SIAM J. COMPUT. © 1998 Society for Industrial and Applied Mathematics
Vol. 27, No. 5, pp. 1273–1282, October 1998 004

A SUBLINEAR SPACE, POLYNOMIAL TIME ALGORITHM FOR DIRECTED *s-t* CONNECTIVITY[*]

GREG BARNES[†], JONATHAN F. BUSS[‡], WALTER L. RUZZO[§], AND BARUCH SCHIEBER[¶]

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.

Key words. graph connectivity, *s-t* connectivity, graph reachability, time-space tradeoff, JAG, NNJAG, NL

AMS subject classifications. 05C40, 05C85, 68Q05, 68Q10, 68Q15, 68Q20, 68Q25

PII. S0097539793283151

1. Introduction.

The *s-t* connectivity problem, detecting whether there is a path from a distinguished vertex s to a distinguished vertex t in a directed graph, 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 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 [12]. 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 [7, 12].

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 logarithmic space algorithm for directed connectivity would prove that $\mathrm{DSPACE}(f(n)) = \mathrm{NSPACE}(f(n))$ for any constructible $f(n) = \Omega(\log(n))$ [12]. 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 the simultaneous space and time required to solve STCON, although Cook and Rackoff [5] and Tompa [13] have obtained lower bounds for restricted models. This paper presents new upper bounds for the problem.

[*]Received by the editors May 11, 1993; accepted for publication (in revised form) July 18, 1996; published electronically May 19, 1998.
http://www.siam.org/journals/sicomp/27-5/28315.html

[†]Computing and Communications, University of Washington, Box 354843, Seattle, WA 98195–4843 (gsbarnes@u.washington.edu). This research was supported by NSF grant CCR-9002891.

[‡]Department of Computer Science, University of Waterloo, Waterloo, ON, Canada N2L 3G1 (jfbuss@math.uwaterloo.ca). This research was supported in part by a grant from NSERC.

[§]Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195–2350 (ruzzo@cs.washington.edu). This research was supported by NSF grant CCR-9002891.

[¶]IBM Research Division, T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (sbar@watson.ibm.com).

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 [12] 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 [3] show the first sublinear space, polynomial time algorithms for undirected connectivity. Nisan [8] shows that $O(\log^2 n)$ space and polynomial time suffice. Nisan, Szemerédi, and Wigderson [9] show the first USTCON algorithm that uses less space than Savitch's algorithm ($O(\log^{1.5} n)$ versus $\Theta(\log^2 n)$).

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 [4] that no deterministic STCON algorithm can run in simultaneous polynomial time and polylogarithmic space. Tompa [13] 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 [4], 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.

Subsequent to the appearance of a preliminary version of this paper [1], Poon [11, Chap. 4] has shown an algorithm for the JAG model that is similar to this algorithm and achieves similar performance. In addition, Barnes and Edmonds [2] and Edmonds and Poon [6] have given improved lower bounds for STCON on the JAG model and the stronger Node-Named JAG (*NNJAG*) model of Poon [10]. Edmonds and Poon's lower bound is particularly strong, since it nearly matches the upper bounds in this paper when the machine uses space $n/2^{O(\sqrt{\log n})}$ or less, and thus suggests that the algorithm in this paper could provide an optimal time-space tradeoff for STCON.

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 [15].

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/λ vertices. We can do this by only storing (the vertices in) every λth level of the tree. Number the levels of the tree $0, 1, \ldots, n-1$, 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 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, $j+\lambda$, $j+2\lambda$, $\ldots$, $j+\lambda \cdot \lfloor n/\lambda \rfloor$. 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, $j+\lambda$, $\ldots$, $j+i\lambda$. 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 2.1. 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 2.1, 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 λ 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 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 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 λ 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 λ. This is because the first three problems are easily reduced to the latter. To test for a path of length at most λ', $\lambda' < \lambda$, 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 λ if and only if there was a path in the original graph from u to v of length at most λ'.

```
Algorithm Bfs (integer: $\lambda$);
                              {remember every $\lambda$th level of the breadth-first search tree}
  for $j = 0$ to $\lambda - 1$ 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 $|S| > n/\lambda$ then try next $j$.   {Don't store more than $n/\lambda$ vertices, + vertex $s$}
        else add $v$ to $S$.
    end;
    for $i = 1$ to $\lfloor n/\lambda \rfloor$ do begin
      $S' = \emptyset$.
      for all vertices, $v$ do begin                   {Find vertices on the next level. $\star$}
        if $v$ within distance $\lambda$ of some vertex in $S$ and
           $v$ not within distance $\lambda - 1$ of any vertex in $S$ then
          if $|S| + |S'| > n/\lambda$ then try next $j$.
          else add $v$ to $S'$.
      end;
      $S = S \cup S'$.
    end;
    if $t$ within distance $\lambda$ of a vertex in $S$ then return (CONNECTED);
    else return (NOT CONNECTED);
  end;
end Bfs.
```

FIG. 2.1. *Details of the breadth-first search algorithm.*

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. 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, 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 *bounded s-t connectivity problem* (*bounded* STCON).

DEFINITION 3.1. *For any real-valued function $f(n)$, the $f(n)$*-bounded *s-t* connectivity problem *is, given an n-vertex directed graph G and two distinguished vertices s and t, to determine whether there is a path in G from s to t of length less than or equal to $f(n)$.*

Solving bounded STCON for general $f(n)$ is at least as hard as solving STCON. We are interested in the *short paths problem*, informally defined as the bounded STCON problem where $f(n)$ is small. 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 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 3.1. 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

```
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 $k$,
      $\langle d_0 = s \bmod k, d_1, \dots, d_{L-1}, d_L = t \bmod k \rangle$ 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.
```

FIG. 3.1. *Details of the short paths algorithm.*

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 $r \geq 1$ 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 3.2, 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 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.

```
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.
```

FIG. 3.2. *Details of the recursive short paths algorithm.*

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 $O(r(n/k + L \log k))$.*

We will close this section with a few 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, actually solves bounded STCON, and is thus a general algorithm for *s-t* connectivity. In fact, given the appropriate parameters, the algorithm exhibits behavior and performance similar to the best-known previous algorithms for STCON. 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

```
                              {$S$ 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 $\bmod k = i_1\}$.
  $P = \emptyset$.                    {$P$ 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 $\bmod k = i_2\}$.
    $Q = S \cap S_{i_2}$.                {$Q$ 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 $Q\}$.
    $P = P \cup (A - B)$.
  end;
  if $|S| + |S' \cup P| > n/L^r$ then try next $j$.
  else $S' = S' \cup P$.
end;
```

FIG. 4.1. *Combining the two algorithms efficiently.*

(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 λ 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

$$O((n \log n)/L^r + r(n/k + L \log k)), \tag{4.1}$$

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 4.1 gives the code that should be used in place of the loop in Figure 2.1 (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 2.1 that finds the vertices on the first level.

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

$$O((k^2n/L^r)\cdot k^{rL}L^r \cdot n^2/k^2) = O(n^3k^{rL}). \tag{4.2}$$

We want to find the minimum amount of space the algorithm can use while still maintaining a polynomial running time. To maintain polynomial time, we must have, for some constant a,

$$k^{rL} = n^a. \tag{4.3}$$

For simplicity, we bound expression (4.1) from below by

$$\Omega(n/L^r + n/k). \tag{4.4}$$

(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 (4.4) 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 (4.3) 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 (4.4) gives a bound of $n/2^{\Theta(\sqrt{\log n})}$. Substituting these same values into the actual space bound expression (4.1) 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* $O(n^3k^{rL})$, *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.

Letting $L = 2$ and $k = 2^r$, we obtain the following corollary.

COROLLARY 5.1. *The combined algorithm of section* 4 *can solve* STCON *using time* $2^{O(\log^2(n/S))}\cdot n^3$ *given space* S.

The recent lower bound of Edmonds and Poon [6] shows that no algorithm that runs on an NNJAG can do better than time $2^{\Omega((\log^2(n/S))/\log\log n)}$ given space S. Ignoring the $\log\log n$ factor in the lower bound, the two bounds therefore match when $S = n/2^{O(\sqrt{\log n})}$. For higher space bounds, Edmonds and Poon's bound does not improve the JAG lower bound in Barnes and Edmonds [2] of $ST = \Omega(n^2/\log(n/S))$ (where T is the time used by the JAG).

Note that only the $\log\log n$ factor separates the lower bound from the upper bound when small space is used. When the space used is larger, there is still a significant gap between the upper and lower bounds (although the gap is not as large as Theorem 4.1 indicates, since the time bound in (4.2) is an overestimate). It is possible that the combined algorithm above is optimal for the NNJAG model, but to prove it, these gaps must be eliminated.

Given that the upper and lower bounds for the problem are now close for the JAG and NNJAG models, we should consider more general models of computation. To improve this algorithm, it seems likely that we must use methods for exploring a graph that cannot be mimicked by an NNJAG. On the other hand, finding a lower bound similar to Barnes and Edmonds's [2] or Edmonds and Poon's [6] on a more general model of computation would be a breakthrough and would help decide the question of the complexity of *DL* versus *NL*.

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

REFERENCES

[1] G. Barnes, J. F. Buss, W. L. Ruzzo, and B. Schieber, *A sublinear space, polynomial time algorithm for directed s-t connectivity*, in Proc. 7th Annual IEEE Conference on Structure in Complexity Theory, Boston, MA, 1992, pp. 27–33.

[2] G. Barnes and J. A. Edmonds, *Time-space lower bounds for directed s-t connectivity on JAG models*, in Proc. 34th Annual IEEE Symposium on Foundations of Computer Science, Palo Alto, CA, 1993, pp. 228–237.

[3] G. Barnes and W. L. Ruzzo, *Undirected s-t connectivity in polynomial time and sublinear space*, Comput. Complexity, 6 (1996–1997), pp. 1–28.

[4] S. A. Cook, *Deterministic CFL's are accepted simultaneously in polynomial time and log squared space*, in Conference Record of the 11th Annual ACM Symposium on Theory of Computing, Atlanta, GA, 1979, pp. 338–345. See also [14].

[5] S. A. Cook and C. W. Rackoff, *Space lower bounds for maze threadability on restricted machines*, SIAM J. Comput., 9 (1980), pp. 636–652.

[6] J. A. Edmonds and C. K. Poon, *A nearly optimal time-space lower bound for directed st-connectivity on the NNJAG model*, in Proc. 27th Annual ACM Symposium on Theory of Computing, Las Vegas, NV, 1995, pp. 147–156.

[7] H. R. Lewis and C. H. Papadimitriou, *Symmetric space-bounded computation*, Theoret. Comput. Sci., 19 (1982), pp. 161–187.

[8] N. Nisan, *$RL \subseteq SC$*, Comput. Complexity, 4 (1994), pp. 1–11.

[9] N. Nisan, E. Szemerédi, and A. Wigderson, *Undirected connectivity in $O(\log^{1.5} n)$ space*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 24–29.

[10] C. K. Poon, *Space bounds for graph connectivity problems on node-named JAGs and node-oriented JAGs*, in Proc. 34th Annual IEEE Symposium on Foundations of Computer Science, Palo Alto, CA, 1993, pp. 218–227.

[11] C. K. Poon, *On the Complexity of the st-Connectivity Problem*, Ph.D. Thesis, University of Toronto, 1996.

[12] W. J. Savitch, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[13] M. Tompa, *Two familiar transitive closure algorithms which admit no polynomial time, sublinear space implementations*, SIAM J. Comput., 11 (1982), pp. 130–137.

[14] B. von Braunmühl, S. A. Cook, K. Mehlhorn, and R. Verbeek, *The recognition of deterministic CFL's in small time and space*, Inform. and Control, 56 (1983), pp. 34–51.

[15] A. Wigderson, *The complexity of graph connectivity*, in Proc. 17th Symposium on Mathematical Foundations of Computer Science, I. M. Havel and V. Koubek, eds., Lecture Notes in Comput. Sci. 629, Springer-Verlag, New York, 1992, pp. 112–132.