
Computational Optimization and Applications, 29, 13–48, 2004

Increasing Internet Capacity Using Local Search*

BERNARD FORTZ fortz@poms.ucl.ac.be
Institut d'Administration et de Gestion, Université Catholique de Louvain, Louvain-la-Neuve, Belgium

MIKKEL THORUP mthorup@research.att.com
AT&T Labs-Research, Shannon Laboratory, Florham Park, NJ 07932, USA

Received April 22, 2002; Revised May 29, 2003

Abstract. Open Shortest Path First (OSPF) is one of the most commonly used intra-domain internet routing protocol. Traffic flow is routed along shortest paths, splitting flow evenly at nodes where several outgoing links are on shortest paths to the destination. The weights of the links, and thereby the shortest path routes, can be changed by the network operator. The weights could be set proportional to the physical lengths of the links, but often the main goal is to avoid congestion, i.e. overloading of links, and the standard heuristic recommended by Cisco (a major router vendor) is to make the weight of a link inversely proportional to its capacity.

We study the problem of optimizing OSPF weights for a given a set of projected demands so as to avoid congestion. We show this problem is NP-hard, even for approximation, and propose a local search heuristic to solve it. We also provide worst-case results about the performance of OSPF routing vs. an optimal multi-commodity flow routing. Our numerical experiments compare the results obtained with our local search heuristic to the optimal multi-commodity flow routing, as well as simple and commonly used heuristics for setting the weights. Experiments were done with a proposed next-generation AT&T WorldNet backbone as well as synthetic internetworks.

Keywords: traffic engineering, shortest path routing, local search

1. Introduction

Provisioning an Internet Service Provider (ISP) backbone network for intra-domain IP traffic is a big challenge, particularly due to rapid growth of the network and user demands. At times, the network topology and capacity may seem insufficient to meet the current demands. At the same time, there is mounting pressure for ISPs to provide Quality of Service (QoS) in terms of Service Level Agreements (SLAs) with customers, with loose guarantees on delay, loss, and throughput. All of these issues point to the importance of *traffic engineering*, making more efficient use of existing network resources by tailoring routes to the prevailing traffic.

*A preliminary short version of this paper appeared under the title "Internet Traffic Engineering by Optimizing OSPF Weights," in *Proc. IEEE INFOCOM 2000—The Conference on Computer Communications*, pp. 519–528.

1.1. *The general routing problem*

Optimizing the use of existing network resources can be seen as a general routing problem defined as follows. We are given a directed network $G = (N, A)$ with a capacity c_a for each $a \in A$, and a demand matrix D that, for each pair $(s, t) \in N \times N$, tells the demand $D(s, t)$ in traffic flow between s and t. We sometimes refer to the non-zero entries of D as the *demands*. The set of arcs leaving a node u is denoted by $\delta^+(u) := \{(u, v) : (u, v) \in A\}$ while the set of arcs entering a node u is denoted by $\delta^-(u) := \{(v, u) : (v, u) \in A\}$.

With each arc $a \in A$, we associate a cost function $\Phi_a(l_a)$ of the load l_a, depending on how close the load is to the capacity c_a. We assume in the following that Φ_a is an strictly increasing and convex function. Our formal objective is to distribute the demanded flow so as to minimize the sum

$$\Phi = \sum_{a \in A} \Phi_a(l_a)$$

of the resulting costs over all arcs. Usually, Φ_a increases rapidly as loads exceeds capacities, and our objective typically implies that we keep the max-utilization $\max_{a \in A} l_a/c_a$ below 1, or at least below 1.1, if at all possible.

In this general routing problem, there are no limitations to how we can distribute the flow between the paths. With each pair $(s, t) \in N \times N$ and each arc $a \in A$, we associate a variable $f_a^{(s,t)}$ telling how much of the traffic flow from s to t goes over a. Moreover, for each arc $a \in A$, variable l_a represents the total load on arc a, i.e. the sum of the flows going over a. With these notation, the problem can be formulated as the following multi-commodity flow problem.

$$\min \quad \Phi = \sum_{a \in A} \Phi_a(l_a)$$

subject to

$$\sum_{a \in \delta^+(u)} f_a^{(s,t)} - \sum_{a \in \delta^-(u)} f_a^{(s,t)} = \begin{cases} D(s,t) & \text{if } u = s, \\ -D(s,t) & \text{if } u = t, \quad u, s, t \in N, \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

$$l_a = \sum_{(s,t) \in N \times N} f_a^{(s,t)} \quad a \in A, \tag{2}$$

$$f_a^{(s,t)} \geq 0 \quad a \in A;\ s, t \in N. \tag{3}$$

Constraints (1) are flow conservation constraints that ensure the desired traffic flow is routed from s to t, and constraints (2) define the load on each arc.

As Φ is a convex objective function and all constraints are linear, this problem can be solved optimally in polynomial time. We denote by Φ_{OPT} the optimal solution of this general routing problem.

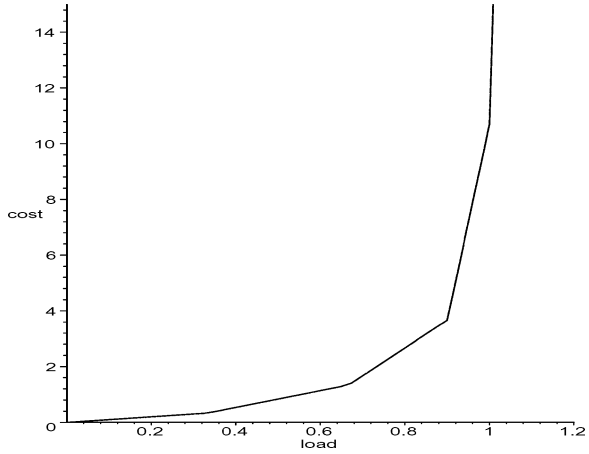

Figure 1. Arc cost $\Phi_a(l_a)$ as a function of load l_a for arc capacity $c_a = 1$.

In our experiments, Φ_a are piecewise linear functions, with $\Phi_a(0) = 0$ and derivative

$$\Phi'_a(l) = \begin{cases} 1 & \text{for} \quad 0 \le l/c_a < 1/3, \\ 3 & \text{for} \quad 1/3 \le l/c_a < 2/3, \\ 10 & \text{for} \quad 2/3 \le l/c_a < 9/10, \\ 70 & \text{for} \quad 9/10 \le l/c_a < 1, \\ 500 & \text{for} \quad 1 \le l/c_a < 11/10, \\ 5000 & \text{for } 11/10 \le l/c_a < \infty. \end{cases} \tag{4}$$

The function Φ_a is illustrated in figure 1, and can be viewed as modeling retransmission delays caused by packet losses. Generally it is cheap to send flow over an arc with a small utilization l_a/c_a. The cost increases progressively as the utilization approaches 100%, and explodes when we go above 110%. With this cost function, the general routing problem becomes a linear program.

The objective function was chosen on the basis of discussions on costs with people close to the AT&T IP backbone. Motivations on our choice for the objective function and the different model assumptions are discussed in detail in [17], and, for a closely related application, in [18]. A description of the general infrastructure behind this kind of traffic engineering is given in [15].

1.2. The OSPF weight setting problem

The most commonly used intra-domain internet routing protocols today are shortest path protocols such as Open Shortest Path First (OSPF) [28]. OSPF does not support a free distribution of flow between source and destination as defined above in the general routing

problem. In OSPF, the network operator assigns a weight w_a to each link $a \in A$, and shortest paths from each router to each destination are computed using these weights as lengths of the links. In practice, link weights are integer encoded on 16 bits, therefore they can take any value between 1 and 65,535. In each router, represented by a node of the graph, the next link on all shortest paths to all possible destinations is stored in a table. A flow arriving at the router is sent to its destination by splitting the flow between the links that are on the shortest paths to the destination. The splitting is done using pseudo-random methods leading to an approximately even splitting. For simplicity, we assume that the splitting is exactly even (for AT&T's WorldNet this simplification leads to reasonable estimates).

More precisely, given a set of weights $(w_a)_{a \in A}$, the length of a path is then the sum of its arc weights, and we have the extra condition that all flow leaving a node aimed at a given destination is evenly spread over the first arcs on shortest paths to that destination. Therefore, for each source-destination pair $(s, t) \in N \times N$ and for each arc $a \in \delta^+(u)$ for some node $u \in N$, we have that $f_a^{(s,t)} = 0$ if a is not on a shortest path from s to t, and that $f_a^{(s,t)} = f_{a'}^{(s,t)}$ if both $a \in \delta^+(u)$ and $a' \in \delta^+(u)$ are on shortest paths from s to t. Note that the routing of the demands is completely determined by the shortest paths which in turn are determined by the weights we assign to the arcs.

The quality of OSPF routing depends highly on the choice of weights. Nevertheless, as recommended by Cisco (a major router vendor) [11], these are often just set inversely proportional to the capacities of the links, without taking any knowledge of the demand into account.

The *OSPF weight setting problem* is to set the weights so as to minimize the cost of the resulting routing. In the remainder of the paper, we denote by Φ_{OptOSPF} the optimal cost with OSPF routing.

1.2.1. The MPLS alternative. We note that the emerging Multi-Protocol Label Switching (MPLS) [31] allows the network operator to specify arbitrary paths for each source destination pair, plus the splitting of packets between these paths. MPLS can thus implement the optimal solution of the general routing problem [2, 27]. Before going bankrupt, Global Crossing [37] attempted such use of MPLS though using a simple non-optimal greedy heuristic to select paths between sources and destinations so as to satisfy a conservative estimate of the demands.

It is, however, not clear how much traffic MPLS will take over from traditional shortest path protocols like OSPF. For example, shortest path weights do form a nice compact description of the routing between any source and destination. Also, OSPF is well-defined for any set of link-failures, routing along shortest paths in the remaining graph.

Choosing which routing protocol to use for what traffic is, of course, a much more complex discussion than indicated above. However, from the perspective of congestion, as defined in Section 1.1, we do provide feedback on the potential gains in switching from a tried and true shortest path protocols such as OSPF to the new MPLS protocol.

Indeed, in a recent white paper, [33, p. 7] refers to the announcement in [17] of the work of the current paper as one of the reasons for not deploying MPLS in their IP backbone.

1.3. Our results

The general question studied in this paper is: *Can a sufficiently clever weight setting make OSPF routing perform nearly as well as optimal general/MPLS routing?*

Our first answer is negative: for any positive integer n, we construct an instance of the routing problem on $\approx n^3$ nodes where *any* OSPF routing has its average flow on arcs with utilization $\Omega(n)$ times higher than the max-utilization in an optimal general solution. With our concrete objective function, this demonstrates a gap of a factor approaching 5000 between the cost of the optimal general routing and the cost of the optimal OSPF routing.

The next natural question is: how well does OSPF routing perform on real networks. In particular we wanted to answer this question for a proposed next-generation AT&T WorldNet backbone with projected demands. In addition, we studied synthetic internetworks, generated as suggested by Calvert et al. [8] and Zegura et al. [39]. Finding a perfect answer is hard in the sense that it is NP-hard to find even an approximately optimal setting of the OSPF weights for an arbitrary network. More precisely, in our experimental setting, we show that it is NP-hard to optimize the OSPF weight setting with respect to our cost function within a factor 3.1 from optimality.

Therefore, instead of exactly solving the optimization problem, we resorted to a local search heuristic, not guaranteed to find optimal solutions. Very surprisingly, it turned out that for the proposed AT&T WorldNet backbone, as well as for the synthetic networks, the heuristic found weight settings making OSPF routing perform within a few percent of the optimal general routing. Thus for the proposed AT&T WorldNet backbone with our projected demands, and with our concrete objective function, there would be no substantial traffic engineering gain in switching from the existing well-tested and understood robust OSPF technology to the new MPLS alternative.

We also compared our local search heuristic with standard heuristics, such as weights inversely proportional to the capacities or proportional to the physical distances, and found that, for the same network and capacities, we could support a 50–115% increase in the demands, both with respect to our concrete cost function and, simultaneously, with respect to keeping the max-utilization below 100%.

1.4. Technical contributions

Our local search heuristic is original in its use of hash tables both to avoid cycling and for search diversification. Using hash tables to avoid cycling in local search was already proposed by Woodruff and Zemel [36], but our approach differs in the sense that we eliminate completely all solutions already encountered, and we do not need the concept of solution attributes. More precisely, our approach is closely related to strict tabu search [3], where each solution is mapped to a hash value (that can be seen as the unique solution attribute), and we do not allow the same value twice during the complete search. Using the same mechanism to obtain diversification is, to our knowledge, a new idea that has not been tested before.

Our local search heuristic is also original in its use of more advanced dynamic graph algorithms like those of Ramalingam and Reps [29]. Computing the OSPF routing resulting

from a given set of weights turned out to be the computational bottleneck, as many different solutions are evaluated during a neighborhood exploration. However, our neighborhood structure allows only a few local changes in the weights. We therefore developed efficient algorithms to update the routing and recompute the cost of a solution when a few weights are changed. These speed-ups are critical for the local search to reach a good solution within reasonable time bounds, as they typically reduce the computing time by a factor of 15.

1.5. Related work

To the best of our knowledge, there has not been any previous work dealing with the even splitting in OSPF routing. In previous work on optimizing OSPF weights [6, 26, 30], they have either chosen weights so as to avoid multiple shortest paths from source to destination, or applied a protocol for breaking ties, thus selecting a unique shortest path for each source-destination pair.

Unlike AT&T, some operators, e.g. France Telecom [4, 5], require the routing to follow unique shortest paths in order to better understand what happens in the network. Therefore, the set of weights must be such that the shortest path between any pair of nodes is unique, which restrict the set of feasible solutions. Our problem with even splitting is therefore a relaxation of this problem. It follows immediately that the worst case result of Section 2 also holds when unique shortest paths are required. The NP-hardness of finding an approximate solution to this problem was shown by Roughan and Thorup [32], with a proof similar to the complexity proofs presented here. Even if our local search heuristic cannot be applied directly to this problem, Roughan and Thorup reported some encouraging preliminary results using our heuristic modified such as to penalize splitting.

Besides the difference in model, where we deal with even splitting, our approach is also technically different. First of all, we deal with much larger networks. In more detail [30] present a local search like ours, but using only a single descent. In contrast, we consider non-improving moves and hence we have to deal with the problem of avoiding cycles. The largest network considered in [30] has 16 nodes and 18 links. Also Bley et al. [6] use a single descent local search. The largest network they consider has 45 links, but it should be mentioned that they simultaneously deal with the problem of designing the network, whereas we only try to optimize the weights for a given network. Finally, Lin and Wang [26] present a completely different approach based on Lagrangian relaxation, and consider networks with up to 26 nodes. In our experiments, the proposed AT&T WorldNet backbone has 90 nodes and 274 links, and our synthetic networks have up to 100 nodes and 503 links. The scale of our experiments makes speed an issue, motivating our innovative use of dynamic graph algorithms in local search.

1.6. Contents

In Section 2, a family of networks is constructed, demonstrating a large gap between OSPF and multi-commodity flow routing, and in Section 3, we show that the problem of optimizing OSPF weights is NP-hard. In Section 4 we present our local search algorithm, guiding the search with hash tables. In Section 5 we show how to speed-up the calculations using

dynamic graph algorithms. In Section 6, we report the numerical experiments, and we conclude in Section 7 with some comments about the results obtained.

2. Gap between general routing and OSPF

In Lemma 1, we show that the gap between optimal general routing and optimal OSPF routing can approach the ratio between the marginal cost of sending flow above capacity of a link and the marginal cost of sending flow on an empty link. For the particular cost function defined by (4), this gap can be close to 5000.

Our proof is based on a construction where OSPF leads to very bad congestion for any natural definition of congestion. In fact, the construction provides a negative example for any type of routing where if the flow from a node to a given destination splits, it splits evenly between the outgoing links that it uses.

Lemma 1. *For any positive integer n, there is a network G_n with $O(n^3)$ nodes so that any OSPF routing has its average flow on arcs with utilization $\Omega(n)$ times higher than the max-utilization in an optimal general solution.*

Suppose our total routing cost is $\Psi = \sum_{a \in A} \Psi_a(l_a)$ where Ψ_a is increasing and convex, and that there exist positive constants $\alpha, \beta, \gamma, \delta$, with $\delta < 1$, such that

$$\begin{aligned} &\Psi_a(l) \le \alpha l && \text{if } l \le \delta c_a, \\ &\Psi_a(l) \ge \gamma c_a + \beta(l - c_a) && \text{if } l > c_a. \end{aligned}$$

Then the optimal general routing solution on G_n approaches being $\frac{\beta}{\alpha}$ times better than the optimal OSPF routing as $n \to \infty$.

Proof: For $n \ge 1$, let $G_n = (N_n, A_n)$ be the graph defined by node set

$$N_n = \{s, t\} \cup \{v_i : 1 \le i \le n\} \cup \left\{w_j^i : 1 \le i \le n,\ i+1 \le j \le n^2 - 1\right\}$$

and arc set

$$\begin{aligned} A_n = {} & \{(s, v_1)\} \cup \{(v_i, v_{i+1}) : 1 \le i \le n-1\} \cup \left\{\left(v_i, w_{i+1}^i\right) : 1 \le i \le n-1\right\} \\ & \cup \left\{\left(w_j^i, w_{j+1}^i\right) : 1 \le i \le n,\ i+1 \le j \le n^2 - 2\right\} \cup \left\{\left(w_{n^2-1}^i, t\right) : 1 \le i \le n\right\}. \end{aligned}$$

The capacities of the arcs are

$$c_a = \begin{cases} n/\delta & \text{if } a \in \delta^-(v_i) \text{ for some } i,\ 1 \le i \le n, \\ 1/\delta & \text{otherwise.} \end{cases}$$

Finally, we have a single demand of size n with source s and destination t.

The graph G_n has $O(n^3)$ nodes and arcs, and is illustrated in figure 2 for $n = 5$ with high capacity arcs represented by thick lines.

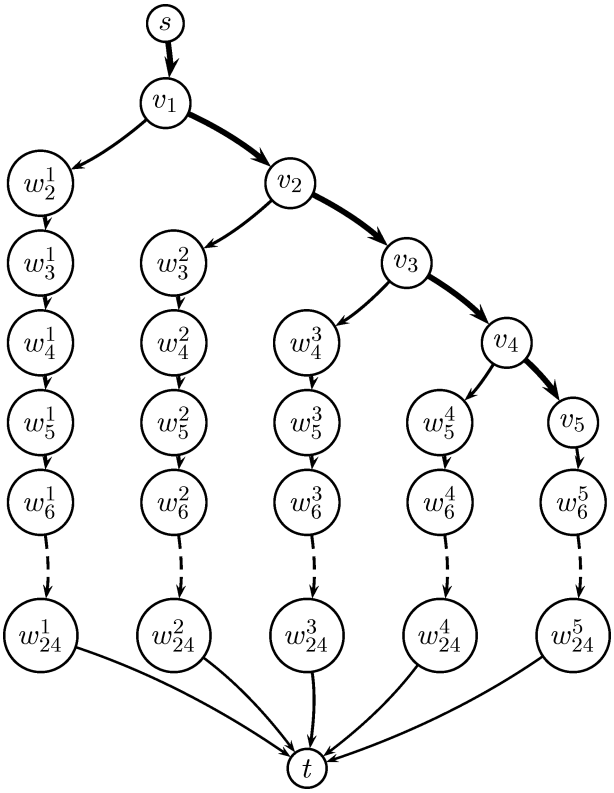

Figure 2. G_n for $n = 5$.

By our construction, there are exactly n paths from s to t, each with n^2 links. These paths are:

$$(s, v_1), \ldots, (v_{i-1}, v_i), \left(v_i, w_{i+1}^i\right), \left(w_{i+1}^i, w_{i+2}^i\right), \ldots, \left(w_{n^2-1}^i, t\right) \quad 1 \le i \le n.$$

The optimal solution of the general routing model is obviously given by sending one unit of flow along each path, meaning that no arc gets more flow than δ times its capacity. Therefore, as each unit of flow from s to t has to follow a path of length n^2, the optimal cost Ψ_{OPT}^n is less than or equal to $n^3\alpha$.

In the OSPF model, we can freely decide which paths we use, but because of the even splitting, the first path used gets half the flow, i.e. $n/2$ units, the second gets $n/4$ units, and so on. Asymptotically this means that almost all the flow goes along arcs with load a factor $\Omega(\delta n)$ above their capacity, and since all paths use at least $n^2 - n$ arcs of small capacity, the optimal OSPF cost Ψ_{OptOSPF}^n satisfies

$$\Psi_{\text{OptOSPF}}^n \ge (1 - o(1))\beta n^3.$$

We conclude that the ratio of the OSPF cost over the optimal cost is such that $\frac{\Psi^n_{\text{OptOSPF}}}{\Psi^n_{\text{OPT}}} \geq (1 - o(1))\frac{\beta}{\alpha} \to \frac{\beta}{\alpha}$ as $n \to \infty$. □

Corollary 2. *For the cost function defined by* (4)*, there is a family of networks G_n so that the optimal general routing approaches being* 5000 *times better than the optimal OSPF routing as $n \to \infty$.*

Proof: In order to get into the hypothesis of Lemma 1, it is sufficient to see that an equivalent problem is obtained by defining new capacities $\overline{c}(a) = 1.1c_a$, for all $a \in A$, and by using the modified cost functions defined by $\Phi_a(0) = 0$ and

$$\Phi'_a(x) = \begin{cases} 1 & \text{for} \quad 0 \leq l/\bar{c}(a) < 10/33, \\ 3 & \text{for } 10/33 \leq l/\bar{c}(a) < 20/33, \\ 10 & \text{for } 20/33 \leq l/\bar{c}(a) < 9/11, \\ 70 & \text{for } 9/11 \leq l/\bar{c}(a) < 10/11, \\ 500 & \text{for } 10/11 \leq l/\bar{c}(a) < 1, \\ 5000 & \text{for} \quad 1 \leq l/\bar{c}(a) < \infty. \end{cases}$$

This scaling arises from the fact that the highest marginal cost in our cost function is obtained for a flow above 110% of the capacity.

The scaled cost function satisfies the hypothesis of Lemma 1 with $\alpha = 1$, $\beta = 5000$, $\delta = 10/33$ and

$$\gamma = \frac{10}{33}1 + \frac{10}{33}3 + \frac{7}{33}10 + \frac{1}{11}70 + \frac{1}{11}500 = \frac{1820}{33}.$$

□

3. Complexity

In this section, we will formally present our hardness results, stating that it is NP-hard to find even an approximately optimal setting of OSPF weights. All proofs are deferred to Appendix A. The hardness will be presented, not only for our concrete cost function Φ as defined in Section 1, but also for much more general classes of cost functions. Also, we will prove hardness of approximation with respect to max-utilization, which is another natural measure for the quality of routing. In all cases, our inapproximability factors are much worse than the results we will later obtain experimentally.

For our cost function from Section 1, we have

Theorem 3. *It is NP-hard to optimize the OSPF weight setting with respect to the cost function defined by* (4) *within a factor* 3.1 *from optimality.*

Theorem 3 is derived from the following hardness result for a large class of cost functions:

Theorem 4. *Let α and β be fixed constants. Suppose our total routing cost is $\Psi = \sum_{a \in A} \Psi_a(l_a)$ where*

$$\begin{aligned} &\Psi_a(l) \le \alpha l && \text{if } l \le c_a, \\ &\Psi_a(l) \ge \alpha c_a + \beta(l - c_a) && \text{if } l > c_a. \end{aligned}$$

Then, if $\beta \ge 52\alpha$, it is NP-hard to optimize the OSPF weight setting with respect to $\Psi = \sum_{a \in A} \Psi_a(l_a)$ within a factor $0.72 + \beta/38\alpha > 2$ from optimality.

The proof of Theorem 4 is deferred to Appendix A, but here we verify that Theorem 4 does generalize Theorem 3, as claimed.

Proof that Theorem 4 implies Theorem 3: In order to get into the hypothesis of Theorem 4, we can use the same construction as in the proof of Corollary 2, scaling the capacities and the values of the breakpoints in the objective function by a factor 1.1.

The cost function in this construction satisfies the hypothesis of Theorem 4 with

$$\alpha = \frac{10}{33}1 + \frac{10}{33}3 + \frac{7}{33}10 + \frac{1}{11}70 + \frac{1}{11}500 = \frac{1820}{33}$$

and $\beta = 5000$, leading to an inapproximability factor greater than 3.1. □

Finally, we state the hardness for max-utilization.

Theorem 5. *It is NP-hard to optimize the max-utilization in OSPF routing within a factor $<3/2$.*

4. OSPF weight setting using local search

In OSPF routing, for each arc $a \in A$, we have to choose a weight w_a. These weights uniquely determine the shortest paths, the routing of traffic flow, the loads on the arcs, and finally, the value of the cost function Φ. In the rest of this section, we present a local search heuristic to determine weights w_a, $a \in A$, in order to minimize Φ.

Suppose that we want to minimize a function f over a set X of feasible solutions. Local search techniques are iterative procedures that for each iteration define a neighborhood $\mathcal{N}(x) \subseteq X$ for the current solution $x \in X$, and then choose the next solution x' from this neighborhood. Often we want the neighbor $x' \in \mathcal{N}(x)$ to improve on f in the sense that $f(x') < f(x)$.

Differences between local search heuristics arise essentially from the definition of the neighborhood, the way it is explored, and the choice of the next solution from the neighborhood. Descent methods consider the entire neighborhood, select an improving neighbor and stop when a local minimum is found. Meta-heuristics such as Tabu search or simulated annealing allow non-improving moves while applying restrictions to the neighborhood to avoid cycling. An extensive survey of local search and its applications can be found in [1].

In the remainder of this section, we first describe the neighborhood structure we apply to solve the weight setting problem. Second, using hash tables, we address the problem of avoiding cycling. These hash tables are also used to avoid repetitions in the neighborhood exploration. While the neighborhood search aims at intensifying the search in a promising region, it is often of great practical importance to search a new region when the neighborhood search fails to improve the best solution for a while. These techniques are called search diversification and are addressed at the end of the section.

As a very first step, we choose a maximal weight $w_{\max} = 20$, and then restrict our attention to weights in $W := \{1, \ldots, w_{\max}\}$. The idea behind using small weights is that we increase the chance of even splitting due to multiple shortest paths from a node to some destination.

4.1. Neighborhood structure

A solution of the weight setting problem is completely characterized by its vector $w = (w_a)_{a \in A}$ of weights. We define a neighbor $w' \in \mathcal{N}(w)$ of w by one of the two following operations applied to w.

Single weight change. This simple modification consists in changing a single weight in w. We define a neighbor w' of w for each arc $a \in A$ and for each possible weight $t \in W \backslash \{w_a\}$ by setting $w'(a) = t$ and $w'(b) = w_b$ for all $b \neq a$.

Evenly balancing flows. Assuming that the cost function Φ_a for an arc $a \in A$ is increasing and convex, meaning that we want to avoid highly congested arcs, we want to split the flow as evenly as possible between different arcs.

More precisely, consider a demand node t such that $\sum_{s \in N} D(s, t) > 0$ and some part of the demand going to t goes through a given node u. Intuitively, we would like OSPF routing to split the flow to t going through u evenly along arcs leaving u. This is the case if every arc in $\delta^+(u)$ belongs to a shortest path from u to t. More precisely, if $\delta^+(u) = \{a_i : 1 \leq i \leq p\}$, and if P_i is one of the shortest paths from the head of a_i to t, for $i = 1, \ldots, p$, as illustrated in figure 3, then we want to set w' such that

$$w'_{a_i} + w'(P_i) = w'_{a_j} + w'(P_j) \quad 1 \leq i, j \leq p,$$

where $w'(P_i)$ denotes the sum of the weights of the arcs belonging to P_i. A simple way of achieving this goal is to set

$$w'(a) = \begin{cases} w^* - w(P_i) & \text{if } a = a_i, \text{ for } i = 1, \ldots, p, \\ w_a & \text{otherwise.} \end{cases}$$

where $w^* = 1 + \max_{i=1,\ldots,p}\{w(P_i)\}$.

A drawback of this approach is that an arc that does not belong to one of the shortest paths from u to t may already be congested, and the modifications of weights we propose

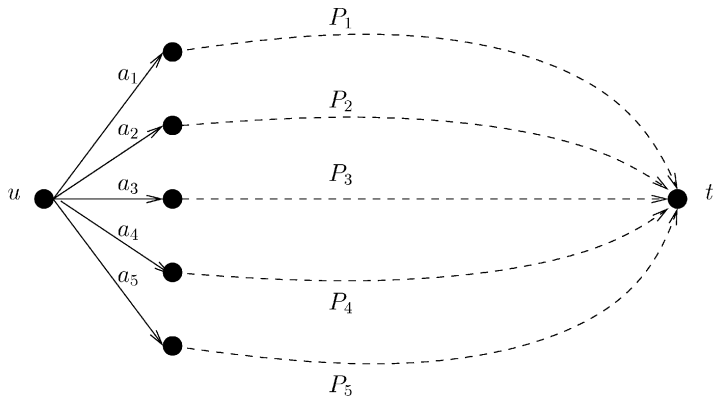

Figure 3. The second type of move tries to make all paths form u to t of equal length.

will send more flow on this congested arc, an obviously undesirable feature. We therefore decided to choose at random a threshold ratio θ between 0.25 and 1, and we only modify weights for arcs in the maximal subset B of $\delta^+(u)$ such that

$$w_{a_i} + w(P_i) \leq w_{a_j} + w(P_j) \quad \forall\, i : a_i \in B,\ j : a_j \notin B,$$
$$l_a^w \leq \theta\, c_a \quad \forall\, a \in B,$$

where l_a^w denotes the load on a resulting from weight vector w. The last relation implies that the utilization of an arc $a \in B$ resulting from the weight vector w is less than or equal to θ, so that we can avoid sending flow on already congested arcs. In this way, flow leaving u towards t can only change for arcs in B, and choosing θ at random allows to diversify the search.

This choice of B does not ensure that weights remain below $w_{\max}$. This can be done by adding the condition $\max_{i:a_i \in B} w(P_i) - \min_{i:a_i \in B} w(P_i) \leq w_{\max}$ when choosing B.

The first neighborhood is made of $w_{\max}|A|$ elements while the second one is made of $|N|^2$ elements. As detailed in the rest of this section, we will not explore the full neighborhoods. Of all neighbors explored, our next solution will be the best according to our cost function.

4.2. Guiding the search with hash tables

The simplest local search heuristic is the descent method that, at each iteration, selects the best element in the neighborhood and stops when this element does not improve the objective function. This approach leads to a local minimum that is often far from the optimal solution of the problem, and heuristics allowing non-improving moves have been considered. Unfortunately, non-improving moves can lead to cycling, and one must provide mechanisms to avoid it. Tabu search algorithms [20–23], for example, make use of a Tabu

list that records some attributes of solutions encountered during the recent iterations and forbids any solution having the same attributes.

As our neighborhood structure for the weight setting problem is quite complex, efficiently designing Tabu attributes and search parameters such as the length of the Tabu list would have required a lot of work. We instead developed a search strategy that completely avoids cycling without the need to store complex solution attributes. This approach, called *Strict Tabu Search* was studied by Battiti and Tecchiolli [3]. They tested different techniques for implementing it, namely the *Reverse Elimination Method* [22], hashing [36] and digital trees [25].

It turns out that the Reverse Elimination Method is much more expensive both in memory and computing requirements. Furthermore, digital trees are best suited for binary variables, and would be too expensive memory-wise for encoding our solutions as $|A|$-dimensional 16-bit integer vectors. We therefore resorted to hashing. Hash functions compress solutions into single integer values, sending different solutions into the same integer with small probability. To implement Strict Tabu Search, we use a boolean table T to record if a value produced by the hash function $h(\ )$ has been encountered. At the beginning of the algorithm, all entries in T are set to false. If w is the solution produced at a given iteration, we set $T(h(w))$ to true, and, while searching the neighborhood, we reject any solution w' such that $T(h(w'))$ is true. Checking that a solution has been encountered is therefore performed in constant time.

As pointed out by Carlton and Barnes [9], we risk collisions whenever $w' \neq w$ but $h(w') = h(w)$. They show that collisions arise with a high probability after just a few iterations. In our case, roughly 10% of all solutions are eliminated because of collisions (we select 5000 solutions out of 2^{16}). In classical tabu search implementations, an aspiration rule is used that accepts a move that improves the best solution so far, even if it is tabu (or has a forbidden hash value in our case). But computing the cost of a solution is the computational bottleneck of our approach, so we decided not to use this rule. As the eliminated solutions are essentially random, and since our approach is already highly randomized, this is not critical. Moreover, for the weight setting problem, many different solutions (i.e. weight vectors) lead to the same arc loads and total cost. Therefore, even if a good solution is killed by a collision, there is a high chance that some other solution leading to the same distribution of flows will survive.

The hash functions we use are based on developments from [10, 13, 34], and have the property that the probability that any two different weight settings get the same hash value is the same as if the functions were truly random, which is not the case for functions used by Woodruff and Zemel [36]. Suppose the weights are represented as m-bit integers, where $m \geq \log_2 w_{\max}$, and we want to map them to n-bit integers. In our case, we had $m = 5$ and $n = 16$. This value of n was chosen because storing hash values as 16-bit integers is a natural choice and leads to an array T of 2^{16} booleans, a manageable size.

With each arc $a \in A$, we associate a random $(m + n - 1)$-bit integer p_a. We then define the hash function $h_a(w_a)$ of a single weight w_a by considering $p_a w_a$, which is a $(2m + n - 1)$-bit integer, and taking the n-bit integer obtained by dropping the m highest bits and the $m - 1$ lowest bits of $p_a w_a$, as illustrated in figure 4.

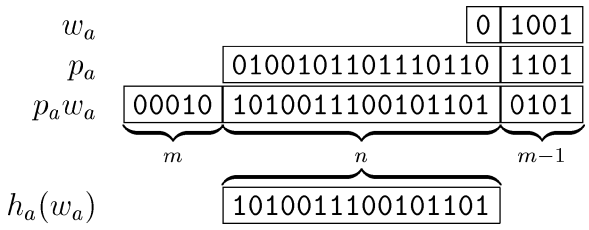

Figure 4. Example of hash function for a single weight with $w_a = 9,\ p_a = 309101,\ h_a(w_a) = 42797$.

The hash value of w is then defined by

$$h(w) = \bigoplus_{a \in A} h_a(w_a),$$

where $\oplus$ denotes the bitwise XOR operation. A big advantage of using the XOR operation is that it allows a fast update of the hash value when a single weight is changed. More precisely, if w' is equal to w except for a given a for which $w'_a \neq w_a$, then $h(w')$ can be computed as

$$h(w') = h(w) \oplus h_a(w_a) \oplus h_a(w'_a).$$

We note that the basic idea of hash vectors coordinate-wise so that one can locally update the overall hash value when a single coordinate changes goes back to chess playing computers [40]. The hash function that we use was, however, not known at the time.

In our local search we performed 5000 iterations. The corresponding 5001 solutions occupied at most a fraction $5001/2^{16} < 1/10$ of the potential hash values. The number of iterations performed was chosen after observing that a very good solution was usually obtained in the first 3000 iterations. Allowing for 5000 iterations is sufficient to get a robust behavior of the algorithm, in the sense that the deviation in the quality of the solutions obtained over several runs is negligible.

4.3. Speeding up neighborhood evaluation

Due to our complex neighborhood structure for evenly balancing flows, it turned out that several moves (of the second type) often lead to the same weight setting. A simple example is if we have a node w with a single incoming arc (v, w). Then from any node $u \neq v, w$, we will do exactly the same balancing with destination v as with destination w.

For efficiency, we would like to avoid evaluation of these equivalent moves. Again, hash tables are a useful tool to achieve this goal: inside a neighborhood exploration, we define a secondary hash table used to store the encountered weight settings as above, and we do not evaluate moves leading to a hash value already met. Note that looking for a hash value in the table takes a single operation compared to an evaluation of the cost of a solution that takes $O(|N|^2)$ operations.

The hash table is generally reset at the end of each iteration, since we want to avoid repetitions inside a single iteration only. An exception to this rule is when the move does not improve the solution, as detailed in the next section. As we explain in this next section, the size of the secondary hash table must be small compared to the primary one. In our experiments, its size S is 20 times the number of arcs in the network. Instead of computing a secondary hash value from scratch, we take the 16-bit hash value $h(w)$ already computed for the primary hash table, and compute the hash value for the secondary table as $h(w) \bmod S$. For AT&T WorldNet's proposed backbone with 274 links, this gives a secondary size of $5,480$ whereas the primary size is $2^{16} = 65,536$.

The neighborhood structure we use has also the drawback that the number of neighbors of a given solution is very large, and exploring the neighborhood completely may be too time consuming. To avoid this drawback, we only evaluate a randomly selected set of neighbors.

We start by evaluating 20% of the neighborhood. For the single weight change, this is done by selecting 20% of the set of possible weights for each arc, while for the second neighborhood, we reject any move with probability 80% before evaluating it. Each time the current solution is improved, we divide the size of the sampling by 3, while we multiply it by 2 each time the current solution is not improved. Moreover, we enforce sampling at least 1% of the neighborhood.

4.4. Diversification

Another important ingredient for local search efficiency is diversification. The aim of diversification is to escape from regions that have been explored for a while without any improvement, and to search regions as yet unexplored.

In our particular case, many weight settings can lead to the same distribution of flows. Therefore, we observed that when a local minimum is reached, it has many neighbors having the same cost, leading to long series of iterations with the same cost value. To escape from these "long valleys" of the search space, the secondary hash table is again used.

This table should be reset at the end of each iteration to avoid repetitions inside a single iteration only. However, if the neighborhood exploration does not lead to a solution better than the current one, we do not reset the table. If this happens for several iterations, more and more collisions occur and more potentially good solutions are excluded, forcing the algorithm to escape from the region currently explored. Our aim here is to diversify the search by eventually rejecting good solutions and accepting bad moves, to reach other regions of the search space (as opposed to an intensification of the search when the objective solution decreases strictly).

For these collisions to appear at a reasonable rate, the size of the secondary hash table must be small compared to the primary one. This approach for diversification is useful to avoid regions with a lot of local minima with the same cost, but is not sufficient to completely escape from one region and go to a possibly more attractive one. Therefore, each time the best solution found is not improved for 300 iterations, we randomly perturb the current solution in order to explore a new region from the search space. The perturbation consists of adding a randomly selected perturbation, uniformly chosen between -2 and $+2$, to 10% of the weights.

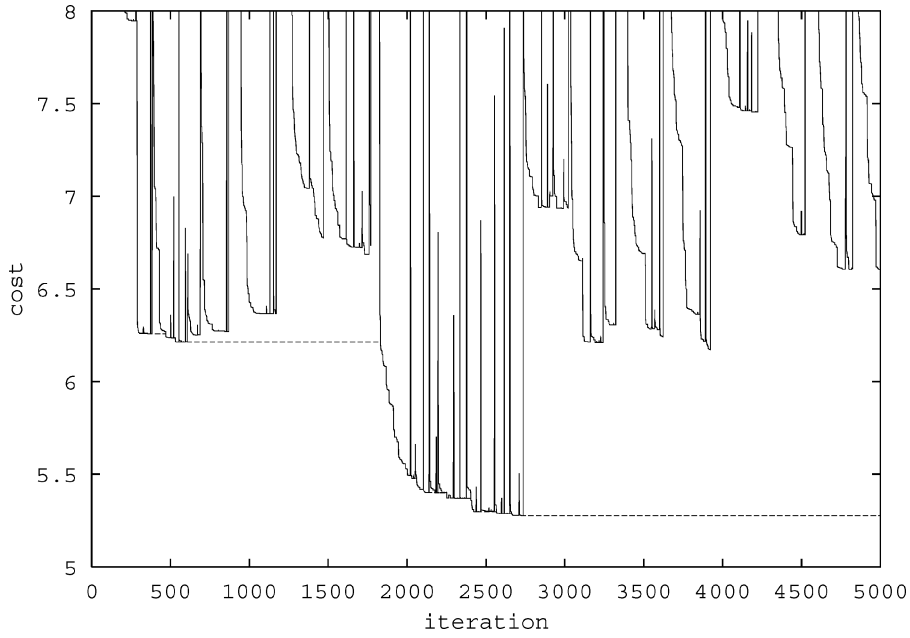

Figure 5. Evolution of the local search over iterations, AT&T WorldNet backbone,

We present in figure 5 the evolution of the cost at each iteration of the search for one run of an instance for AT&T WorldNet backbone. The dotted line represents the cost of the best solution found so far. The impact of our diversification scheme is the high variation in cost that can be observed in the figure, but as the algorithm only allows a few diversifying moves, we can observe a quick descent to a good region after a "bad move".

As we see in figure 5, it is quite random how good a region we get to after a diversifying bad move. Hence it is not particularly surprising that best region and solution is found half way through the 5000 iterations. As mentioned in Section 4.2, it was common in our experiments that a very good, if not best, solution was found within the first 3000 moves.

The impact of hash tables on rejected moves is also illustrated in Table 1, in which we measured, for different network sizes, the total number of moves, and the number of moves rejected with each one of the two hash tables. The number of moves rejected because of collisions in the primary table remains quite small, while the proportion of collisions in the secondary table is quite important, because diversification plays an important role in our heuristic.

5. Cost evaluation

In this section, we first show how to evaluate our cost function for the static case of a network with a specified weight setting. Computing this cost function from scratch is unfortunately

Table 1. Impact of the hash tables.

Network size		Total number of moves	Rejected moves	
Nodes	Arcs		Primary	Secondary
90	274	14,941,588	1,788,103 (12.0%)	11,070,388 (74.1%)
50	148	5,504,970	363,821 (6.6%)	3,739,470 (67.9%)
50	212	9,200,783	713,887 (7.8%)	6,552,817 (71.2%)
100	280	17,531,422	1,392,694 (7.9%)	13,412,086 (76.5%)
100	360	10,670,683	637,434 (6.0%)	6,375,143 (59.7%)
100	403	10,165,707	595,688 (5.9%)	5,572,911 (54.8%)
100	391	9,869,432	558,478 (5.7%)	5,023,112 (50.9%)

too time consuming for our local search, so afterwards, we show how to reuse computations, exploiting the fact that there are only few weight changes between a current solution and any solution in its neighborhood.

5.1. The static case

We are given a directed graph $G = (N, A)$ with arc capacities $\{c_a\}_{a\in A}$, a demand matrix D, and weights $\{w_a\}_{a\in A}$. For the instances considered, the graph is sparse with $|A| = O(|N|)$. Moreover, in the weighted graph the maximal distance between any two nodes is $O(|N|)$.

We want to compute our cost function Φ. The basic problem is to compute the loads resulting from the weight setting. We consider one destination t at a time, and compute the total flow from all sources $s \in N$ to t. This gives rise to a certain partial load $l_a^t = \sum_{s\in N} f_a^{(s,t)}$ for each arc. Having done the above computation for each destination t, we can compute the load l_a on arc a as $\sum_{t\in N} l_a^t$.

To compute the flow to t, our first step is to use Dijkstra's algorithm to compute all distances to t (normally Dijkstra's algorithm computes the distances away from some source, but we can just apply such an implementation of Dijkstra's algorithm to the graph obtained by reversing the orientation of all arcs in G). Having computed the distance d_u^t to t for each node u, we compute the set A^t of arcs on shortest paths to t, that is,

$$A^t = \left\{(u, v) \in A : d_u^t - d_v^t = w_{(u,v)}\right\}$$

For each node u, let $\delta_t^+(u) = \{v \in N : (u, v) \in A^t\}$ denote the set of arcs leaving u. The out degree of u in A^t is therefore $|\delta_t^+(u)|$.

Observation 6. For all $(v, w) \in A^t$,

$$l_{(v,w)}^t = \frac{1}{|\delta_t^+(v)|}\left(D(v, t) + \sum_{(u,v)\in A^t} l_{(u,v)}^t\right).$$

Using Observation 6, we can now compute all the loads $l^t_{(v,w)}$ as follows. The nodes $v \in N$ are visited in order of decreasing distance d^t_v to t. When visiting a node v, we first set $l = \frac{1}{|\delta^+_t(v)|}(D(v,t) + \sum_{(u,v)\in A^t} l^t_{(u,v)})$. Second we set $l^t_{(v,w)} = l$ for each $(v, w) \in A^t$.

To see that the above algorithm works correctly, we assume, inductively, that we have dealt correctly with all nodes visited before v. Since every arc (u, v) entering v in A^t stems from a node u strictly further from t, we know u has been visited previously. Hence, by induction, the load $l^t_{(u,v)}$ is correct. Since all the incoming arc loads are correct when v is visited, the outgoing arc loads are computed correctly by Observation 6.

Using bucketing for the priority queue in Dijkstra's algorithm, the computation for each destination takes $O(|A|) = O(|N|)$ time, and hence our total time bound is $O(|N|^2)$.

5.2. *The dynamic case*

In our local search we want to evaluate the cost of many different weight settings, and these evaluations are a bottleneck for our computation. To save time, we try to exploit the fact that when we evaluate consecutive weight settings, typically only a few arc weights change. Thus it makes sense to try to be lazy and not recompute everything from scratch, but to reuse as much as possible. With respect to shortest paths, this idea is already well studied [29], and we can apply their algorithm directly. Their basic result is that, for the recomputation, we only spend time proportional to the number of arcs incident to nodes u whose distance d^t_u to t changes. In our experiments there were typically only very few changes, so the gain was substantial—in the order of factor 15 for a 100 node graph. Similar positive experiences with this laziness have been reported in [19].

The set of changed distances immediately gives us a set of "update" arcs to be added to or deleted from A^t. We now present a lazy method for finding the changes of loads. We operate with a set M of "critical" nodes. Initially, M consists of all nodes with an incoming or outgoing update arc. We repeat the following until M is empty: First, we take the node $v \in M$ which maximizes the updated distance d^t_v and remove v from M. Second, set $l = \frac{1}{|\delta^+_t(v)|}(D(v,t) + \sum_{(u,v)\in A^t} l^t_{(u,v)})$. Finally, for each (v, w) in the updated A^t, if (v, w) is new or $l \neq l^t_{(v,w)}$, set $l^t_{(v,w)} = l$ and add w to M.

To see that the above suffices, first note that the nodes visited are considered in order of decreasing distances. This follows because we always take the node at the maximal distance and because when we add a new node w to M, it is closer to t than the currently visited node v. Consequently, our dynamic algorithm behaves exactly as our static algorithm except that it does not treat nodes not in M. However, all nodes whose incoming or outgoing arc set changes, or whose incoming arc loads change are put in M, so if a node is skipped, we know that the loads around it would be exactly the same as in the previous evaluation.

In order to measure the impact of the dynamic update of the cost on the performance of our algorithm, we performed 1000 iterations on various networks with different demand sets. Our experiments showed that dynamic updates make the algorithm from 5 up to 25 times faster, with an average of 15 times faster. This improvement was critical to us, as we typically ran the local search for a bit more than 1 hour in order to get good solutions.

In Table 2, we report some of these results. Both for static computations and dynamic updates, we measured the fraction of the computing time that was spend to compute shortest

Table 2. Impact of the dynamic update on computing times.

Network size		CPU time (seconds)						
		Static computations			Dynamic computations			
Nodes	Arcs	Total	Sh. paths	Flows	Total	Sh. paths	Flows	Gain (%)
90	274	411	243 (59%)	146 (36%)	29	11 (38%)	12 (41%)	93
50	148	570	263 (46%)	289 (51%)	66	21 (32%)	38 (58%)	88
50	212	912	408 (45%)	476 (52%)	87	26 (30%)	53 (61%)	90
100	280	5787	2628 (45%)	2960 (51%)	393	139 (35%)	228 (58%)	93
100	360	8440	3806 (45%)	4326 (51%)	679	188 (28%)	447 (66%)	92
100	403	7717	3496 (45%)	3898 (51%)	660	158 (24%)	447 (68%)	91
100	391	7931	3633 (46%)	3986 (50%)	775	180 (23%)	533 (69%)	90

paths and the resulting flows. The table clearly shows that these computations take more than 90% of the computing times in all cases. Our proposed techniques for dynamically updating the flows are as important for the efficiency of our code as the already known shortest paths updating techniques, even if the improvement is slightly better for shortest paths.

6. Numerical experiments

We present here our results obtained with a proposed AT&T WorldNet backbone as well as synthetic internetworks.

The cost function used in the experiments is defined by (4). As discussed in [18], a problem in the current formulation of Φ is that it does not provide a universal measure of congestion. Independently of the network topology and demand matrix, it is natural to require that the maximum utilization remains below 1. Similarly, we would like a universal cut-off value for our cost function (4), independent of the network topology and demand matrix. To achieve this, we use the normalized cost function

$$\Phi^* = \Phi/\Psi$$

where Ψ is the cost we would have had if all flow was sent along hop-count shortest paths and the capacities matched the loads. When capacity matches load on a link a, we pay $\Phi_a(c_a)/c_a$ per unit of flow on a. With our cost function, for any arc a,

$$\Phi_a(c_a)/c_a = 1/3 + 3 \cdot 1/3 + 10 \cdot 7/30 + 70 \cdot 1/10 = 10\frac{2}{3}.$$

This cost per unit of flow when load matches capacity is thus constant over all arcs and is equal to $10\frac{2}{3}$. It follows that if $\Delta(s, t)$ is the hop-count distance between s and t,

$$\Psi = \sum_{(s,t)\in N\times N} \left(10\frac{2}{3} \cdot D[s, t] \cdot \Delta(s, t)\right).$$

Note that for a given network and demand matrix, the division by a constant Ψ doesn't affect which routings are considered good. With this scaling, if $\Phi^* \geq 1$, then the routing is as bad as if all flows were along hop-count shortest paths with loads matching the capacities. The same cost can, of course, also stem from some loads going above capacity and others going below, or by flows following longer detours via less utilized arcs. Nevertheless, it is natural to say that a routing *congests* a network if $\Phi^* > 1$.

Besides comparing our local search heuristic (HeurOSPF) with the general optimum (OPT), we compared it with OSPF routing with "oblivious" weight settings based on properties of the arc alone but ignoring the rest of the network. The oblivious heuristics are

- InvCapOSPF, setting the weight of an arc inversely proportional to its capacity as recommended by Cisco [11]. In each of our experiments, we set the weight of each link by dividing the maximal capacity over all links by the capacity of the link, and rounding so to get integer weights,
- UnitOSPF, setting all arc weights to 1,
- L2OSPF, setting the weight proportional to its physical Euclidean distance (L_2 norm), and
- RandomOSPF, just choosing the weights randomly.

We also applied a simple descent method (DescentOSPF) to our neighborhood, i.e. we did not use hash tables and diversification, and stopped at the first local minimum encountered.

Our local search heuristic starts with randomly generated weights and performs 5000 iterations. The average CPU time of HeurOSPF over all experiments was 1370 seconds, for a maximum of 6221 seconds for the largest network. In comparison, DescentOSPF took on average 882 CPU seconds for a maximum of 4378 seconds. The gain in quality and robustness obtained with HeurOSPF completely justifies the increased CPU time.

The initial solution for HeurOSPF and DescentOSPF was weights chosen for RandomOSPF, so the initial cost of our local search is that of RandomOSPF. We also performed some experiments taking UnitOSPF as initial setting, without any improvement in the quality of the solutions (convergence was even slower).

All tests were performed on an Intel Pentium 3 processor cadenced at 1.13 Ghz, running under RedHat Linux 7.3. The linear programs for OPT were solved by calling CPLEX version 7.1 via AMPL.

The results for the AT&T WorldNet backbone with different scalings of the projected demand matrix are presented in Table 3. Here, by scaling a demand matrix, we mean that we multiply all entries with a common number. Each algorithm was run once independently for each scaling. Hence OPT, RandomOSPF, and HeurOSPF use different routings for different scalings whereas UnitOSPF and InvCapOSPF always use the same routing. In each entry we have the normalized cost Φ^*. The normalized cost is followed by the max-utilization in parenthesis. The bold line in the table corresponds to the original non-scaled demand. In order to show the robustness of our method, we also performed 10 runs starting with different random weight settings. The results of these experiments are reported in Table 4.

For all the OSPF schemes, the normalized cost and max-utilization are calculated for the same weight setting and routing. However, for OPT, the optimal normalized cost and the optimal max-utilization are computed independently with different routing. We do not

Table 3. Results for proposed AT&T WorldNet backbone with 90 nodes and 274 arcs.

	OPT		Unit		InvCap		L2		Random		Descent		Heur	
Total demand	Cost	Max-util	Cost	Max-util	Cost	Max-util	Cost	Max-util	Cost	Max-util	Cost	Max-util	Cost	Max-util
3919	0.09	0.10	0.09	0.16	0.09	0.16	0.11	0.25	0.11	0.20	0.10	0.14	0.10	0.16
7837	0.09	0.20	0.09	0.33	0.09	0.32	0.11	0.49	0.12	0.79	0.10	0.35	0.10	0.32
11756	0.09	0.31	0.10	0.49	0.10	0.48	0.12	0.74	0.14	0.78	0.10	0.38	0.10	0.41
15675	0.10	0.41	0.11	0.65	0.11	0.64	0.14	0.98	14.94	1.54	0.10	0.56	0.10	0.50
19593	**0.11**	**0.51**	**0.13**	**0.81**	**0.13**	**0.80**	**1.40**	**1.23**	**1.73**	**1.28**	**0.12**	**0.62**	**0.12**	**0.62**
23512	0.13	0.61	0.17	0.98	0.17	0.95	5.49	1.47	3.70	1.24	0.13	0.71	0.13	0.71
27430	0.14	0.72	0.69	1.14	0.48	1.11	8.96	1.72	37.22	2.34	0.16	0.87	0.15	0.83
31349	0.16	0.82	3.19	1.30	2.95	1.27	11.95	1.96	18.64	1.64	0.18	0.94	0.17	0.94
35268	0.20	0.92	6.55	1.46	7.40	1.43	17.52	2.21	22.74	3.13	0.23	1.01	0.20	0.95
39186	0.27	1.02	11.88	1.63	13.64	1.59	24.69	2.45	46.87	2.72	0.42	1.18	0.29	1.06
43105	0.72	1.12	18.89	1.79	20.48	1.75	31.87	2.70	64.76	2.97	3.03	1.30	0.84	1.15
47024	2.29	1.23	26.87	1.95	28.15	1.91	40.79	2.94	76.75	3.56	3.90	1.49	2.54	1.41

Table 4. Mean (μ) and standard deviation (σ) for 10 runs on proposed AT&T WorldNet backbone.

	Cost			Max-utilization		
Total demand	Random $\mu\,(\sigma)$	Descent $\mu\,(\sigma)$	Heur $\mu\,(\sigma)$	Random $\mu\,(\sigma)$	Descent $\mu\,(\sigma)$	Heur $\mu\,(\sigma)$
3919	0.110 (0.000)	0.091 (0.003)	0.091 (0.003)	0.255 (0.052)	0.186 (0.047)	0.167 (0.011)
7837	0.270 (0.391)	0.091 (0.003)	0.091 (0.003)	0.662 (0.304)	0.322 (0.013)	0.315 (0.007)
11756	0.745 (1.363)	0.100 (0.000)	0.091 (0.003)	0.985 (0.403)	0.381 (0.043)	0.359 (0.019)
15675	3.800 (5.172)	0.100 (0.000)	0.100 (0.000)	1.332 (0.512)	0.521 (0.021)	0.526 (0.018)
19593	5.519 (5.049)	0.113 (0.005)	0.111 (0.003)	1.478 (0.481)	0.658 (0.019)	0.625 (0.016)
23512	13.206 (8.384)	0.130 (0.000)	0.130 (0.000)	1.754 (0.533)	0.718 (0.017)	0.710 (0.000)
27430	21.626 (11.068)	0.152 (0.004)	0.150 (0.000)	1.875 (0.269)	0.856 (0.023)	0.821 (0.003)
31349	37.286 (17.154)	0.179 (0.003)	0.170 (0.000)	2.988 (1.939)	0.933 (0.016)	0.940 (0.000)
35268	47.034 (16.721)	0.228 (0.013)	0.203 (0.005)	2.573 (0.555)	1.011 (0.055)	0.950 (0.000)
39186	68.977 (26.803)	0.401 (0.073)	0.291 (0.003)	2.964 (0.656)	1.118 (0.066)	1.057 (0.007)
43105	83.035 (28.215)	2.440 (2.442)	0.871 (0.059)	2.925 (0.433)	1.346 (0.212)	1.170 (0.046)
47024	96.216 (31.983)	4.225 (0.640)	2.539 (0.048)	4.936 (3.126)	1.543 (0.171)	1.403 (0.022)

expect any general routing to be able to get the optimal normalized cost and max-utilization simultaneously. The results are also depicted graphically in figure 6. The first graph shows the normalized cost and the horizontal line shows our threshold of 1 for regarding the network as congested. The second graph shows the max-utilization. As L2OSPF and RandomOSPF

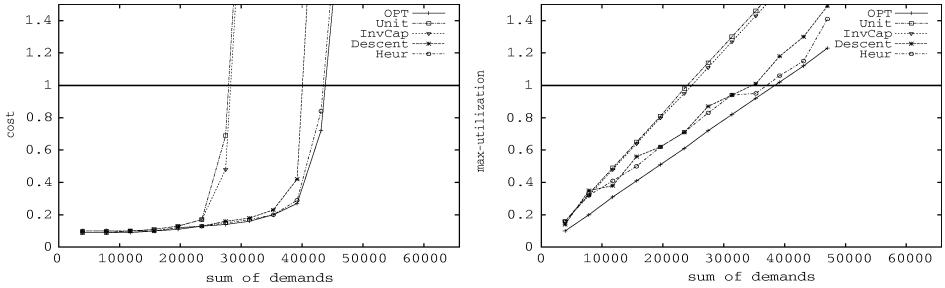

Figure 6. AT&T's proposed backbone with 90 nodes and 274 arcs.

clearly perform very badly (as could be expected), we do not depict the associated results in the graphs.

We also generated three flavors of synthetic graphs.

2-level hierarchical graphs: these graphs were produced using the generator GT-ITM [38], based on a model of [8, 39]. Arcs are divided in two classes: *local access* arcs and *long distance* arcs. Arc capacities are equal to 200 for local access arcs and to 1000 for long distance arcs.

Purely random graphs: the probability of having an arc between two nodes is given by a constant parameter used to control the density of the graph. All arc capacities are set to 1000.

Waxman graphs: nodes are uniformly distributed points in a unit square and the probability of having an arc between two nodes u and v is given by

$$p(i, j) = \alpha e^{\frac{-L_2(u,v)}{\beta\Delta}}$$

where α and β are parameters used to control the density of the graph, $L_2(u, v)$ is the Euclidean distance between u and v and Δ is the maximum distance between two nodes [35]. All arc capacities are set to 1000.

Note that in random and Waxman graphs, all capacities are equal, so InvCapOSPF and UnitOSPF will run identically. While random and Waxman graphs are the cleaner from a mathematical perspective, the 2-level hierarchical graphs are the most realistic known models for internetworks.

The demands are generated as follows. For each node u, we pick two random numbers $O_u, D_u \in [0, 1]$. Further, for each pair (u, v) of nodes we pick a random number $C_{(u,v)} \in [0, 1]$. The demand between u and v is

$$\alpha O_u D_v C_{(u,v)} e^{\frac{-L_2(u,v)}{2\Delta}}$$

Here α is a parameter and Δ is again the largest Euclidean distance between any pair of nodes. Above, the O_u and D_v reflect the fact that different nodes can be more or less active

senders and receivers, thus modeling hot spots on the net. Because we are multiplying three random variables, we have a quite large variation in the demands. The factor $e^{-L_2(u,v)/2\Delta}$ implies that we have relatively more demand between close pairs of nodes.

Five instances were generated and tested for each class of graphs. However, for space reasons, we present here only results for four hierarchical, one random, and one Waxman graphs. These results are presented in figures 7–12.

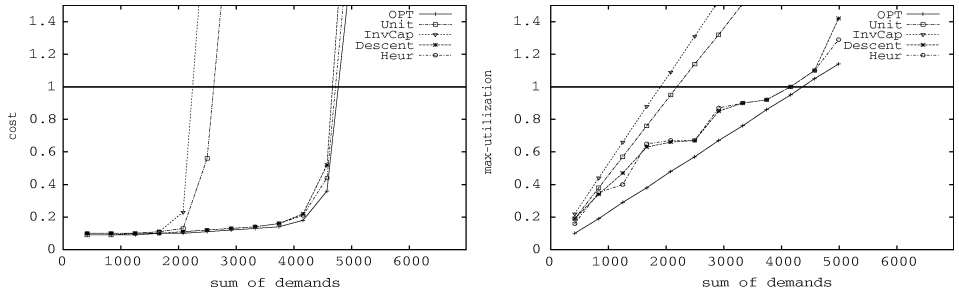

Figure 7. 2-level graph with 50 nodes and 148 arcs.

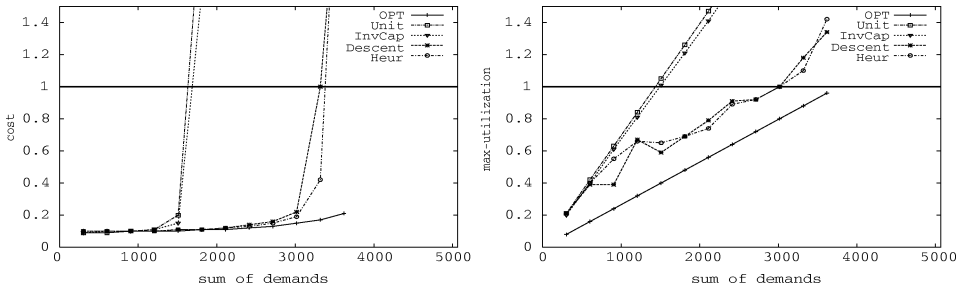

Figure 8. 2-level graph with 50 nodes and 212 arcs.

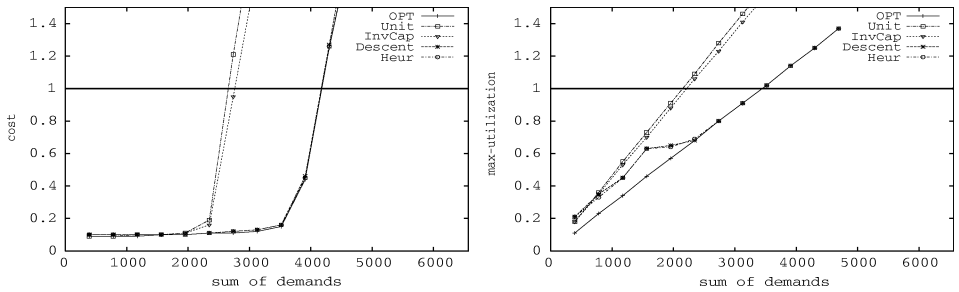

Figure 9. 2-level graph with 100 nodes and 280 arcs.

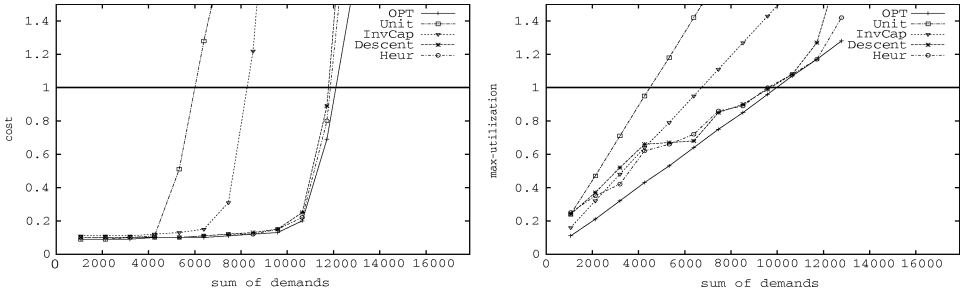

Figure 10. 2-level graph with 100 nodes and 360 arcs.

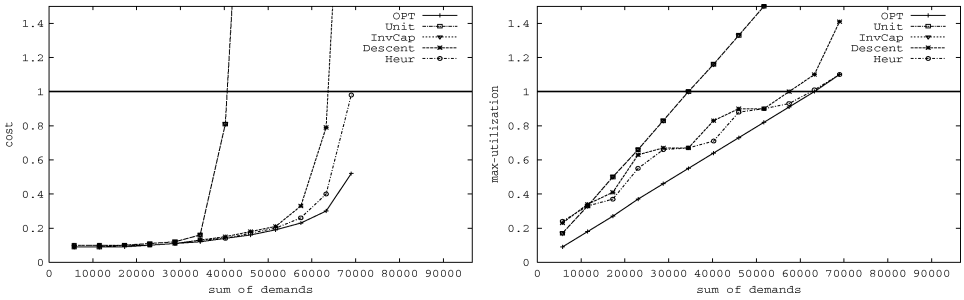

Figure 11. Random graph with 100 nodes and 403 arcs.

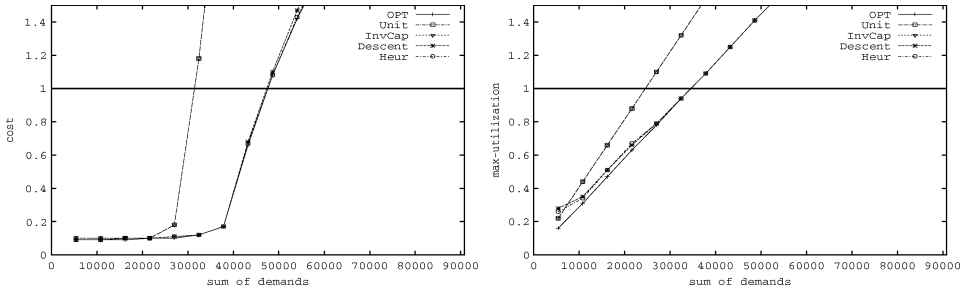

Figure 12. Waxman graph with 100 nodes and 391 arcs.

7. Discussion

We presented in this paper a local search heuristic for optimizing OSPF weights used for intra-domain internet routing. This heuristic was tested on a substantial set of instances, including a proposed AT&T WorldNet backbone with projected demands. It would have been nice to run on more real world data, but such data are typically company secrets.

If we consider the results for the AT&T WorldNet backbone with projected (non-scaled) demands, i.e. the bold line in Table 6, we observe that our heuristic, HeurOSPF, is within 1.8% from optimality. In contrast, the oblivious methods are all off by at least 15%.

Considering the general picture for the normalized costs in figures 6–12, we see that L2OSPF and RandomOSPF typically do worst. Then comes InvCapOSPF and UnitOSPF closely together, with InvCapOSPF being generally slightly better (for the random and Waxman graphs in figures 11 and 12, InvCapOSPF coincides with UnitOSPF since all links have the same capacity). Recall that InvCapOSPF is Cisco's recommendation, so it is comforting to see that it is the best of the oblivious heuristics. The clear winner of the OSPF schemes is our HeurOSPF, which is, in fact, much closer to the general optimum than to the oblivious OSPF schemes. The descent method also performs well compared to oblivious schemes, but our heuristic still produces better results. Moreover, Table 4 shows that HeurOSPF is far less sensitive to the initial solution than the DescentOSPF.

Consider the different schemes with respect to our scaled cost function Ψ^*. Note that all curves but those for RandomOSPF, start off pretty flat, and then, quite suddenly, start exploding. This pattern is somewhat similar to that in figure 1. This is not surprising since figure 1 shows the curve for a network consisting of a single arc. The reason why RandomOSPF does not follow this pattern is that the weight settings are generated randomly for each entry. The jumps of the curve for RandomOSPF in figure 9 nicely illustrate the luck impact in the weight setting. One could, of course, have run RandomOSPF several times, and used the best solution, but from the figures it is clear that even if it is erratic how badly RandomOSPF performs, it is never the best. The purpose of including RandomOSPF is to be able to compare the other weight settings with random choices. Interestingly, for any particular demand, the value of RandomOSPF is the value of the initial solution for our local search heuristic. However, the jumps of RandomOSPF are not transferred to HeurOSPF which hence seems oblivious to the quality of the initial solution.

When comparing different schemes on a given network, one can typically demonstrate huge gaps if considering a level of demand for which one but not the other has started exploding. For example, consider the AT&T graph in figure 6 and compare HeurOSPF with InvCapOSPF with demands around 3300. At that point, InvCapOSPF has left the plot, but in Table 3 we see that with demand 33378, HeurOSPF has a cost of 0.19 while InvCapOSPF has a cost of 5.18. These gaps are, however, not very informative as they are an artifact of the explosive nature of our cost function.

The most interesting comparison between the different schemes is the amount of demand they can cope with before the network gets too congested, i.e. when Φ^* gets bigger than 1. Considering the proposed AT&T WorldNet backbone in figure 6 and for the 2-level graphs in figures 7–10, we see that HeurOSPF allows us to cope with 50–115% more demand than Cisco's recommended InvCapOSPF, with an average around 70%. The maximum of 115% is achieved in figure 8. In this particular figure, L2OSPF actually does quite well, but generally InvCapOSPF is the better of the oblivious heuristics, and hence the most challenging one to compete with. Moreover, in all but figure 8, HeurOSPF is less than 2% from being able to cope with the same demands as the optimal general routing OPT. In figure 8, HeurOSPF is about 20% from OPT. Recall from Theorem 3 that it is NP-hard to approximate the optimal cost of an OSPF solution within a factor 3, and we have

no idea whether there exist OSPF solutions closer to OPT than the ones found by our heuristic.

The above picture is repeated for the random graph in figure 11, except that our HeurOSPF tends to be a little further away from OPT, though still by less than 10%. For the Waxman graph in figure 12, the most significant difference relative to the other models is that Unit/InvCapOSPF tends to get closer to OPT, with OPT only allowing for in average 30% increase in demands, thus leaving less scope for improvement. On the other hand, for the Waxman graphs, HeurOSPF nearly coincides with OPT. The random and Waxman graphs are considered much less realistic than the 2-level graphs, but we see them as supporting evidence for conjecturing that for typical internetworks, HeurOSPF will be able to supply large parts of the gain that OPT may have over InvCapOSPF, or any other of the oblivious heuristics.

If we now turn our attention to max-utilization, we get the same ordering of the schemes, with InvCapOSPF the winner among the oblivious schemes and HeurOSPF the overall best OSPF scheme. The advantage of using HeurOSPF for max-utilization is interesting in that our local search did not use max-utilization as its objective.

The step-like pattern of HeurOSPF shows the impact of the changes in Φ'_a. For example, in figure 6, we see how HeurOSPF fights to keep the max utilization below 1, in order to avoid the high penalty for getting load above capacity. Following the pattern in our analysis for the normalized cost, we can ask how much more demand we can deal with before getting max-utilization above 1, and we see that HeurOSPF again beats the oblivious schemes by at least 50% for the proposed AT&T WorldNet backbone and the 2-level graphs.

The fact that our HeurOSPF provides weight settings and routings that are simultaneously good both for our best effort type average cost function, and for the performance guarantee type measure of max-utilization indicates that the weights obtained are "universally good" and not just tuned for our particular cost function. Recall that the values for OPT are not for the same routings, and there may be no general routing getting simultaneously closer to the optimal cost and the optimal max-utilization than HeurOSPF. Anyhow, our HeurOSPF is generally so close to OPT that there is little scope for improvement.

Finally, it may be interesting to see to which extent splitting was achieved. Looking at the best solutions found by HeurOSPF, we noticed that flows for a given destination split on average 5 times.

8. Conclusion

We have presented worst-case examples showing that there are cases where even the best OSPF weight setting leads to very bad routing as compared with the best general routing. Also, we have shown that it is NP-hard to find even an approximately optimal OSPF weight setting. These negative findings are contrasted by the positive findings in our experimental work, where our weight setting heuristic produces OSPF routings that are quite close in performance to that of the best possible general routings. This indicates that the negative examples for OSPF routing are too contrived to dominate in practice.

For the proposed AT&T WorldNet backbone and for the 2-level graphs suggested in [8, 39], our OSPF weight setting heuristic further distinguished itself by producing weight

settings allowing for a 50–115% increase in demands over what is achieved with standard weight setting heuristics, such as using inverse capacity as recommended by Cisco [11]. Thus we have shown that in the context of known demands, a clever weight setting algorithm for OSPF routing is a powerful tool for increasing a network's ability to honor increasing demands, and that OSPF with clever weight setting can provide large parts of the potential gains of traffic engineering for supporting demands, even when compared with the possibilities of the much more flexible MPLS schemes.

9. Subsequent and future work

Building upon the techniques from this paper, we have developed a network management system dealing with more complex issues such as link-failures, hot spots, and predicted periodic changes in the demand matrix, say between day and night traffic [18].

Also, the work reported in this paper, which has circulated as a technical report [16], has inspired other researchers to work on the OSPF weight setting problem. Ericsson et al. [14] have applied a genetic algorithm Buriol et al. [7] have applied a memetic algorithm. Both of these papers reuse the experimental framework from this paper, which is now becoming a standard test bed for the OSPF weight setting problem. Also, Buriol et al. reused the idea of using dynamic graph algorithms to speed up their memetic algorithm. We note that none of these later approaches have reported substantially better solutions for the test instances, though the convergence is often faster.

We leave open the problem of developing exact methods for the OSPF weight setting problem.

Appendix A: NP-hardness proofs

In this appendix, we prove the NP-hardness results claimed in Section 3. The proofs are done partly in collaboration with David Johnson and Christos Papadimitriou. First we prove the hardness of minimizing the max-utilization, and second we prove hardness with respect to summation-based cost functions like Φ. The proof for max-utilization is much easier, and helps presenting some of the basic ideas needed for the summation-based cost functions.

Recall from Section 3:

Theorem 5. *It is NP-hard to optimize the max-utilization in OSPF routing within a factor* $<3/2$.

Proof of Theorem 5: We prove this result by reducing 3SAT to the problem of optimizing OSPF weight setting with respect to max-utilization. More precisely, let $S = (X, C)$ be an instance of 3SAT with variable set X and clause set C where each clause has 3 literals [12]. We construct an instance of the OSPF weight setting problem such that

- there exists a satisfiable assignment for the 3SAT instance if and only if the max-utilization in the OSPF instance is equal to 1;

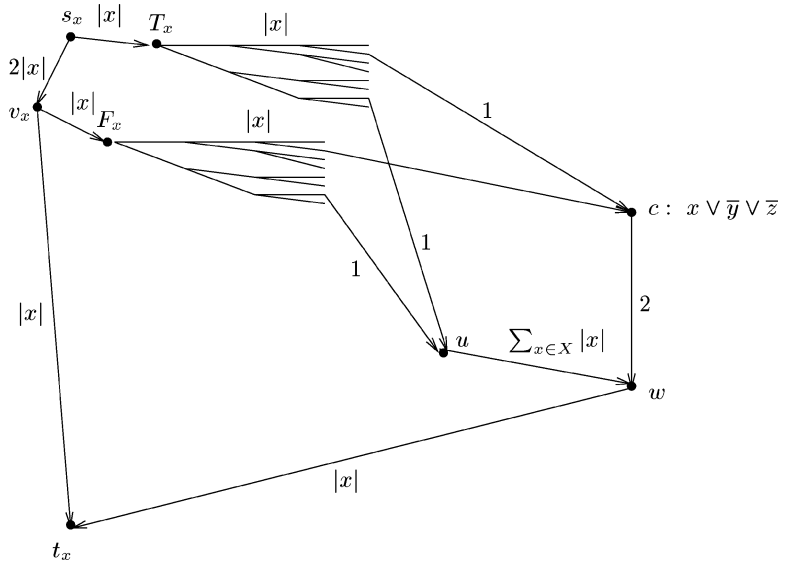

Figure 13. Reduction from 3SAT to OSPF max-utilization.

- if there is no satisfiable assignment for the 3SAT instance, the max-utilization for any weight setting is at least $3/2$.

The construction of the graph $G_S = (N_S, A_S)$ corresponding to the 3SAT instance is illustrated in figure 13, where arcs are annotated by their capacities. With each clause $c \in C$, we associate a node $c \in N_S$. All these nodes are connected by an arc of capacity 2 to a global node w.

For each variable $x \in X$, let $|x|$ denote the least power of 2 bounding both the number of negative and the number of positive occurrences of x in S. With each $x \in X$, we associate a source s_x and a destination t_x, with demand $2|x|$ between them. Furthermore, for each x, we have three nodes v_x, F_x, and T_x, and arcs (s_x, v_x), (v_x, t_x), (v_x, F_x), and (s_x, T_x). The arcs (v_x, t_x), (v_x, F_x), and (s_x, T_x) have capacity $|x|$ while (s_x, v_x) has capacity $2|x|$. Balanced binary trees with $|x|$ leaves are rooted at F_x and T_x. For each positive (negative) occurrence of x in a clause c, we have an arc from a leaf under F_x (T_x) to the node c. Each leaf can only be used for one occurrence of x in the clauses. Each leaf which is not connected to a clause is connected to a global node u, which in turn is connected to w. The arcs inside the binary trees have capacity $|x|$ while the arcs from the leaves to the clauses or to u all have capacity 1. Arc (u, w) has capacity $\sum_{x \in X} |x|$, and for all x, there is an arc from w to t_x with capacity $|x|$.

A *canonical flow* in this network is defined as a flow corresponding to an assignment of values to the variables in X. More precisely, in a canonical flow, we have $|x|$ units of flow going down (v_x, t_x) and $|x|$ units going to either T_x or F_x depending on whether x is true or false. If x is true, then the flow comes to T_x and it spreads evenly so that we get 1 unit leaving each leaf going either to a clause node c or to u, and then these $|x|$ units of flow

are sent to t_x through w. If x is false, the flow goes to F_x, then is spread in the same way as above. With this assignment of flows, for each clause c, there is 1 unit of flow through (c, w) for each literal in c not satisfied by the assignment. Thus, if the clause c is satisfied, it has at most 2 literals that are not satisfied by the assignment and the load stays within the capacity of (c, w). Otherwise, the load on (c, w) is 3 and the utilization of (c, w) goes to $3/2$. All other loads in a canonical flow stay within the capacities.

Now suppose we are given a non-canonical flow satisfying the OSPF routing condition that if the flow to a destination splits, it splits evenly. We will show that max-utilization for this non-canonical flow is at least $3/2$. It is easy to see that a non-canonical flow satisfies at least one of the following conditions for some x:

(i) (v_x, t_x) is not used;
(ii) (v_x, t_x), (s_x, T_x) and (v_x, F_x) are all used;
(iii) (v_x, t_x) and exactly one of (s_x, T_x) and (v_x, F_x) are used; however, there is at least one internal node in one of the binary trees where the flow does not split down to both children;
(iv) (v_x, t_x) is used, (s_x, T_x) and (v_x, F_x) are not used.

In case (i) where (v_x, t_x) is not used, we get all the flow from s_x through (w, t_x), leading to max-utilization 2. In case (ii) where all of (v_x, t_x), (s_x, T_x) and (v_x, F_x) are used, even splitting implies that $1/2$ of the flow goes down (s_x, T_x) while $1/4$ goes down each of (v_x, t_x) and (v_x, F_x). As a result, we must get $3/4 \cdot 2|x| = 3/2|x|$ flow down (w, t_x), leading to a max-utilization of $3/2$. In case (iii) we have $|x|$ units of flow arriving at T_x (or F_x). Consider a node a of the binary tree below T_x (or F_x) where the flow does not split and which is closest possible to T_x (or F_x). Then the flow splits evenly above a, so a receives exactly one unit of flow for each leaf descending from a. However, all the flow to a is sent down to one child with only half as many descending leaves, so one of these leaves must receive 2 units of flow, which is twice the capacity of its outgoing arc. In case (iv), (v_x, t_x) gets a flow of $2|x|$, leading to max-utilization 2.

We have now shown that an evenly splitting flow leads to a max-utilization equal to 1 if and only if the flow is canonical and corresponds to a satisfiable assignment; any other evenly splitting flow leads to a max-utilization greater than or equal to 3/2.

It remains to show that any satisfiable canonical flow can be achieved by a suitable set of weights for OSPF routing. This is done as follows. All paths from s_x to t_x going through T_x use $4 + \log_2 |x|$ arcs, those going through F_x use $5 + \log_2 |x|$ arcs, and the direct path s_x, v_x, t_x uses 2 arcs. If the canonical flow leads to a true value for x, then flow splits equally at s_x and does not split at v_x, which is achieved if the paths going through T_x and the direct path have the same weight, while the paths going through F_x have a larger weight. Giving a weight equal to $3 + \log_2 |x|$ to (v_x, t_x) and a weight equal to 1 to all the other arcs leads to such a weight setting. Note that these weights are all integers as $|x|$ was defined as a power of 2. If the canonical flow leads to a false value for x, then flow does not split at s_x and splits equally at v_x, which is achieved if the paths going through F_x and the direct path have the same weight, while the paths going through T_x have a larger weight. Giving a weight equal to $4 + \log_2 |x|$ to (v_x, t_x), a weight equal to 3 to (s_x, T_x) and a weight equal to 1 to all the other arcs leads to such a weight setting.

In conclusion, there exists a satisfiable assignment if and only if there exists an OSPF weight setting leading to a max-utilization of 1. Conversely, we have shown that any evenly splitting flow that does not correspond to a satisfying truth assignment has max-utilization greater than or equal to 3/2. Hence, approximating the max-utilization within a factor $<3/2$ implies a solution to 3SAT. Since the reduction is polynomial, the NP-hardness result follows. □

We will now prove the other result from Section 3, stating the inapproximability of the OSPF weight setting problem with respect to a large class of cost functions, including our cost function Φ.

Theorem 4. *Let α and β be fixed constants. Suppose our total routing cost is $\Psi = \sum_{a\in A} \Psi_a(l_a)$ where*

$$\begin{aligned} \Psi_a(l) &\le \alpha l && \text{if } l \le c_a, \\ \Psi_a(l) &\ge \alpha c_a + \beta(l - c_a) && \text{if } l > c_a. \end{aligned}$$

Then, if $\beta \ge 52\alpha$, it is NP-hard to optimize the OSPF weight setting with respect to $\Psi = \sum_{a\in A} \Psi_a(l_a)$ within a factor $0.72 + \beta/38\alpha > 2$ from optimality.

Recall from Section 3 that Theorem 4 implies an inapproximability factor of 3.1 for our cost function Φ. The proof follows the same pattern as we used for max-utilization, but is more complex because we are dealing with a sum rather than a maximum. It is based on the following deep inapproximability result implicit in [24] (Håstad, personal communication):

Lemma 7 (Håstad)**.** *Given a satisfiable instance of Max-SAT where each clause has* 3 *literals and each variable has the same number of positive and negative occurrences, it is NP-hard to satisfy a fraction* $701/800$ *of the clauses.*

Proof of Theorem 4: We prove this result by reducing a satisfiable instance of Max-SAT where each clause has 3 literals and each variable has the same number of positive and negative occurrences to the OSPF weight setting problem, such that we can use Lemma 7. Our reduction is similar to that for max-utilization, and is illustrated in figure 14, where arcs are annotated by their capacities. The main differences relative to construction for max-utilization are that we have subdivided some arcs into paths so as to give them higher weight in the cost function, and that we have contracted the binary trees rooted in T_x and F_x.

Formally, let $S = (X, C)$ be a satisfiable instance of Max-SAT with variable set X and clause set C where each clause has 3 literals and each variable has the same number of positive and negative occurrences. Let $|x|$ denote this number of occurrences for a clause x in C.

As for max-utilization, there is a node c associated to each clause $c \in C$, connected to a node w through a path P_c—instead of a single arc—where P_c has $|P_c| = 100$ arcs of

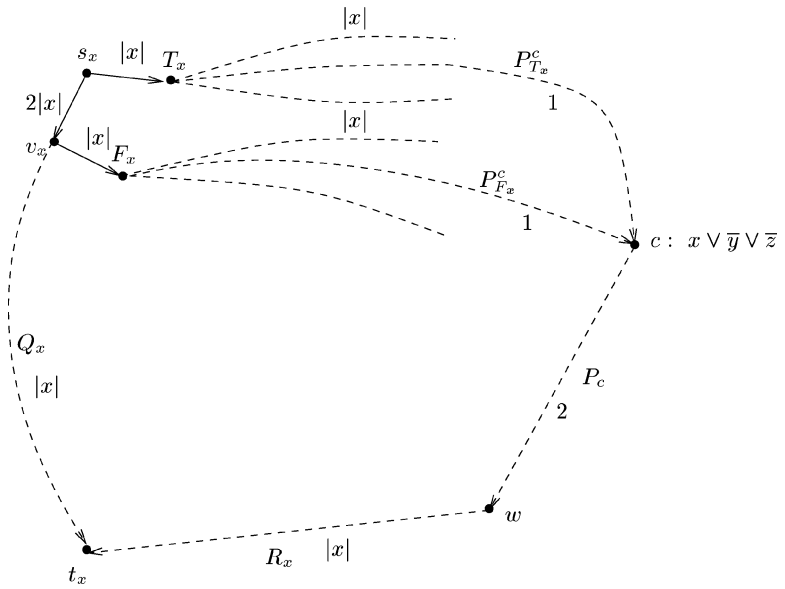

Figure 14. Reduction from Max-SAT to our OSPF cost function.

capacity 2. With each variable $x \in X$, we associate a source s_x and a destination t_x, and a demand of $2|x|$ between them. From s_x, there are outgoing arcs to two nodes v_x and T_x and there is an arc from v_x to a node F_x. Instead of the arc (v_x, t_x), there is now a path Q_x from v_x to t_x. The arcs (s_x, T_x), (s_x, v_x), and (v_x, F_x) all have capacity $|x|$. Path Q_x has $|Q_x| = 103$ arcs, each of capacity $|x|$.

¿From T_x, there is a path $P^c_{T_x}$ to c for each clause c in which x occurs negatively. Similarly, from F_x there is a path $P^c_{F_x}$ to c for each clause c in which x occurs positively. Path $P^c_{F_x}$ has $|P^c_{F_x}| = 102$ arcs and path $P^c_{T_x}$ has $|P^c_{T_x}| = 103$ arcs, each of capacity 1. Finally, for each $x \in X$, we have a path R_x from w to t_x with 3 arcs, each with capacity $|x|$.

As in Theorem 5, we define a *canonical flow* in this network as a flow corresponding to an assignment of values to the variables in X. In a canonical flow, we have $|x|$ units of flow going down Q_x and $|x|$ units going to either T_x or F_x depending on whether x is true or false. If the flow comes to T_x, it spreads evenly so that each $P^c_{T_x}$ gets 1 unit of flow going to c. Similarly, if the flow is sent to F_x, each $P^c_{F_x}$ gets 1 unit of flow going to c. All flow arriving a clause c is then sent to w via P_c, and finally to t_x via R_x. With this assignment of flows, for every clause c, there is 1 unit of flow through P_c for each negative literal in c. Thus, if the clause c is satisfied, it has at most 2 negative literals and then the load stays within the capacity on P_c. Otherwise, the load on P_c is one unit above capacity. All flows outside the clause paths P_c stay within the capacities.

Our hardness result is based on showing that it is hard to get a flow of cost close to that of the canonical flow corresponding to a satisfying truth assignment. Since, such a satisfying flow has all loads within capacity, we only make the approximation easier if we reduce the

costs of loads above the capacity. Hence, for proving the hardness, we may assume

$$\Psi_a'(l) = \beta \quad \text{if } l > c_a.$$

The remainder of the proof is organized as follows: first we show that we can transform in polynomial time a non-canonical even splitting flow to a canonical flow of lower cost. Next we show that any canonical flow can be achieved with a suitable weight setting. Finally, we apply Lemma 7 to show that a canonical flow of low cost is hard to find.

Lemma 8. *A non-canonical even splitting flow can be transformed in polynomial time to a canonical flow of lower cost.*

Proof: Suppose we have a non-canonical flow. We show we can always find a flow of lower cost. It is easy to see that a non-canonical flow satisfies at least one of the following conditions for some x:

(i) Q_x is used, (s_x, T_x) and (v_x, F_x) are not used;
(ii) Q_x is not used and only one of (s_x, T_x) and (v_x, F_x) is used;
(iii) Q_x is not used and both (s_x, T_x) and (v_x, F_x) are used;
(iv) Q_x, (s_x, T_x) and (v_x, F_x) are all used;
(v) (a) if there is flow through F_x, not all $P^c_{F_x}$ paths are used;
(b) if there is flow through T_x, not all $P^c_{T_x}$ paths are used.

In case (i), Q_x receives $2|x|$ units of flow, i.e. $|x|$ units above capacity. If we move this extra flow—by splitting equally in v_x—to F_x and then send one unit through each $P^c_{F_x}$, we decrease the cost on Q_x by $\beta|Q_x||x|$, while none of (v_x, F_x), $P^c_{F_x}$ and R_x get above capacity. As (v_x, F_x), $P^c_{F_x}$ and R_x were not used before, each unit of flow transferred to these paths costs at most α per arc. Only P_c could get an increase of $|x|$ units above capacity, so the total increase in cost due to the new routing is bounded by $(\alpha(1+|P^c_{F_x}|+|R_x|)+\beta|P_c|)|x|$. The net decrease in cost is thus

$$\big(\beta|Q_x| - \big(\beta|P_c| + \alpha\big(1 + \big|P^c_{F_x}\big| + |R_x|\big)\big)\big)|x| = (3\beta - 106\alpha)|x| > 0,$$

as $\beta \geq 52\alpha$.

Similarly for case (ii), if only (s_x, T_x) or only (v_x, F_x) is used, moving half the flow to Q_x (which was empty) leads to a decrease in cost of at least

$$\big(\beta\big(1 + \min\big\{\big|P^c_{F_x}\big|, \big|P^c_{T_x}\big|\big\} + |R_x|\big) + \alpha|P_c| \;-\; \alpha|Q_x|\big)|x| = (106\beta - 3\alpha)|x| > 0.$$

Above we did not count gain from moving flow from clause paths P_c. The point is that these paths may have had load strictly below capacity, and then we have no lower bound on what they cost.

In case (iii), R_x has $|x|$ units of flow above capacity. Moving the $|x|$ units of flow going through F_x to the empty $|Q_x|$ leads to a decrease in cost of at least

$$\big(\beta|R_x| + \alpha\big(1 + \big|P^c_{F_x}\big| + |P_c|\big) \;-\; \alpha|Q_x|\big)|x| = (3\beta + 100\alpha)|x| > 0.$$

In case (iv), the even splitting implies that we get $|x|/2$ units of flow to Q_x and F_x while we get $|x|$ units of flow to T_x. We then move the flow from F_x to Q_x. By doing so, we reduce the flow through R_x from $3|x|/2$ to $|x|$. The only place where the flow and hence the cost increases is in Q_x, from $|x|/2$ to $|x|$. However, the flow in Q_x remains within the capacity, and the total cost on this path is bounded by $\alpha|Q_x||x|$, leading to a decrease in cost of

$$\big(\beta|R_x| + \alpha\big(1 + \big|P^c_{F_x}\big| + |P_c|\big) \; - \; 2\alpha|Q_x|\big)\frac{|x|}{2} = (3\beta - 3\alpha)\frac{|x|}{2} > 0.$$

Thus we can now assume that Q_x and exactly one of F_x and T_x are used, as in (v). To deal with case (v)(a), it remains to show that if F_x is included, then it is beneficial to include all paths $P^c_{F_x}$. Suppose q of them were not included, and now we include them. Afterwards, each path $P^c_{F_x}$ has exactly 1 unit of flow corresponding to its capacity. This means that the q units of flow that we moved to the empty paths were previously above capacity. However, now q units of flow are sent differently through P_c, each at cost at most $\beta|P_c|$, leading to a cost decrease of at least

$$\big(\beta\big|P^c_{F_x}\big| \; - \; \big(\alpha\big|P^c_{F_x}\big| + \beta|P_c|\big)\big)q = (2\beta - 102\alpha)q > 0.$$

In case (v) (b) the cost decrease becomes

$$\big(\beta\big|P^c_{T_x}\big| \; - \; \big(\alpha\big|P^c_{T_x}\big| + \beta|P_c|\big)\big)q = (3\beta - 103\alpha)q > 0.$$

□

Lemma 9. *Any canonical flow can be achieved with a suitable OSPF weight setting.*

Proof: All paths from s_x to t_x going through T_x or F_x use 207 arcs, while path s_x, v_x, Q_x uses 104 arcs. If the canonical flow leads to a true value for x, then flow splits equally in s_x and does not split in v_x, which is achieved if the paths going through T_x and path s_x, v_x, Q_x have the same weight, while the paths going through F_x have a larger weight. Giving a weight equal to 2 to arcs in Q_x and to (v_x, F_x) and a weight equal to 1 to all the other arcs leads to such a weight setting. In the case a false value is assigned to x, similar arguments lead to a weight equal to 2 for arcs in Q_x and for (s_x, T_x) and a weight equal to 1 for all the other arcs. □

We are now ready for finishing the proof of Theorem 4. By Lemma 8, we know that only canonical flows are of interest, and Lemma 9 ensures these flows can be obtained using OSPF routing.

A canonical flow corresponding to a satisfiable assignment of values to variables in X is such that all loads remain within capacities. Moreover, for each variable x, $|x|$ units of flow are sent through T_x or F_x on paths using 207 arcs, and $|x|$ units of flow are sent through Q_x on paths using 104 arcs, leading to a total cost of at most $311\alpha \sum_{x\in X} |x|$.

On the other hand, consider the canonical flow corresponding to an assignment satisfying at most $701/800$ of the clauses. The routing down Q_x, the $P^c_{T_x}$ or $P^c_{F_x}$, and R_x costs α per unit

of flow and per arc, since all the used paths are exactly loaded to their capacity. Moreover, for each unsatisfied clause c, P_c has a load of 3 while the capacity is 2, thus contributing $(2\alpha + \beta)|P_c|$ to the total cost. As the total number of clauses is $\sum_{x \in X} 2|x|/3$, the total cost is at least

$$\sum_{x \in X} \left(\left(|Q_x| + \left| P^c_{T_x} \right| + |R_x| \right) \alpha |x| \right) + (2\alpha + \beta)|P_c| \frac{99}{800} \frac{2}{3} \sum_{x \in X} |x|$$
$$= (225.5\alpha + 8.25\beta) \sum_{x \in X} |x|,$$

where we ignore any load that may be below capacity for some P_c.

By Lemma 7, it is NP-hard to get an assignment satisfying more than a fraction $701/800$ of the clauses, so we can conclude that it is NP-hard to approximate the optimal OSPF weight setting within a factor of

$$\frac{225.5\alpha + 8.25\beta}{311\alpha} > 0.72 + \frac{\beta}{38\alpha},$$

which concludes the proof of Theorem 4. □

Acknowledgment

We would like to thank David Johnson and Jennifer Rexford for some very useful comments. The first author was sponsored by the AT&T Research Prize 1997.

References

1. E.H.L. Aarts and J.K. Lenstra, Local Search in Combinatorial Optimization. Discrete Mathematics and Optimization Wiley-Interscience: Chichester, England, 1997.
2. D. Applegate and M. Thorup, "Load optimal mpls routing with $n + m$ labels," in Proc. 22nd IEEE Conf. on Computer Communications (INFOCOM), (to appear), 2003.
3. R. Battiti and G. Tecchiolli, "The reactive tabu search," ORSA Journal on Computing, vol. 6, no. 2, pp. 126–140, 1994.
4. W. Ben-Ameur and E. Gourdin, "Internet routing and related topology issues," Technical report, France Telecom R&D, 2001.
5. W. Ben-Ameur, N. Michel, E. Gourdin, and B. Liau, "Routing strategies for IP networks," Telektronikk, vols. 2/3, pp. 145–158, 2001.
6. A. Bley, M. Grötchel, and R. Wessäly, "Design of broadband virtual private networks: Model and heuristics for the B-WiN," in Proc. DIMACS Workshop on Robust Communication Networks and Survivability, AMS-DIMACS Series, vol. 53, 1998, pp. 1–16.
7. L.S. Buriol, M.G.C. Resende, C.C. Ribeiro, and M. Thorup, "A memetic algorithms for OSPF routing," in Proceedings of the 6th INFORMS Telecom, 2002, pp. 187–188.
8. K. Calvert, M. Doar, and E.W. Zegura, "Modeling internet topology," IEEE Communications Magazine, vol. 35, no. 6, pp. 160–163, 1997.
9. W. Carlton and J. Barnes, "A note on hashing functions and tabu search algorithms," European Journal of Operational Research, vol. 95, pp. 237–239, 1996.

10. J.L. Carter and M.N. Wegman, "Universal classes of hash functions," Jounal of Computer and System Sciences, vol. 18, pp. 143–154, 1979.
11. Cisco, "Configuring OSPF," 1997, Documentation at http://www.cisco.com/[4]univercd/cc/td/doc/product/software/ios113ed/113ed_cr/np1_c/1cospf.htm.
12. S. Cook, "The complexity of theorem proving procedures," in Proc. 3rd ACM Symp. on Theory of Computing (STOC), 1971, pp. 151–158.
13. M. Dietzfelbinger, "Universal hashing and k-wise independent random variables via integer arithmetic without primes," in Proc. 13th Symp. on Theoretical Aspects of Computer Science (STACS), LNCS vol. 1046, Springer, 1996, pp. 569–580.
14. M. Ericsson, M. Resende, and P. Pardalos, "A genetic algorithm for the weight setting problem in OSPF routing," 2001, To appear in J. Combinatorial Optimization in 2002.
15. B. Fortz, J. Rexford, and M. Thorup, "Traffic engineering with traditional IP routing protocols," IEEE Communications Magazine, vol. 40, no. 10, pp. 118–124, 2002.
16. B. Fortz and M. Thorup, "Increasing internet capacity using local search," Technical Report IS-MG 2000/21, Université Libre de Bruxelles, 2000a. http://smg.ulb.ac.be/Preprints/Fortz00_21.html.
17. B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," in Proc. 19th IEEE Conf. on Computer Communications (INFOCOM), 2000b, pp. 519–528.
18. B. Fortz and M. Thorup, "Optimizing OSPF/IS-IS weights in a changing world," IEEE Journal on Selected Areas in Communications, vol. 20, no. 4, pp. 756–767, 2002.
19. D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone, "Experimental analysis of dynamic algorithms for the single-source shortest path problem," ACM Jounal of Experimental Algorithmics, vol. 3, no. 5, 1998.
20. F. Glover, "Future paths for integer programming and links to artificial intelligence," Computers & Operations Research, vol. 13, pp. 533–549, 1986.
21. F. Glover, "Tabu search–Part I," ORSA Journal on Computing, vol. 1, no. 3, pp. 190–206, 1989.
22. F. Glover, "Tabu search–Part II," ORSA Journal on Computing, vol. 2, no. 1, pp. 4–32, 1990.
23. F. Glover and M. Laguna, Tabu Search, Kluwer Academic Publishers, 1997.
24. J. Håstad, "Some optimal inapproximability results," Journal of the ACM, vol. 48, no. 4, pp. 798–859, 2001.
25. D.E. Knuth, "The Art of Computer Programming III: Sorting and Searching," Addison–Wesley: Reading, MA, 1973.
26. F. Lin and J. Wang, "Minimax open shortest path first routing algorithms in networks supporing the smds services," in Proc. IEEE International Conference on Communications (ICC), vol. 2, 1993, pp. 666–670.
27. D. Mitra and K. Ramakrishnan, "A case study of multiservice, multipriority traffic engineering design for data networks," in Proc. IEEE GLOBECOM, 1999, pp. 1077–1083.
28. J.T. Moy, OSPF: Anatomy of an Internet Routing Protocal, Addison-Wesley, 1999.
29. G. Ramalingam and T. Reps, "An incremental algorithm for a generalization of the shortest-path problem," Jounal of Algorithms, vol. 21, no. 2, pp. 267–305, 1996.
30. M. Rodrigues and K. Ramakrishnan, "Optimal routing in data networks," Presentation at International Telecommunications Symposium (ITS), Rio de Genero, Brazil, 1994.
31. E.C. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture," Network Working Group, Internet Draft (work in progress), 1999. http://search.ietf.org/internet-drafts/draft-ietf-mpls-arch-05.txt.
32. M. Roughan and M. Thorup, "Avoiding ties in shortest path first routing," Technical report, AT&T Labs-Research, 2002.
33. Sprint, "Sprint IP backbone network and MPLS," 2002, White Paper http://www.sprintbiz.com/resource_library/resources/SprintCiscoMPLS.pdf.
34. M. Thorup, "Even strongly universal hashing is pretty fast," in Proc. 11th ACM-SIAM Symp. on Discrete Algorithms (SODA), 2000, pp. 496–497.
35. B.M. Waxman, "Routing of multipoint connections," IEEE Jour. Selected Areas in Communications (Special Issue on Broadband Packet Communications), vol. 6, no. 9, pp. 1617–1622, 1988.
36. D.L. Woodruff and E. Zemel, "Hashing vectors for tabu search," Annals of Operations Research, vol. 41, pp. 123–137, 1993.
37. X. Xiao, A. Hannan, B. Bailey, and L. Ni, "Traffic engineering with MPLS in the Internet," IEEE Network Magazine, vol. 14, no. 2, pp. 28–33, 2000.

38. E.W. Zegura, "GT-ITM: Georgia tech internetwork topology models (software)," 1996. http://www.cc.gatech.edu/fac/Ellen.Zegura/gt-itm/gt-itm.tar.gz.
39. E.W. Zegura, K.L. Calvert, and S. Bhattacharjee, "How to model an internetwork," in Proc. 15th IEEE Conf. on Computer Communications (INFOCOM), 1996, pp. 594–602.
40. A.L. Zobrist and F.R. Carlson, Jr., "Detection of combined occurrences," Comm. ACM, vol. 20, no. 1, pp. 31–35, 1977.