Efficient Network Search Algorithms for Automatic Route Generation

Telcordia Technologies, Technical Report, February 2000

Clyde Monma, Prasad Rao, Paul Seymour and David Shallcross

Abstract: We describe state-based search approaches for determining routing for capacity and service activation as part of the connection management process for providing network services. These algorithms help to automate the process of route generation to provision end-to-end circuits through a transport network. The initial focus is on layer one with possible future applications to levels two and three. Whereas today this is mostly a manual task, these algorithms provide capabilities to automatically route, based on the configuration of existing transport and spare capacity, and rules to help engineers with a candidate list of pre-qualified routes.

These algorithms are applied to an extract of the database of transport capacities and their port connectivities and spare capacities that are then intelligently used to generate feasible routes. Various types of constraints on feasible paths are modeled, including the rate (DS0, DS1, DS3, etc) requested, min/max number of x-connects, transport links or any set of facility types which can/must be used. The output of these algorithms is a list of "good" routes meeting all of the specified constraints.

0. Introduction

The Network Configuration Software System (NCON) is a distributed software system, intended to provide support for management of a company's backbone and transport network. NCON manages the assignable equipment and facilities in a company's network. NCON reduces the time, effort and errors associated with activating customer circuits, trunks, carrier systems and networks by eliminating manual record keeping and automating design and assignable tasks. NCON also supports service assurance processes by providing an easily accessible database of circuit, service and facility configurations. In addition, NCON helps network planners and engineers improve network capacity utilization by providing a centralized database that can be used to determine the status of network equipment and facilities, and how they are being used.

NCON is part of the Telcordia Media Vantage product line. The Connection Manager (CM) Tier of NCON includes the CM Core, Service Adapter and Network Adapter. The CM Core implements a Generic Network Model (GNM) that can be used to model a broadband network in an abstract protocol neutral manner. The connection manager functions provide cost effective route selection and automated end-to-end connection provisioning.

In this document, we describe several generic routing algorithms developed in the Applied Research Area to provide a set of good routes given a network model representation and user specified constraints. In particular, we describe state-based search approaches for determining capacity and service routings as part of the connection management process for providing network services. These algorithms help to automate the process of route generation to provision end-to-end circuits through a transport network consisting of add-drop multiplexors, terminal

multiplexors, and cross connects. These algorithms provide capabilities to automatically route, based on the configuration of existing transport and spare capacity, and rules to help engineers with a candidate list of pre-qualified routes.

Section 1 describes the problem and data. In Section 2, we describe the various routing algorithms. Section 3 describes the problem generators for testing these algorithms and the computational results are given in Section 4. Appendices A and B describe the standard input and output file formats, respectively, for the prototype software.

1. Problem and Data Description

1.1 The Data Description

The inputs to the optimization procedure include a graph G = (V,E) consisting of a set of vertices or locations V, together with a set of edges or links E. This graph is an abstract representation of the underlying broadband network. Each vertex and edge may have certain attributes associated with them, such as a switch node or link on a SONET ring. These attributes are used as part of the user-input constraints to determine the set of feasible paths. Most of the constraints are all of the form: the count of nodes (or edges) of some particular type over the entire path lies within some range (i.e., between a specified minimum and maximum value, which allows for a zero minimum or an "infinite" maximum). We can also handle additional constraints, such as forcing a path to visit the hub nodes for the source and sink nodes.

Additional inputs include the starting node (s) and the ending node (t) between which a connection is desired, together with the number of feasible paths desired. A list of feasible paths is generated to allow the system or the end user to select the most appropriate route (among the list of optimal paths) based on other preferences or constraints not modeled by our algorithms, or in case some of the generated paths are no longer viable given a very recent change in the state of the underlying network at that particular moment.

1.2 The Problem Description

The goal of our algorithms is to provide a list of good routes satisfying the constraints. We will describe several algorithms, each with different characteristics. The performance of each algorithm, in terms of running time or "quality" of paths produced differs depending on the size and structure of the underlying network, the number and "difficulty" of the constraints involved and the number of feasible paths desired.

The ultimate performance of these algorithms will depend on the types of problems likely to be encountered in practice. Since data from an existing customer is not yet available, we have tested these algorithms on data modeled after LATA-type networks, as well as randomlygenerated networks of varying size and complexity.

We have produced a robust set of algorithms that seem to work well on a wide variety of large and complex networks as well as simpler cases. These algorithms are all contained in a single prototype research software package and use the same standard input and output formats. This allows easy future comparisons on real networks once this data becomes available.

These algorithms are applied to an extract of the database of transport capacities and their port connectivities and spare capacities that are then intelligently used to generate feasible routes.

Various types of constraints on feasible paths are modeled, including the rate (DS0, DS1, DS3, etc) requested, min/max number of x-connects, transport links or any set of facility types which can/must be used. The output of these algorithms is a list of "good" routes meeting all of the specified constraints.

2. The Routing Algorithms

2.1 Radial Search (RS) Algorithm

The core of the radial search algorithm is a recursive backtracking search, similar to a depth first search. Given a source, destination, and a desired hop count, it looks for paths from the source to the destination whose length is the given hop count. These paths will be true paths; that is, they will not repeat nodes. If the desired hop count is the actual minimal hop count between source and destination, then any pair of generated paths will intersect only at some sub-path starting at the source, and at the destination. We do not necessarily generate all possible paths.

The search proceeds by starting at a node (originally the source), looking at each of its neighbors, and recursively searching from each neighbor for which the proposed sub-path from the source to this neighbor (1) hasn't already visited this neighbor already, and (2) is shorter than every other sub-path found so far from the source to the neighbor. If we reach the destination, we output the constructed path, if it is of the correct length. If it is too short, any extension to the desired length will repeat the destination, so we back up. If we reach the desired hop count without reaching the destination, we back up.

The overall radial search algorithm takes a source, a destination, a graph, a hop limit, and a desired number of paths, and tries to generate at least that number of paths. It repeats the core algorithm, with the desired hop count starting at one, and successively increasing, until we either have generated the desired number of paths (or more), or we reach the hop limit. No constraints beside the hop limit are enforced in the current implementation. Constraints would be enforced by post-processing, i.e., checking each path generated, and removing those that violate the constraints.

Another, faster algorithm, called Quick Tree Radial Search (QTRC), to generate paths is based on the structure of the set of paths given by the radial search core algorithm when the desired hop count is the minimum possible hop count. This new algorithm first builds a breadth-first shortest path tree from the source node. It then generates, for each neighbor of the destination, a path consisting of the path in the breadth-first tree from the source to the neighbor, followed by the arc from the neighbor to the destination. This will generate, in time linear in the number of edges, all of the minimum-length paths that the radial search algorithm generated, plus possibly some other paths, up to two hops longer.

It is important to note that no constraints, beside the hop limit, are enforced by either the RS or QTRS algorithms and that they only guarantee to produce a short path as opposed to a list of good paths. These algorithms are simply used as a benchmark against which the later algorithms that do incorporate multiple constraints and give a list of candidate paths are measured.

2.2 Diverse Feasible Paths (DFP) Algorithm

The input (of the network, the two nodes to be joined, the number of paths required, and the various restrictions and bounds) is in the standard format common to all three algorithms. (See Appendix A. Standard Input Format.) Also the output (i.e., the list of paths selected by the algorithm) is in the standard output format. (See Appendix B. Standard Output Format.) What needs to be described here is the algorithm itself.

There are five main parts to the DFP algorithm:

1. Preprocessing the bounds.
2. Constructing a state graph, one where each node of the original network is represented by several nodes, one for each state that the original node may be in.
3. Running a breadth-first search in the state graph, starting from the "sink".
4. Using the distances to generate random paths in the state graph from source to sink.
5. Selecting a diverse subset of the paths found.

Now we describe each part in more detail.

Part 1. Preprocessing the Bounds

For each type of node there is a lower and upper bound on the number of times an acceptable path may use nodes of this type; and similarly each type of edge has an upper bound and a lower bound of zero. (Remark: so at the moment we are assuming that for each edge type there is nontrivial lower bound, and also that each node and edge has a unique type and so contributes to only one constraint. Both these restrictions could be lifted if necessary.) Let us define "statecount" to mean 1 plus the product, over all node types and edge types, of (upper bound + 1), or (lower bound + 1) if the upper bound is infinite.

It is important for the time and space performance of the algorithm to keep statecount small, in the hundreds rather than in the millions; the space required by the algorithm is linear in statecount, and the running time is at least linear in statecount (rather more in practice). So, we first scan all the bounds to see if any can be safely disregarded. For instance, any upper bound that is bigger than the limit on total number of hops ("hoplim") can be treated as infinity and disregarded; also, any type of node or edge which does not actually appear in the network can be disregarded. Then, whatever lower and upper bounds remain are renumbered in a convenient order, grouping together bounds of the same form (i.e., "lower bound trivial", "upper bound trivial", or "both bounds nontrivial").

We then enumerate all possible "states" at a general node v, that is, all possible combinations of node and edge types that could have occurred in a path from the source to this node (e.g., 1 edge of type 1, 3 edges of type 2, 1 node of type 1, and 4 nodes of type 2) - assuming that if a node or edge type has an upper bound, then the path obeys that upper bound (so no numbers bigger than the upper bound are enumerated), while if it has a lower bound and no upper bound then no numbers bigger than the lower bound are enumerated (because as long as the lower bound has been satisfied, we do not need to record by how much it was exceeded). In this enumeration, we only count nodes and edges whose type has an associated nontrivial lower or upper bound. The number of states enumerated here is in fact equal to statecount (we also list one special state, used to represent that the path has violated some upper bound). We number all states from 0 to statecount - 1, in a convenient order.

Next we construct a matrix representing transitions between states. (This is independent of the actual network, and determined just from knowledge of the constraints.) For every edgetype x and

every nodetype y, and every state s, we wish to calculate the following: suppose an edge e of type x has nodes u and v, and v has type y, and a path from the source has arrived at u in state s. Now this path is to be extended to v via the edge e; what will be its state when it arrives at v? The path may become illegal if some bound becomes violated by the addition of e and v, and otherwise the new state will be one of the possible states we computed earlier (possibly the same as s). Let us call this resultant state "transition [x][y][s]".

By means of this device we will not need to read through the list of constraints or to check if any are violated, during the breadth-first search part of the algorithm, which is where the performance is critical.

Part 2. Constructing the State Graph

For each node of the network, we take several copies of it (in fact, statecount + 1 copies), one for each possible state of a path from the source arriving at it. For our purposes here, let us call the copies of node v (v,0),..,(v, statecount). To keep the space requirements down, we do NOT take corresponding copies of each edge; we just read edges off the original network when we need them. However, for the purposes of this description, let us imagine that for each edge uv of the network, and every state s, there is an edge of the state graph from (u, s) to (v, t), where t is the state transition [x][y][s] (where uv has edgetype x and v has nodetype y). It should be stressed that the state graph is a directed graph, although (in the current implementation) the input graph is undirected.

Part 3. Finding a Breadth-First Tree in the State Graph

If a path from the source arrives at the sink, and its length is at most hoplim, then whether this is considered a legal path is determined by its state. We list all the states that correspond to legal paths; let us call them "sinkstates". These are represented by some of the copies of the sink in the state graph.

For each node (v, s) of the state graph we wish to compute the minimum length of a path Q in the network from v to the sink such that, if P were some hypothetical path from the source to v in the network which arrived at v in state s, then P + Q would be a legal path if it were not too long. In other words, Q is to be a path from v to the sink in the network, achieving the "complement" of state s; for any nodetype which, according to the state s, have not yet been used enough, the path Q must pass through at least the missing number of nodes with this nodetype, and similarly for nodetypes and edgetypes with upper bounds. This is equivalent to asking for the shortest directed path in the state graph from (v,s) to a sinkstate. (We are using the term "path" loosely here - it is permitted to pass through a node or edge more than once.)

So, we wish to find the minimum distance from (v,s) to a sinkstate in the state graph, for all pairs (v,s). There are several ways to do this. After a good deal of experiment we selected one designed to take advantage of two special features of the input network, namely that the number of nodes remains fairly small, in the thousands (although the number of edges can be in the hundreds of thousands), and the hoplim is small (typically less than 20).

Recursively for i = 1,.., hoplim, we find all the pairs (v,s) whose minimum distance to the sinkstates is i. To do so, we preserve two overlapping sets of data. At the start of the ith iteration, for each node v of the network, we have a list (L(v) say) of all the states s such that the distance

from (v,s) to the sinkstates is at least i; and for every pair (v,s) a toggle which (for the pairs (v,s) with distance at least i) indicates whether the distance from (v,s) to sinkstates equals i or not. The iteration has two parts. First, for each v we scan through L(v), and divide the list into two shorter lists, depending whether (v,s) is toggled or not (call the first list T(v); the second will become the new L(v)). Then set all toggles to NULL; and for each v, and for each member (v,s) of T(v) we run through all neighbours of (v,s) and toggle them.

This procedure has the advantage over the usual breadth-first tree algorithm that it avoids asking, for each EDGE from (u,s) to (v,t), whether we already know the best path for (v,t); and the disadvantage that we have to scan through the list L(v) for each node v, once for each value of i. In practice this appears to yield a net reduction in running time.

Part 4. Using the distances

Now we have found the minimum distance from (v,s) to a sinkstate, for each node (v,s) of the state graph. Next we use this to randomly generate legal paths from source to sink. A first attempt to do so might be to generate a path starting at the source (in the state graph) and grow it randomly somehow in the state graph, making sure at each step that the number of hops needed to complete this to a legal path to the sink (which we computed in Part 3) will yield a path from source to sink of total length at most hoplim. But this does not work well; the path tends to walk randomly around at the start with no urge to go in any direction, until the hoplim bound bites, and then it takes off along a shortest path route to the sink. This is unsatisfactory for several reasons - the paths we generate are too wild at the source and too tame at the sink, and also the paths tend to have many self-intersections, even in the state graph.

But there is a simple trick to make it work better. Let n be the length of the shortest legal path from source to sink. (This was one of the things we computed in the previous part.) Choose a number, say d, between n and hoplim, and let us look for a random path of length at most d. Generate a random "waste function" f. (We take f(0) = 0 and f(d) = d - n, and fill in the intermediate values of f with a random increasing step function.) Now grow the random path as before, but being more careful not to waste steps in the early part of the path - we insist that at the (i+1)th node of the path we must not have wasted more than f(i) steps, that is, our distance to the sink must have been reduced by at least i - f(i). This is only a heuristic, but it seems to work quite well in practice.

Let us explain the path-growing procedure. We are given d and the waste function f, and suppose we have grown a partial path P in the state graph, starting from the source but not yet having reached a sinkstate. Suppose also that the last node of P ((u,s) say) has just been added to it, and P currently has i-1 edges. We randomly order the edges of the network incident with u (unless they have already been reordered in growing this path), and try all these edges in order until we find one, uv say, so that if (v,t) denotes the corresponding neighbour of (u,s) in the state graph, then: (a) t does represent a legal state (not the extra state meaning violation) (b) (v,t) is not currently in the path P (we permit (v,t') to be in P for t' different from t), and (c) the length of the shortest path in the state graph from (v,t) to a sinkstate is at most d - i + f(i). If we find such an edge uv we add it to P and repeat. If there is no such edge then we discard the final node of P, reduce i by 1, and try the next neighbour of what is now the last node of P. When we reach a sinkstate the process terminates.

This eventually will find a legal path from source to sink of length at most d, if there is one. The above is called repeatedly starting with d = n, and after "ntries" calls with no good new paths

found we increase d by 1, and repeat until d exceeds hoplim. (ntries is a parameter set by the user: ntries = 500 is reasonable.)

It remains to explain what is a "good new path" - that is the next part.

Part 5. Selecting a diverse subset of the paths found

Part 4 above provides a fast subroutine, which will generate random legal paths in the network from source to sink, shorter paths first. We need to make a good selection of some of these to output. At any step we will keep a list of the best 10 (say - this parameter is set by the user) paths found so far. Now we are presented with another legal path.

First, we check it for minimality - that if it passes through some node more than once, then shortcutting does not give a legal path. (If it is not minimal we abandon it and wait for another proposal from part 4).

Next we check that this path is not already in our top ten. (It could be; this is a random process, there is no checking for repetition until this point.)

If not, we try replacing each member of our top ten by it, in turn, and check whether the new set of 10 paths we get is more diverse than the old set. If so we make the replacement, and otherwise retain the old set.

To measure the diversity of a set of 10 paths, we measure, for each of them, its maximum "closeness" to any other member of the set, and add the answers; the larger the total the smaller the diversity. The closeness of two paths means the sum of (a) "nodecost" times the sum of the lengths of the two paths (nodecost is a parameter set by the user - high nodecost, e.g. nodecost = 20, will give a set of paths which tend to be shorter but may have more nodes and edges in common, while nodecost = 1 will do the reverse) (b) 3 times the number of nodes they have in common (c) the number of edges they have in common.

2.3 Random Feasible Paths (RFP) Algorithm

Input and output for the algorithm are in the standard formats, specifying an undirected graph G with a type for each node and for each arc, the source s and destination t, the number of desired paths, and constraints on allowable paths.

Constraints and State Graphs.

The constraints are all of the form: the count of nodes or edges of some particular type over a path lies in some range. The standard input format requires that no node or edge be of more than one type. The algorithm itself as coded allows for an edge to have a vector of attributes, and allows constraints such as: the count of edges whose third attribute is four should be between one and eight. A hop limit is also of this form.

We transform the input graph G=(V,E) into a directed state graph G'=(V',E'), where each node v' in V' corresponds to a node v in V, together with for each constraint, a count of objects of the specified type. The counts can range from zero up to the maximum allowed in the constraint. If a constraint has no maximum, or if the maximum equals or exceeds the maximum of another

constraint that counts a superset of the objects of the original constraints, we can stop counting as soon as we hit the minimum. (With the standard input format for constraints, this can only happen if the other constraint is the hop limit.) Note that the hop count is included in the state information.

An edge e'=(u',v') in E' exists if e={u,v} is in E, and if traversing e from u to v would increase the counts in u' to the counts in v'. The counts in v' must not exceed the constraints. Counts for constraints that have no maximums need not increase once their minimums are reached.

We are only concerned with accessible nodes -- nodes which can be reach from s'=(s,0,...,0) using edges in E'.

A path P' in G', from s'=(s,0,...,0) to some node $(t,c_1,...,c_k)$ where c_1, ..., c_k are feasible counts, corresponds (by projecting the individual nodes and edges) to an s-t path P in G that satisfies the constraints, except that different edges in P' may project to the same edge, giving repeated edges.

The RFP Algorithm

The RFP algorithm consists of three parts:

1. Pruning the original graph.
2. Searching the state graph, and counting the number of feasible ways to get to each state from the source.
3. Generating random feasible paths from source to the destination.

We will describe these three parts out of order.

Searching the State Graph

We build a representation of the part of the state graph accessible from s' by a breadth-first search, using the description of the original graph G and the constraints to determine the edges of G'. We also record how many different feasible ways there are to get from s' to each node. When we try traversing an edge e'=(u',v'), we run through each constraint to compute the counts for v', and verify that v' is feasible. We add the number of ways to get to u' to the number of ways to get to v'. We also add e' to a list of usable incoming links for v'. Because this is a breadth-first search, and because the state information includes the hop count, we visit all nodes at k hops from the source before we visit any node at k+1 hops from the source. This means the count of ways to get to u' will no longer change by the time we add it into the number of ways to get to v'.

Pruning the Original Graph

The number of states can be exponential in the number of constraints. To reduce the work that the search has to do, we prune the original graph, removing edges that we can show do not lie in feasible s-t paths for some subset of the constraints.

For a growing subset of the constraints, starting with just the hop limit we do the following. We search the accessible region of the state graph, by breadth-first search. From every feasible destination state we do a depth-first search backwards in this accessible region, marking each link of the original graph that we traverse. We then delete from the original graph the unmarked edges.

We add in the next constraint to our set of constraints and repeat, until we have pruned for all but the last of the constraints.

Generating Random Feasible Paths

We generate feasible paths randomly, so that each feasible path is equally likely to be generated. We generate the specified number of feasible paths by repeating the following procedure:

First, we add up the numbers of ways to get to each feasible state of the destination, obtaining the total number np of feasible paths. We generate a pseudo-random integer z from 1 to np. We translate this into a path working backwards, starting with the destination. We run through the destination states in some order, and pick the state so that the earlier states total fewer than z paths, but with this state totals at least z paths. We now subtract off the total of earlier states and repeat the following, until we reach the source s'.

The current value of z should range from 1 to the number of feasible paths that reach the current state. Run through the list of usable incoming links at the current state, and sum up the number of ways to reach the states at the other side of these incoming links. When the partial sum first reaches or exceeds z, choose that incoming link, prepend it to the path constructed so far, and subtract the previous partial sum from z.

This process tends to give relatively long paths, because there are usually more paths close to the hop limit than there are near the minimum hop count. This effect is worsened by the flaw that the algorithm allows repeated nodes and edges. It might be advisable to process the resulting paths to remove non-essential cycles, that is, cycles that are not required to satisfy minimum constraints.

3. **Problem Generators**

We use two types of randomly generated problems, and here the two generators are described.

Problem Generator #1

The user specifies
(a) the number of nodes and edges desired,
(b) the number of nontrivial edge-types and nontrivial node-types (there is automatically also a trivial edge- and node-type),
(c) what percentages of nodes and edges should have trivial type, and
(d) a number d. (d will be smaller than, but roughly equal to, the minimum distance from source to sink in the network.)

Let n be the number of nodes, and let c = n/d, rounded down. The program generates nodes numbered from 1 to n; and then repeatedly chooses a pair u,v of numbers between 1 and n, so that $0 < |u-v| <= c$, uniformly at random, and adds the edge uv. This is repeated until the number of edges is as specified. (There is no check for parallel edges.)

Then, again randomly, types are assigned to the nodes; the probability of getting the trivial type 0 is as specified by the user, and the other types all have equal probability. The same procedure is repeated for edge types. Nodes 1 and n get trivial type 0.

This procedure allows us to generate networks of varying size (in terms of locations and links), varying "width" (i.e., distance from source to sink), and varying number of constraints. This allows us to exercise the algorithms and evaluate their performance across a broad range of problem types.

Problem Generator #2:

This problem generator was designed to produce networks, which exhibit some local "geography". It is considerably more complicated (and slower) than the first generator described above. The process breaks into three stages:

Stage 1: The user specifies a width and height for the display; let R be a rectangle with the specified dimensions. The user specifies a number of centres, and a minimum distance between centres. The program chooses points from R one at a time, retaining a point (and calling it a "centre") if its euclidean distance from all the previously selected centres is at least the minimum distance. This process continues until the specified number of centres has been chosen or for 1000 tries, whichever is first.

Stage 2: The user specifies the number of nodes n, and a number called radius, which should be less than half the minimum distance input previously. The program randomly selects points from R, and checks whether its distance from some centre is at most the radius. If so the point is retained, and called a node of the graph (and the centre it is close to is called its centre). This is repeated until the required number of nodes is obtained. One exception: in order to ensure that the source and sink are reasonably far apart, we choose the two centres that are furthest apart, and make sure that the first node is chosen from one of these two centres, and the last node from the other.

This results in a selection of nodes, each with a position in R, and falling into groups, one group around each centre. Next we select the edges. There will be two kinds of edges, long ones (between nodes in distinct groups) and short ones (between nodes in the same group).

Stage 3: The user specifies how many short edges are wanted. (The program in general may find considerable fewer than specified.) The user species the number of nontrivial short edge types (there is automatically also a trivial type). For each type t in turn, the program then randomly (uniformly) selects a pair u,v of distinct nodes, and checks that u and v belong to the same group, and have not yet been joined by an edge, and that the straight line in R between u and v will not cross any other edge with type t; and if all these tests are satisfied, it adds the edge uv to the network with type t. This is repeated until the number of edges with type t is correct (equal to the total number of short edges divided by the number of short edge types) or for 10000 tries, whichever is first.

Stage 4: For each group of nodes, the program checks that the subgraph induced on the group is connected, and if not adds more short edges joining nodes in different components of this group until it becomes connected (again, trying to distribute types equally, and insisting that no two edges of the same type can cross). It continues until it succeeds or for 10000 tries.

Stage 5: Now we repeat stage 3 for long edges.

Stage 6: We check if the whole network is connected, and if not try to add more long edges joining nodes in different components until it becomes connected, as usual keeping the number of

long edges of each type the same as far as possible, and making sure that no two edges of the same type cross.

Stage 7: So far, all nodes and edges have nontrivial type. Now the user specifies what proportion of the nodes and of the edges should have trivial type, we randomly select a set of the edges of the specified size, and change their types to trivial, and repeat for the nodes. The first and last node gets trivial type.

4. Computational Results

In this section, we examine the suitability of our algorithm for solving problems that are likely to arise in practice.

We tested the DFP algorithm (see Section2.2) and the RFP algorithm (see Section 2.3) on problems generated by our problem generators (see Section3). The networks have the following characteristics:

Problem	**Nodes**	**Edges**	**Generator**
Prob1	200	400	1
Prob2	1000	3000	1
Prob3	3000	30000	1
Prob4	5000	100000	1
Prob5	200	422	2
Prob6	1000	2484	2
Prob7	3000	6741	2

Edges were of five types and nodes of three types, including type zero, which cannot appear in bounds (other than the hop limit). For each graph we requested 10 paths from the first node to the last node, subject to a hop limit of 20. We used three different sets of edge and node bounds. "limits0" had no node or edge bounds beyond the hop limit. "limits1" forbade edges of type two and three altogether, required exactly one node of type one, and restricted the other edge types. "limits2" restricted all edge types, and required exactly one node of type one and exactly one node of type two.

We tested the algorithms on Applied Research public compute server "wind", at this time a Sun S670 with a number of other users. The time reported is user CPU time in seconds. The space is the "average amount of unshared data space used in Kilobytes", reported by the csh "time" command. The average length (in edges) of the ten paths generated is given next. The last column is the average, over all ordered pairs of paths, of the fraction of edges in the first path that lie in the second path.

The results for the DFP algorithm (see Section 2.2) are contained in the following table.

Prob	Constraints	Time (secs)	Space (bytes)	Avg Path Length	Avg Path Overlap
Prob1	Limits0	0.1	268K	7.9	40.46%
Prob1	Limits1	0.4	456K	10.8	37.86%
Prob1	Limits2	0.9	964K	12.0	32.31%
Prob2	Limits0	0.4	548K	11.0	18.18%
Prob2	Limits1	1.8	1400K	14.0	29.78%
Prob2	Limits2	3.0	3588K	16.7	27.72%
Prob3	Limits0	3.0	3056K	11.0	13.94%
Prob3	Limits1	8.1	5460K	15.6	22.94%
Prob3	Limits2	13.7	11676K	16.6	17.78%
Prob4	Limits0	9.3	9220K	11.0	17.78%
Prob4	Limits1	18.9	12948K	14.9	13.48%
Prob4	Limits2	29.0	22860K	17.10	34.41%
Prob5	Limits0	0.2	304K	7.6	10.58%
Prob5	Limits1	0.3	472K	10.8	30.27%
Prob5	Limits2	0.8	960K	9.0	19.44%
Prob6	Limits0	0.4	496K	5.9	35.67%
Prob6	Limits1	1.0	1172K	9.1	17.22%
Prob6	Limits2	2.4	3232K	8.1	37.48%
Prob7	Limits0	1.3	892K	7.2	14.09%
Prob7	Limits1	3.7	3112K	9.3	40.97%
Prob7	Limits2	7.9	9324K	14.7	23.42%

The results for RFP algorithm (see Section 2.3) are contained in the following table.

Prob	Constraints	Time (secs)	Space (bytes)	Avg Path Length	Avg Path Overlap
Prob1	Limits0	0.3	648K	19.7	17.76%
Prob1	Limits1	8.1	3676K	19.6	26.65%
Prob1	Limits2	23.6	9548K	19.0	49.43%
Prob2	Limits0	2.5	2680K	20.0	7.22%
Prob2	Limits1	61.1	16324K	19.7	36.53%
Prob2	Limits2	120.8	29492K	19.9	36.23%
Prob3	Limits0	38.5	20764K	18.2	2.09%
Prob3	Limits1	764.4	89140K	19.9	19.50%
Prob3	Limits2	1294.3	37744K	19.9	12.51%
Prob4	Limits0	174.8	63660K	18.5	1.46%
Prob4	Limits1	3465.5	*****K	19.9	11.29%
Prob4	Limits2	1523.7	*****K	19.7	44.09%
Prob5	Limits0	0.3	692K	19.9	10.72%
Prob5	Limits1	8.5	3820K	19.0	28.87%
Prob5	Limits2	22.4	8860K	19.0	29.52%
Prob6	Limits0	2.8	2884K	20.0	6.33%
Prob6	Limits1	84.4	21164K	19.2	12.71%
Prob6	Limits2	197.1	45948K	19.0	17.59%
Prob7	Limits0	10.7	7632K	19.9	1.56%
Prob7	Limits1	112.0	29220K	19.1	44.06%
Prob7	Limits2	169.1	33764K	19.5	18.78%

The space number for prob4, limits1 was negative, and thus certainly an overflowed variable. The space number for prob4, limits2 seemed anomalously small, and may have also been an overflow.

The results for the RS and QTRS algorithms (see Section 2.1) are contained in the following table. It is important to note that no constraints, beside the hop limit, are enforced by either the RS or QTRS algorithms and that they only guarantee to produce a short path as opposed to a list of good paths. These algorithms are simply used as a benchmark against which the later algorithms that do incorporate multiple constraints and give a list of candidate paths are measured. Hence, these results are only for unconstrained (Limit0) versions of the problems.

Problem	Method	Time (secs)	Space (bytes)	No. Paths	Avg Path Length	Avg Path Overlap
Prob1	RS	0.0	200K	17	8.35	40.71
Prob1	QTRS	0.0	156K	1	7	***
Prob2	RS	0.2	324K	13	11	20.75
Prob2	QTRC	0.0	284K	4 1	1.5	25.88
Prob3	RS	354.4	1684K	523	11	20.60
Prob3	QTRS	1.6	1100K	8	11.62	12.07
Prob4	RS	82548.3	844K	16	1116 11	***
Prob4	QTRS	8.6	3236K	15	11.33	8.19
Prob5	RS	0.0	180K	11	7.90	21.32
Prob5	QTRS	0.0	196K	4	7.5	27.38
Prob6	RS	0.1	292K	14	6.71	35.67
Prob6	QTRS	0.0	272K	16	6.88	73.21
Prob7	RS	0.5	484K	15	7.4	21.27
Prob7	QTRS	0.2	416K	20	8.25	29.34

Two table entries contain asterisks since the QTRS algorithm for Problem 1 only generated one path so there is no comparison to be made and the RS algorithms for Problem 4 generated too many paths to compare.

We note that both the RS and QTRS algorithms are very fast and generate paths with short hops. However, the number of paths generated by the QTRS algorithm is generally quite small.

In addition, since constraints are not taken into account and since these algorithms will not generate many paths beyond those with nearly minimum hop values, that these algorithms are not useful when there are constraints, as we shall see next.

We ran the RS algorithm and allowed it to generate all possible paths that it could find to see now many would be feasible. This is summarized in the following table with the conclusion being that when there are constraints, it is very, very unlikely that a feasible path will be found. Naturally, there are many possible paths (including feasible ones) which the RS algorithm simply did not find. (The same would be true for the QTRS algorithm as well.)

Problem	Time (seconds)	No. Paths	No. Feasible (Limits1)	No. Feasible (Limits2)
Prob1	0.1	51	0	0
Prob2	1.9	291	0	0
Prob3	4215.5	51134	0	0
Prob4	***	***	***	***
Prob5	0.2	54	0	1
Prob6	1.3	78	0	0
Prob7	5.9	93	1	0

5. Concluding Remarks

We have presented various algorithms to help automate the process of route generation to provision end-to-end circuits through a transport network. The main feature of these algorithms is the ability to generate a list of good paths all satisfying certain constraints. We have tested these algorithms on small-medium networks with 200 nodes and 400 edges, to much larger networks with 3000 nodes and 6741 edges. We have also included up to five edge constraints and three node constraints.

For problems with no constraints, the RS algorithm quickly generates a list of paths with minimum or near-minimum hop length; it does not take into account diversity, however. The QTRS algorithm does not allow specifying the number of paths to generate and often the number of paths generated is quite small. In either case, when there are constraints, even generating a huge number of paths does not yield any path that satisfies the constraints. **The RS algorithms could be used if a list of paths with short hop lengths is desired and there are no other constraints or costs associated with the problem. However, even in this case, the DFP algorithm is comparable in time and better in route diversity and probably should be used as an alternative.**

For problems with several constraints, the DFP and RFP algorithms find a set of paths that are "diverse" as well as relatively short. However, the latter algorithms require considerably more time and space to be able to handle these constraints and to find diverse paths. (The DFP algorithm required at most 29 seconds of CPU time and 22Mb of space, while the RFP algorithm occasionally required much more time and space for highly constrained problems. See Section 4 for more details.)

So for problems with several constraints, the DFP algorithm quickly produces a list of relatively short feasible paths with the additional benefit of added diversity and is preferable to the RS, QTRS and RFP algorithms. We have successfully solved the problem originally posed in the NCON routing algorithms project document. However, it remains to be seen what types of problems and constraints arise within the real NCON systems and based on inputs from real customers. Also, it remains to be seen whether this approach can be incorporated into the NCON system in an efficient and effective manner.

Appendix A. Standard Input Format

The algorithms all use the same input format, and in this section it is described. There must be two input files, which can be called anything but for our purposes here will be called "graphfile" and "limitsfile".

In graphfile, the network is described, together with the type of each node and each edge. In limitsfile, the demands and constraints are specified.

Here is a sample graphfile:

```
4 5
1 0 128.040531 28.922303
2 0 127.727730 16.768486
3 1 140.510495 19.950633
4 0 106.973539 175.129595

1 2 1 0
2 1 3 1
3 1 4 1
4 2 4 0
5 2 3 0
```

There must be a carriage return at the end of every line. The first line specifies the total number of nodes and of edges (4 and 5 in this case). The second line is blank. The next set of lines (one for each node) have four entries:
(a) the number of the node (nodes must be numbered consecutively from 1 up to the number of nodes, so the ith node line must start with the number i),
(b) the type of the node (any nonnegative integer < 100), and
(c) two doubles, giving the coordinates of the position of the node in the x,y-plane, for use with a graphical interface.

Then there is a blank line, and then one line for each edge. The line for edge j contains again four entries:
(a) the number of the edge (the edge described in line j must have number j),
(b) the numbers of the two ends of the edge
(c) the type of the edge (any nonnegative integer < 100).

Here is a sample limitsfile:

```
source 1
sink -1
nwanted 10
hoplim 20

2 edgebounds
1 1
2 3

2 nodebounds
1 0 2
2 1 100
```

There must be a carriage return at the end of every line. The first two lines give the numbers of the two nodes that we are to find paths between; they should read "source a" and "sink b", where a and b are integers. The number a must be the number of a node. The number b may be the number of a node, or it may be negative, when it means the node with number n + b + 1, where n is the total number of nodes.

The third line reads "nwanted w", where w is a positive integer; w will be the total number of paths in the set of paths output by the algorithm.

The fourth line reads "hoplim h", where h is a nonnegative integer; this instructs the algorithm only to consider paths with at most h edges.

Then there is a blank line, and a line "p edgebounds", where p is a nonnegative integer, the number of edge types for which there is a constraint.

Then there are p lines, one for each edgebound. Each line must be of the form "x y" where x is a positive integer (< 100) and y is a nonnegative integer. This line will enforce that the selected paths will each have at most y edges with type x. (So the example above requires that every path has at most 1 edge of type 1 and at most 3 edges of type 2.) Note that these edgebound lines cannot involve edgetype 0, although there may be edges with type 0; such edges are "free". Also, note that we do not permit any lower bound on the number of times a path uses edges of a given type, only an upper bound.

Then there is a blank line, and a line "q nodebounds", where q is a nonnegative integer, the number of node types for which there is a constraint. Then there are q lines, one for each node constraint. Each node constraint line is of the form "x y z", where x is a positive integer (< 100), and y and z are both nonnegative integers with y no bigger than z. This line will enforce that every selected path has at least y and at most z nodes with type x. Once again, no constraint can involve nodes with type 0; such nodes are free. The example above requires that every path contains at most two nodes of type 1, and at least one node of type 2. (The other two constraints are trivial, since hoplim = 20 makes the upper bound of 100 effectively infinite.)

Appendix B. Standard Output Format

This describes the common format of the output from all the algorithms. Here is a sample output file:

```
5
path 1 length 5 1 1102 982 437 202 1032 65 265 519 2182 1000
path 2 length 6 1 161 955 196 565 1355 327 395 132 872 519 2182 1000
path 3 length 6 1 161 955 712 574 1934 202 1032 65 265 519 2182 1000
path 4 length 6 1 1102 982 427 305 1516 238 834 65 265 519 2182 1000
path 5 length 6 1 1102 982 1558 923 613 49 245 132 872 519 2182 1000
```

The first line consists of one nonnegative integer, p say, which will be the number of paths described in the file. Then there follow p more lines, one for each path. The line describing the ith path will start "path i length j" where j is the number of edges in the path. Then there follow 2j+1 integers, which are the numbers of the nodes and edges in the path as it is traversed from source to sink. (So in the example above, the nodes of path 1 in order, starting from the source, are: 1, 982, 202, 65, 519, 1000; and its first edge has number 1102 and joins nodes 1 and 982.)