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

Greg Barnes   Jonathan F. Buss
Walter L. Ruzzo   Baruch Schieber

Technical Report 92-03-05
March, 1992

To appear, *Proceedings of the 7^{th} Annual Structure in Complexity Theory Conference*, Boston, MA, June 1992. Also available as University of Waterloo Department of Computer Science Technical Report CS-92-18 and IBM Technical Report RC 17856.

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

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

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

October 16, 1992

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

*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 addresses: greg@cs.washington.edu, ruzzo@cs.washington.edu.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. Research supported in part by a grant from NSERC. Email address: jfbuss@maytag.waterloo.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 $\text{DSPACE}(f(n)) = \text{NSPACE}(f(n))$ 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: 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 path 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 path 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 path algorithm. Section 4 shows how the two algorithms can be combined to yield the desired result. Section 5 presents some notes and concluding remarks.

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

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, $j+k$, $j+2k$, $\ldots$, $j+k\cdot\lfloor n/k \rfloor$. Given a set S of vertices, we can find all vertices within distance k of S in time $n^{O(k)}$ and space $O(k \log n)$ by enumerating all possible paths of length at most k 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, $j+k$, $\ldots$, $j+ik$. Consider the set of vertices that are within distance k of a vertex in V_i. Clearly, this set contains all the vertices in level $j+(i+1)k$. However, it may also contain vertices in lower numbered levels. The vertices in level $j+(i+1)k$ are those vertices in the set that are not within distance $k-1$ of a vertex in V_i. Thus, to get V_{i+1} we add to V_i all vertices that are within distance k but not $k-1$ of V_i.

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

Referring to Figure 1, assuming that $k \leq \sqrt{n}$, the algorithm's space bound is dominated by the number of vertices in S and S', which is never more than $n/k+1$, so it uses space $O(\frac{n}{k} \log n)$. The time bound is dominated by repeatedly testing whether a vertex is within distance k of a vertex in S. Using a straightforward enumeration of all paths, this requires $n^{O(k)}$ time.

This algorithm is not sufficient for our purposes. In particular, if k is

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

Figure 1: Details of the breadth-first search algorithm

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 k from a vertex is at most d^k, so for these graphs, k 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 k does.

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

3 The Short Path Tradeoff

Consider the *short path problem*, which is to detect whether there is a path from s to t of length $f(n)$, for a given $f(n) = o(n)$. The short path problem is a special case of STCON that seems to encapsulate many of the difficulties of the general problem. It is particularly interesting given the breadth-first search algorithm above, because a more efficient method of finding short paths would clearly lead to an improvement in that algorithm's time bound.

Our second tradeoff is an algorithm that solves the short path problem in sublinear space and polynomial time for many $f(n)$. 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 path problem with $f(n) = \log^c n$, for 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 outedges from a 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 can be mapped to an $L + 1$ digit number in base k, where digit number 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-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.

The algorithm uses space $O(n/k)$ to store the vectors, and $O(L \log k)$ to write down the path to be tested. For all steps in each path, we do at most $O(n/k + (n/k)^2)$ work zeroing the vector and testing for edges from d_{i-1} to d_i, so the algorithm runs in $O(k^L L(n/k)^2) = O(k^L n^3)$ time to test all L steps on each of the k^L paths. Unfortunately, this does not quite achieve our goal of polynomial time and sublinear space when $L = \log^c n$. With a distance as small as $\log n$, the time is only polynomial if k is constant, which does not yield sublinear space. We *can* achieve polynomial time and sublinear space by reducing L. 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 $O(n^3 (\log n)^{c \log n/\log\log n}) = O(n^{c+3})$ 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 path 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 path algorithm

an algorithm that uses twice as much space, but finds paths of length L^2. In general, by recurring r times, the improved algorithm can find paths of length L^r using $O(r(n/k + L\log k))$ space and $O(n^{3r}k^{rL})$ time. A further refinement improves the time bound to $O(n^3 k^{rL})$ — one recursive call will find all vertices in d_i reachable from any reachable vertex in d_{i-1}.

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 $O(rn/\log^r n)$.

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 constant $c \geq 2$), with the recursive short path algorithm as a subroutine to find the paths between levels. With careful choices of the parameters k, L and r, however, the algorithm can use even less space while still maintaining polynomial time.

In general, the breadth-first search algorithm finds every (L^r)-th level of the tree, and the short path algorithm searches to distance L^r. The total space used by this algorithm is

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

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. The short paths subroutine is called $O(n^3)$ times, and thus the running time of the algorithm is $O(n^6 k^{rL})$.

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

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

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

(That is, we omit the $\log n$ factor in the first summand and the r factor in the second summand, and leave out the third summand altogether.) Given that $xLr = c_1 \log n$ for some constant c_1, expression (2) is minimized when L is constant. The expression can therefore be written as

$$O(n/2^{(c_2 \log n)/x} + n/2^x) \tag{3}$$

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

Plugging these same values ($r = \sqrt{\log n}$, $k = 2^{\Theta(\sqrt{\log n})}$, and constant L) into the actual space bound expression (1) yields the same asymptotic space bound of $n/2^{\Theta(\sqrt{\log n})}$. Since this matches the minimum for the simplified expression, we cannot do any better, and this must be the minimum space bound for the algorithm when using 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. Note that our so-called "short paths" algorithm is actually a general algorithm for s-t connectivity, with behavior and performance similar to the best-known previous algorithms. If we let $k = 1$, $L = n$, and $r = 1$, the algorithm is a somewhat inefficient variant of breadth-first search that uses $O(n)$ space and $O(n(n+m))$

time. Savitch's algorithm is just the special case of this algorithm where $k = n$, $L = 2$, and $r = \lceil \log n \rceil$. 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 both 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.

6 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.

[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, 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*, 19:161–187, 1982.

[5] N. Nisan. $RL \subseteq SC$. Preprint, Hebrew University, Mar. 1991. To appear in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing,* 1992.

[6] N. Nisan, E. Szemeredi, and A. Widgerson. Undirected connectivity in $O(\log^{1.5} n)$ space. Manuscript, Dec. 1991.

[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.