Deterministic Algorithms for Undirected s-t Connectivity Using Polynomial Time and Sublinear Space

Greg Barnes and Walter L. Ruzzo

Technical Report 91-06-02
September, 1991

Preliminary version appeared in *Proceedings of the* 23^{rd} *ACM Symposium on Theory of Computing*, New Orleans, LA, May 1991.

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

Deterministic Algorithms for Undirected s-t Connectivity Using Polynomial Time and Sublinear Space*

Greg Barnes and Walter L. Ruzzo

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

October 16, 1992

Abstract

The s-t connectivity problem for undirected graphs is to decide whether two designated vertices, s and t, are in the same connected component. This paper presents the first known deterministic algorithms solving undirected s-t connectivity using sublinear space and polynomial time. There is some evidence that such algorithms are impossible for the analogous problem on directed graphs. Our algorithms provide a nearly smooth time-space tradeoff between depth-first search and Savitch's algorithm. For n vertex, m edge graphs, the simplest of our algorithms uses space s, $O(n^{1/2}\log n) \leq s \leq O(n\log n)$, and time $O(((m+n)n^2\log^2 n)/s)$. We give a variant of this method that is faster at the higher end of the space spectrum. For example, with space $\Theta(n\log n)$, its time bound is $O((m+n)\log n)$, close to the optimal time for the problem. Another generalization uses less space, but more time: space $O((n^\epsilon \log n)/\epsilon)$, for $1/\log n \leq \epsilon \leq 1/2$, and time $n^{O(1/\epsilon)}$. For constant ϵ the time remains polynomial.

*E-mail addresses: greg@-, ruzzo@cs.washington.edu. Research Supported by NSF Grants CCR-8703196 and CCR-9002891. Portions of this work performed while the authors visited the University of Toronto Computer Science Department, whose hospitality is gratefully acknowledged. An extended abstract of these results appeared in 23rd STOC, 1991 [5].

1 Introduction

The *s-t* connectivity problem is a fundamental one, since it is the natural abstraction of many computational search processes, and a basic building block for more complex graph algorithms. In computational complexity theory, it has an additional significance: understanding its complexity is a key to understanding the relationships among deterministic, nondeterministic, and probabilistic space bounded complexity classes. In particular, the *s-t* connectivity problem for *directed* graphs (STCON) is the prototypical complete problem for nondeterministic logarithmic space [22]. The problem for *undirected* graphs (USTCON) is complete for the seemingly weaker "symmetric" variant of nondeterministic space bounded machines [18], and is known to be solvable by logarithmic space and polynomial time bounded randomized algorithms, even errorless ones [1, 7]. Any problem solvable deterministically in logarithmic space can be reduced to either STCON or USTCON — i.e., both are DLOG-hard [18, 22].

The fundamental open problem in this area remains whether deterministic, nondeterministic, and probabilistic space bounded complexity classes are equal or distinct. The complexity of USTCON sits squarely in the middle of this question. For example, if deterministic and nondeterministic space bounded classes are equal, then USTCON is solvable by a deterministic logarithmic space algorithm, and perhaps showing this would be an easier first step towards the main goal. In fact, considerable effort has been expended on this step, for example in studying and attempting to constructively generate universal traversal sequences [1, 2, 3, 4, 8, 9, 12, 13, 14, 15, 19, 25]. Alternatively, if deterministic and nondeterministic classes are distinct, then USTCON is a likely candidate for a problem that will separate the classes. In either case, its complexity is of interest.

Settling the deterministic space complexity of USTCON is a very difficult open problem. A fruitful intermediate step is to explore *time-space tradeoffs* for the problem: the *simultaneous* time and space requirements of algorithms for USTCON. Even for this simpler intermediate problem, no nontrivial lower bounds are known for general models of computation (such as Turing machines), although Cook and Rackoff [11] and Beame, *et al.* [6] have obtained lower bounds for restricted models. This paper presents new upper bounds

for the problem.

For probabilistic algorithms, much is known about the simultaneous time and space requirements for USTCON. The random walk result of Aleliunas, Karp, Lipton, Lovász, and Rackoff [1] provides a space-optimal, but somewhat slow algorithm for USTCON. At the other extreme, well-known deterministic methods like depth- and breadth-first search provide time-optimal, but space-intensive solutions. Both extremes exhibit a time-space product of $O(mn \log n)$. The probabilistic algorithm of Broder, Karlin, Raghavan, and Upfal [10] provides a spectrum of compromises roughly between these two extremes: time $O(m^2 \log^5 n/s)$ with space s.

For deterministic algorithms, less is known. For *directed* graphs, depth- and breadth-first search are still applicable, still time-optimal, and still require space $\Theta(n \log n)$ in the worst case. Savitch's Theorem [22] provides a deterministic $\Theta(\log^2 n)$ space algorithm for STCON, but it requires time exponential in its space bound: $n^{\Theta(\log n)}$. A similar but more subtle algorithm by Cook and Rackoff [11] for their more restricted "JAG" model has essentially the same performance. No sublinear space, polynomial time algorithm is known for STCON, and there is evidence suggesting that none is possible. Specifically, Tompa [24] has shown that certain natural approaches to solving STCON admit no such solution. Indeed, he shows that for these approaches, performance degrades sharply with decreasing space: space $o(n)$ implies time $n^{\Omega(\log n)}$, essentially as slow as Savitch's algorithm. Proving a nonpolynomial time bound in general would, of course, prove that STCON is not solvable deterministically in logarithmic space, separating the deterministic and nondeterministic space classes.

For undirected s-t connectivity, all of the algorithms cited in the previous paragraph apply as well, with the same complexity bounds. In addition, there is one sublinear space deterministic algorithm for USTCON that does not also solve STCON. Nisan [19], improving on results of Babai, Nisan and Szegedy [3], has shown how to "derandomize" the probabilistic algorithm of Aleliunas, *et al.* [1] to obtain a small-space deterministic algorithm for USTCON. It again has essentially the same performance as Savitch's: very small space, namely $\Theta(\log^2 n)$, but superpolynomial time, namely $n^{\Theta(\log n)}$. Also, there is one algorithm solving a restricted case of USTCON in $O(\log^2 n / \log\log n)$ space, namely the case of undirected graphs of $(\log n)$-

bounded genus, solved by Kriegel [17]. None of these results provides the sort of spectrum of compromises between time and space that Broder, *et al.* [10] give.

The main results of our paper are three new 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, and time-intensive, space-efficient algorithms such as Savitch's, Cook and Rackoff's, and Nisan's. The first algorithm, called the *simple* algorithm below, uses space s, $O(n^{1/2}\log n) \leq s \leq O(n\log n)$, 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 $O((n^\epsilon \log n)/\epsilon)$, for $1/\log n \leq \epsilon \leq 1/2$, and time $n^{O(1/\epsilon)}$. 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 s, $O(n^{1/2}\log n) \leq s \leq O(n\log n)$, 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. As noted in Bar Noy, *et al.* [4], no previously known deterministic algorithm for USTCON simultaneously achieves polynomial time and sublinear space.

Returning to the motivating questions about the relationships among the various space bounded complexity classes, note that if USTCON is solvable deterministically in logarithmic space, then, of course, it is solvable in logarithmic space and polynomial time simultaneously. Thus our sublinear space, polynomial time algorithm might be seen as a small step towards an affirmative answer to this question. On the other hand, our method runs into the same $\log^2 n$ space barrier as all previous methods, so a negative answer is still a distinct possibility.

The remainder of the paper is organized as follows. Sections 2, 3 and 4 present the simple, recursive, and batched algorithms, respectively. Section 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 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 $\mathsf{bbfs}(v, b)$ 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 [21], 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 are not 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 L of vertices called *landmarks*. One goal of the algorithm is to find enough landmarks l so that their associated neighborhoods, $\mathsf{bbfs}(l, b)$, nearly cover the graph, and so finding connected components will be quick. On the other hand, the landmark set 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 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 with weighted union and path compression [23]. 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 $v_1, v_2, \ldots, v_k$, after processing the $k-1$ edges $\{v_1, v_2\}$, $\{v_2, v_3\}$, $\ldots$, $\{v_{k-1}, v_k\}$ 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 last edge $\{v_i, v_{i+1}\}$ processed. 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 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 k^{th} edge processed. If Find($\mathsf{cl}(u)$) = Find($\mathsf{cl}(v)$) holds after $\{x, y\}$ is processed, but not before, then it must be that Find($\mathsf{cl}(u)$) and Find($\mathsf{cl}(v)$) were the two (distinct) sets joined when processing $\{x, y\}$. Without loss of generality, suppose Find($\mathsf{cl}(u)$) = Find($\mathsf{cl}(x)$) and Find($\mathsf{cl}(v)$) = Find($\mathsf{cl}(y)$). Then by induction u and x are connected, as are v and y, hence u and v are connected through edge $\{x, y\}$. □

A detailed "pseudocode" version of the algorithm is presented in Figure 1.

Next, we turn to the 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 [16]. 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. 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 takes time $O(b^2 ln \log b)$, using $O(b + l)$ registers. The Union-Find operations take time $O(m\alpha(m))$ [23] and use $O(l)$ registers. The cl routine is called $2m$ times, each call taking time $O(b^2 l \log b)$, so, assuming $m = \Omega(n)$, connecting the union-find sets dominates the algorithm's running time, taking time $O(b^2 lm \log b)$.

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 $O(bnm \log b)$. When $b = \sqrt{n}$, space is minimized, and the running time is $O(mn^{1.5} \log n)$. Increasing b from $\sqrt{n}$ increases both the space and time 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 $O(mn^2 \log n \log b/s)$, for a time-space product of $O(mn^2 \log n \log b)$. 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; see Broder *et al.* [10]), 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,*

```
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.


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 1: Details of the simple algorithm

presented above, solves USTCON *for arbitrary n vertex, m edge graphs in space $O(s)$, and time $O(((m+n)n^2 \log^2 n)/s)$.*

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, 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 ϵ, $1/\log_2 n \leq \epsilon \leq 1/2$, an approximate indication of the amount of space any given routine in the algorithm should use. We will assume ϵ is the reciprocal of a positive integer

— if not, the algorithm rounds ϵ down to the nearest integer reciprocal. The actual bound on space for any given routine (i.e., exclusive of recursive calls) will be $O(b)$ registers, where $b = \lceil n^\epsilon \rceil$, and the total space bound for the algorithm will be $O(b/\epsilon)$ 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 full, 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, page 5).

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,

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

- The $(k-1)$-neighborhoods of the vertices in $L_{v,k}$ are full, with the possible exception of v's $(k-1)$-neighborhood when $|L_{v,k}| = 1$.

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

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

If $L_{v,k}$ satisfies the above properties, and if the size of $L_{v,k}$ is b, then since the $(k-1)$-neighborhoods of the members of $L_{v,k}$ are of size b^{k-1}, we have encoded a full (k)-neighborhood using the desired amount of space. 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 full and 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 the v's (k)-neighborhood is always a subset of v's $(k+1)$-neighborhood. Furthermore, if v's $(k+1)$-neighborhood is full, so is its (k)-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 $(1/\epsilon - 1)$-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 $(1/\epsilon - 1)$-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 $\mathsf{nv}(\cdot, 1/\epsilon - 1)$. 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. $\mathsf{nv}(s, 1/\epsilon)$. However, guaranteeing connectivity turns out to be relatively expensive computationally, so the faster method of building union-find sets is used at the top level. This also insures that the simple algorithm is essentially a special case of this algorithm, i.e., the case $\epsilon = 1/2$.

The search procedure, nv, raises a problem that was not present with a simple breadth-first search: what if the search at a vertex fails to find enough (b^k) vertices? All our algorithms dismiss a vertex whose search is unsuccessful, because a lower bound on the size of the landmarks' neighborhoods is

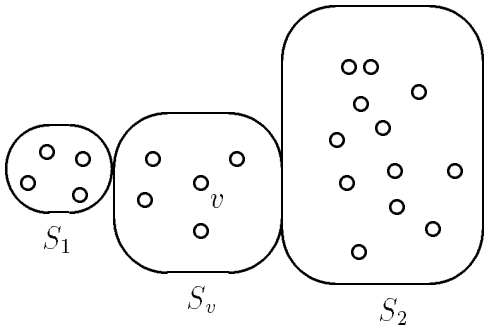

Figure 2: A graph where the search might fail.

necessary to insure an upper bound on the number of global landmarks. As mentioned in Section 2 (page 5), 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 depicting a connected graph, with S_v the $(k-1)$-neighborhood of v, S_1 a set of size less than b^{k-1}, and S_2 the rest of the graph, not directly connected to S_1. Suppose the algorithm is trying to find the (k)-neighborhood of v, has generated S_v, and is now looking for $b-1$ other $(k-1)$-neighborhoods. It will not find a large enough set in S_1, and may never be able to use these vertices, which may in the end cause the search at v to fail. Intuitively, though, a search for a (k)-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 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 3.1 *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 there are 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. First, note that v's (k)-neighborhood must be full, for otherwise by induction it would overlap w's $(k-1)$-neighborhood, contradicting disjointness. By a similar argument w's (k)-neighborhood is also full. Now, 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 $L_{v,k+1}$, the $(k+1)$-landmark set of v. But this contradicts the maximality of $L_{v,k+1}$. □

Given this lemma, we can show that at the topmost level, the algorithm works as desired. As in the simple algorithm, we first generate the $(1/\epsilon - 1)$-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 global landmark set. Finally, we examine all the edges as in the simple algorithm to determine which landmarks are in the same connected component. If the $(1/\epsilon - 1)$-neighborhoods of s and t are disjoint, we must be assured of two facts:

- If either is not full, then s and t are not connected. Proof: If s and t were connected, then by Lemma 3.1 their neighborhoods would intersect.

- If both are full, then for all vertices v, it must be true that v is in the global landmark set, or that v's $(1/\epsilon - 1)$-neighborhood overlaps a global landmark's $(1/\epsilon - 1)$-neighborhood, or that v is connected to neither s nor t. Proof: If v's neighborhood is full, then by construction it must either be a global landmark or its neighborhood must overlap the neighborhood of one of the global landmarks. If v's neighborhood is not full, then by Lemma 3.1 it cannot be connected to s or t without overlapping their neighborhoods.

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 k, for each possible pair $\{v, w\}$, $v \in S_1$, $w \in S_2$, the intersection routine generates the (k)-landmark sets of v and w, and calls itself recursively, with these two sets and the integer $k - 1$ as parameters. For the base case, $k = 0$, the sets are just arbitrary sets of vertices, and can be compared directly.

The code for the nv and recursive intersection (ri) routines is given in Figure 3. The code for the main routine of the recursive algorithm is nearly identical to the code for the simple algorithm (see Figure 1), and is omitted here. The only changes are as follows:

- The algorithm is parameterized by ϵ instead of b, where $0 < \epsilon \leq 1$. The constants $\hat{k}$ and b are defined, where $\hat{k} = \lceil 1/\epsilon \rceil$ and $b = \left\lceil n^{1/\hat{k}} \right\rceil$.

- All references to bbfs(x, b) are replaced by nv$(x, \hat{k} - 1)$. Similarly, all references to the intersection routine should be calls to ri$(\cdot, \cdot, \hat{k} - 2)$.

Analysis of the space complexity of the algorithm is easy: exclusive of recursive calls, each routine uses $O(n^\epsilon)$ registers, and the maximum recursion depth is $O(1/\epsilon)$, hence at most $O(n^\epsilon/\epsilon)$ registers are needed.

The following time analysis is based on the code for the recursive algorithm given in Figure 3. $R(\epsilon)$ is an expression bounding the running time of the algorithm for parameter ϵ, and $N(k)$ is a recurrence relation bounding the running time of nv$(\cdot, k)$. Because the sizes of the set parameters to ri can

```
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                                                    {Base case}
    return (S1 ∩ S2 ≠ ∅);
  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                                          {Base case. Use bbfs}
    return (bbfs(v, b));
  if |nv(v, k − 1)| < b then                             {nv(v, k − 1) not full}
    return ({v});
  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}. Find next landmark;
      end;
    end;
    return (L_{v,k});                        {Not full, but no other landmarks}
  end;
  return (L_{v,k});                                                  {Full}
end nv.
```

Figure 3: Details of the recursive algorithm.

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 $c_i, 0 \le i \le 11$, we have:

$$I(j) = \begin{cases} O(b \log b) & \text{if } j = 0 \\ b^2(I(j-1)+c_1) + b(b+1)(N(j)+c_2)+c_3 & \text{if } j > 0 \end{cases}$$

$$I_1(j) = b\,(I(j-1)+c_1) + (b+1)(N(j)+c_2)+c_3 \qquad j > 0$$

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

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

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

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

$$\begin{aligned} NI(k) &\le (b^2(2+c)m + b^2 + b)(NI(k-1)+c_{11}) \\ &= O((b^2((2+c)m+1)+b)^{k-1}(N(1)+I(0))) \\ &= O((b^2((2+c)m+1)+b)^{k-1} b^2 \log b) \end{aligned}$$

Recall that $\hat{k} = \lceil 1/\epsilon \rceil$ and $b = \left\lceil n^{1/\hat{k}} \right\rceil$. Note that $b \le n^\epsilon + 1 = O(n^\epsilon)$. Since $\epsilon \ge 1/\log_2 n$, note that

$$NI(\hat{k}-1) = O((n^{2\epsilon}(n+2m))^{\hat{k}-2} n^{2\epsilon} \log n^\epsilon).$$

Thus

$$R(\epsilon) = O((n+2m)(N(\hat{k}-1) + bN(\hat{k}-1) + bI(\hat{k}-2) + b) + m\alpha(m))$$

$$
\begin{aligned}
&= O((n+2m)bNI(\hat{k}-1)) \\
&= O((n+2m)n^{\epsilon}(n^{2\epsilon}(n+2m))^{\hat{k}-2}n^{2\epsilon}\log n^{\epsilon}) \\
&= O((n+2m)^{\hat{k}-1}n^{2\epsilon\hat{k}-\epsilon}\epsilon\log n)
\end{aligned}
$$

If $1/\epsilon$ is an integer, this gives:

$$R(\epsilon) = O(\epsilon(n+2m)^{1/\epsilon-1}n^{2-\epsilon}\log n)$$

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

Theorem 3.2 *The recursive algorithm, described above, solves* USTCON *for arbitrary n-vertex, m-edge graphs in space $O((n^{\epsilon}\log n)/\epsilon)$ and time $O(\epsilon(n+2m)^{1/\epsilon-1}n^{2-\epsilon}\log n) = n^{O(1/\epsilon)}$, for any $1/\log_2 n \leq \epsilon \leq 1/2$ with $1/\epsilon$ an integer.*

Note that when $\epsilon = 1/2$, this time bound matches that of the simple algorithm.

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 generating

the value of the cl function, the batched algorithm generates the neighborhoods for a batch of l/b vertices, B. The algorithm then sorts all vertices in the l/b neighborhoods into one list of 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 construct the intersection information for each member of B. To generate the value of the cl function, it must find the lowest numbered landmark whose neighborhood intersected a given vertex's neighborhood; to construct 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 actually just 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 4 (page 20), 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 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 effi-

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 4: The batched algorithm's data structure

ciently. 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 in the batched algorithm is given in Figure 5 (page 22). To evaluate the cl function for all edges, similar code is used — the main differences between the code in Figure 5 and code needed to evaluate the cl function are:

- The loop must process all edges, not all vertices, so the index runs from 1 to $2mb/l$, and B is a set of the endpoints of the next $l/2b$ edges. The interior of the loop functions as the cl procedure did in the simple algorithm.
- Before the loop labeled check intersection, the algorithm checks whether the vertex had a full neighborhood. If not, the special value SMALL is stored for this vertex.
- During the loop labeled check intersection, the algorithm determines the name of the lowest numbered landmark that intersected this vertex's neighborhood, instead of merely checking for intersection.
- After the loop, the value of the lowest numbered intersecting landmark is stored for this vertex. All code after the loop (from new landmark on), is unnecessary when evaluating the cl function.

Next we analyze the batched algorithm. It 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 $l = n/b$).

Finding the neighborhood of a vertex requires time $O(b^2 \log b)$, so finding 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 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}
  for all vertices, v ∈ B do begin
    generate bbfs(v, b);
  end;
  Created a sorted list, Q, of all vertices in the bbfs sets of the members of B.
  for all landmarks, l do begin                    {mark overlapping vertices}
    for all vertices, w ∈ bbfs(l, b) do begin
      if w is in Q and the cl field is unmarked then w's cl field = l.
    end;
  end;
  for all vertices, v ∈ B do begin                 {evaluate vertices in B}
    for all vertices, u ∈ bbfs(v, b) do begin        {check intersection}
      if u's cl field is marked in Q then
        v is not a landmark. Go to next vertex.
    end;
    Add v to the landmark set.                        {new landmark}
    for all vertices, u ∈ bbfs(v, b) do begin
      u's cl field in Q = v.
    end;
  end;
end;
```

Figure 5: Finding landmarks using 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 5 takes time $O(l/b \cdot b) = O(l)$. Similar analysis applies to the modified version of the code, used for evaluating 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.

Thus we have the following.

Theorem 4.1 *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 $1 \le b \le \sqrt{n}$.*

5 Conclusions and Future Work

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 $1/\epsilon$ is increased by one.

When the recursive algorithm is taken to extremes, its time and space bounds resemble those for Savitch's result [22]. The lower limit for the space

used by the algorithm is reached when $\epsilon = 1/\log n$. With this low value for ϵ, the algorithm uses space $O(\log^2 n)$ and time $n^{O(\log n)}$, similar to Savitch's space and time bounds [22].

There are three obvious open questions about s-t connectivity. First, for a given space bound, can our time bound be improved? Second, can the space bound for undirected s-t connectivity be reduced to $o(\log^2 n)$, i.e., can Savitch's Theorem be bettered for undirected graphs? Third, are sublinear space and polynomial time simultaneously achievable for *directed* s-t connectivity?

Recent work by Nisan [20] has made surprising progress on the first question. Using an extension of work by Babai, Nisan and Segedy [3] and Nisan [19] on pseudorandom generators, he shows that polynomial time and $O(\log^2 n)$ space are simultaneously achievable. Note that the space bound for his algorithm is again $\Theta(\log^2 n)$, making the second open question all the more intriguing.

The third open question is also interesting. Unlike well-known deterministic algorithms for USTCON, our algorithms cannot be immediately generalized to the directed case. This is because they rely heavily on the symmetry of the connectivity relation in undirected graphs; all three algorithms use overlapping sets of vertices to discover that two non-overlapping sets of vertices are connected. This method would fail miserably were the graph directed. However, studying STCON in another context, Ullman and Yannakakis [26] employed vaguely similar techniques involving multiple small searches, so generalization of our methods to directed graphs is not out of the question.

6 Acknowledgements

We thank Simon Kahan, who presented a similar problem that led to our discussion of this one. Also, Richard Anderson, Paul Beame, and Martin Tompa gave many helpful suggestions and comments.

References

[1] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff. Ran-

dom 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, Oct. 1979. IEEE.

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

[3] 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.

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

[5] 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.

[6] 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, Oct. 1990. IEEE.

[7] 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, Dec. 1989.

[8] 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*.

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

[10] 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.

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

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

[13] 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.

[14] 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, Oct. 1990. IEEE.

[15] 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, Aug. 1988.

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

[17] K. Kriegel. The space complexity of the accessibility problem for undirected graphs of $\log n$ bounded genus. In *Mathematical Foundations of Computer Science: Proceedings, 12th Symposium*, volume 233 of *Lecture Notes in Computer Science*, pages 484–492, Bratislava, Czechoslovakia, Aug. 1986. Springer-Verlag.

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

[19] 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.

[20] N. Nisan. $RL \subseteq SC$. Preprint, Hebrew University, Mar. 1991.

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

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

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

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

[25] 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.

[26] J. D. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. In *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 200–209, Crete, Greece, July 1990.