Routing Strategies for Fast Networks [*]

Yossi Azar [†] *Joseph (Seffi) Naor* [‡] *Raphael Rom* [§]

Abstract

Modern fast packet switching networks forced to rethink the routing schemes that are used in more traditional networks. The reexamination is necessitated because in these fast networks switches on the message's route can afford to make only minimal and simple operation. For example, examining a table of a size proportional to the network size is out of the question.

In this paper we examine routing strategies for such networks based on flooding and predefined routes. Our concern is to get both efficient routing and an even (balanced) use of network resources. We present efficient algorithms for assigning weights to edges in a controlled flooding scheme but show that the flooding scheme is not likely to yield a balanced use of the resources. We then present efficient algorithms for choosing routes along: (i) bfs trees; and (ii) shortest paths. We show that in both cases a balanced use of network resources can be guaranteed.

Index terms: routing strategies, controlled flooding, network resources, load balancing, shortest paths, bfs trees, conditional probabilities.

[*]Preliminary version of this paper appeared in the Proceedings of the 11th IEEE INFOCOM, Florence, Italy, (1992), pp. 170-179.

[†]Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel 69978. E-mail: azar@math.tau.ac.il

[‡]Computer Science Department, Technion, Haifa, Israel 32000. E-mail: naor@cs.technion.ac.il

[§]Department of Electrical Engineering, Technion, Haifa, Israel 32000, and Sun Microsystems, Mountain View, CA 94043-1100. E-mail:rom@ee.technion.ac.il

1 Introduction

Traditional computer networks were designed on the premise of fast processing capability and relatively slow communications channels. This manifested itself by burdening network nodes with frequent network management decisions such as flow control and routing [1, 2, 3]. In a typical packet-switching network the routing decision at every node is based on the packet's destination and on routing information stored locally. This routing information may become quite voluminous, increasing the per-packet processing time.

Changes in technology, applications, and network sizes have forced to rethink these strategies. Modern fast packet switching networks [4, 5] relegate most of the routing computation to the end-nodes leaving all but the minimal computation to the intermediate nodes once the packet is on its way. This paper considers and compares several routing strategies for such fast networks. We assume that links are of high capacity so that message length is of no great concern. Computation capability in intermediate nodes is assumed limited so that all decisions made enroute should be simple and could not rely, for example, on generating random numbers or on tables that grow with the size of the network.

The first to encounter similar problems were the designers of parallel computers. Their solution, in the form of an interconnection network, typically derives the route directly from the destination address [6]. This approach, however, is limited to specific types of network topology and a structured layout which cannot be assumed for a general network. Furthermore, deriving the route from the address in general conflicts with alternate routing approach.

Flow-based techniques, used in many existing networks [7, 8], are also inadequate for our environment. These routing strategies are destination based (typically require a table entry per destination) but more importantly, result in bifurcated routing necessitating intermediate nodes to generate random numbers.

Two strategies are considered in this paper – controlled flooding and fixed routing. Flooding is a routing strategy that guarantees fast arrivals with minimal enroute computation at the expense of excessive bandwidth use. The scheme we use here, first proposed in [9], limits the extent to which a message is flooded through the network. Essentially, each link is assigned a *weight* (also known as its *cost*) and every message carries with it a *wealth*. A message arriving at an intermediate node will be duplicated and forwarded along all outgoing links (except the one it came from) whose cost is lower than the message wealth. The cost of the link is then deducted from the message wealth. The problem is to assign the link costs so as to achieve best performance. In [9] it is shown that for such a scheme to be optimal the shortest path between every pair of nodes (based on link costs) must be unique. We show two methods of computing optimal weights that are drawn from a polynomial range (as opposed to the exponential range proposed in [9]). However, we do show that the assignment does not result in a routing scheme that uses network resources in a balanced way.

In the fixed routing scheme the route of the message is determined at the source node and is included in the message. No further routing decision are done enroute. The problem is therefore to find a set of routes, one for each pair of nodes, such that all the network's links will be used in a balanced manner. We propose two methods to achieve this. In the first one, we force the messages to be routed along a (topological) breadth first search tree. The problem can be formulated as finding a set of rooted BFS trees such that the maximum load on a link is minimized (messages are routed along the tree toward the root as a destination). Notice that no link in the network remains unused. We provide polynomial algorithms to generate such a set of balanced routes.

In the second method, routing is done along paths that do not necessarily form trees. One of the shortest paths between every pair of nodes is designated as the path along which these two nodes exchange messages. We prove that a set of paths can be chosen that yields a balanced load. We define the notion of a balanced load with respect to randomized choices of paths, i.e., every pair chooses uniformly in random one of the shortest paths connecting them. We first show that with high probability the load on every edge will be close to its expected value. We then show how to construct deterministically in polynomial time such a set of balanced paths via the method of conditional probabilities.

2 Assigning Weights for Controlled Flooding

In this section we focus on the controlled flooding scheme and address the problem of assigning weights to the links.

Since the controlled flooding scheme is a derivative of a flooding algorithm, it is impossible to assure that a message always arrives only at the nodes it is intended to. In particular, when used for point-to-point routing it is evident that more nodes than necessary might receive a message. Clearly, different weight assignments may change the pattern of flooding. Thus, to find an optimal assignment a figure of merit is defined which is proportional to the (average) number of nodes that will receive every message. An optimal weight assignment is one that minimizes the figure of merit. To formalize our discussion let the network be represented by the graph $G(V, E)$ with $|V| = n$ and $|E| = m$, let the length of a path in the network be defined as the sum of the weights of the edges of the path, and let the shortest path between two nodes be the path with minimal length. Then, it is shown in [9] that for an assignment to be optimal, the following requirements (referred to as *optimality* requirements) must hold for every vertex (node) r:

- For every vertex $v \in V$, the shortest path from r to v is unique.
- For any two vertices $u, v \in V$, the length of the shortest path from r to u is different from the length of the shortest path from r to v.

Assignments that satisfy the above requirements are called *good*. An assignment is good with respect to r if all shortest paths from r satisfy the above requirements. Let us assume without loss of generality that the weights assigned are all positive integers.

Let $[1 \dots R]$ denote the range of numbers from which weights are drawn and let n denote the number of nodes in the network. If $R = 2^{|E|}$, it is easy to find a good assignment. For example, assigning 2^i as the weight of edge e_i assures that any two different paths will have different lengths. However, because the length of the path is carried by every message it is desirable to reduce R as much as possible.

We present two methods for constructing good assignments such that R is polynomial in n. In the first method the communication is restricted to a spanning tree T of the graph. This is done by assigning infinite weight to edges that are not in the tree. Denoting the tree edges by $e_1, \dots e_i \dots$, the algorithm is recursively defined as follows. Let v_l be a leaf of T, let u_l be its neighbor in the tree, and let e_l be the edge connecting u_l and v_l.

1. Compute (recursively) a good assignment for the tree $T - v_l$.
2. Extend the good assignment from $T - v_l$ to T.

We assume inductively that a good assignment was computed in Step 1. Step 2 can be implemented by checking all the values in the range $1 \dots R$ and finding one that satisfies the requirements for a good assignment. Obviously, a good value for e_l exists if R is large enough. The next lemma bounds the value of R.

Lemma 2.1 *If $R \geq n^2$, then there exists a good assignment.*

Proof: Since a good assignment was computed for $T - v_l$ at Step 1, any value assigned to e_l will complete a good assignment with respect to v_l. The number of distinct values that e_l cannot assume is at most $(n-1)(n-2)$: for each vertex $r \in T - v_l$, the distance from r to v_l should be different from the distance from r to any other vertex, and thus, there can be at most $n-2$ forbidden values (with respect to r), and the claim follows. □

The complexity of the algorithm is $O(n^3)$ since each step can be implemented in $O(n^2)$ time. For each vertex $v_i \in V$, a table of all its distances to the other vertices is maintained and for each node all the forbidden values in the range $[1 \dots n^2]$ are marked. One of the unmarked numbers is chosen arbitrarily for e_l. Then, the tables of all other nodes are updated.

The above assignment, being tree based, makes no use of many of the network links. The second assignment, which we present next, has the property that the whole network participates in the communication. We present two algorithms; the first is a randomized one that lends itself to distributed computation because the weight for each edge is chosen independently of the other edges. This algorithm generates a good assignment with high probability. The second algorithm is deterministic, and the weights are chosen from a smaller range than in the randomized algorithm.

Our main tool in the randomized case is the *Isolating Lemma* of Mulmuley, Vazirani and Vazirani [10]. A set system (S, F) consists of a finite set S of elements, $S = \{x_1, \ldots, x_n\}$, and a family F of subsets of S, $F = \{S_1, \ldots, S_k\}$. Let a weight w_i be assigned to each element of S. The weight of a subset is defined to be the sum of the weights of its elements.

Lemma 2.2 (Isolating Lemma) *Let $R \geq n$ and let (S, F) be a set system whose elements are assigned integer weights chosen uniformly and independently from the range $[1 \ldots R]$. Then,*

$$\text{Prob}[\textit{There is a unique minimum (maximum) weight set in } F] \geq 1 - \frac{n}{R} \quad .$$

(Note: the lemma in its original form in [10] was proven for $R = 2n$ but actually holds for all $R \geq n$). □

We start by proving that the following randomized process will generate a good assignment with high probability. Let a weight for each edge be chosen randomly and uniformly from the range$[1 \ldots R]$.

Lemma 2.3 *For $R \geq n^5$ the probability that an assignment is good is at least $\frac{1}{2}$.*

Proof: Let A_{ij} be the event *the shortest path between nodes v_i and v_j is not unique.* Then $A = \cup_{i,j} A_{ij}$ is the event indicating the existence of at least one pair of nodes with non-unique shortest path between them. For each pair of nodes v_i and v_j let the set system F be the set of all paths connecting them. From the isolating lemma we have that the shortest path between them will be unique with probability at least $1 - \frac{m}{R}$, or, $\text{Prob}[A_{ij}] \leq \frac{m}{R}$. Hence, $\text{Prob}[A] \leq \sum_{i,j} \text{Prob}[A_{ij}] \leq \binom{n}{2} \cdot \frac{m}{R}$.

Let B_{ijk} represent the event that nodes v_i, v_j, and v_k form a bad triplet, namely that the length of the shortest path between v_i and v_k equals that between v_j and v_k. $B = \cup_{ijk} B_{ijk}$ then represents the existence of at least one bad triplet in the network. In a way similar to the above we get $\text{Prob}[B] \leq \binom{n}{3} \cdot \frac{m}{R}$.

Finally, $A \cup B$ is the event indicating that the requirements are <u>not</u> met, and thus

$$\text{Prob}[\textit{good assignment}] \geq 1 - Prob[A] - Prob[B] \geq 1 - \frac{mn(n-1)}{2R} - \frac{mn(n-1)(n-2)}{6R} \quad .$$

Since $m \leq n^2$, for $R \geq n^5$, the right handside exceeds $\frac{1}{2}$. □

The last lemma provides us with a randomized distributed algorithm for constructing a good assignment. The probability of failure can be made arbitrarily small by increasing the value of R.

Notice that this method does not ensure that every edge participates in at least one shortest path. This can be fixed by forcing the weight assignment so that the BFS tree resulting from the weight assignment is also a BFS tree in the underlying graph without weights. This can be done in the following way. Assign weights to the edges according to any of the above described algorithms and then add the value $n \cdot R$ to each weight. Now every edge takes part in at least one shortest path.

Next we show how a good assignment can be constructed deterministically. One way would be to derandomize the above randomized process. Notice that the proof of Lemma 2.1 actually implies that every partial assignment that does not violate the optimality requirements can be completed to a good assignment. We can thus assign weights to the edges one-by-one ensuring at every step that none of the requirements is violated.

A better way of doing this is by the following algorithm that constructs a good assignment with $R = n^3$ (compared with n^5). Initially, every edge e_i is assigned weight $n^4 \cdot 2^i$. The weights of the edges are then changed one-by-one to fit into the range $[1 \dots R]$ while maintaining the goodness of the assignment. At each step, the weight of the heaviest edge is changed.

Lemma 2.4 *If $R \geq n^3$, a good assignment can be constructed.*

Proof: The invariant which is maintained at the end of each step is that the assignment remains good. This is true initially. Let w_i be the new weight assigned to edge e_i at step i, where e_i connects vertices x and y. We prove that w_i can be fitted into the range $[1 \dots R]$ by bounding the number of forbidden values for w_i and showing that at least one permitted number exists. Let l_{uv} denote the value of the shortest distance between vertex u and vertex v when edge e_i is removed from the graph (l_{uv} might be infinite).

To maintain goodness we must accommodate both optimality requirement. We first show how to maintain the uniqueness of the shortest path between every pair of vertices. Let r and v be a pair of vertices, and assume without loss of generality that $l_{rx} < l_{ry}$. (They cannot be equal by the invariant). Let e_i be the edge of largest weight. If the removal of edge e_i from the graph leaves vertices r and v in different connected components, then any value can be chosen for w_i with respect to r and v. Assume this is not the case. Since edge e_i had the largest weight in the graph (i.e., $n^4 \cdot 2^i$), the shortest path from r to v cannot contain edge e_i and l_{rv} is the value of the shortest distance from r to v. Hence, to maintain the uniqueness of the shortest path requirement, it is enough that

$$l_{rv} \neq l_{rx} + w_i + l_{yv}.$$

(Notice that the shortest path will remain unique even if it contains edge e_i, because of the uniqueness of the shortest paths from r to x and from y to v). This condition generates at most $n-1$ forbidden values for w_i with respect to every vertex r in the graph, or $n(n-1)$ forbidden values altogether.

Let us now show how the second requirement of optimality is maintained. Let r, u and v be a triplet of vertices. Again, notice that if the removal of edge e_i from the graph leaves vertex r in one connected component, and vertices u and v in a different connected component, then any value can be chosen for w_i with respect to r, u and v. The same holds if the removal of e_i leaves y separated from r, u, and v. Assume this is not the case. It follows from the above discussion that the shortest distance from r to u is either l_{ru}, or $l_{rx} + w_i + l_{yu}$. Similarly, the shortest distance from r to v is either l_{rv}, or $l_{rx} + w_i + l_{yv}$.

By the invariant,

$$l_{rv} \neq l_{ru} \qquad \text{and} \qquad l_{rx} + w_i + l_{yu} \neq l_{rx} + w_i + l_{yv}.$$

Hence, to maintain the second requirement of optimality, it is enough that

$$l_{rv} \neq l_{rx} + w_i + l_{yu}$$

and

$$l_{ru} \neq l_{rx} + w_i + l_{yv}.$$

These two conditions add at most $2 \cdot \binom{n-1}{2}$ forbidden values for w_i with respect to every vertex r in the graph, for a total of $2n \cdot \binom{n-1}{2}$.

Altogether, the number of forbidden values for w_i is $n(n-1)(n+1) < n^3$, and the lemma follows. □

Note that the initial assignment ($e_i = n^4 \cdot 2^i$) is chosen to ensure that every edge is treated exactly once, and when it is treated it does not participate in any shortest path unless it is a bridge.

The complexity of the algorithm is $O(n^3 m)$ since each step can be implemented in $O(n^3)$ time. Every vertex $v_i \in V$ maintains a table with all its shortest distances to the other vertices; it then marks all the forbidden values in the range $[1 \ldots n^3]$. One of the unmarked numbers is chosen arbitrarily for e_i. Then, the tables of all other vertices are updated.

The reason why the range can be made smaller in the deterministic case is that it is enough to ensure at each step that there is *one* good value, whereas in the randomized case, one has to ensure success with high probability.

A desirable property of a routing scheme is having the traffic be evenly distributed among the edges. Unfortunately, this is the drawback of routing with random weights. The following example shows that with high probability this scheme does not yield a balanced load.

Let the load on an edge be defined as the number of shortest paths that contain it, and consider a graph made of two cliques of size k that are interconnected by two edges, e_1 and e_2. The weight for each edge is chosen uniformly and independently from the range $[1 \ldots R]$. In each clique, the distribution of the weights is uniform and thus, if the weights of e_1 and e_2 are not close to one another, most of the traffic between the two cliques would go through the edge with smaller weight. Since this event will happen with high probability, the communication would not be balanced with high probability. The next section dwells on routing via trees in a way that will allow us to optimally balance the load.

3 Routing Along Trees

In this section we consider our second option of routing namely, routing along BFS trees. Routing along trees can be viewed in two ways: (1) the tree rooted at a node specifies the routes used by the root when acting as a source of messages, or (2) the tree rooted at the node specifies the routes used by the other nodes with the root serving as the destination. From a design standpoint these are identical and in both we strive to balance the load on the links as much as possible.

As before we consider the network as a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. In addition we single out a vertex r called the root. The graph is divided into layers relative to root r by conducting a breadth-first search on G from r (i.e., we construct a tree of the shortest paths from r to all the other nodes in the graph). In this division, layer i, $0 \leq i \leq n-1$, contains all the vertices whose distance from r is i. The corresponding resultant tree is denoted T_r. Note that for a given G and r, the layers are defined uniquely but the BFS tree is not. Also note that given a BFS tree, the edges of the original graph connect vertices only from adjacent layers or in the same layer.

Let $v \in V$ be some vertex in layer i (for some $1 \leq i \leq n-1$). Define d_v^r as the number of neighbors of v at layer $i-1$ in graph G rooted at r; by convention $d_r^r = 0$. The following proposition establishes relations which we shall use later on.

Proposition 3.1 *For any graph G*

1. *The number of different BFS trees from root r is $\prod_{v \in G-r} d_v^r$*
2. *For any r, $\sum_{v \in V} d_v^r \leq m$*

Proof:

1. All the BFS trees can be constructed by having each vertex $v \in G - r$ choose independently a parent out of its neighbors in the previous layer, and each such construction corresponds to a legal and different BFS tree rooted at r. Hence the claim follows.
2. Each edge contributes unity to the sum if its two endpoint vertices are not in the same layer, and zero otherwise. Thus, this sum is exactly equal to the number of edges connecting vertices of different (and therefore adjacent) layers.

□

3.1 Homogeneous Sources

In this section we assume that each node sends (or receives) the same amount of data to every other node, and our aim, as we indicated, is to use the resources evenly. To that end we define the load on an edge as follows. Assume that for every vertex r in the graph we are given a single BFS tree rooted at that vertex (thus determining node's r routing). The load on an edge is defined (relative to this set of trees) as the number of trees which contain this edge. Formally, we are given a set $\{T_r\}_{r\in V}$ containing a single T_r for every $r \in V$ and we define the *load* as

$$l(e) = |\{r \in V | e \in T_r\}| .$$

Note that $l(e) \leq n$ and $\sum_{e\in E} l(e) = n(n-1)$, since there are n BFS trees with $n-1$ edges in each and each edge in a BFS tree contributes a unity to the sum. The capacity of an edge e, denoted $c(e)$, is defined as the maximum number of BFS trees that may contain it.

Our goal is to choose a set $\{T_r\}_{r\in V}$ such that the maximum load of the edges is minimized. We do this by solving a more general problem in which edges have limited capacities that are not necessarily equal. Assume that we are given the edge capacity $c(e)$ for each edge $e \in E$. We are seeking a feasible solution that is, a set $\{T_r\}_{r\in V}$ such that $l(e) \leq c(e)$ for all e. A solution for the capacitated problem can be easily used to solve the problem of minimizing the maximum load (in the uncapacitated problem). We just let $c(e) = c$ for all e and perform a binary search on $1 \leq c \leq n$, thereby increasing the complexity by a factor of $\log n$.

In order to solve the capacitated problem we define the following bipartite graph $H = (A \cup B, F)$. Side A consists of $n(n-1)$ vertices denoted by pairs (r, v) for all $v, r \in V$, $v \neq r$ (this pair will subsequently be interpreted as a root r and some vertex v in G). Side B consists of m vertices, each corresponding to (and denoted by) an edge e for all $e \in E$. Each vertex $(r, v) \in A$ is connected to a vertex $e \in B$ iff $\exists T_r$ (i.e., a tree rooted at r) in which $e \in E$ connects v to a vertex from the previous level.

Note that the degree of vertex (r, v) is d_v^r as per the definition of d_v^r. Also, from proposition 3.1 $|F| = \sum_{v,r} d_v^r \leq \sum_r m = nm$.

The key observation is that in order to solve our problem we need to find $n(n-1)$ edges in the graph H such that the degree of each vertex in A is exactly 1 (matching), and the degree of vertex $e \in B$ is at most $c(e)$. These edges define the n BFS trees in G. Specifically, the edges of T_r are the vertices in B which are adjacent to the vertices (r, v) for all $v \in G - r$. We present two algorithms for finding these trees.

Algorithm 1. Each vertex $e \in B$ with all its incident edges is duplicated $c(e)$ times, generating an "exploded" graph. Now, it is clear that solving the problem is equivalent to finding a perfect matching for side A into side B. The number of vertices in the exploded graph is $n(n-1) + \sum_e c(e) < n^2 + mn$ and the number of edges is at most $n|F| \leq n^2 m$. The complexity of computing a maximum matching in a bipartite graph is $O(|E|\sqrt{|V|}) = O(m^{3/2} n^{5/2})$ [11].

The latter complexity can be improved by the next algorithm.

Algorithm 2. Add to the graph $H = (A \cup B, F)$ a source node s and sink t. Add directed edges from s to all the vertices in A, each with capacity 1, and directed edges from each vertex $e \in B$ to t, each with capacity $c(e)$. Finally, direct all the edges from A to B and assign each the capacity 1 (any capacity greater than 1 will also do).

Consider an integer flow problem with source s and destination t obeying the specified capacities. It is clear that any such legal flow starts with some edges from s to A with flow 1. Then, each vertex in A that has an incoming edge with one unit of flow also has one outgoing edge with one unit flow to a vertex in B. Finally, all the flow reaching B continues to t. Thus we conclude that there is a feasible solution to our problem iff the maximum flow between s and t is exactly $n(n-1)$.

We will use Dinic's algorithm for finding the max-flow [12]. A careful analysis of the algorithm for our case yields a better complexity than more recent max-flow algorithms that perform better on general graphs. We first give a short review of Dinic's algorithm. The algorithm has $O(|V|)$ phases; at each phase only augmenting paths of length i, $1 \leq i \leq |V|$, are considered. The invariant maintained at phase i is that there are no augmenting paths of length less than i. The complexity of each phase is $O(|E||V|)$ in general graphs and $O(|E|)$ in 0-1 networks.

We first convert our graph into a 0-1 network. Each edge of capacity $c(e)$ is duplicated into $c(e)$ unity capacity edges which yields a 0-1 network. Since $c(e) \leq n$ for every edge e, the total number of new edges is at most nm and thus the number of edges remains $O(nm)$. As mentioned before, the complexity of Dinic's algorithm for 0-1 network is $O(|E||V|)$ which in our case becomes

$$O([n^2 + m][n^2 + mn + mn]) = O(n^2 \cdot mn) = O(mn^3)$$

In fact, the running time can be reduced to $O(mn^2)$. Let the residual graph be defined as the graph (obtained from a given flow) that consists of all edges with positive residual capacities, where the residual capacity of edge (u, v) represents the maximum additional flow that can be sent using edges (u, v) and (v, u). In our graph, there are no edges between vertices in A and also none between vertices in B, and there will not be such in any of the residual graphs. In fact, the residual graph will always start with s, end with t, have only vertices of A in the other even numbered layers and only vertices of B in the other odd-numbered layers. Moreover, the vertices of A will always have, in any residual graph, at most one incoming edge. Let us run the first $n-1$ phases of Dinic's algorithm (where each phase takes time $O(|F|) = O(nm)$). In phase n there will be at least n layers of A (unless we have already finished), one of them having at most $n(n-1)/n = n-1$ vertices. The incoming edges into this layer of A define a cut separating s from t whose capacity is at most $n-1$. Thus, Dinic's algorithm will terminate after at most additional $n-1$ phases, which gives the desired time bound.

3.2 Heterogeneous Sources

The situation at hand in this section is similar to that of the previous subsection except that we no longer assume homogeneous traffic but rather that each node generates a different amount of traffic. Translated into our model, this results in a problem with weighted trees. Formally, let the relative traffic intensity associated with node r be $w(r)$ (assumed to be an integer). This means that the tree associated with r (where r is the root) has a weight of $w(r)$ and we seek a set of BFS trees

$\{T_r\}_{r\in V}$ with load $l(e) \leq c(e)$ for all e, where the load $l(e)$ is defined in the natural way, i.e.,

$$l(e) = \left\{ \sum_r w(r) | e \in T_r \right\}$$

The Capacitated Problem of the previous subsection is the special case of our problem with $w(r) = 1$ for all $r \in V$. While the Capacitated Problem in the homogeneous case has an efficient solution, we prove that in the heterogeneous case this problem is NP-complete (it is clear that the problem belongs to class NP). We base our proof on a reduction from the "knapsack" problem which is known to be NP-complete [13], defined as follows.

The Knapsack Problem: Given are integers $x_1 \ldots x_n$ and s. Are there $\alpha_i \in \{0, 1\}$, $1 \leq i \leq n$, such that $\sum \alpha_i x_i = s$?

The Reduction: Consider a graph whose vertices are $v_1, \ldots v_n, u_1, u_2, t$. Connect v_i to u_j for $1 \leq i \leq n$, $j = 1, 2$ and connect u_1 and u_2 to t. Let the weight of the sources be $w(v_i) = x_i$ for all i, $w(u_1) = w(u_2) = w(t) = 0$. Finally, let the capacities of the edges be $c(u_1 t) = s$, $c(u_2 t) = \sum_i x_i - s$, and infinite (or big enough) for all the rest. It is clear that each BFS tree from v_i, $1 \leq i \leq n$, contains exactly one of the edges $u_1 t$ or $u_2 t$. Since $c(u_1 t) + c(u_2 t) = \sum_i x_i$, there is a solution iff there is a subset of the integers x_i that sums up to s.

Note that it is possible to eliminate the zero weights (and have the proof still hold) by assigning $w(u_1) = w(u_2) = w(t) = 1$ and also adding 2 to the capacities of the edges $u_1 t$ and $u_2 t$.

3.3 Randomized Capacity Bounds

In this section we develop upper bounds on the capacities that are needed for the edges in the Capacitated Problem of the homogeneous case (section 3.1) in order to achieve "good" load balancing. Our reference is a *random tree* routing scheme in which every node, whenever it needs to send a message, randomly and uniformly chooses a BFS tree in which it is a root, and routes according to this tree. Intuitively, such a routing scheme is likely to achieve a good balancing.

We start by calculating P_e^r – the probability that an edge e participates in a randomly and uniformly chosen BFS tree rooted at r. Let x_e^r be an indicator random variable indicating whether edge e belongs to the BFS tree rooted at r. By our definition

$$l(e) = \sum_{r\in V} x_e^r.$$

Consider an edge $e = (x, y)$. If both x and y are in the same layer (i.e., equidistant from r), then $P_e^r = 0$. Otherwise, they belong to adjacent layers (without loss of generality let x be the vertex that is further away from r), and $P_e^r = \frac{1}{d_x^r}$.

Let $\bar{l}(e)$ be the expected load of e. Clearly $E[x_e^r] = P_e^r$ and also

$$\bar{l}(e) = E\left[\sum_{r\in V} x_e^r\right] = \sum_{r\in V} E[x_e^r] = \sum_{r\in V} P_e^r$$

We cannot expect to find a set of BFS trees in which $l(e) \leq \bar{l}(e)$ for every edge e ($\bar{l}(e)$ is not necessarily an integer for instance). However, we can find a set which is almost as good. We show that there always exists a set of BFS trees $\{T_r\}_{r\in V}$ such that the load on any edge satisfies the following:

$$l(e) \leq \bar{l}(e) + 2\sqrt{\bar{l}(e)\log n}.$$

We will prove the claim via the probabilistic method; one can easily find such a set by applying the algorithm from section 3.1 as we are guaranteed that a solution exists.

To prove the bound on the load, we show that for each edge e, the probability that $l(e)$ exceeds the claimed bound is less than $\frac{1}{2m}$. Hence, there is a positive probability that the claim holds for all edges in the network. ¿From Chernoff's bounds it can be shown that for all $\lambda \geq 0$,

$$\mathrm{Prob}[l(e) > (1+\gamma)\bar{l}(e)] \leq \frac{E[e^{\lambda l(e)}]}{e^{(1+\gamma)\lambda\bar{l}(e)}}$$

and it can be shown [14] that there exists a choice of λ such that

$$\frac{E[e^{\lambda l(e)}]}{e^{(1+\gamma)\lambda\bar{l}(e)}} \leq e^{-\gamma^2\bar{l}(e)/2}.$$

Assigning $\gamma = 2\sqrt{\frac{\log n}{\bar{l}(e)}}$, results in

$$\mathrm{Prob}[l(e) > \bar{l}(e) + 2\sqrt{\bar{l}(e)\log n}] \leq \frac{1}{n^2} < \frac{1}{2m}$$

which finally yields

$$\mathrm{Prob}[\forall e, l(e) \leq \bar{l}(e) + 2\sqrt{\bar{l}(e)\log n}] > \frac{1}{2}$$

By choosing other values for γ such as $\gamma = k\sqrt{\log n/\bar{l}(e)}$, it can be shown that

$$|l(e) - \bar{l}(e)| \leq O(\sqrt{\bar{l}(e)\log n})$$

is fulfilled almost surely for all edges $e \in E$.

4 Routing Along Shortest Paths

In this section we consider a different option of routing namely, routing along paths that do not necessarily form trees. One of the shortest paths between every pair of nodes is designated as the path along which these two nodes exchange messages. We prove that a set of paths can be chosen that yields a balanced load. (Finding an optimal set of paths is NP-complete by reduction from multi-commodity flow).

The proof we present follows the exact same lines of the proof in section 3.3 and we adopt the same notation. Again, our reference for a good load balancing is the *random path* routing scheme

We first evaluate P_e^{uv}–the probability that an edge e participates in a randomly and uniformly chosen shortest path connecting vertices u and v. (We will denote this event by the indicator variable x_e^{uv}). To compute this probability, we must count the shortest paths connecting u and v that contain edge e. Let $M_p(u,v)$ denote the number of paths of length p between the vertices u and v. The number of shortest paths between u and v can be computed in polynomial time by the following recursive formula. Let the vertices adjacent to u be $a_1,\ldots,a_d$ and let p be the length of the shortest path from u to v, then

$$M_p(u,v) = \sum_{i=1}^{d} M_{p-1}(a_i,v).$$

We consider a pair of nodes u and v and an edge $e=(x,y)$ (assume without loss of generality that vertex x is closer to u than vertex y). Denote by p_{uv} the distance between the vertices u and v, by p_{ux} the distance between u to x, and by p_{yv} the distance between v and y. Define $p' = p_{uv} - p_{ux} - 1$. If $p_{yv} > p'$, then $P_e^{uv} = 0$; otherwise,

$$P_e^{uv} = \frac{M_{p_{ux}}(u,x)\cdot M_{p_{yv}}(y,v)}{M_{p_{uv}}(u,v)}$$

Similar to the derivation in section 3.3 the expected load on an edge e is $\bar{l}(e) = \sum_{u,v\in V} P_e^{uv}$ and thus we cannot expect to find a set of shortest paths in which $l(e) \le \bar{l}(e)$ for every edge e. However, again, we can find a set which is almost as good, namely, a set of shortest paths such that the load on any edge satisfies

$$l(e) \le \bar{l}(e) + 2\sqrt{\bar{l}(e)\log n}.$$

An edge whose load does not satisfy the above condition is called an *overloaded* edge. If there are no overloaded edges, then the set of paths is called a *good set*. We will prove that a good set of paths exists via the probabilistic method and then show how to find such a set of paths deterministically.

Let every pair of vertices choose its path uniformly in random (among the shortest paths between them). We show that with high probability, the set of paths chosen is good. The random variable $l(e)$ is a sum of $\binom{n}{2}$ indicator variables x_e^{uv}. These variables are independent because each pair of vertices chooses its path independently of the other pairs. If we show that the probability that edge e is overloaded is less than $\frac{1}{2m}$, then with high probability the claim holds for all edges in the network. As stated in

Section 3.3, it can be shown that for all $\lambda \geq 0$,

$$\text{Prob}[l(e) > (1+\gamma)\bar{l}(e)] \leq \frac{E[e^{\lambda l(e)}]}{e^{(1+\gamma)\lambda \bar{l}(e)}}$$

furthermore, there exists a choice of λ [14] such that

$$\frac{E[e^{\lambda l(e)}]}{e^{(1+\gamma)\lambda \bar{l}(e)}} \leq e^{-\gamma^2 \bar{l}(e)/2}$$

Similar to Section 3.3, assigning $\gamma = 2\sqrt{\frac{\log n}{\bar{l}(e)}}$, results in

$$\text{Prob}[l(e) > \bar{l}(e) + 2\sqrt{\bar{l}(e)\log n}] \leq \frac{1}{n^2} < \frac{1}{2m}$$

which finally yields

$$\text{Prob}[\forall e, l(e) \leq \bar{l}(e) + 2\sqrt{\bar{l}(e)\log n}] > \frac{1}{2}$$

as was claimed.

Having established that there exists a good set of paths we now show how to find this good set deterministically in polynomial time by the *method of conditional probabilities* [15],[16]. This method was introduced by Spencer [15] with the intention of converting probabilistic proofs of existence of combinatorial structures into efficient deterministic algorithms for actually constructing these structures. The idea is to perform a binary search of the sample space associated with the random variables so as to find a good set. At each step of the binary search, the current sample space is split into two halves and the conditional probability of obtaining a good set is computed for each half. The search is then restricted to the half having a higher conditional probability. The search terminates when only one sample point remains in the subspace, which must correspond to a good set.

To apply this method to our case for finding a good set of paths, we will consider the indicator variables one-by-one. In a typical step of the algorithm, the value of some of the indicator variables has already been set, one variable is currently being considered, and the rest are chosen in random. (By choosing in random we mean that for the pair of vertices which is now being considered, the remainder of the path is chosen uniformly in random.) At each step we will compute the (conditional) probability of finding a good set if the variable considered is set to 0 and if it is set to 1.

We denote by P_j the probability of finding a <u>b</u>ad set of paths after the variable considered at step j has already been assigned a value and by P_j^i the probability of obtaining a bad set of paths by assigning the value i, for $i = 0, 1$, to the variable considered at step j. Initially, it follows from the existence proof that the probability of choosing a good set of paths is positive; we inductively maintain that $P_j < 1$ for $j \geq 1$, and hence, either $P_j^0 < 1$ or $P_j^1 < 1$.

For the sake of simplicity, assume the following on the order in which the variables are considered:

- For a pair of vertices u and v, for all edges e, the variables x_e^{uv} are considered consecutively.
- For a pair of vertices u and v, the edges are considered according to their distance from u. (Ties are broken arbitrarily).

For example, suppose that we are considering the variable x_e^{uv} where $e = (a, b)$ and assume that vertex a is closer to u than b. Notice that by assigning a value to x_e^{uv},

- The probability P_f^{uv} may change for edges f for which x_f^{uv} has not been determined yet. (These changes in the probabilities can be computed in polynomial time.)
- The value of x_f^{uv} for other edges f may also be determined, e.g., if $x_e^{uv} = 1$, then for all edges f adjacent to a, $x_f^{uv} = 0$.

A major stumbling block in applying the method of conditional probabilities is always the computation of the conditional probabilities. In our case, we do not compute the exact probability that there exists an overloaded edge (even initially), but rather only *estimate* it. Consequently, if the estimator is not chosen judiciously, it may happen that when a variable is considered, according to the estimator, no value assigned to it can lead to a good solution. To overcome this difficulty, following Raghavan [16], the notion of a pessimistic estimator is introduced. We call $\hat{P}_j$ a *pessimistic estimator* of the conditional probability P_j if it satisfies the following conditions:

1. $\hat{P}_0 < 1$.
2. For any partial assignment of the first j variables, $P_j \leq \hat{P}_j$.
3. $\min\{\hat{P}_j^0, \hat{P}_j^1\} \leq \hat{P}_{j-1}$ where $\hat{P}_j^i$ is the estimator of P_j^i for $i = 0, 1$.
4. The pessimistic estimators can be computed in polynomial time.

It is not very hard to see that such a pessimistic estimator can equally well be used in the method of conditional probabilities instead of the exact conditional probabilities which are hard to compute in general. We now show that the pessimistic estimator that we will choose indeed satisfies the above conditions. We have earlier proved that initially,

$$\text{Prob}\,[\text{the set is bad}] \leq \sum_{f \in E} \text{Prob}[l(f) > (1+\gamma_f)\bar{l}(f)] \leq \sum_{f \in E} \frac{E[e^{\lambda_f l(f)}]}{e^{(1+\gamma_f)\lambda_f \bar{l}(f)}} < 1$$

Notice that λ_f and γ_f depend on the edge f. We define

$$\hat{P}_0 = \sum_{f \in E} \frac{E[e^{\lambda_f l(f)}]}{e^{(1+\gamma_f)\lambda_f \bar{l}(f)}}$$

The estimator at Step j is defined to be

$$\hat{P}_j = \sum_{f \in E} \frac{E[e^{\lambda_f l_j(f)}]}{e^{(1+\gamma_f)\bar{l}(f)\lambda_f}}$$

where $l_j(f)$ is a random variable denoting the load on edge f at the end of Step j. For example, suppose that $l(f) = x_1 + x_2 + x_3 + x_4$ and at the end of Step j, $x_2 = 0$ and $x_4 = 1$. Then, $l_j(f) = 1 + x_1 + x_3$. ($\bar{l}(f)$, γ_f and λ_f retain their original values).

Condition (4) holds since the changes in the probabilities at each step can be computed in polynomial time as mentioned earlier. (Notice that the random variable $l_j(f)$ is the sum of independent random variables). Condition (2) holds since

$$P_j \leq \sum_{f \in E} \text{Prob}[l_j(f) > (1+\gamma_f)\bar{l}(f)] \leq \sum_{f \in E} \frac{E[e^{\lambda_f l_j(f)}]}{e^{(1+\gamma_f)\lambda_f \bar{l}(f)}} = \hat{P}_j.$$

Let us show that condition (3) holds as well. Suppose that at Step $j+1$ variable x_e^{uv} is being considered. By definition,

$$\sum_{f \in E} E[e^{\lambda_f l_j(f)}] = P_e^{uv} \cdot \sum_{f \in E} E[e^{\lambda_f l_j(f)} | x_e^{uv} = 1] + (1 - P_e^{uv}) \cdot \sum_{f \in E} E[e^{\lambda_f l_j(f)} | x_e^{uv} = 0]$$

where the probability of choosing edge e as part of the path from u to v is P_e^{uv} (given the assignments of the previous j steps). Now,

$$\hat{P}_{j+1}^1 = \sum_{f \in E} \frac{E[e^{\lambda_f l_j(f)} | x_e^{uv} = 1]}{e^{(1+\gamma_f)\bar{l}(f)\lambda_f}}; \qquad \hat{P}_{j+1}^0 = \sum_{f \in E} \frac{E[e^{\lambda_f l_j(f)} | x_e^{uv} = 0]}{e^{(1+\gamma_f)\bar{l}(f)\lambda_f}}$$

Hence,

$$\hat{P}_j = P_e^{uv} \cdot \hat{P}_{j+1}^1 + (1 - P_e^{uv}) \cdot \hat{P}_{j+1}^0$$

and clearly, $\min\{\hat{P_{j+1}^0}, \hat{P_{j+1}^1}\} \leq \hat{P}_j$. The value of x_e^{uv} is set to the value for which $\hat{P}_{j+1}^i$ is minimized, for $i = 0, 1$.

Acknowledgemnet

We would like to thank Noga Alon for many helpful discussions on this paper and in particular for his help in analyzing the algorithm of Section 3.1. Most of this work was done while the first two authors were at the Computer Science Department of Stanford University and supported by contract ONR N00014-88-K-0166.

References

[1] A. Ephremides, "The routing problem in computer networks," in *Communications and Networks* (I. Blake and H. Poor, eds.), pp. 299–324, New York: Springer Verlag, 1986.

[2] M. Schwartz and T. Stern, "Routing techniques used in computer communication networks," *IEEE Trans. on Communications*, vol. COM-28, pp. 539–555, April 1980.

[3] P. Green, "Computer communications: Milestones and prophecies," *IEEE Communications*, pp. 49–63, 1984.

[4] I. Cidon and I. Gopal, "Paris: An approach to integrated high-speed private networks," *International Journal of Digital and Analog Cabled Systems*, vol. 1, pp. 77–86, April-June 1988.

[5] J. Turner, "Design of a broadcast packet switching network," *IEEE Trans. on Communications*, vol. COM-36, pp. 734–743, June 1988.

[6] H. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies.* Lexington, MA: Lexington Books, 1984.

[7] L. Fratta, M. Gerla, and L. Kleinrock, "The flow deviation method: An approach to store and forward communication network design," *Networks*, vol. 3, no. 2, pp. 97–133, 1973.

[8] R. Gallager, "A minimum delay routing algorithm using distributed computation," *IEEE Trans. on Communications*, vol. COM-25, pp. 73–85, January 1977.

[9] O. Lesser and R. Rom, "Routing by controlled flooding in communication networks," in *Proccedings of IEEE Infocom'90*, (San Francisco, California), pp. 910–917, IEEE, June 1990.

[10] K. Mulmuley, U. Vazirani, and V. Vazirani, "Matching is as easy as matrix inversion," *Combinatorica*, vol. 7, no. 1, pp. 105–113, 1987.

[11] J. Hopcroft and R. Karp, "An $n^{5/2}$ algorithm for maximum matching in bipartite graphs," *Siam J. Computing*, vol. 2, pp. 225–231, 1973.

[12] S. Even, *Graph Algorithms.* New York: Computer Science Press, 1979.

[13] M. Garey and D. Johnson, *Computers and Intractability.* San Francisco: W.H. Freeman and Company, 1979.

[14] D. Angluin and L. G. Valiant, "Fast probabilistic algorithms for hamiltonian circuits and matchings," *Journal of Computer and System Sciences*, vol. 18, pp. 155–193, 1979.

[15] J. Spencer, *Ten Lectures on the Probabilistic Method.* Philadelphia, Pennsylvania: SIAM, 1987.

[16] P. Raghavan, "Probabilistic construction of deterministic algorithms: Approximating packing integer programs," *Journal of Computer and System Sciences*, vol. 37, pp. 130–143, October 1988.