An Exact Sublinear Algorithm for the Max-Flow, Vertex Disjoint Paths and Communication Problems on Random Graphs

Dorit S. Hochbaum

Operations Research, Vol. 40, No. 5 (Sep. - Oct., 1992), 923-935.

Stable URL:
http://links.jstor.org/sici?sici=0030-364X%28199209%2F10%2940%3A5%3C923%3AAESAFT%3E2.0.CO%3B2-8

Operations Research is currently published by INFORMS.

Your use of the JSTOR archive indicates your acceptance of JSTOR's Terms and Conditions of Use, available at http://www.jstor.org/about/terms.html. JSTOR's Terms and Conditions of Use provides, in part, that unless you have obtained prior permission, you may not download an entire issue of a journal or multiple copies of articles, and you may use content in the JSTOR archive only for your personal, non-commercial use.

Please contact the publisher regarding any further use of this work. Publisher contact information may be obtained at http://www.jstor.org/journals/informs.html.

Each copy of any part of a JSTOR transmission must contain the same copyright notice that appears on the screen or printed page of such transmission.

JSTOR is an independent not-for-profit organization dedicated to creating and preserving a digital archive of scholarly journals. For more information regarding JSTOR, please contact support@jstor.org.

http://www.jstor.org/
Thu Oct 28 02:25:21 2004

AN EXACT SUBLINEAR ALGORITHM FOR THE MAX-FLOW, VERTEX DISJOINT PATHS AND COMMUNICATION PROBLEMS ON RANDOM GRAPHS

DORIT S. HOCHBAUM

University of California, Berkeley, California

(Received January 1989; revisions received August 1990, July 1991; accepted September 1991)

This paper describes a randomized algorithm for solving the maximum-flow maximum-cut problem on connected random graphs. The algorithm is very fast—it does not look up most vertices in the graph. Another feature of this algorithm is that it almost surely provides, along with an optimal solution, a proof of optimality of the solution. In addition, the algorithm's solution is, by construction, a collection of vertex-disjoint paths which is maximum. Under a restriction on the graph's density, an optimal solution to the NP-hard communication problem is provided as well, that is, finding a maximum collection of vertex-disjoint paths between sender-receiver pairs of terminals. The algorithm lends itself to a sublogarithmic parallel and distributed implementation. Its effectiveness is demonstrated via extensive empirical study.

In this paper, we study three problems on undirected random graphs: the maximum-flow, the maximum number vertex-disjoint paths, and the communication problem. Whereas the first and second problems can be solved in polynomial time, the third is NP-complete (Lynch 1975). With the algorithms described in this paper, the first two problems are equally easy on random graphs—they are solvable in sublinear time, without looking up all the vertices and most of the edges. Moreover, along with the solution, a certificate of optimality is delivered almost surely. All these features apply to the most difficult of the three problems—the communication problem, with the exception of a certain graph density range (or alternatively, for up to a given number of communicating pairs, depending on the graph's density).

The maximum-flow and the related minimum-cut problems have been studied extensively in the literature of operations research and computer science. Given a pair of vertices, a source and a sink, the problem is to push the maximum amount of flow from the source to the sink, subject to capacity constraints on the edges, and conservation of flow constraints on the vertices. We shall study this problem with 0-1 edge capacities.

There have been numerous results on improving the running time of algorithms for the maximum-flow problem. Given a graph $G = (V, E)$ with $|V| = n$, $|E| = m$, an algorithm by Goldberg and Tarjan (1988) runs in time $O(mn \log_e(n^2/m))$ (see the Notes section for definitions of the O-notations). Improvements include a randomized algorithm by Cheriyan, Hagerup and Mehlhorn (1990) with an expected running time of $O(mn + n^2(\log n)^2)$. This algorithm has been "derandomized" to a deterministic algorithm by King, Pao and Tarjan (1991) with a reported running time of $O(mn + n^{2+\epsilon})$ for any positive ϵ. For the case of 0-1 capacities, Even and Tarjan (1975) offer a more efficient algorithm with a running time of $O(\min\{m^{2/3}n, n^{2/3}m\})$.

In certain real-time applications, such as routing messages, this running time is considered too slow. In environments where the network structure resembles a random network, i.e., the availability of each edge for sending a unit of flow is an i.i.d. random variable, our algorithm offers a super-fast alternative. Moreover, in routing applications it is particularly desirable to use algorithms that can be implemented in a distributed environment. That is, each vertex in the network can be an independent processor that "decides" on further routing of the message when only local information is available, such as the status of the vertex and its neighbors. The algorithms proposed here possess natural distributed implementation because every step uses only local information.

The model of random graphs used here is the

Subject classifications: Analysis of algorithms, computational complexity: sublinear flow algorithms. Communications: communication problem on random graphs. Networks/graphs, flow algorithms: algorithms for communication and max-flow on random graphs.
Area of review: OPTIMIZATION.

0030-364X/92/4005-0923 $01.25
© 1992 Operations Research Society of America

common one. A random graph $\mathscr{G}_{n,p}$ is a graph on n vertices where an edge exists (or has unit capacity in our interpretation), with probability p independently of other edges. Also, p is a function of n denoted by $p = c_n/n$, where the expected degree of a vertex in $\mathscr{G}_{n,p}$ is c_n. As c_n grows larger, the graph becomes denser.

The algorithm here has the feature that in addition to an optimal solution, it also provides, almost surely, a certificate of optimality in the form of a minimum cut. This feature permits the use of the algorithm as a preprocessor to a maximum-flow algorithm even if the random structure of the network is not evident. It can be thought of as a heuristic that frequently provides an optimal solution and a proof of its optimality. In the absence of the proof of optimality, one can always resort to a selected maximum-flow algorithm, while the penalty in terms of the added running time, being sublinear, is heavily dominated by the running time of any maximum-flow algorithm, and thus is negligible.

Algorithms typically run faster on the average on random graphs because pathological structures that bring about the worst-case running time are avoided with high probability. The maximum-flow problem has been studied on random graphs: Grimmet and Welsch (1978) showed that for a complete undirected graph, where the capacities are i.i.d. random variables with finite mean μ, the maximum flow is asymptotically equal almost surely to $n\mu$. This result is also a by-product of the analysis of our algorithm for the 0-1 capacities. Karp, Motwani and Nisan (1987) and Hassin and Zemel (1988) consider dense capacitated graphs, $0 < p < 1$, and identify a linear time algorithm $O(m)$ for producing, with probability going to 1, a maximum flow on such graphs. For this range of densities, p constant, our algorithm runs in $O(n)$ time, which is sublinear in m. Motwani (1990) took the approach of studying the length of an augmenting path in a random graph. He shows that the length of such paths is bounded in random graphs. Consequently, all augmenting path algorithms have a faster running time on random graphs. This running time $O(m \log n/\log np)$ is more than quadratic in the running time of our algorithm. Motwani makes the point that his proof shows that the average case behavior of the algorithms used for maximum flow is very efficient. Here we show that algorithms constructed to take advantage of the structure of random graphs can do substantially better.

On networks with 0-1 capacities, finding the maximum flow is equivalent to finding the maximum number of edge-disjoint paths. By the construction of our algorithm, it will not only provide a solution along edge-disjoint paths, but these paths will also be *vertex-disjoint*. Therefore, it will find in sublinear time, almost surely, the maximum number of vertex-disjoint paths between the source and the sink, along with a certificate of optimality (the number of vertex-disjoint paths cannot exceed the number of edge-disjoint paths).

In summary, compared to other algorithms devised in the literature, the analysis here holds for the full range of densities, the running time of the algorithm is faster, and the flow it produces is along vertex-disjoint paths. This latter feature allows the algorithm to be used for solving two problems other than the maximum-flow problem, at the same running time.

The **communication problem** is to find a maximum collection of vertex-disjoint paths between specified sender-receiver pairs. Although this is more difficult than the maximum vertex-disjoint paths problem, it is also solvable by our algorithm on random networks except that this solution is limited to a certain range of graph densities. Formally, to find paths between $\bar{c}$-specified pairs of nodes, the requirement is that $\bar{c}$ not exceed $1/16\sqrt{nc_n/\log_e n}$, where the random graph is $\mathscr{G}_{n,p}$.

Our results apply to connected random graphs, i.e., those with an expected degree of at least $\log_e n + w_n$, where w_n is any function of n going asymptotically to infinity. The condition on the expected degree is necessary to assure connectedness with probability going to 1 (Erdös and Rényi 1960).

In addition to the algorithmic results, we also derive certain theoretical results concerning the properties of random graphs:

1. The minimum cut is almost surely a partition of the vertices into two sets, one of which is the source (sink) and the other is the remainder of the graph. Whether it is the source or the sink that is separated by the minimum cut from the rest of the graph is determined by which of the two is of lower degree. For dense graphs, this result has also been observed in Grimmet and Welsch (1978), Karp (1979), Karp, Motwani and Nisan (1987), and Hassin and Zemel (1988). The significance of this observation is that it makes a certificate of optimality in the form of a minimum cut readily available.
2. There exists a maximum flow along vertex-disjoint paths almost surely.
3. For $c_n \geq \sqrt{n} \log_e n$, the flow is along (vertex-disjoint) paths of length 3 or less. A similar result

has been established by Hassin and Zemel for capacitated dense random graphs.

The first property has been generalized to the minimum k-cut problem, which is a generalization of the minimum cut problem. Goldschmidt and Hochbaum (1990a) prove that a partition of a random graph into k components with the minimum number of edges between components is achieved when $k - 1$ of the components consists of single vertices, and the kth one is the remainder of the graph.

The bidirectional tree search is the technique at the heart of all the procedures we use. The bidirectional tree search begins with two sets of vertices, the neighbors of the source s, $N(s)$, and the neighbors of the sink t, $N(t)$. The search process attempts to create $\bar{c} = \min\{|N(s)|, |N(t)|\}$ vertex-disjoint linking paths between these two sets. Assume without loss of generality that $|N(s)| = |N(t)| = \bar{c}$, otherwise the excess number of vertices can be deleted without affecting the algorithms or their analysis. We denote the vertices in these two sets by $N(s) = \{s_1, \ldots, s_{\bar{c}}\}$ and $N(t) = \{t_1, \ldots, t_{\bar{c}}\}$.

We distinguish between the so-called *sequential* and *parallel* implementations of the search. These are italicized to emphasize the difference between them and the concepts of sequential algorithms running on a single processor versus parallel algorithms running on multiple parallel processors. The implementation of both the *sequential* and *parallel* will require a single processor. Whereas the *sequential* procedure searches for one path at a time between a specific pair of vertices, the *parallel* procedure searches in "parallel" for any possible path connection between all neighbors of the source and all neighbors of the sink.

In the *sequential* implementation, two breadth-first-like trees are grown, one from each of the pair of vertices to be linked. This is done by alternately adding a layer of leaves to each tree. The leaves added are the set of all neighbors unvisited so far of the previous layer of leaves. Once a neighbor of a leaf of one tree is in the other tree the trees "meet" and a path has been created. For denser graphs this is slightly modified by adding only a subset of the neighbors of the leaves. (The full details of the *sequential* procedure are given in Section 5). The *parallel* version simultaneously grows several trees, rooted at all the neighbors of the source and the sink. It works iteratively and at each iteration it adds a single layer of leaves to all trees. When a node in one of the trees on the source side has a neighbor on the sink side (or vice versa), a path is created between the respective roots of these two trees. As a result, the *parallel* version of the algorithm does not assure the pairing of specific vertices in contrast to the *sequential* procedure. Whenever a path is identified, the vertices in the two trees, rooted at the vertices now joined with this path, are all deleted from the graph.

The critical part of the analysis is to show that not too many vertices get deleted in the process of creating the trees (once added to a tree each vertex is no longer considered for any other tree or path). The *parallel* version tends to eliminate fewer vertices than the *sequential*, hence it is the implementation of choice for solving the maximum-flow and the vertex-disjoint paths problems. We demonstrate the circumstances under which the *sequential* and the *parallel* approaches are effective. We also analyze the number of neighbors selected for each leaf, which is not necessarily all neighbors like in a breadth-first-search procedure, and study the properties of two such search strategies.

In case it is desirable to produce short paths, we use Goldschmidt and Hochbaum's (1990b) greedy matching algorithm to derive such paths when $c_n \geqslant \alpha\sqrt{n}\log_e n$, for some specific constant α, in time which is sublinear in the number of vertices. The running time of the greedy matching is linear in the number of vertices when $c_n = \Theta(n)$, and there are $\Theta(n)$ pairs to match; hence, this running time is the fastest possible. The $\Theta(n)$ running time is still sublinear in the input size which is dominated by the expected number of edges, $\frac{1}{2}pn^2$.

Finally, we present a range of empirical results. In less than ten minutes we can solve to optimality, on an IBM/PC, problems on networks of 30,000 vertices. Frequently, asymptotic theoretical results only hold in implementations for astoundingly large graphs. For our algorithm, its high speed and performance, as indicated by finding and verifying an optimal solution within a few seconds, is already realizable for graphs with as few as 1,000 vertices.

We conclude that for any network problem with the appropriate random structure, our algorithm is simple, practical and very fast. It also lends itself easily to the distributed or parallel implementations in an environment of multiple processors that run faster than $\Theta(\log n)$.

The plan of the paper is as follows. The *parallel* procedure is described in the next section, followed by a probabilistic analysis of this procedure in Section 2. Section 3 describes the use of the greedy matching algorithm for dense graphs. Section 4 summarizes the recommended use of the *parallel* and matching

algorithms, and comments on the issue of identifying the graph density required to determine which procedure to run. Section 5 describes the *sequential* algorithm, and Section 6, its probabilistic analysis. Section 7 studies the application of the algorithms for the communication and the maximum vertex-disjoint paths problems. Here the problems are given as two sets of vertices in a random graph that have to be linked via vertex-disjoint paths. It is proved that the size of such sets is unrestricted for the vertex-disjoint paths problem, and is restricted to $\frac{1}{16}\sqrt{n/\log_e n}\,\sqrt{c_n}$ vertices for the communication problem. In Section 8, we assume the existence of multiple processors and describe how to implement the algorithms in a distributed or a parallel processing environment. The distributed version runs in sublogarithmic time. Section 9 describes the empirical results that indicate that the theoretical results in the preceding sections are indeed realizable in practice.

Extra care has been taken to derive almost sure probability expressions, as opposed to results with probability going to 1. We shall say that a property $\mathscr{P}$ holds for random graphs almost surely, if the probability P_n that the property $\mathscr{P}$ holds for a graph $\mathscr{G}_{n,p}$ satisfies that $\sum_{n=1}^{\infty}(1-P_n)$ is finite. To have a property hold almost surely it suffices to have $P_n > 1 - 1/(n^{1+\delta})$ for some $\delta > 0$.

1. THE PARALLEL BIDIRECTIONAL TREE SEARCH

The bidirectional tree search is a procedure for identifying paths between $N(s)$ and $N(t)$. There may also exist other paths between s and t. Therefore, we first apply a preprocessing phase for removing these other paths that are of length 1 or 2 between s and t. This is done by recording the existence of an edge (s, t), or edges of the type (s_i, t) and (s, t_j) for all $s_i \in N(s)$ and $t_j \in N(t)$. These paths and all vertices in the paths are deleted, and their count is eventually added to the maximum flow or to the maximum number of vertex-disjoint paths identified by the *parallel* search procedure. The sets $N(s)$ and $N(t)$ are also searched for vertices of degree zero, that is, vertices with no neighbors in $V - \{s, t\} - N(s) - N(t)$. Such vertices are eliminated and the value of $\bar{c}$ is determined subsequently.

We assume that the random graph's realization G is given by an adjacency matrix and an adjacency list. We maintain two arrays $root_v$ and $pred_v$, of length n each, and an array of length $2\bar{c}$. These arrays contain labels and are initially empty. The array $root_v$, contains in the ith position once vertex i is reached, the name s_p or t_q of the root of the tree to which vertex i belongs. The second array $pred_v$, contains in the ith position for any vertex i reached, the index of its immediate predecessor in the tree. The third array $delete_{s_i}$ has a flag in the s_i position or the t_j position if the tree rooted at that vertex has been deleted, otherwise the entry is empty.

The process of "growing" trees in the *parallel* procedure amounts to finding two neighbors for each of the leaves of the $\bar{c}$ trees rooted at $\{s_1, \ldots, s_{\bar{c}}\}$, and then finding two neighbors for each of the leaves of the $\bar{c}$ trees rooted at $\{t_1, \ldots, t_{\bar{c}}\}$. An identified neighbor of v, u is labeled with the name of the root of the tree $root_v$. It also gets the label $pred_u = v$ to enable the recovery of the path. This process doubles the number of leaves with each addition of a layer. The doubling procedure may not always work because some leaves may not have two neighbors, but rather one or none. The double procedure reports failure whenever it finds two vertices considered consecutively, with zero or one neighbor each in the remaining graph. This is implemented as follows: If a vertex has only one neighbor v, then we look for two neighbors of v from the remaining graph, and consider those two vertices as the leaves in the new layer; if a vertex has no neighbors, then we search for two additional neighbors of its predecessor, and consider those as leaves in the new layer. It is proved (Lemma 1) that failure does not occur almost surely.

When the total number of leaves on the s-side and on the t-side exceeds $4\sqrt{n \log_e n}$, the *linking* phase takes place. At this point, there exist edges between the leaves of half of the trees at least, almost surely (this is proved in Theorem 2). The procedure for identifying the edges linking half pairs of the trees has two variants that depend on the density of the graph (the details are described later). Given a procedure for identifying those edges, the *linking* phase is accomplished by alternately linking half of the trees, deleting the linked trees, and then doubling the number of leaves in the remaining trees. As a result, the number of leaves at the beginning of each iteration, in which half of the trees are linked, is fixed. At most, $\log_2\lceil\bar{c}\rceil$ such stages are performed to find a maximum flow and a maximum number of vertex-disjoint paths.

The method of identifying the links between half of the trees on the s-side and half of the trees on the t-side depends on the graph density: If the graph is sparse, c_n is $O(\sqrt{n \log n})$, and the linking is performed by inspecting all neighbors of one leaf of a tree at a time. If, on the other hand, the graph is dense, c_n is $\Omega(\sqrt{n \log n})$, then all edge slots between one selected leaf of a tree and all the leaves of the opposite side are

inspected. This means that for each specific pair of vertices, we check whether there is an edge with these endpoints. This is done by looking up the appropriate entry in the vertex-vertex adjacency matrix.

The matching algorithm on random bipartite graphs is proposed as an alternative *linking* algorithm. For $p = \Omega(\sqrt{\log n/n})$ (more precisely for $\alpha\sqrt{\log_e n/n} < p < 1$, for some $\alpha > 0$), we use the matching algorithm to link directly, via paths of length one, the sets $N(s)$ and $N(t)$, without applying the doubling phase first.

For the *parallel* search, a failure is reported when the doubling has failed or when there are still unlinked trees on both the s and t sides, but no more undeleted neighbors.

The *parallel* procedure is applicable to the full range of p where the graph $\mathscr{G}_{n,p}$ is connected, i.e., for $(\log_e n + w_n)/n \leqslant p \leqslant 1$, with any function w_n such that $\lim_{n\to\infty} w_n = \infty$.

2. PROBABILISTIC ANALYSIS OF THE PARALLEL ALGORITHM

All the theorems and lemmas are stated in terms of n, the initial number of vertices in the random graph. We utilize the validity of these theorems to establish results for the remaining graph that contains less than n vertices. This is justified since in Corollary 3 we prove that there are at least $\Theta(n)$ vertices remaining at any iteration. Hence, all the results hold asymptotically as claimed.

Lemma 1. *The doubling procedure can be accomplished almost surely.*

Proof. The doubling procedure fails when two given vertices are of a degree less than two.

$$\begin{aligned}
&\Pr[\text{a given vertex has 0 or 1 neighbors}]\\
&\quad\leqslant (1-p)^{n-1} + (n-1)p(1-p)^{n-2}\\
&\quad= (1-p)^{n-2}(1+pn).
\end{aligned}$$

For p constant, $0 < p < 1$, this probability is exponentially diminishing to zero. For $p \to 0$,

$$\begin{aligned}
(1-p)^{n-2}(1+pn) &\leqslant 2(1-p)^n pn\\
&= 2c_n\left[\left(1-\frac{c_n}{n}\right)^{-n/c_n}\right]^{-c_n} \leqslant 2\frac{c_n}{e^{c_n}}.
\end{aligned}$$

So, for $c_n \geqslant 2\log_e n$, none of the leaves will have less than two neighbors almost surely. For $c_n = \log_e n + w_n$, with any w_n such that $\lim_{n\to\infty} w_n = \infty$ (the lower limit on c_n), given two vertices, at least one of them has two neighbors or more almost surely. As a result, the doubling process can be accomplished successfully almost surely.

Theorem 1. *If the linking phase of the parallel procedure starts when each side has at least $4\sqrt{n\log_e n}$ leaves, then it is completed successfully almost surely.*

Proof. The value of L is selected in such a way that it is possible to link almost surely one half of the remaining tree pairs at each iteration. One iteration of the *linking* procedure will consist of either inspecting the edge slots in the bipartition induced by the s-leaves and the t-leaves or the neighbors of certain nodes depending on the graph's density.

Suppose that at a given iteration there are i trees to be connected on each side, or i is the minimum of the number of trees in each side. Due to the construction in which the number of leaves of each tree is doubled at each iteration, all trees have precisely the same number of leaves L/i. The probability that a vertex will have no neighbor on the opposite side is $(1-p)^L$. The probability that all leaves of one tree will have no link to the opposite side is $(1-p))^{L^2/i}$. Hence,

$$\begin{aligned}
P(i) &= \Pr\left[\left[\frac{i}{2}\right]\text{ subtrees get linked}\right]\\
&= [1-(1-p)]^{L^2/i}][1-(1-p)^{L/i(1-1/i)L}]\\
&\quad\cdots[1-(1-p)^{L/i(1-\lceil i/2\rceil-1/i)L}]\\
&> [1-(1-p)^{L^2/2i}]^{i/2}.
\end{aligned}$$

The probability of success in all $\lceil\log_2\bar{c}\rceil$ iterations is then for $\bar{c} = \min\{|N(s)|, |N(t)|\}$:

$$\begin{aligned}
\prod_{j=0}^{\lfloor\log_2\bar{c}\rfloor} P\left(\frac{\bar{c}}{2^j}\right) &\geqslant [1-(1-p)^{L^2/2\bar{c}}]^{\bar{c}/2}\\
&\quad\cdot[1-(1-p)^{L^2/2\bar{c}}]^{\bar{c}/4}\\
&\quad\cdot\ldots\cdot[1-(1-p)^{L^2}]\\
&> 1-(1-p)^{L^2/2\bar{c}}]^{\bar{c}\log\bar{c}/2}.
\end{aligned}$$

We now determine the least size of L, the number of leaves, that guarantees that the *linking* algorithm is completely successfully almost surely; i.e., there exists some $\delta > 0$ such that,

$$[1-(1-p)^{L^2/2\bar{c}}]^{\bar{c}\log_2\bar{c}/2} > 1-\frac{1}{n^{(1+\delta)}}.$$

Since

$$e^{-1/n^{(1+\delta)}} \geqslant 1-\frac{1}{n^{(1+\delta)}},$$

it suffices to have

$$[1 - (1 - p)^{L^2/2\bar{c}}]^{\bar{c}\log_2\bar{c}/2} > e^{-1/n^{(1+\delta)}}.$$

Letting $\delta = 1$, we have

$$1 - e^{-2/n^2\bar{c}\log_2\bar{c}} > (1 - p)^{L^2/2\bar{c}}.$$

It is easy to see that if $x > 0$ and x is sufficiently small, then $1 - e^{-x} > x/2$, so again it suffices to have

$$\frac{1}{n^2\bar{c}\log_2\bar{c}} \geq (1 - p)^{L^2/2\bar{c}} \text{ or,}$$

$$\log_e\left(\frac{1}{n^2\bar{c}\log_2\bar{c}}\right) \geq \frac{L^2}{2\bar{c}}\log_e(1 - p).$$

For $0 < p < 1/2$, due to the concavity of the function $\log_e(1 - p)$,

$$\log_e(1 - p) \geq 2p\log_e 1/2.$$

(Note that both sides of the inequality are negative). Hence,

$$\frac{L^2 p}{\bar{c}}\log_e\frac{1}{2} \leq \log_e\left(\frac{1}{n^2\bar{c}\log_2\bar{c}}\right),$$

and L satisfying

$$L \geq \sqrt{\frac{\log_e(n^2\bar{c}\log\bar{c})}{\log_e 2}\frac{\bar{c}}{p}}$$

will allow completion of the *linking* successfully almost surely. Since $\bar{c} < n$ and $\bar{c} \leq 2pn$ almost surely, a choice of $L \geq 3\sqrt{n\log_e n}$ will generate the successful completion as required. For $1/2 \leq p < 1$, we should have,

$$L^2 \geq \frac{2\bar{c}\log_e(1/(n^2\bar{c}\log_2\bar{c}))}{\log_e(1-p)} = \frac{2\bar{c}\log_e(n^2\bar{c}\log\bar{c})}{\log_e(1/1-p)}.$$

But the last term is less than or equal to,

$$\frac{2\bar{c}\log_e(n^2\bar{c}\log_2\bar{c})}{\log_e 2} \leq 4n\log_e(n^3\log_2 n) \leq 4n\log_e n^4.$$

Hence, for all p it suffices to choose $L \geq 4\sqrt{n\log_e n}$.

Corollary 1. *The total number of vertices checked by the parallel algorithm is* $O(\sqrt{n\log n}(\log c_n))$.

Proof. The algorithm checks vertices either during the phase of "doubling" until there are at least L leaves on each side, or subsequently, after half of the trees get linked. In the former phase, the total number of vertices in the trees is $O(L)$ because these are essentially binary trees. In the latter phase, the number of vertices in the remaining trees is doubled $\log_2 c_n$ times. In each such layer we check $O(L)$ vertices. Hence, the total number of vertices checked is $O(L\log_2 c_n)$.

This corollary justifies the implicit assumption that at any iteration at least $\Theta(n)$ vertices remain in the random graph.

The *linking* procedure may be applied, depending on the density of the graph, by either checking all neighbors of each leaf until a link is found, or alternatively checking the edge slots in the bipartition induced by the two sets of leaves until a link is found. In the following lemma, it is proved that checking all edge slots is preferred if c_n is large and checking all neighbors, if c_n is small.

Lemma 2. *Given two sets of* L *leaves in* $\mathcal{G}_{n,p}$, *the expected number of steps required to find a link is:*

a. $O(n/L)$ *when checking all neighbors of one vertex at a time until a link is found; or*
b. $O(n/c_n)$ *when checking all edge slots in the bipartition.*

For $L = \Theta(\sqrt{n\log n})$, *checking all edge slots is at least as fast as checking all neighbors when* c_n *is* $\Omega(\sqrt{n\log n})$.

Proof. a. All vertices in the graph are equally likely on the adjacency list of the L vertices considered, i.e., neighbors of the set of L leaves. The probability of a given node on the opposite side to be among the neighbors is thus L/n. Therefore, $O(n/L)$ of the neighbors are to be looked up before a link is found.

b. Since the probability that an edge occupies an edge slot in the graph is p, the expected number of checkings until an edge is found is at most $1/p$. Hence, it takes $O(n/c_n)$ steps almost surely.

To find the break-even point between inspecting all neighbors and all edge slots, we compare the expected values of both search methods. Checking the neighbors is more efficient when $n/L = O(n/c_n)$. This holds for $c_n = O(L)$, or equivalently, $c_n = O\sqrt{n\log n})$, thus completing the proof.

Corollary 2. *The total expected number of steps required by the parallel procedure is*

$$\begin{cases} O(\sqrt{n\log n}\log c_n) + O(n) & \text{for } c_n = O(\sqrt{n\log n}) \\ O(\sqrt{n\log n}\log c_n) + O(\sqrt{n/(\log n)}\,c_n) & \text{for } c_n = \Omega(\sqrt{n\log n}) \end{cases}$$

which is $O(n)$ *for all* c_n.

Proof. The work of the algorithm is in the "doubling" and "linking" phases. If $c_n = O(\sqrt{n\log n})$, by Lemma 2, it takes $\Theta((n/L)\bar{c})$ for the linking of all

$\bar{c}$ trees. Now, $(n/L)\bar{c} = O((n/L)c_n)$, and

$$\frac{n}{L} c_n = O\left(\frac{n\sqrt{n \log n}}{\sqrt{n \log n}}\right) = O(n).$$

The number of steps taken for doubling is $O(L + L \log c_n)$, since the number of vertices checked for doubling is $O(L)$ during the "growing" phase and $O(L \log c_n)$ during the "linking" phase. Hence, the entire doubling phase takes $O(L \log c_n)$, which is $O(\sqrt{n \log n} \log c_n)$. This is dominated by $O(n)$. Thus the total number of steps is $O(n)$.

For $c_n = \Omega(\sqrt{n \log n})$, the doubling's running time remains the same, and for the linking the stated running time follows from Lemma 2.

3. MATCHING

The simplest way to link $N(s)$ and $N(t)$ is by pairing them up directly. This amounts to finding a bipartite matching the bipartite graph induced by $N(s)$ and $N(t)$. Erdös and Rênyi proved that for $p \geq (\log_e n + w_n)/n$, where $\lim_{n\to\infty} w_n = \infty$, almost all random graphs $\mathcal{G}_{n,p}$ have a perfect matching. Such perfect matching exists with probability $e^{-e^{-w_n}}$. The same applies to $\mathcal{B}_{n,n,p}$, the random bipartite graph with n vertices on each side such that an edge exists with probability p. In our case, we seek a perfect matching in a bipartite graph on $2\bar{c}$ vertices $\mathcal{B}_{\bar{c},\bar{c},p}$, hence, p has to satisfy $p \geq (\log_e \bar{c} + w_n)/\bar{c}$ in order to have a perfect matching between $N(s)$ and $N(t)$. Since $\bar{c}$ is $O(np)$ in the random graph $\mathcal{G}_{n,p}$ almost surely, it is sufficient to have $p = c_n/n$ satisfying

$$p \geq \frac{2 \log_e c_n}{c_n}.$$

Any $c_n \geq 2\sqrt{n \log_e n}$ satisfies that inequality. Thus, the bipartite graph $B(N(s), N(t), E)$ is a random bipartite graph $\mathcal{B}_{\bar{c},\bar{c},p}$ that contains a perfect matching almost surely.

In a random bipartite graph $\mathcal{B}_{n,n,p}$ with $p \geq (\alpha \log_e n)/n$, for some constant $\alpha > 2$, the *greedy matching* algorithm applies (Goldschmidt and Hochbaum 1990b) with an expected running time of $O(n + n \log 1/p)$. That greedy matching algorithm has the same underlying idea as the linking procedure. That is, each node is linked with the first available neighbor found. The failures are then matched together with a set of uninspected vertices, that we set aside for that purpose, using Angluin and Valiant's (1979) algorithm. Applying it to the bipartite graph $\mathcal{B}_{\bar{c},\bar{c},p}$, with $\bar{c} \geq \alpha\sqrt{n} \log_e n$, the expected running time of the greedy matching algorithm is

$$O(\bar{c} + \bar{c} \log_e(1/p)),$$

which is $O(np + np \log_e 1/p)$. Since $0 < p \leq 1$, we can write

$$np \log_e \frac{1}{p} = n \frac{\log_e(1/p)}{(1/p)},$$

where

$$\lim_{n\to\infty} \frac{\log_e(1/p)}{1/p} = 0 \quad \text{if } p \to 0,$$

and is a constant when p is a constant. The running time of the mtching is $o(n)$ for $p = o(1)$, and $O(n)$ for $p = O(1)$.

4. SUMMARY OF THE ALGORITHM FOR THE MAXIMUM-FLOW PROBLEM

The most efficient implementation of the algorithm to solve the maximum-flow problem uses the *parallel* algorithm up to the range of densities where the matching works. At that range, it is more efficient to use the greedy matching algorithm. For

$$o(n) > c_n > \alpha\sqrt{n \log_e n}$$

for some $\alpha > 0$ the matching algorithm runs in $o(n)$ time, and hence is faster than the *parallel* procedure. The recommendation is to run the *parallel* procedure for $c_n \leq \alpha\sqrt{n} \log_e n$, and the matching for $c_n \geq \alpha\sqrt{n} \log_e n$. The running time of this combined procedure follows from Corollary 2 and the running time of the matching stated in the previous section,

$$\begin{cases} O\left(\sqrt{\frac{n}{\log n}} c_n\right) & \text{for } \log_e n + w_n \leq c_n \leq \alpha\sqrt{n \log_e n} \\ O\left(c_n\left(1 + \log\frac{n}{c_n}\right)\right) & \text{for } \alpha\sqrt{n \log_e n} \leq c_n \leq n. \end{cases}$$

This running time is $o(n)$ except for the functional values of c_n, $c_n = \Theta(\sqrt{n \log n})$ or $c_n = \Theta(n)$, where it is $O(n)$.

In principle, the density of the random graph is not given along with a realization of the graph. There are several ways to circumvent this difficulty. One is to use only the *parallel* procedure for any graph realization. The running time is less than $\Theta(c_n n)$—the expected number of edges and input length—and hence, is still sublinear. Note that the *parallel* procedure includes the linking that depends on the density as well. Here we can alternate between a single operation of checking all neighbors and a single step of checking all edge slots, and terminate with the first one to

succeed. This will at most double the running time. If we want the running time not to exceed $O(n)$, we can use the same idea for the *parallel* and matching algorithms—alternately running one step of each algorithm appropriate for a different density range and terminate with the first one to succeed. A third approach is to sample the degree of a few vertices to guess the density of the graph.

Both algorithms leave most of the nodes in $V - \{N(s) \cup N(t)\}$ uninspected.

5. THE SEQUENTIAL ALGORITHM

The *sequential* algorithm is used to solve the communication problem. Although this algorithm could also be used to solve the maximum-flow problem, it is less efficient and inspects more vertices than the *parallel* algorithm.

For the communication problem a set of $\bar{c}$ terminals to be paired is given, $(s_1, t_1), \ldots, (s_{\bar{c}}, t_{\bar{c}})$. To find vertex-disjoint paths connecting each pair (s_i, t_i), we grow a tree from each vertex by recursively finding all neighbors of each of the leaves and alternating from the s-side to the t-side. More precisely, consider a pair of vertices s', t', one in $N(s)$ and one in $N(t)$. The algorithm searches to construct a path between s' and t'. These two vertices are initialized to be two singleton sets of the s-leaves and the t-leaves, respectively. The search now alternates between the s-side and the t-side. It finds *all* neighbors of the current leaves, inspecting them one at a time, in a random order, and labeling them s' or t', depending on whether they were reached from the s- or t-side. All these neighbors become the new set of leaves. The search now adds a new layer to the tree on the opposite side using the same procedure. The search alternates until one of the identified neighbors of a leaf on one side belongs to the tree created on the opposite side. If a neighbor of a vertex on the s-side is also on the s-side, then this link is ignored.

Once the number of leaves on each side exceeds $\sqrt{((2+\delta)n \log_e n)/c_n}$ for any $\delta > 0$, it is guaranteed almost surely that the two sets of leaves are directly linked (see Lemma 3). At this point, the *linking* algorithm is introduced. For $c_n = O(\sqrt[3]{n} \log n)$, the *linking* works by continuing the same procedure, i.e., checking all neighbors until one neighbor on the opposite side is identified. For larger values of c_n, $c_n = \Omega(\sqrt[3]{n} \log n)$, *the linking* procedure checks the edge slots in the bipartite graph induced by the two sets of leaves.

In case there are no more neighbors to create the next layer, the procedure terminates and reports a failure. Otherwise, a path is eventually identified between the selected pair of vertices. All the vertices in the two trees rooted at s' and t' are then deleted from the graph. The process is repeated for another pair of vertices until either $N(t)$ and $N(s)$ have been exhausted or a failure has occurred.

6. PROBABILISTIC ANALYSIS OF THE SEQUENTIAL ALGORITHM

At each iteration of the *sequential* algorithm, we have two trees and two sets of leaves. We study the number of leaves L required on each side sufficient to have both trees linked, almost surely.

Lemma 3

a. *When two disjoint sets of nodes in a random graph* $\mathcal{G}_{n,p}$ *have at least* $\sqrt{((1+\delta)n \log_e n)/c_n}$ *nodes each, for some* $\delta > 0$, *there is a link between the two sets, almost surely.*

b. *For up to* $\bar{c}$ *pairs of the sets of* L *nodes each to be linked almost surely it suffices that* $L > \sqrt{((2+\delta)n \log_e n)/c_n}$.

Proof

a. Pr[two sets of leaves of size L each are not connected] $= (1 - c_n/n)^{L^2}$.

For $c_n/n \to 0$, this probability goes asymptotically to $e^{-(c_n/n)L^2}$. To have $e^{-(c_n/n)L^2}$ going to zero at least as fast as $1/n^{1+\delta}$ for some $\delta > 0$ (so that almost sure convergence is guaranteed), we have $e^{-(c_n/n)L^2} < n^{-(1+\delta)}$. That implies that $L > \sqrt{((1+\delta)n \log_e n)/c_n}$. When $c_n/n \not\to 0$, it is a constant between zero and one, and then $L > \sqrt{2/p \log_e n}$ suffices. Note, however, that this case will not be in the range of the *sequential* algorithm, so it is of no interest as far as the properties of this algorithm are concerned.

b. Pr[at least one of the pairs of $\bar{c}$ sets is not linked]

$$\leq \sum_{i=1}^{\bar{c}} \Pr[\text{set } i \text{ is not linked}]$$

$$= \bar{c}(1 - c_n/n)^{L_2}$$

$$\leq \bar{c}e^{(-c_n/n)L^2} \leq ne^{(-c_n/n)L^2} \leq n^{-(1+\delta)}.$$

Since the expected value of $\bar{c} = \min\{|N(s)|, |N(t)|\}$ is less than or equal to c_n, it is $O(c_n)$ almost surely. The total number of vertices in each tree with L leaves is no more than $2L$ (at each layer the number is expected to go up by a factor of c_n compared to the previous layer). Hence, the total number of vertices examined and deleted in the process of establishing

the $\bar{c}$ paths is

$$O\left(c_n \sqrt{\frac{n \log n}{c_n}}\right) = O(\sqrt{c_n n \log n})$$

almost surely. Lemma 4 follows from these arguments.

Lemma 4. *For $c_n = O(n/\log n)$, the total number of vertices used by the sequential procedure is $o(n)$ almost surely. If $c_n \leq n/(2(2+\delta)\log_e n)$ for some constant $\delta > 0$, no more than $n/2$ vertices are used.*

This lemma justifies the assumption that the number of deleted vertices throughout the procedure is a diminishing fraction of n, or that at least $\frac{1}{2}n$ vertices remain at the termination of the algorithm. The latter fact is sufficient for our purposes because all the proofs assume the availability of $\Theta(n)$ undeleted vertices in the graph.

In the next lemma we compare the alternative methods for linking a pair of trees.

Lemma 5. *Given two sets of L leaves in $\mathcal{G}_{n,p}$, the expected number of steps required to find a link is:*

a. *$O(n/L)$ when checking all neighbors of one vertex at a time until a link is found; or*
b. *$O(n/c_n)$ when checking all edge slots in the bipartition.*

For $L = \sqrt{((2+\delta)n \log_e n)/c_n}$ for some $\delta > 0$, checking all edge slots is at least as fast as checking all neighbors for $c_n = \Omega(\sqrt[3]{n \log n})$, i.e., if $c_n = O(\sqrt[3]{n \log n})$ checking all neighbors is preferred, which takes $O(n^{2/3}/(\log n)^{1/3})$.

Proof. The proofs of parts a and b are identical to the respective proofs in Lemma 2. To find the break-even point between inspecting all neighbors and all edge slots, we compare the expected time required for the success of both search methods. Checking all neighbors is more efficient when $n/L = O(n/c_n)$. This holds for $c_n = O(L)$, or equivalently, $c_n^3 = O((2+\delta)n \log n)$. So, for $c_n = O(\sqrt[3]{n \log n})$, this method is indeed preferred with running time

$$O\left(\frac{n}{L}\right) = O\left(\sqrt{\frac{nc_n}{\log n}}\right) = O\left(\sqrt{\frac{n(n \log n)^{1/3}}{\log n}}\right)$$
$$= O\left(\frac{n^{2/3}}{(\log n)^{1/3}}\right).$$

Similarly, for the range where n/c_n is the smaller value, a simple calculation shows that the number of checks is $O(n^{2/3}/(\log n)^{1/3})$.

Since the number of vertices remaining in the graph is $\Theta(n)$, when $c_n \leq n/(2(2+\delta)\log_e n)$ (Lemma 4), this is also the range of densities assumed in the following corollary.

Corollary 3. *An edge that connects the two sets of at least $L = \sqrt{((2+\delta)n \log_e n/)c_n}$ leaves each can be found in the expected number of steps, $O((n \log n)^{2/3}$.*

The total running time of the *sequential* algorithm derives from the phase in which the necessary number of leaves is identified, followed by the work required to link the two sets of leaves. As proved in Lemma 4, the bidirectional search takes $O(\sqrt{nc_n \log_e n})$ steps for generating the leaves. The linking phase takes $O(\sqrt{nc_n^3/\log n})$ steps for $c_n = O(\sqrt[3]{n \log n})$ and $O(n)$ for $c_n = \Omega(\sqrt[3]{n \log n})$. Since the running time is $o(n)$ for the range

$$\log_e n + w_n \leq c_n = o\left(\frac{n}{\log n}\right),$$

the dominating complexity is that of the linking phase.

Corollary 4. *The expected running time of the sequential procedure is*

$$\begin{cases} O\left(\sqrt{\dfrac{nc_n^3}{\log n}}\right) & \textit{for} \log_e n + w_n \leq c_n = O\sqrt[3]{n \log n}) \\ O(n) & \textit{for}\, \Omega(\sqrt[3]{n \log n}) = c_n = o\left(\dfrac{n}{\log n}\right). \end{cases}$$

Note that the complexity of the *sequential* procedure is sublinear in the length of the input, since it is at most linear in the number of nodes n, but the expected number of edges that determines the input length is $\Theta(n^2 p)$, which is $\Omega(n \log n)$. Also note that the total number of steps is almost surely bounded by the value specified in Corollary 4, multiplied by a factor of $\log_e n$.

The expected length of the paths found in the *sequential* procedure are easy to compute. Whenever all neighbors are found, the number of leaves is expected to grow by a factor of $O(c_n)$. The length of the path from each side until linkage is therefore expected to be $\log_{c_n} L$, so the expected length of each such path is

$$2 \log_{c_n} \sqrt{\frac{(2+\delta)n \log_e n}{c_n}},$$

which is $O(\log n)$.

7. THE FEASIBILITY OF THE COMMUNICATION AND THE VERTEX-DISJOINT PATHS PROBLEMS

Both the communication and the vertex-disjoint paths problems may be presented independently of the maximum-flow problem. That is, rather than finding paths between two sets of vertices, one which is the neighbor of a source vertex, and the other of a sink vertex, the problem is posed for two arbitrary sets of vertices. This is a generalization, since in the context of maximum flow, the two sets to be linked are not of arbitrary size, but rather of random size with an expected value of $O(c_n)$. Given two arbitrary sets of vertices in a random graph, we investigate how large they can be while the algorithms still succeed in providing a solution, i.e., a collection of vertex-disjoint paths between the two sets. Here the certificate of optimality is simply the description of the vertex-disjoint paths between all pairs in the two sets.

For the communication problem we are given two sets S, T of vertices in a random graph and a specified pairing $(s_1, t_1), \ldots, (s_r, t_r)$. The question is how large r can be so that the *sequential* algorithm solves the problem and links all pairs via vertex-disjoint paths almost surely. A failure occurs if the algorithm runs out of vertices in the process of a bidirectional tree search. For each pair to be linked, we need $2L$ vertices on each side with $L \leq \sqrt{((2+\delta)n \log_e n)/c_n}$ (Lemma 3). For all r pairs to be linked, we need $2r \cdot 2L \leq \frac{1}{2}n$, so there are always at least $\frac{1}{2}n$ vertices which remain. Hence, the largest value of r is determined as a function of c_n:

$$r \leq \frac{1}{16} \sqrt{\frac{n}{\log_e n}} \sqrt{c_n}.$$

Since $c_n > \log_e n$, r can be set to be $O(\sqrt{n})$ regardless of what c_n is, and can go up to $O(n/\sqrt{\log n})$ pairs on dense graphs. Hence, the communication problem can be solved almost surely for sets of size $O(\sqrt{n})$, and for larger sets on denser graphs.

For the vertex-disjoint paths problem, we are given two sets of r vertices each in a random graph. The question is how many vertex-disjoint paths can be found between them. Here the *parallel* procedure is used. In the *parallel* procedure, $\log_2 r$ stages are needed to link r pairs along vertex-disjoint paths. Each stage requires $2\sqrt{n} \log_e n$ leaves. The total number of vertices used is no more than

$$2 \cdot 2\sqrt{n} \log_e n(1 + \log_2 r).$$

For this number not to exceed $\frac{1}{2}n$ it is sufficient for r to be less than $2^{O(\sqrt{n}/\log n)}$. Since the $2r$ vertices are part of the graph, their number may not exceed n. Hence, it is possible to link *any* number of pairs along vertex-disjoint paths in a connected random graph.

8. PARALLEL AND DISTRIBUTED IMPLEMENTATIONS

Both the *sequential* and matching algorithms lend themselves readily to a parallel and distributed implementation. The *sequential* algorithm has a fast and simple distributed implementation because the only processing that is performed by this algorithm involves specific nodes and their immediate neighbors. Let there be $|E|$ processors, one at each edge. There are r specified pairs, $(s_1, t_1), (s_2, t_2), \ldots, (s_r, t_r)$. In the procedure, each node will either get no label or two labels. The first of the two labels is either ls_i or s_i(lt_j or t_j, respectively), where s_i designates the neighbor of s from which the current node is reachable and included in the tree rooted at that s_i. The label is ls_i when the node is currently a leaf of such a tree rooted at s_i. The second label is the index of the node that is the immediate predecessor of the current node on the path from s_i. The purpose of that second label is to make possible the tracing of the identified path. The labeling of t vertices is analogous.

Let L be the number of leaves required for linking in the *sequential* procedure, as established in Lemma 3. The distributed algorithm will consist of $\lceil \log_2 L \rceil$ synchronized pulses. A pulse is identical for all processors. It uses as input the sets $N(s)$ and $N(t)$, the value of $\bar{c}$, and the pairing.

We define an array LINK, initially empty (all zeros), of length $2\bar{c}$, with each of $\bar{c}$ positions containing two words for the two endpoints of the edge which is the linking edge of the pair of trees. Let $(u, v) \in E$.

pulse (u, v)
  If exactly one end point (say u) is labeled(ls_i, z) **do**
    label v, (ls_i, u)
    relabel u, (s_i, z)
  else, end
  If both endpoints are labeled and
  one, (say u), has a leaf labeled (ls_i, z) **do**
    if v is labeled (lt_i, w) or
    (t_i, w) and LINK$(i) = 0$, set LINK$(i) = (u, v)$
    else, end
  else, end.

For simultaneous WRITE, any rule such as random write, first write, last write, or lowest index write will be appropriate. Simultaneous READ is allowed.

This procedure, once terminated, establishes all required linkages under the same conditions as the *sequential* procedure. The paths created are of length $2\lceil \log_2 L \rceil$ at most. Note that the expected length of a path is $2\lceil \log_c L \rceil$. Since we do not count the number of leaves in the distributed procedure, $\lceil \log_2 L \rceil$ layers are sufficient to guarantee almost surely, with at least L leaves for each subtree (in fact, $O(\log_{c_n} L)$ will suffice).

The path identification can be executed in logarithmic time in the length of the path (i.e., $\log_2 \log_2 L$) using the procedure in which the node index is communicated to nodes that are at a distance of l edges away with the value of l doubling at each iteration (see, e.g., Cole and Vishkin 1986). Actually, such a sophisticated procedure is not necessary in our case. Even a strightforward procedure that sends back the labels of the path, one layer at a time, will take at most $\lceil \log_2 L \rceil$ stages which is the maximum number of layers. Using this straightforward procedure increases the running time $\log_2 L$ by a factor of two at most.

This distributed procedure runs faster than $\Theta(\log n)$ since L is $o(n)$. It is applicable for c_n up to $O(n/\log n)$ (see Lemma 4).

As a version of the *sequential* procedure, the distributed algorithm works for $c_n = O(n/\log n)$. Beyond that range, it might run out of vertices. For the range $c_n = \Omega(n/\log n)$, there is almost surely a perfect matching between the two sets of vertices, $N(s)$ and $N(t)$. (Following the analysis of Section 3, such matching already exists for $c_n = \Omega(\sqrt{n \log n})$.) The parallel and distributed version we suggest for this range of densities involves a procedure similar to the greedy matching algorithm of Goldschmidt and Hochbaum (1990b). Namely, after setting aside a small set of vertices on each side of the bipartition, we find a greedy matching which is a maximal matching. Vertices that were not successfully matched are then matched with the set of vertices which was set aside—of size $(1/p)\log_e(1/p)$, using any perfect matching algorithm. We propose to execute the maximal matching using Luby's NC^2 parallel (and distributed) algorithm (Luby 1985). That will take $O(\log^2 r)$ steps. The matching can then be completed by using Angluin and Valiant's algorithm. It runs in $O(n \log n)$ expected time on a graph with n nodes. On the remaining graph, there are $O(\log n \log \log n)$ nodes, so the expected running time is $O(\log^2 r)$. Hence, the overall procedure has a distributed and parallel implementation which is faster than $O(\log^2 n)$.

9. EMPIRICAL STUDY

The algorithms were implemented on an IBM/RT using APL. Problems with up to 20,000 vertices were tested and none took more than 30 seconds. For larger and denser graphs, the only difficulty is storing the graph. The number of steps is still very small, but the access to secondary memory to look up an edge or an adjacency list becomes the dominant factor.

To separate the actual running time of the algorithm from the factor of memory lookup, we provide, in addition to the *running time of the program* (CPU + disk accesses), the *number of edge slots and nodes looked up* by the algorithm. This measures the actual efficiency of the algorithm.

Table I illustrates the results of the runs on graphs with 5,000 vertices with various densities. For each density (designated as a function of n), there are five graphs tested. On each such graph, we ran both the *parallel* and the *sequential* procedures. The values given are the *average* of the five runs. The proven optimal solution was attained for all graphs tested.

As the graph density increases there are more paths (or, equivalently, more flow) to identify. This naturally

Table I
Random Graph on 5,000 Vertices

		Sequential Neighbor Linking			Sequential Slots Linking			Parallel		
c_n	Average Flow	Number of Vertices	Steps/Phase	Sec. Run	Number of Vertices	Steps/Phase	Sec. Run	Number of Vertices	Steps/Phase	Sec. Run
$\log_e$	7.1	1,134	159	4	1,014	282	43	650	62	16
$\frac{1}{2}\sqrt[3]{n \log_e n}$	14.8	1,569	142	9	502	187	92	931	40	24
$\sqrt[3]{n \log_e n}$	31	2,834	157	16	2,320	181	101	444	35	12
$2\sqrt[3]{n \log_e n}$	65	4,599	240	32	2,873	142	185	529	39	22
$\sqrt{n \log_e n}$	223	(1) 5,000	352	81	4,528	187	210	675	58	30
			½ fail							

increases the time requirement. For this reason, the relevant running time or number of steps is specified *per path*.

Table I also provides the data for both linking procedures. As is evident from the table, and graphed for the sequential algorithm in Figure 1, the breakpoint between the neighbor linking and the slots linking procedures conforms to the theoretical result in Lemma 5. That is, the breakpoint for the sequential algorithm occurs for the density of $\sqrt[3]{n \log_e n}$.

As it turns out, the number of steps per path generated is close to a constant (though the sample size is too small to determine that with high confidence). An illustration of that is given in Figure 2. For the *sequential* procedure we take the running time from the neighbor linking procedure, for densities up to $\sqrt[3]{n \log_e n}$, and then we use the running time from the slots linking procedure, which is the lower envelope of the two running time functions. Figure 2 also illustrates that the *parallel* algorithm is much more efficient and should be used whenever a specific pairing of vertices is not required.

Figure 3 illustrates another advantage of the *parallel* algorithm: The total number of vertices remains essentially constant as the graph density goes up, whereas it grows for the *sequential* algorithm. The rate of growth of the number of vertices used by the *sequential* algorithm goes up, according to the analysis, at a rate of the square root of the density (see the statement preceding Lemma 4), where for the *parallel* algorithm Corollary 1 indicates that the rate of growth

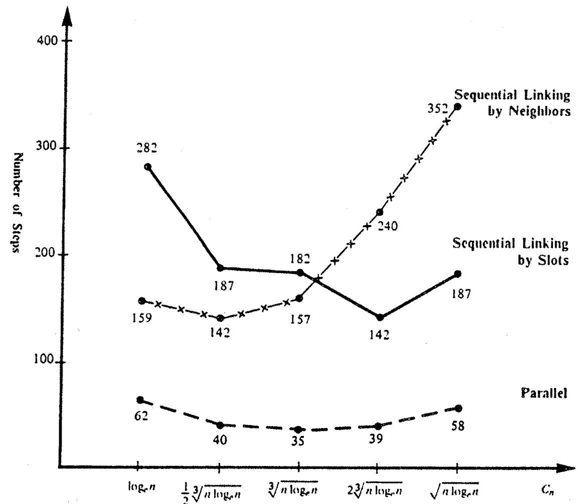

Figure 2. Number of steps versus density graphs having 5,000 vertices.

is at most the logarithm of the density. These are both conservative measures according to our empirical results.

Table II gives the results of running the *sequential* algorithm on graphs on n vertices, with n varying from 500 to 20,000. All these graphs have density determined by $c_n = \log_e n$. Higher densities were not tested due to data storage difficulties. Yet the theoretical and empirical results indicate that in terms of the number of steps the performance required to achieve the opitmal solution would be similarly small. The running times reported in Table II are larger than those

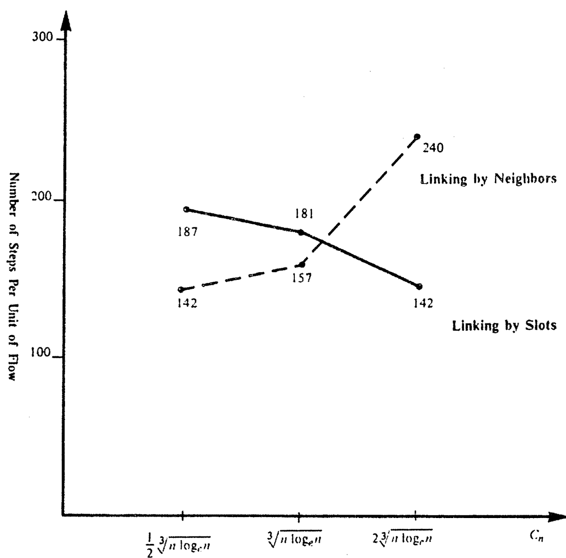

Figure 1. Breakpoint for linking procedures graphs having 5,000 vertices.

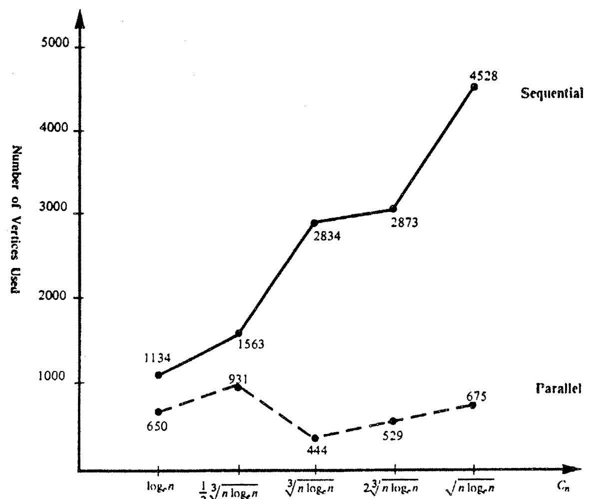

Figure 3. Number of vertices used versus density graphs having 5,000 vertices.

Table II
Random Graphs With $c_n = \log_e n$

Number of Vertices	Flow	Percent Vertices	Steps/ Phase	Msec./ Phase
500	5.2	34.73	53.06	2,894
1,000	3.4	18.3	73.90	4,160
2,000	5	18.46	101.93	4,837
5,000	7.1	22.7	159	5,122
10,000	6.8	13.63	254.80	2,155
20,000	7.2	9.2	303.18	3,092

reported in Table I because the tests in Table II were run on an IBM/AT instead of an IBM/RT.

To conclude, the empirical results verify the theoretical ones. They indicate that the theoretical bounds are conservative and the actual performance is substantially better. It is important to note that all the functions given with the big O notation have very small constant coefficients, so this notation does not hide horrendously large values. The algorithms work well for the entire range from very small to very large graphs. Due to the range of applicability and the fact that the algorithms here do not even read the entire input, they constitute a substantial improvement over all known algorithms that require at least the reading of all the input.

NOTES

Throughout, we use the o, O, Ω and Θ notation. A function $f(n)$ is said to be $o(g(n))$ if $\limsup_{n\to\infty} f(n)/g(n) = 0$. It is said to be $O(g(n))$ if there is a constant c such that $f(n) \leqslant cg(n)$ for all but finitely n, while $f(n)$ is said to be $\Omega(g(n))$ if there is a constant c such that $f(n) \geqslant cg(n)$ for all but finitely n. Finally, $f(n)$ is said to be $\Theta(g(n))$ if there exist $c_1 \geqslant c_2 > 0$ such that $c_1 g(n) \geqslant f(n) \geqslant c_2 g(n)$ for all large enough n.

ACKNOWLEDGMENT

I wish to thank the referees for the detailed comments and editorial suggestions. These considerably improved the presentation. This research was supported in part by National Science Foundation grant ECS-85-01988, and by Office of Naval Research grants N00014-88-K-0377 and N00014-91-J-1241.

REFERENCES

ANGLUIN, D., AND L. G. VALIANT. 1979. Probabilistic Algorithms for Hamiltonian Circuits and Matchings. *J. Comput. and Syst. Sci.* **18,** 155–190.

CHERIYAN, J., T. HAGERUP AND K. MEHLHORN. 1990. Can a Maximum Flow Be Computed in $o(mn)$ Time? *ICALP,* 118–123.

COLE, R., AND U. VISHKIN. 1986. Deterministic Coin Tossing With Applications to Optimal Parallel List Ranking. *Inform. Control* **70,** 32–53.

ERDÖS, P., AND A. RÉNYI. 1960. On the Evolution of Random Graphs. *Publ. Math. Inst. Hung. Acad. Sci.* **5,** 17–61.

EVEN, S., AND R. E. TARJAN. 1975. Network Flow and Testing Graph Connectivity. *J. SIAM Comput.* **4,** 507–518.

GOLDBERG, A. V., AND R. E. TARJAN. 1988. A New Approach to the Maximum Flow Problem. *J. ACM* **35,** 921–940.

GOLDSCHMIDT, O., AND D. S. HOCHBAUM. 1990a. Asymptotically Optimal Linear Algorithm for the Minimum k-Cut in a Random Graph. *SIAM J. Discr. Math.* **3,** 58–73.

GOLDSCHMIDT, O., AND D. S. HOCHBAUM. 1990b. A Fast Perfect Matching Algorithm in Random Graphs. *SIAM J. Discr. Math.* **3,** 48–57.

GRIMMETT, G. R., AND A. D. R. WELSCH. 1978. Flow in Networks With Random Capacities. *Stochastics* **7,** 205–229.

HASSIN, R., AND E. ZEMEL. 1988. Probabilistic Analysis of the Capacitated Transportation Problem. *Math. Opns. Res.* **13,** 80–89.

KARP, R. M. 1979. The Probabilistic Analysis of Combinatorial Optimizations Algorithms. Presented at the 10th International Symposium on Mathematical Programming, Montreal.

KARP, R. M., R. MOTWANI AND N. NISAN. 1987. Probabilistic Analysis of Network Flow Algorithms. Report No. UCB/CSD 87/392, University of California, Berkeley.

KING, V., S. RAO AND R. TARJAN. 1991. A Faster Deterministic Maximum Flow Algorithm. Extended Abstract.

LUBY, M. 1985. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *17th Annual ACM STOC* 1–10.

LYNCH, J. F. 1975. The Equivalence of Theorem Proving and the Interconnection Problem. *ACM SIGDA Newsletter* **5,** 3.

MOTWANI, R. 1990. Expanding Graphs and the Average-Case Analysis of Algorithms for Matchings and Related Problems. *21st Annual ACM STOC,* 550–561.