A Hybrid Genetic Algorithm for the Weight Setting Problem in OSPF/IS-IS Routing

L.S. Buriol*
Faculdade de Engenharia Elétrica e de Computação, UNICAMP, 13083-970 Campinas, SP, Brazil

M.G.C. Resende
Internet and Network Systems Research Center, AT&T Labs Research, 180 Park Avenue, Room C241, Florham Park, New Jersey 07932

C.C. Ribeiro
Department of Computer Science, Catholic University of Rio de Janeiro, R. Marquês de São Vicente, 225, Rio de Janeiro, RJ 22453-900 Brazil

M. Thorup
Internet and Network Systems Research Center, AT&T Labs Research, 180 Park Avenue, Room C227, Florham Park, New Jersey 07932

Intradomain traffic engineering aims to make more efficient use of network resources within an autonomous system. Interior Gateway Protocols such as OSPF (Open Shortest Path First) and IS-IS (Intermediate System-Intermediate System) are commonly used to select the paths along which traffic is routed within an autonomous system. These routing protocols direct traffic based on link weights assigned by the network operator. Each router in the autonomous system computes shortest paths and creates destination tables used to direct each packet to the next router on the path to its final destination. Given a set of traffic demands between origin-destination pairs, the *OSPF weight setting problem* consists of determining weights to be assigned to the links so as to optimize a cost function, typically associated with a network congestion measure. In this article, we propose a genetic algorithm with a local improvement procedure for the OSPF weight-setting problem. The local improvement procedure makes use of an efficient dynamic shortest path algorithm to recompute shortest paths after the modification of link weights. We test the algorithm on a set of real and synthetic test problems, and show that it produces near-optimal solutions. We compare the hybrid algorithm with other algorithms for this problem illustrating its efficiency and robustness.
 NETWORKS, Vol. 46(1), 36–56 2005

Keywords: OSPF routing; IS-IS routing; Internet; metaheuristics; genetic algorithm; optimized crossover; local search

1. INTRODUCTION

The Internet is divided into many routing domains, called autonomous systems (ASes). ASes are networks that consist of routers and links connecting the routers. The number of backbone and access routers in 10 Internet service providers studied by Spring et al. [21] vary from as few as 11 to as many as 862. There are 12 links in the smallest network studied and 5299 in the largest. When customer and peer routers are considered, the networks can have thousands of routers and links. These ASes interact to control and deliver IP traffic. They typically fall under the administration of a single institution, such as a company, a university, or a service provider. Neighboring ASes use the Border Gateway Protocol (BGP) to route traffic [23].

The goal of intradomain traffic engineering [9] consists in improving user performance and making more efficient use of network resources within an AS. Interior Gateway Protocols (IGPs) such as OSPF (Open Shortest Path First) and IS-IS (Intermediate System-Intermediate System) are commonly used to select the paths along which traffic is routed within an AS.

Received July 2004; accepted March 2005
Correspondence to: M.G.C. Resende; e-mail: mgcr@research.att.com
*This research was done while the first author was a visiting scholar at the Internet and Network Systems Research Center at AT&T Labs Research (AT&T Labs Research Technical Report TD-5NTN5G).
DOI 10.1002/net.20070
Published online in Wiley InterScience (www.interscience.wiley.com).

These routing protocols direct traffic based on link weights assigned by the network operator. Each router in the AS computes shortest paths and creates destination tables used to direct each IP packet to the next router on the path to its final destination. OSPF calculates routes as follows. To each link is assigned an integer weight ranging from 1 to $65535 (= 2^{16} - 1)$. The weight of a path is the sum of the link weights on the path. OSPF mandates that each router computes a graph of shortest paths with itself as the root [13]. This graph gives the least weight routes (including multiple routes in case of ties) to all destinations in the AS. In the case of multiple shortest paths originating at a router, OSPF is usually implemented so that it will accomplish load balancing by splitting the traffic flow over all shortest paths leaving from each router [16]. In this article, we consider that traffic is split evenly between all outgoing links on the shortest paths to the destination IP address. OSPF requires routers to exchange routing information with all the other routers in the AS. Complete network topology knowledge is required for the computation of the shortest paths.

Given a set of traffic demands between origin-destination pairs [8], the *OSPF weight setting problem* consists in determining weights to be assigned to the links so as to optimize a cost function, typically associated with a network congestion measure.

In reality, a global optimization involving all weights in the network is impractical because it can cause major disruptions in traffic. For the daily maintainance of a network, it is much more attractive to adapt the weight setting with a few weight changes at the time. Nevertheless, as a benchmark, it is important for the network operator to know how well the newtork could perform with a complete change of weights. A complete change of weights could be considered an option if the network performance has deteriorated too much. A comprehensive description of the reality of traffic engineering is given in Fortz et al. [9].

The NP-hardness of the OSPF weight-setting problem was established in [10]. Previous work on optimizing OSPF weights 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 [3, 14, 18]. Fortz and Thorup [10] were the first to consider even traffic splitting in OSPF weight setting. They proposed a local search heuristic and tested it on a realistic AT&T backbone network and on synthetic networks. Pióro et al. [17] propose heuristics based on local search, simulated annealing, and Lagrangian relaxation for several optimization problems related to setting OSPF weights with even traffic splitting. The computational results presented, however, were limited to small instances with up to 12 routers, 18 links, and 66 demand pairs. Ericsson et al. [7] proposed a genetic algorithm and used the set of test problems considered in [10]. Sridharan et al. [22] developed another heuristic for a slightly different version of the problem, in which flow is split among a subset of the outgoing links on the shortest paths to the destination IP address.

In this article, we propose a hybrid genetic algorithm incorporating a local improvement procedure to the crossover operator of the genetic algorithm proposed in [7]. The local improvement procedure makes use of an efficient dynamic shortest path algorithm to recompute shortest paths after the modification of link weights. We compare the hybrid algorithm with the genetic algorithm as well as with the local search procedure in [10].

In the next section, we give the mathematical formulation of the OSPF weight-setting problem. The hybrid genetic algorithm is described in Section 3, and the local improvement procedure in Section 4. Section 5 describes efficient algorithms for solution update used in the local improvement procedure. Computational results are reported in Section 6. Concluding remarks are made in the last section.

2. PROBLEM FORMULATION

In a data communication network, nodes and arcs represent routers and transmission links, respectively. Let N and A denote, respectively, the sets of nodes and arcs. Data packets are routed along links, which have fixed capacities. Consider a directed network graph $G = (N, A)$ with a capacity c_a for each $a \in A$, and a demand matrix D that, for each pair $(s, t) \in N \times N$, gives the demand d_{st} in traffic flow from node s to node t. Then, the OSPF weight-setting problem consists in assigning positive integer weights $w_a \in [1, w_{\max}]$ to each arc $a \in A$, such that a measure of routing cost is optimized when the demands are routed according to the rules of the OSPF protocol. The OSPF protocol allows for $w_{\max} \leq 65535$.

For each pair (s, t) and each arc a, let $f_a^{(st)}$ indicate how much of the traffic flow from s to t goes over arc a. Let l_a be the total load on arc a, that is, the sum of the flows going over a, and let the trunk utilization rate $u_a = l_a/c_a$. The routing cost in each arc $a \in A$ is taken as the piecewise linear function $\Phi_a(l_a)$, proposed by Fortz and Thorup [10] and depicted in Figure 1, which increasingly penalizes flows approaching or violating the capacity limits:

$$\Phi_a(l_a) = \begin{cases} u_a, & u_a \in [0, 1/3) \\ 3 \cdot u_a - 2/3, & u_a \in [1/3, 2/3), \\ 10 \cdot u_a - 16/3, & u_a \in [2/3, 9/10), \\ 70 \cdot u_a - 178/3, & u_a \in [9/10, 1), \\ 500 \cdot u_a - 1468/3, & u_a \in [1, 11/10), \\ 5000 \cdot u_a - 16318/3, & u_a \in [11/10, \infty). \end{cases} \tag{1}$$

Given a weight assignment w and the loads $l_a^{\mathrm{OSPF}(w)}$ associated with each arc $a \in A$ corresponding to the routes obtained with OSPF, we denote its routing cost by $\Phi_{\mathrm{OSPF}(w)} = \sum_{a \in A} \Phi_a \left(l_a^{\mathrm{OSPF}(w)} \right)$. The OSPF weight-setting problem is then equivalent to finding arc weights $w^* \in [1, w_{\max}]$ such that $\Phi_{\mathrm{OSPF}(w)}$ is minimized.

The general routing problem can be formulated as the following linear programming problem with a piecewise linear

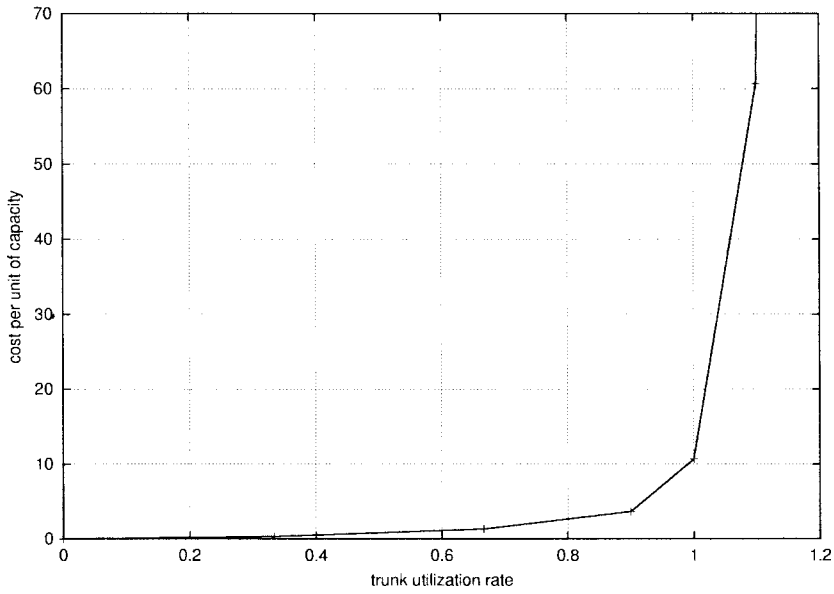

FIG. 1. Piecewise linear function $\Phi_a(l_a)$.

objective function:

$$\Phi_{\mathrm{OPT}} = \min \Phi = \sum_{a \in A} \Phi_a(l_a) \tag{2}$$

subject to

$$\sum_{u:(u,v)\in A} f^{(st)}_{(u,v)} - \sum_{u:(v,u)\in A} f^{(st)}_{(v,u)} = \begin{cases} -d_{st} & \text{if } v = s, \\ d_{st} & \text{if } v = t, \\ 0 & \text{otherwise,} \end{cases} \qquad v, s, t \in N, \tag{3}$$

$$l_a = \sum_{(s,t)\in N\times N} f^{(st)}_a, \qquad a \in A, \tag{4}$$

$$\Phi_a(l_a) \geq l_a, \qquad a \in A, \tag{5}$$

$$\Phi_a(l_a) \geq 3l_a - 2/3c_a, \qquad a \in A, \tag{6}$$

$$\Phi_a(l_a) \geq 10l_a - 16/3c_a, \qquad a \in A, \tag{7}$$

$$\Phi_a(l_a) \geq 70l_a - 178/3c_a, \qquad a \in A, \tag{8}$$

$$\Phi_a(l_a) \geq 500l_a - 1468/3c_a, \qquad a \in A, \tag{9}$$

$$\Phi_a(l_a) \geq 5000l_a - 16318/3c_a, \qquad a \in A, \tag{10}$$

$$f^{(st)}_a \geq 0, \quad a \in A; s, t \in N. \tag{11}$$

Constraints (3) are flow conservation constraints that ensure routing of the desired traffic. Constraints (4) define the load on each arc a and constraints (5–10) define the cost on each arc a according to the cost function $\Phi_a(l_a)$.

The above is a relaxation of OSPF routing, as it allows for arbitrary routing of traffic. Then, Φ_{OPT} is a lower bound on the optimal OSPF routing cost $\Phi_{\mathrm{OSPF}(w^*)}$. Also, if $\Phi_{\mathrm{OSPF}(1)}$ denotes the optimal OSPF routing cost when unit weights are used, then $\Phi_{\mathrm{OSPF}(w^*)} \leq \Phi_{\mathrm{OSPF}(1)}$.

Fortz and Thorup [10] proposed a normalizing scaling factor for the routing cost that makes possible comparisons across different network sizes and topologies:

$$\Phi_{UNCAP} = \sum_{(s,t)\in N\times N} d_{st} h_{st},$$

where h_{st} is the minimum hop count between nodes s and t. For any routing cost Φ, the scaled routing cost is defined as

$$\Phi^* = \Phi / \Phi_{UNCAP}.$$

Using this notation, the following results hold:

- The optimal routing costs satisfy

$$1 \leq \Phi^*_{\mathrm{OPT}} \leq \Phi^*_{\mathrm{OSPF}(w^*)} \leq \Phi^*_{\mathrm{OSPF}(1)} \leq 5000.$$

- Given any solution to (2–11) with normalized routing cost Φ^*, then $\Phi^* = 1$ if and only if all arc loads are below 1/3 of their capacities and all demands are routed on minimum hop routes.
- Given any solution to (2–11) where all arcs are at their maximum capacity, then the normalized routing cost $\Phi^* = 10(2/3)$. We say that a routing *congests* a network if $\Phi^* \geq 10(2/3)$.

3. HYBRID GENETIC ALGORITHM FOR OSPF WEIGHT SETTING

In this section, we summarize the detailed description of the genetic algorithm given in [7] and propose a hybrid genetic algorithm by adding a local improvement procedure after the crossover.

A genetic algorithm is a population-based metaheuristic for combinatorial optimization. In this context, a population is simply a set of feasible solutions. Solutions in a population are combined (through crossover) and perturbed (by mutation) to produce a new generation of solutions. When solutions are combined, attributes of higher quality solutions have a greater probability to be passed down to the next generation. This process is repeated over many generations as long as the quality of the solutions in the new population improves over time. We next show how this idea can be explored for weight setting in OSPF routing.

Each solution is represented by an array of integer weights, where each component corresponds to the weight of an arc of the network. Each individual weight belongs to the interval $[1, w_{\max}]$. Each solution w is associated with a fitness value defined by the OSPF routing cost $\Phi_{\text{OSPF}(w)}$. The initial population is randomly generated, with arc weights selected from a uniform distribution in the interval $[1, w_{\max}/3]$. The population is partitioned into three sets $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. The best solutions are kept in $\mathcal{A}$, while the worst ones are in $\mathcal{C}$. All solutions in $\mathcal{A}$ are promoted to the next generation. Solutions in $\mathcal{B}$ are replaced by crossover of one parent from $\mathcal{A}$ with another from $\mathcal{B} \cup \mathcal{C}$ using the *random keys* crossover scheme of Bean [2]. All solutions in $\mathcal{C}$ are replaced by new randomly generated solutions with arc weights selected in the interval $[1, w_{\max}]$.

In the random keys scheme, crossover is carried out on a selected pair of parent solutions to produce an offspring solution. Each selected pair consists of an elite parent and a non-elite parent. The elite parent is selected, at random, uniformly from solutions in set $\mathcal{A}$, while the nonelite parent is selected, at random, uniformly from solutions in set $\mathcal{B} \cup \mathcal{C}$. Each weight in the offspring solution is either inherited from one of its parents or is reset by mutation. With mutation probability p_m, the weight is reset to a value selected at random in the interval $[1, w_{\max}]$. If mutation does not occur, then the child inherits the weight from its elite parent with a given probability $p_{\mathcal{A}} > 1/2$. Otherwise, it inherits the weight from its nonelite parent.

The hybrid genetic algorithm proposed in this article, applies a local improvement procedure to each offspring solution obtained by crossover. This local improvement procedure is described in the next section.

4. LOCAL IMPROVEMENT PROCEDURE

In this section, we describe the local improvement procedure. Starting from an initial solution, the local improvement procedure analyzes solutions in the neighborhood of a current solution w in the search for a solution having a smaller routing cost. If such a solution exists, then it replaces the current solution. Otherwise, the current solution is returned as a local minimum.

The local improvement procedure is incorporated in the genetic algorithm, described in Section 3, to enhance its ability to find better-quality solutions with less computational effort. Local improvement is applied to each solution generated by the crossover operator. Besides being computationally demanding, the use of large neighborhoods in a hybrid genetic algorithm can lead to loss of population diversity, and consequently premature convergence to low-quality local minima. We next describe the local improvement procedure using a reduced neighborhood.

As before, let l_a denote the total load on arc $a \in A$ in the solution defined by the current weight settings w. We recall that $\Phi_a(l_a)$ denotes the routing cost on this arc. The local improvement procedure examines the effect of increasing the weights of a subset of the arcs. These candidate arcs are selected among those with the highest routing costs and whose weight is smaller than $w_{\max}$. To reduce the routing cost of a candidate arc, the procedure attempts to increase its weight to induce a reduction on its load. If this leads to a reduction in the overall routing cost, the change is accepted and the procedure is restarted. The procedure stops at a local minimum when no improvement results from changing the weights of the candidate arcs. The pseudocode in Figure 2 describes the local improvement procedure in detail.

```
procedure LocalImprovement(q, w)
1    dontlook_a ← 0, ∀a ∈ A;
2    i ← 1;
3    while i ≤ q do
4        Renumber the arc indices such that
         Φ_a(l_a) ≥ Φ_{a+1}(l_{a+1}), ∀a = 1, ..., |A| − 1;
5        a' ← 0;
6        for a = 1, ..., |A| while a' = 0 do
7            if dontlook_a = 1 then dontlook_a ← 0;
8            else if w_a < w_max then a' ← a;
9        end for;
10       if a' = 0 then return;
11       dontlook_{a'} ← 1;
12       for ŵ = w_{a'} + 1, ..., w_{a'} + ⌈(w_max − w_{a'})/4⌉ do
13           w'_a ← w_a, ∀a ∈ A, a ≠ a';
14           w'_{a'} ← ŵ;
15           if Φ_{OSPF(w')} < Φ_{OSPF(w)} then
16               w ← w';
17               dontlook_{a'} ← 0;
18               i ← 0;
19           end if
20       end for
21       i ← i + 1;
22   end while
end LocalImprovement.
```

FIG. 2. Pseudocode of procedure `LocalImprovement`.

The procedure `LocalImprovement` takes as input parameters the current solution defined by the weights w and a parameter q, which specifies the maximum number of candidate arcs to be examined at each local improvement iteration. To speed up the search, we disallow the weight increase of arcs for which no weight increase leads to an improvement in the routing cost in the previous iteration. To implement this strategy, we make use of a *don't look* bit for each arc.

The *don't look* bits are set unmarked in line 1, and the counter of candidate arcs is initialized in line 2. The loop in lines 3 to 22 investigates at most q selected candidate arcs for weight increase in the current solution. The arc indices are renumbered in line 4 such that the arcs are considered in nonincreasing order of routing cost. The loop in lines 6 to 9 searches for an unmarked arc with weight less than $w_{\max}$. Marked arcs, which cannot be selected at the current iteration, are unmarked in line 7 for future investigation. Arc a' is selected in line 8. If no arc satisfying these conditions is found, the procedure stops in line 10 returning the current weights w as the local minimum. In line 11, arc a' is temporarily marked to disallow its investigation in the next iteration, unless a weight change in $w_{a'}$ results in a better solution.

The loop in lines 12 to 20 examines all possible weight changes for arc a' in the range $[w_{a'} + 1, w_{a'} + \lceil (w_{\max} - w_{a'})/4 \rceil]$. A neighbor solution w', keeping all arc weights unchanged except for arc a', is built in lines 13 and 14. If the new solution w' has a smaller routing cost than the current solution (test in line 15), then the current solution is updated in line 16, arc a' is unmarked in line 17, and the arc counter i is reset in line 18. In line 21, we increment the candidate arc counter i.

The routing cost $\Phi_{\mathrm{OSPF}}(w')$ associated with the neighbor solution w' must be evaluated in line 15. Instead of computing it from scratch, we use fast update procedures for recomputing the shortest path graphs as well as the arc loads. These procedures are considered in the next section of the article. Once the new arc loads are known, the total routing cost is computed as the sum of the individual arc routing costs.

5. FAST UPDATES OF ARC LOADS AND ROUTING COSTS

In this section, we describe the procedures used for fast update of the cost (line 15 of procedure `LocalImprovement`) and arc loads. We are in the situation where l are the loads associated with the current weight settings w and the weight of a unique arc a' is increased by exactly a unit.

Let T be the set of destination nodes and denote by $g^t = (N, A^t)$ the shortest path graph associated with each destination node $t \in T$. One or more of these shortest path graphs will be affected by the change of the weight of arc a' from $w_{a'}$ to $w_{a'} + 1$. Consequently, the loads of some of the arcs in each graph will change.

procedure `UpdateCost`(a', d, l)

1 **forall** $a \in A$ **do** $l_a \leftarrow 0$;
2 **forall** $t \in T$ **do**
3 `UpdateShortestPaths`(a', w, t);
4 `UpdateLoads`(d, t);
5 **forall** $a \in A^t$ **do** $l_a \leftarrow l_a + l_a^t$;
6 **end forall**
7 $\Phi_{OSPF(w')} \leftarrow \sum_{a \in A} \Phi_a(l_a)$;

end `UpdateCost`.

FIG. 3. Pseudocode of procedure `UpdateCost`.

The procedures described in this section use a set of data structures that work like a memory for the solution. With a weight change, the shortest path graph and the loads can change, and the memories are updated instead of recomputed from scratch.

Each shortest path graph with node t as destination has an $|A|$-vector A^t indicating the arcs in g^t. If arc a is in the shortest path graph, then $A_a^t = 1$. Otherwise, $A_a^t = 0$. Another $|A|$-vector, l^t, associated with the arcs, stores the partial loads flowing to t traversing each arc $a \in A$. The total load from each arc is represented in the $|A|$-vector l_a, which stores the total load traversing each arc $a \in A$. The $|N|$-vectors π^t and δ^t are associated with the nodes. The distance from each node to the destination t is stored in π^t, while δ^t keeps the number of arcs outgoing from each node in g^t. All these structures are populated in the beginning and set free at the end of procedure *LocalImprovement*. For simplicity, they were omitted from this procedure and from the parameter list of the procedures described in this section.

The pseudocode in Figure 3 summarizes the main steps of the update procedure. The new load l_a on each arc $a \in A$ is set to zero in line 1. The loop in lines 2 to 6 considers each destination node $t \in T$. For each one of them, the shortest path graph g^t is updated in line 3 and the partial arc loads are updated in line 4. The arc loads are updated in line 5. Lines 1 and 5 are removed from this procedure in case the arc loads be updated inside procedure `UpdateLoads`. Finally, the cost $\Phi_{\mathrm{OSPF}(w')}$ of the new solution is computed in line 7. In the remainder of this section we describe the procedures `UpdateShortestPaths` and `UpdateLoads` used in lines 3 and 4.

5.1. Dynamic Reverse Shortest Path Algorithm

We denote by $g^t = (N, A^t)$ the shortest paths graph associated with each destination node $t \in T$. Because the weight of a unique arc a' was changed, the graph g^t does not have to be recomputed from scratch. Instead, we update the part of it which is affected by the weight change. Ramalingam and Reps [19] and Frigioni et al. [12] proposed efficient algorithms for these dynamic computations in Dijkstra's

```
procedure UpdateShortestPaths(a' = (u→,v→), w, t)
1    if a' ∉ A^t return;
2    A^t ← A^t \ {a'};
3    HeapInsertMax(H, u, π^t(u));
4    δ_u^+ ← δ_u^+ − 1;
5    if δ_u^+ > 0 then return
6    Q = {u};
7    forall v ∈ Q do
8        π^t(v) ← π^t(v) + 1;
9        forall a = (u, v) ∈ IN(v) ∩ A^t do
10           A^t ← A^t \ {a};
11           HeapInsertMax(H, u, π^t(u));
12           δ^+(u) ← δ^+(u) − 1;
13           if δ^+(u) = 0 then Q ← Q ∪ {u};
14       end forall
15   end forall
16   forall u ∈ Q do
17       forall a = (u, v) ∈ OUT(u) do
18           if π^t(u) = w_a + π^t(v) then
19               A^t ← A^t ∪ {a};
20               HeapInsertMax(H, u, π^t(u));
21               δ^+(u) ← δ^+(u) + 1;
22           end if
23       end forall
24   end forall
end UpdateShortestPaths.
```

FIG. 4. Pseudocode of procedure UpdateShortestPaths.

algorithm. These two algorithms are compared experimentally in [11]. Although the algorithm of Frigioni et al. is theoretically better, the algorithm of Ramalingam and Reps usually runs faster in practice. Due to the nature of the OSPF weight setting problem, we use the reversed version of Dijkstra's shortest path algorithm.

The pseudocode of the specialized dynamic shortest path algorithm for unit weight increases is given in Figure 4. Buriol et al. [4] showed empirically that this algorithm is faster than the general dynamic reverse shortest path algorithm of Ramalingam and Reps [19]. First, it identifies the set Q of nodes whose distance labels change due to the increased weight. Next, the shortest paths graph is updated by deleting and adding arcs for which at least one of its extremities belongs to Q.

Algorithm UpdateShortestPaths takes as parameters the arc $a' = (\overrightarrow{u,v})$, whose weight changed, the current setting of weights w, and a destination node $t \in T$. The algorithm checks in line 1 if arc a' does not belong to the shortest paths graph g^t, in which case the weight change does not affect the latter and the procedure returns. Arc a' is eliminated from the shortest paths graph g^t in line 2. In line 3 the tail node of arc a' is inserted in a heap containing all nodes for which the load of its outgoing links might change. This heap will be used later by the load update procedure. The distances to the destination node t are used as priority keys and the root contains the node with maximum distance. The outdegree δ_u^+ of the tail node of arc a' is updated in line 4. In line 5, we check if there is an alternative path to the destination starting from u. If this is the case, the procedure returns in line 5, because no further change is needed. The set Q of nodes affected by the weight change in arc a' is initialized with node u in line 6. The loop in lines 7 to 15 builds the set Q. For each node identified in this set (line 7), its distance $\pi^t(v)$ to the destination node is increased by 1 in line 8. The loop in lines 16 to 24 updates the shortest paths. Each node u in set Q is investigated one-by-one in line 16 and each outgoing arc a is scanned in line 17. We check in line 18 if arc a belongs to the new shortest path to the destination. If so, arc a is inserted in the shortest paths graph in line 19, its tail u is inserted in the heap in line 20, and its outdegree $\delta^+(u)$ is updated in line 21.

```
procedure UpdateLoads(H, d, t)
1    while HeapSize(H) > 0 do
2        u ← HeapExtractMax(H);
3        load ← (d_ut + Σ_{a=(v,u)∈A^t} l_a^t)/δ^+(u);
4        forall a = (u, v) ∈ A^t : l_a^t ≠ load do
5            l_a^t ← load;
6            HeapInsertMax(H, v, π(v));
7        end forall
8    end while
end UpdateLoads.
```

FIG. 5. Pseudocode of procedure UpdateLoads.

5.2. Dynamic Load Update

We first recall that procedure UpdateShortestPaths built a heap H containing all nodes for which the set of outgoing arcs was modified in the shortest paths graph.

The pseudocode in Figure 5 summarizes the main steps of the load update procedure. We denote by l_a^t the load on arc $a \in A$ associated with the destination node $t \in T$. Procedure UpdateLoads takes as parameters the demands d and a destination node t. The loop in lines 1 to 9 removes nodes from the heap until the heap becomes empty. The node u with maximum distance to the destination node is removed in line 2. The total load flowing through node u is equal $d_{ut} + \sum_{a=(v,u)\in A^t} l_a$. The load in each arc leaving node u is

computed in line 3. The loop in lines 4 to 7 scans all arcs leaving node u in the current shortest paths graph for which the partial load l_a^t has to be updated. The new partial load l_a^t is set in line 5 and the head v of arc a is inserted in the heap H in line 6.

6. COMPUTATIONAL RESULTS

In this section, we describe the experimental results using the hybrid genetic algorithm introduced in this article. We describe the computer environment, list the values of the algorithm parameters, present the test problems, and outline the experimental setup.

In the experiments, we compare the hybrid genetic algorithm with the lower bound associated with the linear program (2–11) and other heuristics.

6.1. The Setup

The experiments were done on an SGI Challenge computer (28 196-MHz MIPS R10000 processors) with 7.6 Gb of memory. Each run used a single processor.

The algorithms were implemented in C and compiled with the MIPSpro `cc` compiler, version 7.30, using flag `-O3`. Running times were measured with the `getrusage` function. Random numbers were generated in the hybrid genetic algorithm as well as in the pure genetic algorithm using Matsumoto and Nishimura's *Mersenne Twister* [15].

The following parameters were set in both the pure and the hybrid genetic algorithms:

- Population size: 50.
- Weight range: $[1, w_{\max} = 20]$.
- Population partitioning placed the top 25% of the solutions (rounded up to 13) in set $\mathcal{A}$, the bottom 5% of the solutions (rounded up to 3) in set $\mathcal{C}$, and the remaining solutions in set $\mathcal{B}$.
- Probability that mutation occurs: $p_m = 0.01$.
- Probability that an offspring inherits the weight from the elite parent during crossover: $p_{\mathcal{A}} = 0.7$.
- The number of generations varied according to the type of experiment.

In addition, the maximum number of candidate arcs is set to $q = 5$ in the local improvement procedure in the hybrid genetic algorithm.

The experiments were done on 13 networks from four classes proposed by Fortz and Thorup [10] and also used in [7]. The networks are summarized in Table 1. The *AT&T Worldnet backbone* is a proposed real-world network of 90 routers and 274 links. The *2-level hierarchical* networks are generated using the GT-ITM generator [26], based on a model of Calvert et al. [5] and Zegura et al. [27]. This model uses two types of arcs: local access arcs have capacities equal to 200, while long distance arcs have capacities equal to 1000. For the class of *random* networks, the probability of having an arc between two nodes is given by a parameter that controls the density of the network. All arc capacities are set to 1000. In *Waxman* networks, the nodes are points uniformly distributed in the unit square. The probability of having an arc between nodes u and v is $\eta e^{-\Delta(u,v)/(2\theta)}$, where η is a parameter used to control the density of the network, $\Delta(u, v)$ is the Euclidean distance between u and v, and θ is the maximum distance between any two nodes in the network [25]. All arc capacities are set to 1000. Fortz and Thorup generated the demands to force some nodes to be more active senders or receivers than others, thus modeling hot spots on the network. Their generation assigns higher demands to closely located nodes pairs.

For each instance, 12 distinct demand matrices $D^1, D^2, \ldots, D^{12}$ are generated. Starting from demand matrix D^1, the other demand matrices are generated by repeatedly multiplying D^1 by a scaling factor: $D^k = \rho^{k-1} D^1, \forall\ k = 1, \ldots, 12$. Table 1 summarizes the test problems. For each problem, the table lists its class, name, number of nodes, number of arcs, number of destination nodes, number of origin-destination pairs, the total demand of D^1, and the scaling factor.

TABLE 1. Network characteristics: Class name, instance name, number of nodes ($|N|$), number of arcs ($|A|$), number of destination nodes ($|T|$), number of demand pairs (o-d pairs), total demand ($\sum d_{uv}$), and demand scaling factor (ρ).

Class	Name	$\|N\|$	$\|A\|$	$\|T\|$	o-d pairs	$\sum d_{uv}$	ρ
AT&T backbone	att	90	274	17	272	18465	0.2036885
2-level hierarchical	hier100	100	280	100	9900	921	0.41670955
	hier100a	100	360	100	9900	1033	1.0008225
	hier50a	50	148	50	2450	276	1.4855075
	hier50b	50	212	50	2450	266	1.0534975
Random	rand100	100	403	100	9900	994	5.8109135
	rand100b	100	503	100	9900	1026	8.1702935
	rand50	50	228	50	2450	249	14.139605
	rand50a	50	245	50	2450	236	18.941735
Waxman	wax100	100	391	100	9900	1143	3.5349025
	wax100a	100	476	100	9900	858	6.1694055
	wax50	50	169	50	2450	277	7.6458185
	wax50a	50	230	50	2450	264	12.454005

6.2. *Fixed Time Comparison*

In this subsection, we compare the hybrid genetic algorithm with three heuristics:

- InvCap: weights are set proportional to the inverse of the link capacity, that is, $w_a = \lceil c_{\max}/c_a \rceil$, where $c_{\max}$ is the maximum link capacity;
- GA: the basic genetic algorithm without the local search used by the hybrid genetic algorithm;
- LS: the local search algorithm of Fortz and Thorup [10];

as well as with LPLB, the linear programming lower bound Φ_{OPT}. InvCap is used in Cisco IOS 10.3 and later by default [6, 24]. GA is derived from the genetic algorithm in [7], which will be also compared with the hybrid genetic algorithm later in Section 6.4. LS is the implementation used in [10].

For each of the 13 networks, 12 increasingly loaded traffic demand matrices are considered. InvCap and LPLB were run a single time. Ten 1-hour runs were done with GA, HGA, and LS on each instance and average routing costs computed.

Tables 2 to 14 and Figures 6 to 18 summarize these results. For each demand level, the tables list the normalized costs for InvCap and LPLB as well as the average normalized costs over 10 1-hour runs for GA, HGA, and LS. The last row in each table lists the sum of the normalized average costs for

TABLE 2. Routing costs for `att` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
3761.179	1.013	1.000	1.000	1.000	1.00
7522.358	1.013	1.000	1.000	1.000	1.00
11,283.536	1.052	1.010	1.008	1.008	1.01
15,044.715	1.152	1.057	1.050	1.050	1.05
18,805.894	1.356	1.173	1.168	1.168	1.15
22,567.073	1.663	1.340	1.332	1.331	1.31
26,328.252	2.940	1.520	1.504	1.506	1.48
30,089.431	21.051	1.731	1.689	1.691	1.65
33,850.609	60.827	2.089	2.007	2.004	1.93
37,611.788	116.690	2.663	2.520	2.520	2.40
41,372.967	185.671	5.194	4.382	4.377	3.97
45,134.146	258.263	20.983	16.433	16.667	15.62
Total	652.691	40.760	35.093	35.322	33.57

Solutions are averaged over 10 1-hour runs.

TABLE 3. Routing costs for `hier50a` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
410.641	1.016	1.000	1.000	1.000	1.00
821.281	1.028	1.000	1.000	1.000	1.00
1231.922	1.056	1.011	1.010	1.010	1.01
1642.563	1.150	1.049	1.044	1.043	1.04
2053.204	2.345	1.116	1.107	1.106	1.10
2463.844	21.890	1.209	1.194	1.193	1.17
2874.485	37.726	1.328	1.302	1.307	1.27
3285.126	56.177	1.490	1.434	1.443	1.39
3695.766	75.968	1.771	1.603	1.644	1.53
4106.407	106.904	2.243	2.013	2.129	1.89
4517.048	140.516	5.273	3.674	3.975	3.44
4927.689	180.299	20.968	15.123	16.837	14.40
Total	626.075	39.458	31.504	33.687	30.24

Solutions are averaged over 10 1-hour runs.

TABLE 4. Routing costs for `hier50b` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
280.224	1.005	1.000	1.000	1.000	1.00
560.449	1.012	1.001	1.001	1.001	1.00
840.673	1.039	1.018	1.017	1.017	1.01
1120.898	1.110	1.058	1.054	1.054	1.03
1401.122	1.268	1.098	1.092	1.092	1.06
1681.346	6.281	1.146	1.137	1.137	1.11
1961.571	27.661	1.227	1.208	1.206	1.16
2241.795	44.140	1.352	1.319	1.316	1.24
2522.020	63.905	1.520	1.453	1.452	1.35
2802.244	95.131	1.875	1.718	1.691	1.47
3082.468	128.351	3.153	2.264	2.205	1.61
3362.693	159.848	12.318	4.221	4.166	1.83
Total	530.751	27.766	18.484	18.337	14.87

Solutions are averaged over 10 1-hour runs.

TABLE 5. Routing costs for `hier100` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
383.767	1.020	1.000	1.000	1.000	1.00
767.535	1.020	1.000	1.001	1.000	1.00
1151.303	1.030	1.006	1.007	1.005	1.01
1535.070	1.090	1.036	1.037	1.033	1.03
1918.838	1.170	1.083	1.078	1.076	1.06
2302.605	1.590	1.143	1.135	1.132	1.11
2686.373	8.870	1.234	1.220	1.217	1.20
3070.140	17.500	1.337	1.313	1.311	1.28
3453.908	24.930	1.602	1.557	1.558	1.52
3837.675	38.540	3.343	3.244	3.258	3.18
4221.443	70.250	11.976	11.812	11.823	11.71
4605.210	114.550	19.730	19.214	19.245	19.06
Total	281.560	45.490	44.618	44.658	44.16

Solutions are averaged over 10 1-hour runs.

TABLE 6. Routing costs for `hier100a` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
1033.879	1.170	1.001	1.006	1.000	1.00
2067.757	1.170	1.002	1.008	1.000	1.00
3101.636	1.190	1.012	1.018	1.005	1.00
4135.515	1.270	1.043	1.048	1.028	1.02
5169.394	1.390	1.098	1.091	1.069	1.06
6203.272	1.530	1.160	1.142	1.128	1.10
7237.151	2.390	1.279	1.221	1.208	1.16
8271.030	10.660	1.441	1.331	1.312	1.25
9304.909	24.770	1.696	1.518	1.483	1.38
10,338.788	53.240	2.536	2.063	2.077	1.76
11,372.667	112.110	8.123	5.846	5.568	4.48
12,406.545	181.100	23.401	15.169	18.547	13.32
Total	391.990	44.792	33.461	36.425	29.53

Solutions are averaged over 10 1-hour runs.

TABLE 7. Routing costs for `rand50` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
3523.431	1.000	1.000	1.000	1.000	1.00
7046.861	1.000	1.000	1.000	1.000	1.00
10,570.292	1.043	1.002	1.001	1.001	1.00
14,093.723	1.136	1.056	1.036	1.036	1.03
17,617.154	1.296	1.179	1.151	1.144	1.13
21,140.585	1.568	1.327	1.292	1.286	1.27
24,664.015	3.647	1.525	1.455	1.447	1.42
28,187.446	27.352	1.780	1.672	1.672	1.61
31,710.877	66.667	2.173	1.977	1.976	1.90
35,234.308	122.869	3.201	2.556	2.569	2.43
38,757.739	188.778	7.738	4.607	4.683	4.26
42,281.169	264.611	27.375	15.041	15.585	13.75
Total	680.967	50.356	33.788	34.399	31.8

Solutions are averaged over 10 1-hour runs.

TABLE 8. Routing costs for `rand50a` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
4463.462	1.000	1.000	1.000	1.000	1.00
8926.923	1.011	1.000	1.000	1.000	1.00
13,390.385	1.074	1.014	1.008	1.008	1.01
17,853.847	1.217	1.110	1.071	1.069	1.05
22,317.308	2.133	1.258	1.221	1.214	1.19
26,780.770	16.601	1.479	1.398	1.391	1.35
31,244.232	42.653	1.770	1.637	1.628	1.57
35,707.693	81.279	2.323	1.970	1.982	1.87
40,171.155	131.857	3.590	2.519	2.529	2.29
44,634.617	203.556	8.355	3.919	3.891	3.02
49,098.079	278.997	55.041	10.239	10.374	5.98
53,561.540	357.045	123.604	50.631	53.301	19.65
Total	1118.420	201.544	77.613	80.387	40.98

Solutions are averaged over 10 1-hour runs.

TABLE 9. Routing costs for `rand100` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
5774.737	1.000	1.005	1.006	1.000	1.00
11549.474	1.000	1.008	1.006	1.000	1.00
17,324.211	1.040	1.034	1.013	1.001	1.00
23,098.948	1.130	1.119	1.061	1.036	1.03
28,873.685	1.310	1.299	1.217	1.156	1.14
34,648.422	1.680	1.546	1.394	1.312	1.29
40,423.159	9.250	1.932	1.601	1.507	1.47
46,197.896	37.220	2.849	1.882	1.757	1.71
51,972.633	71.520	4.375	2.320	2.112	2.02
57,747.370	115.260	13.822	3.131	2.703	2.46
63,522.107	173.790	41.105	6.729	4.175	3.27
69,296.844	238.560	108.485	25.283	10.942	5.79
Total	652.760	179.579	47.643	29.701	23.18

Solutions are averaged over 10 1-hour runs.

TABLE 10. Routing costs for `rand100b` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
8382.862	1.000	1.015	1.011	1.000	1.00
16,765.725	1.000	1.023	1.011	1.000	1.00
25,148.587	1.020	1.063	1.013	1.000	1.00
33,531.449	1.090	1.172	1.041	1.011	1.01
41,914.312	1.220	1.385	1.186	1.093	1.07
50,297.174	1.400	1.698	1.358	1.236	1.20
58,680.037	1.960	2.665	1.540	1.407	1.36
67,062.899	8.130	4.890	1.773	1.605	1.54
75,445.761	20.510	12.206	2.170	1.851	1.76
83,828.624	48.850	30.794	2.788	2.286	2.10
92,211.486	94.050	77.555	5.179	3.151	2.78
100,594.349	155.680	154.880	14.857	7.029	5.87
Total	335.910	290.346	34.927	23.669	21.69

Solutions are averaged over 10 1-hour runs.

TABLE 11. Routing costs for `wax50` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
2117.622	1.000	1.000	1.000	1.000	1.00
4235.244	1.000	1.000	1.000	1.000	1.00
6352.865	1.015	1.000	1.000	1.000	1.00
8470.487	1.080	1.023	1.018	1.017	1.02
10,588.109	1.171	1.100	1.088	1.086	1.08
12,705.731	1.316	1.205	1.188	1.183	1.18
14,823.353	1.605	1.339	1.315	1.309	1.29
16,940.975	3.899	1.531	1.483	1.475	1.45
19,058.596	20.630	1.798	1.756	1.754	1.72
21,176.218	45.769	2.498	2.373	2.361	2.31
23,293.840	84.409	6.280	5.988	5.998	3.27
25,411.462	139.521	13.633	12.849	12.883	4.36
Total	302.415	33.407	32.058	32.066	20.68

Solutions are averaged over 10 1-hour runs.

TABLE 12. Routing costs for `wax50a` with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
3287.217	1.000	1.000	1.000	1.000	1.00
6574.434	1.000	1.000	1.000	1.000	1.00
9861.651	1.002	1.000	1.000	1.000	1.00
13,148.868	1.049	1.010	1.009	1.009	1.01
16,436.085	1.129	1.050	1.031	1.028	1.03
19,723.302	1.230	1.137	1.108	1.103	1.09
23,010.519	1.393	1.250	1.218	1.210	1.19
26,297.735	1.634	1.398	1.357	1.349	1.32
29,584.952	2.706	1.593	1.514	1.510	1.48
32,872.169	12.816	1.980	1.885	1.875	1.83
36,159.386	38.708	4.042	3.874	3.870	2.84
39,446.603	78.084	11.588	11.288	11.281	4.30
Total	141.751	28.048	27.284	27.235	19.09

Solutions are averaged over 10 1-hour runs.

TABLE 13. Routing costs for wax100 with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
4039.477	1.000	1.005	1.006	1.000	1.00
8078.954	1.000	1.006	1.005	1.000	1.00
12,118.431	1.010	1.010	1.007	1.002	1.00
16,157.908	1.030	1.029	1.014	1.006	1.01
20,197.385	1.090	1.065	1.029	1.012	1.01
24,236.863	1.240	1.136	1.075	1.048	1.04
28,276.340	4.660	1.268	1.186	1.130	1.11
32,315.817	12.370	1.435	1.324	1.247	1.23
36,355.294	23.020	1.835	1.698	1.599	1.57
40,394.771	32.230	4.807	4.501	4.391	4.36
44,434.248	40.640	8.953	8.317	8.191	8.14
48,473.725	54.560	12.647	11.575	11.409	11.35
Total	173.850	37.196	34.737	34.035	33.82

Solutions are averaged over 10 1-hour runs.

TABLE 14. Routing costs for wax100a with scaled projected demands.

Demand	InvCap	GA	HGA	LS	LPLB
5291.092	1.000	1.011	1.011	1.000	1.00
10,582.185	1.000	1.012	1.011	1.000	1.00
15,873.277	1.010	1.022	1.011	1.001	1.00
21,164.370	1.060	1.064	1.031	1.015	1.02
26,455.462	1.130	1.138	1.064	1.036	1.03
31,746.555	1.250	1.259	1.169	1.105	1.09
37,037.647	1.490	1.458	1.313	1.218	1.19
42,328.740	3.380	1.679	1.468	1.354	1.32
47,619.832	18.080	2.224	1.732	1.596	1.54
52,910.925	38.940	3.832	2.682	2.530	2.41
58,202.017	69.400	12.132	9.998	10.436	9.62
63,493.110	103.430	26.675	20.213	19.775	19.49
Total	241.170	54.506	43.703	43.066	41.71

Solutions are averaged over 10 1-hour runs.

each algorithm. Normalized cost values less than 10(2/3) (i.e., recall that when the routing cost exceeds 10(2/3) we say that the routing congests the network; see Section 2) are separated from those for which the network is congested by a line segment in the tables. The distribution of the costs can be seen in the figures, where all 10 cost values for each algorithm and each demand point are plotted together with the average costs.

We make the following remarks about the computational results. The pure genetic algorithm (GA) consistently found better solutions than InvCap. Solution differences increased with traffic intensity. The relative differences $|\Phi^*_{\text{InvCap}} - \Phi^*_{\text{GA}}|/\Phi^*_{\text{InvCap}}$ between the solution values obtained by GA and InvCap varied from 13.6% on rand100b to 94.8% on hier50b.

HGA found solutions at least as good as GA on all networks and for all demand levels. Solution differences increased with traffic intensity. The relative differences $|\Phi^*_{\text{GA}} - \Phi^*_{\text{HGA}}|/\Phi^*_{\text{GA}}$ varied from as little as 1.9% on hier100 to as large as 88.0% on rand100b. HGA not only found better-quality solutions, but did so in less CPU time (see Figs. 19 and 20, which compare one run of HGA and GA each on the att network with demand $D = 45134.146$). Figure 19 shows the value of the best-quality solution in the population as a function of CPU time, while Figure 20 shows the value of the best-quality solution in the population as a function of the generation of the algorithm. These figures illustrate how close to the LP lower bound the HGA comes and how much faster HGA is to converge compared to GA.

Of the 13 network classes, HGA found the best average solutions in seven classes, while LS found the best in the remaining six. On classes rand100 and rand100b, solution values found by LS were 37.7 and 32.2% smaller with

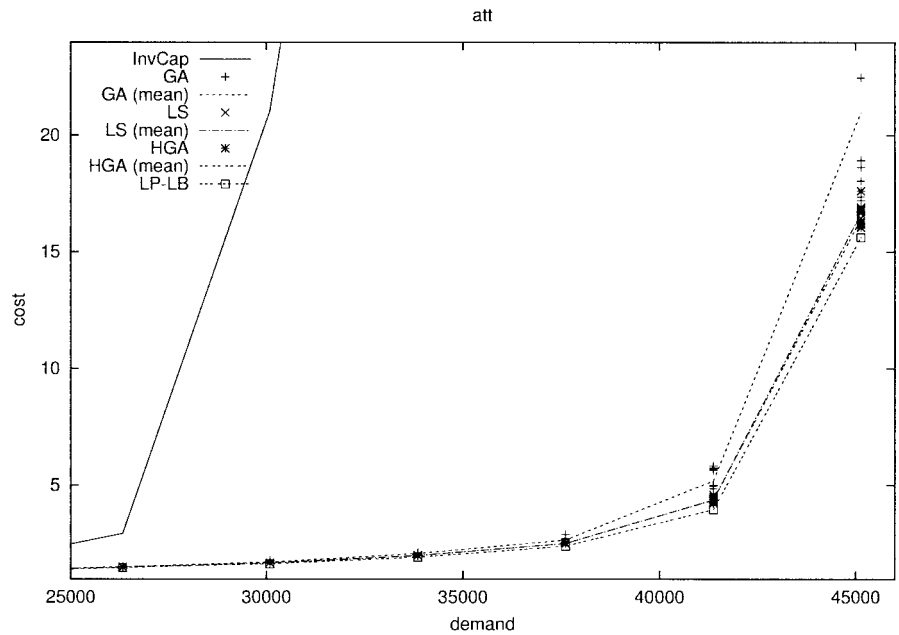

FIG. 6. InvCap, GA, HGA, LS, and LP lower bound on att.

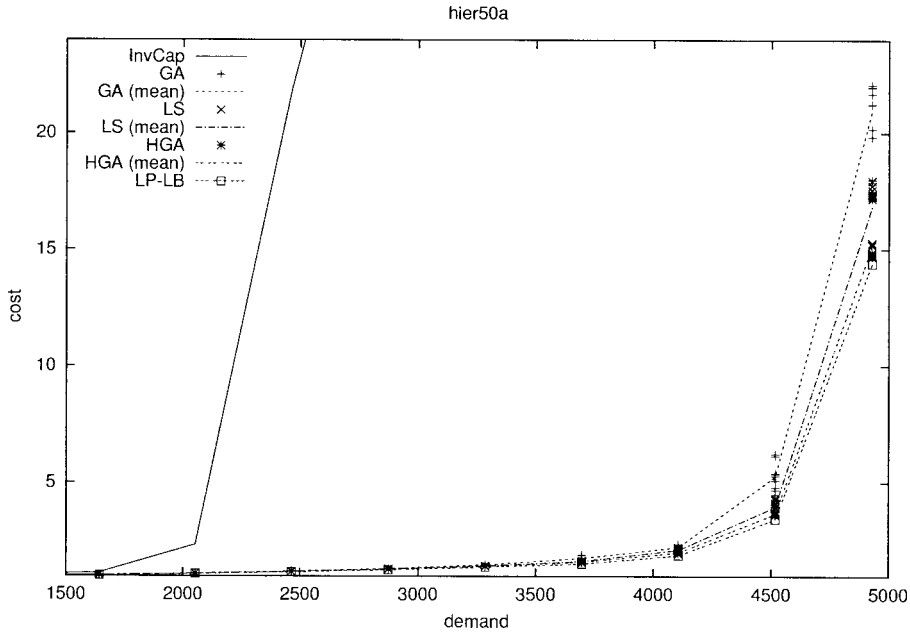

FIG. 7. InvCap, GA, HGA, LS, and LP lower bound on `hier50a`.

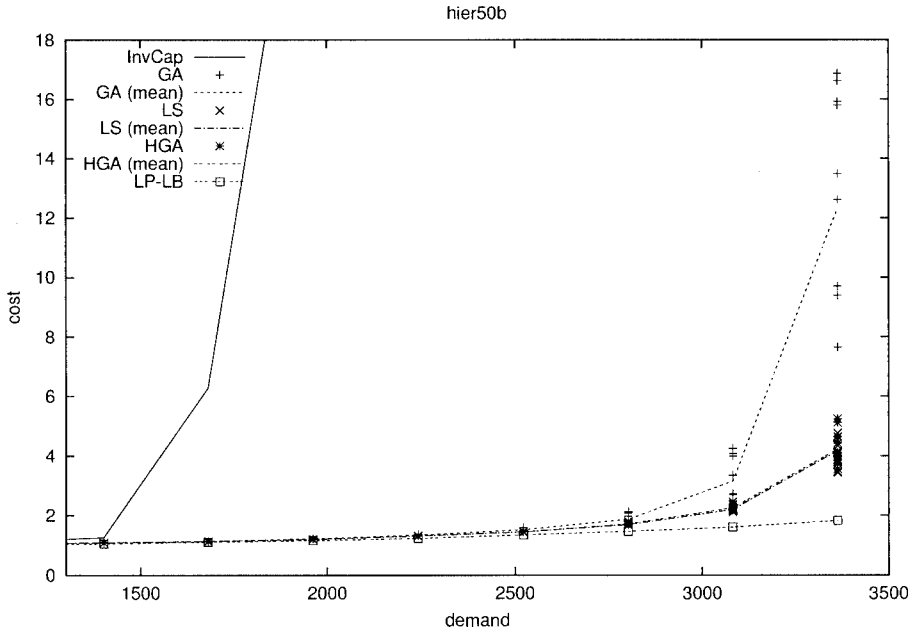

FIG. 8. InvCap, GA, HGA, LS, and LP lower bound on `hier50b`.

respect to those found by HGA. On the other hand, on classes `hier50a` and `hier100a`, solution values found by HGA were 6.5 and 8.1% smaller with respect to those found by LS. On the remaining nine network instances, the relative difference between average solution values found by HGA and LS was small, varying from 0.09 to 3.5%.

HGA and LS on average found solution values that varied from 0.6 to 105.5% of the linear programming lower bound. HGA found solution values with a relative gap of less than 5% for five of the 13 network classes, while LS did so for three classes. Both HGA and LS found solution values with less than 10% relative gap on six classes.

6.3. Distribution of Time-to-Target-Solution-Value

To study and compare the computation times, we used the methodology proposed by Aiex et al. [1] and further explored by Resende and Ribeiro, for example, in [20].

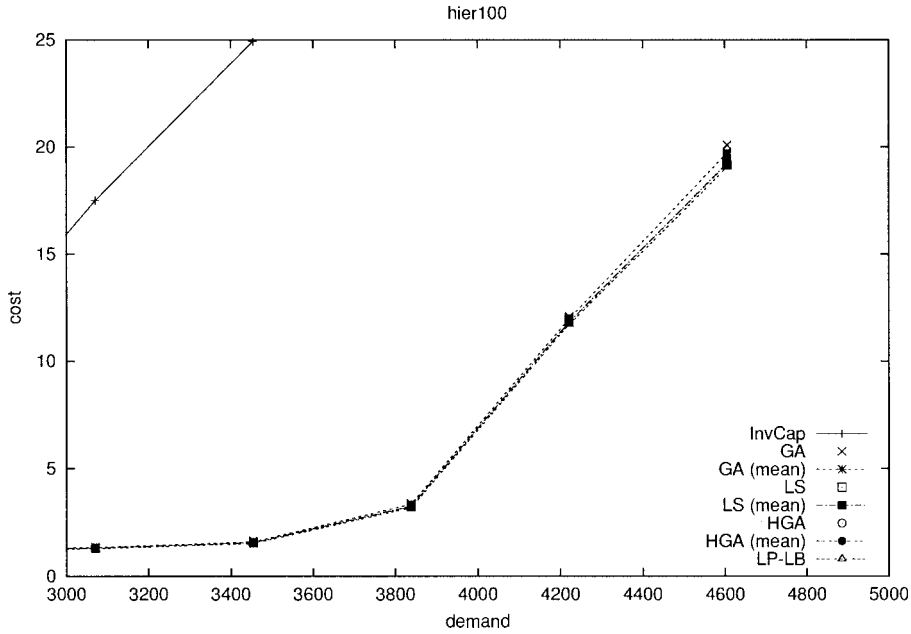

FIG. 9. InvCap, GA, HGA, LS, and LP lower bound on `hier100`.

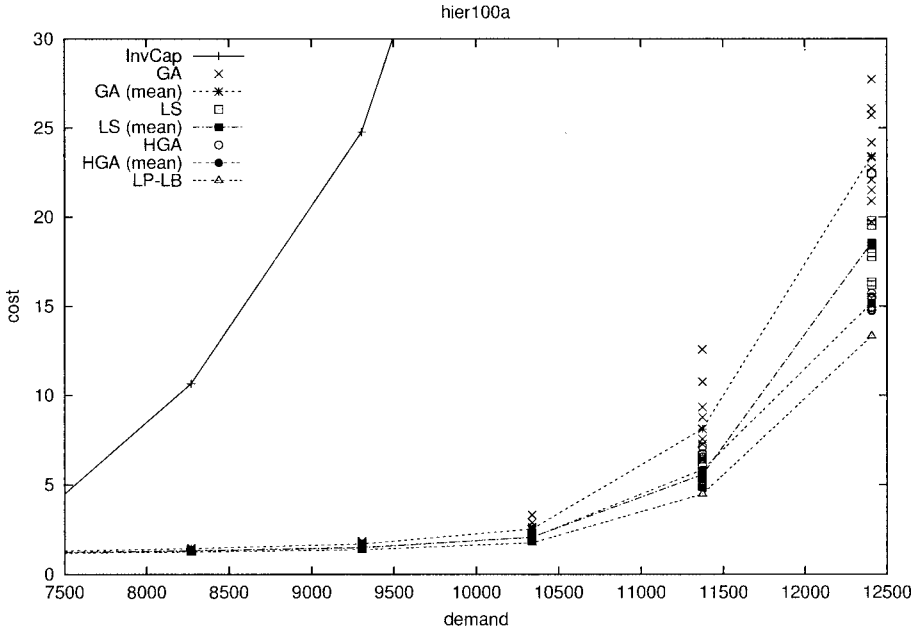

FIG. 10. InvCap, GA, HGA, LS, and LP lower bound on `hier100a`.

Without loss of generality, we considered networks `att` with the demand equal to 37,611.788, `hier50a` with the demand equal to 41,06.407, and `rand50` with the demand equal to 35,234.308 to illustrate the general behavior observed for most instances. For each of them, we performed 100 independent runs with different seeds of each algorithm GA, HGA, and LS, considering a given parameter value `look4`. Each execution was terminated when a solution of value less than or equal to the target value `look4` was found or when the time limit of one hour was reached. Three different values (corresponding to easy, medium, and difficult cases) of `look4` were investigated for each network: 2.89, 2.77, and 2.64 for network `att`, 2.32, 2.21, and 2.11 for network `hier50a`, and 2.93, 2.81, and 2.68 for network `rand50`. Different target values were used because the networks and demands were different. Empirical probability distributions for the time-to-target solution value are plotted in Figures 21 to 24. Runs that failed to find a solution of value less than or equal to the target value `look4` within the 1-hour time limit were discarded in these plots. To plot the empirical distribution for each algorithm and each instance, we associate with the i-th smallest running time t_i a probability

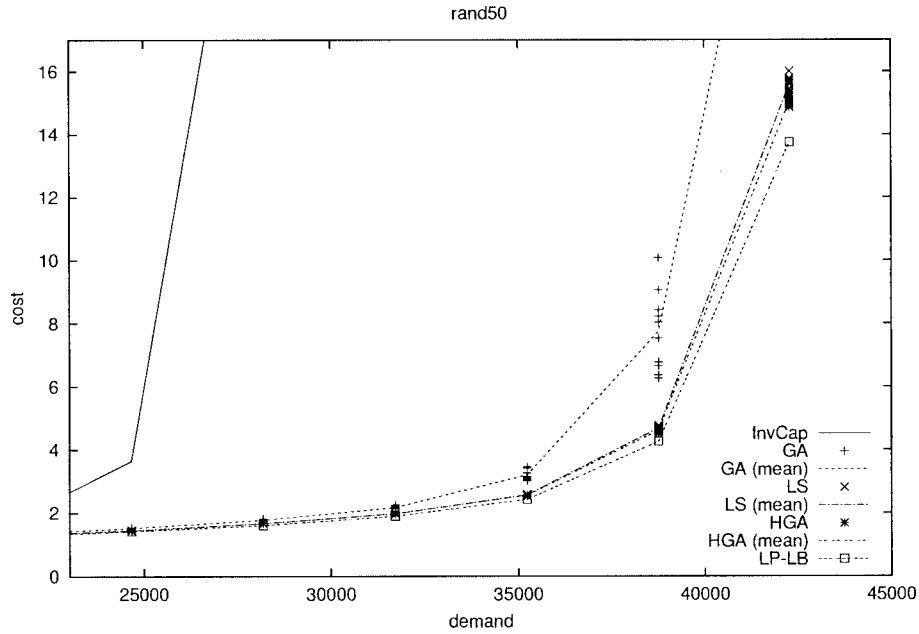

FIG. 11. InvCap, GA, HGA, LS, and LP lower bound on `rand50`.

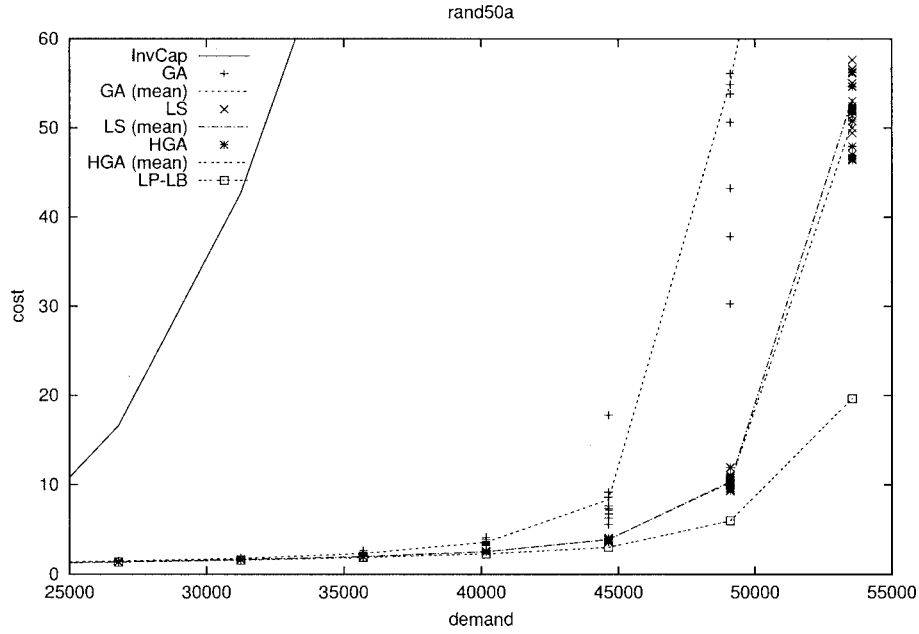

FIG. 12. InvCap, GA, HGA, LS, and LP lower bound on `rand50a`.

$p_i = (i - (1/2))/100$, and plot the points $z_i = (t_i, p_i)$, for $i = 1, \ldots, n_r$, where $n_r \leq 100$ is the number of runs which found a solution of value less than or equal to the target value `look4` within the 1-hour time limit.

HGA and LS found solutions with value less than or equal to the target in all runs associated with network `att` (Figs. 21 and 22). GA failed to find solutions at least as good as the target value on eight runs with the easiest target, 19 runs with the medium target, and 59 runs with the hardest target. HGA is not only much faster than GA, but also the computation times of the former are more predictable than those of the latter. As we can see from Figure 21, in many runs of GA the computation times are several orders of magnitude larger than those of HGA. Considering Figure 22, we notice that the computation times of HGA are more predictable than those of LS. The latter are several times larger than the former in many runs. In more than 30% of the runs LS encounters difficulties to converge and requires very long computation times. The figure seems to suggest that this is related to the fact that LS frequently gets stuck at a local minimum and the escape mechanism often needs to be applied repeatedly to succeed.

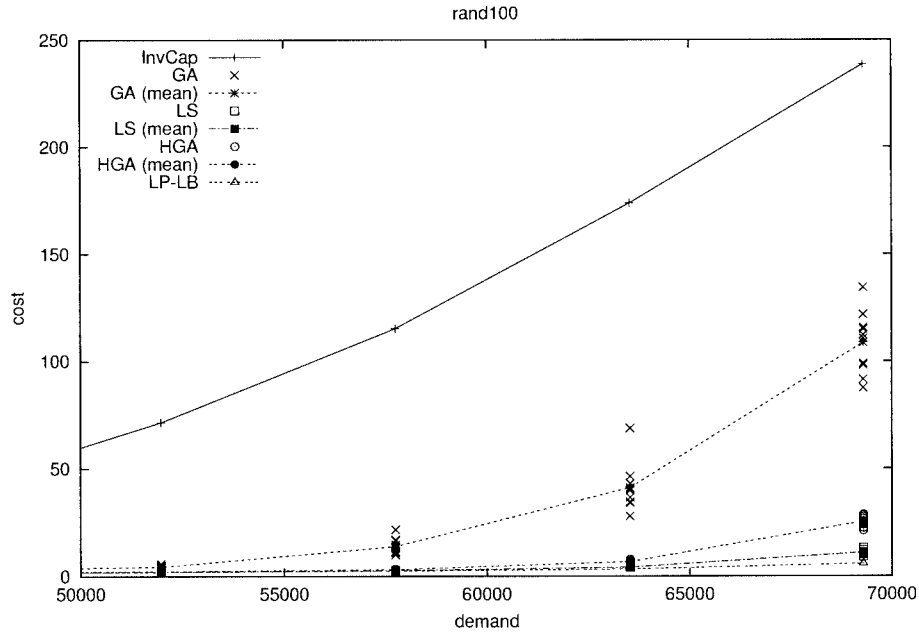

FIG. 13. InvCap, GA, HGA, LS, and LP lower bound on `rand100`.

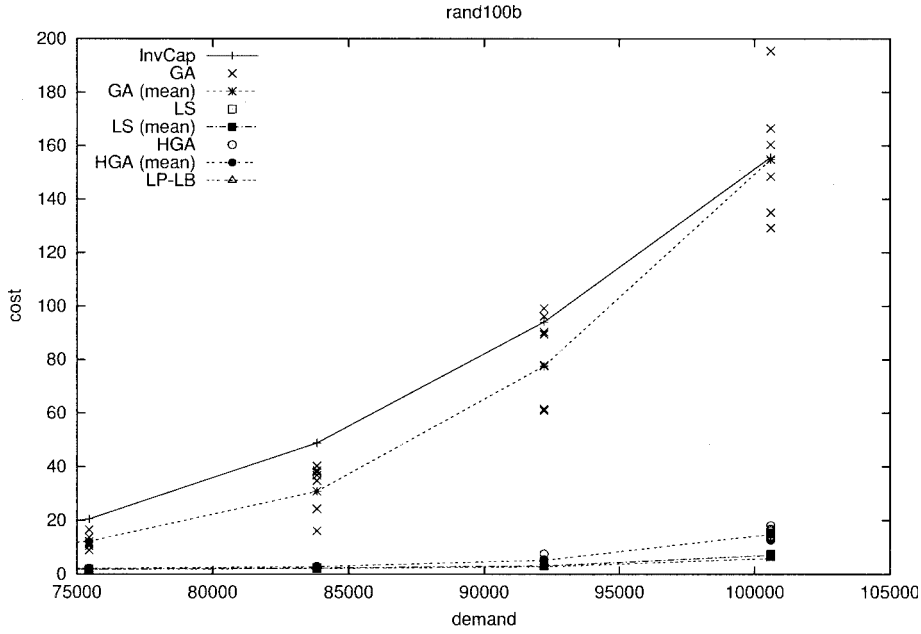

FIG. 14. InvCap, GA, HGA, LS, and LP lower bound on `rand100b`.

In the case of network `hier50a` (Figs. 23 and 24), HGA performed much better than GA, as we can see from Figure 23. HGA found solutions at least as good as the target within the 1-hour time limit in all runs for the easy and medium target values, and in 94 out of the 100 runs for the more difficult target. Contrarily, GA rarely found solutions at least as good as the target: 44 times for the easy target, 14 for the medium target, and only once in the hardest case (which cannot be plotted in the figure). HGA also did better than LS for this network, as shown in Figure 24. LS found solutions at least as good as the target in all runs for the easier target value. However, it failed in two runs with the medium target. For the harder target value, it missed the target in 48 runs.

The instances associated with random graphs are among the hardest ones for the genetic algorithms. Network `rand50` is particularly difficult for GA, which failed to find a solution at least as good as the target within the 1-hour time limit in all 300 runs. Contrarily, HGA found solutions at least as good as the target in all runs. Although HGA performs better than LS for many classes of instances (as illustrated in Figs. 22

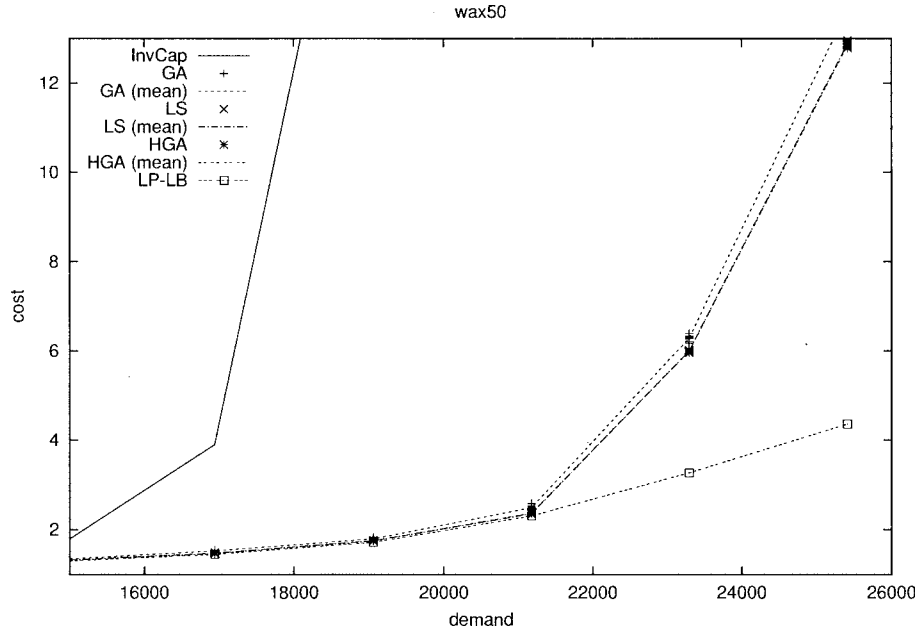

FIG. 15. InvCap, GA, HGA, LS, and LP lower bound on `wax50`.

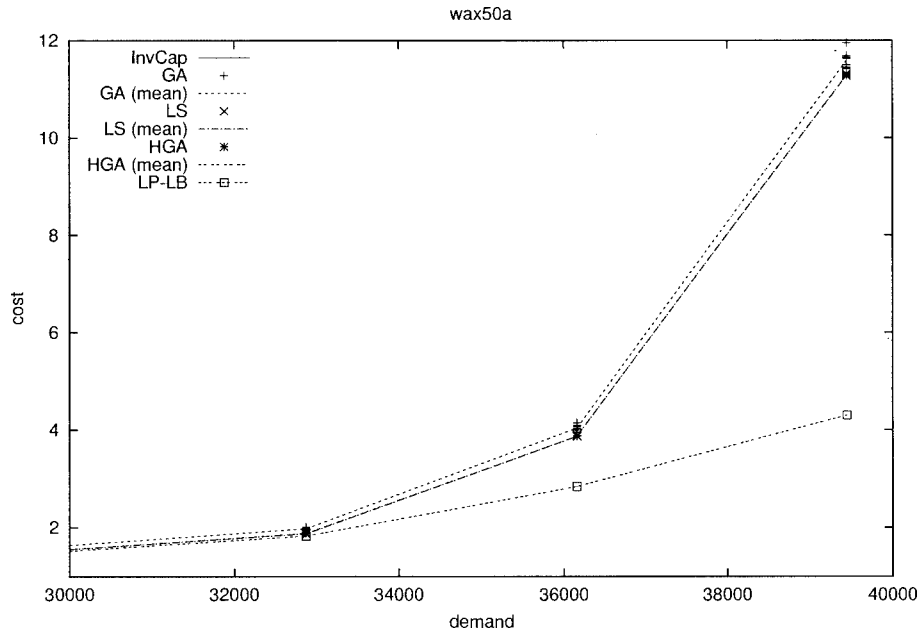

FIG. 16. InvCap, GA, HGA, LS, and LP lower bound on `wax50a`.

and 24), LS did better in this case. Both algorithms are robust and found solutions at least as good as the target in all runs, but LS was usually faster.

6.4. Comparison with the Original GA

We conclude the experimental results section by comparing the GA used in the experiments described above with the GA presented in [7] to show that both implementations produce comparable results. Consequently, the conclusions we make regarding the comparison of HGA and GA should carry over to the GA of [7], which we refer to as GA^0.

Although GA and GA^0 are based on the same C code, they differ in some aspects:

- GA uses a population of size 50, while GA^0 uses one of size 200. Note that the size 200 population was used in [7] for the instances in this experiment. For larger instances, the experiments in [7] used a population of size 100.
- GA randomly generates an initial population with weights in the interval [1, 7], while GA^0 randomly generates all but

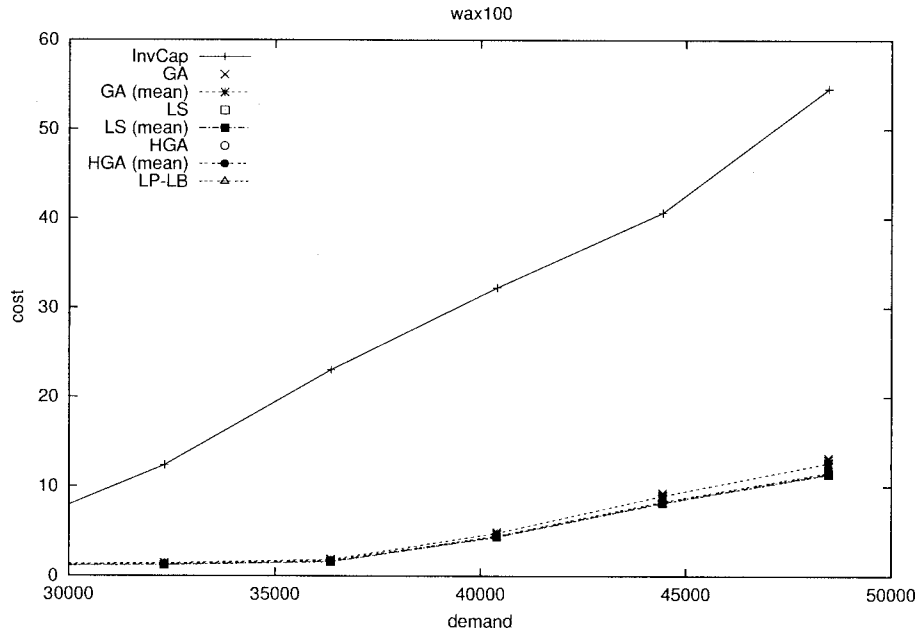

FIG. 17. InvCap, GA, HGA, LS, and LP lower bound on `wax100`.

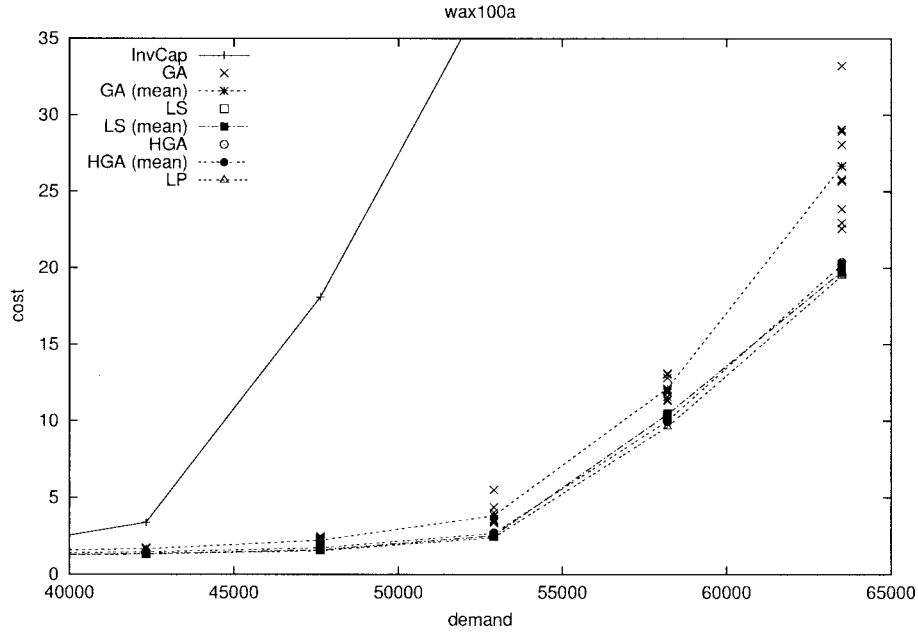

FIG. 18. InvCap, GA, HGA, LS, and LP lower bound on `wax100a`.

one element of the initial population with weights in the interval [1, 20] and completes the remaining element of the population with the weights corresponding to the heuristic InvCap.

- Define the population parameters $\alpha = |\mathcal{A}|/|\mathcal{A} \cup \mathcal{B} \cup \mathcal{C}|$ and $\beta = |\mathcal{C}|/|\mathcal{A} \cup \mathcal{B} \cup \mathcal{C}|$. GA uses population parameters $(\alpha, \beta) = (.25, .05)$ while GA^0 uses population parameters $(\alpha, \beta) = (-.20, -.10)$.

We ran both genetic algorithm implementations for 10 independent 1-hour trials on three networks, `att`, `hier50a`, and `rand50`, with two demand matrices each. We used the matrices with the two largest demands in the experiments described earlier in this section. For network `att`, we use demands 41,372.967 and 45,134.146. For network `hier50a`, we use demands 45,17.048 and 49,27.689. For network `rand50`, we use demands 38,757.739 and 42,281.169.

For GA, we used the parameter setting described earlier in this section. For GA^0, we used the parameter setting described in [7].

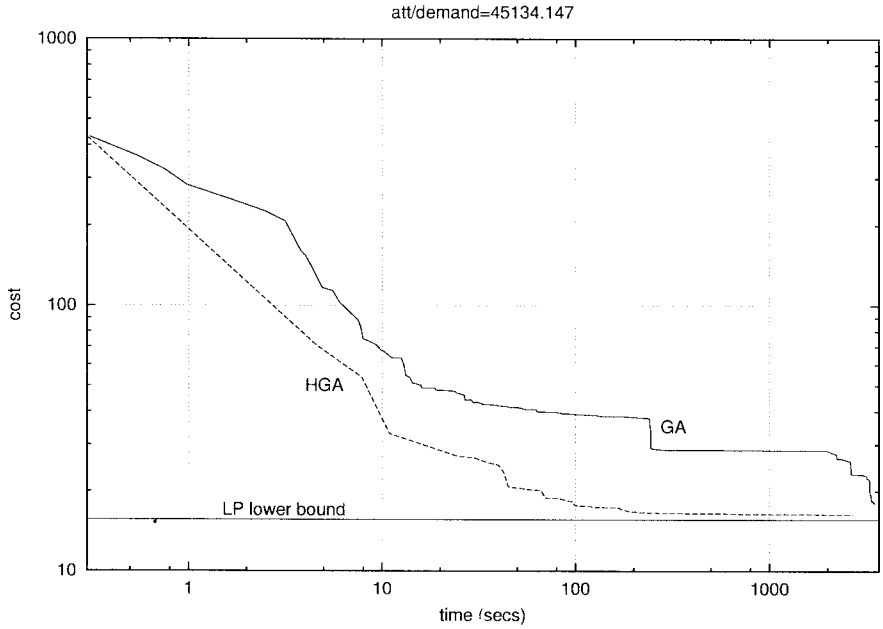

FIG. 19. Cost as a function of time on 1-hour run: HGA versus GA on `att` with demand 45,134.146.

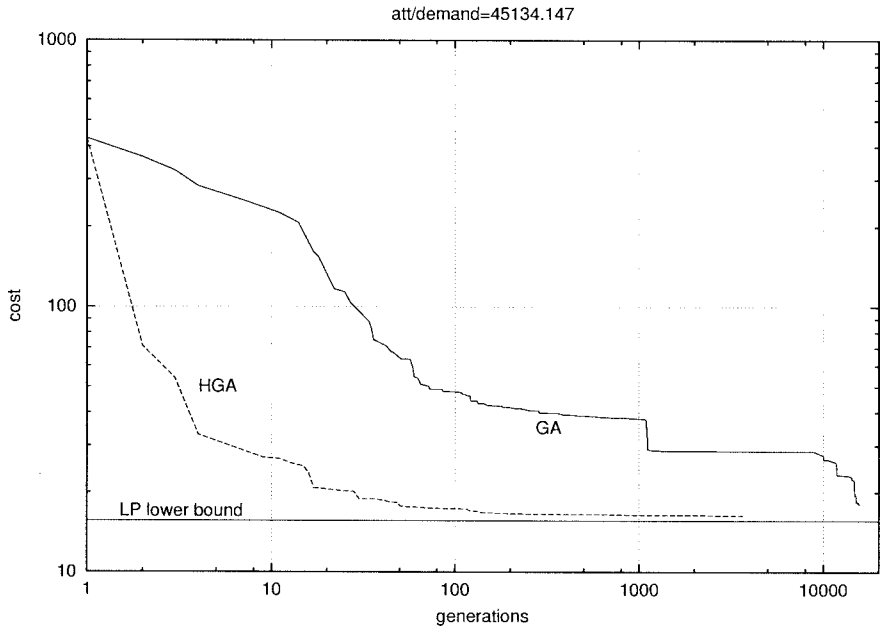

FIG. 20. Cost as a function of generations on 1-hour run: HGA versus GA on `att` with demand 45,134.146.

Table 15 shows minimum, average, and maximum final solution values for 10 1-hour runs for HGA, GA, and GA^0. The last row of the table lists the sums of each column.

The experimental results clearly show that HGA finds better minimum, average, and maximum solution values than GA and GA^0 for all networks and demand levels. Overall, GA does better than GA^0, finding on average solutions that are 48% smaller than those found by GA^0. This big difference is mainly due to GA^0's poor performance on network `rand50`. Discarding GA^0's results on network `rand50`, GA and GA^0 are comparable, with GA finding solution that are on average 3.5% larger than those found by GA^0. On the other hand, with the exception of network `att` with demand 41,372.968, the minimum solution value found by GA was always smaller than the one found by GA^0.

7. CONCLUDING REMARKS

We presented a new hybrid genetic algorithm (HGA) for solving the OSPF weight-setting problem, combining the

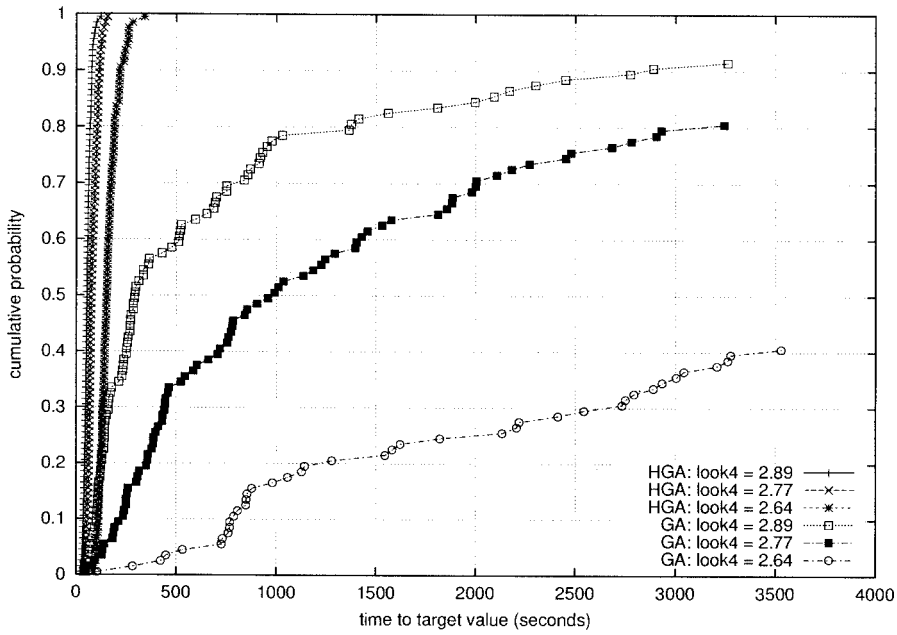

FIG. 21. Time-to-target solution value: HGA versus GA on network `att` with demand 37,611.788.

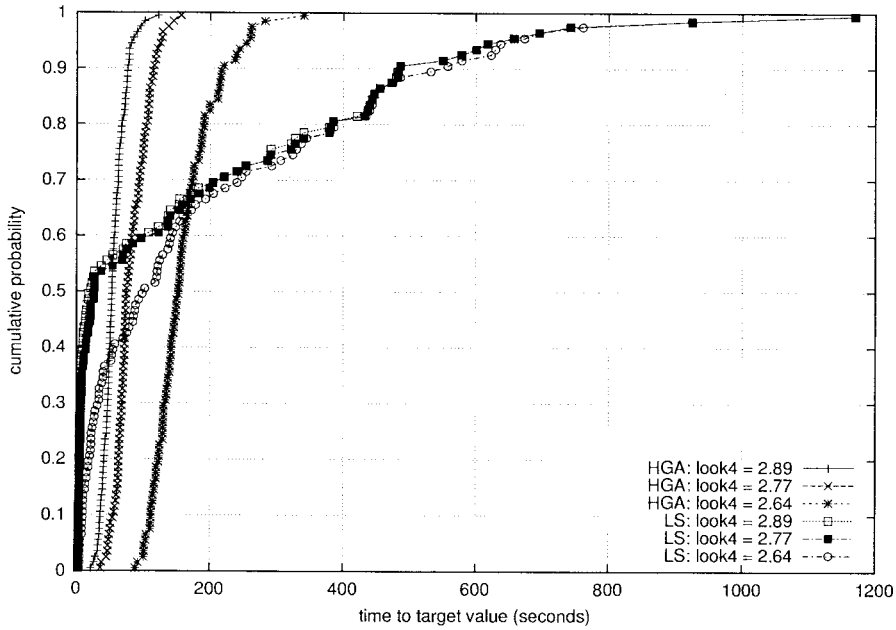

FIG. 22. Time-to-target solution value: HGA versus LS on network `att` with demand 37,611.788.

traditional genetic algorithm (GA) strategy with a local search procedure to improve the solutions obtained by crossover.

The local search procedure uses small neighborhoods, and is based on the fast computation of dynamic shortest paths. Because it considers only unit weight increments with respect to the weights in the current solution, our implementation of algorithm `UpdateShortestPaths` is two to three times faster than its original implementation. This specialization accounts significantly to speed up the implementation of the hybrid genetic algorithm.

The new heuristic performs systematically better than the genetic algorithm without local search (GA and GA^0).

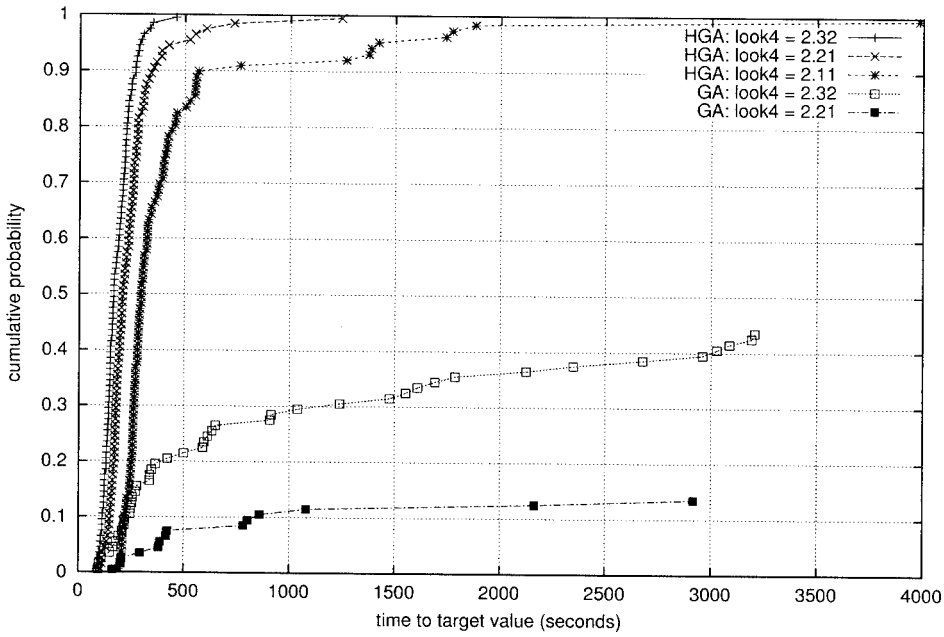

FIG. 23. Time-to-target solution value: HGA versus GA on network `rand50` with demand 35,234.308.

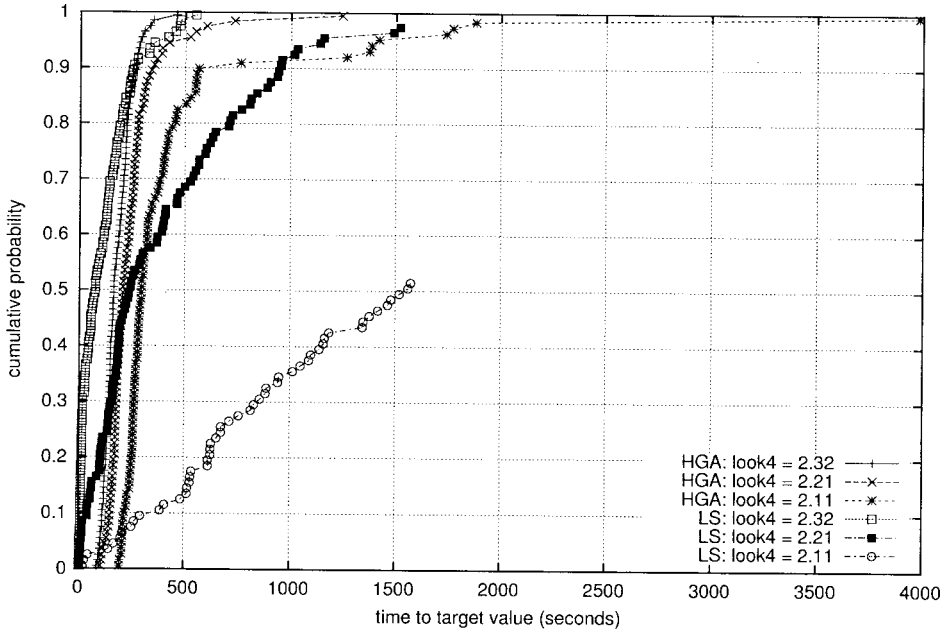

FIG. 24. Time to target solution value: HGA versus LS on network `hier50a` with demand 4106.407.

HGA finds better solutions in substantially less computation times. The experimental results also showed that it is also more robust, in the sense that it rarely gets stuck in suboptimal local minima, while the genetic algorithm often does so.

We also compared the new algorithm with a local search heuristic (LS). Once again, HGA is more robust than LS. Algorithms HGA and LS are competitive in terms of solution quality and time. HGA is better than LS for some classes of test problems, while LS is better for others. Moreover, the implementation of LS is based on limited-size hashing tables that limits the number of iterations it can perform and, consequently, the solution quality that can be obtained for larger problems.

TABLE 15. Experiments comparing GA and HGA with the genetic algorithm GA^0 from [7].

		HGA			GA			GA^0		
Network	Demand	min	avg	max	min	avg	max	min	avg	max
att	41,372.967	4.227	4.382	6.604	4.624	5.194	5.833	4.469	5.034	7.517
att	45,134.146	16.043	16.433	17.622	17.210	20.983	26.802	17.551	20.017	24.244
hier50a	4517.048	3.616	3.674	3.978	4.501	5.273	6.235	4.672	5.045	6.206
hier50a	4927.689	14.695	15.123	17.184	17.347	20.968	25.252	18.489	20.461	23.309
rand50	38,757.739	4.503	4.607	4.669	6.259	7.738	10.073	22.599	25.623	30.404
rand50	42,281.169	14.852	15.041	15.331	20.165	27.375	33.994	80.535	93.876	98.750
	Sum	55.936	59.260	65.388	70.106	87.531	109.189	148.315	170.056	190.430

Ten 1-hour runs were done for each algorithm and minimum, average, and maximum final solution values found are listed for each algorithm on each instance.

We are currently working on a parallel implementation of HGA for clusters using the Message Passing Interface (MPI). Encouraging preliminary computational results have shown significant reductions in elapsed times in the time-to-target solution value.

REFERENCES

[1] R.M. Aiex, M.G.C. Resende, and C.C. Ribeiro, Probability distribution of solution time in GRASP: An experimental investigation, J Heuristic, 8 (2002), 343–373.

[2] J.C. Bean, Genetic algorithms and random keys for sequencing and optimization, ORSA J. Comp. 6 (1994), 154–160.

[3] A. Bley, M. Grötchel, and R. Wessläy, Design of broadband virtual private networks: Model and heuristics for the B-WiN, Technical Report SC 98-13, Konrad-Zuse-Zentrum fur Informationstecknik Berlin, 1998. (To appear in Proc. DIMACS Workshop on Robust Communication Network and Survivability, AMS-DIMACS Series).

[4] L.S. Buriol, M.G.C. Resende, and M. Thorup, Speeding up dynamic shortest path algorithms, Technical report, AT&T Labs Research, Florham Park, NJ, 2003.

[5] K. Calvert, M. Doar, and E.W. Zegura, Modeling internet topology, IEEE Commun Magazine, Indianapolis, IN, 35 (1997), 160–163.

[6] Cisco, Configuring OSPF, Cisco Press, Indianapolis, IN, 1997.

[7] M. Ericsson, M.G.C. Resende, and P.M. Pardalos, A genetic algorithm for the weight setting problem in OSPF routing, J Combin Optimizat 6 (2002), 299–333.

[8] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, Deriving traffic demands for operational IP networks: Methodology and experience, IEEE/ACM Trans Networking 9 (2001), 265–279.

[9] B. Fortz, J. Rexford, and M. Thorup, Traffic engineering with traditional IP routing protocols, IEEE Commun Magazine October (2002), 118–124.

[10] B. Fortz and M. Thorup, Increasing internet capacity using local search, Comput Optimizat Applicat 29 (2004), 13–48 [preliminary short version of this paper published as "Internet Traffic Engineering by Optimizing OSPF weights," in Proc. 19th IEEE Conf. on Computer Communications (INFOCOM 2000)].

[11] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone, Experimental analysis of dynamic algorithms for the single source shortest path problem, ACM J Exp Alg 3 (1998) article 5.

[12] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully dynamic output-bounded single source shortest-paths problem," Proc. ACM-SIAM Symp. Discrete Algorithms, 1996, pp. 212–221.

[13] Internet Engineering Task Force, Ospf version 2, Technical Report RFC 1583, Network Working Group, 1994.

[14] F. Lin and J. Wang, "Minimax open shortest path first routing algorithms in networks supporting the smds services," Proc. IEEE International Conference on Communications (ICC), 1993, vol. 2, pp. 666–670.

[15] M. Matsumoto and T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, ACM Trans Modeling Comput Simulat 8 (1998), 3–30.

[16] J.T. Moy, OSPF, Anatomy of an Internet Routing Protocol, Addison-Wesley, Reading. MA, 1998.

[17] M. Pióro, Á. Szentesi, J. Harmatos, A. Jüttner, P. Gajowniczek, and S. Kozdrowski, On open shortest path first related network optimisation problems, Performance Evaluat 48 (2002), 201–223.

[18] K.G. Ramakrishnan and M.A. Rodrigues, Optimal routing in shortest-path data networks, Bell Labs Tech 6 (2001), 117–138.

[19] G. Ramalingam and T. Reps, An incremental algorithm for a generalization of the shortest-path problem. J Algorithms 21 (1996), 267–305.

[20] M.G.C. Resende and C.C. Ribeiro, "Greedy randomized adaptive search procedures," Handbook of metaheuristics, F. Grover and G. Kochen Gerger (Editors), Kluwer Academic Publishers, Dordrecht, 2003, pp. 219–249.

[21] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, Measuring ISP topologies with Rocketfuel, IEEE/ACM Trans Networking 12 (2004), 2–16.

[22] A. Sridharan, R. Guérin, and C. Diot, Achieving near-optimal traffic engineering solutions for current OSPF/IS-IS networks, Sprint ATL Technical Report TR02-ATL-022037, Sprint Labs, 2002.

[23] L. Subramanian, S. Agarwal, J. Rexford, and R.H. Katz, "Characterizing the Internet hierarchy from multiple vantage points," Proc. 21st IEEE Conf. on Computer Communications (INFOCOM 2002), 2002, vol. 2, pp. 618–627.

[24] T.M. Thomas II, OSPF network design solutions, Cisco Press, Indianapolis, IN, 1998.

[25] B.M. Waxman, Routing of multipoint connections, IEEE J Selected Areas Commun (Special Issue on Broadband Packet Communication) 6 (1998), 1617–1622.

[26] 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.

[27] E.W. Zegura, K.L. Calvert, and S. Bhattacharjee, "How to model an internetwork," Proc. 15th IEEE Conf. on Computer Communications (INFOCOM), 1996, pp. 594–602.