New Internet Routing and Forwarding for the Knowledge Plane

Bradley R. Smith and J.J. Garcia-Luna-Aceves
brad@soe.ucsc.edu, jj@soe.ucsc.edu
Computer Engineering Department, Jack Baskin School of Engineering
University of California, Santa Cruz, CA 95064

***Abstract*— A key reason for the success of the Internet architecture is its routing model based on fully distributed routing computation and hop-by-hop forwarding. However, limitations of the architecture are being encountered as it is applied to ever more demanding applications. Real time applications have different performance requirements, and production network management and security require policy-based control of forwarding topologies used for different classes of traffic. To satisfy these new requirements a new construct called the knowledge plane has been proposed, which "gathers and provides an integrated view and consistent set of control signals" for controlling the use of network resources in an internet. To satisfy these requirements, the knowledge plane must control the routing and admission control functions to enforce network resource usage policies while maintaining the successful distributed, hop-by-hop routing model. This requires the computation and implementation of multiple routes per destination in the context of administrative policies and multi-dimensional link metrics using a fully distributed routing computation and hop-by-hop forwarding mechanisms. This paper presents an enhanced routing architecture and a family of new algorithms that provide the first comprehensive solution to these requirements; furthermore, it is shown that this solution involves minimal additional overhead compared with the current single-class routing model.**

I. INTRODUCTION

The architecture of today's Internet is based on the *catenet model for internetworking* [1], [2], [3]. The two basic components of a catenet are networks and gateways, where a catenet is formed by the interconnecting of networks with gateways. A primary goal of the catenet model, and therefore the Internet architecture, was to encourage the development and integration of new networking technologies into the developing catenets. To achieve this goal, only minimal assumptions were made of networks and the routing computation by the catenet model. Networks were assumed to support the attachment of a number of computers, transport datagrams, allow switched access so that attached computers could "quickly" send datagrams to different destinations, and provide best-effort delivery, where the definition of best-effort allowed datagrams to be dropped, or delivered out of order. The routing computation was assumed to support a single forwarding class chosen by the optimization of a single, typically delay-related metric.

This best-effort model of communication has proven surprisingly powerful. Indeed, much of the success of the Internet architecture can be attributed to this inspired design decision. However, largely as a product of its own success, limitations of the Internet architecture are being encountered as it is applied to ever more demanding applications [4]. A primary limitation of this communications model is that it only supports one path to any destination in an internet. With this limitation, the requirements of real-time applications cannot be supported. Examples of such applications include on-demand streaming, audio and video conferencing, visualization, and virtual reality. Each of these applications have different performance requirements of the network: on-demand streaming requires low delay jitter while conferencing requires both low delay and low delay jitter; audio has minimal bandwidth but strict loss rate requirements while video has high bandwidth but more tolerant loss requirements; etc. Effective support of a range of such applications requires the ability to compute (in the routing plane) and implement (in the forwarding plane) a set of paths to each destination providing the range of performance characteristics supported by an internet from which applications can pick.

Similarly, the network management and traffic engineering requirements of production internets cannot be supported by this single-class communications model. The goal of network management and traffic engineering is to control the use of network resources according to resource management and security policies. This requires that routes be computed in the context of administrative

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant N66001-00-8942.

constraints on the characteristics of traffic allowed to traverse different portions of an internet, with the result that, in general, multiple paths will be computed to each destination that support different classes of traffic. Again, effective support of such functionality requires the ability to compute and implement a set of paths to each destination providing the range of traffic classes supported by an internet from which the traffic classification and forwarding functions can choose. One particular manifestation of the inability of the current architecture to support such policies is the limited support for security and attack response available in the network today. Due to the limitation of one path per destination, network-based security services are currently limited to monolithic firewalls between a set of protected information resources and an internet, and attack response to reconfiguration of the filters and proxies on these firewalls. Specifically, it is not possible to define custom forwarding topologies for different administrative classes of traffic in an internet, nor to modify these topologies in response to network-based attacks or faults.

A secondary limitation of the single-class communication model is that, since routing and forwarding in the context of policies and diverse application requirements is not supported in an internet, existing policies must be implemented external to the internet configuration. As a result, policies must be relatively static, they cannot be defined in terms of dynamic topology or network state information available to the routing and forwarding processes, and the mapping from policy to network configuration is, of necessity, very device specific, and therefore error-prone and expertise-intensive.

To address the above limitations of today's Internet architecture, a new construct called the *knowledge plane* [5] (KP) has been proposed. The knowledge plane is "a distributed and decentralized construct within the network that gathers and aggregates information about network operation, and provides an integrated view and a consistent set of control signals." Its primary functions are to provide automatic fault and attack detection and response, to provide a high-level, "do what I mean" interface to network configuration, to enforce administrative policies regarding the use of network resources, and to allow applications to exploit special circumstances and network technologies to satisfy the varied quality-of-service requirements of applications in an internet. The KP must satisfy a number of primary requirements:

- It must work at a high level of abstraction; e.g., "the internet must provide at least two (edge disjoint) paths with delay $\leq X$ and available bandwidth $\geq Y$ between all Gold customers and the video server farm."
- It must be able to reason about and react to what is occurring in its network environment.
- It must operate "out-of-band" in the sense that it must not increase the complexity of the forwarding plane.

To implement these functions and satisfy these requirements the KP must control the routing and admission control functions to implement network resource usage policies, manage routes providing the range of performance characteristics available in an internet, and monitor the routing and forwarding activity to ensure policy compliance, modifying the routing and admission control functions on the detection of policy violations or changes to restore policy compliance. This must be accomplished without violating the basic tenets of the Internet architecture that have made it so successful, specifically: distributed routing, hop-by-hop forwarding, and the end-to-end principle [6]. To achieve these goals the implementation of the KP requires a new approach to the way in which routes are computed and packets are forwarded in the Internet. Specifically, it must support the computation and implementation of multiple routes per destination, in the context of administrative policies and multi-dimensional link weights, using a fully distributed routing computation and hop-by-hop forwarding mechanisms.

This paper presents a new approach to Internet routing that enables the implementation of the KP. The contributions of this paper are:

- The first unifying approach to the support of routing with traffic engineering and quality-of-service constraints.
- A family of efficient algorithms for computing routes in the context of traffic-engineering constraints, multi-dimensional link weights (for the satisfaction of quality-of-service requirements), and a combination of the two, which achieve new levels of computational efficiency.
- A forwarding architecture that efficiently supports hop-by-hop forwarding in the context of multiple paths to each destination.

Sections II, III, and IV present the model and algorithms for computing optimal routes in the context of policy-based link metrics. An outline of the correctness of the algorithms is presented, and they are shown to be highly efficient.

In this paper, the inclusion of multiple metrics in a routing computation is called *policy-based* routing [9], [11]. Policy-based routing supports *traffic engineering* by the computation of routes in the context of administrative constraints on the type of traffic allowed over links in

an internet. Analogously, policy-based routing supports *quality-of-service* (QoS) by the computation of routes in the context of multi-dimensional weights (i.e. link weights composed of multiple metrics) assigned to the links in an internet. The metrics used in routing computations are assigned to individual links in the network. For a given routing application, a set of link metrics is identified for use in computing the path metrics used in the routing decision. Link metrics can be assigned to one of two classes based on how they are combined into path metrics. *Concave (or minmax) metrics* are link metrics for which the minimum (or maximum) value (called the bottleneck value) of a set of link metrics defines the path metric of a path composed of the given set of links. Examples of concave metrics include residual bandwidth, residual buffer space, and administrative traffic constraints. *Additive metrics* are link metrics for which the sum (or product, which can be converted to a sum of logarithms) of a set of link metrics defines the path metric of the path composed of the given set of links. Examples of additive metrics include delay, delay jitter, cost, and reliability.

The foundational work on the problem of computing routes in the context of more than one additive metric was done by Jaffe [8], who defined the multiply-constrained path problem (MCP) as the computation of routes in the context of two additive metrics. He presented an enhanced distributed Bellman-Ford algorithm that solved this problem with time complexity of $O(n^4 b \log(nb))$, where b is the largest possible metric value. Since Jaffe, a number of solutions have been proposed for computing exact routes in the context of multiple metrics for special situations. Wang and Crowcroft [11] were the first to present the solution to computing routes in the context of a concave and an additive metric discussed above. Ma and Steenkist [12] presented a modified Bellman-Ford algorithm that computes paths satisfying delay, delay-jitter, and buffer space constraints in the context of weighted-fair-queuing scheduling algorithms in polynomial time. Cavendish and Gerla [13] presented a modified Bellman-Ford algorithm with complexity of $O(n^3)$ which computes multi-constrained paths if all metrics of paths in an internet are either non-decreasing or non-increasing as a function of the hop count. Recent work by Siachalou and Georgiadis [14] on MCP has resulted in an algorithm with complexity $O(nW \log(n))$ (in terms defined in Section III). This algorithm is similar to the QoS algorithm presented in Section IV in that it is an enhanced version of the Dijkstra algorithm based on invariants similar to those underlying the algorithms presented in Sections III and IV. However, a detailed comparison of the Siachalou algorithms and those presented here is beyond the scope of this paper. Several other algorithms have been proposed for computing approximate solutions to the QoS routing problem. Both Jaffe [8] and Chen and Nahrstedt [9] propose algorithms which map a subset of the metrics comprising a link weight to a reduced range, and show that using such solutions the cost of a policy-based path computation can be controlled at the expense of the accuracy of the selected routes. Similarly, a number of researchers [8], [15] have presented algorithms which compute routes based on a function of the multiple metrics comprising a link weight. These approximation solutions do not work with administrative traffic constraints. In summary, the drawbacks of the current policy-based routing solutions are that they have poor average case performance, they implement inflexible routing models, and those based on algorithms that compute approximate solutions do not work with the administrative constraints used for traffic engineering.

The routing model used by the Internet architecture is a *table-driven, hop-by-hop* routing model. In this model, routers learn about the state of connectivity in an internet by exchanging messages with each other, and run local routing computations whose output is a forwarding table. This forwarding table is used by the router's forwarding process to make per-packet forwarding decisions. In contrast, many recent policy-based routing proposals have used an on-demand, source-driven, *virtual-circuit-based* routing model where routes are computed on receipt of the first packet of a flow, and forwarding is source driven through the use of either path setup or source-routing techniques. There are several weaknesses to this model compared to the table-driven, incremental model. First, changes in network state must be propagated to the source, and topology change requests propagated back into the network to adapt to changes in an internet. Second, the model implements a centralized model of routing control where forwarding state for all sources at a given point in the network is maintained by the router acting for those sources. Lastly, information about a link must be propagated to all routers in an internet. As a result, in these proposals, routing is implemented as a centralized routing computation, with remote control of forwarding state, which is less efficient, responsive, and robust than table-driven, hop-by-hop solutions. Section V presents a new forwarding architecture that efficiently supports hop-by-hop forwarding in the context of policy-based routing with multiple paths to each destination. It should be noted that, while the new algorithms for policy-based routing presented in this paper are applicable to any routing model, a primary goal of their design has been that they be compatible with a table-driven,

hop-by-hop model.

II. A Model for Policy-Based Routing

A network is modeled as a weighted undirected graph $G = (N, E)$, where N and E are the node and edge sets, respectively. By convention, the size of these sets are given by $n = \mid N \mid$ and $m = \mid E \mid$. Elements of E are unordered pairs of distinct nodes in N. $A(i)$ is the set of edges adjacent to i in the graph. Each link $(i, j) \in E$ is assigned a weight, denoted by ω_{ij}. A *path* is a sequence of nodes $< x_1, x_2, \ldots, x_d >$ such that $(x_i, x_{i+1}) \in E$ for every $i = 1, 2, \ldots, d-1$, and all nodes in the path are distinct. The weight of a path is given by $\omega_p = \sum_{i=1}^{d-1} \omega_{x_i x_{i+1}}$. The nature of these weights, and the functions used to combine these link weights into path weights are specified for each algorithm.

In the following, we propose a *declarative* traffic engineering model where network links are labeled with statements declaring *what* the desired routing policies are in the form of constraints of the traffic allowed on each link. These constraints take the form of *link predicates* in a boolean *traffic algebra* which describe the traffic allowed on a link. New, efficient policy-based routing algorithms then compute a minimal set of routes, composed of a *path predicate* and a next hop, for each destination in an internet. These algorithms, in effect, *discover* the optimal set of forwarding classes needed at a given source in the internet to implement the desired policies. These path predicates are then installed in the appropriate traffic classifiers.

The traffic algebra is a boolean algebra used to define traffic classes in a flexible and efficient way. Specifically, it is composed of the standard boolean operations on the set $\{0, 1\}$, where p primitive propositions (variables) are true/false statements describing characteristics of network traffic. The syntax for expressions in the algebra is specified by the BNF grammar:

$$\varphi \ ::= \ 0 \mid 1 \mid v_1 \ldots v_p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid SAT(\varphi)$$

The set of primitive propositions, indicated by v_i in the grammar, can be defined in terms of any globally significant attributes of the ingress router's state that can be expressed as a true/false statement. Link predicates identify the traffic classes allowed to traverse the link, and are denoted by ε_{ij} in the algorithms. Path predicates, denoted by ε_p in the algorithms, and defined as $\varepsilon_p = \varepsilon_{x_1x_2} \wedge \varepsilon_{x_2x_3} \wedge \ldots \wedge \varepsilon_{x_{d-1}x_d}$, specify the set of traffic classes allowed to traverse the path. There is a maximum of 2^p unique sets of traffic classes.

The $SAT(\varphi)$ primitive of the traffic algebra is the SATISFIABILITY problem of traditional Boolean algebra. Satisfiability must be tested in two situations by the algorithms presented below that implement traffic-engineering computations. First, an extension to a known route should only be considered if classes of traffic exist that are authorized to use both the path represented by the known route and the link used to extend the path (at line 13 in Figure 5). This is true *iff* the conjunction of these expressions is satisfiable (i.e. $SAT(\varepsilon_i \wedge \varepsilon_{ij})$). Second, given that classes of traffic exist that are authorized to use a path represented by a new route, the algorithms must determine whether all traffic supported by that route has also been satisfied by other, known shorter routes. This is true *iff* the new route's traffic expression implies the disjunction of the traffic expressions for all known better routes (i.e. $(\varepsilon_i \rightarrow \varepsilon_{i_1}, \ \varepsilon_{i_2}, \ ..)$ is *valid*, which is denoted by $(\varepsilon_i \rightarrow \mathcal{E}_i)$ in the algorithms). Determining if an expression is valid is equivalent to determining if the negation of the expression is unsatisfiable. Therefore the expressions at lines 10 and 13 of Figure 5, of the form $\varepsilon_1 \rightarrow \varepsilon_2$ are equivalent to $\neg SAT(\neg(\varepsilon_1 \rightarrow \varepsilon_2))$ (or $\neg SAT(\varepsilon_1 \wedge \neg\varepsilon_2)$).

The satisfiability decision performed by $SAT(\varepsilon)$ is the prototypical NP-complete problem [7]. As is typical with NP-complete problems, it has many restricted versions that are computable in polynomial time. An analysis of strategies for defining computationally tractable traffic algebras is beyond the scope of this paper, however we have implemented an efficient, restricted solution to the SAT problem by implementing the traffic algebra as a set algebra with the set operations of intersection, union, and complement on the set of all possible forwarding classes.

The routing algorithms presented here are based on an enhanced version of the path algebra defined by Sobrinho [16], which supports the computation of a set of routes for a given destination containing the "best" set of routes for each destination. Formally, the path algebra $P = <\mathcal{W}, \oplus, \preceq, \sqsubseteq, \overline{0}, \overline{\infty}>$ is defined as a set of weights $\mathcal{W}$, with a binary operator $\oplus$, and two order relations, $\preceq$ and $\sqsubseteq$, defined on $\mathcal{W}$. There are two distinguished weights in $\mathcal{W}$, $\overline{0}$ and $\overline{\infty}$, representing the least and absorptive elements of $\mathcal{W}$, respectively. $\oplus$ is the original path composition operator, and $\preceq$ is the original total ordering from [16]. $\oplus$ is used to compute path weights from link weights. $\preceq$ is used by the routing algorithm to build the forwarding set, starting with the minimal element, and by the forwarding process to select the minimal element of the forwarding set whose parameters satisfy a given QoS request.

A new relation on routes, $\sqsubseteq$, is added to the algebra and used to define classes of comparable routes and

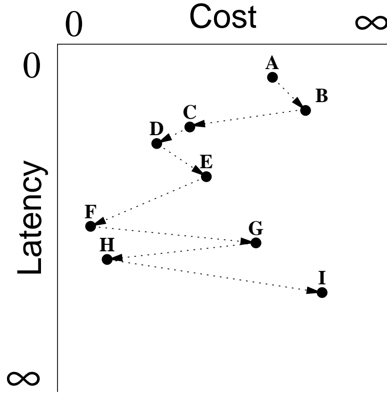

Fig. 1. $\preceq$ relation

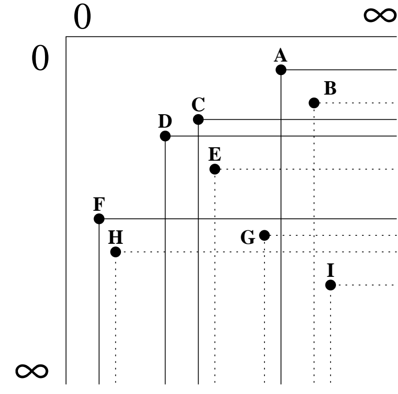

Fig. 2. $\sqsubseteq$ relation

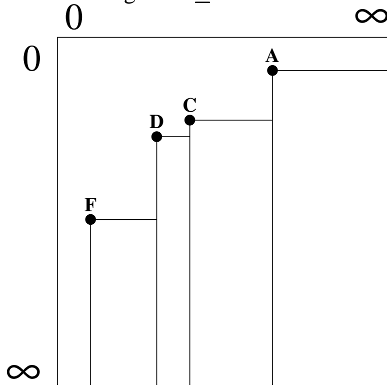

Fig. 3. Forwarding table

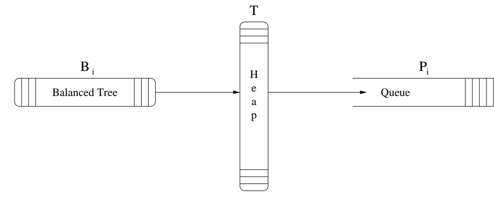

Fig. 4. Model of Data structures for Basic Algorithms

select maximal elements of these classes for inclusion in the set of forwarding entries for a given destination. $\sqsubseteq$ is a partial ordering (reflexive, anti-symmetric, and transitive) with the following, additional property:

Property 1: $(\omega_x \sqsubseteq \omega_y) \Rightarrow (\omega_x \succeq \omega_y)$.

A route r_m is a *maximal element* of a set R of routes in a graph if the only element $r \in R$ where $r_m \sqsubseteq r$ is r_m itself. A set R_m of routes is a *maximal subset* of R if, for all $r \in R$ either $r \notin R_m$, or $r \in R_m$ and for all $s \in R - \{r\}$, $\neg(r \sqsubseteq s)$. The maximum size of a maximal subset of routes is the smallest range of the components of the weights (for the two component weights considered here). An example path algebra based on weights composed of delay and cost is as follows:

$$
\begin{aligned}
\omega_i &\equiv (d_i, c_i) \\
\overline{0} &\equiv (0, 0) \\
\overline{\infty} &\equiv (\infty, \infty) \\
\omega_i \oplus \omega_j &\equiv (d_i + d_j, c_i + c_j) \\
\omega_i \preceq \omega_j &\equiv (d_i < d_j) \vee ((d_i = d_j) \wedge (c_i \leq c_j)) \\
\omega_i \sqsubseteq \omega_j &\equiv (d_j \leq d_i) \wedge (c_j \leq c_i)
\end{aligned}
$$

Figure 1 is a graphical depiction of the $\preceq$ relation on a set of weights for routes (labeled A through I) to a given destination in an internet where the arrows are interpreted as $<tail\ of\ arrow> \preceq <head\ of\ arrow>$. Figure 2 illustrates the $\sqsubseteq$ relation where each route is represented as a subset of the plane with upper left-hand corner at the coordinates for the route. The intuition communicated here is that a route satisfies any constraint pair contained in its sub-region of the plane. Building on this intuition, the $\sqsubseteq$ relation defines an ordering on routes in terms of the containment (subset) of one route's region within another's, i.e. if $\omega_i \sqsubseteq \omega_j$, then the set of constraint pairs that route i can satisfy is a subset of those satisfiable by route j. The maximal subset of a set of such routes (the set of routes shown with solid lines in Figure 2) contain routes that satisfy all constraint pairs satisfiable by any route in the internet, and is the goal of the routing computation. Clearly, any pair of routes in the maximal subset of routes overlap, and can satisfy some set of constraint pairs. The $\preceq$ relation is used to select one of the set of satisfying routes for a given constraint. As defined in this example, the $\preceq$ relation has the affect of truncating the extent of a route's region at the first overlapping route to the right in the maximal subset of routes (as shown in Figure 3). As a result, forwarding table lookups in this example involve choosing the lowest

P	$\equiv$	Queue of permanent routes to all nodes.
P_n	$\equiv$	Queue of permanent routes to node n.
T	$\equiv$	Heap of temporary routes.
T_n	$\equiv$	Entry in T for node n.
B_n	$\equiv$	Balanced tree of routes for node n.
$\mathcal{E}_n$	$\equiv$	Summary of traffic expression for all routes in P_n.

TABLE I
NOTATION.

delay route with acceptable cost.

III. Basic Policy-Based Routing Algorithms

The notation used in the algorithms presented in this paper is summarized in Table I. In addition, the maximum number of unique truth assignments is denoted by $A = 2^p$, the maximum number of unique weights by $W = \min(range\ of\ weight\ components)$, and the maximum number of adjacent neighbors by $a_{max} = \max\{| A(i) | \mid i \in N\}$. Table II defines the primitive operations for queues, heaps, and balanced trees used in the algorithms, and gives their time complexity used in the complexity analysis of the algorithms.

The algorithms presented in this section are based on the data structure model shown in Figure 4. In this structure, a balanced tree (B_i) is maintained for each node in the graph to hold newly discovered, temporary labeled routes for that node. The heap T contains the lightest weight entry from each non-empty B_i (for a maximum of n entries). Lastly, a queue, P_i, is maintained for each node which contains the set of permanently labeled routes discovered by the algorithm, in the order in which they are discovered (which will be in increasing weight). The general flow of these algorithms will be to take the minimum entry from the heap T, compare it with existing routes in the appropriate P_i, if it is incomparable with existing routes in P_i it is pushed onto P_i, and add "relaxed" routes for its neighbors to the appropriate B_x's.

The correctness of these algorithms is based on the maintenance of the following three invariants: for all routes $I \in P$ and $J \in B_*$, $I \preceq J$; all routes to a given destination i in P are incomparable for some set of satisfying truth assignments; and the maximal subset of routes to a given destination j in $P_j \cup B_j$ represents the maximal subset of all paths to j using nodes with routes in P. Furthermore, these invariants are maintained by the following two constraints on actions performed in each iteration of these algorithms: (1) only known-non-maximal routes are deleted or discarded, and (2) only

Notation	Description
Queue	
$Push(r, Q)$	Insert record r at tail of queue Q ($O(1)$)
$Head(Q)$	Return record at head of queue Q ($O(1)$)
$Pop(Q)$	Delete record at head of queue Q ($O(1)$)
$PopTail(Q)$	Delete record at tail of queue Q ($O(1)$)
d-Heap	
$Insert(r, H)$	Insert record r in heap H ($O(\log_d(n))$)
$IncreaseKey(r, r_h)$	Replace record r_h in heap with record r having greater key value ($O(d\log_d(n))$)
$DecreaseKey(r, r_h)$	Replace record r_h in heap with record r having lesser key value ($O(\log_d(n))$)
$Min(H)$	Return record in heap H with smallest key value ($O(1)$)
$DeleteMin(H)$	Delete record in heap H with smallest key value ($O(d\log_d(n))$)
$Delete(r_h)$	Delete record r_h from heap ($O(d\log_d(n))$)
Balanced Tree	
$Insert(r, B)$	Insert record r in tree B ($O(\log(n))$)
$Min(B)$	Return record in tree B with smallest key value ($O(\log(n))$)
$DeleteMin(B)$	Delete record in tree B with smallest key value ($O(\log(n))$)

TABLE II
OPERATIONS ON DATA STRUCTURES [17].

the smallest known-maximal route to a destination i is moved to P_i.

A. *Table-Driven Traffic Engineering*

Figure 5 presents an enhanced version of the Dijkstra routing algorithm that precomputes all optimal routes to all destinations in an internet in the presence of traffic constraints on the links in the network. In effect, this algorithm computes routes in the *virtual graph* induced by the link predicates existing in the internet. This virtual graph is composed of all nodes reachable by some path with a satisfiable path predicate, and all links composing these paths. The virtual graph is "discovered" as needed by the algorithm as the computation progresses.

Similar to Dijkstra, TD-TE-Dijkstra works by maintaining a temporarily labeled set of routes, T, and a queue of permanently labeled set of routes per destination, P_d. Each route is specified by a four-tuple $< d, p_d, \omega_d, \varepsilon_d >$. For routes in P_x, ω_x is the final weight assignment specifying the shortest distance to x for traffic satisfying the path predicate ε_x. For routes in T, ω_x is the current best estimate of this distance based on routes currently contained in P. p_x is the predecessor to x for the route. TD-TE-Dijkstra proceeds using the typical Dijkstra iteration over the nth closest node with

```
algorithm TD-TE-Dijkstra()
     begin
1    Push(<s, s, 0, 1>, P_s);
2    for each {(s, j) ∈ A(s)}
3       Insert(<j, s, ω_sj, ε_sj>, T);
4    while (|T| = 0)
        begin
5       <i, p_i, ω_i, ε_i> ← Min(T);
6       DeleteMin(B_i);
7       if (|B_i| = 0)
8          then DeleteMin(T)
9          else IncreaseKey(Min(B_i), T_i);
10      if (¬(ε_i → 𝓔_i))
           then begin
11            Push(<i, p_i, ω_i, ε_i>, P_i);
12            𝓔_i ← 𝓔_i ∨ ε_i;
13            for each {(i, j) ∈ A(i) |
                    SAT(ε_i ∧ ε_ij) ∧ ¬((ε_i ∧ ε_ij) → 𝓔_j)}
                 begin
14               ω_j ← ω_i + ω_ij; ε_j ← ε_i ∧ ε_ij;
15               if (T_j = ∅)
16                  then Insert(<j, i, ω_j, ε_j>, T)
17               else if (ω_j < T_j.ω)
18                  then DecreaseKey(<j, i, ω_j, ε_j>, T);
19               Insert(<j, i, ω_j, ε_j>, B_j);
                 end
              end
        end
     end
```

Fig. 5. Table-Driven, Traffic-Engineering Dijkstra.

```
algorithm TD-QoS-Dijkstra
     begin
1    Push(<s, s, 0̄>, P_s);
2    for each {(s, j) ∈ A(s)}
3       Insert(<j, s, ω_sj>, T);
4    while (|T| = 0)
        begin
5       <i, p_i, ω_i> ← Min(T);
6       DeleteMin(B_i);
7       if (|B_i| = 0)
8          then DeleteMin(T)
9          else IncreaseKey(Min(B_i), T_i);
10      if (ω_i ⋢ Tail(P_i).ω)
           then begin
11            Push(<i, p_i, ω_i>, P_i);
12            for each {(i, j) ∈ A(i) | ω_i ⊕ ω_ij ⋢ Tail(P_i).ω}
                 begin
13               ω_j ← ω_i ⊕ ω_ij;
14               if (T_j = ∅)
15                  then Insert(<j, i, ω_j>, T)
16               else if (ω_j ≺ T_j.ω)
17                  then DecreaseKey(<j, i, ω_j>, T);
18               Insert(<j, i, ω_j>, B_j);
                 end
              end
        end
     end
```

Fig. 6. Table-Driven, QoS Dijkstra.

the difference that, as new routes are discovered, they are inserted in the heap T if there is some class of traffic that can satisfy the path predicate (the $SAT()$ test at line 13), and at the top of each iteration routes are pulled from T and discarded until one is found that is unique in P (in the sense that it includes a traffic class for which no route currently exists in P_i), which is then added to P.

The time complexity of the TD-TE-Dijkstra algorithm is dominated by the loops at lines 4 and 13. The loop at line 4 is executed at most once for each incomparable path (in terms of path predicates) to each node in the graph for a total of nA times. The loop at line 13 is executed at most once for each distinct instance of an edge in the graph, for a total of mA times. The most time consuming operation performed as part of the loop at line 4 is the deletion from the balanced tree B_i at line 6 of the best temporarily labeled route with per-operation cost of $\log(a_{max}A)$, and an aggregate cost of $nA\log(a_{max}A)$. The accesses in lines 7–9 to the best route in heap T have a per-operation cost $\log_d(n)$, for an aggregate cost of $mA\log(n)$. For the loop at line 13, the most time consuming operation is the addition to the balanced tree B_i at line 19 with a per-operation cost of $\log(a_{max}A)$, and an aggregate cost of $mA\log(a_{max}A)$. Therefore, the worst case time complexity of TD-TE-Dijkstra, dominated by the operation at line 19, is $O(mA\log(A))$.

B. Table-Driven QoS

Figure 6 presents a modified Dijkstra algorithm that computes an optimal set of routes to each destination subject to multiple general (additive or concave) path metrics. An *optimal set of routes* for a graph is a subset R_o of the routes in the graph such that for any requested set of metric constraints ω_r to a destination d, the shortest route $< d, p_s, \omega_s >$ to d in R_o such that $\omega_r \sqsubseteq \omega_s$ is a shortest path in the graph with path metrics satisfying the request constraints. The following theorem defines the goal of the TD-QoS-Dijkstra in terms of the path algebra presented above.

Theorem 1: Any maximal subset of the set of routes R induced by a given G, is an optimal set of routes for the network modeled by G.

Proof: By contradiction. Given a route request ω_r for destination d, let $R_{best} = < d, p_b, \omega_b >$ be the smallest route from a maximal set of routes for the graph where $\omega_r \sqsubseteq \omega_b$. Also assume the route is not the shortest path to d satisfying ω_r. There must exist a route $R_{shorter} = < d, p_s, \omega_s >$ that is not a maximal route for the graph where $\omega_r \sqsubseteq \omega_s$. This implies that $\omega_r \succeq \omega_b \succeq \omega_s$. This either implies that $\omega_b \sqsubseteq \omega_s$, which contradicts the fact that R_{best} is a maximal route

for the graph, or that there exists another maximal route $R'_{best} = < d, p'_b, \omega'_b >$ where $\omega_s \sqsubseteq \omega'_b$ (since $R_{shorter}$ is not a maximal route), which implies that $\omega'_b \preceq \omega_s \preceq \omega_b \preceq \omega_r$, which contradicts the fact that R_{best} is the smallest maximal route $\preceq \omega_r$. ■

Therefore, the route computation involves the computation of a maximal set of routes for the input graph, and the route lookup function involves finding the smallest route from this set such that the request is $\sqsubseteq$ the maximal route.

The complexity of TD-QoS-Dijkstra is of the same form as that for TD-TE-Dijkstra (Section III-A), except for the source of the limit on the maximum number of incomparable paths to a destination. For TD-QoS-Dijkstra the maximum number of incomparable paths (in terms of incomparable path weight values) paths to a given node in the graph is limited by the number of unique path weight values, W. Similar to TD-TE-Dijkstra the time complexity of the TD-QoS-Dijkstra algorithm is dominated by the loops at lines 4 and 12. The loop at line 4 is executed at most once for each incomparable path (in terms of path weights) to each node in the graph for a total of nW times. The loop at line 12 is executed at most once for each distinct instance of an edge in the graph, for a total of mW times. The most time consuming operation performed as part of the loop at line 4 is the deletion from the balanced tree B_i at line 6 of the best temporarily labeled route with per-operation cost of $\log(a_{max}W)$, and an aggregate cost of $nW \log(a_{max}W)$. The accesses in lines 7–9 to the best route in heap T have a per-operation cost $\log_d(n)$, for an aggregate cost of $mW \log(n)$. For the loop at line 12, the most time consuming operation is the addition to the balanced tree B_i at line 18 with a per-operation cost of $\log(a_{max}W)$, and an aggregate cost of $mW \log(a_{max}W)$. Therefore, the worst case time complexity of TD-TE-Dijkstra, dominated by the operation at line 18, is $O(mW \log(W))$.

TD-QoS-Dijkstra performs the same computation as Jaffe's pseudopolynomial time algorithm for the MCP problem [8] using only worst case $O(nW)$ space complexity compared with guaranteed space complexity of $O(n^2 b)$ (where b is the largest metric value) of Jaffe's algorithm. The space savings comes from directly computing the maximal set of routes. From this set of routes, all $n^2 b$ routes computed by Jaffe's algorithm can be inferred with minimal overhead.

C. Table-Driven Traffic Engineering and QoS

Figure 7 presents a modified Dijkstra algorithm that computes an optimal set of routes to each destination subject to multiple general (additive or concave) path metrics, in the presence of traffic constraints on the links. The time complexity of Policy-Based-Dijkstra is dominated by the loops at lines 4, 11, and 15. The loop at line 4 is executed nWA times, and the loop at line 15 mWA times. The loop at line 11 scans the entries in P_i to verify a new route is best for some truth assignment. For a given destination, this loop is executed at most an incrementally increasing number of times, starting at 0 and growing to $WA - 1$ (the maximum number of unique routes to a given destination) for a total of $\sum_{i=1}^{WA-1} i = \frac{(WA-1)WA}{2}$ times. For completeness, the statements at lines 6 and 21 take time proportional to $\log(a_{max}WA)$ for a total of $nWA \log(a_{max}WA)$ and $mWA \log(a_{max}WA)$, respectively; and those in lines 7–9 and 17–20 proportional to $\log_d(n)$ for a total of $nWA \log_d(n)$ and $mWA \log_d(n)$, respectively. Therefore, the worst case time complexity of Policy-Based-Dijkstra, dominated by the loop at line 11, is $O(nW^2 A^2)$.

```
algorithm Policy-Based-Dijkstra
     begin
1    Push(<s, s, 0̄, 1>, P_s);
2    for each {(s, j) ∈ A(s)}
3       Insert(<j, s, ω_sj, ε_sj>, T);
4    while (|T| = 0)
        begin
5       <i, p_i, ω_i, ε_i> ← Min(T);
6       DeleteMin(B_i);
7       if (|B_i| = 0)
8          then DeleteMin(T)
9          else IncreaseKey(Min(B_i), T_i);
10      ε_tmp ← ε_i; ptr ← Tail(P_i);
11      while ((ε_tmp ≠ 0) ∧ (ptr ≠ ∅))
12         ε_tmp ← ε_tmp ∧ ¬ptr.ε; ptr ← ptr.next;
13      if (ε_tmp ≠ 0)
           then begin
14            Push(<i, p_i, ω_i, ε_i>, P_i);
15            for each {(i, j) ∈ A(i) | SAT(ε_i ∧ ε_ij}
                 begin
16               ω_j ← ω_i ⊕ ω_ij; ε_j ← ε_ij;
17               if (T_j = ∅)
18                  then Insert(<j, i, ω_j, ε_j>, T)
19               else if (ω_j ≺ T_j.ω)
20                  then DecreaseKey(<j, i, ω_j, ε_j>, T);
21               Insert(<j, i, ω_j, ε_j>, B_j);
                 end
              end
        end
     end
```

Fig. 7. General-Policy-Based Dijkstra.

The loop at line 11, which dominates the cost of Policy-Based-Dijkstra, is required because there is no way to summarize the permanent routes for a destination. However, for the traffic engineering and QoS variants of this algorithm, the permanent routes can be summarized by a summary traffic expression (formed by the disjunction of permanent route path predicates) and the weight of the last route, respectively. Using these shortcuts, the

complexity of the traffic engineering and QoS algorithms are $O(mA\log(A))$ and $O(mW\log(W))$, respectively.

D. Performance Results

Figures 8 through 12 present the results of experiments run using the TD-QoS-Dijkstra algorithm. The experiments were run on a 1GHz Intel Pentium 3 based system. The algorithms were implemented using the C++ Standard Template Library (STL) and the Boost Graph Library. Each test involved running the algorithm on ten random weight assignments to ten randomly generated graphs (generated using the GT-ITM package [18]). For each test the worst case measurements are graphed. The metrics were generated using the "Cost 2" scheme from [14] where the delay component is randomly selected in the range $1..MaxMetric$, and the cost component is computed as $cost = \sigma(MaxMetric - delay)$, where σ is a random integer in the range 1..5; this scheme was chosen as it proved to result in the most challenging computations from a number of different schemes considered. The QoS routing problem was used for these tests as it was easiest to generate meaningful random metric assignments for. Space overhead was measured in terms of the maximum number of entries stored in the B_* structures.

Tests were run for performance (both runtime and space) as a function of graph size, average degree of the graph, and the maximum link metric value. Due to space constraints, only the graphs for size are shown here. Also, since the maximum metric was shown to have little impact on performance, only results for tests with a maximum metric of 1000 are presented here. Figures 9 and 11 present the results, showing that, while costs increase with both graph size and average degree, both the magnitude and rate of growth are surprisingly tame for what are fundamentally non-polynomial algorithms. While runtime grows to approximately 4 seconds for the largest problems, for graphs smaller than 500 nodes with an average degree of 8 (well beyond the scale supportable by current Internet routing protocols) the runtime is at most a few hundred milliseconds, and the growth rate is barely beyond linear in this range of parameters. Similarly, the worst-case space utilization stays below 100,000 entries (consuming less than 10MB of memory) with similar growth rates.

One interesting observation made during this testing was that the primary driver of the performance results is the number of routes considered for addition to the set T (e.g. at line 13 of Figure 5), which we call *candidate routes*. Figure 8 plots the number of candidate routes as a function of graph size. Comparing Figure 8 with Figures 9 and 11 shows that the growth rate and relative magnitude of runtime and space reflect those of the number of candidate routes. The graphs of runtime and space normalized per candidate route, shown in Figures 10 and 12, identify strategies for improving the performance of these algorithms. Figure 10 illustrates the logarithmic growth of operations on heap T as the size of T grows, and Figure 12 illustrates that, as the size of the graph grows, the effectiveness of the temporary route discard strategy improves. These observations suggest the strategies for improving the performance of these algorithms of improving the efficiency of the data structures for storing temporary routes, and adopting more aggressive strategies for discarding temporary routes.

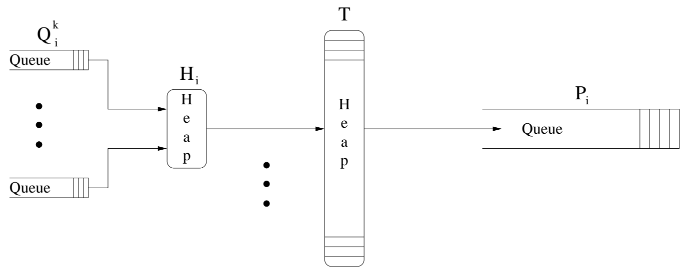

Fig. 13. Model of Data Structures for Enhanced Algorithms

In summary, the results showed: excellent runtime performance in the range of parameters expected from routing domains in the Internet (i.e. average degree less than 5, and routing graph size less than 500); the algorithms exhibited very well-behaved growth rates (given their basic NP-completeness); and they exhibited very reasonable space overhead in all scenarios.

IV. Enhanced Policy-Based Routing Algorithms

The $\log(A)$ and $\log(W)$ factors in the complexity of the traffic engineering and QoS variants of the Policy-Based-Dijkstra algorithm (respectively) are the result of the use of a balanced tree for storing the temporarily labeled nodes for a given destination. This section presents enhanced versions of these algorithms which use a queue-based data structure for this purpose, reducing the cost of managing these structures to a lower order term in the time complexity. As a result the runtime cost of the enhanced algorithms becomes dominated by $\log_d(n)$ factors from the manipulation of the T heap.

This enhancement is based on the property that routes to a given node *with the same predecessor* are discovered in strictly increasing (or non-decreasing, depending on the algorithm) order. This property is a result of the fact that routes to a given predecessor will be discovered in strictly increasing (non-decreasing) order, and therefore

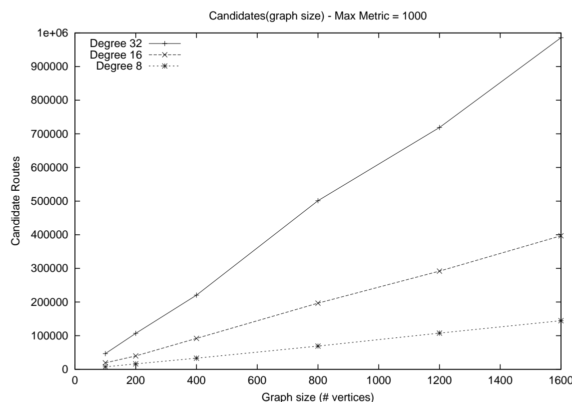

Fig. 8. Basic – Candidate Routes(Size)

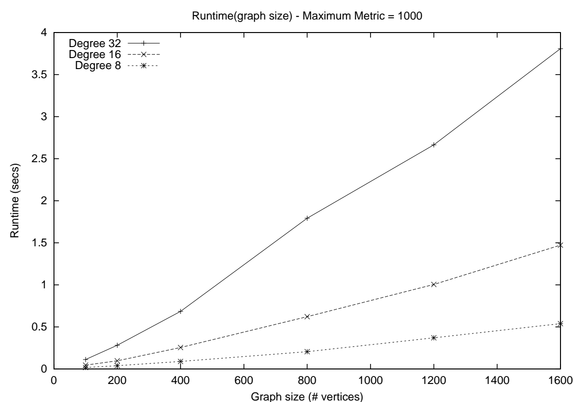

Fig. 9. Basic – Runtime(Size)

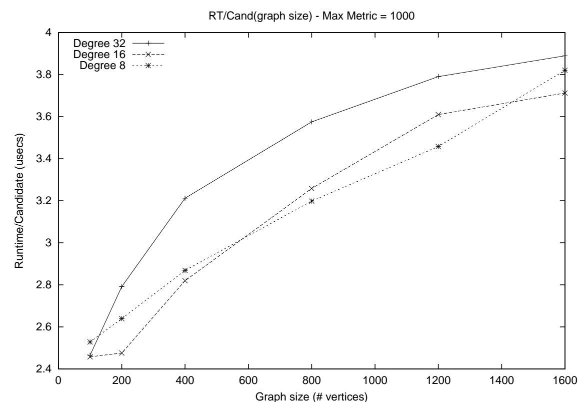

Fig. 10. Basic – Normalized Runtime(Size)

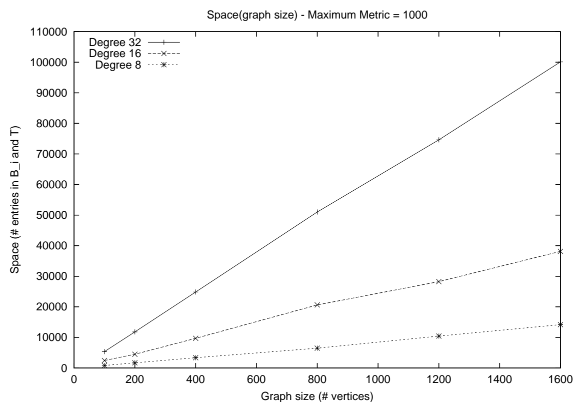

Fig. 11. Basic – Space(Size)

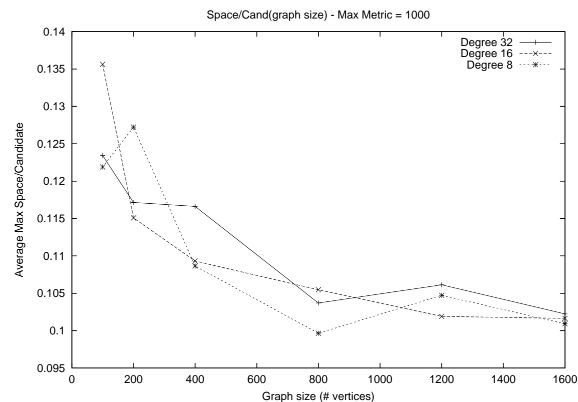

Fig. 12. Basic – Normalized Space(Size)

the order of discovery of routes from a given predecessor to one of its neighbors will have the same property.

Based on this insight, the data structure shown in Figure 13 can be used to improve the performance of the algorithms presented in Section III. In this data structure the balanced trees for each node are replaced with a set of queues for each neighbor of the node, and a summary heap containing the head of each neighbor queue. Exploiting the ordering property of these queues, the algorithms ensure that each node head H_i, and therefore T_i, contain the lightest route in the link queues that is not subsumed by the routes in P_i. Due to space constraints, only the QoS version of these algorithms is presented here.

Figures 26 and 27 present the enhanced versions of the TD-TE-Dijkstra and TD-QoS-Dijkstra algorithms, respectively. As described in Table I, the new notation $\mathcal{E}_n^k$ represents a traffic expression representing all routes that have been added to Q_n^k. Similar to the algorithms presented in Section III-A, the correctness of these algorithms is based on the fact that invariants presented in Section III are maintained by the enhanced algorithm. Specifically, as detailed in the comments to these algorithms, constraints 1 and 2 are maintained by the

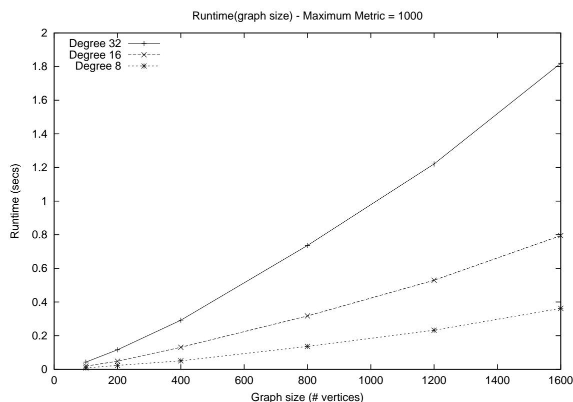

Fig. 14. Enhanced Runtime(Size)

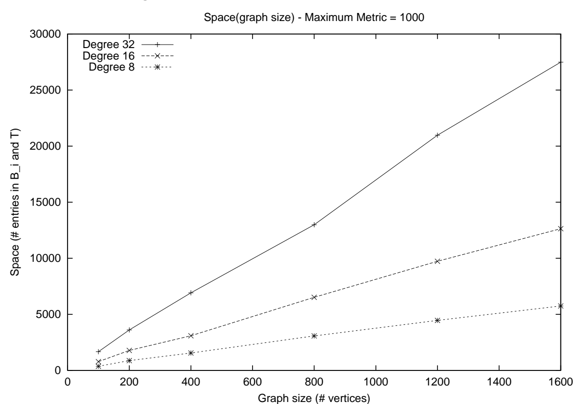

Fig. 15. Enhanced Space(Size)

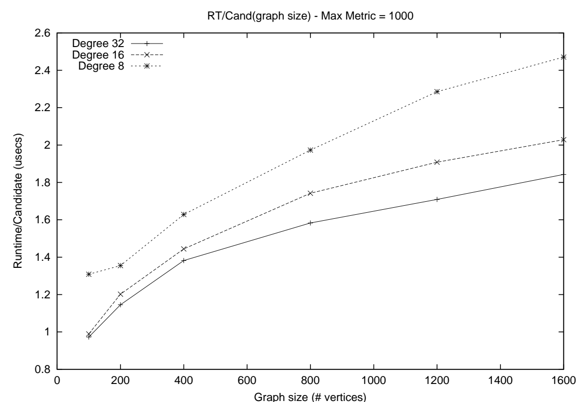

Fig. 16. Enhanced Normalized Runtime(Size)

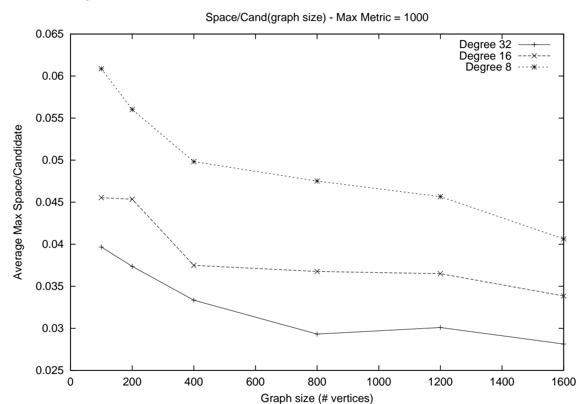

Fig. 17. Enhanced Normalized Space(Size)

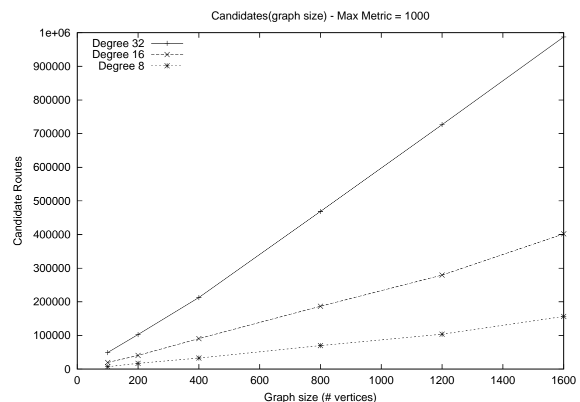

Fig. 18. Enhanced Candidate Routes(Size)

DeleteTMin() and AddCandidate() functions and, based on this, the Dijkstra iteration over the nth best route in the main body of the algorithm maintains the invariants.

The runtime complexity of the TD-QoS-Dijkstra algorithm (again, ignoring the cost for determining satisfiability) is dominated by the loops at lines 6 and 10. The loop at line 6 is executed at most once for each incomparable path to each node in the graph for a total of nW times. The loop at line 10 is executed at most once for each distinct instance of an edge in the graph, for a total of mW times. The most costly operation in the loop at line 6 is the DeleteTMin() call at line 9. In the DeleteTMin() routine, the loop at line 7 will be executed, in total, at most once per neighbor for each forwarding class for a total of $a_{max}W$, and the cost per call of the heap operations at lines 13 and 14 is $d\log_D(n)$. Therefore, the total worst-case cost of the call at line 8 of the main algorithm is $nW\log_d(n)+a_{max}W$. In the AddCandidate() routine, the runtime complexity is dominated by the heap operations at lines 5, 20, and 23, which cost $\log_d(n)$ each, for a total cost of the call to AddCandidate() at line 12 of the main algorithm of $mW\log_d(n)$. Therefore, the worst-case time complexity of the enhanced TD-QoS-Dijkstra algorithm is $O(mW\log(n))$.

Figures 14 through 18 show the results of running the same tests described above with the enhanced version of the TD-QoS-Dijkstra algorithm. These graphs show a significant improvement in the performance of the enhanced algorithms in comparison to the basic algorithms, and give an indication of the sources of these savings. The flattening of the graph of space requirements as a function of the graph size (Figure 17) is a result of the far more aggressive candidate dropping performed in the enhanced algorithms. With the much smaller resulting data structures there is a corresponding decrease in performance costs across all the graphs. Specifically,

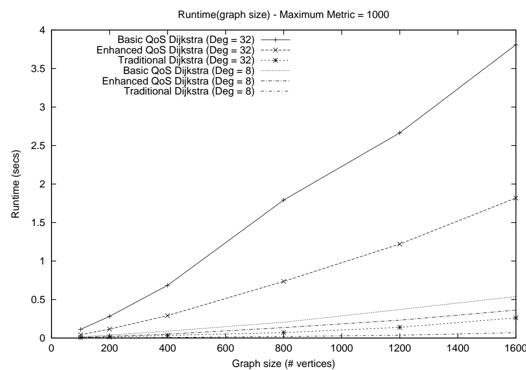

Fig. 19. Compare Runtime(Size)

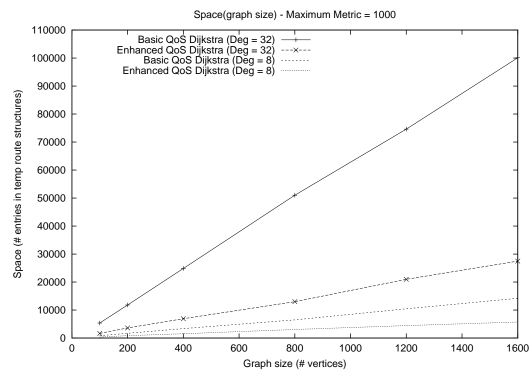

Fig. 20. Compare Space(Size)

the conditional return at line 15 of the enhanced QoS AddCandidate() routine ensures that each Q_i^p contains only incomparable routes at all times, and the loop at line 8 of the QoS DeleteTMin() routine ensures that the front of all Q_i^p for a give i contain only incomparable routes following the transfer of a route from i to P. The only remaining source of "excess" routes held in the data structures, and processed by the algorithms are the comparable routes carried in different Q_i^p for destination i until they are cleaned up the the DeleteTMin() loop.

Similarly, the fact that the graph of runtime as a function of the graph size (Figure 14 has flattened noticeably is the result of replacing the log()-time balanced tree data structure for B_i with a constant-time queue data structure for Q_i^p. In addition, the very low cost of the simple queue data structure contributes to the decrease in cost across all graphs. In summary, the more aggressive candidate drop strategy and cheaper queue-based data structure used in the enhanced algorithms result in a significant general reduction in the cost of the routing computations. Furthermore, the replacement of the log()-time balanced tree structures with constant-time queue data structures by these algorithms has damped the growth rate of their performance. Figures 19 and 20 compare the performance of the basic QoS, enhanced QoS, and traditional (single path) Dijkstra algorithms.

V. Hop-by-Hop Policy-Based Routing

The policy-based routing algorithms presented in Sections III and IV compute multiple routes to the same destination to satisfy the policy requirements of an internet. Such routes are not supported by current, host-address-based packet forwarding mechanisms that only allow one route per destination. The solution to this problem is to use label-swapping technology (such as that underlying MPLS [19]) as a generalized forwarding mechanism that replaces IP addresses as the identifiers used in the forwarding function with arbitrary labels which can be defined by the routing protocol to represent any policy/destination pair for which a route has been computed.

A significant innovation of the policy-based routing architecture presented here is the combination of a table-driven, hop-by-hop routing model with label-swap forwarding mechanisms, which we call *distributed label-swapping*. Traditionally, label-swap forwarding has only been seen as an appropriate match with an on-demand, source-driven routing model. Indeed, to a large extent, the virtual-circuit nature of these previous solutions has been attributed to their use of label-swap forwarding. Contrary to this view, the position taken here is that host addresses and labels are largely equivalent alternatives for representing forwarding state, and that the virtual-circuit nature of prior architectures derives from their use of a source-driven forwarding model. The primary conceptual difference between address and label-swap forwarding is that label-swap forwarding provides a clean separation of the control and forwarding planes [20] within the network layer, where address-based forwarding ties the two planes together. This separation provides what might be called a *topological anonymity* of the forwarding plane that is critical to the implementation of policy-based routes. Chandranmenon and Varghese [21] present a similar notion, which they call *threaded indices*, where neighboring routers share the indexes into their routing tables for specific routes which are then included in forwarded packets to allow rapid forwarding table lookups. In addition they present a modified Bellman-Ford algorithm that exchanges these labels among neighbors. Distributed label-swapping generalizes the threaded index concept to use generic labels (with no direct forwarding table semantics), uses these labels to represent routing policies computed by the routing protocols, and defines a family of routing protocols to exchange local labels among neighbors.

As illustrated in Figure 21, distributed label-swap for-

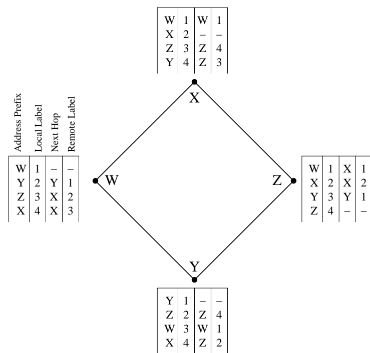

Fig. 21. Labels with Address-Based Forwarding

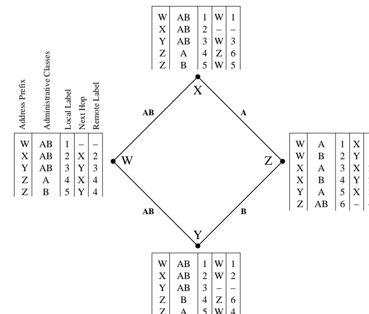

Fig. 22. Labels with Policy-Based Forwarding

warding can be used in the context of traditional address-based forwarding. In this example the forwarding table is referenced for both traffic classification (through the "address prefix" field), and for label-swap forwarding (through the "local label" field). The benefit of this mechanism for traffic forwarding is it can be generalized to handle policy-based forwarding. Specifically, distributed label-swap forwarding can be used to implement traffic engineering via the assignment of traffic to administrative classes which are used to select different paths for traffic to the same destination depending on the labeling of links in the network with administrative class sets. For example, Figure 22 shows a small network with four nodes, two administrative classes A and B, and the given forwarding state for reaching node 4. The benefits of this architecture are that it is based on forwarding state that is agnostic to the definition of forwarding classes, allowing the data forwarding plane to remain simple yet general; and it concentrates the path computation functions in the routing protocol, which is the least time critical, and most flexible component of the network layer.

The resulting routing architecture can be seen as analogous to the Reduced Instruction Set Computer (RISC) processor architecture in which researchers shifted much of the intelligence for managing the use of processor resources to the compilers that were able to bring a higher-level perspective to the task, thus allowing much more efficient use of the physical resources, as well as freeing the hardware designers to focus on performance issues of much simpler processor architectures. Similarly, the communications architecture proposed here requires a shift in intelligence for customized (i.e. policy-based) path composition to the routing protocols and frees the network layer to focus solely on hop-by-hop forwarding issues, adding degrees of freedom to the network hardware engineering problem that allow for significant advances in the performance and effectiveness of network infrastructure.

In this architecture, the role of the routing protocol takes on new significance. The routing protocols used in most of today's computer networks are based on shortest-path algorithms that can be classified as distance-vector or link-state. Distance-vector protocols work by propagating updates giving the distance to a destination to neighboring routers whose routing tables may change as a result of the update. Link-state protocols work by flooding updates describing the state of links in the network to all routers in the network. Recently, a hybrid class of protocols, called link-vector [22], has been defined that works by propagating link-state updates only to routers whose routing tables may change as a result of the update.

The enhancement of traditional unicast routing systems with the policy-based routing technology presented above is straight-forward. The routing protocol must be enhanced to carry the additional link metrics required to implement the desired policies. This requires the use of either a link-state or link-vector routing protocol that exchanges information describing link state. Note, however, that for a system depending on on-demand routing computations a link-state, complete topology protocol is required to ensure an ingress router has the information it needs to compute an optimal route. In contrast, hop-by-hop based routing systems can work with link-vector, partial topology protocols as each routing process is ensured of learning from its neighbors of all links composing optimal routes to all destinations in the internet.

Forwarding state must be enhanced to include local and next hop label information in addition to the destination and next hop information existing in traditional forwarding tables. Traffic classifiers must be placed at the edge of an internet, where "edge" is defined to be any point from which traffic can be injected into the internet. Since each router represents a potential traffic source (for

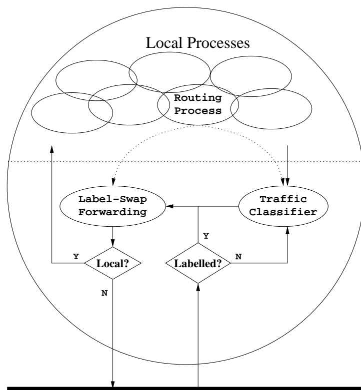

Fig. 23. Traffic Flow in Policy-Enabled Router

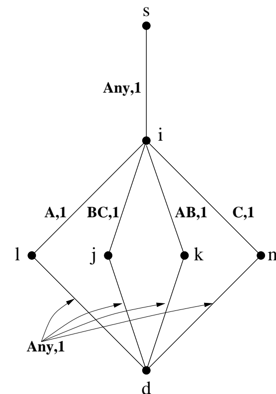

Fig. 24. Next Hop Problem with Policy-Based Routing

CLI and network management traffic), this effectively means a traffic classification component must be present in each router. As illustrated in Figure 23, the resulting traffic flow requirements are that all non-labeled traffic (sourced either from a router itself, or from a directly connected host or non-labeling router) must be passed through the traffic classifier first, and all labeled traffic (sourced either from the traffic classifier or a directly connected labeling router) must be passed to the label-swap forwarding process.

Lastly, the routing protocol must be enhanced to exchange information needed to compute the label swap components of its forwarding tables. The output of the routing algorithm is forwarding information described in terms of a destination, traffic expression, and path weight for each computed route. To be used for forwarding, this information must be augmented with local and next hop labels. To determine the next hop label for a given route the routing process requires the forwarding tables of its neighbors. Therefore, the final enhancement required of routing protocols is that they exchange local forwarding tables and use this information to compute the next hop label for their routes. One challenge presented by this requirement is that the routes computed by the routing algorithm must be assured of matching an active route in the selected next hop neighbor. As illustrated in Figure 24, this is not guaranteed by the algorithms presented above. Specifically, in this internet there are a number of equally "good" routes from nodes s and i to node d. For example, it is possible that the routing process at node i selects the paths through its neighbors l and j to provide two hop paths for traffic classes A, B, and C, while node s selects the paths that go through nodes k and m. In such a case there is no next hop label that can be chosen at s for routes to d that will satisfy the traffic policies.

To address this problem, Figure 25 presents an enhanced version of the basic traffic engineering algorithm for use in the context of hop-by-hop forwarding. In this algorithm, routes are augmented with two additional fields; n_d is the next hop neighbor for a route to destination d, and l_d is the next hop label for d. As described above, a partial forwarding table is maintained for each neighbor, specified by $F_n[d]$, containing an array of routes for each destination in the internet. Each entry in this array, denoted by $< d, \omega_d, \varepsilon_d, l_d >$, gives the weight, traffic expression, and next hop label for each route in the neighbor's forwarding table. In this algorithm, new paths are only considered if they are extensions of paths chosen by the neighbor which is the next hop to the predecessor to the path's destination. For example, from Figure 24, node s will only consider paths to destination d that are extensions of node i's paths to d through nodes l and j. A fringe benefit of this enhancement is the next hop label computation can now be integrated with the routing computation (as shown by the inclusion of the next hop label in the routes computed by the algorithm).

VI. Conclusions

This paper presents an enhanced routing architecture and family of routing algorithms that efficiently support the routing requirements of the knowledge plane. A family of routing algorithms is presented that efficiently compute routes in the context of traffic and performance constraints. The enhanced TD-TE-Dijkstra algorithm is the most efficient algorithm available for computing routes that satisfy traffic engineering requirements, and Policy-Based-Dijkstra is the first algorithm for computing routes that simultaneously satisfy traffic engineering and quality-of-service requirements. A traffic algebra was defined to formalize the notion of traffic constraints,

algorithm Hop-by-Hop-TD-TE-Dijkstra
begin
1 $Push(<s, s, 0, 1, s, \emptyset>,\ P_s);$
2 **for each** $\{(s, j) \in A(s)\}$
3 $Insert(<j, s, \omega_{sj}, \varepsilon_{sj}, j, \emptyset>,\ T);$
4 **while** $(|T| = 0)$
begin
5 $<i, p_i, \omega_i, \varepsilon_i, n_i, l_i> \leftarrow Min(T);$
6 $DeleteMin(B_i);$
7 **if** $(|B_i| = 0)$
8 **then** $DeleteMin(T)$
9 **else** $IncreaseKey(Min(B_i),\ T_i);$
10 **if** $(\neg(\varepsilon_i \rightarrow \mathcal{E}_i))$
then begin
11 $Push(<i, p_i, \omega_i, \varepsilon_i>,\ P_i);$
12 $\mathcal{E}_i \leftarrow \mathcal{E}_i \vee \varepsilon_i;$
13 **for each** $\{(i, j) \in A(i) \mid$
$(\exists <j, \omega'_j, \varepsilon'_j, l'_j> \in F_{n_i}[j] \mid$
$(\varepsilon_{sn_i} \wedge \varepsilon'_j = \varepsilon_i \wedge \varepsilon_{ij}) \wedge$
$(\omega_{sn_i} + \omega'_j = \omega_i + \omega_{ij})) \wedge$
$SAT(\varepsilon_i \wedge \varepsilon_{ij}) \wedge \neg((\varepsilon_i \wedge \varepsilon_{ij}) \rightarrow \mathcal{E}_j)\}$
begin
14 $\omega_j \leftarrow \omega_i + \omega_{ij};\ \varepsilon_j \leftarrow \varepsilon_i \wedge \varepsilon_{ij};$
15 **if** $(T_j = \emptyset)$
16 **then** $Insert(<j, i, \omega_j, \varepsilon_j, n_i, l'_j>,\ T)$
17 **else if** $(\omega_j < T_j.\omega)$
18 **then** $DecreaseKey(<j, i, \omega_j, \varepsilon_j, n_i, l'_j>,\ T);$
19 $Insert(<j, i, \omega_j, \varepsilon_j, n_i, l'_j>,\ B_j);$
end
end
end
end

Fig. 25. Hop-by-Hop TD-TE-Dijkstra.

and a set-based model was identified for efficiently implementing restricted but useful traffic engineering policies. An enhanced path algebra was presented that supports the computation of the best set of routes for a each destination. Lastly, the distributed label-swap forwarding architecture was defined that efficiently implements multiple paths per destination required for hop-by-hop, policy-based packet forwarding based on label-swapping, and a hop-by-hop version of the TD-TE-Dijkstra algorithm was presented for computing routes that can be implemented in this architecture.

REFERENCES

[1] Vinton G. Cerf, "The Catenet Model for Internetworking," IEN 48, July 1978.

[2] Vinton G. Cerf and Edward Cain, "The DoD Internet Architecture Model," *Computer Networks*, vol. 7, pp. 307–318, 1983.

[3] Vinton G. Cerf and Robert E. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Transactions on Communications*, vol. COM-22, no. 5, pp. 637–648, May 1974.

[4] Bob Braden, David Clark, and Scott Shenker, "Integrated Services in the Internet Architecture: an Overview," RFC1633, July 1994.

[5] Dave Clark, "A New Vision for Network Architecture," http://www.isi.edu/knowplane/DOCS/DDC_knowledgePlane_3.pdf, 2002.

[6] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," *ACM Trans. on Computer Systems*, vol. 2, no. 4, pp. 277–288, Nov. 1984.

[7] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman & Co., 1979.

[8] Jeffrey M. Jaffe, "Algorithms for Finding Paths with Multiple Constraints," *Networks*, vol. 14, no. 1, pp. 95–116, 1984.

[9] Shigang Chen and Klara Nahrstedt, "An Overview of Quality of Service Routing for Next-Generation High-Speed Networks: Problems and Solutions," *IEEE Network*, pp. 64–79, Nov. 1998.

[10] Whay C. Lee, Michael G. Hluchyi, and Pierre A. Humblet, "Routing Subject to Quality of Service Constraints in Integrated Communication Networks," *IEEE Network*, vol. 9, no. 4, pp. 46–55, Aug. 1995.

[11] Zheng Wang and Jon Crowcroft, "Quality-of-Service Routing for Supporting Multimedia Applications," *IEEE Journal on Selected Areas in Communications*, pp. 1228–1234, Sept. 1996.

[12] Qingming Ma and Peter Steenkiste, "Quality-of-Service Routing for Traffic with Performance Guarantees," in *Proceedings 4th International IFIP Workshop on QoS*. IFIP, May 1997.

[13] D. Cavendish and M. Gerla, "Internet QoS Routing using the Bellman-Ford Algorithm," in *Proceedings IFIP Conference on High Performance Networking*. IFIP, 1998.

[14] Stavroula Siachalou and Leonidas Georgiadis, "Efficient QoS Routing," in *Proceedigns of INFOCOM'03*. IEEE, Apr. 2003.

[15] Piet Van Mieghem, Hans De Neve, and Fernando Kuipers, "Hop-by-hop quality of service routing," *Computer Networks*, vol. 37, pp. 407–423, Nov. 2001.

[16] João Luís Sobrinho, "Algebra and Algorithms for QoS Path Computation and Hop-by-Hop Routing in the Internet," *IEEE/ACM Transactions on Networking*, vol. 10, no. 4, pp. 541–550, Aug. 2002.

[17] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin, *Network Flows – Theory, Algorithms, and Applications*, Prentice Hall, 1993.

[18] Ellen W. Zegura, Ken Calvert, and S. Bhattacharjee, "How to Model an Internetwork," in *Proceedings INFOCOM '96*. IEEE, 1996.

[19] Bruce Davie and Yakov Rekhter, *MPLS: Technology and Applications*, Morgan Kaufmann, 2000.

[20] George Swallow, "MPLS Advantages for Traffic Engineering," *IEEE Communications Magazine*, vol. 37, no. 12, pp. 54–57, Dec. 1999.

[21] Girish P. Chandranmenon and George Varghese, "Trading Packet Headers for Packet Processing," *IEEE ACM Transactions on Networking*, vol. 4, no. 2, pp. 141–152, Oct. 1995, 1995.

[22] J.J. Garcia-Luna-Aceves and Jochen Behrens, "Distributed, Scalable Routing Based on Vectors of Link States," *IEEE Journal on Selected Areas in Communications*, Oct. 1995.

algorithm TD-TE-Dijkstra
begin
1 **for each** $\{i \in N\}$ $\mathcal{E}_i \leftarrow 0$;
2 **for each** $\{(i,j) \in E\}$ $\varepsilon_i^j \leftarrow \varepsilon_j^i \leftarrow 0$;
3 $Push(<s,s,0,1>,\ P_s)$;
4 **for each** $\{(s,j) \in A(s)\}$
begin
5 $Push(<j,s,\omega_{sj},\varepsilon_{sj}>,\ Q_j^s)$;
6 $Insert(<j,s,\omega_{sj},\varepsilon_{sj}>,\ H_j)$;
7 $Insert(<j,s,\omega_{sj},\varepsilon_{sj}>,\ T)$;
end
8 **while** $(|T| > 0)$
begin
9 $<i,p,\omega,\varepsilon> \leftarrow Min(T)$;
10 $\mathcal{E}_i \leftarrow \mathcal{E}_i \vee \varepsilon$;
11 $Push(<i,p,\omega,\varepsilon>,\ P_i)$;
12 DeleteTMin();
13 **for each** $\{(i,j) \in A(I)\}$
begin
14 $\omega_j \leftarrow \omega \oplus \omega_{ij}$; $\varepsilon_j \leftarrow \varepsilon \wedge \varepsilon_{ij}$;
15 AddCandidate($<j,i,\omega_j,\varepsilon_j>$);
end
end
end

function TE-DeleteTMin()
// Delete minimum entry from T and restore invariants.
// Constraint 1 – only deletes routes (line 9) where
// every STA for the route also satisfies a known
// better route (a route in P_n).
// Constraint 2 – loop at line 7 ensures there are some
// STAs for new T_n that do not satisfy any known
// better route (current T_n or any route in P_n).
// Assumes $\mathcal{E}_n$ has been updated with $Min(T).\varepsilon$.
begin
1 $<i,p,\omega,\varepsilon> \leftarrow Min(T)$;
2 $Pop(Q_i^p)$;
3 **if** $(|Q_i^p| > 0)$
4 **then** $IncreaseKey(Head(Q_i^p),\ H_i^p)$
5 **else** $DeleteMin(H_i)$;
6 **if** $(|H_i| > 0)$
then begin
// Find smallest route in link queues
// where $\neg(\varepsilon \rightarrow \mathcal{E}_i)$.
7 **for each** $\{(i,k) \in A(i) \mid (|Q_i^k| > 0) \wedge (Head(Q_i^k).\varepsilon \rightarrow \mathcal{E}_i)\}$
begin
8 **while** $((|Q_i^k| > 0) \wedge (Head(Q_i^k).\varepsilon \rightarrow \mathcal{E}_i))$
9 $Pop(Q_i^k)$;
10 **if** $(|Q_i^k| > 0)$
11 **then** $IncreaseKey(Head(Q_i^k),\ H_i^k)$
12 **else** $Delete(H_i^k)$;
end
13 **if** $(|H_i| > 0)$
then $IncreaseKey(Min(H_i),\ T_i)$; **return;**
end
14 $DeleteMin(T)$;
end

function TE-AddCandidate($<j,i,\omega_j,\varepsilon_j>$)
// Add new route to appropriate Q and restore invariants.
// Constraint 1 – only drops routes (lines 1, 10, and 21)
// where every STA for the route also satisfies a known
// better route.
// Constraint 2 – ensures there is an STA for $Min(H_j)$
// that is not satisfied by any known better route.
begin
1 **if** $(\varepsilon_j \rightarrow \mathcal{E}_j)$ **then return;**
2 **if** $(|H_j| = 0)$
then begin
3 $Push(<j,i,\omega_j,\varepsilon_j>,\ Q_j^i)$;
4 $Insert(<j,i,\omega_j,\varepsilon_j>,\ H_j)$;
5 $Insert(<j,i,\omega_j,\varepsilon_j>,\ T)$;
6 **return;**
end
7 $<j,k,\omega_m,\varepsilon_m> \leftarrow Min(H_j)$;
8 **if** $(\omega_m \leq \omega_j)$
then // Know $\neg(\varepsilon_j \rightarrow \mathcal{E}_j)$ from line 1.
9 **if** $(\varepsilon_j \rightarrow (\varepsilon_m \vee \varepsilon_j^i)))$
10 **then return;** // All STAs for ε_j satisfy a better route.
11 **else begin** // $\neg(\varepsilon_j \rightarrow (\varepsilon_m \vee \varepsilon_j^i)))$
// There is an STA for ε_j that does not satisfy
// any known better routes.
12 **if** $(|Q_j^i| = 0)$
13 **then** $Insert(<j,i,\omega_j,\varepsilon_j>,\ H_j)$
14 $Push(<j,i,\omega_j,\varepsilon_j>,\ Q_j^i)$;
end
else // $\omega_m > \omega_j$; since $\omega_j > Min(H_j^i)$, it must be
// true that $|Q_j^i| = 0$.
if $(\neg(\varepsilon_m \rightarrow (\varepsilon_j \vee \mathcal{E}_j)))$
then begin
// There is an STA for ε_m that doesn't satisfy any
// known better route.
15 $Push(<j,i,\omega_a,\varepsilon_j>,\ Q_j^i)$;
16 $Insert(<j,i,\omega_j,\varepsilon_j>,\ H_j)$;
// Following replaces $<j,k,\omega_m,\varepsilon_m>$.
17 $DecreaseKey(<j,i,\omega_j,\varepsilon_j>,\ T_j)$;
end
else begin // $(\varepsilon_m \rightarrow (\varepsilon_j \vee \mathcal{E}_j))$
// All STAs for ε_m also satisfy a better route.
18 $Push(<j,i,\omega_j,\varepsilon_j>,\ Q_j^i)$;
// Following replaces $<j,k,\omega_m,\varepsilon_m>$.
19 $DecreaseKey(<j,i,\omega_j,\varepsilon_j>,\ H_j^k)$;
20 $DecreaseKey(<j,i,\omega_j,\varepsilon_j>,\ T_j)$;
21 $Pop(Q_j^k)$;
22 **if** $(|Q_j^k| > 0)$
23 **then** $Insert(Head(Q_j^k),\ H_j)$;
end
end

Fig. 26. Enhanced Traffic Engineering Dijkstra.

algorithm TD-QoS-Dijkstra
begin
1 $Push(<s, s, \overline{0}>,\ P_s)$;
2 **for each** $\{(s, j)\ \in\ A(s)\}$
begin
3 $Push(<j, s, \omega_{sj}>,\ Q_j^s)$;
4 $Insert(<j, s, \omega_{sj}>,\ H_j)$;
5 $Insert(<j, s, \omega_{sj}>,\ T)$;
end;
6 **while** $(|T|\ >\ 0)$
begin
7 $<i, p, \omega> \leftarrow\ Min(T)$;
8 $Push(<i, p, \omega>,\ P_i)$;
9 DeleteTMin();
10 **for each** $\{(i, j)\ \in\ A(i)\}$
begin
11 $\omega_j\ \leftarrow\ \omega \oplus \omega_{ij}$;
12 AddCandidate($<j, i, \omega_j>$);
end
end
end

function QoS-DeleteTMin()
// Delete minimum entry from T and restore invariants:
// Constraint 1 – only deletes routes (line 9) that are
// $\sqsubseteq$ another route.
// Constraint 2 – loop at line 7 ensures new $T_i\ \not\sqsupseteq$
// new $Tail(P_i)$.
begin
1 $<i, p, \omega> \leftarrow\ Min(T)$;
2 $Pop(Q_i^p)$;
3 **if** $(|Q_i^p|\ >\ 0)$
4 **then** $IncreaseKey(Head(Q_i^p),\ H_i^p)$
5 **else** $DeleteMin(H_i)$;
6 **if** $(|H_i|\ >\ 0)$
then begin
// Find smallest route in link queues that is not
// $\sqsubseteq$ the deleted route.
7 **for each** $\{(i, k)\ \in\ A(i)\ \mid\ (|Q_i^k|\ >\ 0) \wedge (Head(Q_i^k).\omega\ \sqsubseteq\ \omega)\}$
begin
8 **while** $((|Q_i^k|\ >\ 0)\ \wedge\ (Head(Q_i^k).\omega\ \sqsubseteq\ \omega))$
9 $Pop(Q_i^k)$;
10 **if** $(|Q_i^k|\ >\ 0)$
11 **then** $IncreaseKey(Head(Q_i^k),\ H_i^k)$
12 **else** $Delete(H_i^k)$;
end
13 **if** $(|H_i|\ >\ 0)$
then $IncreaseKey(Min(H_i),\ T_i)$; **return;**
end
14 $DeleteMin(T)$;
end

function QoS-AddCandidate($<j, i, \omega_j>$)
// Add new route to appropriate Q and restore invariants:
// Constraint 1 – only drops known comparable routes
// (lines 1, 10, 15, and 24).
// Constraint 2 – ensures $Min(H_j)\ \preceq$ (and therefore $\not\sqsupseteq$)
// all routes in Q_j^* queues.
begin
1 **if** $(\omega_j\ \sqsubseteq\ Tail(P_j).\omega)$ **then return;**
2 **if** $(|H_j|\ =\ 0)$
then begin
3 $Push(<j, i, \omega_j>,\ Q_j^i)$;
4 $Insert(<j, i, \omega_j>,\ H_j)$;
5 $Insert(<j, i, \omega_j>,\ T)$;
6 **return;**
end
7 $<j, k, \omega_m> \leftarrow\ Min(H_j)$;
8 **if** $(\omega_m\ \preceq\ \omega_j)$
then
9 **if** $(\omega_j\ \sqsubseteq\ \omega_m)$
10 **then return;**
11 **else begin** // $((\omega_j\ \not\sqsubseteq\ \omega_m)\ \wedge\ (\omega_m\ \preceq\ \omega_j))$
12 **if** $(|Q_j^i|\ =\ 0)$
13 **then** $Insert(<j, i, \omega_j>,\ H_i)$
14 **else if** $(\omega_j\ \sqsubseteq\ Tail(Q_j^i).\omega)$
15 **then return;**
16 $Push(<j, i, \omega_j>,\ Q_j^i)$;
end
else // $\omega_j \prec \omega_m$; since $\omega_j\ \succ\ Min(H_j^i)$, it must be
// true that $|Q_j^i|\ =\ 0$.
17 **if** $(\omega_j\ \not\sqsupseteq\ \omega_m)$
then begin
18 $Push(<j, i, \omega_j>,\ Q_j^i)$;
19 $Insert(<j, i, \omega_j>,\ H_j)$;
// Following replaces $<j, k, \omega_m>$.
20 $DecreaseKey(<j, i, \omega_j>,\ T_j)$;
end
else begin // $(\omega_j\ \sqsupseteq\ \omega_m)$
21 $Push(<j, i, \omega_j>,\ Q_j^i)$;
// Following replaces $<j, k, \omega_m>$.
22 $DecreaseKey(<j, i, \omega_j>,\ H_j^k)$;
23 $DecreaseKey(<j, i, \omega_j>,\ T_j)$;
24 $Pop(Q_j^k)$;
25 **if** $(|Q_j^k|\ >\ 0)$
26 **then** $Insert(Head(Q_j^k),\ H_j)$;
end
end

Fig. 27. Enhanced QoS Dijkstra.