Simulating Undirected *st*-Connectivity Algorithms on Uniform JAGs and NNJAGs*

Pinyan Lu[1], Jialin Zhang[2], Chung Keung Poon[3], and Jin-Yi Cai[4]

[1] State Key Laboratory of Intelligent Technology and Systems, Department of Computer Science and Technology, Tsinghua University, Beijing, China
lpy@mails.tsinghua.edu.cn
[2] Department of Physics, Tsinghua University, Beijing, China
zhanggl02@mails.tsinghua.edu.cn
[3] Department of Computer Science, City University of Hong Kong, Hong Kong, China
ckpoon@cs.cityu.edu.hk
[4] Computer Sciences Department, University of Wisconsin, Madison, WI 53706, USA; and Tsinghua University, Beijing, China
jyc@cs.wisc.edu

Abstract. In a breakthrough result, Reingold [17] showed that the Undirected *st*-Connectivity problem can be solved in $O(\log n)$ space. The next major challenge in this direction is whether one can extend it to directed graphs, and thereby lowering the deterministic space complexity of $\mathcal{RL}$ or $\mathcal{NL}$. In this paper, we show that Reingold's algorithm, the $O(\log^{4/3} n)$-space algorithm by Armoni et al. [3] and the $O(\log^{3/2} n)$-space algorithm by Nisan et al. [14] can all be carried out on the RAM-NNJAG model [15] (a uniform version of the NNJAG model [16]). As there is a tight $\Omega(\log^2 n)$ space lower bound for the Directed *st*-Connectivity problem on the RAM-NNJAG model implied by [8], our result gives an obstruction to generalizing Reingold's algorithm to the directed case.

1 Introduction

The *st*-Connectivity problem is one of the most widely studied problems in computer science. It is a fundamental problem with many applications and yet is simple to state: Given a directed graph G together with two vertices s and t, the (directed) *st*-connectivity problem STCON is to determine if there is a directed path from s to t. In the special case when the graph G is undirected, we denote the problem by USTCON.

STCON is important in complexity theory as it is complete for the complexity class Non-deterministic Logspace $\mathcal{NL}$ under Deterministic Logspace reductions. Thus determining its (deterministic) space complexity is to answer the question

* The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [CityU 1071/02E], an NSF Grant, USA (CCR-0208013) and the Natural Science Foundation of China (60223004, 60321002).

X. Deng and D. Du (Eds.): ISAAC 2005, LNCS 3827, pp. 767–776, 2005.

whether nondeterminism helps in space bounded computation—a space analog of the "$\mathcal{NP} = \mathcal{P}$?" question. The special case USTCON is also important for the dual reasons of being the core problem in many applications as well as being complete for the complexity class Symmetric Logspace $\mathcal{SL}$ [11].

To solve STCON, a natural approach is to perform a Depth-First Search or Breadth-First Search from node s trying to discover node t. This requires $\Omega(n)$ bits of storage in the worst case. Currently, the best known space upper bound is $O(\log^2 n)$ using Savitch's algorithm [19]. Proving any non-trivial $(\omega(\log n))$ space lower bound on a general Turing machine is beyond the reach of current techniques, let alone proving that Savitch's algorithm is optimal. Thus Cook and Rackoff [6] proposed a model called "Jumping Automata for Graphs" (JAG), in order to abstract away certain inessential features of existing st-connectivity algorithms, and to focus on its essential feature of moving from vertices to vertices. Briefly, the JAG model can only examine the input graph by a set of pebbles that traverse the graph from s. Moreover, it can only tell which pebbles are on the same nodes but cannot see the node names. Other than that, a JAG is unrestricted. Although Savitch's algorithm needs to cycle through all nodes in the graph, which is generally impossible for a JAG, Cook and Rackoff adapted the algorithm to run on a JAG in $O(\log^2 n)$ space. Moreover, they proved a space lower bound of $\Omega(\log^2 n/\log\log n)$ for directed STCON on this model. The lower bound was then extended to a randomized JAG (i.e., a JAG that can flip random coins to determine its moves) by Berman and Simon [5].

Later, Immerman [10] and Szelepcsényi [20] discovered a surprising nondeterministic logspace algorithm for st-nonconnectivity which seems to require node names in an essential way and is not known to be implementable on a nondeterministic JAG (i.e., a JAG that can make nondeterministic moves). Then Poon [16] proposed the NNJAG model which is more natural as it can see the names of the pebbled nodes. Further, he showed how to implement the Immerman-Szelepcsényi algorithm on a non-deterministic NNJAG while extending the lower bounds of Cook and Rackoff, and Berman and Simon to the deterministic and randomized NNJAG model. The lower bound is further improved to $\Omega(\log^2 n)$ for a randomized NNJAG by Edmonds et al. [8]. Other major newer algorithms for STCON, including the time-space tradeoff [4] by Barnes et. al., and the randomized STCON algorithm [9] by Gopalan et. al., can all be implemented on a deterministic or randomized NNJAG as appropriate. Thus the NNJAG model seems to be general enough to capture all existing STCON algorithms.

For USTCON, Aleliunas et al. [1] showed that a random walk from any node s will hit all other nodes in its connected component in expected $O(nm)$ steps, m being the number of edges in the graph. This puts USTCON in Randomized Logspace $\mathcal{RL}$ and also implies the existence of a polynomial length Universal Traversal Sequence (UTS), i.e., a sequence of edge labels following which one can reach every node in a connected component from any starting node in that component. A deterministic JAG can trivially simulate such UTS using one pebble and $O(mn)$ states as it is a non-uniform computation device. A randomized JAG can also easily simulate a random walk in the same $O(\log n)$ space.

Restricting our attention to uniform deterministic computation, we witnessed a decrease in the space complexity of USTCON from $O(\log^2 n)$ of Savitch [19] and Nisan [13] to $O(\log^{3/2} n)$ by the NSW algorithm [14] and to $O(\log^{4/3} n)$ by the ATWZ algorithm [3]. Finally, Reingold [17] settled the deterministic space complexity of USTCON by discovering a (deterministic) $O(\log n)$ space algorithm. It follows that $\mathcal{SL} = \mathcal{L}$. Given the success of NNJAG in simulating algorithms for STCON, it is natural to ask if it can simulate those algorithms for USTCON as well. To our knowledge, no one has carefully investigated this question.

In this paper, we show that all the three algorithms: the NSW algorithm, the ATWZ algorithm and Reingold's algorithm can be implemented on a RAM-NNJAG which is the uniform counterpart of an NNJAG. Note that it is important to consider the simulation on a uniform model since a non-uniform one can follow a UTS to visit all nodes in the connected component of s and so USTCON in logspace becomes trivial.

As mentioned, a central focus in this area is the deterministic space complexity of STCON. One possible direction is to extend Reingold's algorithm to the directed case. In fact, Dinur et. al. [7] has extended Reingold's algorithm to directed graphs which are bi-regular. However, the feasibility of these three algorithms on a RAM-NNJAG model implies an obstruction to generalizing them to the directed case: Since Edmonds et. al. [8] proved a tight space lower bound of $\Omega(\log^2 n)$ for solving STCON on the NNJAG model (and hence the RAM-NNJAG model), our result implies that Reingold's algorithm does not immediately apply to directed graphs and, if we are to make progress on improving the space complexity for STCON, some new algorithmic techniques need to be developed.

In the next section, we will introduce the JAG and related models. Section 3 is an overview of the simulations followed by the details in Section 4 and 5.

2 The JAG and Related Models

A JAG as introduced in [6] is a non-uniform model. It consists of a sequence of automata $\{J_1, J_2, \ldots\}$ where the n-th automaton $J = J_n$ consists of a set of p distinguishable pebbles numbered 1 to p, a set of q states and a transition function δ. In general, p and q will be functions of n and the transition function δ also depends on n.

The input to J is a triple (G, s, t) where G is an n-node graph containing nodes s and t. For every node in G, its out-edges are labelled by consecutive integers starting from 0. The nodes in G are also labelled from 0 up to $n-1$. We define the *instantaneous description* (id) of J as the pair (Q, Π) where Q is the current state and Π is a mapping of pebbles to nodes specifying the current location of each pebble in the graph. When J is in id (Q, Π), the transition function δ determines the next move for J based on: (1) the state Q, and (2) which pebbles are on the same node and which are not, according to Π (i.e., for each pair of pebbles P and P', whether $\Pi(P) = \Pi(P')$). A move is either a *walk* or a *jump*. A *walk* (P, i, Q') consists of moving the pebble P along the edge labelled i that comes out of the node $\Pi(P)$ and then assuming state Q'. (If there

is no such edge, the pebble just remains on the same node.) A *jump* (P, P', Q') consists of moving pebble P to the node $\Pi(P')$ and then assuming state Q'. The automaton J is initialized to have state Q_0 and with pebbles $P_1, \ldots, P_{p-1}$ on node s and pebble P_p on node t (which makes node t distinguishable from the rest). It is said to *accept* an input (G, s, t) if it enters an accepting state on this input. It solves STCON for n-node graphs if for every input (G, s, t) where G is an n-node directed graph, it accepts the input iff there is a directed path from s to t in G. The definition for USTCON is similar.

It is easy to see that an id of a JAG can be specified using $p \log n + \log q$ bits by any ordinary computation model such as a Turing machine or a Random Access Machine. Thus we heuristically define this quantity as the space used by a JAG. The time used is the number of moves it made.

An NNJAG [16] is similar to a JAG except that the transition function depends on Q and Π. In other words, in NNJAG the transition function δ can use the actual names $\Pi(P_i)$. Due to its ability to see and work with node names, we need not put a pebble on t to make it distinguishable from the others. So we can put all pebbles on s initially and the definition would guarantee that every node that has ever been pebbled are reachable from s.

The RAM-JAG [15] consists of a finite state control together with p pebbles and a number of $O(\log n)$-bit registers which in total require $O(\log q)$ bits of storage. Its storage is defined as $(p \log n + \log q)$ bits. It can perform the usual RAM operations on the registers and also three special instructions: *walk*, *jump* and *compare*. The instructions *walk(P,j)* and *jump(P,P')* are the same as that in a JAG. The instruction *compare(P,P',R)* checks whether pebbles P and P' are on the same node and stores the result (T or F) in a register R. A RAM-NNJAG is similar to a RAM-JAG except that the instruction $compare(P, P', R)$ is replaced by $copy(P, R)$ which copies the name of the node pebbled by P to the register R.

It is straightforward to show that a RAM-JAG (resp. RAM-NNJAG) can be simulated by an ordinary JAG (resp. NNJAG) with $O(p)$ pebbles and $q^{O(1)}$ states. Thus, any lower bound on the JAG (resp. NNJAG) carries over to the RAM-JAG (resp. RAM-NNJAG) model.

3 Overview of the Simulations

At the highest level, all three algorithms are to (conceptually) construct a sequence of graphs $G_1, G_2, \ldots, G_\ell$ from the input graph $G = G_0$ such that connectivity between s and t in G is the same as that between two nodes s' and t' in G_ℓ.

For the NSW algorithm, G_k is obtained by choosing at most $|G_{k-1}|/f$ representative nodes in G_{k-1} and putting in edges so that two nodes u and v in G_{k-1} are connected if and only if their representatives in G_k are connected. Setting $f = 2^{\sqrt{\log n}}$ and $\ell = \sqrt{\log n}$, G_ℓ contains at most a constant number of nodes and hence *st*-connectivity can be trivially determined, say, by a DFS or BFS from the representative of s in G_ℓ. The ATWZ algorithm is similar except that it chooses $f = 2^{\log^{2/3} n}$ and $\ell = \log^{1/3} n$.

For Reingold's algorithm, G_k is obtained from G_{k-1} by performing a zigzag product with an expander graph H, followed by a graph powering operation. The number of nodes actually increases by a constant factor but the degree remains constant while the graph becomes more expanding. After $\ell = O(\log n)$ levels, the diameter is guaranteed to be $O(\log n)$. Thus *st*-connectivity is solved by exhausting all the (polynomially many) paths of length $L = O(\log n)$ on G_ℓ from one of the nodes corresponding to s in G_ℓ.

Due to the space limitation, one cannot store all the graphs. Instead, these algorithms just find out if an arbitrary pair of nodes u, v are connected by an edge in G_k (or which node is connected to a node u via its x-th edge) when necessary; and they achieve this by recursively asking for the appropriate edges and nodes in G_{k-1}. To carry out this on-demand scheme on a RAM-JAG/NNJAG, we show how an edge traversal in G_k can be effected by traversing appropriate edges in G_{k-1}. Specifically, we take the following approach:

1. Design a way to *store* a node of G_k in a RAM-JAG/NNJAG.
2. Design a procedure that, given a node u in G_k stored in a RAM-JAG/NNJAG and an edge label x, simulates the traversal of the x-th edge emanating from u in G_k such that the node thus reached is stored.

A natural idea to store a node u is to have a pebble on u. This is sufficient for the NSW and ATWZ algorithm since nodes in G_k are also nodes in the original graph G. In contrast, Reingold's algorithm blows up the number of nodes by a constant factor for each level. So we need a more generalized concept of storing a node and traversing an edge in G_k (to be given in next section).

4 Simulating Reingold's Algorithm

4.1 Reingold's Algorithm

In Reingold's algorithm, a preliminary step transforms the input graph G into a D-regular non-bipartite graph for some well chosen constant D. This is done in Logspace by replacing each node with a cycle and adding self-loops if necessary. Furthermore, it assumes that the graph G is specified by a *rotation map* $Rot_G : [n] \times [D] \to [n] \times [D]$ such that $Rot_G(u, a) = (v, b)$ if the a-th edge incident to u is $e = \{u, v\}$ which leads to v and this edge is the b-th edge incident to v.

Given an n-node, D-regular graph G and a D-node, d-regular graph H, the zig-zag product $G ⓩ H$ ([18]) is a graph with vertex set $[n] \times [D]$ such that every vertex has d^2 edges labelled by $(x, y) \in [d] \times [d]$. Its *rotation map* $Rot_{G ⓩ H}$ is defined as:

$$Rot_{G ⓩ H}((u, a), (x, y)) = ((v, b), (y', x')),$$

where $Rot_H(a, x) = (a', x')$, $Rot_G(u, a') = (v, b')$, and $Rot_H(b', y) = (b, y')$, see Figure 1 for an illustration. Note that in reverse, $Rot_{G ⓩ H}((v, b), (y', x')) = ((u, a), (x, y))$.

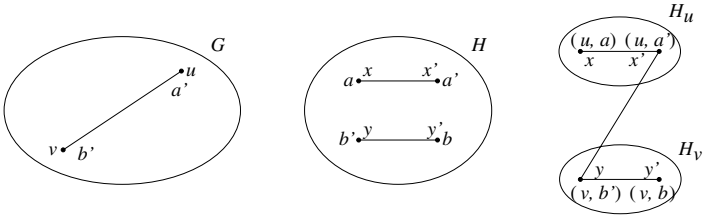

Fig. 1. Zigzag graph product: (Left) An edge in G, (Middle) Two edges in H, (Right) A path of 3 edges in the cross product of G and H. It corresponds to the edge labelled by (x, y) from node (u, a) in $G ⓩ H$.

In [18] a remarkable property concerning the spectral gap (i.e., the difference between 1 and its second largest eigenvalue of the normalized adjacency matrix) is proved which has the following direct consequence [17]:

$$1 - \lambda(G ⓩ H) \geq \frac{1}{2}(1 - \lambda(H)^2)(1 - \lambda(G))$$

where $\lambda(G)$ denotes the second largest eigenvalue of the normalized adjacency matrix of graph G. Essentially this shows that if H is chosen to be a good expander (i.e., a graph with a small second largest eigenvalue), then $G ⓩ H$ will have a spectral gap not much smaller than that of G. Meanwhile a powering of $G ⓩ H$ will increase the spectral gap of $G ⓩ H$. Reingold chose $D = d^{16}$ and an appropriate H with D vertices. Then $(G ⓩ H)^8$ is D-regular again, and has an increased spectral gap than G.

The main part of Reingold's algorithm is to repeatedly apply in turn the zig-zag product and graph powering to the input graph: For $k > 0$, define $G_k = (G_{k-1} ⓩ H)^8$ where ⓩ is the zig-zag operator and H is a fixed D-node, d-regular expander graph. Thus, G_k will be a d^{16}-regular graph of size $D|G_{k-1}|$. As $D = d^{16}$, G_k is D-regular again. For example, one can use the construction by Alon and Roichman [2] which gives a d^{16}-node, d-regular expander graph H for some constant d, with $\lambda(H) \leq 1/2$.

The effect of one step of the above transformation from G_{k-1} to G_k is to turn the (connected components of the) previous graph G_{k-1} into a more expanding one in G_k, measured in terms of the spectral gap. At the same time, the number of nodes of the transformed graph increases by a factor of D while its degree remains to be $D = d^{16}$. Each node u in $G = G_0$ corresponds to D^k nodes in G_k, for $k > 0$. They are connected among themselves (in G_k) and we denote them by $(u, a_0, \ldots, a_{k-1})$ where $(a_0, \ldots, a_{k-1}) \in [D]^k$. The transformation ensures that two nodes u and v in G are connected iff $(u, a_0, \ldots, a_{k-1})$ and $(v, b_0, \ldots, b_{k-1})$ are connected in G_k, for each $(a_0, \ldots, a_{k-1})$ and $(b_0, \ldots, b_{k-1})$ in $[D]^k$.

Reingold showed that after $\ell = O(\log n)$ steps, (any connected component of) G_ℓ has a spectral gap greater or equal to $1/2$. That means any pair of nodes in the same connected component in G_ℓ are joined by a path of length $L = O(\log n)$. Thus, on G_ℓ, we simply enumerate all possible paths of length L from, say, $(s, 1, 1, \ldots, 1)$ and see if we can reach $(t, 1, 1, \ldots, 1)$.

4.2 Simulation on a RAM-JAG

In this section, we will actually describe our simulation on a RAM-JAG as we do not need the power of an NNJAG to see the node names.

Since a node in G_k is of the form $u_k = (u, a_0, a_1, \ldots, a_{k-1})$ where u is a node in G and $(a_0, a_1, \ldots, a_{k-1}) \in [D]^k$, we *store* a node u_k by having a pebble P on the node u in G and storing the values $a_0, \ldots, a_{k-1}$ in k registers, $A_0, \ldots, A_{k-1}$, each of size $\log D$ bits. Note that the RAM-JAG initially has pebbles on node s in G. Therefore it is easy to store node $(s, 1, \ldots, 1)$ in G_ℓ by initializing ℓ registers appropriately.

Next, consider the traversal of an edge labelled a_k from a node u_k in G_k. Let $Rot_{G_k}(u_k, a_k) = (v_k, b_k)$. We will prove by induction on $k \geq 0$ that if u_k and a_k are stored by the RAM-JAG, it can traverse the a_k-th edge of u_k so that v_k and b_k are stored.

For $k = 0$, assume that the RAM-JAG stores u_k in pebble P (i.e., pebble P is on node u_k) and a_k is stored in a register A. Then the RAM-JAG can easily move pebble P from u_k to v_k by walking along the edge labelled a_k in G_k, i.e., the original input graph G. To compute the reverse label b_k, we try all possible edges from v_k and see which one brings us back to node u_k. That is, we first mark the node u_k with an extra pebble P'. Then we move P from node v_k along its first edge and see if it meets P'. If not, we jump P to P' (so P is at u_k again) and walk along the a_k-th edge so that P arrives at v_k again. Then we try the second edge of v_k and so on. In this way, the RAM-JAG can compute and store b_k in a register and have node v_k pebbled.

For $k > 0$, recall that $G_k = (G_{k-1} ⓩ H)^8$. Therefore, an edge in G_k corresponds to a path of length eight in $G_{k-1} ⓩ H$. Thus, we write a_k as a sequence of 8 edge labels, $(x_{k,1}, y_{k,1})$, $(x_{k,2}, y_{k,2})$, $\ldots$, $(x_{k,8}, y_{k,8})$ in $G_{k-1} ⓩ H$. This in turn can be viewed as a path of 8 edges in G_{k-1} beginning from node u_{k-1} but with edge labels "permuted" by the expander H. Suppose $u_k = (u_{k-1}, a_{k-1})$ and $Rot_H(a_{k-1}, x_{k,1}) = (a'_{k-1}, x'_{k-1})$. Then the first edge label in G_{k-1} to follow is a'_{k-1}. Note that the RAM-JAG can figure out (a'_{k-1}, x'_{k-1}) without traversing G_{k-1} since H is fixed. Let $Rot_{G_{k-1}}(u_{k-1}, a'_{k-1}) = (v_{k-1}, b'_{k-1})$. Since u_k stored in the RAM-JAG implies u_{k-1} is also stored, we can assume (by induction hypothesis) that the RAM-JAG can traverse the a'_{k-1}-th edge of u_{k-1} in G_{k-1} so that v_{k-1} and b'_{k-1} is stored. Suppose $Rot_H(b'_{k-1}, y_{k,1}) = (b_{k-1}, y'_{k,1})$. Then again, the RAM-JAG can compute b_{k-1} and $y'_{k,1}$ from b'_{k-1} and $y_{k,1}$ using the fixed structure of H only. In terms of $G_{k-1} ⓩ H$, the first edge leads to the node (v_{k-1}, b_{k-1}). Note that since the reverse edge b_{k-1} is known, one can traverse the next edge $(x_{k,2}, y_{k,2})$ in $G_{k-1} ⓩ H$. Thus, the RAM-JAG can repeat the traversal for the remaining seven edges, $(x_{k,2}, y_{k,2}), \ldots, (x_{k,8}, y_{k,8})$ in order to complete the traversal of one edge in G_k. Finally, observe that the 8 reverse edge labels on the path of length 8 in $G_{k-1} ⓩ H$ arranged in the order $(y'_{k,8}, x'_{k,8}), \ldots, (y'_{k,1}, x'_{k,1})$ is nothing but the reverse edge label, b_k, for the single edge just traversed in G_k. Thus, the RAM-JAG is also able to store b_k.

Thus our induction statement is proved and it follows that the traversals of the D^L paths in G_ℓ can be carried out on a RAM-JAG.

Now consider the storage per level. Storing the 8 reverse edge labels takes $O(\log D)$ space. To store a node $u_k = (u, a_0, \ldots, a_{k-1})$ in G_k, note that the storage for $u_{k-1} = (u, a_0, \ldots, a_{k-2})$ can be charged to level $k-1$ or below. So we just charge $O(\log D)$ bits for storing a_{k-1} in level k. Hence, the total storage for all the levels is $O(\log n)$.

5 Simulating the NSW and ATWZ Algorithms

5.1 The NSW and ATWZ Algorithms

The core of the algorithms is the *shrink* procedure which takes a graph G_{k-1} as input and return G_k. It computes the set of representatives of G_{k-1}, i.e., the node set of G_k, as follows. First, it computes a set L_k of landmark nodes which includes s and t together with some nodes drawn from a pairwise independent space. This can be implemented as $L_k = \{s, t\} \cup \{u \in G_{k-1} | 1 \leq ua_k + b_k \leq q/(6f)\}$ for some pair (a_k, b_k) drawn from a field F_q of size polynomial in n.

Then for any u in G_{k-1}, we find a neighbourhood, $N(u)$, of u. In NSW, we follow a short universal traversal sequence (UTS) of length $2^{O(\log^2 f)}$ from u and define $N(u)$ as the set of all nodes encountered in the UTS as well as their immediate neighbours. In ATWZ, we run a number of pseudo-random walks of length $2^{O(\log^2 f)}$ from u to approximate the average number of such walks hitting each node v or its immediate neighbors. Any node v with approximate average at least $1/n$ is put into $N(u)$.

With $N(v)$, we define the representative, $rep_{L_k}(u)$, of u as follows. If $N(u)$ contains the whole component of u, then u is not represented in G_k (component too small). Otherwise, u is either represented by the minimum v in $N(u) \cap L_k$ or by itself in case $N(u) \cap L_k$ is empty. (We will always treat s and t as the minimum nodes in L_k. This ensures that they are always represented by themselves unless their components are too small.) It was shown in [14] that $|G_{k-1}|/|G_k| \geq f$ for a constant fraction of (a_k, b_k)'s, i.e., the size reduction is guaranteed if we go over the polynomially many (a_k, b_k)'s. Finally, for all $(u, v) \in G_{k-1}$, we have $(rep_{L_k}(u), rep_{L_k}(v)) \in G_k$.

5.2 Simulation on a RAM-NNJAG

Since a RAM-NNJAG cannot cycle through all the nodes in G_k, it cannot immediately see if the choice of (a_k, b_k) achieves the required size reduction factor f. However, a RAM-NNJAG can try all possible sequences of $(a_1, b_1), (a_2, b_2), \ldots (a_\ell, b_\ell)$. At the end, at least one choice of the sequence will be good enough.

Now fix a sequence of $(a_1, b_1), \ldots, (a_\ell, b_\ell)$. We will store a node u by having a pebble on it. We will (conceptually) assign labels to edges in G_k such that the x-th edge of node u will lead to its x-th smallest neighbour in G_k. Suppose node u in G_k is stored and we are to traverse its x-th edge. In other words, we need to find the x-th smallest neighbours of u in G_k, To this end, we compute all those nodes u' in G_{k-1} represented by u in G_k (to be described in next paragraph). Then we compute all nodes v' which is an immediate neigbour of some u', followed

by their representatives, $rep_{L_k}(v')$, using a forward UTS. Finally, we choose the x-th smallest representative.

The most difficult step is to compute those u' in G_{k-1} represented by u in G_k For NSW, we make use of a reversible UTS. The idea is to convert the graph G_{k-1} into one with edges symmetrically labelled, i.e if the ith neighbor of vertex v is the vertex u, then the ith neighbor of vertex u is also the vertex v. Suppose this is done and suppose u is the i-th node along a short UTS from u' and let the sequence of edge labels in the UTS be $\sigma_1\sigma_2\cdots\sigma_i$. Then the i-th reverse UTS, $\sigma_i\cdots\sigma_2\sigma_1$, starting from u will bring us to u'. Thus to find all u' represented by u, we try, for each possible i, to walk from u using the i-th reverse UTS to reach a candidate node u'. Then we walk from u' using the (forward) UTS to verify if u is the minimum node in L_k. For ATWZ, the idea is similar except that we try for each pseudorandom walk and for each i, the i-th reverse pseudorandom walk from u to arrive at a candidate u'. Note that checking for $v \in L_k$ is easily done by checking if $1 \leq a_k v + b_k \leq q/(6f)$. Note also that when $u = s$ and in the process of discovering those u' represented by u, we hit t, then we can stop and answer: "s, t connected". Thus if $N(s)$ contains the whole component of s and we have not discovered t, we can stop and answer: "s, t not connected". More detail will be given in our technical report [12].

To convert the graph into a symmetrically labelled one, we expand a degree-d node v into a cycle of d nodes $v_0, \ldots, v_{d-1}$ if d is even; or $d+1$ nodes $v_0, \ldots, v_d$ if d is odd. The edges on the cycles are labelled such that the edge (v_i, v_{i+1}) (where for the vertex indices, we take modulo d if d is even; or modulo $d+1$ if d is odd) is labelled with 0 (or 1) if i is even (or odd respectively). For arbitrary edge (u, v) in the original graph, if u is the k-th neighbor of v and v is the l-th neighbor of u, then connect v_k and u_l using an edge labelled 2 at both ends. Finally for each node v of odd degree, add a self-loop to v_d with edge label 2.

As for the storage, storing the sequence $(a_1, b_1), \ldots, (a_\ell, b_\ell)$ requires $O(\ell \times \log n)$ space. For the NSW algorithm, each of the ℓ levels requires $O(1)$ pebbles and $O(\log^2 f)$ bits to generate and follow the short UTS. Thus $O(\log^{3/2} n)$ space is needed. For the ATWZ algorithm, each level requires $O(\log n)$ space and overall it also needs $O(\log^2 f + \ell \log n) = O(\log^{4/3} n)$ space to generate the pseudorandom walks. Hence in total $O(\log^{4/3} n)$ space is needed.

Acknowledgments

We would like to thank Xiaotie Deng, Andrew Yao, Frances Yao and Ming Li for several discussions. We especially thank Steve Cook for helpful comments on a previous draft of the paper.

References

1. R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th Annual Symposium on Foundations of Computer Science*, pages 218–223, San Juan, Puerto Rico, October 1979. IEEE.

2. N. Alon and Y. Roichman. Random Cayley graphs and and expanders. *Random Structures and Algorithms*, 5(2):271–284, 1994.
3. R. Armoni, A. Ta-Shma, A. Wigderson, and S. Zhou. An $O(\log(n)^{4/3})$ space algorithm for (s,t) connectivity in undirected graphs. *Journal of the ACM*, 47(2):294–311, 2000.
4. G. Barnes, J. F. Buss, W. L. Ruzzo, and B. Schieber. A sublinear space, polynomial time algorithm for directed *s-t* connectivity. *SIAM Journal on Computing*, 27(5):1273–1282, 1998.
5. P. Berman and J. Simon. Lower bounds on graph threading by probabilistic machines. In *24th Annual Symposium on Foundations of Computer Science*, pages 304–311, Tucson, AZ, November 1983. IEEE.
6. S. A. Cook and C. W. Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM Journal on Computing*, 9(3):636–652, August 1980.
7. I. Dinur, O. Reingold, L. Trevisan, and S. Vadhan. Finding paths in nonreversible markov chains. Technical Report TR05-022, Electronic Colloquium on Computational Complexity, 2005.
8. J. Edmonds, C. K. Poon, and D. Achlioptas. Tight lower bounds for *st*-connectivity on the NNJAG model. *SIAM Journal on Computing*, 28(6):2257–2284, 1999.
9. P. Gopalan, R. Lipton, and A. Mehta. Randomized time-space tradeoffs for directed graph connectivity. In *FSTTCS'03*, pages 208–216, 2003. LNCS 2914.
10. N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, October 1988.
11. H. R. Lewis and C. H. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19(2):161–187, August 1982.
12. P. Lu, J. Zhang, C.K. Poon, and J. Cai. Simulating undirected *st*-oonnectivity algorithms on uniform JAGs and NNJAGs. Technical report, 2005.
13. N. Nisan. $RL \subseteq SC$. In *Proceedings of the Twenty Fourth Annual ACM Symposium on Theory of Computing*, pages 619–623, Victoria, B.C., Canada, May 1992.
14. N. Nisan, E. Szemerédi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *33rd Annual Symposium on Foundations of Computer Science*, Pittsburgh, PA, October 1992. IEEE.
15. C. K. Poon. *On the Complexity of the ST-Connectivity Problem.* PhD thesis, University of Toronto, 1996.
16. C. K. Poon. A space lower bound for *st*-connectivity on node-named JAGs. *Theoretical Computer Science*, 237(1-2):327–345, 2000.
17. O. Reingold. Undirected st-connectivity in log-space. In *Proceedings of the Thirty Seventh Annual ACM Symposium on Theory of Computing*, pages 376–385, 2005.
18. O. Reingold, S. Vadhan, and A. Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders. *Annals of Mathematics*, 155(1), 2001.
19. W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.
20. R. Szelepcsényi. The method of forcing for nondeterministic automata. *Bulletin of the European Association for Theoretical Computer Science*, 33:96–100, October 1987.