A Fast Randomized LOGSPACE Algorithm for Graph Connectivity

Uriel Feige*

August 5, 1996

Abstract

We study the relationship between *undirected graph reachability* and *graph connectivity*, in the context of randomized LOGSPACE algorithms. Aleluinas *et al.* [2] show that graph reachability (checking whether there is a path connecting vertices $\mathcal{S}$ and $\mathcal{T}$) can be decided in logarithmic space and polynomial time, by starting a random walk at $\mathcal{S}$, and checking whether $\mathcal{T}$ is hit within some time limit. The randomized algorithm has one-sided error (with small probability, it fails to determine that $\mathcal{S}$ and $\mathcal{T}$ are connected).

The reachability algorithm may be used in order to decide (with one sided error) whether a graph is connected, by running it $n-1$ times, each time with a different target vertex $\mathcal{T}$. This increases the running time by a factor of n.

In this paper we give an alternative randomized LOGSPACE algorithm for graph connectivity. Its running time varies between $O(n^2)$ steps and $O(n^3)$ steps, depending on the structure of the input graph. This matches the fastest known RLOGSPACE algorithm for reachability, up to a constant factor. Our algorithm has two-sided error.

1 Introduction

We study the problem of deciding *graph connectivity* by algorithms restricted to space logarithmic in the input size. Our research is motivated by the unclear relation between *connectivity* and *reachability* (also known as *undirected $\mathcal{S}$-$\mathcal{T}$-connectivity*, or USTCON) in this context.

Suppose we are given a LOGSPACE algorithm for deciding reachability. (Note: we do not know whether such algorithms exist.) That is, given a graph G and two vertices $\mathcal{S}$ and $\mathcal{T}$, the algorithm determines whether there is a path in G connecting $\mathcal{S}$ and $\mathcal{T}$. Then the

*Department of Applied Math and Computer Science, The Weizmann Institute, Rehovot 76100, Israel. `feige@wisdom.weizmann.ac.il`. Supported by a Koret Foundation fellowship.

same algorithm can be used to test connectivity, by checking separately whether vertex v_1 is reachable from any other vertex of the graph. This procedure of checking connectivity may be slower by a factor of n (where n denotes the number of vertices in G) from the original algorithm for reachability, since the reachability algorithm is employed $n-1$ times. Hence it seems that *connectivity* requires more time than *reachability.* (Remark: alternatively, connectivity of G can be decided by running a reachability algorithm just once, on a larger graph that is composed from n copies of G. However, it is hard to imagine that this would improve the overall running time.)

On the other hand, assume that we are given a LOGSPACE algorithm for connectivity. Can we use it to decide reachability in LOGSPACE? We do not know whether this can be done at all. So perhaps deciding reachability is more difficult than deciding connectivity (in the context of bounded space algorithms)?

Thus an interesting question is: which problem, connectivity or reachability, can be decided more quickly by LOGSPACE algorithms? This is a hypothetical question, since we do not know that any of these problems can be decided in deterministic LOGSPACE.

The above question becomes more concrete in the context of randomized algorithms. Aleluinas *et al.* [2] show that reachability can be decided in logarithmic space and polynomial time, by a randomized algorithm that starts a random walk at $\mathcal{S}$, and checks whether $\mathcal{T}$ is hit within some time limit. This time limit is polynomially related to n, and is easily computable in LOGSPACE from the description of G. The random algorithm has one-sided error – with small probability, it fails to determine that $\mathcal{S}$ and $\mathcal{T}$ are connected. (A "Las Vegas" algorithm for reachability is designed in [4], but its expected running time is much larger than the running time of [2]'s algorithm.)

It follows that connectivity can also be decided with one sided error in RLOGSPACE, and the time required is at most a factor of n higher than that required for reachability. But can we do better?

In this paper we give an alternative randomized LOGSPACE algorithm for graph connectivity. Its running time matches that of the fastest known RLOGSPACE algorithm for reachability, up to a constant multiplicative factor. This running time is $O(n^2 d_{ave}(1/d)_{ave})$, where d_{ave} is the average degree of vertices in G, and $(1/d)_{ave}$ is the average of the inverse of degrees. In particular, for graphs that are "nearly regular" (that is, $d_{ave}(1/d)_{ave}$ is bounded by a constant) our algorithm decides connectivity in $O(n^2)$ steps, and for graphs that are highly irregular, our algorithm requires $O(n^3)$ steps. Our algorithm has two-sided error.

2 The Main Idea

Like the algorithm for reachability, we also use random walks on graphs. However, we do not use them in order to test whether two vertices are connected, but as a procedure for estimating the size of a connected component within a constant factor. (In this paper, the size of a connected component is the number of edges that it contains. This is a natural choice in the context of random walks.) We show that the number of random walk steps required in order to estimate the size of the connected component depends on the size of the connected component itself, and not on the size of the whole graph. On the intuitive level, this certainly makes sense. But proving this fact (with tight enough bounds) is not so simple, and our proof uses proof techniques that were developed only recently.

Using our estimation procedure, we decide connectivity by trying to detect connected components that are small. This we do by random sampling as follows. Assume that there is a connected component containing $s < m$ edges, where m is the number of edges in the graph. Then in m/s random tries, we are expected to sample an edge of this component. Now using a random walk, it can be determined in time that depends on s, say $f(s)$, that the component is really smaller than m, and hence the graph is not connected. Thus in order to detect a component of size roughly s, we spend $O(\frac{m}{s}f(s))$ time. We try this sample-and-test procedure for values of s that are powers of 2. It turns out that the dominating term in the running time happens when $s \simeq m$, where we spend $f(m)$ time. Hence if our goal is to decide connectivity as quickly as reachability, it suffices that $f(m)$ (the time to approximate the size of a connected component of size m) be similar to the time for deciding reachability.

We remark that we have an additional phase in our algorithm, due to the poor ratio of approximation of our estimation algorithm (our analysis does not give ratios arbitrarily close to 1 – see the discussion in Section 8.) We use the estimation procedure in order to rule out the existence of small components. Once we know that the graph has only large components (each comprising of a constant fraction of the edges), we can use a reachability algorithm in order to decide connectivity. We select at random two edges, and see if they are connected. If the graph is not connected but all connected components are large, then there is constant probability that the two edges are in different connected components, and hence are not connected. This procedure can be repeated a constant number of times to increase the confidence of detecting disconnectivity. (Actually, we use a slightly more efficient procedure for increasing the confidence in this test).

3 Approximating the Size of a Connected Component

One method for approximating the size of a connected component is based on hashing, as used by Sipser [13] in a different context. Suppose we want to distinguish between the case that the component of edge e is small (has less then s edges), or large (more that $8s$ edges).

1. Select a random hash function $h : E \to [2s]$ (where E is the name space for edges, and $[2s]$ denotes the integers from 1 to $2s$), and a random $r \in [2s]$.

2. Start a random walk of length $t = O(s^2)$ at an end point of e. If for some edge e' along the walk, $h(e') = r$, output *large*. Else output *small*.

Clearly, if the component is small, then the probability of outputting *small* is at least $1/2$. If the component is large, we may use the result of [3] to show that with high probability, the random walk visits at least $8e$ edges (regardless of the size and shape of the connected component). If h is chosen from a universal family of hash functions [5, 13], then it is likely that the hashed value of some visited vertex agrees with r.

A different method for approximating the size of a connected component uses the fact that the expected time for a random walk to return to an edge is $2s$, where s is the number of edges of the connected component. Hence a good estimate on the expected return time gives a good estimate on s. The problem in estimating the expected return time is that the return time has very large deviations. (Consider the extreme example of an infinite path – the expected return time to the starting vertex is infinite, though the return time is finite with probability 1.) Thus this method requires us to determine how many sampled return times suffice in order to obtain a good estimation of the expected return time.

Analysis shows (after certain optimizations to be described in Sect. 4) that both methods require roughly the same number of random walk steps in order to estimate the size of a connected component. We choose the latter method (based on return time) over the former method (based on hash functions). There are two main reasons for this choice.

1. The need to apply a hash function on every step of the random walk may slow down the random walk by more than a constant factor (depending on the model of computation – if it supports random walk moves more easily than hashing).

2. The second method requires less sophistication. E.g., if a person actually takes a random walk in order to estimate the size of maze, then the first method is not applicable, since vertices of the maze have no name that can be hashed. The second method is applicable: by leaving his hat on the edge on which the walk starts, the traveller recognizes this edge each time he returns to it.

4 Getting Tighter Results

We want to show that our algorithm decides connectivity as quickly as the fastest known RLOGSPACE algorithm decides reachability. We first need to understand how quickly can reachability be decided. The RLOGSPACE algorithm of [2] decides reachability by starting a random walk at $\mathcal{S}$, and seeing if it eventually hits $\mathcal{T}$. If $\mathcal{S}$ and $\mathcal{T}$ are connected, then the expected running time of the algorithm is $H[\mathcal{S}, \mathcal{T}]$ – the expected *hitting time* from $\mathcal{S}$ to $\mathcal{T}$. But the algorithm also needs an estimate on the number of steps it should run before it can safely decide that $\mathcal{S}$ and $\mathcal{T}$ are not connected. This estimate should be based on properties of the input graph that are quickly computable in RLOGSPACE.

If we only consider n, the number of vertices in G, then [2] prove that $H[\mathcal{S},\mathcal{T}] < n^3$. A better bound takes into account m, the number of edges in G, namely - $H[\mathcal{S},\mathcal{T}] < 2mn$ [2]. This improvement is significant for sparse graphs. By bounds on the diameter (see e.g. [1]), it can be shown that $H[\mathcal{S},\mathcal{T}] < 6mn/d_{min}$, where d_{min} is the minimum degree of a vertex of G. This improvement shows that for regular graphs, $H[\mathcal{S},\mathcal{T}] < 3n^2$. The best known bound on the hitting time uses the parameter $\hat{R} = \sum_{v\in G} 1/(d_v+1)$, and gives $H[\mathcal{S},\mathcal{T}] < 6m\hat{R}$ (see [9, 7]). Observe that for any graph, $\hat{R} < n/d_{min}$. Observe also that $\hat{R}$ can be approximated to arbitrary (fixed) presicion in LOGSPACE.

Hence our goal is to decide connectivity in time $O(m\hat{R})$. This places strict limits on the running time of the procedure for estimating the size of connected components. The tricky issue is to have this running time depend on $\hat{R}$, which is a global property of the whole graph, and may not be reflected in local regions of the graph (a single connected component). To overcome this problem, we use an idea of [3, 9]. We artificially increase the number of edges in G by a constant factor, by placing self loops on vertices of G. This is done in such a way that guarantees quick estimation of sizes of connected components, where the size of a connected component now also takes into account its self-loops.

Additional complications in our algorithm result from our care not to lose logarithmic factors in the running time. Our algorithm uses a randomized procedure for estimating the size of connected components. This procedure is used many times, each time with a different starting edge. There is constant probability for each of these estimates to be incorrect (e.g., due to unlucky coin tosses, the procedure may falsely think that a connected component is small, when in fact it comprises the whole graph). The simple way of decreasing the probability of such errors is by repeating each experiment $O(\log n)$ independent times, and taking a majority vote as to whether the connected component in question is large or small. This makes each decision highly reliable, at the cost of a logarithmic overhead. However, we use a different method that has only constant overhead. This method is based on efficient techniques of handling unreliable information, as developed in [10].

5 The Algorithm

The algorithm FASTCON checks in randomized logspace whether a graph G is connected. The algorithm has two sided error.

The input to the algorithm is an undirected graph with n vertices and m edges. Equivalently, we may view the input as a symmetric directed graph with $2m$ directed edges, which we call *arcs*. For an arc $e = (u, v)$ leading from vertex u to vertex v, we call u the *head* of e and v the *tail* of e. Though the input graph does not contain self loops, our algorithm will introduce additional arcs that are self loops (of the form (u,u)).

Algorithm FASTCON[G]:

1. Compute $\hat{R} = \sum_v 1/(d_v + 1)$.
2. Modify G to $\hat{G}$, by (implicitly) placing $m/\hat{R}(d_v+1)$ self loops on each vertex v. Hence $\hat{G}$ has $3m$ arcs.
3. For each s a power of 2 satisfying $m/\hat{R} \le s \le m/6$, do:
   - Scan arcs of $\hat{G}$ in order. For each arc e, with probability $2/s$ perform procedure COMPONENT[e,s]. (Procedure COMPONENT[e,s], described below, returns *small* if the connected component of arc e contains less than s arcs.)
4. If at least one of the calls to procedure COMPONENT returned *small*, then declare *not connected.*
5. Select at random 25 arcs in $\hat{G}$, and consider the vertices that are their heads. Start a random walk on G at the first such vertex. If all these vertices are visited within $30m\hat{R}$ steps, then declare *connected.* Otherwise, *not connected.*
6. If at any time, the overall number of random walk steps made by the algorithms exceeds $c'm\hat{R}$, where c' is a suitably large constant, then terminate the algorithm and declare *not connected.*

Remark: For simplicity, we assume that for each vertex v the number $m/\hat{R}(d_v+1)$ of self loops is an integer. The algorithm with minor modifications works even if the number of self loops is not an integer, since the use that the algorithm makes of this number is in computing probabilities (e.g., of making transitions for random walks), and integrality is irrelevant for computing probabilities. Alternatively, $m/\hat{R}(d_v+1)$ can be rounded to the nearest integer, requiring a change in the constants in our analysis.

We now describe the procedures used by Alg. FASTCON. Procedure TEST[e,s] checks the hypothesis that the connected component in which arc e is a member contains at most s arcs.

TEST[e,s]: Let u be the tail of arc e. Start at u a random walk of length $t = cs^2\hat{R}/m$, where c is some universal constant (our current proof uses $c = 1026$, but this can be improved). If e was traversed more than $t/3s$ times, then output *small.* Else output *large.*

Procedure COMPONENT[e,s] increases the confidence in procedure TEST[e,s], to the extent that it is highly unlikely that Procedure COMPONENT[e,s] ever declares the connected component of e to be smaller than s, when in fact it is larger than $18s$.

COMPONENT[e,s]:

1. $i \leftarrow i_0$, where i_0 is a universal constant.
2. Repeat $TEST[e, s]$ for i times.
3. If the majority of all i answers was *large*, then return *large*.
4. If $i < 96 \ln n$, then return to Step 2 with $i \leftarrow i + i_0$.
5. Return *small*.

6 Measuring Time Complexity

It is most convenient to measure the time complexity of our algorithm in terms of the number of random walk steps that it performs. This abstracts away details of the computation model, of the encoding of the input, and of the actual implementation of the algorithm. We remark that these details make huge differences in the actual time that it takes to perform a single random walk step. For example, making a single random walk step is very slow on a Turing machine that has only sequential access to its input, and much faster on a RAM that has random access to the input. However, the actual time it takes to perform a random walk step affects our algorithm for connectivity and [2]'s algorithm for reachability in the same way. Hence it plays no role in determining which algorithm is faster.

Measurement of time in terms of number of random walk steps makes most sense if the following two assumptions hold:

1. The expected time to make a single random walk step is independent of the current vertex at which the walk is located (up to multiplicative constant factors).
2. The time that the algorithm spends in doing nonrandom walk operations (e.g., maintaining counters) is dominated by the time the algorithm spends in making random walk steps.

If the graph G is regular, both assumptions are reasonable. We argue informally that even for nonregular graphs the assumptions above are reasonable.

1. For the first assumption, consider a graph G that is encoded as a list of $2m$ arcs in lexicographic order. For example, the 3-cycle would be encoded as

   $(1, 2)$, $(1, 3)$, $(2, 1)$, $(2, 3)$, $(3, 1)$, $(3, 2)$.

   If the algorithm can compare two $O(\log n)$-bit numbers in unit time, and has random access to its input, then a basic step of a random walk can be performed in LOGSPACE and time $O(\log n)$. If the walk is at vertex i, the algorithm can use binary search to find the first and last appearances of arcs that originate at i, and then select an outgoing arc at random. We note that regardless of the degree of vertex i, $\Omega(\log n)$ time would typically be necessary just to locate the first outgoing arc from i.

2. To justify the plausability of our second assumption, we need to consider the operations performed by our algorithm other than random walk steps. Recall that in our algorithm, we modify G into a new graph $\hat{G}$, by adding self loops on vertices. The number of self loops on each vertex depends on its degree, and on $\hat{R} = \sum_{v \in G} 1/(d_v+1)$. Due to the limited space available to the algorithm, this modification cannot be done explicitly on the input graph. Instead, the number of self loops is recomputed for each vertex reached by the random walk. We argue that this recomputation at every vertex slows down the performance of a random walk step by at most a constant multiplicative factor.

   The value of $\hat{R}$ is precomputed once (up to precision of $O(\log n)$ bits), and stored. The time to do so is dominated by the time to take $m\hat{R}$ random walk steps. Thereafter, for each vertex v visited by the random walk, its degree d_v would typically be computed by the random walk procedure, and thereafter, the number of its self loops (which is $m/\hat{R}(d_v + 1)$) can be computed by a constant number of arithmetic operations on $O(\log n)$-bit numbers. (If needed, this computation can be speeded up by rounding each number to the nearest power of 2. This only affects the constants in our analysis.)

7 Proof of Correctness

For simplicity of the analysis, we will assume that all numbers are of convenient form (e.g., divide with no remainder, are exact powers of 2, and so on). In the actual implementation of the algorithm, this would not be the case, and numbers would need to be rounded (to nearest integer, etc.). We do not wish to elaborate on this point, but the principles of analysis are robust enough to withstand any (intelligent) rounding scheme.

The stopping condition (item 6 in the description of Alg. FASTCON) ensures that the total number of random walk steps taken by the algorithm is $O(m\hat{R})$. In the proof of Thm. 1 below, we ignore the existence of this stopping condition, and prove correctness of the resulting modified algorithm. The analysis of the effect of the stopping condition on the correctness of the algorithm is postponed until later (around Lemma 6).

Theorem 1 *Algorithm FASTCON decides connectivity with two sided error of* $1/3$*. If* G *is connected, then with probability at least* $2/3$*, Alg. FASTCON returns* connected*. If* G *is not connected, then with probability at least* $2/3$*, Alg. FASTCON returns* not connected*.*

Proof: Algorithm FASTCON tries to identify a small connected component in G. For technical reasons, G is transformed into a new graph $\hat{G}$ with $3m$ arcs. The size of components in $\hat{G}$ is measured by the number of arcs, rather than number of vertices. We first show that the connected components in $\hat{G}$ cannot be too small.

Lemma 2 *The smallest connected component in $\hat{G}$ contains at least $m/\hat{R}$ arcs.*

Proof: Let $\mathcal{C}$ be the component with the smallest number of arcs in $\hat{G}$. Let $v \in \mathcal{C}$ be the vertex of $\mathcal{C}$ that has highest degree in G. It has d_v neighbors in $\mathcal{C}$, and each such neighbor has at least $m/\hat{R}(d_v+1)$ self loops. Counting also the number of self loops on v, we have altogether at least $m/\hat{R}$ self loops in $\mathcal{C}$. $\Box$

Let s be the size of the smallest connected component in $\hat{G}$, and assume that $s \leq m/6$. Then with probability at least $1 - e^{-2} > 5/6$, Step 3 of FASTCON selects an arc e of this connected component, and invokes COMPONENT[e,s]. The next two lemmas show that the fact that G is not connected would be detected with high probability.

Lemma 3 *Procedure TEST[e,s] approximates the number of arcs in the connected component of e. If the number of arcs is at most s, then TEST[e,s] returns* small *with probability at least $2/3$. If the number of arcs is at least $18s$, then TEST[e,s] returns* large *with probability at least $2/3$.*

Proof: Consider the case that the connected component of e contains at most s arcs. Then by standard analysis of random walks (see [2], among others), the expected return time to e is at most s. By linearity of expectation, the expected time until e is traversed $t/3s$ times is at most $st/3s = t/3$. By Markov's inequality, with probability at most $1/3$ it takes more than t random walk steps until e is traversed $t/3s$ times. This completes the proof for the case that the connected component is small.

We are left the case that the connected component of e contains more then $18s$ arcs. The proof of this case relies heavily on the proof technique in [1, 9], and is deferred to the appendix. $\Box$

Lemma 4 *Procedure COMPONENT[e,s] increases the confidence in procedure TEST[e,s]. If the number of arcs is at most s, then COMPONENT[e,s] returns* small *with probability at least $5/6$. If the number of arcs is at least $18s$, then COMPONENT[e,s] returns* large *with probability at least $1 - n^{-3}$.*

Proof: By Lemma 3, procedure TEST returns the correct answer with probability at least $2/3$. Using the Chernoff bound it follows that the probability that the majority of i independent runs of TEST return the wrong answer is at most $e^{-i/32}$. For successive values of i, the probability of error decreases in an exponential rate. For procedure COMPONENT to incorrectly return *large*, one such majority vote must be wrong. The probability of this happening is at most $\sum_{k=1}^{\infty} e^{-ki_0/32} < 1/6$, when i_0 is sufficiently large. To incorrectly return *small*, the last majority vote must be wrong, and this has probability at most $e^{-96 \ln n/32} = n^{-3}$. $\square$

The case that $\hat{G}$ is not connected, but the smallest connected component contains at least $m/6$ arcs, is handled by Step 5 of FASTCON, as the following lemma shows.

Lemma 5 *If each connected component of $\hat{G}$ contains at least $m/6$ arcs, then Step 5 of Alg. FASTCON decides connectivity with two sided error* $1/4$.

Proof: Assume first that G is connected. Then the cover time of G is at most $\frac{20}{3} m\hat{R}$ [7]. By Markov inequality, there is probability at least $7/9$ that all vertices are visited in $30m\hat{R}$ steps.

Assume now that G is not connected, and that each connected component contains at least $m/6$ arcs. Altogether, $\hat{G}$ has $3m$ arcs. Hence the arcs of $\hat{G}$ can be partitioned into two parts disconnected from each other, such that the smallest of the parts contains at least a fraction of $1/18$ of the arcs of $\hat{G}$. If 25 arcs are chosen independently at random, then with probability at least $3/4$ not all of them are in the same part. A walk cannot visit vertices that are the heads of arcs that belong to different parts of the graph. $\square$

Now we are ready to complete the proof of Theorem 1. Consider first the case that the graph is connected. Procedure COMPONENT is invoked at most $m \log m$ times. On each of these times, it returns *large* with probability at least $1 - n^{-3}$ (by Lemma 4). Hence with overwhelming probability, Step 5 of Alg. FASTCON is reached, and then the algorithm decides *connected* with probability at least $3/4$ (by Lemma 5).

Consider now the case that G is not connected. Assume first that $\hat{G}$ has a connected component of size $s < m/6$. Then with probability at least $5/6$, step 3 of Alg. FASTCON invokes procedure COMPONENT on an edge of this connected component, and then COMPONENT returns *small* with probability at least $5/6$. Hence the small connected component is detected with probability at least $(5/6)^2 > 2/3$.

Assume now that $\hat{G}$ is not connected but has only large connected components. Then we may assume that Step 5 of Alg. FASTCON is reached (otherwise, FASTCON must have announced *not connected*, which is correct), and then Lemma 5 ensures that the answer is *not connected* with probability at least $3/4$. $\square$

We now show that Alg. FASTCON remains correct even if item 6 is included in the description of the algorithm. Notice that item 6 can introduce an error only if G is connected. Hence it suffices to show that without item 6, the expected number of random walk steps that FASTCON makes on connected graphs is $O(m\hat{R})$. Then declaring *not connected* whenever the algorithm exceeds a total of $c'm\hat{R}$ random walk steps (for large enough c') has only small effect on the error of the algorithm.

Step 5 of FASTCON is performed within $O(m\hat{R})$ random walk steps. For the other parts of FASTCON, we need the following lemma.

Lemma 6 *If G is connected, then procedure COMPONENT[e,s] runs in expected time $O(s^2\hat{R}/m)$.*

Proof: If $\hat{G}$ is connected, then the probability that COMPONENT[e,s] goes k times through its main loop decreases exponentially as k increases. Hence the expected number of calls that COMPONENT[e,s] makes to TEST[e,s] is bounded by a constant (that depends on i_0), and the proof follows since TEST[e,s] takes $O(s^2\hat{R}/m)$ steps. □

For connected G, for each value of s, Step 3 of algorithm FASTCON is expected to invoke procedure COMPONENT $O(m/s)$ times. Hence, by Wald's equation ([12], for example) and Lemma 6, the expected number of random walk steps per value of s is $O((\frac{m}{s})(\frac{s^2\hat{R}}{m})) = O(s\hat{R})$. Since s advances by multiples of 2, the total sum converges to $O(m\hat{R})$, as desired.

8 Conclusions

For randomized LOGSPACE algorithms, we showed that connectivity can be decided without the overhead incurred by repeatedly applying algorithms for reachability. Instead, we based our algorithm on a procedure for approximating the size of a connected component.

We remark that a procedure for estimating the size of a connected component can also be used to decide reachability. First estimate the size of the connected component of the starting vertex $\mathcal{S}$. Then "hang" a large subgraph on the target vertex $\mathcal{T}$. If $\mathcal{S}$ and $\mathcal{T}$ are connected, then the estimate on the size of the connected component of $\mathcal{S}$ increases. This remark has interesting consequences:

1. If the size of connected components can be approximated in deterministic LOGSPACE, then it can also be computed exactly.

2. Reachability can be decided in randomized LOGSPACE in time that depends on the size of the connected component of $\mathcal{S}$ (or of $\mathcal{T}$), rather than the size of the whole graph.

Our procedure TEST(e,s) approximates the size of a connected component up to a ratio of 18, by counting the number of returns to the starting edge of a random walk. If this ratio of approximation is improved to below 2, then Step 5 of the algorithm becomes unnecessary, since any disconnected graph must have a connected component with less than half of the edges. We do not know whether our analysis (the proof of Lemma 3) can be improved to show ratios of approximations arbitrarily close to 1 (without loosing more than a constant factor in the length of the random walk). Gillman [11] proves a Chernoff type bound on random walks for a similar problem, but his results are applicable in our context only in the special case that the connected component is known to be an expander (otherwise, the rate of convergence that Gillman proves is too slow for our purpose). In Section 3 we described an alternative procedure, based on hashing, for estimating the size of a connected component. We remark that this alternative procedure can be used in order to obtain ratios of approximation arbitrarily close to 1. All that is required in order to distinguish between components of size s and $(1+\epsilon)s$ is that the range of the universal family of hash functions has size cs/ϵ, where c is some constant ($c = 4$ would do).

Acknowledgements: Moni Naor suggested the question of how quickly can connectivity be decided in randomized LOGSPACE. He also took time to listen to the answer.

I thank the referees for the ICALP94 conference for their useful comments.

References

[1] D. J. Aldous. "Reversible Markov chains and random walks on graphs". *Draft of first six chapters of book*, January 26, 1993.

[2] 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 FOCS*, 218–223, 1979.

[3] G. Barnes and U. Feige. "Short Random Walks on Graphs". To appear in *SIAM Journal on Discrete Mathematics.* Preliminary version appeared in *Proc. of* 25^{th} *STOC, 1993, pp. 728-737.*

[4] A. Borodin, S. Cook, P. Dymond, W. Ruzzo, and M. Tompa. "Two applications of inductive counting for complementation problems". *SIAM J. Comput.,* 18(3):559-578, 1989.

[5] L. Carter and M. Wegman. "Universal Hash Functions". *JCSS, 18(2): 145–154, 1979.*

[6] A. Chandra, P. Raghavan, W. Ruzzo, R. Smolensky, and P. Tiwari. "The electrical resistance of a graph captures its commute and cover times". In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 574-586, Seattle, WA, May 1989.

[7] D. Coppersmith, U. Feige, and J. Shearer. "Random Walks on Regular and Irregular Graphs". To appear in *SIAM Journal on Discrete Mathematics.* Preliminary version appeared as *Technical report CS93-15*, the Weizmann Institute, Israel, 1993.

[8] P. G. Doyle and J. L. Snell. "Random Walks and Electrical Networks". *The Mathematical Association of America*, 1984.

[9] U. Feige. "A Randomized Time-Space Tradeoff of $\tilde{O}(m\hat{R})$ for USTCON". In *34th FOCS*, 238–246, 1993.

[10] U. Feige, D. Peleg, P. Raghavan, and E. Upfal. "Computing With Noisy Information". *SIAM Journal on Computing, 23(5), 1001–1018, 1994.*

[11] D. Gillman. "A Chernoff Bound for Random Walks on Expander Graphs". In *34th FOCS*, 680–691, 1993.

[12] S. Ross. "Applied probability models with optimization applications". Holden-Day, San Francisco, California. 1970.

[13] M. Sipser. "A Complexity Theoretic Approach to Randomness". In *15th STOC*, 330–335, 1983.

[14] P. Tetali. "Random walks and the effective resistance of networks". *Journal of Theoretical Probability*, 4:101-109, 1991.

APPENDIX.

We continue the proof of Lemma 3. Recall that we need to prove the following. Assume that a connected component of arc e contains at least $18s$ arcs in $\hat{G}$. Then for a random walk of length $t = cs^2\hat{R}/m$ (where c is a constant to be defined later) that starts at the tail of e, the probability of more than $t/3s$ traversals of e is at most $1/3$. In the proof, we use the following notation:

e, e' – arcs.
d_v – number of vertices adjacent to vertex v.
S^c – the complement of set S.
T_S – the time by which a random walk first hits a member of set S.
$E_e[.]$ – expectation for quantity in [.], for a random walk that starts at the tail of e.
N_e – number of traversals of arc e.
$N_e(T)$ – number of traversals of arc e up to time T.

The proof of Lemma 3 is based on a proof technique that was developed in [1, 9].

Proof: Let $A = \{e' | E_{e'}[N_e(t)] > t/18s\}$, and let $A^c = \{e' | E_{e'}[N_e(t)] \leq t/18s\}$.

Lemma 7 *The number of arcs in A satisfies $|A| < 18s$.*

Proof: Let $\bar{e}$ denote the arc that is antiparallel to the arc e for which we measure N_e in the definition of set A. Let $B = \{(u,v) | E_{\bar{e}}[N_{(u,v)}(t)] > t/18s\}$, where (u,v) denotes an arc with head u and tail v. We show that $(u,v) \in B$ if and only if $(v,u) \in A$.

Let (x,y) and (u,v) be two arcs. For any t', the probability that a walk that starts at y crosses (u,v) at time t' is equal to the probability that a walk that starts at u crosses (y,x)

at time t' (by the reversiblity of the random walk process). It follows that $E_{(x,y)}[N_{(u,v)}(t)] = E_{(v,u)}[N_{(y,x)}(t)]$, and therefore an edge is in B if and only if the anti-parallel edge is in A.

To complete the proof of Lemma 7, observe that by linearity of expectation, $|B| < 18s$. It follows that $|A| < 18s$. □

By our assumption that the connected component contains at least $18s$ arcs, it follows that A^c is nonempty. Hence $E_e[N_e(t)] \leq E_e[N_e(T_{A^c})] + t/18s$. We wish to bound $E_e[N_e(T_{A^c})]$. In order to do so, we use known relations between random walks and *effective resistance* in graphs [8]. The effective resistance between two vertices u and v, denoted by $R[u,v]$, is the effective resistance between the two vertices in an electrical network that has the same topology as the graph, where each edge of the graph represents a resistor of 1 ohm in the network. Interestingly, for any arc e in the same connected component as vertex u, $R[u,v]$ is exactly the expected number of times that e is traversed in a random walk that starts at u, hits v, and then returns to u.

Lemma 8 *If vertices u, v, and arc e are in the same connected component, then*

$$R[u,v] = E_u[N_e(T_v)] + E_v[N_e(T_u)]$$

The proof of Lemma 8 can be found in [6, 14]. In order to obtain an upper bound on $E_e[N_e(T_{A^c})]$, we shall associate a resistance R_A with the set A of arcs, and use a lemma similar to Lemma 8.

Let V_A denote the set of vertices that are tails of arcs in A, and let V_A^c denote its complement. Clearly, if a random walk reaches V_A^c, then the random walk must have traversed an arc not in A. Hence $E_e[N_e(T_{V_A^c})] \geq E_e[N_e(T_{A^c})]$. Let B_A be the "boundary" of V_A, that is, $v \in B_A$ if and only if $v \in V_A$ and v has a neighbor in V_A^c. Similarly, let I_A be the "interior" of V_A, that is, $v \in I_A$ if and only if $v \in V_A$ and $v \notin B_A$.

We assume throughtout the proof that the tail of e is in V_A, as otherwise $E_e[N_e(T_{V_A^c})] = 0$. The process by which a random walk that starts at V_A reaches a vertex in V_A^c can be broken into two phases. In the first phase, a vertex in B_A is reached. In the second phase, the walk starts in B_A and reaches a vertex in V_A^c. Regardless of which vertex $u \in B_A$ is the starting vertex for the second phase, it always has some neighbor $v \in V_A^c$. By standard properties of electrical resistance (monotonicity), $R[u,v] \leq 1$. It now follows from Lemma 8 that the expected number of times that arc e is traversed in the second phase is at most 1. Hence $E_e[N_e(T_{B_A})] + 1 \geq E_e[N_e(T_{V_A^c})]$. To bound $E_e[N_e(T_{B_A})]$, it is convenient to assume that the boundary vertices B_A form a clique. This assumption can be made without loss of generality, because a random walk that stops the moment it reaches B_A is not affected by the internal pattern of edges in B_A. We now define R_A as $\max_{u \in I_A, v \in B_A} R[u,v]$. (We define $R_A = 0$ if I_A is empty.)

Lemma 9 *For R_A and B_A as defined above*

$$R_A \geq E_e[N_e(T_{B_A})]$$

Proof: If the tail of e is not in I_A, then it is in B_A, and $E_e[N_e(T_{B_A})] = 0$, satisfying the lemma (resistances are always nonnegative). For the case that the tail of e is in I_A, denote it by v, and let w be any vertex in B_A. Then $E_e[N_e(T_{B_A})] \leq E_v[N_e(T_w)]$. By Lemma 8, $E_v[N_e(T_w)] \leq R[v, w]$, and by definition $R[v, w] \leq R_A$. $\square$

Hence it remains to bound R_A. In order to do so, we quote without proof a result of [7].

Lemma 10 *Let G be an undirected graph in which vertices are allowed to have any number of self loops. Recall that d_v denotes the number of vertices adjacent to vertex v. Then if there is a path connecting vertices s and t, then*

$$R[s,t] \leq 3 \sum_v \frac{1}{d_v + 1} - 1$$

By monotonicity of effective resistance, R_A cannot decrease if we remove all vertices of V_A^c, together with their incident edges.

Corollary 11

$$R_A \leq 3 \sum_{v \in I_A \bigcup B_A} \frac{1}{d_v + 1} - 1$$

where the vertices of V_A^c are ignored when computing the degrees of vertices in B_A.

Proof: By our assumption that the vertices of B_A form a clique, the vertex induced subgraph on $I_A \bigcup B_A$ is connected, and Lemma 10 applies. $\square$

Now it remains to bound $\sum_{v \in I_A \bigcup B_A} \frac{1}{d_v+1}$. Let A_I be the set of arcs that are incident with I_A. Then $A_I \subset A$, and $|A_I| \leq |A|$. Recall our convention regarding placing self loops on vertices of G. Then

$$A_I \geq \sum_{v \in I_A} (d_v + \frac{m}{\hat{R}(d_v + 1)}) \geq \frac{m}{\hat{R}} \sum_{v \in I_A} \frac{1}{d_v + 1}$$

Hence $\sum_{v \in I_A} 1/(d_v + 1) \leq \hat{R}\frac{|A|}{m}$. Observe also that $\sum_{v \in B_A} 1/(d_v + 1) < 1$, since the vertices in B_A are assumed to form a clique.

Now the proof of Lemma 3 can be completed. $E_e[N_e(T_{A^c})] \leq R_A + 1$. From Corollary 11, $R_A \leq 3 \sum_{v \in I_A \bigcup B_A} \frac{1}{d_v+1} - 1$. Hence we obtain $R_A \leq 2 + 3\hat{R}|A|/m$. Substituting in the value of $|A|$ from Lemma 7, we obtain $E_e[N_e(T_{A^c})] \leq 3(1 + 18s\hat{R}/m) \leq 57s\hat{R}/m$ (the last inequality follows from our assumption that $m/\hat{R} \leq s$, see Step 3 in the description of the algorithm FASTCON). For $t > 1026 s^2 \hat{R}/m$, we have that $E_e[N_e(T_{A^c})] < t/18s$, and $E_e[N_e(t)] < t/9s$. By Markov's inequality, $Pr[N_e(t) \geq t/3s] \leq 1/3$. $\square$