# A Sublinear Space, Polynomial Time Algorithm for Directed $s$-$t$ Connectivity

Greg Barnes *
Jonathan F. Buss †
Walter L. Ruzzo ‡
Baruch Schieber §

April 4, 1995

**Abstract**

Directed $s$-$t$ connectivity is the problem of detecting whether there is a path from vertex $s$ to vertex $t$ in a directed graph. We present the first known deterministic sublinear space, polynomial time algorithm for directed $s$-$t$ connectivity. For $n$-vertex graphs, our algorithm can use as little as $n/2^{\Theta(\sqrt{\log n})}$ space while still running in polynomial time.

## 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

*Max-Planck-Institut für Informatik, Im Stadtwald, W 6600 Saarbrücken, Germany. This research was conducted while the author was at the University of Washington. Research supported by NSF Grant CCR-9002891. Email address: barnes@mpi-sb.mpg.de.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. Research supported in part by a grant from NSERC. Email address: jfbuss@math.uwaterloo.ca.

‡Dept. of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195, U.S.A. Research supported by NSF Grant CCR-9002891. Email address: ruzzo@cs.washington.edu.

§IBM - Research Division, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, U.S.A. Email address: sbar@watson.ibm.com.

block for more complex graph algorithms. In computational complexity theory, it has an additional significance: understanding its complexity is a key to understanding the relationship between deterministic and nondeterministic space bounded complexity classes. In particular, the $s$-$t$ connectivity problem for *directed* graphs (STCON) is the prototypical complete problem for nondeterministic logarithmic space [7]. Both STCON and the undirected version of the problem, USTCON, are DLOG-hard — any problem solvable deterministically in logarithmic space can be reduced to either problem [4, 7].

Establishing the deterministic space complexity of STCON would tell us a great deal about the relationship between deterministic and nondeterministic space bounded complexity classes. For example, showing a deterministic log space algorithm for directed connectivity would prove that $\mathrm{DSPACE}(f(n)) = \mathrm{NSPACE}(f(n))$ for any constructible $f(n) = \Omega(\log(n))$ [7]. Unfortunately, this remains a difficult open problem. A fruitful intermediate step is to explore *time-space tradeoffs* for STCON; that is, the *simultaneous* time and space requirements of algorithms for directed connectivity. No nontrivial lower bounds are known for general models of computation (such as Turing machines) on either the space, or on the simultaneous space and time required to solve STCON, although Cook and Rackoff [3] and Tompa [8] have obtained lower bounds for restricted models. This paper presents new upper bounds for the problem.

The standard algorithms for connectivity, breadth- and depth-first search, run in optimal time $\Theta(m+n)$ and use $\Theta(n \log n)$ space. At the other extreme, Savitch's Theorem [7] provides a small space ($\Theta(\log^2 n)$) algorithm that requires time exponential in its space bound (i.e., time $n^{\Theta(\log n)}$). Cook and Rackoff show an algorithm for their more restricted "JAG" model that is similar to, but more subtle than Savitch's; it has essentially the same time and space performance.

Recent progress has been made on the time-space complexity of USTCON. Barnes and Ruzzo [1] show the first sublinear space, polynomial time algorithms for undirected connectivity. Nisan [5] shows that $O(\log^2 n)$ space and polynomial time suffice. Nisan *et al.* [6] show the first USTCON algorithm that uses less space than Savitch's algorithm ($O(\log^{1.5} n)$ vs. $\Theta(\log^2 n)$).

Prior to the present paper, there was no corresponding sublinear space, polynomial time algorithm known for STCON, and there was some evidence suggesting that none was possible. It has been conjectured [2] that no deterministic STCON algorithm can run in simultaneous polynomial time and polylogarithmic space. Tompa [8] shows 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 superpolynomial time, and space $n^{1-\epsilon}$ for fixed $\epsilon > 0$ implies time $n^{\Omega(\log n)}$, essentially as slow as Savitch's algorithm.

The main result of our paper is a new deterministic algorithm for directed $s$-$t$ connectivity that achieves polynomial time and sublinear space simultaneously. While not disproving the conjecture of [2], it shows that the behavior elicited from certain algorithms by Tompa is not intrinsic to the problem. Our algorithm can use as little as $n/2^{\Theta(\sqrt{\log n})}$ space while still running in polynomial time. As part of this algorithm, we present 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. We are not aware of any previous algorithms that solve this problem in sublinear space and polynomial time. 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.

Our algorithm to solve STCON in polynomial time and sublinear space is constructed from two algorithms with different time-space tradeoffs. The first performs a modified breadth-first search of the graph, while the second finds short paths. Alone, neither algorithm can solve STCON in simultaneous polynomial time and sublinear space. In the following two sections, we present the breadth-first search algorithm and the short paths algorithm. Section 4 shows how the two algorithms can be combined to yield the desired result. Section 5 presents some notes and concluding remarks.

For more information on graph connectivity, see the survey paper by Wigderson [9].

## 2 The Breadth-First Search Tradeoff

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 $n/\lambda$ vertices. We can do this by only storing (the vertices in) every $\lambda$th level of the tree. Number the levels of the tree 0, 1, ..., $n-1$, where a vertex $v$ is on level $l$ if the shortest path from $s$ to $v$ is of length $l$. Divide the levels into equivalence classes $C_0, C_1, \ldots, C_{\lambda-1}$ based on their number mod $\lambda$. 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, $n/\lambda$.

The algorithm constructs this partial tree one level at a time. It begins with level 0, which consists of $s$ only, and generates levels $j$, $j+\lambda$, $j+2\lambda$, $\ldots$, $j+\lambda \cdot \lfloor n/\lambda \rfloor$. Given a set, $S$, of vertices, we can find all vertices within distance $\lambda$ of $S$ in time $n^{O(\lambda)}$ and space $O(\lambda \log n)$ by enumerating all possible paths of length at most $\lambda$ 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 $0, j, j+\lambda, \ldots, j+i\lambda$. Consider the set of vertices, $U$, that are within distance $\lambda$ 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 $\lambda$ but not $\lambda-1$ of $V_i$.

Pseudocode for the algorithm appears in Figure 1. Note that to find an equivalence class with at most $n/\lambda$ vertices, the algorithm just tries all classes in order, discarding a class if it generates too many vertices.

Referring to Figure 1, the algorithm's space bound is dominated by the number of vertices in $S$ and $S'$, and the space needed to test whether a vertex is within distance $\lambda$ of a vertex in $S$. There are never more than $n/\lambda+1$ vertices in $S$ and $S'$, so the algorithm uses $O((n \log n)/\lambda)$ space to store these vertices. The time bound is dominated by repeatedly testing whether a vertex is within distance $\lambda$ of a vertex in $S$. This test is performed $O(n^3/\lambda)$

```
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 1: Details of the breadth-first search algorithm

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 2.1** *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 $O((n \log n)/\lambda + S_{PATH}(\lambda, n))$ and time $O((n^3/\lambda) \cdot T_{PATH}(\lambda, n))$, where $S_{PATH}(\lambda, n)$ and $T_{PATH}(\lambda, n)$ denote the space and time bounds, respectively, of the algorithm used to test for a path of length at most $\lambda$ between two vertices in an n-vertex graph.*

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 or space than testing for a path of length at most $\lambda$. This is because the first three problems are easily reduced to the latter. To test for a path of length at most $\lambda'$, $\lambda' < \lambda$, from some vertex in a set $S$ to some vertex $v$, connect $\lambda - \lambda'$ new vertices, $v_1, v_2, \ldots, v_{\lambda-\lambda'}$, in a chain to $v$ by adding the edges $(v, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{\lambda-\lambda'-1}, v_{\lambda-\lambda'})$. There will be a path in the new graph from some vertex $u \in S$ to $v_{\lambda-\lambda'}$ of length at most $\lambda$ if and only if there was a path in the original graph from $u$ to $v$ of length at most $\lambda'$.

Using a straightforward enumeration of all paths, testing whether a vertex is within distance $\lambda$ requires $n^{O(\lambda)}$ time and $O(\lambda \log n)$ space. This algorithm is not sufficient for our purposes. In particular, if $\lambda$ 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 $\lambda$ from a vertex is at most $d^\lambda$. For these graphs, $\lambda$ 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 $\lambda$ does.

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

# 3 The Short Paths Tradeoff

Consider the *short paths problem.*

**Definition 3.1** *Given a directed graph, $G$, and two distinguished vertices, $s$ and $t$, the* short paths problem for function $f(n)$ *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 retain 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 $i$th 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 2. Notice that the algorithm as given does not solve the short paths problem, as it tests for the existence of a path from $s$ to $t$ of length *exactly* $L$, not *at most* $L$. Any such algorithm can easily be converted into an algorithm for the short paths problem by adding a self-loop to $s$. For simplicity, we omit this detail from our algorithms.

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 $D$ be the maximum number of edges from one set of vertices $d_i$ to another set of vertices $d_j$ ($i$ and $j$ can be the same). For all steps in each path, we do at most $O(n/k + D)$ work zeroing the vector and testing for edges from $d_{i-1}$ to $d_i$. Since $D = O(n^2/k^2)$, the algorithm uses $O(k^L L \cdot n^2/k^2) = O(k^L n^3)$ 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

```
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 ⌈n/k⌉ bit vectors.
  for all (L+1)-digit numbers in base k,
      ⟨d_0 = s mod k, d_1, ..., d_{L-1}, d_L = t mod k⟩ 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 mod 2} to zero.
                                                  {Find edges from d_{i-1} to d_i}
      for all u in d_{i-1} marked in V_{(i-1) mod 2} and all v in d_i do begin
        if (u, v) is an edge then
          mark v in V_{i mod 2}.
      end;
    end;
    if t is marked in V_{L mod 2} then return (CONNECTED);
  end;
  return (NOT CONNECTED);
end SP.
```

Figure 2: Details of the short paths algorithm

an algorithm that uses twice as much space, but finds paths of length $L^2$. In general, using $r \geq 1$ 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 bound of $O((k^L L \cdot n^2/k^2)^r) = O(n^{2r+1} k^{rL})$, since $L^r = O(n)$. In Figure 3, we present the pseudocode for our recursive algorithm. This algorithm uses a further refinement to improve the time bound — one recursive call is used to find all vertices in $d_i$ reachable from any reachable vertex in $d_{i-1}$.

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

**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.
  **if** $r = 0$ **then** {base case}
    **for all** $u$ in $d_s$ marked in $V_s$ **and all** $v$ in $d_t$ **do begin**
      **if** $(u, v)$ is an edge **then**
        mark $v$ in $V_t$.
    **end**;
  **else**
    **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 paths from $d_{i-1}$ to $d_i$}
        $V_{i \bmod 2} = \mathsf{SPR}(k, L, r-1, d_{i-1}, d_i, V_{(i-1) \bmod 2})$.
      **end**;
      Set all bits in $V_t$ that are set in $V_{L \bmod 2}$.
    **end**;
  **return** $(V_t)$;
**end** SPR.

Figure 3: Details of the recursive short paths algorithm

constant $c$,

$$T(j) = \begin{cases} O(n^2/k^2) & \text{if } j = 0. \\ k^L L(T(j-1) + cn/k) & \text{if } j > 0. \end{cases}$$

In the base case, the algorithm does $O(n^2/k^2)$ 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 $O((k^L L)^r \cdot n^2/k^2) = O(n^3 k^{rL})$.

In summary, we have shown the following.

**Theorem 3.2** *For arbitrary integers $r$, $k$, and $L$, such that $r \geq 1$, $L \geq 1$, $n \geq k \geq 1$, and $L^r \leq n$, the recursive short paths algorithm, presented above, can search to distance $L^r$ in time $O(k^{rL} L^r \cdot n^2/k^2)$ $(= O(n^3 k^{rL}))$ and space $O(r(n/k + L \log k))$.*

## 3.1 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 $O(rn/\log^r n)$. However, as mentioned in the introduction, this algorithm does not by itself give a polynomial time, sublinear space algorithm for STCON. 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.

The algorithm, which was designed to solve the short paths problem, 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 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.

# 4 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, if we set $\lambda$ in the breadth-first search algorithm to be $L^r$, 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 3.2) for the term $S_{PATH}(\lambda, n)$ in the breadth-first search algorithm (see Theorem 2.1), we get a space bound for this algorithm of

$$O((n \log n)/L^r + r(n/k + L \log k)), \tag{1}$$

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 $T_{PATH}(\lambda, n)$ in the breadth-first search time bound gives a time bound of

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

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 algorithm, and $Q$ is a subset of one of the $k$ sets, one call to the short paths algorithm can be used to find all vertices in $R$ within distance $L^r$ of a vertex in $Q$. Thus, the short paths algorithm only needs to be called $2k^2$ times to generate the next level of the tree, twice for each possible pair of the $k$ sets in the short paths algorithm. Figure 4 gives the code that should be used in place of the loop in Figure 1 (marked with a $\star$) that finds vertices on the next level of the breadth-first search tree. Similar code should replace the earlier loop in Figure 1 that finds the vertices on the first level.

The improved version makes a total of $O(k^2 n/L^r)$ calls to the short paths algorithm, for a time bound of

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

We want to find the minimum amount of space the algorithm can use while still maintaining a polynomial running time. To maintain polynomial

```
                  {S is the set of vertices on previous tree levels. S' (initially
                   the empty set) will be the set of vertices on the next level}
for i_1 = 0 to k - 1 do begin
   S_{i_1} = {all vertices whose vertex number mod k = i_1}.
   P = ∅.                      {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 ∩ 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}.
      P = P ∪ (A - B).
   end;
   if |S| + |S' ∪ P| > n/L^r then try next j.
   else S' = S' ∪ P.
end;
```

Figure 4: Combining the two algorithms efficiently.

time, we must have, for some constant, $a$,

$$k^{rL} = n^a. \tag{2}$$

For simplicity, we bound expression (1) from below by

$$\Omega(n/L^r + n/k). \tag{3}$$

(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.) The minimum value of the bound (3) is reached when the denominators are equal. For any given $k$, the product $rL$ is fixed; thus the quantity $L^r$ reaches its maximum, and the bound reaches its minimum, when $L$ is a constant. Substituting $L^r$ for $k$ in Equation 2 and solving for $r$ yields $r = \sqrt{(a/L)\log_L n} = \Theta(\sqrt{\log n})$, and thus $k = 2^{\Theta(\sqrt{\log n})}$.

Substituting these values, $\sqrt{\log n}$ for $r$, $2^{\Theta(\sqrt{\log n})}$ for $k$, and a constant for $L$, into the simplified space bound expression (3) gives a bound of $n/2^{\Theta(\sqrt{\log n})}$. Substituting these same values into the actual space bound expression (1) yields the same asymptotic space bound, $n/2^{\Theta(\sqrt{\log n})}$. 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 4.1** *The combined algorithm, described above, solves* STCON *in space $O((n \log n)/L^r + r(n/k + L \log k))$ and time $O(n^3 k^{rL})$, for any integers $r, k$, and $L$ that satisfy $n \geq k \geq 1, r \geq 1, L \geq 1$, and $L^r \leq n$.*

**Corollary 4.2** *The combined algorithm can solve* STCON *in time $n^{O(1)}$ and space $n/2^{\Theta(\sqrt{\log n})}$.*

**Proof:** Choose $r = \sqrt{\log n}$, $k = 2^{\Theta(\sqrt{\log n})}$, and $L = 2$ in Theorem 4.1. As discussed above, these choices minimize space while retaining polynomial time. □

# 5 Conclusions and Future Work

The obvious open problem raised by this work is to improve our algorithm to use less space while maintaining polynomial time. There is good reason to believe that this is possible. First of all, the current bound of simultaneous $n/2^{\Theta(\sqrt{\log n})}$ space and polynomial time is not aesthetically appealing. More concretely, our algorithm 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.

Of the two, the breadth-first search tradeoff seems more open to improvement or even replacement. 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.

The short paths problem seems more central to solving STCON in sublinear space. Many of the difficulties one faces when designing small space STCON algorithms exist for the short paths problem as well. As noted in Section 3.1, our algorithm designed to solve the short paths problem actually turns out to be a general algorithm for $s$-$t$ connectivity. Any improvement to the short paths algorithm would probably be very useful in designing future small space STCON algorithms.

Our algorithm for STCON does not perform nearly as well as the recent sublinear space algorithms for USTCON by Barnes and Ruzzo [1], Nisan [5], and Nisan *et al.* [6]. This may be due to a fundamental difference between connectivity on directed and undirected graphs. The results of Barnes and Ruzzo, and of Nisan *et al.* exploit the symmetry of undirected graphs to group many vertices into one vertex that has the same connectivity properties. Nisan and Nisan *et al.* use techniques based on a random walk on a graph, a process that is surprisingly efficient for discovering connectivity in undirected graphs, but woefully inadequate on directed graphs. Generalizing these algorithms to directed graphs, or finding a directed graph property that can be similarly exploited, might yield an improved algorithm for STCON (and USTCON). In the absence of such shortcuts, it seems likely that future algorithms must also rely on enumerating all possible paths in limited time and space.

# Acknowledgements

Allan Borodin pointed us toward the short paths problem. Uri Feige helped find the minimum space bound.

# References

[1] 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. To appear *Computational Complexity.*

[2] 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, Apr.-May 1979.

[3] 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.

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

[5] 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.

[6] 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, Oct. 1992. IEEE.

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

[8] 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.

[9] A. Wigderson. The complexity of graph connectivity. In *Mathematical Foundations of Computer Science: Proceedings, 17th Symposium*, volume 629 of *Lecture Notes in Computer Science*, pages 112–132, Prague, Czechoslovakia, Aug. 1992. Springer-Verlag.

RCS Revision: 2.25 Date: 1993/04/28 20:20:35 .