MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

Time-Space Lower Bounds for Directed *s-t* Connectivity on JAG Models

G. Barnes J.A. Edmonds

MPI–I–94–119 April 1994

Im Stadtwald
66123 Saarbrücken
Germany

Time-Space Lower Bounds for Directed s-t Connectivity on JAG Models

G. Barnes J.A. Edmonds

MPI-I-94-119 April 1994

Time-Space Lower Bounds for Directed s-t Connectivity on JAG Models

(Extended Abstract)

Greg Barnes* Jeff A. Edmonds†

April 28, 1994

Abstract

Directed s-t connectivity is the problem of detecting whether there is a path from a distinguished vertex s to a distinguished vertex t in a directed graph. We prove time-space lower bounds of $ST = \Omega(n^2/\log n)$ and $S^{1/2}T = \Omega(mn^{1/2})$ for Cook and Rackoff's JAG model [8], where n is the number of vertices and m the number of edges in the input graph, and S is the space and T the time used by the JAG. We also prove a time-space lower bound of $S^{1/3}T = \Omega(m^{2/3}n^{2/3})$ on the more powerful node-named JAG model of Poon [14]. These bounds approach the known upper bound of $T = O(m)$ when $S = \Theta(n \log n)$.

1 Introduction

The s-t connectivity problem is a fundamental one in computational complexity theory. The s-t connectivity problem for *directed* graphs (STCON) is the prototypical complete problem for nondeterministic logarithmic space [16]. Both STCON and the corresponding problem for undirected graphs, USTCON, are DLOG-hard — any problem solvable deterministically in logarithmic space can be reduced to either problem [13, 16]. Understanding the complexity of s-t connectivity is, therefore, a key to understanding the relationship between deterministic and nondeterministic space bounded complexity classes. For example, showing that there is no deterministic logarithmic space algorithm for directed connectivity would separate the classes DSPACE($\log n$) and NSPACE($\log n$), while devising such an algorithm would prove that DSPACE($f(n)$) = NSPACE($f(n)$) for any constructible $f(n) = \Omega(\log(n))$

*Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. E-mail address: barnes@mpi-sb.mpg.de.

†International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, California 94704-1198. E-mail address: edmonds@icsi.berkeley.edu.

[16]. Unfortunately, determining the complexity of STCON 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.

Proving lower bounds on the time or space requirements of STCON for a general model of computation, such as a Turing machine, is beyond the reach of current techniques. Thus, it is natural to consider a *structured* model [4] whose basic operations are based on the structure of the graph, as opposed to being based on the bits in the graph's encoding. A natural structured model for the problem of *s-t* connectivity is the "jumping automaton for graphs", or *JAG*, introduced by Cook and Rackoff [8]. A JAG moves a set of pebbles on the graph. There are two basic operations — moving a pebble along a directed edge in the graph, and jumping a pebble from its current location to the vertex occupied by another pebble. Although the JAG model is structured, it is not weak. In particular, it is general enough that most known deterministic algorithms for graph connectivity can be implemented on it. Poon [14] introduces the more powerful node-named JAG (*NNJAG*), an extension of the JAG model where the computation is allowed to depend on the names of the nodes on which the pebbles are located.

Cook and Rackoff [8] prove a lower bound of $\Omega\left(\log^2 n/\log\log n\right)$ on the space required for a JAG to compute directed *s-t* connectivity (STCON). Berman and Simon [3] extend this result to randomized JAGs, and Poon [14] extends it to a probabilistic version of the NNJAG. Tompa [18] shows lower bounds on the product of the time and space needed when using certain natural approaches to solve STCON. Many time-space lower bounds have been proved for *undirected s-t* connectivity on various weak versions of the JAG model [2, 7, 8]. Edmonds was the first to prove a time-space lower bound for USTCON on the unrestricted JAG model [9].

The standard algorithms for *s-t* 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 [16] provides a small space ($\Theta(\log^2 n)$) algorithm that requires time exponential in its space bound (i.e., time $n^{\Theta(\log n)}$). Barnes *et al.* [1] show the first sublinear space, polynomial time algorithm for STCON. All of these algorithms can be implemented on the standard JAG [8, 15]. Using the NNJAG's ability to access the names of the nodes in the graph, Poon [14] shows how to implement Immerman's and Szelepcsényi's nondeterministic $O(\log n)$-space algorithm for directed *s-t* nonconnectivity [11, 17] on a nondeterministic NNJAG. It is not clear that this algorithm can be implemented on a standard nondeterministic JAG.

Our main results are to prove lower bounds of $ST = \Omega(n^2/\log n)$ and $S^{1/2}T = \Omega(mn^{1/2})$ for STCON on the JAG model, and of $S^{1/3}T = \Omega(m^{2/3}n^{2/3})$ on the more powerful Node-Named JAG model, where S is the space and T the time used by the JAG, This last bound is proved on probabilistic NNJAGs by transforming the machine into a structured

branching program, and following the framework introduced by Borodin *et al.* [6]. These lower bounds approach the known upper bound of $T = O(m)$ when $S = \Theta(n \log n)$, and are the first time-space tradeoff on JAGs with an unrestricted number of jumping pebbles.

In the following section, we formally define the models used in this paper. In Section 3, we describe the family of *layered graphs*, the graphs we use to prove our lower bounds. In Section 4 we prove the $ST = \Omega(n^2/\log n)$ lower bound for the original JAG model, and in Section 5 we prove the $S^{1/3}T = \Omega(m^{2/3}n^{2/3})$ bound for the probabilistic node-named JAG. Finally, Section 6 presents some notes and a discussion of future work. The bound in Section 5 is proved on probabilistic machines with one-sided error. We can also show the same bound for the stronger class of probabilistic machines with two-sided error. To save space, the proofs of this result and of the bound of $S^{1/2}T = \Omega(mn^{1/2})$ on the JAG are omitted in this abstract.

For more information on graph connectivity, see the survey paper by Wigderson [19].

2 Definitions

A JAG [8] is a finite automaton with p distinguishable pebbles and q states. The input to a JAG is an instance of STCON, $\mathcal{G} = \langle G, s, t \rangle$, where G is a directed graph on n vertices with maximum outdegree d, and s and t are two distinguished nodes in the graph. (Note: For certain classes of graphs, we will sometimes define s and t to be specific nodes. We can think of a graph in one of these classes as a complete instance of STCON, since s and t are predefined for these graphs.) For each node in the input graph, the outgoing edges are given a unique label in $\{1, \ldots, d\}$. The JAG begins its computation in state Q_0, with one of the pebbles on the distinguished node t and the other $p - 1$ on s.

The program of the JAG may depend non-uniformly on n and on the degree d of the graph. What the JAG does each time step depends on the current state, the list of the pebbles that are on the distinguished vertices s and t, and the partition of the pebbles not on s and t, according to which pebbles are on the same vertices. Based on this information, the automaton changes state and either *walks* or *jumps* a pebble. Walking a pebble consists of selecting a pebble $P \in \{1, \ldots, p\}$ at some vertex v and some label $l \in \{1, \ldots, d\}$ and moving P along the edge out of v with label l. If there is no edge out of v with that label, the pebble stays at v. Jumping a pebble consists of selecting two pebbles $P, P' \in \{1, \ldots, p\}$ and moving P to the node occupied by P'. A JAG that solves STCON enters an accepting state if and only if there is a path from s to t in the input graph.

The space used by a JAG is defined to be $S = p \log_2 n + \log_2 q$, where p is the number of pebbles and q is the number of states. This corresponds to the $\log_2 n$ bits needed to store which of the n nodes a pebble is on and the $\log_2 q$ bits needed to record the current state.

An NNJAG is the same as a JAG except that the actions of the model are able to

depend on which node each of its pebbles is currently on. The nodes are distinguished by means of names in $\{1, \ldots, n\}$. The names are assigned arbitrarily to the nodes of the input graph and included as part of the input description.

The *(r-way) branching program* [5], used in the proof of Theorem 2, is the most general and unstructured model of computation and is at least as powerful as all reasonable models. Depending on the current state, one of the input variables is selected and its value is queried. Based on this value, the state changes. These state transitions are represented by a directed acyclic rooted graph of outdegree r, where r is the maximum number of different values possible for an input variable. Each node represents a possible state that the model might be in, and is labeled with the input variable queried in this state. The edges emanating out of a node are labeled with the possible values of the queried variable and the adjacent nodes indicate the results of the corresponding state transitions. Because any transition is allowed, no assumption is made about the way the model's workspace is managed, and any function of the known information can be computed in one time step. In order to indicate the outcome of the computation, each sink node is labeled with either *accept* or *reject*. A computation for an input consists of starting the input at the root of the branching program and having it follow a *computation path* through the program as explained above, until an *accept* or *reject* sink node is reached.

The space used by a branching program is the logarithm base 2 of the number of nodes in the program. This is equivalent to viewing the space as the number of bits of workspace, or as the number of bits required to specify the current state. A branching program is said to be *leveled* if the nodes can be assigned levels so that the root has level 0 and all edges go from one level to the next.

We add probabilism to the above models in the following way: For every bit string $R \in \{0,1\}^*$, there is an associated algorithm (a JAG or branching program). At the beginning of the computation a random $R \in \{0,1\}^*$ is chosen, and the corresponding algorithm is run on the input. The space of a probabilistic NNJAG or a branching program is defined to be the maximum space used over all associated algorithms. This way of representing probabilism effectively provides the model with $|R|$ bits of read only input containing R. These bits can be accessed in their entirety every time step, as long as the algorithm has enough space to store the time step, i.e. $S \geq \log T$. This method is therefore stronger than supplying the model with a few random bits at each time step.

3 Comb and Layered Graphs

We prove the time-space lower bound of Section 4 on a class of graphs known as *layered graphs*. A layered graph consists of $l = \frac{n-1}{\chi}$ layers, each containing χ vertices, plus the extra distinguished vertex t. The vertices in layer i are denoted $v_{\langle i,1\rangle}, v_{\langle i,2\rangle}, \ldots, v_{\langle i,\chi\rangle}$. The

distinguished vertex s will be set to be $v_{\langle 1,1\rangle}$. A layered graph has three types of edges. The vertices in the first layer are connected using a directed path of $\chi - 1$ *crossedges*, $(v_{\langle 1,1\rangle}, v_{\langle 1,2\rangle}), (v_{\langle 1,2\rangle}, v_{\langle 1,3\rangle}), \ldots, (v_{\langle 1,\chi-1\rangle}, v_{\langle 1,\chi\rangle})$. Every vertex in layers 1 through $l-1$ has two *downedges* connecting it to vertices on the next layer. Finally, there may or may not be an edge from a vertex on layer l to the distinguished vertex t. The edges are labeled in a straightforward way, say with 1 and 2 for the downedges of each node and 3 for the crossedges. See Figure 1 for an example of a layered graph.

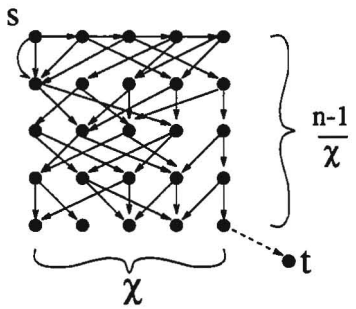

Figure 1: A layered graph

We prove the time-space lower bound of Section 5 on a different class of graphs, known as *comb graphs*. A comb graph, illustrated in Figure 2, is composed of a *back*, χ *teeth*, plus the distinguished node t. The back of the comb consists of a directed path of n nodes $v_1, \ldots, v_n$. The first node v_1 is the distinguished node s. The r^{th} tooth consists of the directed path $u_{\langle r,1\rangle}, \ldots, u_{\langle r,l\rangle}$. The length of each tooth will be $l = \frac{n}{\chi}$ so that the total number of nodes in a comb graph is $N = 2n + 1$.

There are m ($\geq n$) directed **connecting edges** $e_1, \ldots, e_m$ each going from a back node v_i to the top of one of the teeth in such a way that that the out-degree of any two back nodes can differ by at most 1. In particular, if $m = n$, the out-degree of the graph is two. More formally, for $j \in \{1, \ldots, m\}$, the connecting edge edge e_j is the $\lceil \frac{j}{n} \rceil^{th}$ edge emanating from back node $v_{\langle (j \bmod n)+1\rangle}$ and has label $\lceil \frac{j}{n} \rceil$. The variables $y_1, \ldots, y_m \in \{1, \ldots, \chi\}$ will be used to specify which tooth each of the connecting edges leads to. Specifically, $y_j = r$ means that the edge e_j leads to the top node of the r^{th} tooth. We will allow double edges, so it is of no concern if two edges from the same back node v_i go to the same tooth.

If there is to be a directed path from s to t then the node t is attached to the bottom of at least one of the teeth. The variables $\alpha_1, \ldots, \alpha_\chi \in \{0, 1\}$ will be used to specify for each of the teeth whether this is the case. Specifically, $\alpha_r = 1$ if the bottom node of the r^{th} tooth has a directed edge to node t. If $\alpha_r = 0$, then there is a self loop edge from the bottom node to itself in order to keep the degree fixed.

χ, the number of vertices in a layer (for a layered graph) or the number of teeth (for a

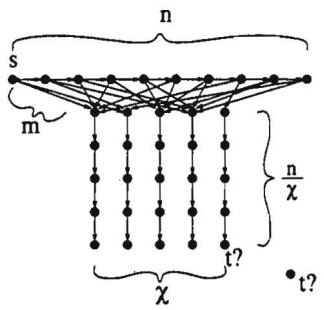

Figure 2: A comb graph

comb graph), is a parameter that will be chosen after the space allocated to the model is fixed. In Theorem 1 it is set to $n/\log(n/p)$, and in Theorem 2, it is set to $m^{\frac{1}{3}}n^{\frac{1}{3}}$. Intuitively, solving STCON for layered graphs is difficult because there are $\chi 2^l$ possible paths from s to vertices on layer l. The JAG must potentially check each such path before it can be sure whether or not t is not connected to s. Of course, these paths will overlap in many places, but because the model is allocated a bounded amount of space, it is difficult for it to "remember" which subpaths have been traversed already. Therefore, many subpaths must be traversed many times before the JAG is sure they have all been traversed. Similarly, it is difficult for a JAG to "remember" which teeth in a comb graph have been traversed, so some teeth may get traversed many times before the JAG is sure that they all have been traversed.

4 A Lower Bound for JAGs

Theorem 1 *Any JAG with p pebbles that solves* STCON *requires time (even with an arbitrarily large number of states) $\Omega\left(\frac{n^2}{p\log^2(n/p)}\right)$ on graphs with n vertices.*

Since the space of a JAG, S, is defined to be at least $p\log n$, the time-space tradeoff $ST = \Omega(\frac{n^2\log n}{\log^2(n/p)})$ follows. Note that the theorem sets no limit on the number of states in the JAG.

Proof of Theorem 1: We will show that to solve STCON on layered graphs of size n, a JAG requires time $\text{Min}(\chi 2^l, \frac{\frac{1}{2}(\chi-1)^2}{(p-1)})$. If we set the parameter χ to be $\frac{n}{\log(n/p)}$ then, since $l = \frac{n-1}{\chi}$, $T = \Omega\left(\frac{n^2}{p\log^2(n/p)}\right)$.

Suppose by way of contradiction that there is a JAG, J, that solves STCON on graphs of size n using p pebbles and less than the stated time. In order to bound how quickly the JAG can gain information, we will run J for this amount of time on graphs that have many

more than n vertices. This is not strictly allowed by the definition of JAGs, since a JAG is defined non-uniformly based on the size of the graph. However, because all the vertices in the input graph except s and t are indistinguishable during the computation of J, it is well-defined how the computation would proceed if J is given a larger graph. We cannot expect the JAG to solve STCON on this large graph, but we can run the JAG for T time steps and see what happens.

More formally, the next move taken by a JAG is specified by the transition function. The input to this function is the following information: the current state, the list of the pebbles that are on the distinguished vertices s and t, and the partition of the pebbles not on s and t, according to which pebbles are on the same vertices. Let us call this information the current *configuration* of a JAG. A computation on an input graph is formally defined to be a sequence of such configurations. Given this definition, we can say that J's computation on two different graphs is identical, even if the graphs have a different number of vertices.

We run J on a set of larger graphs referred to as *k-tree graphs*, one k-tree graph for each $k \in \{1, \ldots, l\}$. A k-tree graph consists of a layered graph with k layers and χ vertices per layer. Each vertex $v_{\langle k,i \rangle}$ on the kth layer is the root of a directed binary tree of depth $l-k+1$, with edges directed down from the root. As with the layered graph, the downedges are labeled 1 and 2 and the crossedges are labeled 3. In addition, each k-tree graph has an isolated vertex t. The distinguished vertex s is defined to be $v_{\langle 1,1 \rangle}$ for every k-tree graph.

We prove by induction that for every $k \in \{1, \ldots, l\}$, there exists a k-tree G_k and a leaf vertex v_* of this graph such that during the computation of the JAG J on G_k, there is never a pebble on the vertex v_*. At the end of the proof, we need to find a graph with n vertices on which the JAG J gives the incorrect answer. We use G_l, which is a layered graph with n vertices, to find such a graph.

For the base case of the induction, $k = 1$, there is only one graph G_1 in the class of 1-trees. This graph has $\chi 2^l$ leaf vertices. Because JAG J runs fewer than $\chi 2^l$ time steps, there must be some leaf vertex v_* that is never accessed in J's computation on G_1.

For the inductive step, assume there is a $(k-1)$-tree, G_{k-1}, and a leaf v_* in G_{k-1} such that the computation of the JAG J on G_{k-1} never places a pebble on v_*. Think of G_{k-1} as follows. It has $k-1$ layers of a layered graph. Layer k has 2χ vertices, the vertices in the second level of the binary trees rooted at layer $k-1$. Each of these vertices is the root of a directed binary tree of depth $l-k+1$. Denote these 2χ disjoint binary trees by $\mathcal{T}_1, \ldots, \mathcal{T}_{2\chi}$. Denote the downedges going from layer $k-1$ to the roots of these trees by $e_1, \ldots e_{2\chi}$.

The goal of the inductive step is to produce a k-tree, G_k. Think of G_k as follows. Like G_{k-1}, it has $k-1$ layers of a layered graph. We will choose G_k so that G_{k-1} and G_k are identical on the first $k-1$ layers. Like G_{k-1}, G_k will have 2χ downedges $e_1, \ldots e_{2\chi}$ going from layer $k-1$ to layer k. Layer k of the k-tree, however, has only χ vertices, which are the roots of χ directed binary tree of depth $l-k+1$. Denote these χ binary trees by

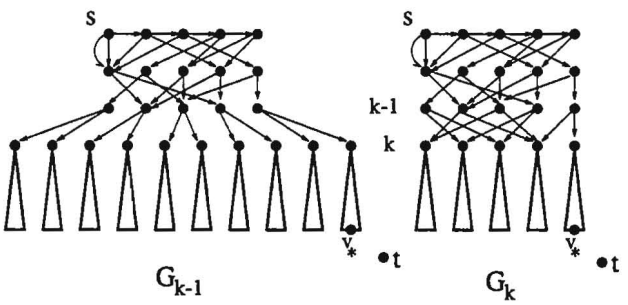

Figure 3: Collapsing the trees in G_{k-1} to form G_k

$\mathcal{T}'_1, \ldots, \mathcal{T}'_\chi$. What remains to be chosen in order to specify G_k are the connections between the downedges $e_1, \ldots e_{2\chi}$ and the trees $\mathcal{T}'_1, \ldots, \mathcal{T}'_\chi$. These connections can be specified by choosing a partition of the trees $\mathcal{T}_1, \ldots, \mathcal{T}_{2\chi}$ into χ groups, $\mathcal{S}_1, \ldots, \mathcal{S}_\chi \subseteq \{\mathcal{T}_1, \ldots, \mathcal{T}_{2\chi}\}$. For $i \in \{1, \ldots, 2\chi\}$ and $h \in \{1, \ldots, \chi\}$, if $\mathcal{T}_i \in \mathcal{S}_h$, then the downedge e_i is connected to the root of the tree $\mathcal{T}'_h$ in the graph G_k. This can be thought of as collapsing the trees in the group $\mathcal{S}_h$ into the one tree $\mathcal{T}'_h$. See Figure 3.

We want to find a partition $\mathcal{S}_1, \ldots, \mathcal{S}_\chi$ with the property that the computation on the corresponding graph G_k is identical to that on G_{k-1}. Below, we show how to find a partition with the following property: for any two trees $\mathcal{T}_i$ and $\mathcal{T}_j$, if there is ever a time in the computation of J on G_{k-1} when one pebble is in $\mathcal{T}_i$ and another pebble is in $\mathcal{T}_j$, then these two trees will be in different groups in the partition. If this property is preserved, we can show that the sequence of configurations in the computation of J on G_{k-1} is the same as the sequence in the computation of J on a graph G_k. One difference between G_{k-1} and G_k is that in G_{k-1} the vertices in layer k all have in-degree one. But the JAG model is defined so that it has no access to the in-degree of a vertex. It is not hard to see that if the two computations were to deviate, the first deviation would occur because two pebbles collide in G_k that do not collide in G_{k-1}. To be more precise, in the computation on G_k, one pebble enters a tree $\mathcal{T}'_h$ via the downedge e_i and another pebble enters the same tree via the downedge e_j. Within this tree the two pebbles meet. In the computation on G_{k-1}, which is the same up to this point, the one pebble enters the tree $\mathcal{T}_i$ via the downedge e_i and the other pebble enters the tree $\mathcal{T}_j$ via the downedge e_j. Clearly, these pebbles do not meet. Hence, the partition of the pebbles according to which are on the same vertices becomes different for the two computations. However, such an event is not possible. It would mean that at some point during the computation on G_{k-1}, there is a pebble in the tree $\mathcal{T}_i$ and at the same time there is a pebble in the tree $\mathcal{T}_j$. By the property of the partition, these trees would be in different groups, e_i and e_j would be connected to different trees in G_k, and the pebbles entering these trees would not meet. It follows that the two sequences of

configurations are the same.

The next step is to explain how a partition $\mathcal{S}_1, \ldots, \mathcal{S}_\chi \subseteq \{\mathcal{T}_1, \ldots, \mathcal{T}_{2\chi}\}$ with this property is found. Run the JAG J on the graph G_{k-1}, while maintaining an undirected graph H with vertex set $\{\mathcal{T}_1, \ldots, \mathcal{T}_{2\chi}\}$. The undirected edge $\{\mathcal{T}_i, \mathcal{T}_j\}$ is added to H if there is ever a time during the computation when one pebble is in the tree $\mathcal{T}_i$ and another pebbles is in the tree $\mathcal{T}_j$.

We claim that H will contain at most $E = \frac{(\chi-1)^2}{2}$ edges. During one time step, only one pebble is allowed to move. This move can add new edges to H only if the pebble moves into some tree $\mathcal{T}_i$. There are only $p-1$ other pebbles, so there are at most $p-1$ trees $\mathcal{T}_j$ already containing pebbles. Therefore, at most $p-1$ edges can be added to H at this step, one edge for each possible pair $\{\mathcal{T}_i, \mathcal{T}_j\}$. The computation lasts for fewer than $\frac{\frac{1}{2}(\chi-1)^2}{(p-1)}$ time steps, so at most $\frac{(\chi-1)^2}{2}$ edges are added. The following lemma shows that the chromatic number of H is then at most $\chi - 1$.

Lemma 1 *Every undirected graph with no more than E edges has chromatic number at most $\sqrt{2E}$.*

Proof of Lemma 1: Fix a graph. At most $\sqrt{2E}$ vertices have degree at least $\sqrt{2E}$. Give each of them its own colour. The remaining vertices can be coloured with the same $\sqrt{2E}$ colours—each vertex in turn is given a colour that has not been assigned to one of its fewer than $\sqrt{2E}$ neighbors. ■

Because H has chromatic number no more than $\chi - 1$, the vertices $\{\mathcal{T}_1, \ldots, \mathcal{T}_{2\chi}\}$ can be partitioned into $\chi - 1$ groups $\mathcal{S}_1, \ldots, \mathcal{S}_{\chi-1}$ such that no edge of H has both ends in the same group. It follows that this partition has the required property.

To complete the induction step, we must find a leaf vertex of the k-tree that is never visited during the computation by J on the graph. Let $\mathcal{T}_*$ be the tree of G_{k-1} containing the leaf vertex v_*. Delete $\mathcal{T}_*$ from the group $\mathcal{S}_h$ that contains $\mathcal{T}_*$ and form a new group $\mathcal{S}_\chi$ containing only $\mathcal{T}_*$. This new partition also has the required property. Consider the leaf vertex of $\mathcal{T}'_\chi$ corresponding to the leaf vertex v_* of $\mathcal{T}_*$. The k-tree G_k, defined by this new partition, has the property that the only way to get from s to this leaf vertex is to traverse to the downedge e_* and then to follow the path through the tree to the leaf. In fact, inductively we can prove that this path is unique and is defined by the same sequence of labeled edges in both G_{k-1} and G_k. Suppose that, in the graph G_{k-1}, there is a unique path from s to the leaf v_*. It would follow that there is the same unique path from s to the leaf vertex in question. Hence, it is reasonable to denote both the leaf vertex of G_{k-1} and this leaf vertex of G_k by v_*.

By the induction hypothesis, the computation of J on G_{k-1} never reaches the vertex v_*. By the stated property of the partition, the computation on G_k is identical to that on G_{k-1}. The same sequence of labels that must be traversed to reach v_* in G_k must be traversed to

reach v_* in G_{k-1}. It follows that the computation on G_k never reaches the vertex v_*. This completes the inductive step.

After collapsing the k-tree graphs at each layer, we obtain a layered graph G_l with n vertices. We also find a leaf v_* of this graph that never contains a pebble during the computation of J. Let G'_l be the same graph as G_l except that there is a directed edge from the leaf v_* to the distinguished vertex t. Because J never places a pebble on vertex v_*, it can never detect whether or not there is an outgoing edge from v_* to t. Therefore, J's computation is the same on both G_l and G'_l, and hence J gives an incorrect answer for one of the graphs. Note that pebbles located on vertex t do not give the JAG any information about incoming edges. In fact, because t has no outgoing edges, pebbles on t can only move by jumping. ■

5 Node Named JAGs

In this section, we strengthen the previous result so that it applies to the stronger NNJAG model. The tradeoff is proved by translating any NNJAG algorithm into an (r-way) branching program. For a description of the branching program model, see Section 2. The lower bound follows the framework introduced by Borodin *et al.* [6] and used by Borodin and Cook [5]. If the computation time is short, then for each input there must be a short sub-branching program in which a lot of the "progress" required for the input is made. However, no sub-branching program is able to accomplish this for many inputs. Therefore, in order to handle all of the inputs, the branching program must be composed of many sub-branching programs. This means that the branching program has many nodes and hence uses a lot of space.

A probabilistic algorithm is said to allow *one-sided error* for a language L if for every input not in L, the correct answer is given, but for inputs in L, the incorrect answer may be given with some probability bounded by a constant. We consider algorithms with one-sided error for s-t nonconnectivity. The algorithms must answer correctly when there is a directed path from s to t. The result is strengthened by considering a random input chosen from a natural input distribution $\mathcal{D}$ on instances of the STCON problem that are not s-t connected.

Theorem 2 *There exists a probability distribution $\mathcal{D}$ on directed graphs of out-degree two that are not s-t connected such that for every probabilistic NNJAG solving directed s-t nonconnectivity with one-sided error*

$$\Pr_{\substack{G\in\mathcal{D},\\ R\in\{0,1\}^*}}\left[T_{\langle G,R\rangle} \le \frac{0.09m^{\frac{2}{3}}n^{\frac{2}{3}}}{S^{\frac{1}{3}}} \text{ and } \langle G,R\rangle \in C\right] \le 2^{-S}$$

where $T_{\langle G,R\rangle}$ is the computation time for input G and random bit string $R \in \{0,1\}^*$, and C is the set of $\langle G,R\rangle$ for which the correct answer to s-t connectivity is given.

Using an argument even simpler than Yao's, it is sufficient to prove the theorem for a fixed random string $R \in \{0,1\}^*$. Therefore, from here on, the probabilistic aspect of the algorithm is dropped. The only randomness will come from the choice of the input $G \in \mathcal{D}$.

The input domain consists of the same comb graphs as used in Theorem 1. The only difference is that the model requires that the input includes a "name" for each of the nodes. These names could be assigned arbitrarily. However, we will simply use the names v_i and $u_{\langle r,j \rangle}$ that were used to describe the graph. The effect is that the model always knows which node within the comb graph structure each pebble is on. Considering this fixed naming only strengths the lower bound result.

The probability distribution $\mathcal{D}$ is defined by constructing comb graphs as follows. Set $\alpha_r = 0$ for all $r \in \{1, \ldots, \chi\}$. Thus all inputs $G \in \mathcal{D}$ are not in STCON. What remains is to set the random variables $y_1, \ldots, y_m$ specifying which teeth the connecting edges $e_1, \ldots, e_m$ are attached to. Randomly partition the teeth into two equal size subsets $easyteeth_G$ and $hardteeth_G \subseteq \{1, \ldots, \chi\}$. Randomly choose $\frac{\chi}{2}$ of the connecting edges and put the associated variables y_j in the set $hardedges_G$. Randomly attach each of these "hard" connecting edges to one of the "hard" teeth in a one-to-one way. The set $easyedges_G$ is defined to contain the remaining y_j variables. Independently assign each $y_j \in easyedges_G$ a tooth from $easyteeth_G$ chosen uniformly at random.

5.1 The Definition of Progress

The lower bound measures, for each input G and each step in the computation, how much progress has been made towards solving s-t connectivity. We will say that the amount of **progress** made is the number of hard teeth, i.e. $r \in hardteeth_G$, that have had a pebble on their bottom node $u_{\langle r,l \rangle}$ at some point so far during the computation. It turns out that if the correct answer has been obtained for $G \notin$ STCON, then lots of progress must have been made.

Lemma 2 *For every comb graph $G \notin$ STCON, if $G \in \mathcal{C}$, then the computation for G must make $\frac{\chi}{2}$ progress.*

Proof of Lemma 2: Suppose that $G \notin$ STCON and there is a hard tooth $r \in hardteeth_G$ such that during the computation there is never a pebble on bottom node of this tooth. Let G' be obtained from G by connecting the bottom node of the r^{th} tooth to t, i.e. set $\alpha_r = 1$. The NNJAG model is defined such that it can only learn whether there is a directed edge from $u_{\langle r,l \rangle}$ to t by having a pebble on node $u_{\langle r,l \rangle}$. Therefore, the computation on G and G' is the same. Since $G' \in$ STCON, the answer given must be that the graph is connected. Since $G \notin$ STCON, this implies that $G \notin \mathcal{C}$. ■

The next lemma uses the fact that NNJAG is not a random access machine to prove that l time steps are required to make progress for one tooth.

Lemma 3 *If at some step, the r^{th} tooth does not contain a pebble, then a pebble must enter the tooth via one of the connecting edges and each edge in the tooth must be traversed by some pebble, before a pebble arrives at the bottom of this tooth.*

Proof of Lemma 3: The NNJAG model does not allow a pebble to arrive at a node unless there is another pebble to jump to or it walks there. ■

Moving a pebble to the bottom of a tooth in itself requires too little time for progress to be sufficiently costly for a superlinear lower bound. Additional cost occurs because many easy teeth must be traversed before a hard tooth is found. The distribution $\mathcal{D}$ on comb graphs is defined so that the easy teeth are accessed by most of the connecting edges, hence are easy to find. This is why arriving at the bottom of these teeth is not considered to be progress. On the other hand, the hard teeth are each attached to only one connecting edge and hence are hard to find.

5.2 Converting an NNJAG into a Branching Program

The proof technique is to convert a NNJAG algorithm into a branching program. In general, proving lower bounds on branching programs is very difficult. However, the branching program that we will obtain will have "structure" imposed on it by the structure of the NNJAG model. Lemmas 2 and 3 characterize the required structure.

Consider any fixed NNJAG algorithm. The leveled branching program $\mathcal{P}$ is formed as follows. There is a node $\langle Q, \Pi, T\rangle$ in $\mathcal{P}$ for every configuration $\langle Q, \Pi\rangle$ of the NNJAG algorithm and time step $T \in \{1, \ldots, T_{max}\}$, where T_{max} is the bound given in the theorem. An NNJAG configuration $\langle Q, \Pi\rangle$ is specified by the current state $Q \in \{1, \ldots, q\}$ and the position of the p pebbles $\Pi \in \{1, \ldots, N\}^p$. *Start, accept,* and *reject* states of the NNJAG translate to *start, accept,* and *reject* configuration nodes of the branching program, respectively. There is a directed edge from configuration node $\langle Q, \Pi, T\rangle$ to $\langle Q', \Pi', T+1\rangle$ in $\mathcal{P}$, if there exists a comb graph for which our fixed NNJAG algorithm would cause this transition.

The time step T is included in the configuration $\langle Q, \Pi, T\rangle$ so that $\mathcal{P}$ is acyclic and leveled. Although the NNJAG computation may run arbitrarily long for some G, we will only be concerned about the first $T_{max} \leq n^2$ steps. The number of nodes in $\mathcal{P}$ is $q \times n^p \times n^2 \in 2^{(1+o(1))S}$, where $S = \log_2 q + p \log_2 n$ is the **space** of the NNJAG. Hence, $(1 + o(1))\, S$ is the space used by $\mathcal{P}$.

5.3 Proving Lower Bounds on a Branching Program

The first step of the Borodin *et al.* [6] framework is to break the leveled branching program $\mathcal{P}$ into a collection of shallow sub-branching programs. This is done by breaking $\mathcal{P}$ into layers of $h = \frac{x}{4}$ levels each and considering the sub-branching programs rooted at each node on the inter-layer boundaries. We now prove that for the inputs that make lots of progress

in a small amount of time, there must be a sub-branching program $\widehat{\mathcal{P}}$ that makes quite a bit of progress for this input.

Lemma 4 *If $G \notin$ STCON, $T_G \leq T_{max}$, and $G \in \mathcal{C}$, then at least one of these sub-branching programs makes at least $\frac{\frac{\chi}{2}}{\frac{T_{max}}{h}}$ progress on input G.*

Proof of Lemma 4: Consider such an input G. By Lemma 2, the computation on G must make at least $\frac{\chi}{2}$ progress. Because the branching program $\mathcal{P}$ is leveled, the computation on G passes the root of only one sub-branching program $\widehat{\mathcal{P}}$ at each of the $\frac{T_{max}}{h}$ inter layer boundaries. Therefore, one of these sub-branching programs must make $\frac{\chi}{2}/\frac{T_{max}}{h}$ of the required $\frac{\chi}{2}$ progress. ■

The next step is to prove that a shallow sub-branching program cannot make this much progress from many input. Consider one of the sub-branching programs $\widehat{\mathcal{P}} \in \mathcal{P}$. We will determine how much progress it makes for every input $G \in \mathcal{D}$ (even if the computation on input G never reaches the root of $\widehat{\mathcal{P}}$).

Recall that each node $\langle Q, \Pi, T \rangle$ of the branching program specifies the location of every pebble in the comb graph. Define $\mathcal{F} \subseteq \{1, \ldots, \chi\}$ to be the set of teeth that contain pebbles at the root of $\widehat{\mathcal{P}}$. Because there are only p pebbles, $|\mathcal{F}| \leq p$. For each input $G \in \mathcal{D}$, define $\mathcal{C}_G \subseteq \{1, \ldots, \chi\}$ to be the set of teeth that do not contain pebbles at the root of $\widehat{\mathcal{P}}$, yet whose bottoms contain a pebble at some point during the computation by $\widehat{\mathcal{P}}$ on G. By Lemma 3, each edge of each tooth in $\mathcal{C}_G$ must be traversed by some pebble. The teeth have length l and the computation by $\widehat{\mathcal{P}}$ performs only h steps; therefore $|\mathcal{C}_G| \leq \frac{h}{l}$. Let c denote this bound.

The lower bound considers that progress has been made only when a pebble arrives at the bottom of hard teeth. Hence,

$$\begin{aligned} & |(\mathcal{F} \cup \mathcal{C}_G) \cap \mathit{hardteeth}_G| \\ & \quad \leq \quad |\mathcal{C}_G \cap \mathit{hardteeth}_G| + |\mathcal{F}| \\ & \quad \leq \quad |\mathcal{C}_G \cap \mathit{hardteeth}_G| + p \end{aligned}$$

is an upper bound on the amount of progress made by the sub-branching program $\widehat{\mathcal{P}}$ on input G. (The lower bound credits the algorithm with p progress, even if the teeth that initially contain pebbles are never traversed to the bottom and even if they are not hard. Because $p << \chi$, this is not a problem.) What remains is to prove that $\widehat{\mathcal{P}}$ makes lots of non-free progress, $|\mathcal{C}_G \cap \mathit{hardteeth}_G|$ is large, for very few comb graph inputs.

Lemma 5 *If $h \leq \frac{|\mathit{easyteeth}_G|}{2}$, then $\Pr_{G \in \mathcal{D}}\left[\, |\mathcal{C}_G \cap \mathit{hardteeth}_G| \geq 2\rho c \,\right] \leq 2^{-0.38\rho c}$, where $\rho = \frac{\chi}{m}$.*

The proof is left for the full paper. The idea is as follows. Within the distribution $\mathcal{D}$, the probability of a particular tooth $r \in \{1, \ldots, \chi\}$ being hard is $\frac{1}{2}$. However, the NNJAG is

not able to move a pebble to a particular tooth. Instead, it must select a connecting edge e_i and move a pebble into what ever tooth it is attached to. The model can identify the tooth found by the name of its top node. However, the bounded space model cannot have stored very much information about whether or not this tooth is hard. Therefore, the algorithm has little information on which to base the decision as to whether to move the pebble to the bottom of the tooth (i.e. $r \in C_G$) or not. It turns out that the tooth is hard iff the connecting edge e_i is hard and the probability of this is only $\frac{\chi/2}{m} \approx \rho$, because only $\frac{\chi}{2}$ of the m different connecting edges are chosen to be connected to hard teeth. Using a more formal argument, each of the at most c teeth in C_G is shown to be hard with probability at most ρ. Hence, we can expect ρc of the teeth in C_G to be hard. Chernoff's bounds prove that $|C_G \cap hardteeth_G|$ will not deviate far from this expectation.

To be a little more formal, note that the set of teeth C_G depends on the input G only as far as which computation path γ it follows. Therefore, it is well defined to instead refer to the set C_γ. Because every input follows one and only one computation path γ through $\widehat{\mathcal{P}}$, it is sufficient to prove that for every path γ, a lot of progress is made for very few of the inputs that follow the computation path γ. Specifically, for each path γ through $\widehat{\mathcal{P}}$, we prove that $\Pr_{G \in \mathcal{D}}\left[|C_\gamma \cap hardteeth_G| \geq 2\rho c \;\middle|\; G \text{ follows } \gamma \right] \leq 2^{-0.38\rho c}$.

Each tooth $r \in C_\gamma$ can be thought of as a trial. The r^{th} trial consists of choosing a random input G subject to the condition that G follows the computation path γ. The trial succeeds if the r^{th} tooth is hard. These trials may not be independent. Hence, the Chernoff bounds cannot be applied directly. Instead, for each trial $r \in C_\gamma$ and each outcome $\mathcal{O} \in \{succeeds, fails\}^{C_\gamma - \{r\}}$ of the other trials, we prove that $\Pr_{G \in \mathcal{D}}\left[r \in hardteeth_G \;\middle|\; G \text{ follows } \gamma \text{ and satisfies } \mathcal{O} \right] \leq \frac{\chi}{m} = \rho$. Given this, a version of the Chernoff bounds by Hisao Tamaki [personal communication] can be applied.

Because $r \in C_\gamma$, we know that at the beginning of the sub-branching program $\widehat{\mathcal{P}}$, the r^{th} tooth does not contain a pebble and at some point in the computation a pebble arrives at the bottom of this tooth. Therefore, by Lemma 3, we know a pebble must enter the r^{th} tooth via one of the connecting edges during the computation path γ. Without loss of generality, let the connecting edge in question be e_j. Note that when a pebble walks the connecting edge e_j into the top of the r^{th} tooth, the branching program learns that $y_j = r$. Hence, the condition that "G follows γ" includes the condition that $y_j = r$.

How does this condition by itself affect the probability that r is in $hardteeth_G$? From the definition of $hardedges_G$, we know that if $y_j = r$, then $y_j \in hardedges_G$ if and only if $r \in hardteeth_G$. This gives us that $\Pr_{G \in \mathcal{D}}\left[r \in hardteeth_G \;\middle|\; y_j = r \right] = \Pr_{G \in \mathcal{D}}\left[y_j \in hardedges_G \;\middle|\; y_j = r \right]$. This is equal to $\Pr_{G \in \mathcal{D}}\left[y_j \in hardedges_G \right]$, because we know that y_j has some value and there is a symmetry amongst all the possible values. Hence, telling you that $y_j = r$ gives you no information about whether $y_j \in hardedges_G$. Finally,

$\Pr_{G\in\mathcal{D}}\left[y_j \in hardedges_G\right] = \frac{\chi/2}{m}$, because $\frac{\chi}{2}$ of the variables $y_1,\ldots,y_m$ are randomly chosen to be in $hardedges_G$. How other conditions γ and $\mathcal{O}$ effect the probability will be left for the full version.

The next step after Lemma 5 is to prove is that if each sub-branching program $\widehat{\mathcal{P}}$ makes sufficient progress for very few inputs, then not too many inputs have a sub-branching program in which sufficient progress is made.

Lemma 6 $\Pr_{G\in\mathcal{D}}\left[\exists\widehat{\mathcal{P}}\ that\ makes \geq 2\rho c + p\ progress\ for\ G\right] \leq 2^{(1+o(1))S} \times 2^{-0.38\rho c}$.

Proof of Lemma 6: From Lemma 5, for any sub-branching program $\widehat{\mathcal{P}}$, $\Pr_{G\in\mathcal{D}}\left[\widehat{\mathcal{P}}\right.$ that makes $\geq 2\rho c + p$ progress for $G\left.\right] \leq 2^{-0.38\rho c}$. The number of nodes in the entire branching program $\mathcal{P}$ and hence the number of sub-branching programs $\widehat{\mathcal{P}}$ is no more than $q \times n^p \times T_{max} \in 2^{(1+o(1))S}$. Thus, the number of inputs that make the stated progress within some sub-branching program $\widehat{\mathcal{P}}$, is no more than $2^{(1+o(1))S}$ times the number makes it with one fixed sub-branching program. The lemma follows. ■

The final step combines Lemma 4 and Lemma 6.

Proof of Theorem 2: Recall, $l = \frac{n}{\chi}$, $h = \frac{\chi}{4}$, $c = \frac{h}{l}$, and $\rho = \frac{\chi}{m}$. Set the number of teeth χ to $2.77m^{\frac{1}{3}}n^{\frac{1}{3}}S^{\frac{1}{3}}$ in order to insure that $0.38\rho c = 0.38\frac{\chi^3}{4mn} = 2.01\ S$. Finally, set the time bound T_{max} to $0.09\frac{m^{\frac{2}{3}}n^{\frac{2}{3}}}{S^{\frac{1}{3}}}$ or equivalently to $\frac{mn}{4.02\chi} = \frac{ml}{4.02}$ (which is the time for the brute force algorithm). By Lemma 4, $\Pr_{G\in\mathcal{D}}\left[T_G \leq T_{max}\ \text{and}\ G \in \mathcal{C}\right] \leq \Pr_{G\in\mathcal{D}}\left[\exists\widehat{\mathcal{P}}\ \text{that makes} \geq \frac{\frac{\chi}{2}}{\frac{T_{max}}{h}}\ \text{progress for}\ G\right]$. Because $\frac{\chi/2}{T_{max}/h} = 2.01\left(\frac{\chi}{m}\right)\left(\frac{h}{l}\right) = 2.01\rho c \geq 2\rho c + \frac{S}{\log n} \geq 2\rho c + p$, it follows that this probability is no more than $\Pr_{G\in\mathcal{D}}\left[\exists\widehat{\mathcal{P}}\ \text{that makes} \geq 2\rho c + p\ \text{progress for}\ G\right] \leq$ (by Lemma 6) $2^{(1+o(1))S} \times 2^{-0.38\rho c} \leq$ (by the def^{n} of χ) $2^{(1+o(1))S} \times 2^{-2.01S} \leq 2^{-S}$. ■

6 Open Problems

The obvious open problems presented by this work are to improve the STCON lower bounds for the JAG and NNJAG. Currently, there are two lower bounds for the JAG, the $ST = \Omega(n^2/\log n)$ bound presented in Section 4, and the $S^{1/2}T = \Omega(mn^{1/2})$ bound, which is proved using a variant of the partitioning process in Theorem 1. The former is stronger if $mS^{1/2} = O(n^{3/2}/\log n)$, the latter is stronger otherwise. One obvious approach would be to combine the two techniques, but we have so far been unsuccessful at combining the different partitioning process used to prove the second tradeoff with the inductive shrinking of the