $SL = NL$ (where $\leq_{\log}$ stands for "logspace reducible to"; see Section 1.5, below).

Time-Space Tradeoffs

Unfortunately, proving separation or equality between the classes in Table 1.1 is a difficult open problem. A fruitful intermediate step is to explore *time-space tradeoffs*

for USTCON and STCON: determining lower or upper bounds on the simultaneous time and space complexity of these problems. For example, there are deterministic STCON algorithms that use linear space, and deterministic STCON algorithms that use $O(\log^2 n)$ time, so one might ask whether there is a deterministic algorithm that solves STCON *simultaneously* in these time and space bounds (denoted as STCON $\stackrel{?}{\in} DTIME, SPACE(n+m, \log^2 n)$). As we shall see, simpler questions, such as whether there is a deterministic STCON algorithm that runs in polylogarithmic space and polynomial time (STCON $\stackrel{?}{\in} DTIME, SPACE(n^{O(1)}, \log^{O(1)} n)$), are also interesting. In this thesis, we will be concerned largely with the minimum space needed for a polynomial time STCON or USTCON algorithm. This is because a *DL* algorithm must be a polynomial time algorithm. Since our ultimate goal is to show the existence or impossibility of a *DL* algorithm for either problem, it makes sense to start with a high space bound and polynomial time and see how far we can lower the space bound while maintaining polynomial time.

1.2 Overview of This Thesis

This thesis presents time-space tradeoffs for undirected and directed s-t connectivity. Using these tradeoffs, we construct algorithms for undirected and directed s-t connectivity that use simultaneous polynomial time and sublinear space. Previously, no such algorithms were known for either problem. We present a subproblem, the *short paths problem*, that could separate STCON and USTCON: we present evidence that finding path lengths is intrinsic to solving STCON, but not USTCON, in small space.

More specifically, in Chapter 2, we present time-space tradeoffs for USTCON. We begin with a simple algorithm for USTCON that uses space $O(s)$, $n^{1/2} \log n \leq s \leq n \log n$, and runs in time $O(((m+n)n^2 \log^2 n)/s)$. A generalization of this algorithm gives a second algorithm using space $O(\lambda n^{1/\lambda} \log n)$, for $2 \leq \lambda \leq \log n$, and time $n^{O(\lambda)}$. It can use as little as $\Theta(\log^2 n)$ space, but its running time becomes superpolynomial whenever its space is constrained to $n^{o(1)}$. An optimization to the simple algorithm gives a more time-efficient third algorithm: for space $O(s)$, $n^{1/2} \log n \leq s \leq n \log n$, it uses time $O((m+n)((n/s)^2 \log^3 n)(\log((n \log n)/s)))$. Together, these algorithms provide a nearly smooth tradeoff for USTCON between time-efficient, space-intensive algorithms, such as depth- and breadth-first, and time-intensive, space-efficient algorithms

such as Savitch's (see Section 1.3 below). As noted in Bar-Noy, *et al.* [BNBK+89], no previously known deterministic algorithm for USTCON achieved simultaneous polynomial time and sublinear space.

In Chapter 3, we present three time-space tradeoffs for STCON. The first is a variant of depth- and breadth-first search that can use between $\Theta(n \log n)$ and $\Theta(n)$ space. The second is another variant of breadth-first search, which, when combined with the third, gives a sublinear space, polynomial time algorithm for STCON. Again, no previously known deterministic algorithm for STCON achieved simultaneous polynomial time and sublinear space. The result is somewhat surprising given the work of Tompa [Tom82], who showed that such an algorithm is impossible for certain common approaches to solving STCON. The third tradeoff solves the *short paths problem*, a variant of STCON where we assume the distance from s to t is "short" (at most $f(n)$ for some $f(n) = o(n)$). We are not aware of any previous algorithms that solve this problem in sublinear space and polynomial time. The short paths tradeoff turns out to be more powerful than it at first appears: it is a generalization of two well-known STCON algorithms, breadth-first search and Savitch's algorithm (see Section 1.3 below).

In Chapter 4, we conclude with some observations on the difference between USTCON and STCON. In particular, we concentrate on the power of knowing a bound on the length of a simple path from s to t. We show that many path length problems, including finding the length of the shortest path from s to t in a directed or undirected graph, and solving the short paths problem to a sufficiently long distance in a directed or undirected graph, are *NL*-complete. We generalize this result to show a close relationship between algorithms for the short paths problem and for STCON: for a wide range of space bounds, an algorithm for the short paths problem to a sufficiently long distance yields an algorithm for STCON with the same asymptotic space bound. The reverse is nearly true: an STCON algorithm implies an algorithm for the short paths problem with nearly the same asymptotic space bound. None of these observations about path lengths appears to hold for small space USTCON algorithms, suggesting that knowing the length of a path is intrinsic to solving STCON, but not USTCON.

Before presenting these results, it will be useful to review previously known upper and lower bounds for USTCON and STCON.

1.3 Previous Algorithms

The well-known depth- and breath-first search algorithms give a time-optimal solution for STCON and USTCON. For n vertex, m edge graphs, both algorithms run in optimal time $\Theta(m+n)$. If one is interested in space, though, these algorithms are inefficient: the size of the stack (in depth-first search) or queue (in breadth-first search) of visited but unexplored vertices can become linear in the number of nodes, giving a space bound of $\Theta(n \log n)$ ($\Theta(n)$ vertex names, each taking $\Theta(\log n)$ bits to store).

On the other end of the time-space spectrum, Savitch [Sav70] devised a deterministic algorithm for STCON that uses little space but superpolynomial time. His algorithm is conceptually simple: to discover whether there is a path from s to t, look for a path of length at most n. To look for a path of length at most k, choose a vertex, v, as the midpoint of the path, and look for a path of length at most $\lceil k/2 \rceil$ from s to v, and a path of length at most $\lfloor k/2 \rfloor$ from v to t. If such a path cannot be found, backtrack and try a different vertex in place of v. This recursive algorithm stores only a constant number of vertex names at each level of recursion, and recurses to depth $\Theta(\log n)$, since the path lengths are halved at each level. Therefore, it uses space $\Theta(\log^2 n)$. However, because it can try all possible choices of midpoints, it can use time exponential in its space bound, or $n^{\Theta(\log n)}$.

There are two other algorithms that solve s-t connectivity deterministically with essentially the same time and space performance as Savitch's algorithm: Cook and Rackoff [CR80] present an algorithm similar to Savitch's that solves STCON on their more restrictive Jumping Automata on Graphs (JAG) model. Nisan [Nis90] uses pseudorandom generators to "derandomize" the random walk algorithm of Aleliunas *et al.* [AKL$^+$79] (for more on the random walk algorithm, see Section 1.3.1 below). Nisan's algorithm is unique among the algorithms discussed so far, because it works only on undirected graphs.

Substantial progress on the time-space behavior of USTCON has been made since the results of Chapter 2 appeared [BR91]. First, Nisan [Nis92] improved his earlier pseudorandomness result to show that USTCON is solvable deterministically in polynomial time and $O(\log^2 n)$ space. Nisan's result was a breakthrough, since it showed that USTCON algorithms with small space bounds can perform asymptotically better than Savitch's algorithm. Unfortunately, the running time is something like n^{45},

making the algorithm extremely impractical. More recently, Nisan *et al.* [NSW92] show the first deterministic algorithm for USTCON with a space bound below Savitch's algorithm. Their algorithm uses pseudorandom generators and a contraction technique similar to that of the algorithms in Chapter 2 to solve undirected connectivity in space $O(\log^{1.5} n)$. For a further discussion of these results, see Section 4.2.

Before the work in this thesis, there were no deterministic s-t connectivity algorithms known between the time-space extremes of breadth- and depth-first search and Savitch's algorithm. So, for example, it might have been true that any algorithm using less space than depth-first search was forced to use as much time as Savitch's algorithm. The recent results of Nisan, and Nisan *et al.* help fill in some of the blanks in the time-space spectrum for USTCON, particularly for space bounds of $O(\log^2 n)$ or less. For space bounds above $O(\log^2 n)$ these results are less interesting, and, of course, they say nothing about the time-space behavior of STCON.

1.3.1 Randomized Algorithms for Connectivity

More is known about the time-space complexity of probabilistic algorithms for USTCON. The random walk algorithm of Aleliunas *et al.* [AKL+79] uses optimal space $\Theta(\log n)$ but is a factor of n slower than the time-optimal depth-first search ($\Theta(mn)$ instead of $\Theta(m+n)$). The algorithm works by repeatedly moving from a vertex to a new vertex, where the new vertex is chosen at random from the neighbors of the old vertex. Aleliunas *et al.* show that if s is connected to t, then such a walk of length $2m(n-1)$ taken from s will visit t with probability at least $1/2$.

Broder *et al.* [BKRU89] show a nearly smooth time-space tradeoff between the space-optimal random walk algorithm and the time-optimal depth- and breadth-first search algorithms. Note that both algorithms have a time-space product of $\Theta(nm \log n)$. For space $O(s)$, $\log n \leq s \leq n \log n$, Broder *et al.*'s algorithm uses $\Theta(m^2 \log^6 n/s)$ time. The algorithm scatters p pebbles ($s = p \log n$) on the graph at random according to the stationary distribution of the random walk, and repeatedly takes random walks of length $O(m^2 \log^3 n/p^2)$ from these pebbles, as well as from s and t. If a walk from a pebble discovers another pebble, the algorithm notes that the two pebbled vertices are in the same connected component. After $\Theta(\log n)$ walks from each pebble, if the algorithm has not determined that s and t are connected, the

pebbles are redistributed at random and new walks are taken. Broder *et al.* show that if s and t are connected, the algorithm will discover this fact with high probability within $\Theta(\log n)$ repetitions of the pebble placement and random walk process.

1.3.2 Nonuniform Algorithms for Connectivity

Considerable effort has been expended studying *universal traversal sequences* (UTS's). For simplicity, we will only consider UTS's for regular graphs — the definition can be extended to non-regular graphs as well. Given a d-regular graph, G, we can give the edges adjacent to each vertex a unique label from 0 to $d-1$. Note that each edge is labeled twice, once from each of its endpoints, and these two labels can be different. Given this labeling, a sequence of numbers between 0 and $d-1$ can be interpreted as a sequence of moves (a *traversal sequence*) between the vertices of G, where each number gives the edge that should be traversed from the current vertex. If, for any connected d-regular graph with n vertices, and any possible starting vertex, a given traversal sequence visits all vertices in the graph, then the traversal sequence is said to be *universal*.

While it is not difficult to show that UTS's for directed graphs are of exponential length, Aleliunas *et al.* [AKL+79] prove (nonconstructively) that universal traversal sequences of polynomial length exist for undirected graphs. Upper and lower bounds on the length of UTS's are known for various types of graphs [AAR90, BNBK+89, BRT89, HW89, Tom90]. Some authors show how to construct superpolynomial length sequences [BNS89, BNBK+89, Bri87, KPS88, Nis90], and Istrail shows how to construct polynomial length sequences for certain classes of graphs [Ist88, Ist90]. If one could construct polynomial length UTS's for arbitrary graphs in space $O(\log n)$, this would give an $O(\log n)$ space algorithm for USTCON, since a UTS can easily be followed in $O(\log n)$ space. Currently, though, universal traversal sequences show only that USTCON is in nonuniform $O(\log n)$ space.

1.4 Lower Bounds for s-t Connectivity

Only trivial lower bounds are known for the time-space behavior of s-t connectivity on general machines. Before presenting these lower bounds, we must define the most

common ways to encode a graph.

1.4.1 Graph Encodings

The two most common graph encodings are the edge list and the adjacency matrix.

- Adjacency matrix encoding — The edges in the graph are encoded using an $n \times n$ matrix, A, where $A(i,j)$ is 1 if there is an edge from i to j, and 0 otherwise. This encoding is more efficient for dense graphs (graphs with a large number of edges). For undirected graphs, the matrix is symmetric about the diagonal, of course.

- Edge list encoding — The graph is encoded using a vector of size n representing the vertices. Each vertex has associated with it a linked list of its neighbors in some arbitrary order. This encoding is more efficient for sparse graphs.

Either encoding can be converted to the other in deterministic logarithmic space. Given the edge list encoding of a graph, G, we can determine whether there is an edge in G from i to j in $O(\log n)$ space by reading the encoding and looking for an entry for j in i's edge list. The adjacency matrix for G can be constructed in $O(\log n)$ space by repeating this process for all n^2 entries in the matrix in order. The conversion from an adjacency matrix to an edge list encoding works in a similar way.

Other graph encodings besides adjacency matrix and edge list are possible, but not all encodings seem reasonable. For example, consider a connectivity matrix encoding: the graph is given as an adjacency matrix and an $n \times n$ matrix, C, where $C(i,j)$ is 1 if there is a path from i to j and 0 otherwise. Obviously, an algorithm which uses such an encoding on a machine with random access to its input can solve s-t connectivity in $O(\log n)$ time.

A definition of reasonable encodings for graphs seems difficult. We can, nevertheless, give a lower bound on the size of a graph encoding for USTCON.

Theorem 1.1 *Any graph encoding used to solve* USTCON *must be of size* $\Omega(n \log n)$*, assuming the encoding is independent of the values of* s *and* t*.*

Proof: For any graph, we can ask n^2 possible different s-t connectivity questions. Consider the possible (n^2)-tuples of answers to these queries. Some tuples are impossible — for example, for any v, the query "Is v connected to v?" can never be answered "no."

The number of valid tuples for an undirected graph with n vertices is equal to the number of ways of dividing the vertices into connected components, which can be expressed mathematically as:

$$\sum_{r=1}^{n} \left\{ {n \atop r} \right\},$$

where $\left\{ {n \atop r} \right\}$ is a Stirling number of the second kind [Coh78, pages 118-134], the number of possible ways to partition a set of n elements into r subsets. Closed form expressions for most Stirling numbers of the second kind are difficult to evaluate for variable n, but we can show the following useful lower bound:

$$\left\{ {n \atop r} \right\} \geq r^{n-r}.$$

Consider the partitions with r subsets where we place the first r elements into r different subsets, and the remaining $n - r$ elements in any subset. There is only one way to partition the first r elements, and r^{n-r} ways to place the remaining elements. Any such partition is one of the valid partitions of n elements into r sets. Furthermore, no valid partition is generated twice in this way, since the placement of the first r elements effectively "distinguishes" each subset. r^{n-r} is therefore a lower bound on the size of $\left\{ {n \atop r} \right\}$.

Letting $r = n/2$ in this bound, we can see that there are at least $(n/2)^{n/2}$ possible tuples of answers. Any graph encoding must be able to distinguish between all possible tuples, so any encoding scheme for graphs with n vertices must use at least $\log(n/2)^{n/2} = \Omega(n \log n)$ bits. □

1.4.2 Time Lower Bounds

There has been some previous work on lower bounds to determine monotone graph properties given an adjacency matrix encoding. See, for example, Rivest and

Vuillemin's and Kahn *et al.*'s work [RV76, KSS84]. This work is not directly applicable to s-t connectivity, since their bounds apply only to global graph properties (properties that are independent of vertex numbering). So, for example, their work gives an $\Omega(n^2)$ time lower bound to determine whether a graph contains a pair of connected vertices, but not to determine whether a specific pair of vertices is connected.

The following proposition gives a lower bound on the time requirements of USTCON for the adjacency matrix and edge list encodings. Note that the bounds given in this proposition, as well as the bounds in the corollaries that follow, apply to Random Access Machines (RAMs) as well as Turing machine.

Proposition 1.2 (a) *Any deterministic algorithm that solves* USTCON *on n vertex graphs uses time $\Omega(n^2)$ given an adjacency matrix encoding.*

(b) *Let m be a function of n such that $\alpha n \leq m \leq n^2/4$, for some constant $\alpha > 1$. Then, any deterministic algorithm that solves* USTCON *on n vertex, m edge graphs uses time $\Omega(m)$ given an edge list encoding.*

Proof: The basic idea of the proof is that if the algorithm always fails to examine a sufficiently large number of edges in every graph, it cannot distinguish between two similar graphs: one in which an edge e exists and there is a path from s to t, and another in which there is no edge e and therefore no path from s to t.

(a) Let G be the following graph with 4 or more vertices and $n-2$ edges: s is vertex number 1, with edges to vertices in the set $V_s = \{2, 3, \ldots, \lceil n/2 \rceil\}$. t is vertex number n, with edges to vertices in the set $V_t = \{\lceil n/2 \rceil + 1, \ldots, n-2, n-1\}$. Note that there are at least $\lfloor (n-2)/2 \rfloor^2 \geq n^2/16$ (because $n \geq 4$) possible pairs of vertices, (u, v), such that $u \in V_s$ and $v \in V_t$. Suppose M is an algorithm for USTCON, and M runs in fewer than $n^2/32$ steps on an adjacency matrix encoding. Given G, M must reject, since s is not connected to t. Furthermore, there must be one pair of vertices, $(u, v), u \in V_s, v \in V_t$, for which M failed to examine both $A(u, v)$ and $A(v, u)$, the entries in the adjacency matrix that determine whether there is an edge between u and v. Let G' be the same graph as G, except with an edge from u to v. M will also reject G', since G' and G differ only in two entries in the adjacency matrix, neither of which is examined by M. But s is connected to t in G', so we have a contradiction.

(b) Fix m to be a function of n such that $\alpha n \leq m \leq n^2/4$, for some constant $\alpha > 1$. Suppose M is an algorithm for USTCON, and M runs in fewer than $\lfloor (\alpha - 1)m/2\alpha) \rfloor /2$ steps on n vertex, m graphs using an edge list encoding. Let G be a graph similar to the graphs in part (a): it has 6 or more vertices, with $\lceil n/2 \rceil - 1$ edges from s to the vertices in V_s, and $\lfloor n/2 \rfloor - 1$ edges from t to the vertices in V_t. Add other edges to G, E_{V_s}, between vertices in V_s, and E_{V_t}, between vertices in V_t, so that G has m edges in total, and so that $|E_{V_s}|$ and $|E_{V_t}|$ differ by at most one. The fraction of edges in E_{V_s} and E_{V_t} is at least $(\alpha - 1)/\alpha$, since only $n - 2$ edges are not in these two sets. This means the number of edges in either set is at least $\lfloor (\alpha - 1)m/2\alpha) \rfloor$. Given G, M must reject, and furthermore, M will always read fewer than half the edges in E_{V_s} and fewer than half the edges in E_{V_t}. Therefore, there must be some pair of edges $e_1 = \{v_1, v_2\} \in E_{V_s}$ and $e_2 = \{v_3, v_4\} \in E_{V_t}$, such that M doesn't read either entry of either edge (that is, it doesn't read v_1's entry in v_2's edge list, v_2's entry in v_1's list, v_3's entry in v_4's list, or v_4's entry in v_3's list).

Let G' be the same graph as G, except that e_1 and e_2 are *crossed*. To cross the edges, replace e_1 and e_2 with $e_1' = \{v_1, v_4\}$ and $e_2' = \{v_3, v_2\}$, making sure to put the entries for these new edges in the positions formerly held by e_1 and e_2 in the respective edge lists. M will also reject G', since G' and G differ only in the entries for e_1' and e_2' in the edge lists of v_1, v_2, v_3, and v_4, none of which is examined by M. But s is connected to t in G', so we have a contradiction.

Note that this proof requires the ordering of the edge lists to be somewhat arbitrary. For example, if all edge lists were sorted by the vertex number of the neighbors, one could not guarantee that the entries for the crossed edges occurred in the same positions in the edge lists as the old edges.

□

The same arguments can be used directly to give lower bounds for undirected s-t nonconnectivity (the complement of USTCON) on nondeterministic machines:

Corollary 1.3 (a) *Any nondeterministic algorithm that solves undirected s-t nonconnectivity on n vertex graphs uses time $\Omega(n^2)$ given an adjacency matrix encoding.*

(b) *Let m be a function of n such that $\alpha n \leq m \leq n^2/4$, for some constant $\alpha > 1$. Then, any nondeterministic algorithm that solves undirected s-t nonconnectivity on n vertex, m edge graphs uses time $\Omega(m)$ given an edge list encoding.*

Proof:[of part (a)] Suppose M is a nondeterministic algorithm for undirected s-t nonconnectivity that runs on an adjacency matrix encoding, and M uses fewer than $n^2/32$ steps. Given G from Proposition 1.2[a], M must accept. Let Q be an accepting path in M on input G. On path Q there must be a pair of vertices, $(u, v), u \in V_s, v \in V_t$, for which M failed to examine both $A(u, v)$ and $A(v, u)$. Let G' be the same graph as G, except with an edge from u to v. s is connected to t in G', but M will also accept G' along the same path, Q, since G' and G only differ in two entries in the adjacency matrix, neither of which is examined along path Q. Again, we have a contradiction.

The proof of part [b] is by a similar modification to the proof of Proposition 1.2[b]. □

Note that the same lower bound *cannot* be shown for USTCON on nondeterministic machines, since to show connectivity, an algorithm need only find a path, which takes nondeterministic time $O(n \log n)$. Finally, note that these proofs for undirected s-t connectivity and nonconnectivity extend to directed s-t connectivity and nonconnectivity as well.

Corollary 1.4 (a) *Any deterministic algorithm that solves* STCON *or nondeterministic algorithm that solves directed s-t nonconnectivity on n vertex graphs uses time $\Omega(n^2)$ given an adjacency matrix encoding.*

(b) *Let m be a function of n such that $\alpha n \leq m \leq n^2/4$, for some constant $\alpha > 1$. Then, any deterministic algorithm that solves* STCON *or nondeterministic algorithm that solves directed s-t nonconnectivity on n vertex, m edge graphs uses time $\Omega(m)$ given an edge list encoding.*

1.4.3 Space Lower Bounds

It is easy to show a space lower bound of $\Omega(\log n)$ on a RAM — the above time lower bound proofs show that an algorithm for USTCON must read a constant fraction of

its input, and therefore needs space $\Omega(\log n)$ on a RAM to index the registers holding the input. For Turing machines, a space lower bound can be derived for the edge list and adjacency matrix encodings using a crossing sequences argument. For an introduction to crossing sequence arguments, see Hopcroft and Ullman [HU79, pages 38, 314-315] [HU69], and Hennie [Hen65].

Proposition 1.5 *Any deterministic Turing machine that recognizes* USTCON *using the adjacency matrix or edge list encoding uses space* $\Omega(\log n)$.

Proof: Assume without loss of generality that the graph has $4k + 2$ vertices. Consider the following family of graphs. Besides s and t, each graph contains vertices $u_1, u_2, \ldots, u_{2k}$ and $v_1, v_2, \ldots, v_{2k}$. Each graph has $4k$ edges. $\{u_i, v_i\}$ is an edge for all $1 \leq i \leq 2k$. In addition, s has edges to half of the u vertices, and t has edges to half of the v vertices. Let $C_s = \{i | \{s, u_i\} \in E\}$, and $C_t = \{i | \{t, v_i\} \in E\}$ (in other words, C_s and C_t contain the numbers of the u and v vertices that are neighbors of s and t). There is a path from s to t if and only if $C_s \cap C_t \neq \emptyset$.

With either encoding, we can arrange the vertices so that all edges involving s appear before edges involving t: s will be vertex number 1, t will be vertex number $4k + 2$, and, for all $1 \leq i \leq 2k$, u_i will be vertex number $i + 1$ and v_i will be vertex number $2k + i + 1$. The input can then be written as $\langle e_s\ e_t \rangle$, where e_s is a substring of the input containing all information involving the k edges from s, and e_t is a substring containing all information involving the k edges from t. Examine the behavior of a Turing machine, M, that solves USTCON. In particular, we are interested in the configurations of M when it *crosses* the boundary between e_s and e_t, either by moving the input head right from the rightmost character of e_s, or moving the input head left from the leftmost character of e_t. Each time such a crossing occurs, we will write down the current configuration of M, including the contents of the work tape, the positions of any heads, and the current state. The sequence of all these configurations in a computation of M is a *crossing sequence.*

Intuitively, the crossing sequence must somehow encode all information about each half of the input, and therefore gives a lower bound on the space (and the time) requirements of the machine. Suppose we have two different disconnected graphs in the family, one encoded $\langle e_s\ e_t \rangle$ and the other $\langle e'_s\ e'_t \rangle$. Then M must produce different

crossing sequences for each input. Suppose it did not. Then examine the behavior of M on the graph $\langle e_s\ e'_t \rangle$. M will generate the same crossing sequence on this input as on the other two, so M will give the same answer for this s-t connectivity problem as for the other two. But this graph is connected (for every C_s there is only one possible C_t with an empty intersection, and this C_t is different for each C_s). By contradiction, then, M produces a different crossing sequences for each disconnected graph.

There are $\binom{2k}{k} = 2^{\Omega(n)}$ such disconnected graphs, so there must be $2^{\Omega(n)}$ such crossing sequences. If M uses space S, then there are only $2^{O(S)}$ possible configurations at each crossing: there are two possible input head positions, a constant number of possible states, $O(S)$ possible worktape head positions, and $2^{O(S)}$ possibilities for the contents of the worktape. Furthermore, M can cross c only $2^{O(S)}$ times, since if it repeated a configuration while crossing either to the right or left, it would loop forever. Therefore, the number of possible crossing sequences is $2^{O(S)\cdot 2^{O(S)}}$, which must be more than the number of possible disconnected graphs. For suitable constants c_1 and c_2, and n sufficiently large, this gives us:

$$\begin{aligned} 2^{c_1 S \cdot 2^{c_1 S}} &\geq 2^{c_2 n} \\ c_1 S \cdot 2^{c_1 S} &\geq c_2 n \\ S &= \Omega(\log n) \end{aligned}$$

□

As in Section 1.4.3, the argument carries through for nondeterministic undirected s-t nonconnectivity and for the directed versions of these problems:

Corollary 1.6 *Any nondeterministic Turing machine that recognizes undirected s-t nonconnectivity using the adjacency matrix or edge list encoding uses space $\Omega(\log n)$.*

Corollary 1.7 *Any deterministic Turing machine that recognizes* STCON *using the adjacency matrix or edge list encoding uses space $\Omega(\log n)$. Any nondeterministic Turing machine that recognizes directed s-t nonconnectivity using the adjacency matrix or edge list encoding uses space $\Omega(\log n)$.*

Some tighter lower bounds are known for s-t connectivity on restricted models of computation. Cook and Rackoff [CR80] show an $\Omega(\log^2 n/\log\log n)$ space lower bound for STCON on their JAG model, closely matching Savitch's upper bound. Berman and Simon [BS83] extend this result to give a similar lower bound on a randomized version of the JAG. Beame *et al.* [BBR+90] give time-space lower bounds for USTCON on more restricted versions of the JAG model. Finally, Tompa [Tom82] shows that for certain natural approaches to solving STCON, performance degrades sharply with decreasing space: space $o(n)$ implies superpolynomial time, and space $n^{1-\epsilon}$ for fixed $\epsilon > 0$ implies time $n^{\Omega(\log n)}$, essentially as slow as Savitch's algorithm. It has been conjectured [Coo79] that no deterministic STCON algorithm can run in simultaneous polynomial time and polylogarithmic space. The results of Chapter 3 do not disprove this conjecture, but they do show that the behavior elicited from certain algorithms by Tompa is not intrinsic to the problem.

1.5 Logspace Reducibility

Frequently in this thesis, we will want to show that a language (or function) is *logspace reducible* to another language. The following section is based on Hopcroft and Ullman's discussion of logspace many-one reducibility [HU79, pages 322-324], and Ladner and Lynch [LL76].

Definition: A language, L_1, is logspace many-one reducible to a language, L_2, (denoted $L_1 \leq_{\log} L_2$) if there exists a deterministic machine, M, which, given an input, $y, |y| = n$, can, using $O(\log n)$ space, produce an output, x, such that $x \in L_2$ if and only if $y \in L_1$. The output x is written on a one-way, write-only output tape that is not considered when determining the space bound.

Definition: A function, $f(x)$, (alternately, a language, L_1) is logspace Turing reducible to a language, L_2, (denoted $f(x) \leq_{\log,T} L_2$, or $L_1 \leq_{\log,T} L_2$) if there exists a deterministic machine, M, which, given an oracle for L_2, can, for any $y, |y| = n$, compute $f(y)$ (decide whether $y \in L_1$) using $O(\log n)$ space, not counting the space needed to write down the oracle queries or the function value. The function value is written on a one-way, write-only output tape. The oracle queries are written on a one-way, write-only query tape that is erased when a query is answered.

For our purposes, many-one and Turing reductions are almost always equivalent, since most of our reductions will deal with languages in NL, and Immerman's and Szelepcsényi's proofs that NL is closed under complementation [Imm88, Sze88] imply that NL is also closed under logspace Turing reductions. Still, for simplicity, we will try to use many-one reductions whenever possible. Unless otherwise specified, for the rest of this thesis, logspace reducibility will mean logspace many-one reducibility.

Intuitively, $L_1 \leq_{\log} L_2$ if we can compute L_1 in logarithmic space by calling a subroutine for L_2. As an example of a log space reduction, it's easy to see that USTCON $\leq_{\log}$ STCON. Simply take every edge $\{u, v\}$ in the undirected graph and replace it with a pair of directed edges, (u, v) and (v, u). The replacement can be done in space $O(\log n)$, and there is a path from s to t in the directed graph if and only if s was connected to t in the undirected graph. As pointed out in Section 1.1.1, page 3, determining whether the converse (STCON $\leq_{\log}$ USTCON) is true is an important open problem.

Given a log space reduction, $L_1 \leq_{\log} L_2$, we would like to be able to show that a logarithmic space algorithm for L_2 implies the existence of a logarithmic space algorithm for L_1. The obvious approach is to use the machine M from the reduction to transform an input to L_1 into an input to L_2 in $O(\log n)$ space, and feed this input to the $O(\log n)$ space machine for L_2 (note that the length of the input to the machine for L_2 is $n^{O(1)}$, since M can only use polynomial time). Unfortunately, the input to L_2 could be of length $\omega(\log n)$, as in the example above, where the length of the input to the STCON algorithm is about the same length as the USTCON input.

To solve this problem, we will replace the machine for L_2 with a machine that asks for one bit of its input at a time. It can, for example, read its input all the way through, or read some fraction of the input, then start over again at the beginning, or even skip about in the input. M can then be replaced by a similar machine, M', that generates any specified bit of M's original output. It is not difficult to see that M and the machine for L_2 can always be modified to behave in this way. The modified machines can be merged to get a $O(\log n)$ space algorithm for L_1 — each machine uses space $O(\log n)$, and they use only $O(\log n)$ space to communicate. Multiple reductions can also be composed in the same way. See Hopcroft and Ullman [HU79, Lemmas 13.2 and 13.3] for a detailed proof of this result. Chapter 4 uses this method frequently to compose many space bounded reductions into a single reduction.

Chapter 2

TIME-SPACE TRADEOFFS FOR UNDIRECTED S-T CONNECTIVITY

2.1 Introduction

This chapter presents three deterministic algorithms for undirected s-t connectivity that achieve sublinear space and polynomial time simultaneously. The algorithms provide a nearly smooth tradeoff for USTCON between time-efficient, space-intensive algorithms, such as depth- and breadth-first search, and time-intensive, space-efficient algorithms such as Savitch's, and Cook and Rackoff's (see Section 1.3). The first algorithm, called the *simple* algorithm below, uses space $O(s)$, $n^{1/2}\log n \leq s \leq n\log n$, and runs in time $O(((m+n)n^2\log^2 n)/s)$. The second (the *recursive* algorithm) is a generalization of the first, using space $O(\lambda n^{1/\lambda}\log n)$, for $2 \leq \lambda \leq \log n$, and time $n^{O(\lambda)}$. It can use as little as $\Theta(\log^2 n)$ space, but its running time becomes superpolynomial whenever its space is constrained to $n^{o(1)}$. The third algorithm (the *batched* algorithm) is a more time-efficient variant of the first: for space $O(s)$, $n^{1/2}\log n \leq s \leq n\log n$, it uses time $O((m+n)((n/s)^2\log^3 n)(\log((n\log n)/s)))$. When $s = \Theta(n\log n)$, the time bound is $O((m+n)\log n)$, only a factor of $\log n$ worse than the time-optimal bound of depth- and breadth-first search.

The remainder of the chapter is organized as follows. Sections 2.2, 2.3 and 2.4 present the simple, recursive, and batched algorithms, respectively. Section 2.5 presents some notes and concluding remarks.

For simplicity, when discussing the space used by our algorithms below, we will often give the space used in terms of *registers*, where each register holds $O(\log n)$ bits, enough to specify the name of a vertex, for example.

2.2 The Simple Algorithm

The first algorithm depends upon a space-bounded breadth-first search subroutine (bbfs) to find small sets of connected vertices. Such a routine is easy to implement; basically, it is a standard breadth-first search routine modified to quit as soon as a certain number of vertices, b, have been found, or the component in which it is started is exhausted. Using this routine, a single vertex, v, can be used to implicitly mark a *neighborhood* of up to b vertices, namely, the vertices, including v itself, found by starting the bbfs routine at v. Let bbfs(v, b) denote this set, and define a *full* neighborhood to be one that contains b vertices. All our algorithms depend on repeatedly recomputing the neighborhoods of vertices; such recomputation is common when attempting to use a limited amount of space. (See [Pip80], for example.)

Using the bbfs routine, a few special cases can be eliminated with a little initial work. Begin by generating the neighborhoods of s and t. If they intersect, we are finished: s and t are connected. Next, check that both neighborhoods are full. If bbfs fails to find b vertices connected to a given vertex, the vertex must belong to a *small* connected component, i.e. one with fewer than b vertices. Thus, assuming s's and t's neighborhoods are disjoint, if either neighborhood is not full, the two vertices can not be connected. Furthermore, if both neighborhoods are full, then any other vertex whose neighborhood is not full cannot be connected to s or t, and can safely be ignored.

If the neighborhoods of s and t are full and disjoint, the algorithm identifies and stores the names of a certain set of vertices, L, called *landmarks*. One goal of the algorithm is to find enough landmarks, l, so that their associated neighborhoods, bbfs(l, b), nearly cover the graph, and so finding connected components will be quick. On the other hand, the set of landmarks must be small enough that the space constraint is not violated. To achieve these goals the algorithm constructs L satisfying the following three conditions:

- s and t are both members of L.

- The neighborhoods of the landmarks in L are full and pairwise disjoint. That

is,

$$\forall\, l \in L, \qquad |\mathsf{bbfs}(l, b)| = b, \text{ and}$$

$$\forall\, l_1, l_2 \in L, \qquad (l_1 \neq l_2) \Rightarrow \mathsf{bbfs}(l_1, b) \cap \mathsf{bbfs}(l_2, b) = \emptyset.$$

This insures that there can be no more than n/b landmarks.

- L is a maximal set satisfying the above properties. Thus the neighborhood of any vertex not in L either is not full, or has at least one vertex in common with the neighborhood of some landmark. That is,

$$\forall\, v \notin L\ (|\mathsf{bbfs}(v, b)| < b)\ \vee\ (\exists\, l \in L \text{ s.t. } \mathsf{bbfs}(v, b) \cap \mathsf{bbfs}(l, b) \neq \emptyset).$$

The set L can be built in one pass through all the vertices. Begin by adding s and t to L. For every other vertex, v, generate the neighborhood of v; if it is of size b and disjoint from the neighborhoods of all previous landmarks, then add v to L.

Once the set of landmarks is constructed, we determine which landmarks are in the same connected component. Define a function $\mathsf{cl}(v)$ on the vertices to denote the "closest" landmark to a given vertex. If the vertex is in a small component, cl simply returns a special value, "SMALL", indicating that fact. Otherwise, it returns the lowest numbered landmark whose neighborhood intersects the neighborhood of the vertex:

$$\begin{aligned} \mathsf{cl}(v) &= \begin{cases} \text{"SMALL"} & \text{if } |\mathsf{bbfs}(v, b)| < b, \\ \min I & \text{otherwise, where} \end{cases} \\ I &= \{l \in L \mid \mathsf{bbfs}(l, b) \cap \mathsf{bbfs}(v, b) \neq \emptyset\}. \end{aligned}$$

By the definition of L, $I \neq \emptyset$ if $|\mathsf{bbfs}(v, b)| = b$. Note that if v is a landmark, then $\mathsf{cl}(v) = v$.

The function cl induces a partition on the vertices, with one block per landmark, plus one for all vertices in small components, which necessarily are not connected to either s or t. Furthermore, we know that for each landmark, l, all vertices in the block $\{v \mid \mathsf{cl}(v) = l\}$ are connected. We can use this information to discover which

landmarks are in the same connected component: if there is an edge $\{u, v\}$ such that u is in one block, and v is in another, then we know that the landmarks corresponding to these two blocks are connected. By considering each edge in turn, the algorithm is able to determine the connectivity information for all landmarks.

Connectivity is encoded by constructing sets of landmarks, where two landmarks are members of the same set only if they are known to be in the same connected component. The sets can be manipulated efficiently using Union-Find subroutines; although it does not improve the asymptotic running time of the algorithm, we will use the very fast version of the Union-Find subroutines with weighted union and path compression [Tar75]. We will call these sets *union-find sets*. Begin by constructing a singleton set containing each landmark, plus an extra set containing the special value, "SMALL". For all the edges $e = \{u, v\}$, perform a Union on Find($\mathsf{cl}(u)$) and Find($\mathsf{cl}(v)$). Using the reasoning in the previous paragraph, if the two vertices are in separate blocks before the Union, we now know that the corresponding landmarks are connected, and hence these sets should be joined. The following lemma asserts that this process is sufficient to determine the connectivity between the landmarks:

Lemma 2.1 *After following the above procedure, any two landmarks, in particular s and t, will end in the same union-find set if and only if they are in the same connected component.*

Proof: First, we will show by induction on k that, for any path of length $k-1$ in a component with b or more vertices, $v_1, v_2, \ldots, v_k$, after processing the $k-1$ edges $\{v_1, v_2\}$, $\{v_2, v_3\}$, ..., $\{v_{k-1}, v_k\}$, the landmarks $\mathsf{cl}(v_1), \mathsf{cl}(v_2), \ldots, \mathsf{cl}(v_k)$ will all be in the same union-find set. This is trivially true when $k = 1$. For $k > 1$, consider the edge in the path that is processed last, $\{v_i, v_{i+1}\}$. By the induction hypothesis, just before edge $\{v_i, v_{i+1}\}$ is processed, $\mathsf{cl}(v_1), \ldots, \mathsf{cl}(v_i)$ are all in one union-find set, as are $\mathsf{cl}(v_{i+1}), \ldots, \mathsf{cl}(v_k)$. Then Union(Find($\mathsf{cl}(v_i)$),Find($\mathsf{cl}(v_{i+1})$)) will join these two sets (if necessary). Thus, any two connected landmarks will end in the same set.

For the other direction, we show by induction on the number k of edges processed that for all pairs of vertices (u, v), if Find($\mathsf{cl}(u)$) = Find($\mathsf{cl}(v)$) then either u and v are connected, or both are in small components. When $k = 0$ this easily follows from the definition of cl and the initial construction of the union-find sets. For $k > 0$, let

$\{x, y\}$ be the kth edge processed. If Find(cl(u)) = Find(cl(v)) holds after $\{x, y\}$ is processed, but not before, then it must be that Find(cl(u)) and Find(cl(v)) were the two (distinct) sets joined when processing $\{x, y\}$. Without loss of generality, suppose Find(cl(u)) = Find(cl(x)) and Find(cl(v)) = Find(cl(y)) before $\{x, y\}$ is processed. Then, by induction, u and x are connected, as are v and y, and hence u and v are connected through edge $\{x, y\}$. □

A detailed "pseudocode" version of the main routine algorithm is presented in Figure 2.1. A pseudocode version of the cl function is presented in Figure 2.2.

2.2.1 Analysis of the Simple Algorithm

The bounded breadth-first search routine must check whether the endpoint of each edge explored has been visited or not. Standard breadth-first search routines do this in constant time using a flag for each vertex, but this requires too much space. Instead, our bounded breadth-first search routine looks up the endpoint of the edge in the list of vertices already known to be in the neighborhood. Lookup and insertion in a list can be accomplished in time $O(\log b)$ by using a balanced search tree [Knu73]. Thus, overall, bbfs uses time $O(b^2 \log b)$ with $O(b)$ registers — it can explore only b vertices, which can have only $O(b^2)$ edges among them.

Constructing the intersection between two neighborhoods of size b can be done in time $O(b)$ using $O(b)$ registers since the vertices are sorted in the neighborhoods' balanced search trees. If l is defined to be the number of landmarks, then the cl routine runs in time $O(b^2 l \log b)$ using $O(b)$ registers.

The loop to find the landmarks in Figure 2.1 takes time $O(b^2 ln \log b)$, using $O(b+l)$ registers. The Union-Find operations take time $O(m\alpha(m))$ [Tar75] and use $O(l)$ registers. The cl routine is called $2m$ times, each call taking time $O(b^2 l \log b)$, so connecting the union-find sets and finding the landmark set dominates the algorithm's running time, taking time $O(b^2 l(m+n) \log b)$.

Since the bbfs procedure finds b vertices associated with each landmark, at most n/b landmarks can be found, giving a total space bound of $O((b + n/b) \log n)$ and a time bound of $O(bn(m+n) \log b)$. When $b = \sqrt{n}$, space is minimized, and the running time is $O((m+n)n^{1.5} \log n)$. Increasing b from $\sqrt{n}$ increases both the space and time

```
Algorithm Ustcon (integer: b);
                                                        {1 ≤ b ≤ n}
  generate bbfs(s, b) and bbfs(t, b).
  if s's neighborhood overlaps t's then return (CONNECTED);
  if either neighborhood is not full then return (NOT CONNECTED);
  Initialize the set of landmarks to {s, t};
  for all vertices, v do begin                        {find the landmarks}
    generate bbfs(v, b);
    if v's neighborhood is not full then Not a landmark. Go to next vertex.
    for all landmarks, l do begin
      generate bbfs(l, b);
      if v's neighborhood overlaps l's then Not a landmark. Go to next vertex.
    end;
    Add v to the set of landmarks.
  end;
  Create a singleton Union-Find set containing each landmark, plus one
    containing the special value "SMALL" for small components.
  for all edges, e = {u, v} do begin
    Union(Find(cl(u)), Find(cl(v)));
  end;
  if Find(s) = Find(t) then return (CONNECTED);
  else return (NOT CONNECTED);
end Ustcon.
```

Figure 2.1: The main routine of the simple algorithm

used, so that range is uninteresting. In the other direction, though, as b is decreased from $\sqrt{n}$, the time of the algorithm decreases as the space increases. For $1 \leq b \leq \sqrt{n}$, the algorithm uses space $s = O((n \log n)/b)$ and time $O((m+n)n^2 \log n \log b/s)$, for a time-space product of $O((m+n)n^2 \log n \log b)$. This is off by a factor of at most $n \log n$ from the best known algorithms (depth-first or breadth-first search, and random walk;

```
procedure cl (vertex v): vertex;
                                        {Return the "closest" landmark to v}
  generate bbfs(v, b);
  if v's neighborhood is not full then return ("SMALL");
  for all landmarks, l, in order do begin
    generate bbfs(l, b);
    if v's neighborhood overlaps l's then return (l);
  end;
end cl.
```

Figure 2.2: The cl function of the simple algorithm

see Broder *et al.* [BKRU89]), and a factor of n when the space approaches its upper bound ($O(n \log n)$).

In summary, we have shown the following.

Theorem 2.2 *For any $n^{1/2} \log n \leq s \leq n \log n$, the simple algorithm, presented above, solves* USTCON *for arbitrary n vertex, m edge graphs in space $O(s)$, and time $O(((m+n)n^2 \log^2 n)/s)$.*

2.3 The Recursive Algorithm

The simple algorithm points the way to a more general algorithm that uses less space. Consider the bounded breadth-first search routine: it isn't necessary to use breadth-first search at all; any deterministic graph searching algorithm that can find b connected vertices within the same time and space constraints could be used instead. Suppose the simple algorithm itself is used as the search routine. Superficially, the simple algorithm uses b registers to characterize b^2 vertices. Therefore, it might be possible to use this algorithm, or one like it, to find $n^{2/3}$ neighbors of a vertex using only $n^{1/3}$ registers. Then only $n^{1/3}$ landmarks, each characterizing $n^{2/3}$ vertices, would be needed, and the space bound would be reduced to $O(n^{1/3})$ registers. And, of course,

the technique could be repeated, to ultimately reduce the space to $O(cn^{1/c} \log n)$ for any fixed constant c while still maintaining a (possibly very high) polynomial running time. (The factor of c covers the cost of the implied c levels of recursion.)

To use the simple algorithm in this way, it needs to be changed from a global search routine to a local one. That is, instead of finding *any* b landmarks, it should find b landmarks in the same connected component. One way to accomplish this is to add an extra step to the algorithm when it is searching for new landmarks: a vertex can only be added to the landmark set if its set of vertices is disjoint from the other landmarks' sets *and* if it can be shown to be in the same connected component as one of the other landmarks. We actually impose a stronger condition, very roughly that the new landmark's set must be "close enough" to some old landmark's set that they both overlap the set of some third vertex. Since this test for connectivity is efficient, this modified version of the simple algorithm can be called recursively to build large sets of connected vertices from smaller sets using a small amount of space and time.

The recursive algorithm is parameterized by an integer, λ, $2 \leq \lambda \leq \log n$, which indicates the approximate depth of recursion the algorithm should use (the actual depth of recursion will turn out to be $\lambda - 1$). The bound on space for any given routine (i.e., exclusive of recursive calls) will be $O(b)$ registers, where $b = \left\lceil n^{1/\lambda} \right\rceil$, and therefore the total space bound for the algorithm will be $O(b\lambda)$ registers.

Generalizing the notion of a neighborhood in the simple algorithm, for each $k \geq 0$, each vertex v has a (k)-*neighborhood*, denoted $\mathsf{nv}(v, k)$, of size at most b^k. A (k)-neighborhood is *full* if it is of size exactly b^k. For $k = 0$, we define $\mathsf{nv}(v, k) = \{v\}$. To represent a (k)-neighborhood succinctly for $k \geq 1$, the (k)-neighborhood of v has associated with it a (k)-*landmark set*, a set of at most b vertices, denoted $L_{v,k}$; the (k)-neighborhood of v is the union of the $(k-1)$-neighborhoods of the members of $L_{v,k}$. Informally, the (k)-landmark set for v is $\mathsf{bbfs}(v, b)$ when $k = 1$, and when $k > 1$, it is a maximal set of landmarks of disjoint $(k-1)$-neighborhoods connected to v, where connectivity is tested based on the following property. We say u *is* (k)-*adjacent to* v if there is an edge $\{u, w\}$ such that w's (k)-neighborhood overlaps v's (k)-neighborhood.

More formally, the (k)-landmark set for v has the following properties, similar to the properties of L in the simple algorithm (Section 2.2, page 19).

In the base case, when $k = 1$, $L_{v,k} = \mathsf{bbfs}(v, b)$. When $k > 1$, $L_{v,k} = \{l_1 = v, l_2, \ldots, l_j\}$ is an ordered set of vertices containing v that is maximal in that it satisfies the following properties, but no vertex l_{j+1} can be added without violating one of these properties:

- $L_{v,k}$ contains at most b landmarks, i.e.

$$|L_{v,k}| \leq b.$$

- The $(k-1)$-neighborhoods of the vertices in $L_{v,k}$ are pairwise disjoint. That is,

$$\forall l_i, l_{i'} \in L_{v,k},\ (l_i \neq l_{i'}) \Rightarrow \mathsf{nv}(l_i, k-1) \cap \mathsf{nv}(l_{i'}, k-1) = \emptyset$$

- Every landmark l_i in $L_{v,k}$ (except v) is $(k-1)$-adjacent to an earlier landmark $l_{i'}$, $i' < i$.

  This last property guarantees that the landmarks are all in the same connected component.

Constructing $L_{v,k}$ is a simple task: cycle through the edges, searching for one with an endpoint that satisfies the above properties for $L_{v,k}$ (i.e., an edge $\{u, w\}$ such that u's neighborhood is disjoint from the previously chosen landmarks' neighborhoods, and w's neighborhood overlaps one of them). When such an edge is found, add u to $L_{v,k}$ and repeat the process until either enough landmarks have been discovered or another cannot be found.

To understand the discussion below, it will be helpful to remember that v's (k)-neighborhood is always a subset of v's $(k+1)$-neighborhood.

At the topmost level, the algorithm proceeds much like the simple algorithm: after eliminating certain special cases, it constructs a maximal set of (not necessarily connected) *global* landmarks having full, pairwise disjoint $(\lambda-1)$-neighborhoods, then examines the edges of the graph to discover which landmarks are in the same connected component. Initially, s and t are in this landmark set; every other vertex is tested in turn, and added to the landmark set if its $(\lambda-1)$-neighborhood is full and disjoint from the previous landmarks' sets. There can be at most b such global

landmarks. The main difference between this and the simple algorithm is that instead of using bbfs to find the neighborhoods, the algorithm uses $\mathsf{nv}(\cdot, \lambda - 1)$. Note that we could simplify the algorithm conceptually by using nv at the topmost level to find a global landmark set that is guaranteed to be connected, e.g. $\mathsf{nv}(s, \lambda)$. However, guaranteeing connectivity turns out to take more time asymptotically, so the faster method of building union-find sets is used at the top level. This also insures that the simple algorithm is a special case of this algorithm, i.e., the case $\lambda = 2$.

The search procedure, nv, raises a problem that was not present with a simple breadth-first search: what if the search for a vertex's $(\lambda - 1)$-neighborhood fails to find enough $(b^{\lambda-1})$ vertices? All our algorithms dismiss a vertex whose search is unsuccessful, because a lower bound on the size of the global landmarks' neighborhoods is necessary to insure an upper bound on the number of global landmarks. As mentioned in Section 2.2 (page 19), when using breadth-first search, it is clear that any vertex whose bbfs set is too small belongs to a small component, and is therefore not connected to s or t. With this more complex search procedure, however, we cannot be certain that the failure of the search routine for a given vertex, v, implies that v belongs to a small component and can therefore be disregarded. In fact, it seems likely that, for any given k, a certain number of vertices in v's component will never be part of the (k)-neighborhood of v, and also that searches begun at different vertices could have wildly varying results.

Consider Figure 2.3 depicting a connected graph, with S_v the $(\lambda - 2)$-neighborhood of v, S_1 a set of size less than $b^{\lambda-2}$, and S_2 the rest of the graph, not directly connected to S_1. Suppose the algorithm is trying to find the $(\lambda - 1)$-neighborhood of v, has generated S_v, and is now looking for $b - 1$ other $(\lambda - 2)$-neighborhoods. It will not find a large enough set in S_1, and may never be able to use these vertices, which may mean that v's $(\lambda - 1)$-neighborhood might not be full. Intuitively, though, a search for a $(\lambda - 1)$-neighborhood begun at another vertex "closer to" or "farther from" S_1 might succeed because it is able to use some or all of S_1's vertices. It seems that the search at v, then, might be unsuccessful, while the search at a nearby vertex could have the opposite result. If v is dismissed as irrelevant by the algorithm, how can we be sure this is correct?

To solve this problem, we prove the following lemma, which shows that the process of building $L_{v,k}$ does not fail prematurely. Intuitively, the lemma says that if a vertex

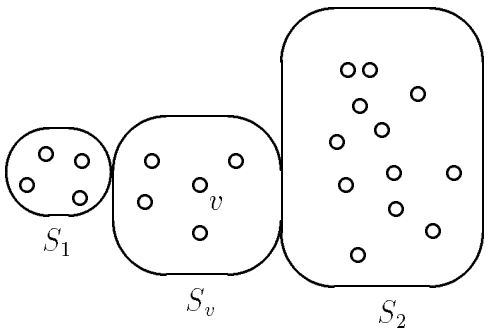

Figure 2.3: A graph where the search might fail

v's connected component isn't "big" enough to allow v a full $(k+1)$-neighborhood, then it isn't big enough to contain even a (k)-neighborhood, full or not, disjoint from v's $(k+1)$-neighborhood. This lemma can be used to show that vertices at which a search fails are not actually a problem.

Lemma 2.3 *For any vertex v, if v's $(k+1)$-neighborhood is not full, then it overlaps the (k)-neighborhood of every vertex in v's connected component.*

Proof: The proof proceeds by induction on k. Let C be the connected component containing v. For the basis, $k = 0$, observe that if v's (1)-neighborhood isn't full, then by the properties of bbfs, v's (1)-neighborhood contains all of C. For the induction step, $k > 0$, for the sake of contradiction suppose that v's $(k+1)$-neighborhood is not full, and there are vertices in C whose (k)-neighborhoods are disjoint from v's $(k+1)$-neighborhood. There are two possible ways in which v's $(k+1)$-neighborhood could fail to be full. First, it could be that $L_{v,k+1}$, the $(k+1)$-landmark set of v, has b elements, but one of the $(k+1)$-landmarks, l, has a (k)-neighborhood that is not full. But this cannot be true, since, by induction, the (k)-neighborhood of l would overlap the $(k-1)$-neighborhoods of all other members of $L_{v,k+1}$, contradicting the disjointness property of the $(k+1)$-landmark set. Second, $L_{v,k+1}$ may have fewer than b elements. Consider all vertices in C whose (k)-neighborhoods are disjoint from v's $(k+1)$-neighborhood. Among all such vertices there must be a vertex w whose distance, d, from v is minimal. Consider a path of length d between v and

w. By the minimality of d, w's predecessor on the path has a (k)-neighborhood that overlaps v's $(k+1)$-neighborhood, so w is (k)-adjacent to some member of $L_{v,k+1}$. But this contradicts the maximality of $L_{v,k+1}$, since w could be added to $L_{v,k+1}$. □

Given this lemma, we can show the algorithm is correct. As in the simple algorithm, we first generate the $(\lambda-1)$-neighborhoods for s and t, returning CONNECTED if the sets overlap, and NOT CONNECTED if they do not overlap and one or both is too small. Next, we cycle through the vertices, building the set of global landmarks, whose elements have full, pairwise disjoint $(\lambda-1)$-neighborhoods. Finally, we examine all the edges as in the simple algorithm to determine which landmarks are in the same connected component. For the algorithm to work correctly, the following three conditions must be true:

- If the $(\lambda-1)$-neighborhoods of s and t are disjoint, and either is not full, then s and t are not connected. Proof: If s and t were connected, then by Lemma 2.3, their neighborhoods would intersect.

- If the $(\lambda-1)$-neighborhoods of s and t are disjoint, and both are full, then after the landmark set is constructed, the following is true: for all vertices, v, v is in the global landmark set, or v's $(\lambda-1)$-neighborhood overlaps a global landmark's $(\lambda-1)$-neighborhood, or v is connected to neither s nor t. Proof: If v's $(\lambda-1)$-neighborhood is full, then by construction it must either be a global landmark or its neighborhood must overlap the $(\lambda-1)$-neighborhood of one of the global landmarks. If v's neighborhood is not full, then by Lemma 2.3 it cannot be connected to s or t without overlapping their neighborhoods.

- After all edges are examined, any two landmarks that are in the same connected component are in the same Union-Find set. The proof is the same as the proof of Lemma 2.1.

Finally, we sketch how the algorithm, in the proper amount of space, determines whether two (k)-neighborhoods intersect. The `nv` routine generates the lists of landmarks for a neighborhood, not the vertices themselves, so the intersection routine receives two sets of size at most b. Given two such sets, S_1 and S_2, and an integer j, for each possible pair (v,w), $v \in S_1$, $w \in S_2$, the intersection routine generates the

(j)-landmark sets of v and w, and calls itself recursively, with these two sets and the integer $j-1$ as parameters. For the base case, $j=0$, the sets are just bbfs sets, and can be compared directly.

The code for the recursive algorithm is given in Figures 2.4, 2.5, and 2.6. The main routine and cl function for the recursive algorithm (Figures 2.4 and 2.5) are nearly identical to their counterparts in the simple algorithm (see Figures 2.1 and 2.2). The only changes are as follows:

- The algorithm is parameterized by λ instead of b, where $2 \leq \lambda \leq \log n$. The constant b is defined, where $b = \lceil n^{1/\lambda} \rceil$.

- All references to bbfs(x, b) are replaced by nv$(x, \lambda - 1)$. Similarly, all references to the intersection routine are calls to ri$(\cdot, \cdot, \lambda - 2)$.

Lemma 2.3 allows us to make two optimizations to the code.

1. When generating the (k)-neighborhood of a vertex, v, if we find that v's ($k-1$)-neighborhood is not full, we can immediately quit, since, by Lemma 2.3, we will not be able to find any other disjoint ($k-1$)-neighborhoods in v's connected component.

2. To test whether a (k)-neighborhood is full, we need only test whether the corresponding (k)-landmark set has b members: If the (k)-landmark set has fewer than b members, it clearly is not full. If the (k)-landmark set has b members, then the ($k-1$)-neighborhoods of each element in the set must be full, otherwise, by Lemma 2.3, the ($k-1$)-neighborhoods would not be disjoint.

2.3.1 Analysis of the Recursive Algorithm

Analysis of the space complexity of the algorithm is easy: Since the ($\lambda-1$)-neighborhoods of the global landmarks are full, there are only $O(n^{1/\lambda})$ global landmarks. Exclusive of recursive calls, each routine uses $O(n^{1/\lambda})$ registers, and the maximum recursion depth is $O(\lambda)$, hence at most $O(\lambda n^{1/\lambda})$ registers are needed.

Algorithm RUstcon (λ);

$\{2 \leq \lambda \leq \log n\}$

$b = \lceil n^{1/\lambda} \rceil$;
$L_{s,\lambda-1} = \mathsf{nv}(s, \lambda - 1)$; $L_{t,\lambda-1} = \mathsf{nv}(t, \lambda - 1)$
if $\mathsf{ri}(L_{s,\lambda-1}, L_{s,\lambda-1}, \lambda - 2)$ **then return** (CONNECTED);
if $|L_{s,\lambda-1}| < b$ **or** $|L_{t,\lambda-1}| < b$ **then return** (NOT CONNECTED);
{One or both neighborhoods not full. See page 30}
Initialize the set of landmarks to $\{s, t\}$;
for all vertices, v **do begin** {find the landmarks}
$L_{v,\lambda-1} = \mathsf{nv}(v, \lambda - 1)$;
if $|L_{v,\lambda-1}| < b$ **then** Neighborhood not full. Go to next vertex. {See page 30}
for all landmarks, l **do begin**
$L_{l,\lambda-1} = \mathsf{nv}(l, \lambda - 1)$;
if $\mathsf{ri}(L_{v,\lambda-1}, L_{l,\lambda-1}, \lambda - 2)$ **then** Not a landmark. Go to next vertex.
end;
Add v to the set of landmarks.
end;
Create a singleton Union-Find set containing each landmark, plus one
containing the special value "SMALL" for small components.
for all edges, $e = \{u, v\}$ **do begin**
Union(Find($\mathsf{cl}(u)$), Find($\mathsf{cl}(v)$));
end;
if Find(s) = Find(t) **then return** (CONNECTED);
else return (NOT CONNECTED);
end RUstcon.

Figure 2.4: The main routine of the recursive algorithm

```
procedure cl (vertex v): vertex;
                                          {Return the "closest" landmark to v}
  L_{v,λ-1} = nv(v, λ − 1);
  if |L_{v,λ-1}| < b then return ("SMALL");      {Neighborhood not full. See page 30}
  for all landmarks, l, in order do begin
    L_{l,λ-1} = nv(l, λ − 1);
    if ri(L_{v,λ-1}, L_{l,λ-1}, λ − 2) then return (l);
  end;
end cl.
```

Figure 2.5: The cl function of the recursive algorithm

The following time analysis is based on the code for the recursive algorithm given in Figures 2.4, 2.5, and 2.6. $R(\lambda)$ is an expression bounding the running time of the algorithm (including the time for the cl function) for parameter λ, and $N(k)$ is a recurrence relation bounding the running time of $\mathsf{nv}(\cdot, k)$. ri is called in two different ways: outside ri, it is usually called with a set of size one as its first parameter, to test whether the neighborhood of a single vertex overlaps the neighborhoods of a set of landmarks. It can also be called with a set of size up to b as its first parameter, to test for intersection between the neighborhoods of two sets of landmarks. Because the sizes of the set parameters to ri can vary, two recurrence relations, $I(k)$ and $I_1(k)$ are defined, where $I(k)$ bounds the running time of $\mathsf{ri}(\cdot, \cdot, k)$ when the sets are of size at most b, and $I_1(k)$ bounds the special case where the first set is of size 1, and the second of size at most b. For suitable $c_i, 0 \leq i \leq 11$, we have:

$$\begin{aligned} R(\lambda) &= O(n(N(\lambda-1) + bN(\lambda-1) + bI(\lambda-2) + b) \\ &\quad +2m(N(\lambda-1) + bN(\lambda-1) + bI(\lambda-2) + b) + m\alpha(m)) \end{aligned}$$

```
procedure ri (set of vertices S1, S2; integer k;): boolean;
          {test if the (k)-neighborhoods of a vertex in S1 and a vertex in S2 intersect}
  if k = 0 then return (S1 ∩ S2 ≠ ∅);                                  {Base case}
  for all pairs of vertices, v ∈ S1, w ∈ S2 do begin
    if ri(nv(v, k), nv(w, k), k − 1) then return (true);
  end;
  return (false);
end ri.

procedure nv (vertex v; integer k): set of vertices;
                                                  {Find the (k)-landmark set of v}
  if k = 1 then return (bbfs(v, b));                          {Base case. Use bbfs}
  if |nv(v, k − 1)| < b then return ({v});      {nv(v, k − 1) not full. See page 30}
  L_{v,k} = {v};
  for i = 2 to b do begin                                          {Find landmarks}
    for all vertices, u do begin
      if ri({u}, L_{v,k}, k − 1)) then
        Not a landmark. Go to next vertex.
      for all neighbors, w, of u do begin
        if ri({w}, L_{v,k}, k − 1) then                              {u is a landmark}
          L_{v,k} = L_{v,k} ∪ {u}. goto (L1);
      end;
    end;
    return (L_{v,k});                                {Not full, but no other landmarks}
  L1:
  end;
  return (L_{v,k});                                                        {Full}
end nv.
```

Figure 2.6: The ri and nv functions of the recursive algorithm

$$
\begin{aligned}
I(j) &= \begin{cases} O(b \log b) & \text{if } j = 0 \\ b^2(I(j-1)+c_1) + b(b+1)(N(j)+c_2) + c_3 & \text{if } j > 0 \end{cases} \\
I_1(j) &= b\,(I(j-1)+c_1) + (b+1)(N(j)+c_2) + c_3 \qquad j > 0 \\
N(k) &= \begin{cases} O(b^2 \log b) & \text{if } k = 1 \\ N(k-1) + (b-1)(n(I_1(k-1)+c_4) + \\ \qquad\qquad 2m(I_1(k-1)+c_5)+c_6)+c_7 & \text{if } k > 1 \end{cases}
\end{aligned}
$$

The first step in the algorithm is to compute b, which can be done in time $O(\lambda \log^3 n)$. b must be between 1 and n, so we can do a binary search among these integers to find b: If we know b is in the range $p \ldots q$, guess that $r = \lfloor (p+q)/2 \rfloor$ is the value of b. If $r^\lambda \geq n$ we know b is in the range $p \ldots r$. If $r^\lambda < n$, we know b is in the range $r+1 \ldots q$. Even using the naive multiplication algorithm, finding r^λ takes time only $O(\lambda \log^2 n)$, so the entire process will take time $O(\lambda \log^3 n)$, which is dominated by the other terms in the expression for the running time of the algorithm.

The equation for $I(j)$ uses a slight optimization. When testing for intersection between the (k)-neighborhoods of all possible ordered pairs of vertices, (u, v), we assume any particular u is fixed until all possible values of v have been tried for that u. This way, we need only generate u's (k)-landmark set once instead of b times, which gives us $b(b+1)$ total calls to $N(k)$.

Let $c = n/m$. Then, for $j > 0, k > 1$, we have:

$$
\begin{aligned}
N(k) &\leq N(k-1) + (b-1)(2+c)m(I_1(k-1)+c_8) \\
&\leq N(k-1) + (b-1)(2+c)m(b(I(k-2)+c_1) \\
&\quad +(b+1)(N(k-1)+c_2)+c_3+c_8) \\
N(k) &\leq b^2(2+c)m(I(k-2)+N(k-1)+c_9) \\
I(j) &\leq (b^2+b)(I(j-1)+N(j)+c_{10})
\end{aligned}
$$

Let $NI(k) = N(k) + I(k-1)$ for $k \geq 1$. Then, for $k > 1$,

$$
NI(k) \leq (b^2(2+c)m + b^2 + b)(NI(k-1) + c_{11})
$$

$$= O((b^2((2+c)m+1)+b)^{k-1}(N(1)+I(0)))$$
$$= O((b^2((2+c)m+1)+b)^{k-1}b^2\log b)$$

Recall that $b = \left\lceil n^{1/\lambda} \right\rceil$. Since $\lambda \leq \log n$, note that $b \leq n^{1/\lambda}+1 = O(n^{1/\lambda})$, and therefore

$$NI(\lambda-1) = O((n^{2/\lambda}(n+2m))^{\lambda-2}n^{2/\lambda}\log n^{1/\lambda}).$$

Thus

$$\begin{aligned} R(\lambda) &= O((n+2m)(N(\lambda-1)+bN(\lambda-1)+bI(\lambda-2)+b)+m\alpha(m)) \\ &= O((n+2m)bNI(\lambda-1)) \\ &= O((n+2m)n^{1/\lambda}(n^{2/\lambda}(n+2m))^{\lambda-2}n^{2/\lambda}\log n^{1/\lambda}) \\ R(\lambda) &= O(\frac{1}{\lambda}(n+2m)^{\lambda-1}n^{2-1/\lambda}\log n) \end{aligned}$$

Summarizing the results of this section, we have shown the following.

Theorem 2.4 *The recursive algorithm, described above, solves* USTCON *for arbitrary n-vertex, m-edge graphs in space $O(\lambda n^{1/\lambda}\log n)$ and time $O(\frac{1}{\lambda}(n+2m)^{\lambda-1}n^{2-1/\lambda}\log n) = n^{O(\lambda)}$, for any integer λ, $2 \leq \lambda \leq \log n$.*

Note that when $\lambda = 2$, the time and space bounds match those of the simple algorithm.

2.4 The Batched Algorithm

The batched algorithm is a variant of the simple algorithm that is faster for space $\omega(n^{1/2}\log n)$. Note that as the space bound increases, the simple algorithm uses more and more space to store landmarks and less and less to generate and store neighborhoods. The idea of the batched algorithm is to use as much space for neighborhoods as landmarks, by storing multiple neighborhoods. In the simple algorithm, most neighborhoods are generated for one reason only: to test them for intersection with the landmarks' neighborhoods. If we can check a landmark's neighborhood for intersection with multiple neighborhoods almost as quickly as the simple algorithm

takes to check for intersection with one, then, as the number of landmarks increases, the batched algorithm's performance compared to the simple algorithm's will become better and better.

As before, b will be the size of the neighborhoods generated, and l the number of landmarks. When building the landmark set, and when evaluating the cl function, the batched algorithm generates the neighborhoods for a batch of $\lfloor l/b \rfloor$ vertices, B. The algorithm then sorts all vertices in the $\lfloor l/b \rfloor$ neighborhoods into one list of $O(l)$ vertices. Using this list, it can efficiently test which vertices in the list are also in the neighborhood of a landmark; it performs a binary search in the list for each member of the landmark's neighborhood, labeling those vertices that match with the name of the landmark whose neighborhood they are in.

After repeating this step for all landmarks' neighborhoods, the batched algorithm must compute the intersection information for each member of B. To evaluate the cl function, it must find the lowest numbered landmark whose neighborhood intersected a given vertex's neighborhood; to build the landmark set, it need only check whether the intersection was empty. In the latter case, if a neighborhood of a vertex in B does not intersect any landmarks' neighborhood, the vertex must be added to the landmark set, and, in addition, the algorithm must mark any vertices in the list that are in the new landmark's neighborhood.

A special data structure for the sorted list is necessary to insure that all these operations can be performed efficiently. The sorted list we use is simply a list of pointers to members of the neighborhoods, plus a cl field for each entry, to store the name of a landmark whose neighborhood contains it. Each member of a neighborhood in turn has a pointer to its entry in the sorted list. However, the pointers are not in a one-to-one correspondence, because the sorted list does not contain duplicate entries for a vertex that appears in multiple neighborhoods. The pointers into the list from the neighborhoods provide efficient lookup for the vertices, but the pointers back to the neighborhoods are used only to find the vertex name for an entry in the list.

As a simple example, see Figure 2.7 (page 37), depicting two neighborhoods, n_u and n_v, and a list that might be constructed for them. In this example, $b = 5$, and the two neighborhoods have two vertices in common, so the sorted list has only 8 entries. Five entries point to vertices in the first neighborhood, and three to vertices

n_u			n_v		
	vertex	list ptr		vertex	list ptr
u_1	11	1	v_1	12	2
u_2	13	3	v_2	13	3
u_3	15	4	v_3	15	4
u_4	17	6	v_4	16	5
u_5	19	8	v_5	18	7

sorted list		
	neighborhood ptr	cl
1	u_1	
2	v_1	
3	u_2	
4	u_3	
5	v_4	
6	u_4	
7	v_5	
8	u_5	

Figure 2.7: The batched algorithm's data structure

in the second neighborhood. Note that the indices for the neighborhoods and the list are for reference purposes only, and are not actually stored in the structure.

Given this data structure, we can perform the needed operations efficiently. The list is sorted, so lookup of a single vertex will take time $O(\log l)$ using binary search. The pointers to the list from the neighborhoods allow constant access time per vertex, so discovering which landmarks' neighborhoods overlap a given neighborhood or marking all list entries for a neighborhood requires time $O(b)$.

The code for constructing the landmark set and computing the cl function in the batched algorithm is given in Figures 2.8 and 2.9, respectively. The code for both is similar — the main differences are as follows:

- The cl code must process all edges, while the landmark code processes all vertices. In the final phase, the landmark code processes each vertex separately, while the cl code processes two vertices together (the two endpoints of each edge).

- During the loop labeled check intersection, the cl function determines the name of the lowest numbered landmark that intersected this vertex's neighborhood, while the landmark code merely checks for intersection.

- After the loops labeled check intersection, the cl function joins the Union-Find sets of the closest leaders to each endpoint. The landmark code must execute an extra loop to mark the vertices in v's neighborhood if v is added to the set of landmarks.

The code for the main routine of the batched algorithm is given in Figure 2.10. The code for constructing the landmark set and computing the cl function (see Figures Figures 2.8 and 2.9) should be inserted in the indicated places to yield the complete algorithm. Note that the cl function is now no longer a separate routine — the batched nature of the algorithm means that we evaluate the cl function for many vertices at once, so the code for the function is incorporated into the main routine.

2.4.1 Analysis of the Batched Algorithm

The batched algorithm requires more space than the simple algorithm to store multiple neighborhoods and the sorted list of vertices, but these structures only use space $\Theta(l \log n)$, asymptotically as much as the simple algorithm ($O((n \log n)/b)$, since $l = \lfloor n/b \rfloor$).

Finding the neighborhood of a vertex requires time $O(b^2 \log b)$, so finding $O(l/b)$ neighborhoods takes time $O(lb \log b)$. The special data structure can be constructed in time $O(l \log l)$, the time required for sorting the $O(l)$ vertices.

After finding the neighborhood of a landmark, time $O(b \log l)$ is needed to search for its vertices in the sorted list. The total time needed for the mark overlapping vertices loop is therefore $O(l(b^2 \log b + b \log l))$.

```
for i = 0 to ⌈n/⌊l/b⌋⌉ − 1 do begin
  B = {1 + i ⌊l/b⌋, 2 + i ⌊l/b⌋, ..., ⌊l/b⌋ + i ⌊l/b⌋}      {the next set of l/b vertices}
  Initialize the batched algorithm's data structure.
  for all vertices, v ∈ B do begin
    generate bbfs(v, b), and add the neighborhood to the data structure;
  end;
  Create a sorted list, Q, of all vertices in the neighborhoods of vertices in B,
    and store it in the data structure. Set each vertex's cl field to +∞
  for all landmarks, l do begin                              {mark overlapping vertices}
    for all vertices, w ∈ bbfs(l, b) do begin
      if w is in Q and w's cl field > l then w's cl field = l.
    end;
  end;
  for all vertices, v ∈ B do begin                           {evaluate vertices in B}
    if v's neighborhood is not full then v is not a landmark. Go to next vertex.
    for all vertices, u ∈ v's neighborhood do begin          {check intersection}
      if u's cl field in Q ≠ +∞ then
        v is not a landmark. Go to next vertex.
    end;
    Add v to the set of landmarks.                           {new landmark}
    for all vertices, u ∈ v's neighborhood do begin
      u's cl field in Q = v.
    end;
  end;
end;
```

Figure 2.8: Code to find landmarks in the batched algorithm.

```
for i = 0 to ⌈2m/⌊l/b⌋⌉ − 1 do begin
   E = edges {e_{1+i⌊l/b⌋}, e_{2+i⌊l/b⌋}, ..., e_{⌊l/b⌋+i⌊l/b⌋}}        {the next set of l/b edges}
   B = The endpoints of the edges in E.
   Initialize the batched algorithm's data structure.
   for all vertices, v ∈ B do begin
      generate bbfs(v, b), and add the neighborhood to the data structure;
   end;
   Create a sorted list, Q, of all vertices in the neighborhoods of vertices in B,
      and store it in the data structure. Set each vertex's cl field to +∞
   for all landmarks, l do begin                                  {mark overlapping vertices}
      for all vertices, w ∈ bbfs(l, b) do begin
         if w is in Q and w's cl field > l then w's cl field = l.
      end;
   end;
   for all edges, e = {u, v} ∈ E do begin                         {process edges in E}
      if u's neighborhood is not full then Go to next edge.
                                                  {u and v in a SMALL component}
      l_u = n + 1; l_v = n + 1;                      {initialize "closest landmarks"}
      for all vertices, w ∈ u's neighborhood do begin             {check intersection}
         if w's cl field is less than l_u then l_u = w's cl field.
      end;
      for all vertices, w ∈ v's neighborhood do begin             {check intersection}
         if w's cl field is less than l_v then l_v = w's cl field.
      end;
      Union(Find(l_u), Find(l_v));
   end;
end;
```

Figure 2.9: Code to find the "closest landmarks" in the batched algorithm.

Algorithm BUstcon (integer: b);

$\{1 \leq b \leq \sqrt{n}\}$

generate bbfs(s, b) and bbfs(t, b).
if s's neighborhood overlaps t's **then return** (CONNECTED);
if either neighborhood is not full **then return** (NOT CONNECTED);
Initialize the set of landmarks to $\{s, t\}$;

{**Insert code to find landmarks**}

Create a singleton Union-Find set containing each landmark, plus one
containing the special value "SMALL" for small components.

{**Insert code to evaluate the** cl **function**}

if Find(s) = Find(t) **then return** (CONNECTED);
else return (NOT CONNECTED);
end BUstcon.

Figure 2.10: The main routine of the batched algorithm

Each operation in the check intersection and new landmark loops requires constant time, so the evaluate vertices in B loop in Figure 2.8 takes time $O(l/b \cdot b) = O(l)$. Similar analysis applies to the code used to evaluate the cl function.

The outermost loop is executed $O(n/(l/b))$ times to build the landmark set, and $O(m/(l/b))$ times to process the edges. The total running time for the algorithm is:

$$\begin{aligned} & O((m+n)b/l)(lb\log b + l\log l + lb^2\log b + lb\log l + l) + m\alpha(m)) \\ = \;& O((m+n)b(b^2\log b + b\log l) + m\alpha(m)) \\ = \;& O((m+n)(b^3\log b + b^2\log n)) \end{aligned}$$

When $b = n^{1/2}$, the batched algorithm runs as quickly as the simple algorithm, but for $b = o(n^{1/2})$, the batched algorithm is asymptotically faster. When $b = 1$, the batched algorithm uses time $O((m+n)\log n)$, only a factor of $O(\log n)$ slower than the time-optimal depth- or breadth-first search. As in the simple algorithm, it is not necessary to use the extremely efficient version of the Union-Find subroutines with

both weighted unions and path compression to reach this asymptotic time bound. Unlike the simple algorithm, however, we must use either one optimization or the other — with neither weighted unions or path compression, the Union-Find operations take total time $O(ml)$, which will dominate the running time for small values of b. If only one optimization is used, the operations take total time $O(m \log l)$, which is always dominated by the other terms.

Thus we have the following.

Theorem 2.5 *The batched algorithm, presented above, solves* USTCON *for arbitrary n-vertex, m-edge graphs in space $O((n \log n)/b)$ and time $O((m+n)(b^3 \log b + b^2 \log n)$ for any $1 \leq b \leq \sqrt{n}$.*

2.5 Conclusions

These algorithms provide a deterministic time-space tradeoff for USTCON. With space $s = \Theta(n \log n)$, the batched algorithm's performance is only a factor of $\log n$ worse than depth- or breadth-first search, and for space $s = \Omega(n^{1/2} \log n)$, the algorithms are no more than a factor of $n \log n$ worse than the best-known algorithms, deterministic and probabilistic. However, as the space is decreased further, things rapidly become worse, with an added factor of roughly m to the running time of the recursive algorithm every time λ is increased by one.

When the recursive algorithm is taken to extremes, its time and space bounds resemble those for Savitch's result [Sav70]. The lower limit for the space used by the algorithm is reached when $\lambda = \log n$. With this value for λ, the algorithm uses space $O(\log^2 n)$ and time $n^{O(\log n)}$, similar to Savitch's space and time bounds [Sav70]. Recent work by Nisan [Nis92] has shown that polynomial time and $O(\log^2 n)$ space are simultaneously achievable, but his algorithm has a very high polynomial running time — he estimates $O(n^{45})$. If this is the actual time bound, the recursive algorithm is asymptotically faster than Nisan's for space down to $O(n^{1/22} \log n)$.

Chapter 3

TIME-SPACE TRADEOFFS FOR DIRECTED S-T CONNECTIVITY

3.1 Introduction

This chapter presents time-space tradeoffs for directed s-t connectivity. We first introduce a simple variant of depth- and breadth-first search that can use space between $\Theta(n \log n)$ and $\Theta(n)$. We then present two different time-space tradeoffs that can be combined to give a deterministic algorithm for directed s-t connectivity that achieves polynomial time and sublinear space simultaneously. The algorithm can use as little as $n/2^{\Theta(\sqrt{\log n})}$ space while still running in polynomial time. One of the tradeoffs is an algorithm that finds short paths in a directed graph in polynomial time and sublinear space. The *short paths problem* is a special case of STCON that retains many of the difficulties of the general problem, and seems particularly central to designing small space algorithms for STCON. Interestingly, our algorithm for the short paths problem is a generalization of two well-known algorithms for STCON. In one extreme it reduces to a variant of the linear time breadth-first search algorithm, and in the other extreme it reduces to the $O(\log^2 n)$ space, superpolynomial time algorithm of Savitch.

The rest of this chapter is organized as follows: In Section 3.2, we present a simple modification to breadth- or depth-first search that allows us to use less space while maintaining a near-optimal running time. In Section 3.3, we present another tradeoff based on a breadth-first search of the graph, and in Section 3.4, we present an algorithm to find short paths in a directed graphs. Section 3.5 shows how these last two tradeoffs can be combined to yield an algorithm that solves STCON in polynomial time and sublinear space. Alone, neither algorithm can solve STCON in simultaneous polynomial time and sublinear space.

3.2 The Bounded Queue Tradeoff

As a simple example, we first present a variant of depth- and breadth-first search that can run in any space bound from $\Theta(n \log n)$ (the space bound of these two algorithms) to $\Theta(n)$. For space $O(s)$, $n \leq s \leq n \log n$, the algorithm uses time $O(m+n^2(\log n)/s)$. Note that the algorithm's running time is always within a $\log n$ factor of the optimal time, and if $m = \Omega(n \log n)$, the algorithm is asymptotically as fast as depth- or breadth-first search. For simplicity, we will describe the algorithm as a variant of breadth-first search only. The generalization to depth-first search is straightforward — simply substitute "stack" for "queue" and "depth-first" for "breadth-first" in the following discussion.

The algorithm works as follows: execute a standard breadth-first search, but don't allow the queue of visited but unexplored vertices to hold more than $s/\log n$ vertices. The standard breadth-first search algorithm maintains a status vector of n bits to indicate which vertices have been visited. We modify the status vector to hold two bits for each vertex, thus allowing a third status besides VISITED and UNVISITED. The third status, UNEXPLORED, indicates the vertex has been visited, but not added to the queue. When an UNVISITED or UNEXPLORED vertex is visited, if the queue is not full, it is added to the queue and its status changed to VISITED. If the queue is full, its status is set to UNEXPLORED. When the queue becomes empty, the status vector is scanned sequentially, and, assuming the queue has not become full again, any vertex with status UNEXPLORED is added to the queue and its status changed to VISITED. Pseudocode for the algorithm appears in Figure 3.1.

As long as the queue is allowed to grow to size at least $n/\log n$, the size of the queue dominates the space bound of the algorithm. Making the queue smaller than $n/\log n$ does not significantly reduce the space bound, since the status vector uses space $\Theta(n)$. If k is the size of the queue, and $k \geq n/\log n$, then the space bound is s, where $s = k \log n$.

The key to the time analysis is to note that the status vector is not scanned for UNEXPLORED vertices very often. In particular, no vertices can have status UNEXPLORED unless the queue was full at some time since the status vector was last scanned. If this occurs, then at least $s/\log n$ vertices were added to the queue and visited since the last scan, and since a vertex can only be visited once, this gives

```
Algorithm Bqueue (integer: k);
      {Breadth-first search from s with a bounded queue of size k, n/log n ≤ k ≤ n}
  Create a queue, Q, and a status vector, S, with n 2-bit entries.
  Initialize each entry in S to UNVISITED.
  S[s] = VISITED; insert(Q, s); Qsize = 1;                    {insert s into Q}
  while Qsize ≠ 0 do begin
    v = delete(Q); Qsize = Qsize − 1;                         {get a vertex from Q}
    for all neighbors, w, of v do begin                       {mark neighbors}
      if S[w] ≠ VISITED and Qsize < k then
        S[w] = VISITED; insert(Q, w); Qsize = Qsize + 1;      {insert neighbor into Q}
      else
        if S[w] ≠ VISITED then S[w] = UNEXPLORED;             {Queue full}
    end;
    if Qsize = 0 then      {Scan the bit vector, looking for UNEXPLORED vertices}
      for i = 1 to n do begin
        if S[i] = UNEXPLORED then
          S[i] = VISITED; insert(Q, i); Qsize = Qsize + 1;
        if Qsize = k then exit scan loop.                     {Queue full}
      end;
  end;
  if S[t] = VISITED then return (CONNECTED);
  else return (NOT CONNECTED);
end Bqueue.
```

Figure 3.1: Details of the bounded queue algorithm

an upper bound on the number of scans of $n(\log n)/s$. Each scan takes time $O(n)$, giving a total time for scans of $O(n^2(\log n)/s)$. The rest of the algorithm is dominated by the mark neighbors loop, which is executed $O(m)$ times. The total time bound for the algorithm, then, is $O(m + n^2(\log n)/s)$.

Summarizing the results of this section, we have the following:

Theorem 3.1 *The bounded queue algorithm, presented above, solves* STCON *in time* $O(m + n^2(\log n)/s)$ *for any space bound,* $O(s)$, $n \leq s \leq n \log n$.

3.3 The Breadth-First Search Tradeoff

The second tradeoff is also a variant of breadth-first search. Consider the tree constructed by a breadth-first search beginning at s. The tree can contain n vertices, and thus requires $O(n \log n)$ space to store. Instead of constructing the entire tree, our modified breadth-first search generates a fraction of the tree.

Suppose we want our modified tree to contain at most n/λ vertices. We can do this by only storing (the vertices in) every λth level of the tree. Number the levels of the tree 0, 1, ..., $n-1$, where a vertex is on level l if its shortest path from s is of length l. Divide the levels into equivalence classes $C_0, C_1, \ldots, C_{\lambda-1}$ based on their number mod λ. Besides s, the algorithm stores only the vertices in one equivalence class, C_j, where j is the smallest value for which C_j has no more than the average number of vertices, n/λ.

The algorithm constructs this partial tree one level at a time. It begins with level 0, which consists of s only, and generates levels j, $j+\lambda$, $j+2\lambda$, ..., $j+\lambda \cdot \lfloor n/\lambda \rfloor$. Given a set, S, of vertices, we can find all vertices within distance λ of S in time $n^{O(\lambda)}$ and space $O(\lambda \log n)$ by enumerating all possible paths of length at most λ and checking which paths exist in G. This can be used to generate the levels of the partial tree. Let V_i be the vertices in levels 0, j, $j+\lambda$, ..., $j+i\lambda$. Consider the set of vertices, U, that are within distance λ of a vertex in V_i. Clearly, U contains all the vertices in level $j+(i+1)\lambda$. However, U may also contain vertices in lower numbered levels. The vertices in level $j+(i+1)\lambda$ are those vertices in U that are not within distance $\lambda - 1$ of a vertex in V_i. Thus, to get V_{i+1} we add to V_i all vertices that are within distance λ but not $\lambda - 1$ of V_i.

Pseudocode for the algorithm appears in Figure 3.2. Note that to find an equivalence class with at most n/λ vertices, the algorithm just tries all classes in order, discarding a class if it generates too many vertices.

Referring to Figure 3.2, the algorithm's space bound is dominated by the number of vertices in S and S', and the $O(\lambda \log n)$ space needed to generate all possible paths

```
Algorithm Bfs (integer: λ);
                                {remember every λth level of the breadth-first search tree}
  for j = 0 to λ − 1 do begin          {first level to remember (apart from level 0)}
    S = {s}.
    for all vertices, v do begin                    {Find vertices on the first level.}
      if v within distance j of s and v not within distance j − 1 of s then
        if |S| > n/λ then try next j.
                                        {Don't store more than n/λ vertices, + vertex s}
        else add v to S.
    end;
    for i = 1 to ⌊n/λ⌋ do begin
      S' = ∅.
      for all vertices, v do begin                  {Find vertices on the next level.}
        if v within distance λ of some vertex in S and
            v not within distance λ − 1 of any vertex in S then
          if |S| + |S'| > n/λ then try next j.
          else add v to S'.
      end;
      S = S ∪ S'.
    end;
    if t within distance λ of a vertex in S then return (CONNECTED);
    else return (NOT CONNECTED);
  end;
end Bfs.
```

Figure 3.2: Details of the breadth-first search algorithm

of length λ from S. There are never more than $n/\lambda + 1$ vertices in S and S', so the algorithm uses space $O((\frac{n}{\lambda} + \lambda)\log n)$. The time bound is dominated by repeatedly testing whether a vertex is within distance λ of a vertex in S. This test is performed $O(n^3/\lambda)$ times — the innermost loop to find the vertices on the next level of the tree

makes $O(n \cdot n/\lambda)$ such tests (testing for a path from the $O(n/\lambda)$ vertices in S to all other $O(n)$ vertices), and is executed $O(\lambda \cdot n/\lambda)$ times.

In summary, we have shown the following:

Theorem 3.2 *For any n vertex directed graph and any integer $\lambda, 1 \leq \lambda \leq n$, the breadth-first search algorithm presented above solves s-t connectivity in space $O(n(\log n)/\lambda + S(PATH(\lambda)))$ and time $O(n^3/\lambda \cdot T(PATH(\lambda)))$, where $S(PATH(\lambda))$ and $T(PATH(\lambda))$ denote the space and time bounds, respectively, of the algorithm we use to test for a path of length at most λ between two vertices.*

Note that we assume that testing for a path of length at most j, $j-1$ or $\lambda - 1$ will not take asymptotically more time than testing for a path of length at most λ. This is because the former problems are trivially reducible to the latter. To test for a path of length at most $\lambda' < \lambda$, connect $\lambda - \lambda'$ new vertices $v_1, v_2, \ldots, v_{\lambda-\lambda'}$ in a chain to s, by adding the edges $(v_1, v_2), \ldots, (v_{\lambda-\lambda'-1}, v_{\lambda-\lambda'}), (v_{\lambda-\lambda'}, s)$. There will be a path in the new graph from v_1 to t of length at most λ if and only if there was a path in the original graph from s to t of length at most λ'.

Using a straightforward enumeration of all paths, testing whether a vertex is within distance λ requires $n^{O(\lambda)}$ time and $O(\lambda \log n)$ space. This algorithm is not sufficient for our purposes. In particular, if λ is asymptotically greater than a constant, the algorithm uses superpolynomial time. If we restrict our input to graphs with bounded degree, there is a slight improvement: in a graph where the outdegree is bounded by d, the number of paths of length λ from a vertex is at most d^λ, so for these graphs, λ can be $O(\log n)$, and the algorithm will run in polynomial time. Note that the overall algorithm still does not use sublinear space in this case, even though the subroutine for finding paths of length λ does.

The problem with this algorithm is its method of finding vertices within distance λ: explicitly enumerating all paths is not very clever, and uses too much time. There is hope for improvement, though, in the fact that this method uses little space: $O(\lambda \log n)$, compared to $O(\frac{n}{\lambda} \log n)$ for the rest of the algorithm. Indeed, in the next section we give an algorithm that uses more space but runs much faster.

3.4 The Short Path Tradeoff

Consider the *short paths problem*:

Definition: Given a directed graph, G, and two distinguished vertices, s and t, the short paths problem for function $f(n)$ (denoted SHORTP($f(n)$)) is to determine whether there is a path in G from s to t of length less than or equal to $f(n)$.

The short paths problem is a special case of STCON that seems to encapsulate many of the difficulties of the general problem. It is particularly interesting given the breadth-first search algorithm above, because a more efficient method of finding short paths would clearly lead to an improvement in that algorithm's time bound.

Our second tradeoff is an algorithm that solves the short paths problem for many $f(n)$ in sublinear space and polynomial time. As will become clear, we will eventually want $f(n) = 2^{\Theta(\sqrt{\log n})}$, but to simplify the following discussion, we begin with the more modest goal of finding a sublinear space, polynomial time algorithm for the short paths problem with $f(n) = \log^c n$, for some integer constant $c \geq 1$.

As noted before, we already have a sublinear space, polynomial time algorithm that searches to distance $\log n$ on bounded degree graphs: because there are a constant number of ways to leave each vertex, we can enumerate and test all paths of length $\log n$ in polynomial time. In a general graph, this approach will not work, because there can be up to $n-1$ possible edges from each vertex, and explicit enumeration can yield a superpolynomial number of paths of length $\log n$. We can avoid this problem by using a labeling scheme that limits the number of possible choices at each step of the path.

Suppose we divide the vertices into k sets, according to their vertex number mod k. Then, every path of length L ($L = f(n)$) can be mapped to an $(L+1)$-digit number in base k, where digit i has value j if and only if the ith vertex in the path is in set j. Conversely, each such number defines a set of possible paths of length L.

Given this mapping, our algorithm is straightforward: generate all possible $(L+1)$-digit k-ary numbers, and check for each number whether there is a path in the graph that matches it. For a given k-ary number, the algorithm uses approximately $2n/k$ space to test for the existence of a matching path in the graph, as follows. Suppose we are looking for a path from s to t, and want to test the $(L+1)$-digit number

$\langle s \bmod k, d_1, d_2, \ldots, d_{L-1}, t \bmod k \rangle$. We begin with a bit vector of size $\lceil n/k \rceil$, which corresponds to the vertex set d_1. Zero the vector, and then examine the outedges of s, marking any vertex v in set d_1 (by setting the corresponding bit in the vector) if we find an edge from s to v. When we are finished, the marked vertices in the vector are the vertices in d_1 that have a path from s that maps to the first two digits of the number. Using this vector, we can run a similar process to find the vertices in d_2 that have a path from s that maps to the first three digits of the number, and store them in a second vector of size $\lceil n/k \rceil$. In general, given a bit vector of length $\lceil n/k \rceil$ representing the vertices in d_i with a path from s that maps to the first $i+1$ digits of the number, we use the other vector to store the vertices in d_{i+1} with a path from s that maps to the first $i+2$ digits. Pseudocode for the algorithm appears in Figure 3.3.

The algorithm uses space $O(n/k)$ to store the vectors, and $O(L \log k)$ to write down the path to be tested. Let C be the maximum number of edges from one set of vertices d_i to another d_j (note: i and j can be the same). For all steps in each path, we do at most $O(n/k + C)$ work zeroing the vector and testing for edges from d_{i-1} to d_i. Since $C = O((n/k)^2)$, the algorithm uses $O(k^L L(n/k)^2) = O(k^L n^3)$ time to test all L steps on each of the k^L paths.

Unfortunately, this does not reach our goal of polynomial time and sublinear space when $L = \log^c n$. With a distance as small as $\log n$, k^L is only polynomial if k is constant, and if k is constant, the algorithm does not use sublinear space. We *can* achieve polynomial time and sublinear space by reducing the distance the algorithm searches. For example, if $L = \log n / \log \log n$, k can be $\log^c n$ for any constant c, and the algorithm will run in $O(n/\log^c n)$ space and $O((\log n)^{c \log n / \log \log n} n^3) = O(n^{c+3})$ time.

The algorithm can be improved by invoking it recursively. Consider the loop in the algorithm that tests for edges between one set of vertices and the next. This loop, in effect, finds paths of length one from marked vertices in the first set to vertices in the second set. Instead of finding paths of length one, we can use the short paths algorithm to find paths of length L, yielding an algorithm that uses twice as much space, but finds paths of length L^2. In general, using r levels of recursion, the improved algorithm can find paths of length L^r using $O(r(n/k + L \log k))$ space. If we make a recursive call for every possible pair of vertices in $d_{i-1} \times d_i$, we get a time

Algorithm SP (integer: k, L; vertex s, t);
{Test for a path of length L between s and t using space $O(n/k)$}
Create V_0 and V_1, two $\lceil n/k \rceil$ bit vectors.
for all $(L+1)$-digit numbers in base k,
$\langle d_0 = s \bmod k, d_1, \ldots, d_{L-1}, d_L = t \bmod k \rangle$ **do begin**
Set all bits in V_0 to zero, and mark s (set the corresponding bit to 1).
for $i = 1$ **to** L **do begin**
Set all bits in $V_{i \bmod 2}$ to zero.
{Find edges from d_{i-1} to d_i}
for all u in d_{i-1} marked in $V_{(i-1) \bmod 2}$ **and all** v in d_i **do begin**
if (u, v) is an edge **then**
mark v in $V_{i \bmod 2}$.
end;
end;
if t is marked in $V_{L \bmod 2}$ **then return** (CONNECTED);
end;
return (NOT CONNECTED);
end SP.

Figure 3.3: Details of the short paths algorithm

bound of $O((k^L L(n/k)^2))^r) = O(n^{2r+1}k^{rL})$, since $L^r = O(n)$. A further refinement improves the time bound: one recursive call will find all vertices in d_i reachable from any reachable vertex in d_{i-1}.

Pseudocode for the recursive version of the algorithm is given in Figure 3.4

Given the discussion above, the time used by the recursive algorithm is bounded by the following recurrence relation, where $T(j)$ is the time used by the algorithm with j levels of recursion. For an appropriately chosen constant c:

$$T(j) = \begin{cases} O(n/k + C) & \text{if } j = 1 \\ k^L L(T(j-1) + cn/k) & \text{if } j > 1 \end{cases}$$

Algorithm SPR (integer: k, L, r, d_s, d_t; vector V_s): vector;
{Return the vector of vertices in set d_t that are reachable by paths of length L^r from vertices in set d_s that are marked in vector V_s.}
Create V_0, V_1, and V_t, three $\lceil n/k \rceil$-bit vectors. Set all bits in V_t to zero.
for all $(L+1)$-digit numbers in base k, $\langle d_0 = d_s, d_1, \ldots, d_{L-1}, d_L = d_t \rangle$ **do begin**
$V_0 = V_s$.
for $i = 1$ **to** L **do begin**
{Find edges from d_{i-1} to d_i}
if $r = 1$ **then** {base case}
Set all bits in $V_{i \bmod 2}$ to zero.
for all u in d_{i-1} marked in $V_{(i-1) \bmod 2}$ **and all** v in d_i **do begin**
if (u, v) is an edge **then**
mark v in $V_{i \bmod 2}$.
end;
else
$V_{i \bmod 2} = \mathsf{SPR}(k, L, r-1, d_{i-1}, d_i, V_{(i-1) \bmod 2})$.
end;
Set all bits in V_t that are set in $V_{L \bmod 2}$. $\{\star\}$
end;
return (V_t);
end SPR.

Figure 3.4: Details of the recursive short paths algorithm

In the base case, the algorithm does $O(n/k + C)$ work. At other levels, the algorithm makes $k^L L$ recursive calls to itself, as well as doing some auxiliary work, such as setting all vector entries to zero. Solving the recurrence relation for $j = r$ gives time $O((k^L L)^r (n/k + C)) = O(n^3 k^{rL})$.

Thus, we have shown the following:

Theorem 3.3 *The recursive short paths algorithm, presented above, can search to distance L^r in time $O(k^{rL} L^r (n/k + C))$ $(= O(n^3 k^{rL}))$ and space $O(r(n/k + L \log k))$,*

for arbitrary integers r, k, and L, such that $n \geq k \geq 1, r \geq 1, L \geq 1$, and $L^r \leq n$.

3.4.1 Notes on the Algorithm

- This recursive algorithm meets our goal of finding a sublinear space, polynomial time algorithm that detects paths of polylogarithmic length. For example, for $L = \log n / \log\log n$, $k = \log^r n$, and constant $r \geq 2$, the algorithm searches to distance $L^r = \omega(\log^{r-1} n)$ in time $O(n^3 k^{rL}) = O(n^{r^2+3})$ and space $O(rn/\log^r n)$.

- As mentioned in the introduction, this algorithm does not give a polynomial time, sublinear space algorithm for STCON by itself. The algorithm searches to distance L^r by testing k^{rL} numbers. If $L^r = n$, then k^{rL} is polynomial only if $k = O(1)$. But if $k = O(1)$, the algorithm does not use sublinear space.

- As given, the algorithm does not solve the short paths problem, since it discovers only paths of length exactly L^r, but no shorter. To solve the short paths problem, we can use the same algorithm with one small change — the line that marks the bits in V_t that are marked in $V_{L \bmod 2}$ (denoted with a $\star$ in Figure 3.4), should be moved inside the **for** loop that it currently follows, and changed to the following:

  if $d_L = d_i$ **then** Set all bits in V_t that are set in $V_{i \bmod 2}$.

  This checks whether the first i digits in the number being tested map to a path to a vertex in d_t — if so, then the algorithm has discovered a path from a vertex in d_s marked in V_s to a vertex in d_t, one possibly shorter than L^r. If there is such a path of distance $< L^r$, then the number that corresponds to the path will be generated by the algorithm (multiple times) as an initial substring of the longer numbers it generates, and this test will detect the path.

- This so-called "short paths" algorithm is actually a general algorithm for s-t connectivity, with behavior and performance similar to the best-known previous algorithms. If we let $k = 1$, $L = n$, and $r = 1$, the algorithm is a somewhat inefficient variant of breadth-first search that uses $O(n)$ space and $O(n(n+m))$ time: the algorithm first finds all vertices at distance 1 from s, then distance

2, etc., until it has searched to distance n. At the other end of the time-space spectrum, Savitch's algorithm (see Section 1.3) is just the special case of this algorithm where $k = n$, $L = 2$, and $r = \lceil \log n \rceil$ — this is also the minimum space bound for the algorithm.

For more on the short paths problem, see Section 4.3.1.

3.5 Combining the Two Algorithms

As an immediate consequence of the previous two sections, we have an algorithm for STCON using sublinear space and polynomial time: use the modified breadth-first search algorithm to find every $(\log^c n)$-th level of the tree (for integer constant $c \geq 2$), with the recursive short paths algorithm (the version that checks for paths of length up to L^r) as a subroutine to find the paths between levels. With careful choices of the parameters k, L and r, however, the algorithm can use even less space while still maintaining polynomial time.

In general, the breadth-first search algorithm finds every (L^r)-th level of the tree, and the short paths algorithm searches to distance L^r. Substituting the space bound for the short paths algorithm (see Theorem 3.3) for the term $S(PATH(\lambda))$ in the breadth-first search algorithm (see Theorem 3.2), we get a space bound for this algorithm of

$$O(n(\log n)/L^r + r(n/k + L \log k)), \tag{3.1}$$

where the first term corresponds to the space used by the partial breadth-first tree, and the second to the space used to find short paths. Substituting the short paths time bound for the term $T(PATH(\lambda))$ in the breadth-first search time bound gives a time bound of

$$O(n^3/L^r \cdot k^{rL} L^r (n/k + C)) = O(n^5 k^{rL-2}).$$

The above time bound applies when we call the short paths algorithm every time the breadth-first search algorithm needs to know whether one vertex is within distance L^r of another. The two algorithms can be combined more efficiently by noticing that the short paths algorithm can answer many short paths queries in one call — for any pair of sets, (Q, R), such that R is one of the k sets of vertices in the short paths

for $i_1 = 0$ **to** $k-1$ **do begin**
  $S_{i_1} = \{$all vertices whose vertex number mod $k = i_1\}$.
  $P = \emptyset$. $\{P$ will be all vertices in S_{i_1} on the next tree level.$\}$
  for $i_2 = 0$ **to** $k-1$ **do begin**
    $S_{i_2} = \{$all vertices whose vertex number mod $k = i_2\}$.
    $Q = S \cap S_{i_2}$. $\{Q$ is all vertices in S_{i_2} on previous tree levels.$\}$
    $A = \{$all vertices in S_{i_1} within distance L^r of a vertex in $Q\}$.
    $B = \{$all vertices in S_{i_1} within distance $L^r - 1$ of a vertex in $Q\}$.
  end;
  $P = P \cup A - B$.
  if $|S| + |S'| + |P| > n/k$ **then** try next j.
  else $S' = S' \cup P$.
end;

Figure 3.5: Combining the two algorithms efficiently.

algorithm, and Q is a subset of one of the k sets, the short paths algorithm can find all vertices in R within distance L^r of a vertex in Q. Thus, the short paths algorithm need only be called $O(k^2)$ times to generate the next level of the tree, once for each possible pair of the k sets in the short paths algorithm. Figure 3.5 gives the code that should be used in place of the loop in Figure 3.2 that finds vertices on the next level of the breadth-first search tree. Similar code should replace the earlier loop in Figure 3.2 that finds the vertices on the first level.

This improvement gives a total of $O(k^2 n/L^r)$ calls to the short paths algorithm, for a time bound of

$$O(k^2 n/L^r \cdot k^{rL} L^r (n/k + C)) = O(n^3 k^{rL}).$$

We want to find the minimum amount of space required by the algorithm while still maintaining a polynomial running time. Let $x = \log k$. Then, to maintain polynomial time we must have $xLr = O(\log n)$.

For simplicity, we bound expression (3.1) from below as

$$\Omega(n/L^r + n/2^x). \tag{3.2}$$

(That is, we omit the $\log n$ factor in the first summand and the r factor in the second summand, and leave out the third summand altogether.) Increasing either x, L, or r in expression (3.2) will decrease the value of the expression, but we have the further restriction that $xLr = c_1 \log n$ for some constant c_1. In other words, the higher the value of one of the three variables, the lower the value of at least one of the other two. Since x and r are both in exponents in Expression (3.2), increasing these variables will decrease the expression more effectively than increasing L, so we must choose L to be a constant to minimize the expression. It can therefore be written as

$$\Omega(n/2^{(c_2 \log n)/x} + n/2^x) \tag{3.3}$$

for some constant c_2. By taking the derivative with respect to x, we find that the minimum of this expression is given when $x^2 = \Theta(\log n)$, and the space is $n/2^{\Theta(\sqrt{\log n})}$.

Substituting these same values, $\sqrt{\log n}$ for r, $2^{\Theta(\sqrt{\log n})}$ for k, and a constant for L, into the actual space bound expression (3.1) yields the same asymptotic space bound of $n/2^{\Theta(\sqrt{\log n})}$. Since this matches the minimum for the simplified expression, which was a lower bound for this expression, we cannot do any better, and this must be the minimum space bound for the algorithm when using polynomial time.

The results of this section are summarized in the following theorem and its corollary:

Theorem 3.4 *The combined algorithm, described above, solves* STCON *in space* $O(n(\log n)/L^r + r(n/k + L \log k))$ *and time* $O(nk^{rL+2}(n/k + C))$, *for arbitrary integers* r, k, *and* L, *such that* $n \geq k \geq 1, r \geq 1, L \geq 1$, *and* $L^r \leq n$.

Choosing $r = \sqrt{\log n}$, $k = 2^{\Theta(\sqrt{\log n})}$, and $L = 2$ in the above theorem, we obtain:

Corollary 3.5 *The combined algorithm solves* STCON *in space* $n/2^{\Theta(\sqrt{\log n})}$ *and time* $n^{O(1)}$.

Chapter 4

CONCLUSIONS

4.1 Introduction

This chapter presents some observations on the difference between USTCON and STCON. In Section 4.2, we discuss some USTCON algorithms that appeared after the results of Chapter 2. These algorithms seem to provide evidence that USTCON is an easier problem than STCON. In Section 4.3, we discuss the techniques used by these USTCON algorithms and why they do not seem applicable to solving directed s-t connectivity. We point out what may be a difference between STCON algorithms and USTCON algorithms: STCON algorithms always seem to find a simple path from s to t. This does not seem to be true for small-space USTCON algorithms. We show that many path length problems, including finding the length of the shortest path from s to t in a directed or undirected graph, and solving the short paths problem to a sufficiently long distance in a directed or undirected graph, are NL-complete. We generalize this result to show a close relationship between algorithms for the short paths problem and for STCON: for a wide range of space bounds, an algorithm for the short paths problem to a sufficiently long distance yields an algorithm for STCON with the same asymptotic space bound. The reverse is nearly true: an STCON algorithm implies an algorithm for the short paths problem with nearly the same asymptotic space bound. We conclude in Section 4.4 with a list of open problems in the area of time-space tradeoffs for graph s-t connectivity.

4.2 Other Work on Undirected s-t Connectivity

As mentioned in Section 1.3, substantial progress has been made toward determining the deterministic complexity of USTCON since the results of Chapter 2 were first published [BR91]. First, Nisan [Nis92] showed that USTCON was in simultaneous

polynomial time and $O(\log^2 n)$ space by improving his earlier pseudorandom generator algorithm [Nis90]. More recently, Nisan *et al.* [NSW92] show how to solve USTCON in space $O(\log^{1.5} n)$ and time $O(n^{\sqrt{\log n}})$. The basic idea of this last result is similar to the recursive algorithm of Section 2.3 — the graph is repeatedly contracted by associating a set of vertices with one vertex. The set of vertices is found using the original pseudorandom generator of Nisan [Nis90]

4.2.1 A Variation of the $O(\log^{1.5} n)$ Space Algorithm

Karger *et al.* [KNP92] and Sinha and Tompa [ST] show that the resemblance between Nisan *et al.*'s $O(\log^{1.5} n)$ space algorithm and the recursive algorithm of Section 2.3 is more than superficial. They adapt the recursive algorithm's scheme of landmarks and neighborhoods to devise an algorithm with the same time and space bounds as Nisan *et al.*'s. The following paragraphs give a sketch of this algorithm.

Given a graph, G_0, with n vertices, we construct a graph, G_1, with $O(n/2^{\sqrt{\log n}})$ vertices that encodes the connectivity information of G_0. Let the *neighborhood* of a vertex v be the vertices found by taking the pseudorandom walks generated by the original pseudorandom generator of Nisan [Nis90] when given $\log n$ space. It can be shown that these walks hit at least $c2^{\sqrt{\log n}}$ distinct vertices, for some constant $c > 0$ (or all the vertices in v's component, if v's component contains fewer than $c2^{\sqrt{\log n}}$ vertices). The vertices of G_1 are the vertices, v, in G_0 such that

1. v's neighborhood is of size $c2^{\sqrt{\log n}}$.

2. v's neighborhood does not overlap the neighborhood of any other vertex $w, w < v$.

It isn't difficult to see that there are only $n/(c2^{\sqrt{\log n}})$ vertices in G_1. Vertices that are in components that are too small can safely be ignored, assuming s and t are not in such components. If s's or t's component is too small, the pseudorandom walks from s and t will discover such a fact, and can easily determine whether the two vertices are connected.

To define the edges of G_1, we must introduce the notion of *extended neighborhoods*. Every vertex, w, in G_0 is in an extended neighborhood. The extended neighborhood

is determined (recursively) as follows: If w is a vertex in G_1, then w is in its own extended neighborhood. Otherwise, let x be the lowest numbered vertex such that x's neighborhood in G_0 overlaps w's. Then w is in the same extended neighborhood that x is in.

There is an edge between two vertices in G_1, v_1 and v_2, if and only if there is an edge in G_0 between a vertex in the extended neighborhood of v_1 and a vertex in the extended neighborhood of v_2.

This contraction can be repeated on G_1 to get G_2, a graph with $O(n/c^2 2^{2\sqrt{\log n}})$ vertices with the same connectivity information, and so on. After $O(\sqrt{\log n})$ contractions, $G_{c'\sqrt{\log n}}$ has only a constant number of vertices, and s-t connectivity can easily be determined. Furthermore, the intermediate graphs do not need to be explicitly constructed; referring back to Section 1.5, computing the connectivity information for G_i is logspace reducible to computing the connectivity information for G_{i-1}, for any integer $0 < i \leq c'\sqrt{\log n}$. The algorithm can therefore be thought of as $c'\sqrt{\log n} + 1$ separate logarithmic space algorithms, $A_0, \ldots, A_{c'\sqrt{\log n}}$, where the output of each algorithm is available to the next algorithm on a bit by bit basis: A_i answers questions of the form "Is v a vertex in G_i?", or "Is there an edge between vertex v and vertex w in G_i?" If $i = 0$, the answers can be determined directly, and if $i > 0$, the answer can be determined by asking a series of similar questions to A_{i-1} about G_{i-1}. Each A_i requires only space $O(\log n)$, so the entire algorithm uses space $O(\log^{1.5} n)$. Because no more than $O(\sqrt{\log n})$ query results are retained at a time, the same query could be asked repeatedly (up to a superpolynomial number of times), and the running time is exponential in the space bound, or $n^{O(\sqrt{\log n})}$.

4.3 Undirected vs. Directed s-t Connectivity

Before this recent work on small space algorithms for USTCON, it was not clear that there was any substantial difference between directed and undirected s-t connectivity. The only algorithms that were asymptotically faster for undirected graphs were the random walk algorithm of Aleliunas *et al.*, and the nonuniform universal traversal sequences. The deterministic algorithms, depth- and breadth-first search, and Savitch's algorithm, are all asymptotically as fast for directed as undirected graphs. Ajtai and Fagin [AF90] show that STCON is harder than USTCON in symbolic logic, but their

result does not seem to translate into a computational complexity lower bound.

The algorithms of Chapter 2, however, along with those of Nisan [Nis92], and Nisan *et al.*, seem to provide evidence that USTCON is an easier problem than STCON. Note that the only known sublinear space, polynomial time algorithm for STCON, the algorithm of Section 3.5, does not even achieve space $O(n^{1-\epsilon})$, for constant $\epsilon > 0$, a far cry from the polylogarithmic space, polynomial time algorithm of Nisan.

A natural question, then, is why these algorithms work so well for undirected, but not directed graphs. The key to all these algorithms is that they exploit the symmetry of the connectivity relation in undirected graphs — in an undirected graph, if there is a path from u to v, then there is a path from v to u. This can be a useful property if we have only a limited amount of space, as these algorithms show. For example, consider the actions of a random walk on a graph. If the walk has a goal of hitting a distant vertex, t, then it seems inevitable that the walk will make an error and stray off the direct path to t. If the graph is undirected, this is not too bad, since the walk can always retrace its steps and return to the proper path. If the graph is directed, then the mistake is much more serious, since the walk may not be able to easily return to the proper path. Many of the small space algorithms for USTCON exploit the symmetry of undirected graphs by taking random or pseudorandom walks (see, for example, [AKL$^+$79, BKRU89, Nis90, Nis92, NSW92]).

The other common way of exploiting the symmetry of the connectivity relation on undirected graphs is to contract the graph by grouping a set of connected vertices into one (see the algorithms of Chapter 2, and [NSW92]). In such a scheme, the symmetry is important, because every vertex in such a set is connected to every other vertex in the set. In a directed graph, it may not be possible to find large strongly connected sets (sets where every vertex is reachable from every other member) — there may not be any large strongly connected components in the graph at all — and contraction may be ineffectual. If one tries to build sets with a weaker connectivity property, the grouping is not nearly as powerful — if two such sets overlap, one cannot conclude that all the vertices in the two sets are connected to each other.

There is another, more subtle difference between the small-space algorithms for USTCON and the algorithms for STCON: the STCON algorithms all find the distance from s to t. Breadth-first search, depth-first search, Savitch's algorithm, and the four

algorithms described in Chapter 3 can all be easily modified to give the distance from s to t along a simple path in G (and, with the exception of depth-first search, all these algorithms can easily be modified to find the *shortest* path from s to t). It is not easy to see how to do the same for the small-space undirected algorithms. For those that use random walks, extracting distances along simple paths seems impossible given the space bounds of the algorithms. For those that contract the graph, the information about simple paths is lost in the contraction. In the simple algorithm of Section 2.2, for example, we learn s is connected to t by discovering that they are in the same Union-Find set at the end of the algorithm. We could find a simple path from s to t if we knew the series of Unions that contributed to putting s and t into the same set, but this information is not discernible from the Union-Find data structure, and the straightforward scheme that retains such information during the execution of the Union-Find phase of the algorithm uses too much space.

This difference between small-space algorithms for USTCON and algorithms for STCON is not coincidental, and is related to the symmetry of connectivity in undirected graphs. Both STCON and USTCON algorithms try to find paths from s to t, but, in general, it doesn't seem profitable for an STCON algorithm to find anything but a simple path (particularly since that may be the only path from s to t). For USTCON algorithms, straying off the path doesn't seem so costly. In fact, the small-space algorithms for USTCON seem to show that it's sometimes much less costly to find an indirect, possibly circuitous route, than to confine oneself to the direct route.

4.3.1 The Importance of Finding Distances

Given the above observation, we might make the following hypothesis:

Conjecture 4.1 *Finding the length of a simple path between s and t is intrinsic to solving* STCON*, but not to solving* USTCON*.*

As worded, the conjecture is somewhat vague. What does it mean that finding a path is "intrinsic" to one problem, but not the other? For the moment, we will leave the meaning of the conjecture fuzzy. Throughout the remainder of this section, we will investigate different interpretations of the conjecture, and see if we can prove

them true. This can give us insight into the relationship between finding distances and solving STCON and USTCON.

Intuitively, when we say finding the length of a simple path is intrinsic to STCON, we mean that one cannot solve STCON without finding a simple path. If we have an algorithm for STCON, we ought to be able to make a few trivial changes, and get an algorithm that finds a simple path. Again, this concept is somewhat vague — what constitutes a "trivial" change? If we define a logspace Turing reduction to be trivial, we get the following mathematically precise version of the conjecture:

Conjecture 4.2 *Finding the length of a simple path between two vertices in a graph is logspace Turing reducible to* STCON*, but not* USTCON.

If one could prove Conjecture 4.2, it would show that STCON is harder than USTCON, and therefore that $NL \neq SL$. Unfortunately, it is unlikely that Conjecture 4.2 can be proved directly. Observing that a particular USTCON algorithm doesn't trivially yield the length of a simple path from s to t is much easier than showing that no algorithm can do so. Note, however, that we can prove the first half of the conjecture:

Lemma 4.3 *Finding the length of a simple path between two vertices in a graph is logspace Turing reducible to* STCON.

Proof: Without loss of generality, assume the graph is directed. The reduction for undirected graphs is similar. The proof is based on a simple transformation of a graph to a *layered graph*. Given a directed graph, G, we can convert it to a layered directed graph with n^2 vertices, G'. G' has n levels, level 0 to level $n-1$, and every vertex v has a copy, v_l, on every level l of G'. G' has edges only from level i to level $i+1$: (u_i, v_{i+1}) is an edge in G' if and only if (u, v) is an edge in G. In addition, there is an edge (u_i, u_{i+1}) for all vertices $u \in G$, and all $i, 0 \leq i < n-1$. Because edges only go from level i to level $i+1$, there is a path from u_0 to v_d in G' if and only if there is a path from u to v of length d or less in G. Note that this transformation can be done in logarithmic space.

Given G', we can find the length of the shortest path from s to t in G. Since the shortest path is a simple path, this will prove the lemma. To find the length of the

shortest path, test for all possible path lengths, k, $0 \leq k \leq n-1$, whether there is a path from s_0 to t_k. The length of the shortest path is the least k for which such a path exists. □

Note that the construction in the proof cannot be used to show that finding the length of a simple path is logspace Turing reducible to USTCON. If one builds a layered undirected graph, then, if s and t are in the same connected component in G, a USTCON algorithm can always find a path from s_0 to t_1 or t_0 using edges between levels 0 and 1.

We now investigate another aspect of Conjecture 4.1. Consider the following problem:

Definition: Given a directed graph, G, two vertices in G, s and t, and an integer k, the *length of the shortest path problem* (denoted LSP) is to determine whether the shortest path from s to t in G is of length k.

LSP is the "language" version of a natural problem — given two vertices in a directed graph, find the length of the shortest path between them. If, as Conjecture 4.1 maintains, finding the length of a simple path is intrinsic to an STCON algorithm, then the LSP problem, which finds the length of a simple path, may be closely related to STCON. It could be that a few changes to an LSP algorithm will yield an STCON algorithm. This turns out to be true.

Proposition 4.4 LSP *is NL-complete.*

Proof: An *NL* algorithm that solves LSP can be constructed using the layered graph of Lemma 4.3. First, verify that s_0 is connected to t_k in the layered graph using a standard *NL* STCON algorithm. This proves that the shortest path is of length k or less. Then, check that there is no path from s_0 to t_{k-1} in the layered graph using the *NL* s-t nonconnectivity algorithm of Immerman or Szelepcsényi [Imm88, Sze88]. This proves that the shortest path is not of length $k-1$ or less.

Given an algorithm for LSP, we can solve STCON in deterministic logarithmic space by constructing the corresponding layered graph, G', from G, and testing whether there is a path of length $n-1$ in G' from s_0 to t_{n-1}. □

Like LSP, the short paths problem (see Section 3.4) tries to find the distance from s to t, but it seems easier than LSP: while an LSP algorithm must determine whether the length of the shortest path is a certain number, a short paths algorithm needs only to determine whether the length of the shortest path is less than a sufficiently large bound. Still, if finding the distance from s to t is important when solving STCON, as Conjecture 4.1 suggests, perhaps the short paths problem is also closely related to STCON. We have already seen circumstantial evidence that this is true: the short paths algorithm described in Section 3.4 is actually a general STCON algorithm (see the note on page 53). The following theorem shows that, like the LSP problem, the short paths problem for certain functions is in some sense as hard as STCON:

Theorem 4.5 *Let $f(n)$ be a function such that $f(n) = \Omega(n^\epsilon)$, for some constant $\epsilon > 0$, and $f(n)$ is computable in logarithmic space. Then* SHORTP$(f(n))$ *is NL-complete.*

Proof: Without loss of generality, we prove the theorem for SHORTP(n^ϵ), constant $\epsilon > 0$. First, SHORTP(n^ϵ) is in NL. Simply guess a path of length at most n^ϵ edge by edge, checking that the guessed edges exist.

To show that SHORTP(n^ϵ) is NL-hard, let A be an algorithm that finds short paths of length up to n^ϵ. We show that STCON $\leq_{\log}$ SHORTP(n^ϵ) by recursively invoking A on itself to solve STCON. Given A and a graph, G, we can construct a graph, G_1, that has the same vertices as G, and an edge from u to v if and only if there is a path from u to v in G of length up to n^ϵ. We can repeat this process $\lceil 1/\epsilon \rceil$ times and generate $\lceil 1/\epsilon \rceil$ graphs, $G_1, \ldots, G_{\lceil 1/\epsilon \rceil}$, where G_i is a graph with the same vertices as G and an edge from u to v if and only if there is a path from u to v in G of length up to $n^{i\epsilon}$. $G_{\lceil 1/\epsilon \rceil}$ tells us whether there is a path from s to t in G.

As in the algorithm described in Section 4.2.1, we do not explicitly construct these intermediate graphs. We assume a chain of algorithms, $A_1, \ldots, A_{\lceil 1/\epsilon \rceil}$, each operating on one of the G_i graphs. A_i's job is to answer questions such as "Is there an edge from u to v in G_i?", which it answers by running A, periodically either querying A_{i-1} whether there is an edge from u' to v' in G_{i-1} (if $i > 1$), or reading the input graph, G (if $i = 1$). Each algorithm uses space $O(\log n)$, and there are only $\lceil 1/\epsilon \rceil$ algorithms, so the entire reduction uses space $O(\log n)$. □

A theorem similar to Theorem 4.5 can be proved using a padding argument. Given an algorithm for SHORTP($f(n)$), suppose for all values of n and a suitable constant c, we could always easily find an $n', n < n' < n^c$, such that $f(n') \geq n$. Then STCON $\leq_{\log}$ SHORTP($f(n)$). To solve STCON on a graph G with n vertices, construct a graph G' with n' vertices by adding $n' - n$ unconnected vertices to G. Then there is a path of length $f(n')$ or less in G' from s to t if and only if there is a path in G from s to t. It is not clear, however, that finding such an n' is always possible. If $f(n) = n^\epsilon$, constant $\epsilon > 0$, such an n' can be found by brute force: $\lceil n^{1/\epsilon} \rceil$ is a valid n', so if we test all numbers starting at n, we can find a suitable n' in polynomial time using $O(\log n)$ space.

The preceding two results show that finding the length of a shortest path in a directed graph, or even finding whether the length is less than a certain bound, is in some sense equivalent to solving STCON. We can try to extend Conjecture 4.1 further — if finding the length of a simple path between two vertices is so intrinsic to STCON, then perhaps finding the distance between two vertices in an *undirected* graph is closely related to STCON as well. The following two theorems show that this conjecture is, somewhat surprisingly, true. Let ULSP and USHORTP($f(n)$) be the versions of the LSP and SHORTP($f(n)$) problems for undirected graphs. According to Borodin *et al.* [BCD+89, page 561], Theorem 4.6 was originally discovered by Ladner.

Theorem 4.6 ULSP *is NL-complete.*

Theorem 4.7 *For constant* $1 > \epsilon > 0$, USHORTP(n^ϵ) *is NL-complete.*

Proof:[of Theorem 4.6] Given a directed graph, G, we can convert it to a layered undirected graph with n^2 vertices, G', similar to the layered graph constructed in the proof of Lemma 4.3. $\{u_i, v_{i+1}\}$ is an edge in G' if and only if (u, v) is an edge in G, and $\{u_i, u_{i+1}\}$ is an edge for all $u \in G$ and all $i, 0 \leq i < n - 1$. Because edges only go from level i to level $i + 1$, there is an undirected path from u_0 to v_d of length d in G' if and only if there is a path from u to v of length d or less in G.

ULSP, like LSP, is clearly in NL. We will show STCON $\leq_{\log}$ ULSP. Construct the undirected layered graph, G', for G. There is a path in G' between s_0 and t_{n-1} of length $n - 1$ if and only if there is a path in G from s to t. □

Proof:[of Theorem 4.7] USHORTP(n^ϵ), like SHORTP(n^ϵ), is easily shown to be in NL. To show the problem is NL-hard, we use a padding argument similar to the one presented above (page 65) to show that STCON $\leq_{\log}$ USHORTP(n^ϵ).

The basic idea of the reduction is to build the layered undirected graph, G', from G, as in the proof of Theorem 4.6 above, and then add extra, unconnected vertices to give n' total vertices, such that $\lfloor (n')^\epsilon \rfloor = n$. Then there is path in the new graph from s_0 to t_{n-1} of length n^ϵ or less if and only if there is a path from s to t in G.

There is a slight problem with this idea: the layered undirected graph already has more than n vertices. If the number of vertices in G' is large enough, then $\lfloor (n')^\epsilon \rfloor$ could be more than n. It is important that $\lfloor (n')^\epsilon \rfloor$ be exactly n. If $\lfloor (n')^\epsilon \rfloor$ is less than n, the USHORTP(n^ϵ) algorithm will not find a path in G' corresponding to a long path in G. If $\lfloor (n')^\epsilon \rfloor$ is more than n, the USHORTP(n^ϵ) algorithm may find a path in G' that does not correspond to a path in G.

If $\epsilon \leq 1/2$, the original idea can be used, as follows. We add vertices to the layered graph for a total of n' vertices, $\lfloor (n')^\epsilon \rfloor = n$. Note that because n^ϵ is a strictly increasing function, and because $k^\epsilon - (k-1)^\epsilon < 1$ for $0 < \epsilon < 1/2$, there is always an $n' \geq n^2$ such that $\lfloor (n')^\epsilon \rfloor = n$. We know that n' is approximately $n^{1/\epsilon}$, and while calculating $1/\epsilon$ and $n^{1/\epsilon}$ may not be easy in logarithmic space, we can again use brute force. If we test all numbers starting with n^2, we find the correct value of n' in polynomial time using $O(\log n)$ space.

Suppose $\epsilon > 1/2$. Then we use a different idea to lengthen the path to t_{n-1} while increasing the size of the graph. We connect k new vertices, $v_1, v_2, \ldots, v_k$, in a chain to s_0, by adding the edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}, \{v_k, s_0\}$. The chain must be the proper length, so that $\lfloor (n^2+k)^\epsilon \rfloor = n + k$. Then there is a path from v_1 to t_{n-1} in G' of length $\lfloor (n^2+k)^\epsilon \rfloor$ or less if and only if there is a path from v_1 to t_{n-1} of length $n + k$, which can only be true if there is a path from s to t in G.

Note that as k increases, $n + k$ grows faster than $(n^2+k)^\epsilon$, but $|(n + k - (n^2 + k)^\epsilon) - (n + k + 1 - (n^2 + k + 1)^\epsilon)| < 1$. Therefore, there is always a k such that $\lfloor (n^2+k)^\epsilon \rfloor = n + k$. For sufficiently large values of n, $k \leq n^{2+\alpha}$, for any constant $\alpha > 0$, so we can again find the correct value of k using brute force in polynomial time using $O(\log n)$ space.

Note that the above construction cannot be extended to all $f(n) = \Omega(n^\epsilon)$. For example, if $f(n) = n$, adding extra unconnected or connected vertices will not make the reduction work, since $f(n)$ grows as fast as the number of vertices added. □

The reductions in the proofs above show that if we could derive path length information from a small space algorithm for USTCON, we would have a corresponding small space algorithm for STCON. The space bound of the STCON algorithm may not be the same as the space bound of the USTCON algorithm, because the layered graph construction squares the size of the graph.

The Short Paths Problem and STCON

The following two propositions use the reductions above to show that STCON and SHORTP($f(n)$) are nearly equivalent for a wide range of space bounds. This information could be useful when trying to improve the results of Chapter 3.

Proposition 4.8 *If* SHORTP(n^α) $\in DTIME, SPACE(T(n), S(n))$, *for any* $0 < \alpha < 1$, $T(n) = n^{\Omega(1)}$, *then* STCON $\in DTIME, SPACE(T(n)^{O(1/\alpha)}, S(n)/\alpha)$.

Proof: Given an algorithm for SHORTP(n^α), A, that runs in the appropriate time and space bounds, we can use the reduction in Theorem 4.5 to obtain an STCON algorithm. This algorithm is essentially the recursive application of A on itself to depth up to $\lceil 1/\alpha \rceil$, so it uses space $O(S(n)/\alpha)$ and time at most $T(n)^{O(1/\alpha)}$. □

Note that if α is constant, then the resulting STCON algorithm has the same asymptotic space bounds as the short paths algorithm. Furthermore, if the short paths algorithm runs in polynomial time, so does the resulting STCON algorithm.

Proposition 4.9 *If* STCON $\in DTIME, SPACE(T(n), S(n))$, *then* SHORTP($f(n)$) $\in DTIME, SPACE(T(n \cdot f(n)^\alpha)^{O(1/\alpha)}, S(n \cdot f(n)^\alpha)/\alpha)$, *for any* $0 < \alpha \leq 1$, *and any* $f(n)$ *computable in* $DTIME, SPACE(T(n \cdot f(n)^\alpha)^{O(1/\alpha)}, S(n \cdot f(n)^\alpha)/\alpha)$.

Proof: Consider the undirected layered graph, G', constructed from G (see the proofs of Lemma 4.3 and Theorem 4.6). Let A be an algorithm for STCON that runs

in $DTIME, SPACE(T(n), S(n))$. If we construct the first k levels of G', we can use A to detect paths of length k in G. We (implicitly) construct the first $f(n)^\alpha$ levels of G', which allows us to solve the short paths problem to distance $f(n)^\alpha$. Using the construction from the proof of Theorem 4.5, we can call this algorithm recursively $\lceil 1/\alpha \rceil$ times to solve the short paths problem to distance $f(n)$. As in Proposition 4.8, this translates into a series of recursive calls to depth $\lceil 1/\alpha \rceil$. Note, however, that the size of the intermediate graphs is $n \cdot f(n)^\alpha$, which slightly increases the time and space requirements of the STCON algorithm at each level of recursion. The complete algorithm runs in time $T(n \cdot f(n)^\alpha)^{O(1/\alpha)}$ and space $S(n \cdot f(n)^\alpha)/\alpha$. □

For certain space bounds, $S(n)$, (for example, n^ϵ, for constant $\epsilon > 0$) the resulting short paths algorithm uses asymptotically more space than the STCON algorithm, so this result is not quite as good as the result of Proposition 4.8. Still, these results show that s-t connectivity and the short paths problem are equivalent in many respects. Consider the following class of languages:

Definition: Let *DPNE* be the class of languages, L, such that L is computable in deterministic polynomial time and $O(n^\epsilon)$ space for some constant $0 \leq \epsilon < 1$.

Corollary 4.10 (to Propositions 4.8 and 4.9) STCON $\in$ *DPNE* $\iff$ SHORTP$(n^c) \in$ *DPNE, for constant* $c, 0 < c \leq 1$.

Note that Proposition 4.9 says something slightly stronger: STCON $\in$ *DPNE* $\Rightarrow$ SHORTP$(f(n)) \in$ *DPNE* for *any* function $f(n) = n^{O(1)}$ computable in the given time and space bounds.

Corollary 4.11 (to Propositions 4.8 and 4.9) *For* $S(n) = \log^{O(1)} n$, STCON $\in DTIME, SPACE(n^{O(1)}, S(n)) \iff$ SHORTP$(n^c) \in DTIME, SPACE(n^{O(1)}, S(n))$, *for constant* $c, 0 < c \leq 1$.

Again, if STCON $\in DTIME, SPACE(n^{O(1)}, S(n))$, then there is a polynomial time, $S(n)$ space short paths algorithm for *any* function $f(n) = n^{O(1)}$ computable in the given time and space bounds.

A plausible next step in settling the space complexity of STCON might be to show that STCON is solvable in polynomial time and $O(n^\epsilon)$ space. Corollary 4.10 says that

this is equivalent to showing that the short paths problem for distance n^c is solvable in polynomial time and $O(n^\epsilon)$ space. Corollary 4.11 shows that Cook's conjecture (STCON is not solvable in simultaneous polynomial time and polylogarithmic space [Coo79]) is true if and only if the short paths problem for distance n^c is also not solvable in simultaneous polynomial time and polylogarithmic space.

4.4 Future Work

The obvious open problems in this area are to improve the bounds on the current best algorithms. The algorithm of Nisan *et al.* [NSW92] seems to suggest that a deterministic $\Theta(\log n)$ algorithm for undirected s-t connectivity is possible. One more insight in this area could settle the deterministic space complexity of USTCON. It might be useful to look slightly higher than the algorithm of Nisan *et al.* to Nisan's polynomial time, $O(\log^2 n)$ algorithm. Currently, it is not known whether one can improve the $\approx n^{45}$ running time of this algorithm by, for instance, giving the algorithm more space. If one could devise such a scheme, perhaps the algorithm could be extended in the other direction to use *less* space and more time.

In contrast, the deterministic space complexity of directed s-t connectivity seems far from settled. The current bound of simultaneous $n/2^{\Theta(\sqrt{\log n})}$ space and polynomial time does not seem natural. The algorithm in Chapter 3 was devised using a small collection of simple but useful ideas for trading time for space while searching a graph. Any new tradeoff, when combined with the old ones, may yield a substantial improvement in the space bound.

Section 4.3.1 shows that the short paths problem is central to solving STCON in small space. The other tradeoff used in the algorithm, the breadth-first search tradeoff, does not seem nearly as important. If we view our algorithm as operating on the breadth-first search tree of s, then it becomes apparent that it uses breadth-first search to slice the graph into pieces, and the short paths algorithm to explore these pieces. Partitioning the graph into sets of vertices with a certain property seems a reasonable approach to solving STCON in small space (in our case, the property relates to the length of the shortest path from s to the vertices in the set). However, it is not clear that viewing the graph as a breadth-first search tree yields the best algorithm. Even if we do fix on the breadth-first search tree, it is not clear that remembering a

fraction of the levels in the tree is the most efficient way to partition the vertices.

While it has barely been touched upon in this thesis, the probabilistic complexities of USTCON and STCON are also interesting. The time bound of Broder *et al.*'s algorithm seems open to improvement — it probably should be $O(mn \log^c n/s)$ for some constant c, instead of $O(m^2 \log^c n/s)$ [Ruz]. More interesting would be time and space bounds for probabilistic algorithms for directed s-t connectivity. While randomness seems to help with USTCON, it is not clear that it helps to solve STCON.

Section 4.3.1 shows that finding the length of a shortest path between two vertices in an undirected graph is *NL*-complete. While existing small space USTCON algorithms do not easily give such a path length, they could give an approximation of the path length. It may be fruitful to investigate approximation problems for *NL*. For example, how well can existing small space USTCON algorithm be made to estimate the path length from s to t? Is estimating the length of an undirected shortest path to a certain accuracy equivalent to knowing the exact path length? The recent work on approximation algorithms for *NP*-complete problems by Arora and Safra, and Arora *et al.* [AS92, ALM+92] gives a new characterization of *NP*. Perhaps work on approximation problem for *NL* could yield a similar new understanding of *NL*.

Finally, it is worthwhile to consider the apparent differences between undirected and directed s-t connectivity, as outlined in Section 4.3. While it does not appear likely that one can prove directly that STCON is harder than USTCON, the approaches of the different algorithms are worth considering for the insight one can gain. For example, it seems self-evident that any algorithm that solves s-t connectivity must somehow consider all possible paths from s to t, and test for their existence. How, then, does one explain the success of the random walk process at solving undirected s-t connectivity? The deterministic pseudorandom algorithms of Nisan show that, in some sense, the random walk *does* consider all possible paths. Understanding the structure of random walks on undirected graphs would be a breakthrough indeed.

Bibliography

[AAR90] N. Alon, Y. Azar, and Y. Ravid. Universal sequences for complete graphs. *Discrete Applied Mathematics*, 27:25–28, 1990.

[Abb68] E. Abbey. *Desert Solitaire; A Season in the Wilderness*. McGraw-Hill, 1968.

[AF90] M. Ajtai and R. Fagin. Reachability is harder for directed than for undirected graphs. *Journal of Symbolic Logic*, 55:113–150, 1990.

[AKL+79] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th Annual Symposium on Foundations of Computer Science*, pages 218–223, San Juan, Puerto Rico, October 1979. IEEE.

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *33rd Annual Symposium on Foundations of Computer Science*, Pittsburgh, PA, October 1992. IEEE.

[AS92] S. Arora and S. Safra. Probabilistic checking of proofs. In *33rd Annual Symposium on Foundations of Computer Science*, Pittsburgh, PA, October 1992. IEEE.

[BBR+90] P. Beame, A. Borodin, P. Raghavan, W. L. Ruzzo, and M. Tompa. Time-space tradeoffs for undirected graph connectivity. In *31st Annual Symposium on Foundations of Computer Science*, pages 429–438, St. Louis, MO, October 1990. IEEE.

[BBRS92] G. Barnes, J. F. Buss, W. L. Ruzzo, and B. Schieber. A sublinear space, polynomial time algorithm for directed *s-t* connectivity. In *Proceedings,*

Structure in Complexity Theory, Seventh Annual Conference, Boston, MA, June 1992. IEEE. To appear.

[BCD+89] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal on Computing*, 18(3):559–578, June 1989. See also 18(6): 1283, December 1989.

[BKRU89] A. Z. Broder, A. R. Karlin, P. Raghavan, and E. Upfal. Trading space for time in undirected *s-t* connectivity. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 543–549, Seattle, WA, May 1989.

[BNBK+89] A. Bar-Noy, A. Borodin, M. Karchmer, N. Linial, and M. Werman. Bounds on universal sequences. *SIAM Journal on Computing*, 18(2):268–277, April 1989.

[BNS89] L. Babai, N. Nisan, and M. Szegedy. Multiparty protocols and logspace-hard pseudorandom sequences. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 1–11, Seattle, WA, May 1989.

[BR91] G. Barnes and W. L. Ruzzo. Deterministic algorithms for undirected *s-t* connectivity using polynomial time and sublinear space. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 43–53, New Orleans, LA, May 1991.

[Bri87] M. F. Bridgland. Universal traversal sequences for paths and cycles. *Journal of Algorithms*, 8(3):395–404, 1987.

[BRT89] A. Borodin, W. L. Ruzzo, and M. Tompa. Lower bounds on the length of universal traversal sequences. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 562–573, Seattle, WA, May 1989. To appear in *Journal of Computer and System Sciences*.

[BS83] P. Berman and J. Simon. Lower bounds on graph threading by probabilistic machines. In *24th Annual Symposium on Foundations of Computer Science*, pages 304–311, Tucson, AZ, November 1983. IEEE.

[Coh78] D.I.A. Cohen. *Basic Techniques of Combinatorical Theory.* John Wiley & Sons, 1978.

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, OH, May 1971.

[Coo79] S. A. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, pages 338–345, Atlanta, GA, April-May 1979.

[CR80] S. A. Cook and C. W. Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM Journal on Computing*, 9(3):636–652, August 1980.

[Hen65] F. C. Hennie. One-tape off-line Turing machine computations. *Information and Control*, 8(6):553–578, 1965.

[HU69] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata.* Addison-Wesley, 1969.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[HW89] S. Hoory and A. Wigderson. Universal sequences for expander graphs. Hebrew University, Jerusalem, December 1989.

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, October 1988.

[Ist88] S. Istrail. Polynomial universal traversing sequences for cycles are constructible. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 491–503, Chicago, IL, May 1988.

[Ist90] S. Istrail. Constructing generalized universal traversing sequences of polynomial size for graphs with small diameter. In *31st Annual Symposium on Foundations of Computer Science*, pages 439–448, St. Louis, MO, October 1990. IEEE.

[KNP92] D.R. Karger, N. Nisan, and M. Parnas. Fast connected components algorithms for the EREW PRAM. In *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 373–382, San Diego, CA, June 1992.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[KPS88] H. J. Karloff, R. Paturi, and J. Simon. Universal traversal sequences of length $n^{O(\log n)}$ for cliques. *Information Processing Letters*, 28:241–243, August 1988.

[KSS84] J. Kahn, M. Saks, and D. Sturtevant. A topological approach to evasiveness. *Combinatorica*, 4:297–306, 1984.

[LL76] R. E. Ladner and N. A. Lynch. Relativization of questions about log space computability. *Mathematical Systems Theory*, 10(1):19–32, 1976.

[LP82] H. R. Lewis and C. H. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19(2):161–187, August 1982.

[Nis90] N. Nisan. Pseudorandom generators for space-bounded computation. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 204–212, Baltimore, MD, May 1990.

[Nis92] N. Nisan. *RL* $\subseteq$ *SC*. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, pages 619–623, Victoria, B.C., Canada, May 1992.

[NSW92] N. Nisan, E. Szemerédi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *33rd Annual Symposium on Foundations of Computer Science*, Pittsburgh, PA, October 1992. IEEE.

[Pip80] N. Pippenger. Pebbling. In *Proceedings of the Fifth IBM Symposium on Mathematical Foundations of Computer Science*. IBM Japan, May 1980.

[Ruz] W. L. Ruzzo. Personal Communication.

[RV76] R. C. Rivest and J. Vuillemin. On recognizing graph properties from adjacency matrices. *Theoretical Computer Science*, 3(3):371–384, December 1976.

[Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[ST] R. Sinha and M. Tompa. Personal Communication.

[Sze88] R. Szelepcsényi. The method of forcing for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[Tar75] R. E. Tarjan. On the efficiency of a good but not linear set merging algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[Tom82] M. Tompa. Two familiar transitive closure algorithms which admit no polynomial time, sublinear space implementations. *SIAM Journal on Computing*, 11(1):130–137, February 1982.

[Tom90] M. Tompa. Lower bounds on universal traversal sequences for cycles and higher degree graphs. Technical Report 90-07-02, Department of Computer Science and Engineering, University of Washington, July 1990. To appear in *SIAM Journal on Computing*.