The Origin-Destination Shortest Path Problem

March 1993 WP # 3546-93

Muralidharan S. Kodialam*
James B. Orlin**

* AT&T Bell Laboratories
Holmdel, NJ 07733

** Sloan School of Management
Massachusetts Institute Technology
Cambridge, MA 02139

The Origin-Destination Shortest Path Problem

Muralidharan S. Kodialam
AT &T Bell Laboratories
Holmdel, NJ 07733

James B. Orlin
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02139

November 16, 1992

The Origin-Destination Shortest Path Problem

Abstract

In this paper we consider the Origin-Destination (O-D) shortest path problem. We define the O-D shortest path problem as follows: We are given the set of nodes and edges in a network. Associated with each edge is a weight. In addition, we are given two subsets of the node set. One set is called the set of *origin nodes* and the other is called the set of *destination nodes*. The objective of the O-D shortest path problem is to determine the shortest path between every node in the set of origin nodes and every node in the set of destination nodes. This problem generalizes several traditional shortest path problems and has applications in transportation and communication networks. We develop two algorithms for this problem and analyse the average case running time of these algorithms.

1 Introduction

Shortest path problems are arguably the most fundamental and also the most commonly encountered problems in the study of transportation and communication networks. The repeated determination of shortest paths in large networks often forms the core of many transportation planning and utilization models. Deo and Pang [2] survey a large number of algorithms and applications of shortest path problems.

At the broadest level, shortest path algorithms can be classified by three taxonomic categories: The first is the type of shortest path problem being solved. The most common being the single source to all nodes shortest path problem and the all pairs shortest path problem. The second is the classification based on the input network: whether the network is sparse or dense, planar or non planar, positive or general edge weights, grid or general structure etc. The third classification is based on the solution methodology. For example, some methods use labelling and others use matrix multiplication to determine the shortest paths. The most common shortest path problems addressed in the literature are the single source and the all pairs shortest path problems. In the single source shortest path problem the aim is to find the shortest path from a given node to all other nodes in a given graph. In the all pairs shortest path problem, the objective is to determine the shortest path between every pair of nodes in the network.

We define a problem that is intermediate between the single source and the all pairs shortest path problems. We refer to this problem as the O-D shortest path problem. We are given a set O of n_O nodes called the *origin nodes* and a set D of n_D nodes called the *destination nodes*. The problem is to find the shortest path from every origin node to every destination node. This problem is interesting from both theoretical and practical viewpoints. It is interesting in a theoretical sense because it generalizes several classical shortest path problems and all these problems are studied in a common framework. From a practical viewpoint, the O-D shortest path problem has numerous applications in transportation and communication networks [10]. For example, road networks sometimes have a specified subset of nodes called the *centroid*. A centroid in a road network is defined as the set of nodes in which the traffic originates of terminates [11]. Typically, each transport zone has one centroid. The input for many transportation problems consists of the shortest path distances between all pairs of nodes in the centroid, which in turn is a special case of the O-D shortest path problem. A related application of the O-D shortest path problem is the generation of data for combinatorial optimization problems on a network. Data for the travelling salesman problem and vehicle routing problems often comes from solving O-D shortest path problems. For example, in the well known 48 city travelling salesman problem between the capitals of the states in the continental United States, the distances between the cities can be obtained by letting O and D to be the set of 48 cities, and by solving an O-D shortest path problem. This data may then be used as the input the TSP algorithm. The O-D shortest problem can also be used to generate data for transportation problems on graphs where the

transportation cost between two nodes is approximately the length of the shortest path between these nodes. Numerous problems in combinatorial optimization use the shortest path length as an approximation to the transportation cost.

An obvious approach for solving the O-D shortest path problem is to solve a single source problem from each source node. While there is no other approach that is known to be better in terms of worst-case analysis, there is strong evidence that this approach is not as good on the average.

For the purpose of performing average case analysis, we first need to specify the class of problems over which we will be averaging the running time of the algorithm. We assume that the input graph has arc lengths that are independent, identically distributed random variables that satisfy some weak assumptions. The details of the probabilistic model is given in Section 2.1.

The average case analysis of shortest path algorithms originated in the study of random graphs and the study of graph properties when the arc lengths are random variables. Pioneering work in the area of random graphs was done by Erdos and Renyi [3]. Using results from the study of probabilistic networks we can analyse the average performance of an algorithm if the input network has a random component.

In the next section we will formally describe the problem that we will solve and the underlying probabilistic assumptions for analysing the average case behavior.

2 Problem Definition

We define the O-D shortest path problem as follows: Let $G = (V, E)$ be a network in which V is a node set with n nodes, E is an arc set with m arcs, and for each arc (i, j) there is an associated arc length c_{ij}. (Occasionally we will use l_e to denote the length of arc e). In addition, let O be a subset of nodes called the *origin nodes*, and let D be a subset of nodes called the *destination nodes*. Let $n_O = |O|$ and $n_D = |D|$. We do not necessarily assume that O and D are disjoint. The objective in the O-D shortest path problem is to find a shortest path from each node in O to each node in D.

The O-D shortest path problem generalizes several related problems. For example, if $|O| = 1$, and if $D = V$, then the O-D shortest path problem is the problem of finding the shortest distance from a single node to all other nodes. If $O = D = V$, then the O-D shortest path problem becomes the problem of finding the shortest path between all pairs of nodes in a network.

The O-D shortest path problem is the same as the D-O shortest path problem on the graph obtained from G by reversing all arcs in the graph. For this reason, we may assume without loss of generality that $n_O \leq n_D$. This assumption will simplify the statement of the running times.

In terms of worst case analysis, the best algorithm for the O-D shortest path problem is the best of the following two approaches:

i. Run the single source shortest path problem for each node $i \in O$.

ii. Run the all pairs shortest path problem using fast matrix multiplication.

In the first case, the running time is $O(n_D(m + n\log n))$ using Fredman and Tarjan's [4] fibonacci heap implementation of Dijkstra's algorithm. In the latter case, the running time is $O(n^{2.376})$ using the results of Coppersmith and Winograd [1].

2.1 Probabilistic Model

The main result of our paper is an algorithm for the O-D shortest path problem that has a very good expected running time on random networks. For our random network model, we assume that each arc (i, j) has a probability $m/n(n-1)$ of being in G. Thus the expected number of arcs in G is m. We also assume for our analysis that $m \geq n\log n$ so that each node has degree $\Omega(\log n)$ and so with very high probability the graph is strongly connected [3]. We assume that the arcs in G have i.i.d. exponentially distributed lengths; however, we may relax the assumption that the distribution is exponential, as did Luby and Ragde [8] in their analysis of the single source single sink shortest path problem. Our results generalize to a number of other distributions on the arcs costs. For the initial analysis, we also assume that the input instance is a complete graph. Therefore the only random component in the input are the arc lengths (costs). Note that all logarithms in the paper are to base e.

2.2 Summary of running times

Our algorithm is based on the bidirectional search algorithm analyzed by Luby and Ragde [8]. They solved the single source single destination problem on random networks in expected running time $O(m/\sqrt{n})$, which is a factor of $\sqrt{n}$ better than that obtainable using Dijkstra's algorithm. Our algorithm for the O-D shortest path problem optimally finds the shortest path from each node in O to each node in D in expected running time $O(n_O n_D \log^2 n + \min(n_D\sqrt{n\log n}\ \ m/n, m + n_D\sqrt{n\log n}\log n))$. In the case that the arcs incident to each node are sorted in increasing order of distance, the running time of Luby and Ragde's algorithm for the single source single sink case reduces to $O(\sqrt{n}\log n)$ and the running time of our algorithm for the O-D shortest path problem reduces to $O(n_O n_D \log^2 n + n_D\sqrt{n\log n}\log n)$. As a result, whenever $n_D = o(\sqrt{n}/\log n)$, the running time of the algorithm is sublinear; i.e., one can find the shortest path between all nodes in O and all nodes in D faster than the time it takes to read the data for a single source shortest path problem. For the case that $n_O = n_D = \sqrt{m}/\log n$, the running time of the algorithm is $O(m)$, which is a factor of $\sqrt{m}/\log n$ better than the running time achieved by solving $\sqrt{m}/\log n$ single source shortest path problems. It is also a factor of $\sqrt{m}/\log n$ faster than the time bound obtained by applying Luby and Ragde's algorithm $n_O n_D$ times, once for each origin-destination pair.

For the all-pairs shortest path problem, the algorithm with the minimum expected running time is due to Grimmett and Frieze [6]. Its running time is $O(n^2 \log n)$. Our algorithm has a running time that is a factor of $\log n$ worse in this case. For the case of the single-origin single-destination shortest path problem, our algorithm is a factor of $\sqrt{\log n}$ slower than that of Luby and Ragde.

3 Single Source Single Sink Case

3.1 On Bidirectional Dijkstra.

Our algorithm is based on the bidirectional Dijkstra algorithm as analyzed in Luby and Ragde [8]. In this section we describe the bidirectional version of Dijkstra's algorithm and state relevant properties of the algorithm.

First, we need some notation. Given two nodes i and j in G, let $d(i,j)$ be the length of the shortest path from i to j. As stated in the introduction, we assume that there is some path from i to j, and thus this distance is finite.

Suppose we want to identify the shortest path from an origin node s to a destination node t. The way that Dijkstra's algorithm works is as follows: Dijkstra's algorithm "permanently labels" the nodes of V in an iterative manner. Let $\pi(k)$ denote the node of V that is "permanently labeled" at the k-th iteration. Dijkstra's algorithm labels the nodes so that

$$d(s,\pi(1)) \leq d(s,\pi(2)) \leq \ldots \leq d(s,\pi(n)).$$

In other words, it identifies the nodes in increasing distance from node s. (For example, $\pi(1) = s$.) By the end of the k-th iteration, it has identified shortest paths to each of the k closest nodes to s. We will refer to the subtree obtained at the end of the k-th iteration of Dijkstra's algorithm as the k-stage source tree for node s, and we will denote it as $S_s(k)$. Similarly, we could have run Dijkstra to find the shortest path distances to node t. In this case, Dijkstra's algorithm would have permanently labeled the nodes in some other order, say $\sigma(1),\ldots,\sigma(n)$. As in the usual implementation of Dijkstra's algorithm, this "reverse Dijkstra" identifies nodes in the order so that

$$d(\sigma(1),t) \leq d(\sigma(2),t) \leq \ldots \leq d(\sigma(n),t).$$

By the end of the k-th iteration, it would have identified a shortest path subtree $T_t(k)$ consisting of the k closest nodes to node t. We will refer to this subtree as the k-stage sink tree for node t. In general, we will refer to the k-stage source tree more briefly as a source tree and we will refer to the k-stage sink tree more briefly as a sink tree.

If one runs Dijkstra's algorithm from the origin node or into the destination node, then one would expect to label $O(n)$ nodes before the shortest path from the origin to the destination is determined. However, one can do better on average if one

simultaneously runs Dijkstra's algorithm from the origin and into the destination. This improvement is based on the following Lemma, which can be found in Luby and Ragde [8]. We find the proof to be elementary and enlightening, and therefore have included it.

Lemma 1 *Let S be a source tree from the origin node s and let T be a sink tree from the destination node t (with $t \neq s$). Suppose that S and T have a node in common.*

Let $\lambda_1 = min(d(s,i) + d(i,t) : i \in S \cap T)$.
Let $\lambda_2 = min(d(s,i) + c_{ij} + d(j,t) : i \in S$ *and* $j \in T - S)$.
Then $d(s,t) = min(\lambda_1, \lambda_2)$.

Proof:
We first observe that each of λ_1 and λ_2 is the length of some path from node s to node t. We want to prove that it is the length of the shortest path. Suppose that $i \in S \cap T$ is the node for which λ_1 is minimum. Let p be any path from s to t such that there is a node $k \notin S \cup T$. Then the length of p is at least $d(s,k) + d(k,t) \geq d(s,i) + d(i,t) = \lambda_1$, since k is on neither the source tree nor the sink tree. We conclude that there is a shortest path from s to t that consists of nodes in $S \cap T$.

We note that the Lemma is true if $t \in S$, since in this case $\lambda_1 \leq d(s,t) + d(t,t) = d(s,t)$. The only remaining case to consider is when the shortest path p consists only of nodes of $S \cup T$, and $t \notin S$. Let j be the first node of p that is in $T - S$, and let i be the node before j on the path. Then the length of p is at least $d(s,i) + c_{ij} + d(j,t) \geq \lambda_2$.

□

In general, if S is the source tree for some origin node s and if T is the sink tree for some destination node t, then we let

$$\lambda_1(S,T) = \min(d(s,i) + d(i,t) : i \in S \cap T),$$

and we let

$$\lambda_2(S,T) = \min(d(s,i) + c_{ij} + d(j,t) : i \in S, j \in T - S).$$

The bidirectional shortest path algorithm works as follows:

Procedure Bidirectional Dijkstra
begin
/;/;/;/; find the minimum k such that the k stage source tree $S(k)$ from node s and /;/;/;/; the k stage sink tree $T(k)$ to node t have a node in common;
/;/;/;/; $d(s,t) = min(\lambda_1(S,T), \lambda_2(S,T))$
end

Theorem 2 (Luby and Ragde) *Let $G = (V, E)$ be a random graph with i.i.d. exponential arc costs. Then the bidirectional Dijkstra's algorithm computes the shortest path from s to t in expected time $O(m/\sqrt{n})$.*

The single source single sink algorithm of Luby and Ragde is a two phase algorithm. In the first phase of the algorithm, we evaluate the value of λ_1 and in the second phase we evaluate the value of λ_2. Therefore the first phase is the execution of Dijkstra's algorithm forward from the source and backward from the sink one node at a time till the source tree and the sink tree meet. In the second phase they implicitly look at all the arcs between the two trees to determine the value of λ_2. One key aspect of the analysis was that the expected number of nodes in the source tree and the sink tree was $O(\sqrt{n})$, and the number of arcs that needed to be scanned by Dijkstra's algorithm was $O(\sqrt{n})$. The second phase is more complex. If the arcs lengths are independent, identically distributed random variables, we can simplify the second phase of the algorithm considerably to give the same running time. The modification of the second phase is based on a special case of the following observation due to Grimmett and Frieze [6]. Consider a subgraph of the input graph restricted to the shortest $20 \log n$ arcs out of every node. The set of arcs in this subgraph is termed *active arcs*. The shortest path between any two nodes lies in this subgraph with high probability. This observation is true for independent, identically distributed arc lengths such that the distribution function of the arc lengths, $F(x)$ satisfies some weak conditions. This is stated formally in the following lemma.

Let

$$d(i, j) = \text{Distance between nodes } i \text{ and } j.$$

$$d^* = \max_{i,j} d(i, j).$$

Let

$$M = \min(c_{ij} : (i, j) \text{ is not active}).$$

Lemma 3 (Grimmett and Frieze) *If $F(x)$ is differentiable at 0 and $F'(0) > 0$ then*

$$Pr\,[d^* > M] = O(\frac{1}{n})$$

Therefore in the second phase of the algorithm, we consider only the shortest $20 \log n$ arcs out of each node in the source tree to do the patching-up. If the first phase of the algorithm ends after K steps (the number of steps is a random variable), then the number of arcs explored in the second phase of the algorithm is $20K \log n$. Since the expected length of the first phase of the algorithm i.e., the expected value of K is $O(\sqrt{n})$, the running time of the second phase of the modified algorithm is $O(\sqrt{n} \log n)$. Another related result due to Zemel and Hassin [7] shows that the length of the shortest path between any two nodes in a random graph

with the lengths satisfying some weak assumptions is bounded by $\frac{c \log n}{n}$ with high probability for some constant c. At this stage we define some notation that we will use in the analysis of our bidirectional search algorithm. Let us first consider the working of the algorithm in the unidirectional case. Usually one views Dijkstra's algorithm as finding the shortest path from the source node to all other nodes in the graph. As in Luby and Ragde [8], we will adopt a related view of Dijkstra's algorithm that is more suited to our analysis. Associated with the algorithm is a real variable L which is initialized to zero and increases continuously during the execution of the algorithm. Let $d(s,i)$ denote the length of the shortest path from s to i. All arcs are initially considered to be *inactive*. An arc (i,j) becomes *active* when $L = d(s,i)$ and is *revealed* when $L = d(s,i) + c_{ij}$. After it is revealed, it is no longer considered active. At time 0, Dijkstra's algorithm labels the origin node s permanent, and waits for the next active arc to be revealed. If arc (s,j) is revealed when $L = l$ then there is a path from s to j with length l. In general, if arc (i,j) is revealed when $L = l$, then there is a path form s to j with distance l. If j is not permanently labelled before this point, then it can become permanently labelled. Note that no arc (i,j) can be revealed till after node i is permanently labelled. Also note that (i,j) may remain active even if both i and j are permanently labelled, so long as (i,j) has not been revealed. If an active arc has both its end points permanently labelled, it is called an *internal* arc. All other active arcs are *external* arcs. In the next lemma we determine the probability that (i,j) will be the next arc to be revealed for each active arc (i,j) at some point in the running of the algorithm.

Lemma 4 *At any stage of the unidirectional Dijkstra's algorithm each active edge is equally likely to be revealed.*

Proof:
The proof of this lemma is very similar to the proof given in Luby and Ragde [8]. Consider the algorithm when $L = t$. Let the active edges at this stage be $e_1, e_2, \ldots e_k$. Let us assume that e_i was activated at $L = x_i$. We therefore know that $l_i \geq t - x_i$. The next edge to be revealed is the active edge that has minimum $l_i + x_i$. For any $y \geq 0$

$$\begin{aligned} \Pr[l_i + x_i \leq y \mid l_i \geq t - x_i] &= \Pr[l_i \leq (t - x_i) + y - t \mid l_i \geq t - x_i] \\ &= \Pr[l_i \leq y - t] \end{aligned}$$

This quantity is independent of the edge e_i since the edge lengths are independent identically distributed exponential random variables. Thus each active edge e_i is equally likely to get chosen.

□

In the bidirectional case we execute Dijkstra steps from the source node and into the sink node. At some intermediate stage of the algorithm, let S be the set of nodes labelled from the source and let T be the set of nodes labelled from the sink.

During the running of the algorithm there are S *external*, S *internal*, T *external* and T *internal* arcs. The definitions of these sets are similar to definitions given in the unidirectional search algorithm. Further we define *cross* arcs as arcs that have their tail in S and head in T. Similar to the case of unidirectional search we define the qunatities L_S and L_T to be the distance searched from s and t at some stage of the algorithm. The source tree and the sink tree behave exactly in the same way as in the unidirectional search case. However, the analysis of the algorithm is complicated by the fact that for the cross arcs, we have information about the length of the arcs both from the source tree and from the sink tree. For instance, if some arc e was activated at time x_1 on the source side and at time x_2 on the sink side then we know that the length of the edge

$$l_e \geq \max\{L_S - x_1, L_T - x_2\}.$$

Thus if $L_S - x_1 < L_T - x_2$ then this arc cannot be discovered from the source side. Therefore we define S *enabled* edges to the cross arcs for which

$L_S - x_1 \geq L_T - x_2$. T *enabled edges* are cross arcs for which $L_S - x_1 < L_T - x_2$. We state this result in the following lemma.

Lemma 5 *In the bidirectional search algorithm, all the S enabled arcs are equally likely to be discovered from the source side search and all T enabled arcs are equally likely to be discovered from the sink side search.*

3.2 Composite Algorithm

Although, the bidirectional search algorithm described in the previous section can be directly extended for the O-D shortest path problem, the analysis cannot be directly extended. We therefore modify this algorithm suitably, and present the modified algorithm first for the single source-single sink case and later for the O-D shortest path problem. The outline of the algorithm is the following: We execute a fast procedure to determine the shortest path between a given source node s and a sink node t. This procedure identifies the shortest $s-t$ path with high probability. If it has identified the shortest $s-t$ path, we stop. If not we run another (slower) procedure which always determines the solution. The probability of identifying the minimum cost $s-t$ path is sufficiently high that the expected running time will be dominated by the running time of the first procedure. In this case the main procedure is a truncated bidirectional search algorithm. We execute the Phase 1 of the bidirectional search algorithm from the source and the sink. In the previous case we executed the algorithm till the source tree and the sink tree intersected. In the truncated bidirectional search algorithm we execute phase 1 of the algorithm for a fixed number $(4\sqrt{n \log n})$ of steps. We can show that the source tree and the sink tree will intersect with high probability by the end of Phase 1. In the second phase of the bidirectional search algorithm, we restrict attention to the shortest $20 \log n$ arcs out of every node in the source tree for "patching up". We can show that

with high probability, the shortest path lies within the subset of arcs considered. We perform a test to determine if the shortest path has indeed been determined. If not, we execute the unidirectional search algorithm that runs in expected time $O(n \log n)$.

We now outline the running time analysis for two versions of this algorithm. In the first version of the algorithm, we assume that the arcs are sorted in increasing length from each of the nodes. In the second version we will assume that the arcs are not sorted. The data structure that is used to implement the algorithm is the following: At each node i that is permanently labelled we initialize the CURRENT ARC(i) to be the shortest arc out of node i. Whenever this arc is revealed, we replace CURRENT ARC(i) with the next longest arc. We maintain a priority queue Q with the CURRENT ARC of all the labelled nodes. At every step of the algorithm, we find the minimum element in Q. If this arc (i, j) is an internal arc, we replace CURRENT ARC(i) with the next longest arc. If the arc is an external arc we permanently relabel node j and add CURRENT ARC(j) to Q. Therefore there are two costs associated with the running of Phase 1 of the algorithm. One is time needed to find the minimum element in the priority queue which is done each time an arc (both internal and external) is discovered. The time for finding the next minimum is $O(\log n)$. The second is the cost associated with initializing and updating the CURRENT ARC at each node. This initializing and updating is done each time an arc (both internal and external) is discovered. If the arcs are sorted then the time for each call is $O(1)$. If the arcs are not sorted, then using a priority queue at every node we can show that the time per call is propotional to the degree of the node. Therefore the expected running time of the first phase if the arc lengths are not sorted is the product of the number of arcs revealed in the first phase and the average degree of a node. (The average degree of a node is $\frac{m}{n}$ in the general case). We will show in lemma 7 that the number of nodes permanently labelled by Dijkstra's algorithm and the number of edges revealed in the first phase is $O(\sqrt{n \log n})$ on average. Since we examine only the shortest $20 \log n$ arcs out of every source tree node in the second phase of the algorithm, the time to run the second phase of the algorithm is $O(\sqrt{n \log n} \log n)$. The overall analysis of the running time of the composite algorithm (Sorted case) is as follows:

$$
\begin{aligned}
T &= \text{Running time of composite algorithm} \\
T_1 &= \text{Running time of main algorithm} \\
T_2 &= \text{Running time of backup algorithm} \\
p &= \text{Probability of success of main algorithm}
\end{aligned}
$$

$$
E[T] = p\, E[T_1] + (1 - p)\, E[T_2]
$$

$$\leq O\left(\left(1-\frac{1}{n}\right)(\sqrt{n\log n}\log n)+\frac{1}{n}n\log n\right)$$

$$= O(\sqrt{n\log n}\log n)$$

The overall analysis of the running time of the composite algorithm (Unsorted case) is as follows:

$$\begin{aligned} T &= \text{Running time of composite algorithm} \\ T_1 &= \text{Running time of main algorithm} \\ T_2 &= \text{Running time of backup algorithm} \\ p &= \text{Probability of success of main algorithm} \end{aligned}$$

$$E[T] = p\,E[T_1] + (1-p)\,E[T_2]$$

$$\leq O\left(\left(1-\frac{1}{n}\right)(\sqrt{n\log n}\frac{m}{n})+\frac{1}{n}n\log n\right)$$

$$= O(\sqrt{n\log n}\frac{m}{n})$$

This algorithm is suitable for generalization to the O-D shortest path problem. We now show some of the detailed analysis of the running time of the algorithm and study some characteristics of the shortest path that will be used in the analysis of the O-D shortest path algorithm.

Lemma 6 *The probability that the source and the sink nodes intersect if we execute* $4\sqrt{n\log n}$ *steps of Dijkstra's algorithm from the source node and to the sink node is greater than* $1-\frac{1}{n^2}$.

Proof:
Let us call the source node i and the sink node j. Consider source tree S_i and sink tree T_j. Let us define the E_{ij} to be the event that source tree S_i and sink tree T_j do not intersect at the end of $4\sqrt{n\log n}$ steps. Let us assume that the source tree and the sink tree are grown simultaneously. One can view this in terms of two clocks (perhaps like chess clocks). First, we identify the next node in S_i by stopping the clock for T_j and starting the clock for S_i. At some point an S_i active arc is revealed. If this arc is internal, we continue till we permanently label some node on the source side. At this point the clock for S_i is stopped and the clock for T_j is started. That is, at stage k there are k nodes in the source tree and k nodes in the

sink tree. All the cross arcs are either S-enabled or T-enabled. Let C be the event that a cross arc is not discovered in $4\sqrt{n \log n}$ steps.

$$\begin{aligned} \text{Number of active cross arcs} &= k^2 \\ \text{Number of external arcs} &= 2k(n-2k) \\ \Pr[\text{ Cross arc is discovered}] &= \frac{k}{(2n-3k)} \geq \frac{k}{2n} \end{aligned} \tag{1}$$

If the arc discovered is an S-external arc, we increase L_T till a T-external arc or a cross arc is discovered and vice versa. Therefore,

$$\begin{aligned} \Pr[\text{ C }] &\leq \left(1-\frac{1}{2n}\right)\left(1-\frac{2}{2n}\right)\cdots\left(1-\frac{4\sqrt{n\log n}}{2n}\right) \\ &\leq e^{-\frac{1}{2n}\left(1+2+\cdots 4\sqrt{n\log n}\right)} \\ &\leq e^{-4\log n} \\ &\leq \frac{1}{n^4} \end{aligned}$$

The probability that a cross arc is not revealed is equivalent to saying that the source tree and sink tree do not intersect. Therefore,

$$\Pr[E_{ij}] \leq \frac{1}{n^4}.$$

□

In the next lemma we shall bound the total number of arcs, both internal and external revealed by the algorithm. This will give us a bound on the running time of Phase 1 of the algorithm as described earlier.

Lemma 7 *The total number of arcs revealed in the first phase is* $O(\sqrt{n \log n})$.

Proof:
Since we run the first phase of the algorithm till $4\sqrt{n \log n}$ nodes are permanently labelled from the source and the sink, the total number of external arcs revealed is $8\sqrt{n \log n}$. The total number of internal arcs revealed can be bounded as follows: Consider the source tree when there are k permanently labelled nodes (stage k). The number of active internal arcs is upper bounded by k^2. The total number of external arcs is $k(n-k)$.

$$\begin{aligned} \Pr[\text{Internal arc is revealed at stage } k] &\leq \frac{k^2}{k(n-k)} \\ &= \frac{k}{n-k} \end{aligned}$$

Let I be the total number of internal arcs revealed during the first phase of the algorithm. Then,

$$E[I] \leq \sum_{k=1}^{4\sqrt{n \log n}} \frac{k}{n-k} = O(\log n).$$

Therefore the total number of arcs revealed in the first phase of the algorithm is $O(\sqrt{n \log n})$.

□

The running time analysis of the second phase was discussed earlier. This completes the running time analysis of the composite algorithm for the single source single sink case. In the next section we will study some properties of the shortest path that will be useful in the analysis of the O-D shortest path algorithm.

3.3 Some Properties of the Shortest Path

In this section we will bound the number of nodes along the shortest path between any two nodes in the graph. We first introduce some notation. Consider the process of growing the shortest path tree in the first phase of the algorithm. Recall that the algorithm is in *stage k* if k nodes have been labelled permanently by the algorithm. A node u in the shortest path tree is at *level t* if the number of arcs in the shortest path from u to the root node is t.

Lemma 8 *Let $H_k(t)$ be the probability that a randomly picked node of the k stage tree is at level t. Then*

$$H_k(t) = \frac{k-1}{k} H_{k-1}(t) + \frac{1}{k} H_{k-1}(t-1) \quad t = 0, 1, \ldots k-1.$$

Proof:
An alternate way of viewing the first phase of the algorithm is as follows: At every stage of the algorithm, we pick one labelled node and one unlabelled node at random. (Each labelled node is equally likely to get picked and so is each unlabelled node). This unlabelled node is made permanent. After this is done, suppose we pick a node at random from all the labelled nodes. Let us assume that we are at stage k of the algorithm. If each of the k nodes is equally likely to be chosen then with probability $\frac{k-1}{k}$ we pick a node that was already labelled by level $k-1$ and this accounts for the first term on the right hand side. With probability $\frac{1}{k}$ we pick the node labelled at level k. This node could have any one of the previously labelled node as its predecessor with equal probability. This accounts for the second term on the right hand side.

□

Lemma 9 *Let $\phi(H_k) = \sum_{t=0}^{\infty} Z^t H_k(t)$ be the Z- transform of $H_k(t)$. Then $\phi(H_k) = \frac{\prod_{i=1}^{k-1}(Z+i)}{k!}$*

Proof:
At the first stage of the algorithm only one node has been labelled. (Either the source of the sink). This node is at level zero. Therefore $H_1(0) = 1$ and $\phi(H_1) = 1$. We know that

$$H_k(t) = \frac{k-1}{k} H_{k-1}(t) + \frac{1}{k} H_{k-1}(t-1) \quad t = 0, 1, \ldots k-1.$$

Multiplying this equation by Z^t and summing over t we get

$$\begin{aligned} \sum_{t=0}^{\infty} Z^t H_k(t) &= \sum_{t=0}^{\infty} Z^t \frac{k-1}{k} H_{k-1}(t) + \sum_{t=0}^{\infty} Z^t \frac{1}{k} H_{k-1}(t-1) \\ \phi(H_k) &= \frac{k-1}{k}\phi(H_{k-1}) + \frac{Z}{k}\phi(H_{k-1}) \end{aligned}$$

Rewriting the previous equation as

$$\frac{\phi(H_k)}{\phi(H_{k-1})} = \frac{k-1+Z}{k}.$$

Therefore,

$$\begin{aligned} \phi(H_k) &= \frac{\phi(H_k)}{\phi(H_{k-1})} \frac{\phi(H_{k-1})}{\phi(H_{k-2})} \cdots \frac{\phi(H_2)}{\phi(H_1)} \phi(H_1) \\ &= \frac{\prod_{i=1}^{k-1}(Z+i)}{k!} \end{aligned}$$

Note that $\phi(H_k)$ is a function of k and Z. □

The next theorem and proof is due to Chernoff. In this theorem we will bound the probability that a positive random variable is greater than some constant in terms of its Z- transform. The Z-transform of a random variable X is $\phi(X) = E[Z^X]$.

Theorem 10 (Chernoff) *Let X be any positive random variable whose Z-transform is $\phi(X)$. Then,*

$$Pr[X > a] \leq \phi(X) Z^{-a}.$$

Proof:
For any positive random variable Y, the well known Markov inequality states that

$$Pr[Y > a] \leq \frac{E[Y]}{a}.$$

We apply this inequality to the Z-transform of X which is just $E[Z^X]$. Therefore,

$$Pr[Z^X > t] \leq \frac{E[Z^X]}{t} = \frac{\phi(X)}{t}.$$

Let $t = Z^a$ then,

$$Pr[Z^X > Z^a] \leq \phi(X)Z^{-a}$$
$$Pr[X > a] \leq \phi(X)Z^{-a}.$$

This is true for all $Z > 1$.

□

Lemma 11 *Let F be the event that at the end of $4\sqrt{n \log n}$ steps, a randomly picked node is at a distance greater than $6 \log n$ from the source. Then,*

$$Pr[F] \leq \frac{1}{n^3}.$$

Proof:
We use the Chernoff bound to get an upper bound for the probability of F. The Z-transform of the distribution for the depth of a randomly picked node after k steps have been executed is

$$\phi(H_k) = \frac{1}{k!} \prod_{i=1}^{k-1} (Z + i)$$

Let X be the depth of a randomly picked node after k steps of the algorithm. We want to determine $Pr[X > 6 \log n]$.

$$Pr[X > 6 \log n] = Pr[Z^X > Z^{6 \log n}]$$

Using the Chernoff bound with $Z = e$ and $k = 4\sqrt{n \log n}$ we get,

$$\begin{aligned} Pr[X > 6 \log n] &\leq \phi(H_{4\sqrt{n \log n}}) \frac{1}{n^6} \\ &\leq \phi(H_n) \frac{1}{n^6} \\ &\leq \frac{1}{n!}(e+1)(e+2)\ldots(e+n-1)\frac{1}{n^6} \\ &\leq \frac{1}{n!}(4)(5)\ldots(n+2)\frac{1}{n^6} \\ &\leq \frac{1}{n^3} \end{aligned} \tag{2}$$

□

Lemma 12 *For any sink tree,*

$$Pr[\textit{All nodes are at a distance} \leq 6\log n] \geq 1 - \frac{1}{n^2}.$$

The expected depth of any node in the sink tree is less than $7\log n$.

Proof:
From the previous lemma, we know that at the end of $4\sqrt{n\log n}$ steps of the algorithm,

$$Pr[\text{Randomly picked node is at depth} > 6\log n] \leq \frac{1}{n^4}.$$

Let,

$$E_i = \{\text{Node } i \text{ is at depth} > 6\log n\} \quad i = 1, 2, \ldots, 4\sqrt{n\log n}$$

Each of the $4\sqrt{n\log n}$ nodes is equally likely to get picked. Therefore,

$$\frac{1}{4\sqrt{n\log n}} \sum_{i=1}^{4\sqrt{n\log n}} Pr[E_i] \leq \frac{1}{n^3}.$$

Then,

$$\sum_{i=1}^{4\sqrt{n\log n}} Pr[E_i] \leq \frac{4\sqrt{n\log n}}{n^3} < \frac{1}{n^2}.$$

Since all the probabilities are non-negative quantities,

$$Pr[E_i] \leq \frac{1}{n^2} \quad \forall\ i.$$

This implies that

$$Pr[\text{All nodes are at depth } \leq 6\log n] \geq 1 - \frac{1}{n^2}.$$

This also implies that

$$E[\text{ Depth of any node }] \leq (1 - \frac{1}{n^2})6\log n + \frac{1}{n^2}4\sqrt{n\log n} \leq 7\log n.$$

□

These results characterizing the number of nodes along the shortest path will be used later on in the analysis of the O-D shortest path algorithm.

4 The Bidirectional O-D Shortest Path Algorithm

Our algorithm is an extension of the composite algorithm described in the previous section. The algorithm may be summarized as follows:

For each node $i \in O$, let $S_i(k)$ denote the k-stage source tree rooted at node i and for each node $j \in D$, let $T_j(k)$ denote the k-stage sink tree rooted at node j.

begin

Execute $4\sqrt{n \log n}$ steps of Dijkstra's algorithm from all $i \in O$ and $j \in D$;

for each $i \in O$ and for each $j \in D$, $d(i,j) = min((\lambda_1(S_i, T_j), \lambda_2(S_i, T_j))$;

end

We show in lemma 13 that at the end of $4\sqrt{n \log n}$ steps of Dijkstra's algorithm from each of the source and sink nodes, all source trees and sink trees intersect with very high probability. (We say that source tree S_i and sink tree T_j intersect if $S_i \cap T_j \neq \emptyset$). This algorithm succeeds with high probability and verifies if it has succeeded. If we determine that the algorithm has failed, we then execute the single souce single sink shortest path algorithm n_D times.

Although the summary description of our algorithm is elementary, there are several implementation details that we need to focus on. In addition, we need to show that the analysis of the running time is accurate. The fact that the algorithm obtains optimal O-D shortest paths follows directly from Lemma 1. The data structure for the implementation of the algorithm is similar to the implementation in the single source-single sink case. The running time of the composite algorithm for the O-D shortest path problem where the arcs are sorted in increasing length from every node are analysed as follows:

$$
\begin{aligned}
T &= \text{Running time of composite algorithm} \\
T_1 &= \text{Running time of main algorithm} \\
T_2 &= \text{Running time of backup algorithm} \\
p &= \text{Probability of success of main algorithm}
\end{aligned}
$$

$$
\begin{aligned}
E[T] &= p\,E[T_1] + (1-p)\,E[T_2] \\
&\leq O\left(\left(1 - \frac{1}{n}\right)(n_D\sqrt{n \log n} \log n + n_O n_D \log^2 n) + \frac{1}{n} n_D n \log n\right) \\
&= O(n_D\sqrt{n \log n} \log n + n_O n_D \log^2 n)
\end{aligned}
$$

If the arcs are not sorted then the analysis of the runing time is as follows:

$$
\begin{aligned}
T &= \text{Running time of composite algorithm} \\
T_1 &= \text{Running time of main algorithm} \\
T_2 &= \text{Running time of backup algorithm} \\
p &= \text{Probability of success of main algorithm}
\end{aligned}
$$

$$
\begin{aligned}
E[T] &= p\,E[T_1] + (1-p)\,E[T_2] \\
&\leq O\left(\left(1-\frac{1}{n}\right)(n_D\sqrt{n\log n}\frac{m}{n} + n_O n_D \log^2 n) + \frac{1}{n} n_D n \frac{m}{n}\right) \\
&= O(n_D\sqrt{n\log n}\frac{m}{n} + n_O n_D \log^2 n)
\end{aligned}
$$

4.1 Analysis of Phase 1

We will first show that the source tree and sink tree intersect with high probability at that end of Phase 1. Since the running time of the algorithm is proportional to the number of arcs examined by the algorithm, we show that the number of external and internal arcs examined during the first phase of the algorithm is $O(\log n)$. This gives a bound on the expected running time of the first phase of the algorithm.

Lemma 13 *The probability that all the source trees and all the sink trees intersect if we execute $4\sqrt{n\log n}$ steps of Dijkstra's algorithm from every source and sink node, is greater than $1-\frac{1}{n^2}$.*

Proof:
Let us first consider a fixed source node i and a fixed sink node j. Let us consider source tree S_i and sink tree T_j. Let us define the E_{ij} to be the event that source tree S_i and sink tree T_j do not intersect at the end of $4\sqrt{n\log n}$ steps. From lemma 6, we know that

$$\Pr[E_{ij}] \leq \frac{1}{n^4}.$$

This is true for any source tree sink tree pair. Using the fact that the probability of the union of events is less than the sum of the probability of the individual events we obtain,

$$\Pr[\cup_i \cup_j E_{ij}] \leq \sum_i \sum_j \Pr[E_{ij}] \leq \frac{n_O n_D}{n^4} \leq \frac{1}{n^2}.$$

This implies that

$$\Pr\text{ [All source trees and sink trees intersect]} \geq 1-\frac{1}{n^2}.$$

□

Recall that active arcs were defined to be the shortest $20 \log n$ arcs out of every node.

Lemma 14 *The running time to determine the active arcs is* $O(min(m, n_D\sqrt{n \log n}\ \ m/n))$.

Proof:
We can easily determine the active arcs by scanning the arc list in $O(m)$ time. Alternatively, we determine the active arcs only for the nodes that we label during the execution of Dijkstra's algorithm. The total number of nodes labelled thus is $O(n_D\sqrt{n \log n})$. For each node the average time taken to determine the active arcs is $O(\frac{m}{n})$ because $\frac{m}{n}$ is the average degree of a node.

□

Lemma 15 *The total time taken to execute Dijkstra's algorithm as restricted to active arcs from all the source and sink nodes is* $O(n_D\sqrt{n \log n} \log n)$.

Proof:
We execute $4\sqrt{n \log n}$ steps of Dijkstra's algorithm from every sink and source node. Each Find Min operation takes $\log n$ steps using a priority queue implementation. Since we have assumed that $n_D \geq n_O$ the result follows.

□

4.2 Analysis of Phase 2

In this section we will analyse the running time of the second phase of the algorithm. This phase involves the patching up operation. We have to look at the cross arcs, i.e., arcs directed from a source tree into a sink tree. Each source tree and sink tree has $4\sqrt{n \log n}$ nodes. There are $n_O n_D n \log n$ cross arcs in total since G is assumed to be fully dense. Therefore scanning all of the cross arcs is not an efficient method of implementing this step. We restrict attention to the shortest $20 \log n$ arcs out of every node in the source tree and use these arcs only for patching up. We call this set of arcs *active* . We will show that the time to carry out patching is $O(n_O n_D \log^2 n)$. In running the $O - D$ shortest path algorithm, we maintain for each source tree S_i and for each sink tree T_j a list of nodes on S_i and T_j. Concurrently, we maintain a doubly linked list of nodes in $\cup S_i$ and another doubly linked list for nodes in $\cup T_j$. For each node j on this list we maintain a list of all source trees and sink trees containing node j. These lists can be maintained in $O(1)$ additional time per tree operation. Thus maintaining these lists is not a bottleneck.

We would also maintain an $n_O \times n_D$ matrix L_1, where the value $L_1(i,j)$ is the best current bound for $\lambda_1(S_i, T_j)$. (Recall the definitions of λ_1 and λ_2 given in Section 3.1). Initially, each entry of L_1 is set to ∞. Whenever a node v is added to the source tree for node i, then for each sink tree T_j containing node v, we let

$$L_1(i,j) = \min(L_1(i,j), d(i,v) + d(v,j)).$$

Similarly we maintain an $n_O \times n_D$ matrix L_2, where the value $L_2(i,j)$ is the current best value of $\lambda_2(S_i, T_j)$. Suppose a node v is added to the source tree for node i, then for each active arc (v,w) among the best $20 \log n$ arcs from node v and for each sink tree T_j containing w

$$L_2(i,j) = \min(L_2(i,j), d(i,v) + c_{vw} + d(w,j)).$$

We also maintain a matrix L whose entry $L(i,j)$ is

$$\min(L_1(i,j), L_2(i,j)).$$

This array is therefore updated whenever either L_1 or L_2 is updated. It is easy to show that the algorithm terminates with $L(i,j)$ being the shortest path length between source node i and sink node j as restricted to the active arcs if source tree S_i and sink tree T_j intersect. We now bound the expected number of steps to maintain L_1 and L_2.

To update $L_1(i,j)$ requires $O(1)$ steps each time S_i intersects T_j. Note that in the single source single sink case, we terminate the execution of Phase 1 of the algorithm when the source tree intersects with the sink tree. In the O-D shortest path case, we continue the execution of the first phase for $4\sqrt{n \log n}$ steps from each source and sink tree. This has to be done because we want all source trees and sink trees to intersect with high probability. Therefore a source tree and sink tree can intersect a multiple number of times.

In the next theorem we will bound the expected number of times S_i and T_j intersect. For proving this theorem we define the following terms. For a fixed sink tree T_j, and for any given node v in the sink tree, the *closure* of v denoted as $\mathrm{CL}(v)$ is defined as union of v and all the nodes that are in the path in T_j from v to j. Given a set W of nodes in T_j, its closure, $\mathrm{CL}(W)$ is defined as the union of the closure of the nodes in W.

Theorem 16 *For a source tree S_i and a sink tree T_j,*

$$E[|S_i \cap T_j|] \leq E[|CL(S_i \cap T_j)|] = O(\log^2 n).$$

Proof:
The inequality is trivially true since the closure of a set subsumes the set itself. We now prove that

$$|CL(S_i \cap T_j)| = O(\log^2 n).$$

Let us assume that we have executed $4\sqrt{n \log n}$ steps of Dijkstra's algorithm from the sink node j. We are executing Dijkstra's steps from the source node and we are

at stage $k+1$ of the source tree. We first observe by Lemma 12 that the length of any path in the sink tree is less than $6 \log n$ with high probability and that the expected length of any path is less than $7 \log n$. Let us assume that the source tree has intersected with the sink tree at some node v at this stage. Let u be the node adjacent to v along the path from v to the sink node along T_j. The node added to the S_i at stage $k+1$ could be one of the following three cases:

1. $v \in CL(S_i \cap T_j)$
2. $v \in T_j - CL(S_i \cap T_j)$
3. $v \in V - (S_i \cup T_j)$

If the next node $v \in CL(S_i \cap T_j)$, then adding it to S_i, does not increase the number of nodes in CL $(S_i \cap T_j)$. If the next node is $w \in T_j - u$, then it can potentially increase the expected number of nodes in the closure by $7 \log n$ by lemma 12. If the next node added to the source tree is some node that is not on the sink tree, then the cardinality of the closure does not increase. Therefore, the only case in which the closure of the set increases is when the next node added to the source tree is a node in the sink tree but not in the closure of $S_i \cap T_j$. Let E_{k+1} be the event that the node added to the source tree at stage $k+1$ belongs to $T_j - CL(S_i \cap T_j)$. In order to upper bound the probability of E_{k+1}, let us first consider the case that all the arcs in the source tree are S-enabled. In this case

$$\begin{aligned} \text{Number of arcs from } S_i \text{ to } T_j &= 4k\sqrt{n \log n} \\ \text{Number of arcs from } S_i \text{ to } V - (S_i \cup T_j) &= k(n - k - 4\sqrt{n \log n}) \end{aligned}$$

This, implies that,

$$Pr[E_{k+1}] \leq \frac{4\sqrt{n \log n}}{n-k} \leq \frac{8\sqrt{n \log n}}{n}.$$

Let I_{k+1} represent the increase in the cardinality of the closure of $S_i \cap T_j$ at stage $k+1$ of the source tree. Then,

$$E[I_k] \leq \frac{8\sqrt{n \log n}}{n}(7 \log n).$$

Therefore the expected increase in the cardinality of the closure over all the $4\sqrt{n \log n}$ stages of the source tree is

$$\sum_{l=0}^{4\sqrt{n \log n}} E[I_l] < \frac{8\sqrt{n \log n}}{n}(7 \log n)(4\sqrt{n \log n}) = O(\log^2 n).$$

This gives an upper bound on the cardinality of $S_i \cap T_j$.

□

Since the number of potential updates to $L_1(i,j)$ is equal to the number of times S_i and T_j intersect, the previous theorem implies that the number of potential updates to $L_1(i,j)$ is $O(\log^2 n)$. Therefore the total number of updates for all source-sink pairs is $O(n_O n_D \log^2 n)$. The expected number of potential updates to $L_2(i,j)$ is the expected number of active arcs between $S_i - T_j$ and $T_j - S_i$. In the next theorem we shall bound the expected number of potential updates to $L_2(i,j)$.

Theorem 17 *The expected number of potential updates to L_2 is $O(\log^2 n)$.*

Proof:

We first consider a particular source tree S_i and a sink tree T_j. We consider these trees after executing $4\sqrt{n \log n}$ steps of Dijkstra's algorithm from the source tree and $4\sqrt{n \log n}$ steps of Dijkstra's algorithm from the sink tree. Consider a node v in $S_i - T_j$. There are $n-1$ arcs directed out of this node. These arcs can be classified into 4 types:

1. Arcs from v to S_i.
2. Arcs from v to $T_j - S_i$ that are S-enabled.
3. Arcs from v to $T_j - S_i$ that are T-enabled.
4. Arcs from v to $V - (S_i \cup T_j)$.

Let us call these sets A, B, C and D respectively. There are $4\sqrt{n \log n}$ arcs in A and therefore there are $n - 1 - 4\sqrt{n \log n}$ arcs in $B \cup C \cup D$. We want to estimate the number of active arcs in $B \cup C$, since this gives the number of active arcs between node v and $T_j - S_i$. The arcs in sets B and C are either S-enabled or T-enabled. By the definitions given in section 3.1, we see that S-enabled arcs are more likely to be active than T-enabled arcs i.e., the arcs in $B \cup D$ are more likely to be active arcs than arcs in C. All the arcs in D are S-enabled. We have to pick at least $20 \log n$ arcs out of $E = B \cup C \cup D$. We assume that all the arcs in $B \cup C \cup D$ are equally likely to be active in order to get an upper bound on the expected number of active arcs in $B \cup C$. Let p denote the probability that an arc in $B \cup C$ is active. Then,

$$p \leq \frac{20 \log n}{n - 1 - 4\sqrt{n \log n}} = O\left(\frac{20 \log n}{n}\right).$$

There are at most $4\sqrt{n \log n}$ nodes in $B \cup C$, therefore

$$\begin{aligned} \text{Expected number of active arcs in } B &\leq 4\sqrt{n \log n} \quad O\left(\frac{20 \log n}{n}\right) \\ &= O\left(\frac{\sqrt{n \log n} \log n}{n}\right) \end{aligned}$$

Therefore for any node v in $S_i - T_j$ the number of active arcs to $T_i - S_j$ is $O(\frac{\sqrt{n \log n} \log n}{n})$. There are $4\sqrt{n \log n}$ nodes in the source tree. Therefore the total number of potential updates to L_2 is

$$O(\frac{\sqrt{n \log n} \log n}{n}) O(\sqrt{n \log n}) = O(\log^2 n).$$

□

Hence the total number of updates for all source-sink pairs is $O(n_O n_D \log^2 n)$.

We now show quote a theorem of Grimmett and Frieze [6] that shows that even though we considered only the active arcs, the probability that the shortest paths computed is wrong is $O(\frac{1}{n})$. For this lemma we assume that that the arcs lengths are independent identically distributed random variables with a cumulative density function $F(x)$. (The exponential distribution satisfies this assumption). Let

$$d^* = \max(d(i,j) : i \in O, j \in D).$$

Let

$$M = \min(c_{ij} : (i,j) \text{ is not active}).$$

Theorem 18 (Grimmett and Frieze [6]) *If $F(x)$ is differentiable at 0 and $F'(0) > 0$ then*

$$Pr\,[d^* > M] = O(\frac{1}{n})$$

Proof:

For each node i scanned by the algorithm, let $\gamma_i = \min(c_{ij} : (i,j)\text{is inactive})$. To verify the optimality of the algorithm it suffices to check if $\gamma_i > d^*$ for each i. If $\gamma_i < L$ for some i, then one can solve n_O single source shortest path problems to solve the O-D shortest path problem. The expected running time is $O(n_O m)$ which is less than the running time of all the other steps.

Therefore we see that this algorithm has a running time of $O(n_O n_D \log^2 n + \min(n_D\sqrt{n \log n}\ m/n, m + n_D\sqrt{n \log n} \log n))$. At the end of the execution of the algorithm, if some $L(i,j)$ is infinite (which occurs with probability $O(\frac{1}{n^2})$) or if the test given in Lemma 9 fails (which occurs with probability $O(\frac{1}{n})$) we run n_O single source problems. The time for the composite algorithm is still $O(n_O n_D \log^2 n + \min(n_D\sqrt{n \log n}\ m/n, m + n_D\sqrt{n \log n} \log n))$.

We extend the same type of analysis to some other non-exponential distributions in the following section. This completes the running time analysis of the O-D shortest path problem on complete graphs with i.i.d exponentially distributed weights.

5 Extension to Non-Exponential Distributions

The results of the previous sections can be extended to non-exponential random variables using the concept of c_b-boundedness as in Luby and Ragde [8]. The graph is still assumed to be complete.

5.1 c_b-Bounded Random Variables

Let $f(x)$ be the probability density function of a random variable and let its cumulative density function be $F(x)$. We define the hazard rate of this probability density function as follows:

$$\gamma(x) = \frac{f(x)}{1 - F(x)}.$$

This is just the probability density that the random variable takes on the value in the interval $[x, x + \delta x]$ as δx approaches 0, given that the random variable is at least as large as x.

Definition 1 *A distribution function is c_b-bounded on $[0, \delta]$ if*

$$\frac{\gamma(x)}{\gamma(y)} \leq c_b \qquad \forall x, y \quad F(x), F(y) \leq \delta.$$

We also say that a random variable is c_b-bounded on $[0, z]$ if its distribution function is c_b-bounded on $[0, z]$. We omit the reference to $[0, \delta]$ and say that a random variable is c_b-bounded if it is c_b-bounded on $[0, 1]$. In the next two lemmas we give examples of c_b-bounded random variables.

Lemma 19 *The exponential random variable with parameter λ is 1-bounded on $[0, 1]$.*

Proof:
In the case of the exponential distribution with parameter λ,

$$f(x) = \lambda e^{-\lambda x}$$

and

$$F(x) = 1 - e^{-\lambda x}.$$

Therefore,

$$\gamma(x) = \frac{f(x)}{1 - F(x)} = \lambda \quad \forall \ x$$

and

$$\frac{\gamma(x)}{\gamma(y)} \leq 1 \qquad \forall x, y \quad F(x), F(y) \leq 1.$$

□

Lemma 20 *The uniformly distributed random variable between 0 and 1 is 2-bounded on $[0, \frac{1}{2}]$.*

Proof:
In the case of the uniform random variable wbetween 0 and 1,

$$f(x) = 1 \quad x \in [0,1]$$

and

$$F(x) = x \quad x \in [0,1].$$

Therefore,

$$\gamma(x) = \frac{f(x)}{1 - F(x)} = \frac{1}{1-x} \quad \forall\ x$$

and

$$\frac{\gamma(x)}{\gamma(y)} = \frac{1-y}{1-x} \leq \frac{1}{1-x} \leq 2 \qquad \forall x, y \quad F(x), F(y) \leq \frac{1}{2}.$$

□

For the rest of the section we consider random variables that are c_b-bounded (on $[0,1]$) . We use the property of c_b-boundedness to get bounds on the probability of certain events that are used in the running time analysis of the O-D shortest path algorithm. We then show that the expected running time of the O-D shortest path algorithm is the same as in the exponential case if the arcs length are chosen from a distribution that is c_b-bounded. We give the following preliminary results towards this end.

Each step of the algorithm involves choosing one of the arcs out of a set of enabled arcs. In the next lemma we bound the probability that a particular arc is chosen in terms of the probability of any other arc being chosen.

Lemma 21 (Luby and Ragde) *Let the arc lengths be c_b-bounded random variables. Let $E = \{e_1, e_2, \ldots, e_k\}$ be the set of active arcs. For each i, let P_i be the probability that arc e_i is revealed first.*

$$P_i \leq c_b P_j \qquad \forall\ i, j.$$

This lemma is proved in Luby and Ragde. In the case of the exponential distribution, all the S-enabled arcs are equally likely to be chosen. The previous lemma indicates that in the case of c_b-bounded random variables the probability of an arc being revealed is within a constant factor of any other arc being revealed. Using this lemma, we can obtain lower and upper bounds on the probability of the selected arc being in a particular subset of the active arcs.

Lemma 22 *Let F denote the set of S-enabled arcs, and let E' denote some specified subset of the arcs of E. Let $k = |E|$ and let $k' = |E'|$. If the arc lengths are c_b bounded then*

$$\frac{1}{c_b}\frac{k'}{k} \leq P(E') \leq c_b \frac{k'}{k}.$$

Proof:
Let P_i be the probability that arc e_i is picked first. Let $P_{max} = \max_{j \in E} P_j$ and $P_{min} = \min_{j \in E} P_j$. By Lemma 20 and since $\sum_{j \in E} P_j = 1$, $P_{max} \leq \frac{c_b}{k}$ and $P_{min} \geq \frac{1}{c_b k}$. Therefore,

$$P(E') = \sum_{i \in E} P_j \leq k' P_{max} \leq \frac{k' c_b}{k}.$$

$$P(E') = \sum_{i \in E} P_j \geq k' P_{min} \geq \frac{1}{c_b} \frac{k'}{k}.$$

□

Lemma 23 *The probability that the source tree and the sink tree intersect after* $4\sqrt{c_b n \log n}$ *steps is greater than* $\frac{1}{n^4}$.

Proof:
Let us assume that Phase 1 of the algorithm is executed from the source node and the sink node for $4\sqrt{c_b n \log n}$ steps. The proof is identical to the exponential case except that in equation 1 in lemma 6 the probability that a cross arc being chosen at stage k is greater than $\frac{1}{2c_b k}$. Let C be the event that the source tree and the sink tree do not intersect in $4\sqrt{c_b n \log n}$ steps. Then,

$$\begin{aligned} Pr[C] &\leq \prod_{i=1}^{4\sqrt{c_b n \log n}} (1 - \frac{1}{2c_b i}) \\ &\leq e^{-\frac{1}{2c_b}(\sum_{i=1}^{4\sqrt{c_b n \log n}} i)} \\ &\leq e^{-\frac{16 c_b}{4 c_b}(\log n)} \\ &\leq \frac{1}{n^4}. \end{aligned}$$

□

The proof of the next three lemmas is similar to the exponential case and therefore we only give a sketch of the proof.

Lemma 24 *Let the arc lengths be* c_b*-bounded random varables. Let* $H_k(t)$ *be the probability that a randomly picked node is at level* t *when we are at stage* k *of the tree. Then*

$$H_k(t) \leq \frac{k-1}{k} H_{k-1}(t) + \frac{c_b}{k} H_{k-1}(t-1).$$

Proof:
The proof is the same as in lemma 8 except that in equation 8 we replace the exact probability by a bound given by the c_b bounded random variable.

□

Lemma 25 *Let*

$$\phi(H_k) = \sum_{t=0}^{\infty} Z^t H_t(k).$$

Then $\phi(H_k) \leq \frac{1}{i!} \prod_{i=1}^{k-1} (c_b Z + i)$ *if the arc lengths are* c_b*-bounded random variables.*

Proof:
Multiply both sides of the equation by Z^k and sum over k as in lemma 9 to get the result.

□

Lemma 26 *Let the arc lengths be* c_b *bounded random variables. Let* E *be the event that at the end of* $4\sqrt{c_b n \log n}$ *steps, a randomly picked node is at a distance greater than* $(3c_b + 3)\log n$ *distance from the source. Then*

$$Pr[E] \leq \frac{1}{n^3}.$$

Proof:
We replace the constant 6 in equation 2 in lemma 11 to get the result . □

Lemma 27 *For any sink tree,*

$$Pr[\textit{All nodes are at a distance} < (3c_b + 3)\log n] \geq 1 - \frac{1}{n^3}.$$

Proof:
Follows directly from the previous lemma. □

Theorem 28 *The expected number of potential updates to* $L_1(i,j)$ *is* $O(\log^2 n)$ *and the expected number of potential updates to* $L_2(i,j)$ *is* $O(\log^2 n)$.

Proof:
Since the length of the paths is $O(\log n)$ with high probability, the proof of this lemma is the same as lemmas 16 and 17.

□

Therefore, all the results of the previous section carry over to the case where the random variables are independent identically distributed random variables that are c_b-bounded.

Till this point we have assumed that the length of the arcs are independent, identically distributed random variables which are c_b-bounded on $[0,1]$. In the next two lemmas we shall see that since we examine only short arcs during the execution of the algorithm, it is sufficient if the random variable is c_b-bounded in a region close

to the origin. The idea is the following: Since we examine only the shortest $20 \log n$ arcs out of every node, the Distribution function value corresponding to the length of all the arcs that are active is close to the origin. In the next two lemmas we show that with high probability we examine arcs whose length l has the property that

$$F(l) \leq \frac{35 \log n}{n}.$$

Lemma 29 *Let $G_{p,n}$ be a random graph on n nodes with the probability of an arc being present equals p. If $p \geq \frac{35 \log n}{n}$, there are more than $20 \log n$ arcs entering and leaving every node with probability not less than $1 - \frac{1}{n^2}$.*

Proof:
The probability that any arc (i, j) is in the graph is $\frac{35 \log n}{n}$. Consider a fixed node v in the graph. Define an indicator random variable X_i that takes on value one with probability $p = 35\frac{\log n}{n}$. There are $n - 1$ such indicator random variables at node v. Let $X = \sum_{i=1}^{n-1} X_i$. We define a new random variable

$$Y_i = Z^{-X_i} \qquad i = 1, \ldots, (n-1), Z > 1.$$

Therefore the

$$E[Y_i] = (1 - p) + \frac{p}{Z} \qquad i = 1, \ldots, (n-1).$$

By a variant of the Chernoff bound (lemma 10) we can show that

$$\begin{aligned} Pr[X < a] &\leq E[Z^{-X}]Z^a \\ &= \left(1 - p + \frac{p}{Z}\right)^{n-1} Z^a. \end{aligned}$$

We will determine the probability that there are less than $20 \log n$ arcs out of node v. We set $p = \frac{35 \log n}{n}$ and $Z = e$ in the previous equation.

$$\begin{aligned} Pr[X < 20 \log n] &\leq \left(1 - p + \frac{p}{e}\right)^{n-1} e^{20 \log n} \\ &\leq \left(1 - \frac{35 \log n}{n}(1 - \frac{1}{e})\right)^{n-1} n^{20} \\ &\leq e^{-35(1-\frac{1}{e})\frac{\log n}{n}(n-1)} n^{20} \\ &\leq e^{-35(1-\frac{1}{e}) \log n} n^{20} \\ &< \frac{1}{n^2}. \end{aligned}$$

This implies that if $p = \frac{35 \log n}{n}$, then all nodes have more than $20 \log n$ arcs with probability greater than $1 - \frac{1}{n}$.

□

We remark that $\frac{35 \log n}{n}$ is needed for the proof, although a tighter bound is possible.

Definition 2 *Let α-short arcs be those arcs whose length has the following property:*

$$F(l) \leq \frac{35 \log n}{n}.$$

Lemma 30 *The algorithm examines only α-short arcs with probability greater than $\frac{1}{n}$.*

Proof:
We showed in lemma 3 that the number of arcs examined in Phase 1 and Phase 2 at every node is $20 \log n$. We know from lemma 29 that with high probability that these arcs are α-short arcs.

□

Lemma 31 *Lemmas 24,25,26, 27 and theorem 28 still hold if the distribution is c_b-bounded on $\left[0, \frac{35 \log n}{n}\right]$.*

Proof:
By lemma 30, the only arcs examined are α-short arcs, we need c_b-boundedness for these arcs only.

□

6 Extension to Sparse Random Graphs

In this section we will consider sparse random graphs in which the average degree $d(n)$ of node n is not less than $35 \log n$. Let us consider random graphs where the average degree of a node is $d(n)$.

Lemma 32 *The results of the previous section holds if the distribution is c_b-bounded on $\left[0, \frac{35 \log n}{d(n)}\right]$.*

Proof:

Let us consider random graphs where the average degree of node n is $d(n)$. Let the length of the arc be chosen from a probability density function $f(x)$. At every arc with probability $\frac{d(n)}{n-1}$ we choose a value from this probability density function and with probability $1 - \frac{d(n)}{n-1}$ we choose infinity. Let us call this modified probability density function $f'(x)$ with the corresponding distribution function $F'(x)$. The average number of finite arcs out of any given node is $d(n)$. This is the average degree of the node of the sparse random graph. Since the algorithm looks at only $20 \log n$ arcs out of every node, if we can show that there are $20 \log n$ finite arcs out of every node with very high probability, we will be done. Further, the modified probability density function also satisfies the conditions of lemma 3 if the original

p.d.f did and therefore the testing can be done in $O(1)$ time. From lemma 29 we know that if $d(n) \geq 35 \log n$ we have more than $20 \log n$ arcs out of every node with high probability.

From lemma 29, we know that the algorithm examines only α-short arcs. This applies to the distribution function $F'(x)$. In this case we define the α-short arcs to be arcs that have

$$F'(l) \leq \frac{35 \log n}{n}.$$

If this is the case then we transform this condition to one that involves $F(l)$. In the initial section of the modified distribution function,

$$F'(l) = F(l)\frac{d(n)}{n-1}.$$

Therefore,

$$\begin{aligned} F(l)\frac{d(n)}{n-1} &\leq \frac{35 \log n}{n} \\ F(l) &\leq \frac{35 \log n}{d(n)} \end{aligned}$$

Therefore it is sufficient if the random variables are c_b-bounded in $\left[0, \frac{c_r \log n}{d(n)}\right]$. □

7 Parallel Algorithm

In this section, we will give a parallel implementation of the O-D shortest path algorithm developed in Sections 1 through 6. We will first give a quick review of the different steps of the algorithm and then show how to implement the steps in parallel. We shall then do a running time analysis of the parallel algorithm. The different steps of the O-D shortest path algorithm are the following:

- For each source node determine the closest $4\sqrt{n \log n}$ nodes.
- For each sink node determine the closest $4\sqrt{n \log n}$ nodes.
- Patching up phase.
- Verification phase.
- Backup algorithm if necessary.

The verification phase is trivial and can be done in $O(1)$ time since it is merely a comparison of two numbers. The backup algorithm is to run Dijkstra's algorithm in parallel in $O(n \log n)$ time and contributes only $O(\log n)$ to the expected running time since the backup algorithm is run with probability $O(\frac{1}{n})$. Therefore, the only steps that need detailed analysis are the tree growing and the patching up phases. In the next two sections we will analyse these two steps in more detail.

7.1 Tree Generation

The generation of the source tree and the sink tree are similar. We shall give a generic procedure for generating a shortest path tree. This procedure can be used to generate the source trees and the sink trees.

Consider a root node s. Let $f^k(j)$ be the distance from the root node to node j using at most k arcs. Let $f(j)$ be the shortest path distance from s to j. It is well known that $f^k(j)$ satisfies the following recursion:

$$f^k(j) = \min\{f^{k-1}(j), \min_{l \in V} f^{k-1}(l) + c_{lj}\}.$$

The dynamic programming recursion that we use to determine the shortest path tree is the following:

$$d^k(j) = \min\{d^{k-1}(j), \min_{l \in B_{k-1}} d^{k-1}(l) + c_{lj}\}.$$

where B_k is the smallest $4\sqrt{n \log n}$ values of $d^k(j)$.

We set the value of $d^0(j)$ to infinity initially for all nodes. The previous equation states that the shortest path between the source node and node j having at most k arcs is the minimum of the shortest path having at most $k-1$ arcs and the shortest path having at most $k-1$ arcs to some node l in B_{k-1} with an extra arc (l, j) from that node to j. We define

$$g_k = \max_{j \in B_k} d^k(j).$$

Therefore g_k is the maximum distance to a node in B_k.

We execute this recursion till for some k , the set B_k is the same as set B_{k-1} and for all nodes $j \in B_k$,

$$d^{k-1}(j) = d^k(j).$$

The set B_k does not change subsequently. In the next theorem we prove that this recursion indeed determines the closest $4\sqrt{n \log n}$ nodes to the source node.

Theorem 33 *Let $d^k(j), f^k(j)$ and B_k be defined as above. Then*

$$d^k(j) = f^k(j) \quad j \in B_k$$

and

$$f^k(j) \geq g_k \quad j \notin B_k.$$

Further let k' be the depth of the shortest path tree. Then

$$d^{k'}(j) = f(j) \quad j \in B_{k'}.$$

Proof:
Let s be the root node from which we want to determine the closest $4\sqrt{n \log n}$ nodes. We will prove that this recursion works by induction on k. Set $B_0 = \{s\}$. In step 1 we take the closest $4\sqrt{n \log n}$ nodes to the root node using exactly one arc. Therefore, when $k = 1$

$$d^1(j) = f^1(j) \qquad j \in B_1$$

and $f^1(j) \geq g_1 \qquad j \notin B_1$. Let us assume that the recursion is true for some k. Therefore,

$$d^k(j) = f^k(j) \qquad j \in B_k$$

and

$$f^k(j) \geq g_k \qquad j \notin B_k.$$

We will prove that the statements hold for $k+1$. First we will prove that for $v \in B_{k+1}$ the following statement is true:

$$d^{k+1}(v) = f^{k+1}(v).$$

It can be proved easily by induction that $d^k(j) \geq f^k(j)$ for all k and j. In other words, the $d^k(j)$ is an upper bound bound on the value $f^k(j)$. We will prove that for $v \in B_{k+1}$,

$$d^{k+1}(v) \leq f^{k+1}(v).$$

For proving this we use the induction hypothesis along with the following (obvious) results:

$$g_k \geq d^k(j) \qquad j \in B_k.$$

$$g_k \geq g_{k+1} \qquad \forall k.$$

$$g_k \leq d^k(j) \qquad j \notin B_k.$$

We will split the proof into two cases and prove each case separately.
CASE 1. Let us assume that $f^{k+1}(v) = f^k(v)$.
If $v \in B_k$ then

$$d^{k+1}(v) \leq d^k(v) = f^k(v) = f^{k+1}(v).$$

If $v \notin B_k$ then

$$d^{k+1}(v) \leq g_{k+1} \leq g_k \leq f^k(v) = f^{k+1}(v).$$

CASE 2. Let $f^{k+1}(v) = f^k(i) + c_{iv}$.
If $i \in B_k$ then

$$d^{k+1}(v) \leq d^k(i) + c_{iv} = f^k(i) + c_{iv} = f^{k+1}(v).$$

If $i \notin B_k$ then

$$d^{k+1}(v) \leq g_{k+1} \leq g_k \leq f^k(i) \leq f^k(i) + c_{ij} = f^{k+1}(v).$$

This therefore proves the first part of the induction hypothesis.
We will now prove the second part of the induction hypothesis, i.e.,

$$f^{k+1}(v) \geq g_{k+1} \qquad v \notin B_{k+1}.$$

We will split the proof into two cases and prove each case separately as in the last statement.

CASE 1. For some $v \notin B_{k+1}$, let us assume that $f^{k+1}(v) = f^k(v)$.
If $v \in B_k$ then

$$f^{k+1}(v) = f^k(v) = d^k(v) \geq d^{k+1}(v) \geq g_{k+1}.$$

If $v \notin B_k$ then

$$f^{k+1}(v) = f^k(v) \geq g_k \geq g_{k+1}.$$

CASE 2. For some $v \notin B_{k+1}$, let us assume that $f^{k+1}(v) = f^k(i) + c_{iv}$.
If $i \in B_k$ then

$$f^{k+1}(v) = d^{k+1}(v) \geq g_{k+1}.$$

If $i \notin B_k$ then

$$f^{k+1}(v) \geq f^k(i) \geq g_k \geq g_{k+1}.$$

Therefore the algorithm determines the shortest $4\sqrt{n \log n}$ nodes.
Note that

$$f^{k'}(v) = f(v) \qquad v \in B_{k'}.$$

By the induction proved earlier

$$f^{k'}(v) = d^{k'}(v) \qquad v \in B_{k'}.$$

Therefore $B_{k'} = B_{k'+1} = \ldots = B_n$ and we are done.

□

Since the shortest path to the closest $4\sqrt{n \log n} - 1$ nodes has at most $4\sqrt{n \log n} - 1$ arcs along the shortest path, we will be done in $4\sqrt{n \log n} - 1$ steps. Note that if the set B_k does not change over one complete iteration, i.e., $B_k = B_{k+1}$ and for all $j \in B_k$ the distance labels $d^k(j) = d^{k+1}(j)$ then we can stop the execution of the algorithm as neither the set nor the distance labels change beyond this step. This will be the termination criterion.

This algorithm in the worst case takes $O(\sqrt{n \log n})$ iterations to determine the closest $4\sqrt{n \log n}$ nodes to the root node. In the next lemma we show that on the average it takes $O(\log n)$ iterations before termination.

Lemma 34 *The expected number of iterations to determine the closest* $4\sqrt{n \log n}$ *nodes is* $O(\log n)$.

Proof:
Note that the number of steps that we run the dynamic programming recursion is the depth of the shortest path tree by the previous lemma. From lemma 11 we know that the depth of the shortest path tree is less than $6 \log n$ with probability greater than $1 - \frac{1}{n^2}$. Therefore the expected depth of the tree $O(\log n)$, and the expected number of iterations to run the dynamic programming algorithm to termination in $O(\log n)$.

□

Another modification to the tree generation procedure which does not improve the time bound but reduces the expected number of processors is the following: Since we know that only the shortest $20 \log n$ arcs out of every node is in the shortest path tree with high probability, we consider only the shortest $20 \log n$ arcs out of every node during the execution of the algorithm. We first consider the case that the arcs are given in sorted order. We may select these arcs and assign $20 \log n$ processors to the first $20 \log n$ arcs per node in $O(\log n)$ time. To evaluate $c_{ij} + d^{k-1}(j)$ takes $O(1)$ time for the processor assigned to arc (i, j). To compute all the minimum in parallel takes $O(\log n)$ time and $80\sqrt{n \log n} \log n$ processors since there are $4\sqrt{n \log n}$ nodes in B_{k-1} and each node has $20 \log n$ arcs emanating from it. We now consider the case in which the arcs are not sorted. We may select the $20 \log n$ shortest arcs from each node in $O(\log n)$ time using $d = m/n$ processors per node on average. Therefore, the running time in both these cases is $O(\log n)$ per iteration and the expected number of iterations is $O(\log n)$ giving an overall running time of $O(\log^2 n)$. The expected number of processors in the first case is $O(\sqrt{n \log n} \log n)$ and in the second case is $O(\sqrt{n \log n} \frac{m}{n})$.

7.2 Patching up

We shall see how to implement the patching up phase efficiently in parallel. Recall that in the sequential algorithm we kept a list of all sink trees and source trees containing some node u. These lists can also be formed in parallel in $O(\log n)$ time and with $O(n_D \sqrt{n \log n})$ processors. Let us say that for some node u these lists are $L_T(u)$ and $L_S(u)$ respectively. The lists will lead to $|L_S(u)| \times |L_T(u)|$ updates of L_1, one for each source tree and sink tree pair containing node u. We assign $|L_S(u)| \times |L_T(u)|$ processors to make these updates. The expected total number of processors needed is the expected total number of intersections which, by lemma 16 is $O(n_O n_D \log^2 n)$. These assignments can be made in parallel in $O(\log n)$ steps. Once the assignments are made, the updates to L_1 can be done in $O(\log n)$ time. Thus the total time to update L_1 is $O(\log n)$.

The updates to L_2 are carried out in a similar fashion. We first determine the potential cross arcs. For each node u in some source tree we scan the $20 \log n$ shortest arcs out of u and determine all arcs (u, v) such that v is in some sink tree. Once the cross arcs have been determined, for each cross arc (u, v) we let $L_S(u)$ be the number of source trees containing u and $L_T(v)$ be the set of sink trees containing v.

We then assign $|L_S(u)| \times |L_T(v)|$ processors to the cross arc (u, v) to perform the updates to L_2. The assignment of the processors takes $O(\log n)$ time. By lemma 17 we know that the expected number of processors required is $O(n_O n_D \log^2 n)$. The total time to upate L_2 is $O(\log n)$.

Therefore from the analysis of the algorithm we see that the expected running time of the algorithm is $O(\log^2 n)$ and the expected number of processors required is $O(n_D\sqrt{n \log n} \log n + n_O n_D \log^2 n)$ if the arcs are in sorted order and the expected number of processors is $O(n_D\sqrt{n \log n}\frac{m}{n} + n_O n_D \log^2 n)$ if the arcs are not in sorted order.

8 Conclusion

We have developed serial and parallel algorithms for the $O-D$ shortest path problem and analysed its average case behavior. We show that the average running time of these algorithms on random graph models are sub-linear for a wide range of input values. These algorithms are easy to implement and should perform well in practice.

9 Acknowledgements

Support from Air Force grant AFOR-88-0088, National Science Foundation grant DDM-8921835, and a grant from UPS is gratefully acknowledged. The first author also wishes to thank Hanan Luss and Beth Munson for all the support during the writing of the paper.

References

[1] Coppersmith, D., and Winograd, S., "Matrix Multiplications via Arithmetic Progressions", *Proceedings of the Ninteenth Annual ACM Symposium on Theory of Computing*, 1987, pp. 1-6

[2] Deo, N., and Pang, C.,"Shortest-Path Algorithms: Taxonomy and Annotation", *Networks 14*, 1984, pp. 275-323

[3] Erdos, P., and Renyi, A., "On the strength of Connectedness of a Random Graph", *Acta Math. Acad. Sci. Hungar. 12*, 1961, pp. 261-267

[4] Fredman, M.L., and Tarjan, R.E., "Fibonacci Heaps and its uses in Improved Network Algorithms", *Journal of the ACM 34(3)*, 1987, pp. 596-615.

[5] Fredrickson, G.N., "Fast Algorithms for Shortest Path in Planar Graphs, with Applications", *SIAM J. 16*, 1987, pp. 1004-1022

[6] Frieze, A., and Grimmett, G., "The Shortest-Path Problem for Graphs with Random Arc-lengths", *Discrete Applied Math. 10,* 1985, pp. 57-77

[7] Hassin, R., and Zemel, E., "On Shortest Paths in Graphs with Random Weights", *Math. of Operations Research 10*, 1985, pp. 557-564

[8] Luby, M. and Ragde, P. , "A Bidirectional Shortest-Path Algorithm with Good Average-Case Behavior",*Algorithmica 4*, 1989, pp. 551-567

[9] Sedgwick, R. and Vitter, J. , "Shortest Path in Euclidean Graphs",*Algorithmica 1*, 1986, pp. 31-48

[10] Steebrink, P.A, "Optimization of Transport Network", *John Wiley and Sons*, 1973

[11] Van Vliet, D., "Improved Shortest Path Algorithms For Transport Networks", *Transportation Research 12,* 1978, pp. 7-20