Efficient Policy-Based Routing without Virtual Circuits *

Bradley R. Smith and JJ 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

The inclusion of multiple metrics in a routing computation is called policy-based routing. Previous work on solutions to this problem have focused on virtual-circuit-based solutions, and have resulted in computationally expensive algorithms. This paper presents a number of advances in the provision of policy-based routing services in networks and internetworks. An integrated policy-based routing architecture is formulated where the general problem is decomposed into a traffic engineering problem of computing routes in the context of administrative traffic constraints, and a quality-of-service (QoS) problem of computing routes in the context of performance-related path constraints. A family of routing algorithms are presented for computing routes in the context of these constraints which achieve new levels of computational efficiency. Lastly, a forwarding architecture is presented that efficiently supports hop-by-hop forwarding in the context of multiple paths to each destination, which is required for policy-based routing.

1. Introduction

The architecture of today's Internet is based on the *catenet model for internetworking* [5, 6, 7]. 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 [3]. Real-time applications, such as on-demand streaming, audio and video conferencing, visualization, and virtual reality require varying degrees of bandwidth, delay, and delay jitter commitments from the network infrastructure, which render shortest-path routing insufficient and call for the generalization of the basic routing model of the Internet to satisfy constraints on multiple metrics.

Supporting the efficient management of network resources, traffic engineering [2] and network management services require the ability to control the allocation of these resources among network flows in an internet. The original motivation for traffic engineering was the need by IP network providers of the ability to manage network bandwidth in the context of the single-path routing model of IP networks [19]. Lacking such capabilities, the tendency of single-path routing is to aggregate traffic for a given destination onto a subset of the possible paths to that destination. As a result, networks frequently experience congestion in spite of the availability of excess capacity for the offered load.

The vulnerability of IP services to basic network disclosure and denial of service threats is another result of the lack of traffic engineering capabilities from current IP mechanisms. Excluding computationally expensive firewall mechanisms, IP traffic can potentially traverse any link in an internet. Therefore, the disclosure of any traffic stream and the denial-of-service attack of any destination in an internet

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

are unavoidable threats. Fundamental to these new requirements is the ability to control the topology used to forward traffic.

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.

While, in general, routing with multiple constraints is an NP-complete problem [12, 13], there are many subclasses of this general problem that have been shown to have polynomial-time solutions. For example, any problem involving two metrics with at least one of them being concave can be solved in polynomial-time by a traditional shortest path algorithm on the graph in which all links that do not comply with the concave constraints have been pruned [9, 14, 20]. However, even for this case, as the number of constraints becomes exponential in the size of the graph, this result no longer holds.

The foundational work on the problem of computing routes in the context of more than one additive metric was done by Jaffe [13], 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 [20] 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 [15] 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 [4] 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 [17] on MCP has resulted in an algorithm with complexity $O(nW \log(n))$ (in terms defined in Section 3). This algorithm is similar to the QoS algorithm presented in Section 4 in that it is an enhanced version of the Dijkstra algorithm based on invariants similar to those underlying the algorithms presented in Sections 3 and 4. However, due to errors in the algorithm (not discussed further here due to space constraints), it does not compute correct results.

Several other algorithms have been proposed for computing approximate solutions to the QoS routing problem. Both Jaffe [13] 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 [13, 16] 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 this paper, the inclusion of multiple metrics in a routing computation is called *policy-based* routing [9, 20]. Policy-based routing supports *traffic engineering* by the computation of routes in the context of administrative constraints on the type of traffic allowed over portions of an internet. Analogously, policy-based routing supports *quality-of-service* (QoS) by the computation of routes in the context of performance-related constraints on the paths specific traffic flows are allowed to use. The drawbacks of the current policy-based routing solutions are that they have poor average case performance, they implement inflexible routing models, and solutions for computing approximate solutions do not work with the traffic constraints used for traffic engineering. 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, quality-of-service constraints, 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 2, 3, and 4 present the model and algorithms for computing optimal routes in the context of policy-based link metrics. The algorithms are shown to be correct and highly efficient.

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 centralized 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 5 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.

2. 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 expressions* 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 expression* 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 expressions 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:

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

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 expressions identify the traffic classes allowed to traverse the link, and are denoted by ε_{ij} in the algorithms. Path expressions, denoted by ε_p in the algorithms, and defined as $\varepsilon_p = \varepsilon_{x_1 x_2} \wedge \varepsilon_{x_2 x_3} \wedge \ldots \wedge \varepsilon_{x_{d-1} x_d}$, 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 15 in Figure 2). 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 (not shown in the algorithms presented in this paper). 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 expressions 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 [12]. 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 [18], 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 [18]. $\oplus$ 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 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 $s \in R - \{r\}$, $r \not\sqsubseteq s$. 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:

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

3. Basic Policy-Based Routing Algorithms

The notation used in the algorithms presented in this paper is summarized in Table 1. In addition, the maxi-

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 1. Notation.

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 2. Operations on Data Structures [1].

mum 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 $a_{max} = \max\{|\ A(i)\ |\ |\ i \in N\}$. Table 2 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 1. In this structure, a

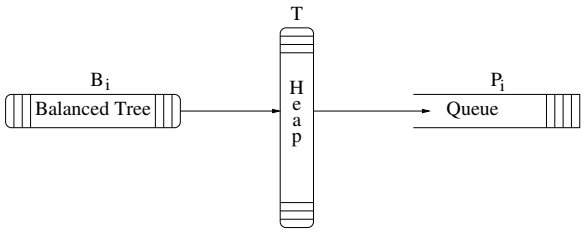

Figure 1. Basic Data Structures

```
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
```

Figure 2. General-Policy-Based Dijkstra.

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 "relaxed" routes for its neighbors are added 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 in $P \cup B_*$ 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 the smallest known-maximal route is moved to P.

Figure 2 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 $O(nW^2A^2)$.

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 (not shown here due to space constraints), the permanent routes can be summarized by a summary traffic expression (formed by the disjunction of permanent route path expressions) 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.

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

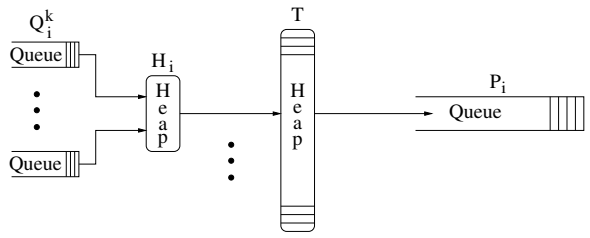

Figure 3. Enhanced Data Structures

Based on this insight, the data structure shown in Figure 3 can be used to improve the performance of the algorithms presented in Section 3. 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.

Figure 11 presents the enhanced version of the TD-QoS-Dijkstra algorithm. Similar to the basic algorithms, the correctness of these algorithms is based on the invariants and constraints presented in Section 3. Specifically, as detailed in the comments from that section, constraints 1 and 2 are maintained by the DeleteTMin() and AddCandidate() functions, and, based on this, the Dijkstra iteration over the n^{th} best route in the main body of the algorithm maintains the invariants.

The runtime complexity of the TD-QoS-Dijkstra algorithm 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 4 and 5 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 [21]). For each test the worst case measurements are graphed. The metrics were generated using the "Cost 2" scheme from [17] 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.

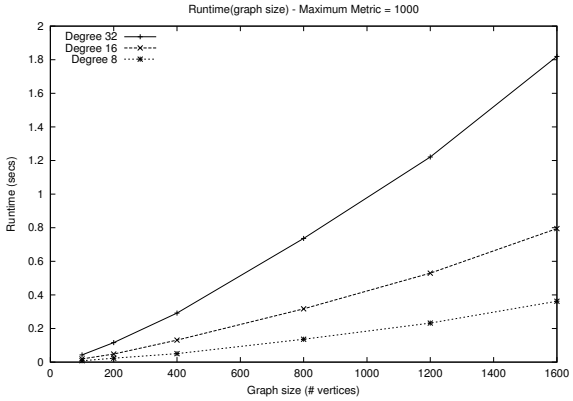

Figure 4. Enhanced Runtime(Size)

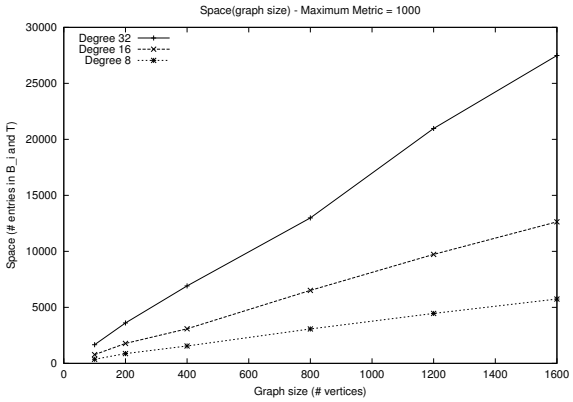

Figure 5. Enhanced Space(Size)

Tests were run for performance (both runtime and space)

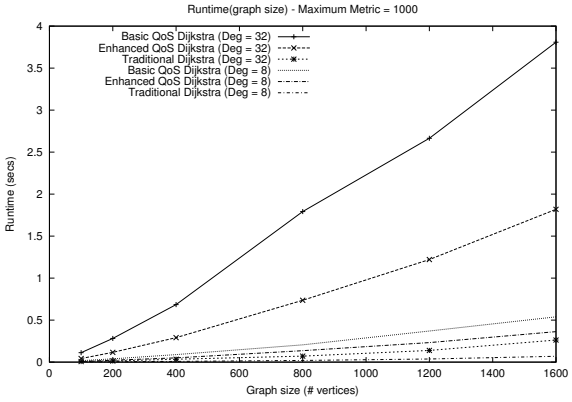

Figure 6. Compare Runtime(Size)

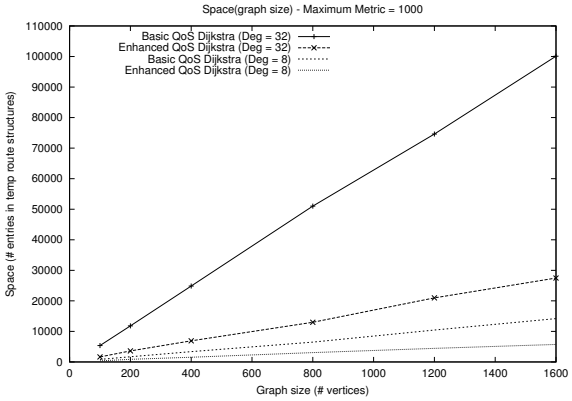

Figure 7. Compare Space(Size)

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 4 and 5 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 2 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 30,000 entries (consuming less than 10MB of memory) with similar growth rates. Figures 6 and 7 compare the performance of the basic QoS (not presented in this paper), enhanced QoS, and traditional (single path) Dijkstra algorithms.

5. Hop-by-Hop Policy-Based Routing

The policy-based routing algorithms presented in Sections 3 and 4 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 (e.g., MPLS [10]) as a generalized forwarding mechanism that replaces IP addresses as the names for network attachment points in the route binding 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. 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 [19] 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 [8] 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. Our solution 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 8, label-swap forwarding 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

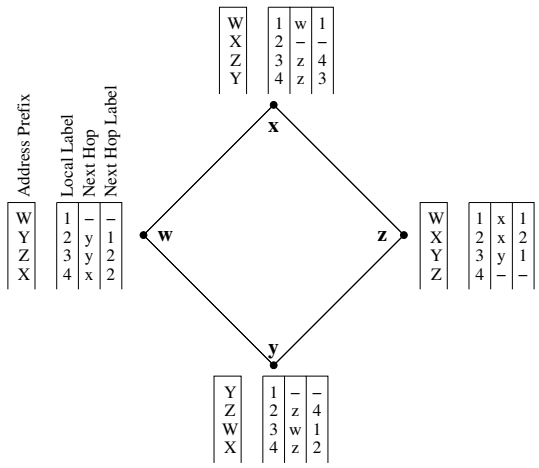

Figure 8. Label-Swapping with Addresses

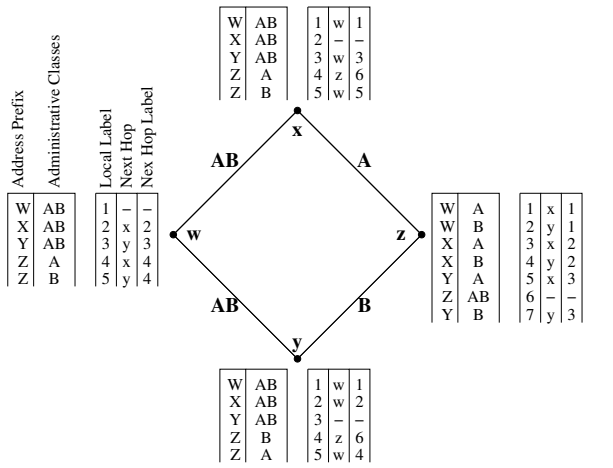

Figure 9. Label-Swapping with Policies

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.

For example, 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. Figure 9 shows the traffic classification and forwarding state for a small network with four nodes and the two administrative classes A and B. 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, hopefully, 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 [11], 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 CLI and network

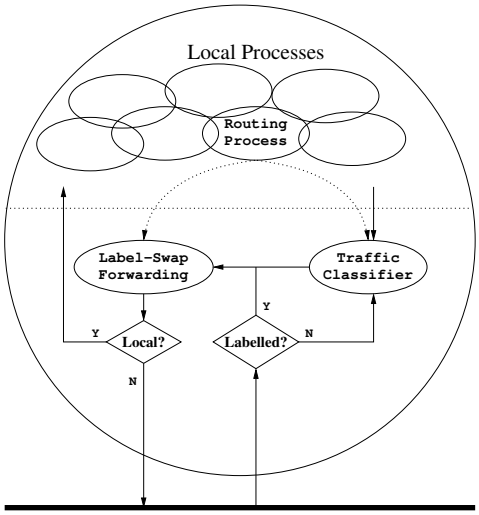

Figure 10. Traffic Flow in Policy-Enabled Router

management traffic), this effectively means a traffic classification component must be present in each router. As illustrated in Figure 10, 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.

6. Conclusions

This paper presents a family of routing algorithms 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, and a set-based model was identified for efficiently implementing restricted but useful traffic engineering policies. Lastly, a forwarding architecture was defined that efficiently implements multiple paths per destination required for hop-by-hop policy-based routing using label-swap-based forwarding.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows – Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] D. O. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for Traffic Engineering Over MPLS. RFC2702, Sept. 1999.

[3] B. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC1633, July 1994.

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

[5] V. G. Cerf. The Catenet Model for Internetworking. IEN 48, July 1978.

[6] V. G. Cerf and E. Cain. The DoD Internet Architecture Model. *Computer Networks*, 7:307–318, 1983.

[7] V. G. Cerf and R. E. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, COM-22(5):637–648, May 1974.

[8] G. P. Chandranmenon and G. Varghese. Trading Packet Headers for Packet Processing. *IEEE ACM Transactions on Networking*, 4(2):141–152, Oct. 1995. 1995.

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

[10] B. Davie and Y. Rekhter. *MPLS: Technology and Applications*. Morgan Kaufmann, 2000.

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

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

[13] J. M. Jaffe. Algorithms for Finding Paths with Multiple Constraints. *Networks*, 14(1):95–116, 1984.

[14] W. C. Lee, M. G. Hluchyi, and P. A. Humblet. Routing Subject to Quality of Service Constraints in Integrated Communication Networks. *IEEE Network*, 9(4):46–55, Aug. 1995.

[15] Q. Ma and P. Steenkiste. Quality-of-Service Routing for Traffic with Performance Guarantees. In *Proceedings 4th International IFIP Workshop on QoS*. IFIP, May 1997.

[16] P. V. Mieghem, H. D. Neve, and F. Kuipers. Hop-by-hop quality of service routing. *Computer Networks*, 37:407–423, Nov. 2001.

[17] S. Siachalou and L. Georgiadis. Efficient QoS Routing. In *Proceedigns of INFOCOM'03*. IEEE, Apr. 2003.

[18] J. L. Sobrinho. Algebra and Algorithms for QoS Path Computation and Hop-by-Hop Routing in the Internet. *IEEE/ACM Transactions on Networking*, 10(4):541–550, Aug. 2002.

[19] G. Swallow. MPLS Advantages for Traffic Engineering. *IEEE Communications Magazine*, 37(12):54–57, Dec. 1999.

[20] Z. Wang and J. Crowcroft. Quality-of-Service Routing for Supporting Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, pages 1228–1234, Sept. 1996.

[21] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings INFOCOM '96*. IEEE, 1996.

```
algorithm TD-QoS-Dijkstra
    begin
1   Push(<s, s, 0̄>, P_s);
2   for each {(s, j) ∈ A(s)}
      begin
3     Push(<j, s, ω_sj>, Q^s_j);
4     Insert(<j, s, ω_sj>, H_j);
5     Insert(<j, s, ω_sj>, T);
      end;
6   while (|T| > 0)
      begin
7     <i, p, ω> ← Min(T);
8     Push(<i, p, ω>, P_i);
9     DeleteTMin();
10    for each {(i, j) ∈ A(i)}
        begin
11      ω_i ← ω ⊕ ω_ij;
12      AddCandidate(<j, i, ω_i>);
        end
      end
    end

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

function QoS-AddCandidate(<i, p, ω_i>)
    // 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_i) ⪯ (and therefore ⋦)
    //       all routes in Q*_i queues.
    begin
1   if (ω_i ⊑ Tail(P_i).ω) then return;
2   if (|H_i| = 0)
      then begin
3       Push(<i, p, ω_i>, Q^p_i);
4       Insert(<i, p, ω_i>, H_i);
5       Insert(<i, p, ω_i>, T);
6       return;
        end
7   <i, k, ω_m> ← Min(H_i);
8   if (ω_m ⪯ ω_i)
      then
9       if (ω_i ⊑ ω_m)
10        then return;
11        else begin // ((ω_i ⋢ ω_m) ∧ (ω_m ⪯ ω_i))
12          if (|Q^p_i| = 0)
13            then Insert(<i, p, ω_i>, H_i)
14            else if (ω_i ⊑ Tail(Q^p_i).ω)
15              then return;
16          Push(<i, p, ω_i>, Q^p_i);
            end
      else // ω_i ≺ ω_m; since ω_i ≻ Min(H^p_i), it must be
           // true that |Q^p_i| = 0.
17      if (ω_i ⋣ ω_m)
          then begin
18          Push(<i, p, ω_i>, Q^p_i);
19          Insert(<i, p, ω_i>, H_i);
            // Following replaces <i, k, ω_m>.
20          DecreaseKey(<i, p, ω_i>, T_i);
            end
          else begin // (ω_i ⊒ ω_m)
21          Push(<i, p, ω_i>, Q^p_i);
            // Following replaces <i, k, ω_m>.
22          DecreaseKey(<i, p, ω_i>, H^k_i);
23          DecreaseKey(<i, p, ω_i>, T_i);
24          Pop(Q^k_i);
25          if (|Q^k_i| > 0)
26            then Insert(Head(Q^k_i), H_i);
            end
    end
```

Figure 11. Enhanced QoS Dijkstra.