Journal of Network and Systems Management, Vol. 3, No. 4, 1995

A Route Pre-Computation Algorithm for Integrated Services Networks

Jean-Yves Le Boudec and Tony Przygienda

We provide an algorithm for computing best paths on a graph where edges have a multidimensional cost, one dimension representing delay, the others representing available capacity. Best paths are those which guarantee maximum capacity with least possible delay. The complexity of the algorithm is of the order of $O(V^3)$ in the bidimensional case, for a graph with V vertices. The results can be used for routing connections with guaranteed capacity in a communication network.

KEY WORDS: Communication networks; guaranteed bit rates; integrated services.

1. INTRODUCTION

1.1. Background

We consider communication networks that support connections with *guaranteed bit rates*; examples for such networks are multirate circuit switched networks, ATM networks, and networks that support IP extensions such as ST.II or RSVP [1, 2]. We call such networks *integrated services* networks. We are interested in the problem of *routing* connections in integrated services networks; routing refers here to finding a path (called *route* in this paper) through the network that is able to support the connection bit rate requirement, and maybe other requirements such as minimum delay.

Figure 1 shows a schematic view of the routing function as can be found for example in [3]. The routing function can be located in network nodes as part of their control point, or in stand-alone route servers. In the application of the algorithm presented in this paper, we assume that the routing function has at its disposal an *image* of the network, stored in the topology database. The network image contains a description of all links, including how much capacity is already allocated to existing connections. It is created from Link State Updates, sent by all nodes in order to propagate the status of their attached

[1] EPFL-LRC, Ecublens (INR), 1015 Lausanne, Switzerland. (E-mail: leboudec@di.epfl.ch.)
[2] IBM Zürich Research Lab, Switzerland. (E-mail: prz@zurich.ibm.com.)

1064-7570/95/1200-0427$07.50/0

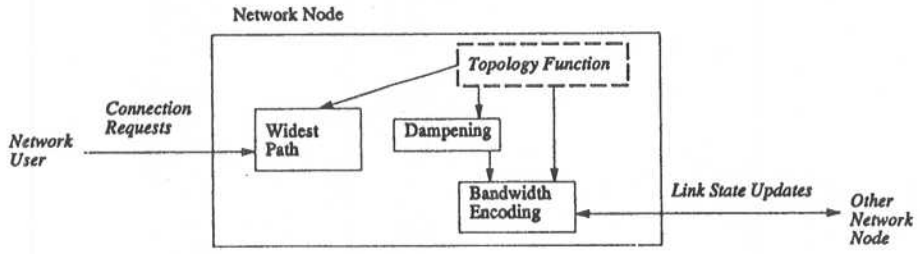

Fig. 1. Conceptual view of routing function.

links. Link states updates are not sent every time the allocation of link capacity is changed, but only when significant changes occur (typically due to accumulation of small changes). The *Dampening* function controls the release of link state updates.

Routing can be performed in many different ways, depending on a number of network design options. Type of routing is classified according to time of topology exchange, and time and place of route creation [4, 5]. Here, we are particularly concerned with the choice of the point in time at which a route is computed; we consider two possibilities: the route may be either computed on demand (at the time when a request for connection setup is presented), or pre-computed and stored. Route pre-computation is attractive for cases where high connection setup rates may be expected to put a high load on real-time resources, but where connections generally request relatively small amounts of link capacity, thus resulting in relatively infrequent link state updates (due to the dampening function).

Route pre-computation is commonly used in many networks [6–11]. However, using route pre-computation in integrated services networks is not easy because the route computation function does not know in advance what the characteristics of the connections to be routed will be, and in particular, its bit rate requirements.

We present here an algorithm that analyzes the network image and outputs lists of optimal routes to all destinations. The list of routes for one destination is not dependent on any *a-priori* classification of the connections to be routed, but it guarantees that no better route can be found than those in the list, regardless of connection requirements at setup time. One way of viewing the algorithm is to consider that it performs an automatic classification of potential connection requests, based on the instantaneous network status, and provides a route for each of the resulting classes.

1.2. What our Method Provides

Before introducing our algorithm, we first explain on an example how it is intended to be used by the Route Computation function.

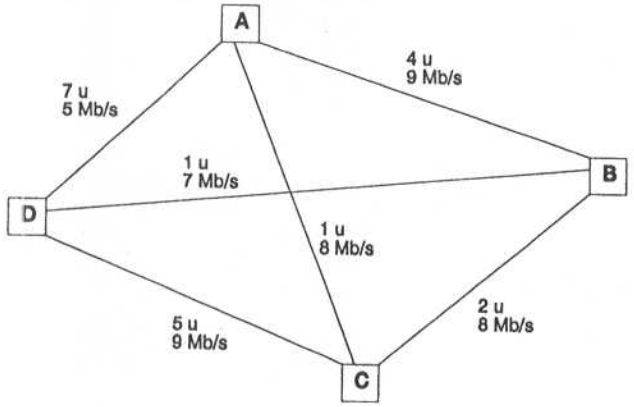

Fig. 2. Sample network (u = cost unit, for example # of hops or delay).

Consider the four-node network illustrated in Fig. 2. We assume that there exists only one link per node pair. Every link has two attributes. The first one represents the static cost of the link; it is the same cost as is used in standard routing algorithms such as OSPF [8]. It is *additive* in that the attribute value for a route (concatenation of links) is the *sum* of the link attributes. To give an example, assume this sample network is a network of ATM Virtual Channel switches, and the links are Virtual Path (VP) Connections; then the cost could represent the number of VP cross-connected hops. The second attribute is dynamic; it represents the link capacity that remains available for reservation. We call it *restrictive* in that this attribute value for a route is the minimum of the link attributes. For the sake of simplicity in this introduction, we assume that the capacity is always allocated symmetrically to both directions on the link (but this is not a restriction of our method).

Table I lists all routes between nodes A and B, with their attributes. We say that a route is better than some other one if it offers at least as much capacity

Table I. Routes on the Example Graph from A to B.

Route	Hops	Capacity	Additive cost
Elementary Routes			
B_1	A, B	9 Mb/s	4 u
B_2	A, C, B	8 Mb/s	3 u
B_3	A, D, B	5 Mb/s	8 u
B_4	A, D, C, B	5 Mb/s	14 u
B_5	A, C, D, B	7 Mb/s	7 u
Extremal Routes			
B_1	A, B	9 Mb/s	4 u
B_2	A, C, B	8 Mb/s	3 u

on its most loaded link, and has a lesser or equal end-to-end delay (or number of hops). With our definition, it is possible that route R is better than route R' and reciprocally, in which case the costs of R and R' are equal. We say that route R is *strictly better* than route R' if R is better than R' and the costs of both are not equal in both of the components. (Thus either R has more capacity and less or equal delay, or it has as much capacity but less delay.) Here, route B_1 is better than routes B_3, B_4 and B_5 (it has more capacity and less cost), and so is B_2. However, B_1 and B_2 cannot be compared. If a connection requires less than 8 Mb/s, then it is probably a good idea to choose route B_2 because it has a lower cost. However, if a connection requires 9 Mb/s, then there is no alternative but to choose B_1. We see that the nature of the problem is such that there exists no best route.

We say that routes B_1 and B_2 are *extremal*: there exists no other route that is strictly better. The algorithm presented in this paper computes all extremal routes for a given source-destination pair. In the example, it would produce the list (B_1, B_2). Generally, the structure of such routes is a lattice [12] and the set of extremal routes the *lower bound* of the lattice.

The algorithm has the following properties:

- The algorithm supports route *precomputation*, namely, it can be used to compute and store routes in advance of connection setup requests.
- Computed routes offer both bit rate guarantees and minimum delays (or minimum number of hops).

Typically, the algorithm can be used to support the Route Computation function of Fig. 1 as follows. The algorithm is run in the background; the result is, for any given source and destination nodes, a set of extremal routes. Those routes are stored and made available to the next connection setup requests; when a connection setup request is processed, the set of extremal routes is scanned and the route with the least cost that can accommodate the required bit rate is selected. Other uses are possible, in particular for various optimization purposes, but such a discussion is outside the scope of this paper.

For an ATM network, it is usually required that connections follow the same path in both directions, although bandwidth and delay requirements normally differ for the direct and return paths. Our method can be applied to such cases with a three dimensional cost as follows.

- The first component (additive) of a link cost is the transit delay, or any cost reflecting the value of the link.
- The second and third component (restrictive) represent the available capacity for the two directions of the link.

The method thus allows pre-computation of routes with bidirectional bandwidth constraints in polynomial time.

If the restriction that the two directions of a connection follow the same path is released, than the two dimensional model can be used instead, with the second component reflecting the available capacity for the downstream link direction. In such a case, two separate routes are computed at the two ends of the connection (unlike for the previous case where only one route exists for both direct and return paths).

The algorithm presented in this paper supports peak rate allocation, if applied to ATM networks. Application to the support of sustainable cell rate allocation is possible; with the equivalent capacity method [13], this would use a number of restrictive costs to account for the equivalent capacity, mean and variance components of the link metrics. This is the object of ongoing work.

1.3. The Algorithm in this Paper

Consider a graph where edges have a multidimensional attribute $(c_1, \ldots, c_r)$. The attributes are used to associate a multidimensional "cost" $(C_1, \ldots, C_r)$ to every route (concatenation of links), using the following rules:

- C_1 is the *sum* of all the attributes c_1 for all links that constitute the route;
- C_i is the *minimum* of all the attributes c_i for all links that constitute the route; this holds for $i = 2 \ldots r$;

In this context, a route R with cost C is said to be *better* than a route R' if

- $C_1 \leq C_1'$ and
- $C_i \geq C_i'$, for $i = 2 \ldots r$

Route R is said to be *strictly better* if it is better than route R' and the costs of the two routes are not equal, i.e. if

- $C_1 \leq C_1'$ and
- $C_i > C_i'$, for $i = 2 \ldots r$

or

- $C_1 < C_1'$ and
- $C_i \geq C_i'$, for $i = 2 \ldots r$

An *extremal* route is defined to be a route for which there exist no strictly better route. As explained earlier, there does not exist in general only a single best route.

The link attributes, or cost, thus consist in one additive dimension, and $r - 1$ restrictive dimensions; R is a better route than R' if it has lower or equal additive cost, but higher or equal restrictive cost, for all of the restrictive dimensions.

When used in a routing context, the vertices of the graph represent network

nodes and the edges represent network links. The additive cost represents a monetary cost, a delay or a number of hops, whereas the restrictive costs represent various aspects of the available capacity. For example, with $r = 3$, the two restrictive components may represent the capacity in both directions of the link. Link costs supporting equivalent capacity [14] may also be represented by higher values of r.

We say that two extremal routes, both with same source and destination, are *equivalent* if they have equal costs. The algorithm presented in this paper produces, for a given vertex on the graph, a *complete set* of extremal routes to all destination vertices. A complete set of extremal routes between nodes i and j is a set S of routes such that, for every extremal route R between i and j, either R is in the set S, or there exists a route in S that is equivalent to R.

The paper is organized as follows. In Section 2, we present the algorithm for the two-dimensional case ($r = 2$) and prove its correctness. In Sections 4 and 6, we analyze its complexity and efficiency. In Section 3, we give the extension for higher dimensions. In Section 5, we mention possible optimizations. The complete algorithm for the case $r = 2$ is given in appendix.

The application of the method to multicast connections is not considered due to the fact that multicast solutions can be solved in an orthogonal way to unicast routing [15]. Also, the considerations of problems with two additive components is omitted due to the well-known fact that this problem is NP-complete [16]. Several authors published either detailed overviews of such multidimensional non-polynomial path problems [16, 17] or proposed algorithms [18–23] to deal with them. The solutions are unfortunately NP-complete or not workable in practical terms. To our knowledge, the best proposal [18] gives an algorithmical approach which runs in pseudo-polynomial time $O(b * n^5 * \log(n * b))$ for n vertices and b as largest possible weight or length.

For a network with 30% connectivity (30% of all possible node pairs have a link between them, according to a uniform distribution), and with 50 nodes, the algorithm for the 3 dimensional case produces a list of 45 routes in average for one source destination pair.

2. THE ALGORITHM FOR THE TWO-DIMENSIONAL CASE

We present in this section the algorithm for the two-dimensional case ($r = 2$).

2.1. Description

2.1.1. Specification

The algorithm takes as input a non-directed, valued graph $G = (V, E, c)$, where V is the list of vertices, E the list of edges, and c a function that attributes

to every edge e a two-dimensional cost $c(e) = (c_1, c_2)$. As introduced earlier, the first component c_1 is handled as an additive cost, while the second component c_2 is handled as a restrictive cost.

The output is a complete set of extremal routes from node A to all other nodes in the graph.

2.1.2. *The Algorithm*

First, the list of edges is sorted according to increasing restrictive costs resulting in a new, sorted, list of edges E'. The algorithm uses a candidate route list L which, at the end, is the complete set of extremal routes from node A to all destinations. It works as follows.

```
While E' IS NOT EMPTY Do
    COMPUTE A SHORTEST SPANNING TREE T, FOR SOURCE NODE A AND FOR THE
            ADDITIVE COST, ON THE SET OF EDGES E';
    ENQUEUE ALL ROUTES IN T INTO THE CANDIDATE ROUTE LIST L AND
            WHILE ENQUEUEING A NEW CANDIDATE ROUTE, REMOVE
            EXISTING ROUTES THAT ARE COMPARABLE AND LESS GOOD, AND
            DROP THE CANDIDATE IF A BETTER ROUTE EXISTS ALREADY;
    REMOVE FROM THE TOP OF E' ALL LINKS THAT HAVE SAME RESTRICTIVE
            COST AS THE FIRST LINK IN E' (NAMELY, THE LINKS WITH THE
            MINIMUM RESTRICTIVE COST IN E);
End;
End;
```

A complete version of the algorithm is given in the appendix; in that version, the shortest path tree computation uses the Dijkstra algorithm.

2.1.3. *Example*

We explain how the algorithm works on the sample network of Fig. 2. For the restrictive components instead of capacities, the value "10-*capacity*" has been used to make smaller numbers act for better routes as for the additive component. Table II lists all the non-looping routes and Table III shows the initial value of the sorted set of links E'.

Figure 3 illustrates the steps followed by the algorithm, showing the shortest path tree and the candidate list. The first picture shows the status at the end of the first iteration, with the first spanning tree that covers the whole set of edges, and just after removal of link (A, D). The removal of this link does not change the spanning tree, so the next iteration will not have to recompute a shortest path tree. The second picture shows the spanning tree after link (B, D) has been removed. This causes a second path to be added to the candidate list for destination D. In the following step, all links with restrictive cost (c_2) more than or equal to that of link (B, C) are removed from the list E', and by now a very small spanning tree is computed. This still causes the addition of a new route to vertex B to be included into the candidate list. On the last picture, the list E' is empty and the computation stops.

Table II. All Elementary Routes for the Example.

Route	Hops	Cost
B_1	A, B	(4,1)
B_2	A, C, B	(3,2)
B_3	A, D, B	(8,5)
B_4	A, D, C, B	(14,5)
B_5	A, C, D, B	(7,3)
C_1	A, C	(1,2)
C_2	A, B, C	(6,2)
C_3	A, D, C	(12,5)
C_4	A, B, D, C	(10,3)
C_5	A, D, B, C	(10,5)
D_1	A, D	(7,5)
D_2	A, B, D	(5,3)
D_3	A, C, D	(6,2)
D_4	A, B, C, D	(11,2)
D_5	A, C, B, D	(4,3)

2.1.4 Proof

The algorithm is actually an application of a search for extremal values on the set of *costs*. Some formalism needs to be introduced at this stage.

Definition 1. Define the comparison $\leq$ on $\mathbf{R}^2$ by: $(x_1, x_2) \leq (y_1, y_2)$ iff $x_1 \leq y_1$ and $x_2 \leq y_2$ and the *strict* comparison $<$ by $x < y$ iff $x \leq y$ and $x \neq y$.

Definition 2. Now let $E \in \mathbf{R}^2$ be a finite set of couples of numbers. We say that x in E is extremal iff there exist no y in E such that $y < x$.

It is straightforward to relate these formal definitions to the ones used in the rest of this paper: a route R is better than a route R' iff $(c_1(R), K - c_2(R)) \leq (c_1(R'), K - c_2(R'))$ and a route R^- is extremal if its modified cost $(c_1(R),$

Table III. Links of the Example Graph Sorted by Bandwidth.

Link From/To	Cost
A, D	(7,5)
B, D	(1,3)
B, C	(2,2)
A, C	(1,2)
C, D	(5,1)
A, B	(4,1)

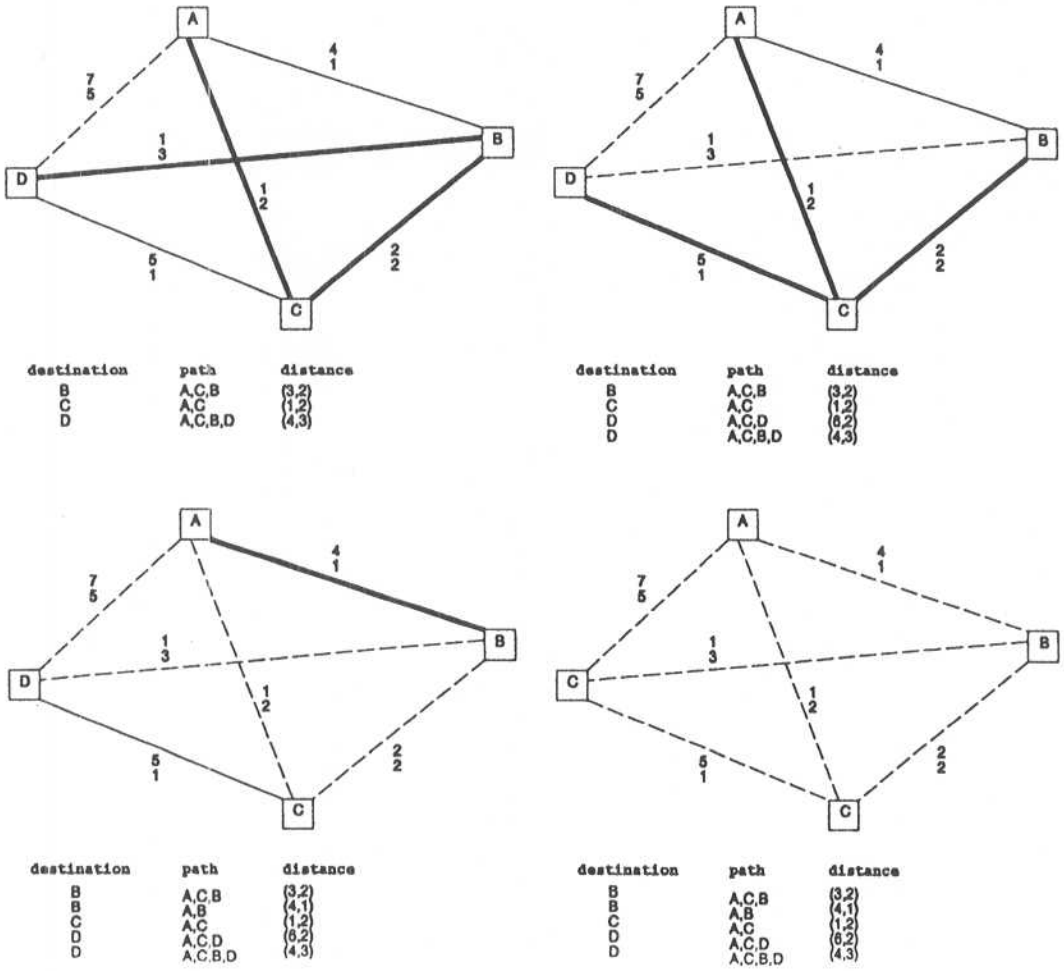

Fig. 3. Computation of the solution space.

$K - c_2(R)$), where K is some arbitrary constant, is extremal in the set of all modified costs for all possible loop-free routes. A complete set of extremal routes is a set of routes whose modified costs constitute *the* set of all extremal costs. The modified cost is introduced in order to account for the inversion in comparison for the ''restrictive'' cost. We have used the notation $c(R) = (c_1(R), c_2(R))$ for route and edge costs, so that $c_1(R)$ is the additive cost, and $c_2(R)$ is the restrictive cost.

Our main algorithm follows from another, simpler algorithm (the ''MinMax Algorithm for Extremal Points'') that determines *the* set of all extremal costs. We call it this way because it selects points with minimal first coordinate, and eliminates points with maximum second coordinate.

The MinMax Algorithm for Extremal Points. Let $E \in \mathbf{R}^2$ be a finite set of couples of numbers. The algorithm uses two subsets of E: F (the working subset) and

CL (the candidate list), as follows. In the algorithm, we use the notation $p_i(F)$ (ith projection) to denote the set of all values of x_i, for all x in F ($i = 1, 2$).

```
CL :=EMPTY;
F  :=E;
While F IS NOT EMPTY Do
      FIND ONE x IN F SUCH THAT x1 IS THE MINIMUM OF p1(F);
      PROCESS x AND CL AS FOLLOWS:
            If SOME y IN CL IS BETTER THAN x
            Then DO NOTHING;
            Else ADD x TO CL AND REMOVE FROM CL ALL z SUCH THAT x
                 IS BETTER THAN z;
            End; /* IF */
      FIND THE LARGEST VALUE c2 OF p2(F) AND REMOVE ALL y IN F
      SUCH THAT y2 = c2;
End; /* WHILE */
```

We now illustrate the MinMax algorithm on the sample network described earlier. Figure 4 plots the set of modified cost points, with the constant K equal to 10 and Table IV visualizes iterations taken by the algorithm to compute the appropriate candidate list.

Theorem 1. The "MinMax" algorithm produces the set of all extremal elements in E.

Proof. The algorithm always terminates because E is finite, so all we need to show is that, at the end of the execution of the algorithm, the candidate list is the set of all extremal elements of E. Note that, also because E is assumed to be finite, the set of extremal values is nonempty.

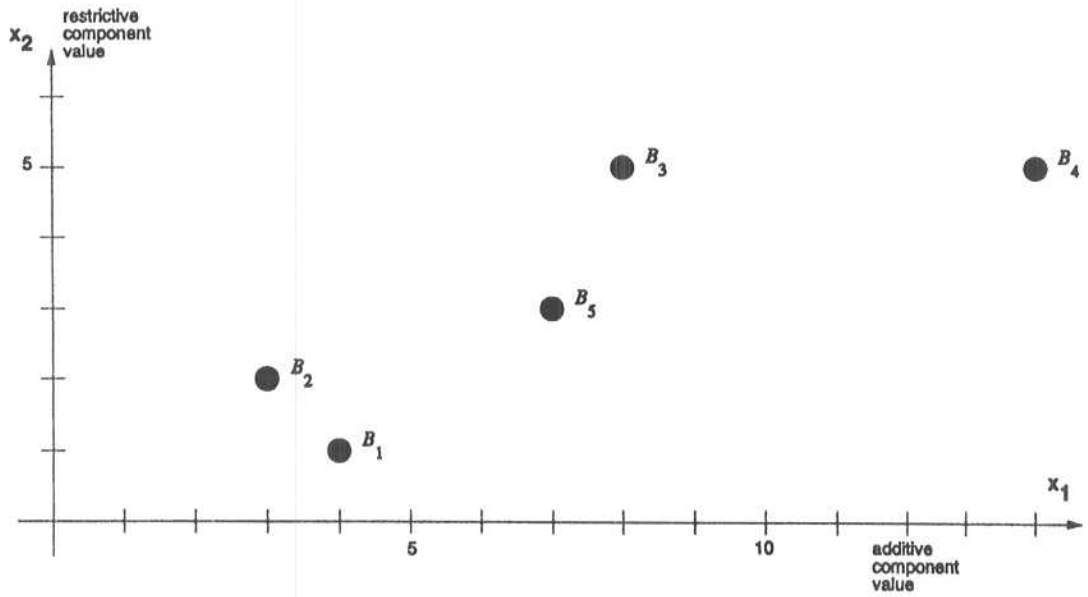

Fig. 4. The cloud of costs for the example of Fig. 2.

Table IV. Iterations on the Example Graph.

i	Actions	CL (candidate list)	F (working set)
0		$\varnothing$	$\{B_1, B_2, B_3, B_4, B_5\}$
1		$\{B_2\}$	$\{B_1, B_2, B_5\}$
	• B_2 is selected for CL • largest value for x_L is 5. B_3 and B_4 are removed from F		
2		$\{B_2\}$	$\{B_1, B_2\}$
	• B_2 is selected for CL • B_5 is removed from F		
3		$\{B_2\}$	$\{B_1\}$
	• B_2 is selected for CL • B_2 is removed from F		
4		$\{B_1, B_2\}$	$\varnothing$
	• B_1 is selected for CL • B_1 is removed from F		

Throughout this proof, we call CL_i and F_i the value of the candidate list and the working set at the end of the ith execution of the loop; by convention, CL_0 is empty and $F_0 = E$.

The demonstration uses a number of lemmas.

Lemma 1. For all i, the elements of CL_i are not comparable to one another.

Proof of Lemma 1. By induction on i. It is trivial for $i = 0$. Assume it holds for some i and call x the next element added to CL_i at some future iteration. The lemma follows from the fact that, by construction, x is not comparable to any element of CL_i. □

Lemma 2. If x is in CL_i for some i, and if y is any element of E such that $y_2 = x_2$, then $x_1 \leq y_1$.

Proof of Lemma 2. Let j be the *entry* iteration step for x into the candidate list, namely: $j := \min(\{k, x \in CL_k\})$. Since x was selected by the algorithm at step j, we have $x \in F_{j-1}$.

Now, it can easily be shown by induction that, for all k, F_k can be expressed under the form $F_k = \{(z_1, z_2) \in E, z_2 \leq \lambda_k\}$ for some λ_k. It follows now from this and the assumption $y_2 = x_2$ that $y \in F_{j-1}$. By definition of the algorithm, $x_1 = \min(p_1(F_{j-1}))$, which proves the conclusion of Lemma 2. □

Lemma 3. If x is element of CL_i for some i, and if x is *not* element of F_i, then x is extremal.

Proof of Lemma 3. Also by induction on i. For $i = 0$ it is trivial.

Assume the property holds for some i, and let m be an element of Cl_{i+1} such that m is not element of F_{i+1}. Call x the element chosen by the algorithm at the $(i + 1)$th iteration. Since $x \in F_{i+1}$, $m \neq x$. Now if $CL_i \neq Cl_{i+1}$, the only element in CL_{i+1} that is not in CL_i is x; therefore, m is also element of CL_i. Thus, if $m \notin F_i$ then by the induction assumption, m is extremal.

We can therefore assume for now that $m \in F_i$ (but $m \notin F_{i+1}$). Now there exists some $j \leq i$ such that m was added to CL_j, and since $x \in F_j \in F_i$, by construction, $m_1 \leq x_1$. Remember also that by assumption $x_2 < m_2$ because m is not in F_{j+1} and x is. Assume that $m_1 = x_1$. Then it follows that x is better than m, which is not possible because x was added to CL_i and m was not removed. We have thus proven that $m_1 < x_1$.

Now let z be some element of E which is better than m: ($z_1 \leq m_1$ and $z_2 \leq m_2$). Assume that $z \in F_{i+1}$, then necessarily $x_1 \leq z_1$ and therefore $m_1 < z_1$ which contradicts the fact that z is better than m. Therefore, z is not element of F_{i+1}. Now, since $m \in F_i$ and $m \notin F_{i+1}$, it follows that $m_2 \leq z_2$ and therefore $m_2 = z_2$. By Lemma 2, it follows that $m_1 \leq z_1$, and finally $m = z$. This proves that m is extremal and ends the demonstration of Lemma 3. □

Lemma 4. If x is extremal, then x is element of CL at the termination of the algorithm.

Proof of Lemma 4. Assume x is extremal and define j as the smallest integer such that $x \notin F_{j+1}$; there exists such an integer since at the termination of the algorithm the set F_I is empty, where I is index of the last iteration. It follows from the definition of j that for all z in F_j, $x_2 \geq z_2$. Assume x is not in CL_j and call y the candidate chosen out of F_j to be added to CL_j; we have thus $y_1 \leq x_1$ and therefore, since x is extremal, since y and x are both in F_j, and since x is extremal, $y = x$. We have thus proven that x is element of either F_{j+1} or F_j. Now since x is extremal, once it is in F_i for some i, it will never be removed. □

Proof of the Theorem. By Lemma 3, all elements of the final candidate list are extremal, since eventually the working set is empty. Now by Lemma 4, all extremal elements are in the final candidate list. □

Back to the routing algorithm, let us call $(x_1, x_2, \ldots, x_K)$ the list of all values of restrictive costs for all links in the graph, in increasing order (K is usually less than the number of edges since two edges can have the same cost components).

Let us also call F_i the value of the list of edges E' at the beginning of the ith iteration of the algorithm. Thus E_1 = the list of all edges, with increasing restrictive cost. The following lemma will be useful.

Lemma 5. For all i and for all route R, $c_2(R) \geq x_i$ iff R is exclusively made of edges in E_i.

Proof of Lemma 5. First note that E_i is the ordered list of all edges e for which $c_2(e) \geq x_i$. The proof follows from the fact that $c_2(R) = min\{c_2(e)\}$, all component edges e of route R. □

Now we can proceed with the proof of the algorithm. All we have to do is apply the basic algorithm to the set of costs, and show that the statements match. This follows directly from Lemma 5. □

Remarks. It derives from the proof that the route determination algorithm is independent of the algorithm chosen to obtain shortest paths, relative to the additive component. In the appendix we use the Dijkstra algorithm, but any algorithm can be used instead.

3. THE ALGORITHM FOR HIGHER DIMENSIONS

The extension to higher dimensions ($r \geq 3$) is straightforward, though it complicates the presentation considerably. We provide a reader-friendly form for $r = 3$:

```
Vs1 := LIST OF ALL LINKS REVERSELY SORTED ON RESTRICTIVE1 COMPONENT;
Vs2 := LIST OF ALL LINKS REVERSELY SORTED ON RESTRICTIVE2 COMPONENT;
EMPTY CANDIDATE LISTS FOR ALL DESTINATIONS;
Spf := SPANNING TREE OF SHORTEST PATHS FOR ADDITIVE COMPONENT OF THE METRIC;
For l := 1 TO LENGTH OF Vs1 Do
    Bound1 := VALUE OF THE RESTRICTIVE1 COMPONENT OF THE l-TH LINK OF V;
    For i := 1 TO LENGTH OF Vs2 Do
        If i-TH LINK OF Vs2 ON Spf OR i=1 Then
          Bound2 := VALUE OF THE RESTRICTIVE2 COMPONENT OF THE i-TH LINK OF V;
          Spf := SPANNING TREE OF SHORTEST PATHS FOR ADDITIVE COMPONENT
                 OF THE METRIC CONTAINING ONLY LINKS HAVING
                 RESTRICTIVE1 COMPONENT < Bound1 AND
                 RESTRICTIVE2 COMPONENT < Bound2;
          For j := 2 TO NUMBER OF VERTICES Do
              pth := PATH TO VERTEX j ON Spf;
              REMOVE ALL PATHS π FROM j-TH VERTEX CANDIDATE LISTS
              FOR WHICH pth'S LENGTH ≤ π'S LENGTH;
              If NO PATH π IN j-TH VERTEX CANDIDATE LIST
                FOR WHICH π'S LENGTH ≤ pth'S LENGTH Then
                INSERT pth INTO j-TH VERTEX'S CANDIDATE LIST;
                End; /* IF */
              End; /* FOR */
          End; /* IF */
      End; /* FOR */
  End; /* FOR */
```

4. COMPLEXITY

The general *worst case* complexity of the algorithm is of the order of V^{r+1}, which will not be proven. This is without taking into account possible implementation optimizations as discussed later. For the two dimensional case, the computation of the additive component spanning tree has to be performed for

all links of the graph which lead to the complexity of $E * O$ (Dijkstra spanning tree computation). Additionally, the insertion of the routes into the candidate list has to be considered. Assuming the length of the list below $\log(V)$, the insertions will be of order $O(V * \log(V))$. Both components together lead to $E * (O(\text{Dijkstra}) + V * \log(V))$. Generally, the upper bound for fully connected graphs of Dijkstra's algorithm is given by V^2 [25], [this upper bound is very pessimistic]. So $O(\text{algorithm}) = E * V^2 + E * V * \log(V)$. If E is expressed as $\alpha * V$ with α being the average connectivity per vertex, the complexity can be expressed as $O(\alpha * V^3 + \alpha * V^2 * \log(V)) = O(V^3)$.

Finally it should be mentioned that the removal of edges from the set E' may result into a disconnected graph; when this happens, the shortest path computation need not consider the vertices that are not connected to the source node A. This significantly reduces the complexity of the algorithm in an average case compared to the worst case.

5. POSSIBLE OPTIMIZATION

A vast amount of optimization work is possible on the algorithm to make it perform better.

- A first optimization is to skip the computation for the tree if the link picked up for restriction of the bandwidth is not on the spanning tree as already presented in the algorithm given earlier. This is because the tree computed in this step would be the same as in the previous one anyway.
- Another optimization is to perform an incremental spanning tree computation when a link from the tree is removed. Only certain branches need normally to be recomputed. We have implemented a naive incremental spanning tree computation where after removal of a link from the spanning tree the detached part of the tree is cleaned and a candidate list for Dijkstra computation reconstructed equal to the one found after computation steps leading to the remaining branches of the spanning tree. Although this approach is far from the optimal incremental spanning tree computation schemes mentioned later, it already showed great improvement in the runtime of the algorithm. Figure 5 shows the number of spanning trees computed (which are the most costly component of the algorithm) as function of the number of links of the graph. The connectivity (ratio of node pairs that have a direct link) α has been chosen as 0.7. Without the optimization, the number of spanning trees computed would be equal to the number of links on the graph.

Further optimizations are described here; they were not implemented in our prototype.

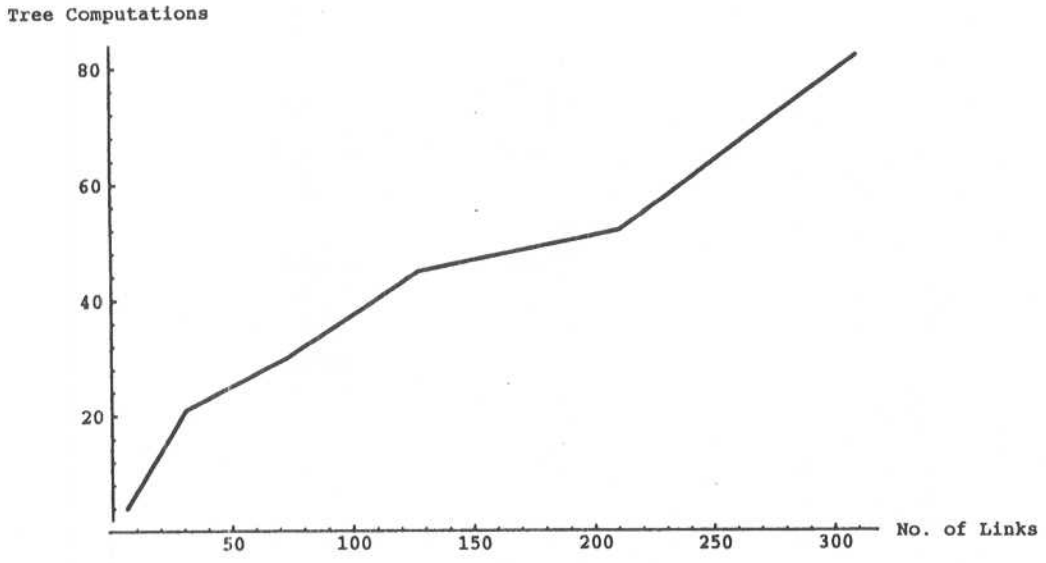

Fig. 5. Spanning tree computation as function of number of links.

- The performance of the algorithm can be improved by application of different methods for incremental computation of spanning trees when edges are removed. Work on this topic has been started very early [25] and gained focus recently again [26–29].
- The algorithm can easily be extended to work with directed graphs. Also, instead of the Dijkstra's algorithm for spanning tree computation, another one like Floyd could be chosen. The decision taken depends basically on topology scenario encountered. Different advantages and disadvantages of shortest spanning tree computation algorithms can be found in Ref. [30] and a more theoretical outlook in Ref. [31].
- An important issue influencing the quality of the algorithm is the representation of the graph in memory. Several papers treat this subject [32–35].
- Handling of path candidate lists and different strategies for fast detection whether a new route can be skipped as worse than any of already known ones can improve the behavior of the algorithm significantly.
- The question of possible parallelization of the algorithm is probably of great importance in case it should be used for precomputation in routers. The nature of the algorithm makes even a massive parallel approach possible and it becomes easier to parallelize with growth of the graphs which is also a positive property.
- A rather unexploited area is the question of how the QoS-vectors as specified by Refs. [13] and [36] can be mapped to the additive and restrictive metric terminology used here.

6. EFFICIENCY

To check the practical applicability of our approach, a first implementation has been analyzed. Because the case for $r = 2$ could not generate any reasonable load, the more interesting case for $r = 3$ has been implemented. As the language 'C' has been used and compiled on a IBM RISC System/6000 250.

We have implemented the optimizations described at the beginning of chapter 5.

The scenario assumes a worst case has been assumed, where all of the weight components of the links are independent and uniformly distributed between 0 and $2^{16} - 1$. Graphs between 10 and 50 vertices with a connectivity between 10 and 40% of a fully meshed network were generated. Different statistical measurements have been performed. The results are presented in Figs. 6–9.

- Figure 6 shows the increase in average amount of optimal paths per destination in the graph when the number of vertices or connectivity raises. For large networks, either an area approach like in OSPF [8] will

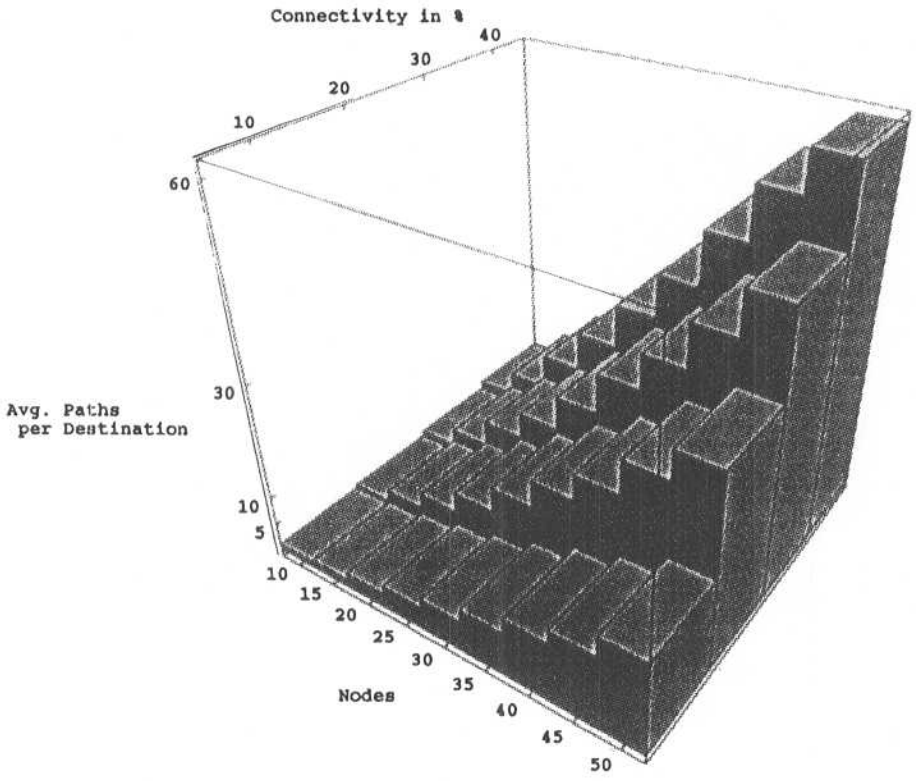

Fig. 6. Average number of extremal paths per destination.

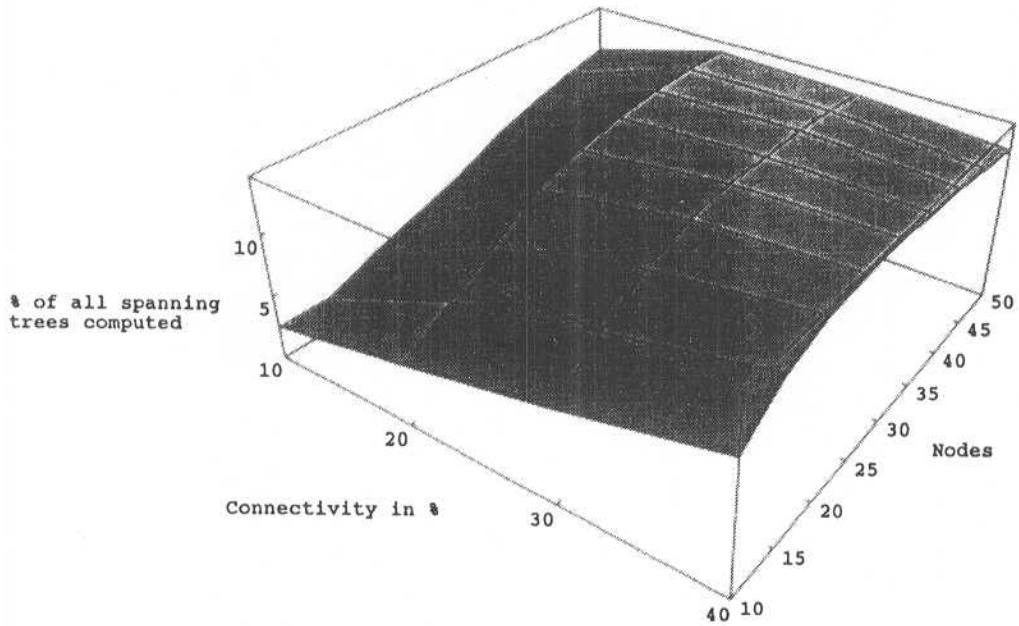

Fig. 7. Percentage computed spanning trees vs. spanning trees for all link combinations.

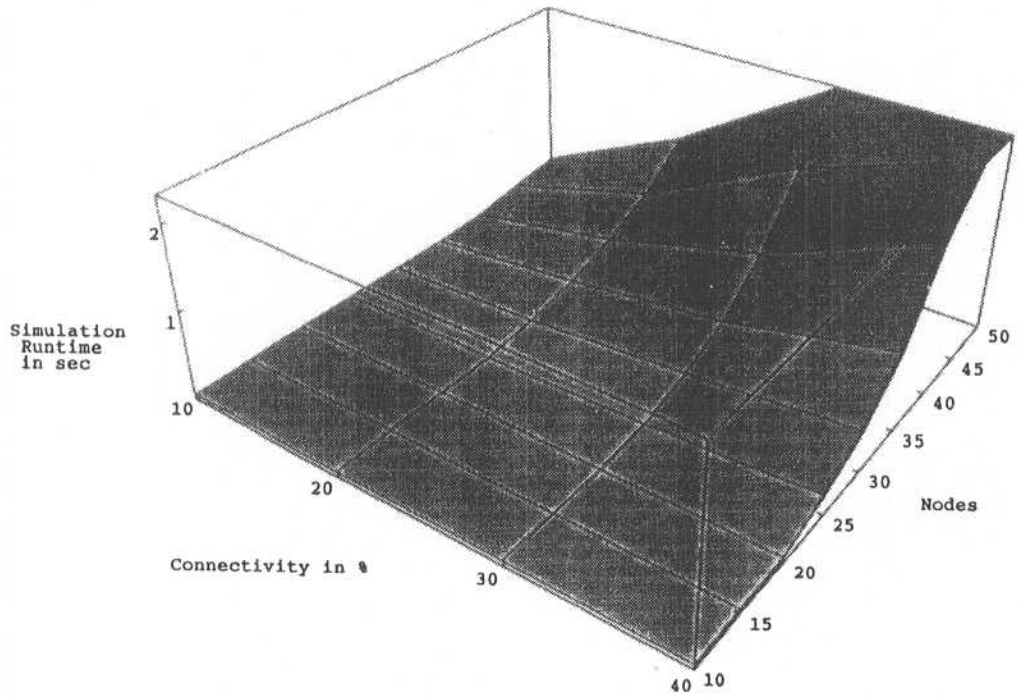

Fig. 8. Simulation runtime in seconds (including gathering of statistical data and graph generation).

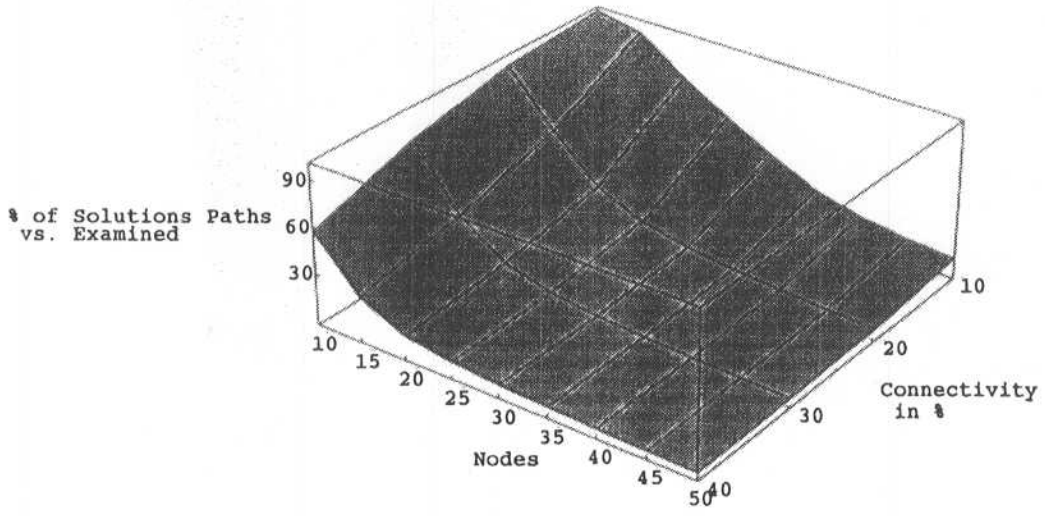

Fig. 9. Percentage of optimal path vs. paths examined by the algorithm.

have to be introduced or a strategy to only keep subsets of computed optimal paths.

- Figure 7 expresses the efficiency of the optimization applied by avoiding the computation of trees when the link removed from the graph is not a part of the spanning tree. Typically, only 10% to 15% of spanning trees are computed which would have to be computed without this improvement.
- Figure 8 shows the simulation runtimes in seconds for different graphs. Although times shown are acceptable, further work is needed to implement all the optimizations proposed to make the algorithm perform even better. In the version implemented today, e.g. graphs of 30 nodes with 30% connectivity, which is a reasonable scenario for a local area network, can already be fully treated in 0.1–0.2 seconds CPU time.
- Figure 9 shows how many of the paths examined by the algorithm actually become a part of the solution. It is not very surprising that for small graphs this percentage is very high and drops drastically later. This is rather a property of the number of elementary paths on the graph, which grows in a NP-complete manner, and not a statement about the quality of the algorithm itself.

7. CONCLUSION

We have presented a solution for combining two requirements that, at first glance, seem contradictory: compute routes in advance, and support connec-

tions of any bit rate requirement. This should be particularly attractive for networks with very large connection setup rates, but where connections request relatively small amounts of link capacity, thus resulting in relatively infrequent topology updates. We have shown that the algorithm we use is the combination of a shortest path computation and a "Min-Max" algorithm for the determination of extremal points. Beside that, a first working implementation for the case of $r = 3$ has been tested and performance data gathered proving that the algorithm behaves well for networks of moderate size.

Future research is now concentrating on further optimizations of the algorithm in order to support large networks, or complex link cost definitions that support sustainable cell rate allocation.

APPENDIX: A DETAILED ALGORITHM

This appendix presents pseudo-code of the algorithm given in this paper. For the sake of simplicity it omits all optimizations. As algorithm for shortest spanning tree computation, Dijkstra's has been chosen. The notation chosen is 'modula'-similar extended by some powerful list operations which make the code more understandable.

A.1. Definitions

- ADR(x) returns the address of x[4] (Note: used for pointers.)
- $\hat{x}$ returns the object x is pointing to
- uses $\exists$ operator to check for existence
- *Length*(q) returns length of the list q
- *Head*(q), *Tail*(q) return head or tail of a list, 0 if empty
- *Head* + (e, q), *Tail* + (e, q) add a element e to list q only if it is not yet in the list
- *Head* − (q), *Tail* − (q) remove head or tail of list and return removed value or 0 if list empty
- *Insert*(e, k, q) inserts element e into list q at position k
- *Remove*(k, q) removes the element in list q at position k
- Σ summs up all elements of the expression following
- *Max* gives the maximal element of the sequence following
- # gives the number of elements in a list
- $q[x]$ is the element at place x in a array or list or set q
- router 1 is the source
- R is number of routers
- { } denotates an empty set or list

A.2. Pseudo Code

```
ROUTER : ARRAY[1..R] OF
                    SET LINKS OF STRUCTURE LINK
                        BEGIN
                        INTEGER DESTINATION;
                        INTEGER ADDITIVE_COST;
                        INTEGER RESTRICTIVE_COST;
                        END; /* TOPOLOGY DATABASE */
SPF    : SET OF INTEGER; /* SET OF ROUTERS ON SPF TREE */
ROUTE  : ARRAY[1..R] OF
                    LIST HOPS OF STRUCTURE ELEMENT
                        BEGIN
                        INTEGER ROUTER;
                        INTEGER LINK_NDX;
                        END;
         /* SET OF ROUTER AND LINK INDICES ON THE ROUTE FROM SOURCE
            TO DESTINATION */
PATHS  : ARRAY[1..R] OF LIST OF
                    LIST HOPS OF STRUCTURE ELEMENT;
CAND   : LIST OF INTEGER;
         /* INDICES OF CANDIDATES TO TEST NEXT STEP SORTED ON DISTANCE */
VERTCND: LIST OF POINTER TO STRUCTURE LINK;
         /* INSTALLED TO LIST OF POINTERS TO ALL EDGES SORTED
            REVERSE ON RESTRICTIVE COMPONENT */

INTEGER FUNCTION ADDITIVE_LENGTH ( ROUT: LIST OF STRUCTURE ELEMENT );
/* COMPUTES LENGTH OF A ROUTE GIVEN AS LIST OF LINKS */
BEGIN
   RETURN
   ∑_{j:=1}^{j<#rout} ROUTER[ROUT.ELEMENT[J].ROUTER].
            LINK[ROUT.ELEMENT[J].LINK_NDX].ADDITIVE_COST;
END; /* OF FUNCTION */

INTEGER FUNCTION RESTRICTIVE_LENGTH ( ROUT: LIST OF STRUCTURE ELEMENT );
/* COMPUTES LENGTH OF A ROUTE GIVEN AS LIST OF LINKS */
BEGIN
   RETURN
   Max_{j:=1}^{j<#rout} ROUTER[ROUT.ELEMENT[J].ROUTER].
            LINK[ROUT.ELEMENT[J].LINK_NDX].RESTRICTIVE_COST;
END; /* OF FUNCTION */

STRUCTURE ELEMENT NEW_ELEMENT INTEGER C;
INTEGER I,J,K,DST,DIST,RESTRICTION;
POINTER TO STRUCTURE LINK LINKPTR;
BOOLEAN FOUND;

/* TOPOLOGY DATABASE ALREADY INSTALLED */
/* VERTCND LIST ALREADY INSTALLED WITH LIST OF LINKS SORTED REVERSE
   ON RESTRICTIVE COMPONENT */
RESTRICTION := 0;
LINKPTR     := 0;

WHILE Length(VERTCND)>0 DO
    ROUTE[1...R] := { };
    LINKPTR      := Head-(VERTCND);
    RESTRICTION  := LINKPTR^.RESTRICTIVE_COST;
    CAND         := { };
    SPF          := { };
    C            := 1; /* INDEX OF THE COMPUTED ROUTER, 1 IS SOURCE */
```

```
    WHILE c≠0 DO
        FOR I:=1 TO #ROUTER[C].LINKS /* NUMBER OF THIS ROUTER'S LINKS */
            DST := ROUTER[ROUTER[C].LINKS[I].DESTINATION];
            Head+(C, SPF);
            IF ∃ X | ROUTER[DST].LINKS[X].DESTINATION = C AND
               ROUTER[C].LINKS[I].RESTRICTIVE_COST >= RESTRICTION AND
               DST ∉ SPF THEN
               /* CHECKS WHETHER A BACK LINK EXISTS AND DESTINATION NOT
                  ALREADY COMPUTED ON TREE ?? */
               DIST := ROUTER[C].LINKS[I].ADDITIVE_COST + ADDITIVE_LENGTH(ROUTE[C]);
               IF (DIST < ADDITIVE_LENGTH(ROUTE[DST]) OR
                   ADDITIVE_LENGTH(ROUTE[DST])=0) THEN
                   ROUTE[DST]:= ROUTE[C];
                   NEW_ELEMENT.ROUTER := C;
                   NEW_ELEMENT.LINK_NDX := I;
                   Tail+(NEW_ELEMENT, ROUTE[DST]);
                   K:=1;
               WHILE K < #CAND AND ADDITIVE_LENGTH(ROUTE[K]) < DIST DO
                   K:= K+1;
                   END;
               Insert(DST, K, CAND ); /* INSERT SORTED ON CANDIDATE LIST */
               END;
           END;
        END;
    C=Head-( CAND );
    END;
/* ALL ROUTES COMPUTED, NOW INSERT INTO LISTS */
FORI:=1 TO R DO
    FOUND:=FALSE;
    J:=1;
    WHILE NOT FOUND AND J<=Length(PATHS[I]) DO
        WHILE RESTRICTIVE_LENGTH(PATHS[I][J])>RESTRICTIVE_LENGTH(ROUTE[I]) AND
              ADDITIVE_LENGTH(PATHS[I][J])   >ADDITIVE_LENGTH(ROUTE[I])
            Remove(J,PATHS[I]);
            END;
        FOUND:=RESTRICTIVE_LENGTH(PATHS[I][J])<RESTRICTIVE_LENGTH(ROUTE[I]) AND
              ADDITIVE_LENGTH(PATHS[I][J])   <ADDITIVE_LENGTH(ROUTE[I])
        J:=J+1;
        END;
    IF NOT FOUND THEN
        Tail+(ROUTE[I],PATHS[I]);
        END;
    END;
END;
```

REFERENCES

1. C. Topolcic, Experimental Internet Stream Protocol, Version 2, (ST-II), *IETF*, 1990.
2. L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zapalla, RSVP: A New Resource ReSerVation Protocol, *IEEE Network*, September 1993.
3. J. Y. Le Boudec and T. Przygienda, Routing Metric for Connections with Reserved Bandwidth, *EFOC-N*, 1994.
4. R. J. Cypser, *Communications Architecture for Distributed Systems*, ISBN 0-201-14458-1, 1971.
5. ISO/IEC TR 9575, Information Technology—Telecommunications and Information Exchange Between Systems—OSI Routeing Framework, *ISO*, 1989.

6. E. Szybicki, The Introduction of an Advanced Routing System in Local Digital Networks and Its Impact on the Networks Economy, Reliability and Grade of Service, Bell Canada, 1981.
7. A. Modarressi and R. Skoog, Signalling System 7: A Tutorial, *IEEE Communications Magazine*, July 1990.
8. J. Moy, RFC 1247, OSPF Version 2, IETF, July 1991.
9. L. Huynh and D. Bryant, APPN Architecture, IBM.
10. ISO 10589, Information Technology—Telecommunications and Information Exchange Between Systems—Intermediate System to Intermediate System Routeing Exchange Protocol for Use in Conjunction with the Protocol for Providing the Connectionless-mMde Network Service, *ISO*, 1990.
11. R. Ullmann, RFC 1476, RAP: Internet Route Access Protocol, *IETF*, June 1993.
12. B. Carré, *Graphs and Networks*, ISBN 0-19-859622-7.
13. I. Cidon, I. Gopal, and R. Guérin, Bandwidth Management and Congestion Control in plaNET, *IEEE Communication Magazine*, April 1988.
14. R. Guérin and L. Gün, A Unified Approach to Bandwidth Allocation and Access Control in Fast Packet-Switched Networks, *Proc. INFOCOM '92*, Florence, Italy, May 1992.
15. T. Ballardie, P. Francis, and J. Crowcroft, *Core Based Trees (CBT)—An Architecture for Scalable Inter-Domain Multicast Routing*, Sigcomm 1993.
16. M. Garey and D. Johnson, *Computers and Intractability*, ISBN 0-7167-1045-5, 1979.
17. D. Paik and S. Sahni, NP-Hard Network Upgrading Problems+, University of Florida Technical Report 91-29, 1991.
18. J. M. Jaffe, Algorithms for Finding Paths with Multiple Constraints, *Networks*, Vol. 14, 1984.
19. G. Y. Handler and I. Zang, A Dual Algorithm for the Constrained Shortest Path Problem, *Networks*, Vol. 10, 1980.
20. R. Kung and N. Shachum, An Algorithm for the Shortest Path under Multiple Constraints, SRI project 5732, November 1983.
21. E. L. Lawler, Optimal Cycles in Doubly Weighted Directed Linear Graphs, *Int. Symp. on Theory of Graphs*, Dunod, Paris, 1966.
22. H. C. Joksch, The Shortest Route Problem with Constraints, *J. of Math. Anal. and Appl.*, 1966.
23. R. C. Berry, A Constrained Shortest Path Algorithm, *39th National ORSA Meeting*, Dallas, Texas, 1971.
24. N. Christofides, *Graph Theory, An Algorithmic Approach*, ISBN 012-174350-0, 1975.
25. J. M. McQuillan, I. Richer, and E. C. Rosen, ARPANET Routing Algorithm Improvements, BBN Technical Report 3803, April 1978.
26. S. Even and Y. Shiloach, An On-Line Edge Deletion Problem, *J. Assoc. Comput. Mach.*, 1981.
27. D. Eppstein, Offline Algorithms for Dynamic Minimum Spanning Tree Problems, 2nd Worksh. Algorithms and Data Structures, Springer Verlag *LNCS* Vol. 519, 1991.
28. D. G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, M. Yung, and Eppstein, Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph, *Algorithms* Vol. 13, 1992.
29. F. Chin and D. Houck, Algorithms for Updating Minimum Spanning Trees, *J. Comput. System. Sci.*, 1978.
30. S. E. Dreyfus, An Appraisal of Some Shortest-Path Algorithms, University of California, Berkeley, 1968.
31. D. Cheriton and R. Tarjan, Finding Minimum Spanning Trees, *SIAM J. Comput.*, May 1976.
32. D. D. Sleator and R. E. Tarjan, A Data Structure For Dynamic Trees, *J. Comput. System Sci.* Vol. 26, 1983.
33. G. N. Frederickson, Data Structures for On-Line Updating of Minimum Spanning Trees, *Pro. 15th ACM Symposium on Theory of Computing*, Boston, April 1983.

34. G. N. Frederickson, Data Structures for On-Line Updating of Minimum Spanning Trees with Applications, *SIAM J. Comput.*, 1985.
35. M. L. Fredman, Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, *Journal of the ACM*, Vol. 34, No. 3, July 1987.
36. ATM Uni Specification 2.4, *ATM Forum*, August 1993.

Prof. Le Boudec graduated from Ecole Normale Superieure de Saint-Cloud, Paris, France, in 1980, and obtained a Ph.D. in mathematics (queueing Theory) from University of Rennes in 1984. From 1984 to 1987, he was assistant professor at INSA/IRISA (France), where he developed modeling techniques and tools for multiprocessor, multiple bus computer architectures. From 1987 to 1988, he worked for Bell-Northern Research, Ottawa, Canada, where he evaluated the performance of ISDN switching equipment, and participated in the development of automatic dimensioning tools. From 1988 to 1994, he was with the IBM Research Lab. in Rueschlikon, Switzerland, where he developed various research projects on ATM. In particular, he was manager of a research group developing control architecture and software for ATM local area networks. In 1994 he was elected Professor at the EPFL, where he now heads the Institute for Communications Networks. He is active in the area of control, performance and architecture for multimedia and integrated services networks.

Antonio B. Przygienda was born in Kalisz, Poland, on March 3, 1967. He received the M.S. and Ph.D. degrees in computer science from the Swiss Federal Institute of Technology in Zurich in 1992 and 1995, respectively. In 1992 he joined IBM Research Laboratory in Rueschlikon, Switzerland. His research interests encompass dynamic routing for broadband networks and applications of graph theory, operating system design and programming paradigms to computer communication systems.