Survivable IP Network Design with OSPF Routing

L.S. Buriol
Computer Science Department, Federal University of Rio Grande do Sul, Av. Bento Gonçalves, 9500, 91501-970 Porto Alegre, RS, Brazil

M.G.C. Resende and M. Thorup
Internet and Network Systems Research Center, AT&T Labs Research, 180 Park Avenue, Florham Park, NJ 07932

Internet protocol (IP) traffic follows rules established by routing protocols. Shortest path-based protocols, such as Open Shortest Path First (OSPF), direct traffic based on arc weights assigned by the network operator. Each router computes shortest paths and creates destination tables used for routing flow on the shortest paths. If a router has multiple outgoing links on shortest paths to a given destination, it splits traffic evenly over these links. It is also the role of the routing protocol to specify how the network should react to changes in the network topology, such as arc or router failures. In such situations, IP traffic is rerouted through the shortest paths not traversing the affected part of the network. This article addresses the issue of assigning OSPF weights and multiplicities to each arc, aiming to design efficient OSPF-routed networks with minimum total weighted multiplicity (multiplicity multiplied by the arc length) needed to route the required demand and handle any single arc or router failure. The multiplicities are limited to a discrete set of values, and we assume that the topology is given. We propose an evolutionary algorithm for this problem, and present results applying it to several real-world problem instances. © 2006 Wiley Periodicals, Inc. NETWORKS, Vol. 49(1), 51–64 2007

Keywords: Internet; OSPF routing; network design; survivability; evolutionary algorithm

1. INTRODUCTION

The Internet is the global network of interconnected communication networks, made up of routers and links connecting the routers. On a network level, the Internet is built up of several autonomous systems (ASs) that typically fall under the administration of a single institution, such as a company, a university, or a service provider. Routing within a single AS is done by an Interior Gateway Protocol (IGP), while an Exterior Gateway Protocol (EGP) is used to route traffic flow between ASs. IP traffic is routed in small chunks called packets. A routing table instructs the router how to forward packets. Given a packet with an IP destination address in its header, the router retrieves from the table the IP address of the packet's next hop.

OSPF (Open Shortest Path First) is the most used IGP. For this protocol, an integer weight must be assigned to each arc, and the entire network topology and weights are known to each router. With this information, each router computes the graphs of shortest (weight) paths from each other router in the AS to itself. The graphs need not be trees, because all shortest paths between two routers need to be considered. Demands are routed on the corresponding shortest path graphs. At each router s, the total demand leaving this node and destined to a target router t is evenly split among all links outgoing from router s on the shortest path graphs ending at t. This demand not only consists of demand originating at s, but also of demand passing through s on its way to t.

The arcs weights are assigned by the AS operator. The lower the weight, the greater the chance that traffic will get routed on that arc. Different weight settings lead to different traffic flow patterns. Weights can be set to optimize network performance, such as to minimize congestion [5,8,9], as well as to minimize network design cost. In this article, we address the latter case.

Given a network topology and predicted traffic demands, the OSPF network design problem is to find a set of OSPF weights that minimizes network cost. More precisely, we are given a directed network $G = (N, A)$, where N is the set of routers and A is the set of potential arcs where capacity can be installed, and a demand matrix D that, for each pair $(s, t) \in N \times N$, specifies the demand D_s^t between s and t.

Received September 2004; accepted November 2005
Correspondence to: M.G.C. Resende; e-mail: mgcr@research.att.com
Contract grant sponsor: Brazilian National Council for Scientific and Technological Development (CNPq) (to L.S.B.).
Contract grant sponsor: AT&T Labs Research (to L.S.B.).
Contract grant sponsor: The European project "Coevolution and Self-organization in Dynamic Networks" (COSIN) (to L.S.B.).
DOI 10.1002/net.20141
Published online 2 October 2006 in Wiley InterScience (www.interscience.wiley.com).

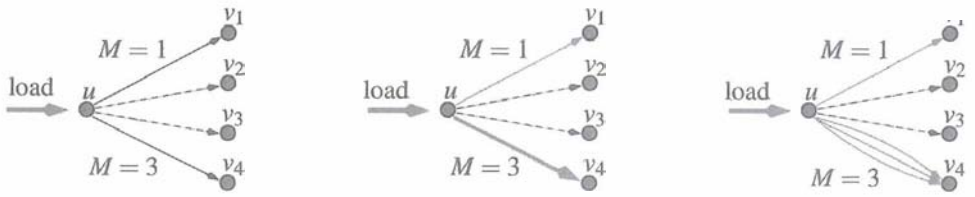

FIG. 1. Load splitting for arc multiplicities. Left: structure of outgoing arcs of node u; Middle: structure considering the `arc` concept; Right: structure considering `link` concept. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

The *arc multiplicity* of an arc a is the number of parallel links associated with arc a. We want to determine a positive integer weight $w_a \in [1, 65535]$, as well as the multiplicity, for each arc $a \in A$, such that the network cost is minimized. Network cost in our application is the overall sum of the products of the multiplicity and the length of each arc. OSPF weights are restricted to be identical for all links on the same arc. Furthermore, we consider each link capacity to be fixed and equal to M. Therefore, each arc capacity is limited to a discrete set of values

$$0, M, 2M, 3M, \ldots.$$

Multiplicities are determined such that all of the demand can be feasibly routed on the network, that is, no arc load exceeds the arc's capacity. For quality of service (QoS) purposes, the definition of feasible route is slightly different. A feasibly routed flow is such that no arc load exceeds a fraction ρ ($0 < \rho \leq 1$) of the arc's capacity.

For traffic splitting purposes, each parallel link is considered as an independent link. For clarity, we use the term `arc` to refer to the structure installed between two nodes and `link` for each capacitated link inserted in this structure. As an example, consider Figure 1, where a load going through router u is destined to a target router t (not shown in the figure). Arcs $(\overrightarrow{u, v_1})$ and $(\overrightarrow{u, v_4})$ belong to the shortest path graph to destination t and arcs $(\overrightarrow{u, v_2})$ and $(\overrightarrow{u, v_3})$ do not. Let the arc multiplicities of $(\overrightarrow{u, v_1})$ and $(\overrightarrow{u, v_4})$ be 1 and 3, respectively. Then, one-fourth of the load will be routed on arc $(\overrightarrow{u, v_1})$, which has one link, and three fourths will be routed on arc $(\overrightarrow{u, v_4})$, which has three links.

Because failures can occur in either arcs or routers, it is desirable to design IP networks that are survivable subject to these types of failures. To overcome the complexity associated with generating all possible combinations of failures, we limit ourselves to single-arc or single-router failure. Because links on a given arc are in some sense dependent, we consider in this article single-arc failure instead of single-link failure. Therefore, when an arc fails, all links associated with that arc fail. When a router fails, all arcs incoming and outgoing to/from this router are disabled. Furthermore, all demands having this router as source or destination are discarded. We also assume that no single-arc failure disconnects the network. A failure usually changes one or more shortest path graphs in the network. Consequently, demand may be rerouted on different paths. If there is sufficient capacity such that all of the demand can be feasibly rerouted for all possible single-arc and single-router failures, the network is called *survivable*. In the case of failures, the fraction of the arc's capacity limiting the load is usually larger than the fraction used to define a feasible flow for the nonfailure case. For example, in the case of no failure, a flow would be feasible if no arc load exceeds 70% of the arc's capacity, while for the case of single failure, this fraction could be, say, 90%. These values, which we call ρ_n and ρ_f for the nonfailure and failure cases, respectively, depend on the network's quality of service requirements. A high quality of service is associated with a low fraction. Similarly, as shown in Figure 2, the cost of the network is inversely proportional to the values of the fractions.

Several articles on OSPF routing and on survivable IP network design have recently appeared in the literature. Fortz and Thorup [9] propose a local search algorithm to determine OSPF weights to minimize network congestion. Ericsson et al. [8] optimize the same objective function, but with a genetic algorithm. A local search is added to a similar genetic algorithm in Buriol et al. [5]. Bley et al. [3] address survivable IP network design for single arc or router failure. Capacities are assigned so that a specified percentage of each demand is satisfied for any single router or arc failure. Capacities are installed in multiples of a capacity unit, and can vary between a minimum value and a maximum value if the arc is utilized. Maximum hop count is imposed on each route. Arcs, in that article, have a single link. Furthermore, routing is done on a single shortest path, that is, load splitting is not implemented. Link weights are adjusted by local search to minimize the total cost of the network, which depends on the installed capacities. Bley [2] considers a similar problem, but also takes into account hardware considerations that affect solution feasibility and cost. A solution method based on Lagrangian Relaxation is used. Holmberg and Yuan [12] use simulated annealing to determine OSPF weights and arc capacities to minimize network cost based on a fixed charge and variable cost model. OSPF routing with load splitting is used. Arcs in this article, however, have a single link. Broström and Holmberg [4] use a similar approach, but tackle the problem of minimizing network cost while maximizing a measure of network survivability. Only arc failures are considered. A mixed integer programming formulation is given. As with the articles above, arcs have a single link.

In this article, we present a genetic algorithm for finding good-quality solutions to the survivable network design problem for single arc or single router failures where arc

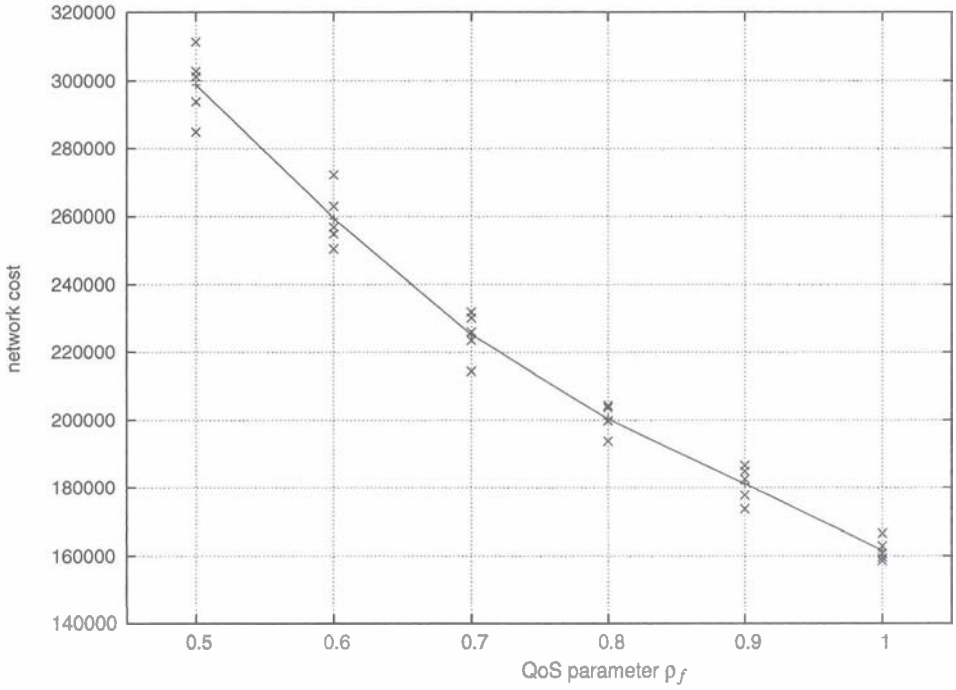

FIG. 2. Effect of varying QoS parameter ρ_f on network cost for a 74-router, 278-arc network (`net-3` in Section 5) with single arc failure. Five independent runs were done for each parameter value. The line connects average network cost values. Runs used $\rho_n = .8\rho_f$.

multiplicities are considered. When simulating arc or router failures, the shortest path graphs are updated using dynamic shortest path algorithms [6], instead of recomputing the shortest path graphs from scratch. Moreover, OSPF weights are computed and OSPF routing is affected by arc multiplicities. Several extensions to the basic model are proposed, including identical or different OSPF weights on symmetric arcs, identical or different multiplicities for symmetric arcs, different objective functions, arc weight range, and latency constraints.

This article is organized as follows. In Section 2, we describe a general evolutionary algorithm for weight setting in OSPF routing. This algorithm calls a procedure that, given a weight setting, routes the demands to determine arc multiplicities and computes the cost of the network. This procedure is described in Section 3. In Section 4, we add survivability to the network design process. Computational results on "real-world" and artificial networks are presented in Section 5. Extensions and concluding remarks are given in Section 6.

2. EVOLUTIONARY ALGORITHM FOR WEIGHT SETTING IN OSPF ROUTING

Evolutionary algorithms, such as genetic algorithms [10, 11], evolve a set of solutions (the population) over time (generations). The solutions of each generation are formed by combining pairs of solutions (mating) from the preceding generation, and randomly perturbing them (mutation). The process is repeated for a fixed number of generations, and the best solution in the last generation is returned by the algorithm as an approximate solution to the optimum.

Ericsson et al. [8] presented a genetic algorithm for setting OSPF weights in an IP network with known link capacities. The objective function used was the one proposed by Fortz and Thorup [9], which increasingly penalizes loads that approach and go over link capacities. The same problem was addressed in Buriol et al. [5], where a local search procedure was applied to the resulting offspring after mating.

Our algorithm uses the same genetic algorithm structure described in [5, 8], but differs in how the solution is evaluated. Furthermore, it considers single arc or router failures. In this section, we describe our genetic algorithm optimizing a general function $f(w)$. In the next section, we show how we compute this function. We do not consider survivability issues yet and address those issues later, in Section 4.

Each of the p elements of the population is an integer weight vector, characterizing a solution. Each arc $a \in A$ has associated with it an integer weight $w_a \in [1, \overline{w}]$. Initially, all but one of the solutions are made up of uniformly randomly generated weights. The remaining solution is made up of unit weights.

The algorithm is run for N generations. We use the random keys crossover scheme of Bean [1] for mating and mutation. At each generation, the solutions are evaluated and sorted in increasing order of objective function value. The N_A solutions with the smallest costs are put in class A, while the N_C solutions with the largest costs are placed in class C. The remaining solutions are assigned to class B.

The next generation is produced as follows. All solutions in class A are automatically promoted, as is, to the next generation. All solutions in class C are discarded and replaced by

random weight solutions in the next generation. The remaining N_B solutions of the next generation are generated as follows. To produce each offspring, one parent (p_1) is selected at random (with replacement) from the elements in class A and one (p_2) from $B \cup C$. The i-th weight of the offspring will be the i-th weight of parent p_1 with probability $\pi_1 > 1/2$, the i-th weight of parent p_2 with probability $\pi_2 < 1 - \pi_1$, or a random weight in the interval $[1, \overline{w}]$ with probability $1 - \pi_1 - \pi_2$.

Several parameters must be set. In Section 5, where computational results are described, we define these values. Guidelines for setting these parameters are as follows. The larger the population size p, the longer will each generation take to be computed. Experiments performed with this problem had shown that the CPU time is linearly proportional to the population size and to the number of generations. We also expect solution quality to improve by increasing the number of generations N and/or increasing the population size p. The mating/mutation probabilities π_1 and π_2 should be such that $\pi_1 > \pi_2$ to avoid mutating too much the solution and generating an offspring with little information derived from the parent solutions, and $\pi_1 + \pi_2 \approx 1$. N_A, N_B, and N_C should be such that $N_B > N_A > N_C$. N_C usually is small to avoid excessive randomization of the population and taking too many generations to converge. However, N_A is usually not large to avoid converging the population too fast and "getting stuck" in a local minimum.

3. HEURISTIC FOR COMPUTING ARC MULTIPLICITIES

In this section, we describe a heuristic that computes arc multiplicities given a topology of potential arcs, lengths, and capacities associated with each arc (all links of the same arc have identical capacities and lengths), OSPF arc weights, and a demand matrix. This heuristic not only returns the arc multiplicities, but also computes the network cost $f(w)$ that guides the genetic algorithm of Section 2. In this section, we describe the heuristic for the no-failure case. A pseudocode for the heuristic is described. Later, in Section 4, we consider the full procedure, with single failures. In all pseudocode, the parameters for some functions are omitted for clarity.

Let T be the set of destination routers. We compute $|T|$ single-destination shortest path graphs g^t, $t \in T$. Each g^t, with destination $t \in T$, has an $|A|$-vector L^t associated with the arcs, that stores the partial loads flowing to t and traversing each arc $a \in A$. The $|A|$-vector l stores the total load traversing each arc $a \in A$. For each destination $t \in T$, the $|N|$-vectors π^t and δ^t are associated with the nodes. The distance from each node to t is stored in π^t, while δ^t keeps the number of arc multiplicities (links) outgoing from each node in g^t.

The heuristic first routes the demand using OSPF routing rules and assuming each arc has unit multiplicity. Arc loads are computed and required multiplicities are determined. Demand is rerouted assuming the updated multiplicities and updated arc load are determined. This cycle (routing, computing arc loads, determining multiplicities) is repeated until the demand can be feasibly routed using the current multiplicities.

```
procedure EvaluateSolution(w, lf, rf)
1   forall a ∈ A do μ_a = 1;
2   forall t ∈ T do
3       π^t ← ReverseDijkstra(w);
4       g^t ← ComputeSPG(w, π^t);
5       δ^t ← ComputeDelta(g^t);
6       L^t ← ComputePartialLoads(μ, δ, π, g^t);
7   end forall
8   l ← ComputeTotalLoads(L);
9   S ← UpdateMultiandDelta();
10  if |S| > 0 UpdateSolution();
11  forall a ∈ A if l_a = 0 then μ_a = 0;
12  f ← Σ_{a∈A} μ_a * length_a;
13  return(f, μ);
end procedure
```

FIG. 3. Pseudocode for heuristic for computing arc multiplicities.

Figure 3 shows the pseudocode for the heuristic for computing arc multiplicities. For each potential arc $a \in A$, the arc's multiplicity μ_a is initially set to 1 in line 1. For each demand destination $t \in T$, the shortest path graph distances are computed from scratch, using Dijkstra's algorithm [7], in procedure `ReverseDijkstra` (line 3). Given the weights w and the shortest path distances, the shortest path graph is identified by procedure `ComputeSPG` in line 4. Given the shortest path graph g^t, δ^t is computed in line 5 by `ComputeDelta`. OSPF routes with load splitting are computed in line 6 and the partial loads vector L^t is determined by procedure `ComputePartialLoads` for each arc in the shortest path graph g^t. The total load l on an arc, computed in line 8 by procedure `ComputeTotalLoads`, is the sum of all partial loads routed on that arc. In the implementation, the total loads are actually computed in procedure `ComputePartialLoads`.

Next, in line 9, for each arc $a \in A$, the multiplicity μ_a, and consequently δ_a, are updated in `UpdateMultiandDelta` according to the total load l_a on the arc. The updated multiplicity of arc a is the maximum of its current multiplicity μ_a and the minimum multiplicity required to route load l_a on arc a, $\lceil l_a/(\rho \cdot c_a) \rceil$, where ρ is such that $0 < \rho \leq 1$ and defines the portion of the capacity that can be used, that is, $\mu_a \leftarrow \max\{\mu_a, \lceil l_a/(\rho \cdot c_a) \rceil\}$. The arcs whose multiplicities were updated are placed in a set S and the loads are updated in procedure `UpdateSolution`, described next. In line 11, the arcs with no load have their multiplicities set to 0 and in line 12, the solution cost is computed as the sum of the products of arc multiplicities and the corresponding arc lengths. This value, as well as the arc multiplicities, are returned in line 13.

Pseudocode for procedure `UpdateSolution` is given in Figure 4. For each shortest path graph g^t, $t \in T$, the loads are updated if at least one of the arcs in S belongs to g^t. In this case, the partial loads are not recomputed from scratch, but are simply updated. In line 4 of `UpdateSolution`, tail nodes

```
procedure UpdateSolution()
1   do
2       forall t ∈ T do
3           H ← {};
4           forall e = (u,v) ∈ S ∩ g^t do InsertIntoHeapMax(H, u, π_u^t);
5           while HeapSize(H) > 0 do
6               u ← FindAndDeleteMax(H);
7               if π_u^t ≠ ∞ then
8                   σ ← (D_u^t + Σ_{a∈g^t∩IN(u)} L_a^t) / δ_u^t;
9                   forall e = (u,v) ∈ OUT(u) do
10                      if e ∉ g^t then λ ← 0;
11                      else λ ← μ_e * σ;
12                      if λ ≠ L_e^t then
13                          l_e ← l_e − L_e^t + λ;
14                          L_e^t ← λ;
15                          InsertIntoHeapMax(H, v, π_v^t);
16                      end if
17                  end forall
18              end if
19          end while
20      end forall
21      S ← UpdateMultiandDelta();
22  until |S| = 0;
end procedure
```

FIG. 4. Pseudocode for the procedure that updates the solution.

u of all arcs $(\overrightarrow{u,v}) \in S$ belonging to g^t are inserted in a priority queue indexed by the distance to t. Nodes u are removed from the heap in line 6 and considered one by one until the heap is empty. The test in line 7 is only used when considering failures, and is always true for the no-failure case. In line 8, the load is evenly split, considering arc multiplicities. The load σ is computed as the ratio between the sum of the load leaving node u and the load passing through node u and δ_u^t. All arcs $(\overrightarrow{u,v}) \in g^t$ outgoing from u have their new loads (λ) computed, and if they have changed, the loads are updated in lines 13–14 and node v is inserted in the heap (line 15).

After the loads are updated, some multiplicities may change. In line 21, procedure `UpdateMultiandDelta` computes the set of arcs S for which multiplicities have changed, and updates the vectors μ and δ. Although set S contains at least one arc, the loop from line 2 to line 20 is repeated. In Section 5, we record the number of times that loop 2–20 is repeated and present statistics indicating that this number is small.

4. SURVIVABLE NETWORK DESIGN

In this section, we add survivability to the network design process. We consider single-arc, single-router, and single-arc or single-router failure. The difference between the genetic algorithm for the no-failure case and the ones for the single failure cases is the procedure `EvaluateSolution`. We describe changes to `EvaluateSolution` and present the procedure that simulates these failures.

Unlike the procedure in the previous section, the solution evaluation not only computes the multiplicities for the no-failure case, but also updates the multiplicities considering every single failure. These changes are shown in the pseudocode in Figure 5.

Lines 9 and 10 of the pseudocode of the no-failure version of `EvaluateSolution` in Figure 3 have been substituted by lines 9 to 21 in the complete version in Figure 5. Furthermore, in line 23, if in no situation (no failure or any single failure) arc a has a positive load, then the arc's multiplicity μ_a is set to 0. In the pseudocode, the maximum load on arc a over no-failure and all single-failure simulations is $\bar{l}_a$.

In lines 9 to 21, the multiplicities are updated considering the cases of no-failure, single-arc failures, and single-router failures, consecutively. In line 10, the multiplicities are updated for the no-failure case `UpdateMultiandDelta`. Here, the QoS parameter ρ_n is used. A change in multiplicity may cause a change in routing that can consequently cause a change in arc loads. A change in an arc load can lead to another change in the arc's multiplicity. For this reason, the steps in lines 10 to 20 are repeated in a circular loop until no further change in arc loads or multiplicities is detected in a full cycle of the loop by procedure `NoMoreChanges`. In Section 5, we study the number of times that procedure `NoMoreChanges` is called, and consequently, the number of times that the loop in lines 9–21 is executed.

Single-arc failures and single-router failures are simulated by the same procedure, `SimulateFail`. This procedure has two input parameters: a set F of sets $F_1, F_2, \ldots, F_q$ of arcs that are consecutively removed from the graph during the simulation; and the cardinality q of F. For the single-arc failure case, `SimulateFail` first takes as input a set of sets F_a of single arcs, where each F_a consists of arc a. The second parameter is m, the cardinality of F. For single-router failure, `SimulateFail` takes as input a set F of sets F_i and

```
procedure EvaluateSolution(w, lf, rf)
1   forall a ∈ A do μ_a = 1;
2   forall t ∈ T do
3       π^t ← ReverseDijkstra(w);
4       g^t ← ComputeSPGandLoad(w, π^t);
5       δ^t ← ComputeDelta(g^t);
6       L^t ← ComputePartialLoads(μ, δ, π, g^t);
7   end forall
8   l ← ComputeTotalLoads(L);
9   while 1 do
10      S ← UpdateMultiandDelta();
11      if |S| > 0 UpdateSolution();
12      if NoMoreChanges() then goto OUTLOOP;
13      if lf = 1 do
14          simulateFail(A, m);
15          if NoMoreChanges() then goto OUTLOOP;
16      end if
17      if rf = 1 do
18          simulateFail(R, n);
19          if NoMoreChanges() then goto OUTLOOP;
20      end if
21  end while
22  OUTLOOP:
23  forall a ∈ A if l̄_a = 0 then μ_a = 0;
24  f ← Σ_{a∈A} μ_a * length_a;
25  return(f, μ);
end procedure
```

FIG. 5. Pseudocode for updating a solution considering failures.

```
procedure SimulateFail(F = {F_1, F_2, ..., F_q}, q)
1    i = 1;
2    while NoMoreChanges() do
3        CopySolution();
4        forall a ∈ F_i do ŵ_a ← w_a;
5        forall a ∈ F_i do w_a ← ∞;
6        UpdateSPGandLoad();
7        forall a ∈ F_i do w_a ← ŵ_a;
8        S ← UpdateMultiandDelta();
9        if |S| > 1 then
10           forall t ∈ T do UpdateDelta(δ^t, g^t);
11           UpdateSolution();
12       end if
13       if i < q then i ← i + 1;
14       else i ← 1;
15   end while
end procedure
```

FIG. 6. Pseudocode for updating the multiplicities and loads simulating a given set of failures.

the cardinality n of F. Each F_i in this case is the set of all incoming and outgoing arcs to and from router i.

We conclude this section with a description of failure simulation, which is described in the pseudocode in Figure 6. In the loop in lines 2 to 15 of the pseudocode, the failure of each set F_i $(i = 1, \ldots, q)$ is simulated. The loop is repeated until one pass is completed over the entire set F without causing any change in the arc multiplicities, and consequently in the arc loads. This condition is tested in `NoMoreChanges`. For each simulation, the current solution is copied to an auxiliary solution in line 3. The current weights of arcs $a \in F_i$ are saved in the auxiliary vector $\hat{w}$ (line 4) and are set to infinity (line 5). Procedure `UpdateSPGandLoad` updates the shortest path graphs and the total and partial loads of the copied solution with weights w used to simulate the failure of arcs $a \in F_i$. The weights are reset to their original values in line 7 and in line 8 the multiplicities are updated for the copied solution. The QoS parameter ρ_f is used in `UpdateMultiandDelta`. If at least one multiplicity has changed, then for each $t \in T$, the original δ^t, L^t, and l^t are updated in lines 10 and 11. The counter i is either incremented in line 13 or reset to 1 in line 14 to force the loop to cycle through all of the sets $F_1, F_2, \ldots, F_q$.

There is another way to simulate failures. Instead of copying the current solution (l and $g^t, \pi^t, \delta^t, L^t$, for each $t \in T$) to the auxiliary solution (line 3), we simulate each failure by first modifying the original solution and then undoing the modification at the end of the loop. We implemented and tested this alternative approach, but surprisingly, it was computationally less efficient than the one we adopted. Perhaps this is due to the fact that copying a block of memory is done very efficiently by modern hardware.

In Section 5, we study the number of times that the failure of set F_i is simulated, discriminating between arc failure and router failure simulations.

5. COMPUTATIONAL RESULTS

We describe computational experiments with a C language implementation of the algorithm described in this article on four test problems derived from real-world IP networks. The dimensions of the four instances are summarized in Table 1. The first two instances (`net-1` and `net-2`) are dense networks in which there is demand between all pairs of routers. Each is made up of nodes that correspond to existing or planned routers of a large tier-1 Internet Service Provider (ISP) in a region (of multiple states) of the United States. The other two instances (`net-3` and `net-4`) are much larger. Both involve sparse networks. Instance `net-3` corresponds to an outdated backbone IP network of a large tier-1 ISP. Only 18 of the 74 nodes are destination routers of demand pairs. Because of this, the algorithm works with only 18 shortest path graphs per solution. The last instance (`net-4`) consists of sparse optimized regional IP networks linked by a dense backbone. It corresponds to the nationwide network of a large tier-1 ISP. All nodes are destination routers of demand pairs, and almost all router pairs have a demand associated with them. For each instance in this study, all links have identical capacities.

Table 2 presents some statistics about the instances. Minimum, average, and maximum values are given for demand, length, and in/out degree. Because each link is present in both directions (for each arc $a \rightarrow b$, the networks have a corresponding arc $b \rightarrow a$), the indegree and the outdegree of every node are equal, and therefore, these values were not repeated in the table. Symmetric arcs have the same capacity and the same length. Demands, however, are rarely symmetric. As can be observed, the demands vary considerably from pair to pair. The minimum demand value is very small compared to the maximum values. Moreover, the larger networks have higher demand values, because they are derived from nationwide backbones as opposed to regional backbones.

The capacity of each link from networks `net-1`, `net-2`, and `net-3` are set to 2.48, whereas links of `net-4` have unit capacities.

The C program was compiled with the `gcc` compiler, version 3.2.3 with optimization flag `-O3` and run on a SGI Altix 3700 Supercluster running RedHat Advanced Server with SGI ProPack. The cluster is configured with 32 1.5-GHz Itanium-2 processors (Rev. 5) and 245 Gb of main memory. Each run was limited to a single processor. User running times

TABLE 1. Test problem network dimensions.

Network	$\|N\|$	$\|A\|$	$\|T\|$	$\|D\|$
`net-1`	10	90	10	90
`net-2`	11	110	11	110
`net-3`	74	278	18	306
`net-4`	71	350	71	4960

For each of the four networks used in the computational experiments, this table lists the number of routers ($|N|$), the number of potential arcs ($|A|$), the number of distinct destination routers taking part in the list of demands ($|T|$), and the number of demand pairs ($|D|$).

TABLE 2. Minimum (min), average (avg), and maximum (max) values for demand, length, and in/out degrees.

	Demand			Length			In/out degree		
Instance	min	avg	max	min	avg	max	min	avg	max
net-1	0.0000001631	0.18	1.54	3.70	142.88	335.50	9	9.00	9
net-2	0.0000000040	0.11	1.06	1.36	55.981	126.29	10	10.00	10
net-3	0.0016299999	0.50	6.07	0.00	635.36	2468.89	2	5.15	12
net-4	0.0000008064	0.06	3.36	4.50	507.08	3669.80	2	4.93	13

were measured with the `getrusage` system call. Running times exclude problem input.

As mentioned in Section 2, solution quality improves with population size and number of generations (see Figs. 7 and 8). On the other hand, running times increase as population size and number of generations increase. Throughout these computational experiments, we use a population of size $p = 50$, mating/mutation probabilities $\pi_1 = 0.7$ and $\pi_2 = 0.29$, and define classes A, B, and C with $N_A = 0.25p$, $N_B = 0.7p$, and $N_C = 0.05p$ elements, respectively. Weights can take values in the interval $[1, \overline{w} = 20]$. Larger and shorter ranges were tested, but the one adopted seems to drive to a good network optimization. Short ranges have too many ties and large ones almost do not have any. Experimental testing suggests that having ties helps in finding better solutions. With respect to the QoS parameters, we use $\rho_n = 0.8$ for no-failure routing and $\rho_f = 0.95$ for the cases where failures occur. On all experiments, the objective function is the sum of the weighted multiplicities, where the weighted multiplicity of an arc is the multiplicity of the arc multiplied by its length.

Figure 9 shows network cost as a function of running time for four runs on network `net-3`. The algorithm was run for no-failure, single-arc failure, single-router failure, and single-arc/single-router failure. Each run was limited to 10,000 seconds. The figure shows that the algorithm produced designs having costs: 104,528.32 for the no-failure case, 168,801.88 for single-arc failure, 172,570.86 for single-router failure, and 185,709.16 for single-arc/single-router failure. Hence, to achieve protection from single-arc and single-router failures on `net-3` results in about 78% more network cost than when survivability is not required. Figure 10 illustrates the progress of the best solutions produced by the genetic algorithm. The figure plots the relative errors of each run as a function of CPU time. Errors are computed relative to the value of the best solution found during each run, that is, all relative errors are zero at time 10,000 seconds. The figure shows that for the no-failure run, the algorithm produces solutions within 10 and 1% of the best solution found (104,528.32) in 6.86 and 1921.53 seconds, respectively. For the single-arc failure run, solutions within 10 and 1% of the best solution

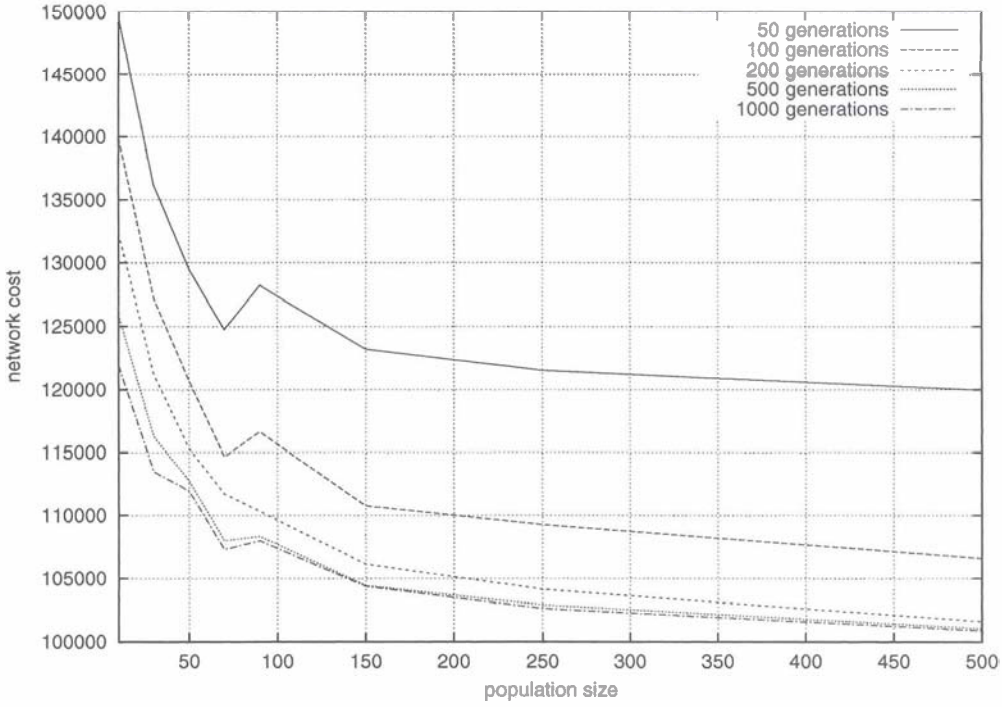

FIG. 7. Network cost as a function of number of genetic algorithm population size for 50, 100, 200, 500, and 500 generations. Experiment done on a 74-router, 278-arc network (`net-3` in Section 5) with no router or arc failure.

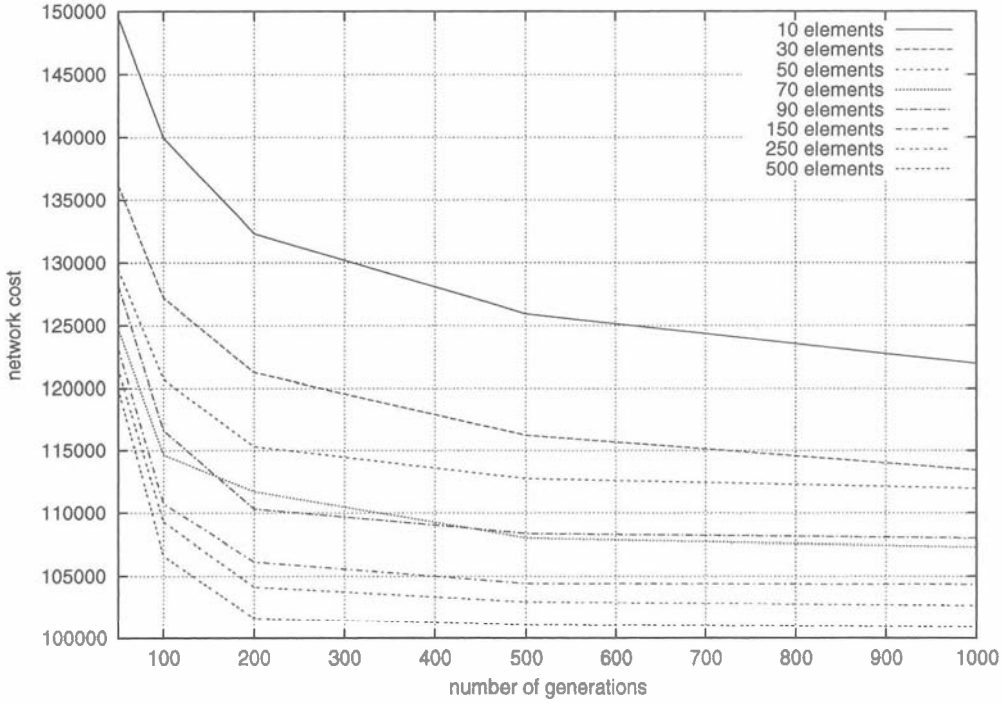

FIG. 8. Network cost as a function of number of genetic algorithm generations for population sizes 10, 30, 50, 70, 90, 150, 250, and 500 elements. Experiment done on a 74-router, 278-arc network (`net-3` in Section 5) with no router or arc failure.

found (168,801.88) are found in, respectively, 374.11 and 3365.66 seconds. In 85.18 and 6925.90 seconds, the algorithm produces solutions within 10 and 1%, respectively, of the best solution found (172,570.86) for the single-router failure run. Finally, for the single-arc / single-router failure run, the algorithm produces solutions within 10 and 1% of the best solution found (185,709.16) in 356.78 and 9249.97 seconds, respectively.

In the next experiment, the genetic algorithm (GA) was run on each of the four test problems using five different

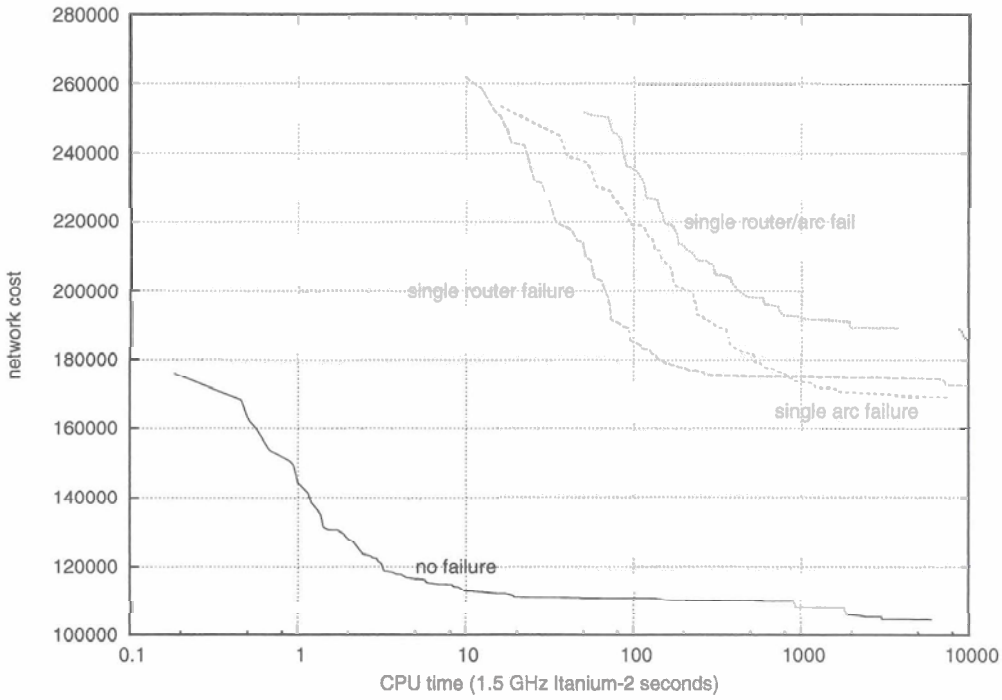

FIG. 9. Network design cost for genetic algorithm runs for 10,000 seconds on instance `net-3` for no-failure, single-router failure, single-arc failure, and single-arc or single-router failure.

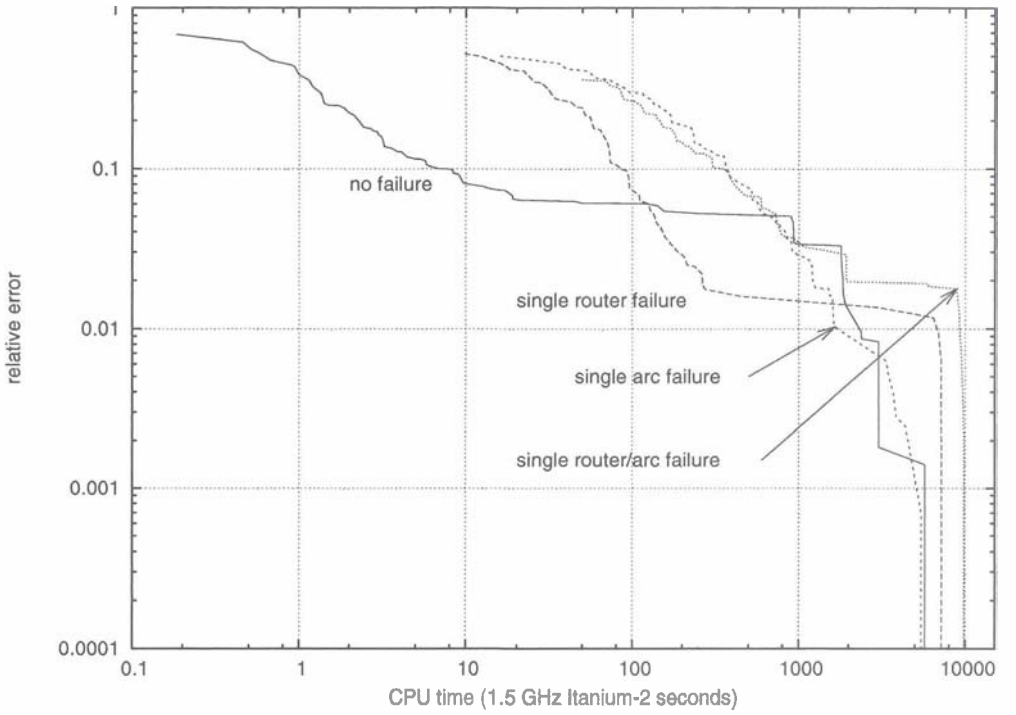

FIG. 10. Relative error with respect to the best solution found for genetic algorithm runs for 10,000 seconds on instance `net-3` for no-failure, single-router failure, single-arc failure, and single-arc or single-router failure.

random number generator seeds. For problems `net-1` and `net-2`, the algorithm was run for 200 generations, whereas for the larger problems `net-3` and `net-4`, the number of generations was fixed at 100. We compare the average network cost for weights computed by the genetic algorithm with average network cost produced using unit and random (rand) weights. For both random and unit weight solutions, we apply the multiplicity setting heuristics described in Sections 3 and 4, that is, repeatedly, demand is routed following OSPF rules, and loads and multiplicities are computed, until there is enough capacity so that all of the demand can be feasibly routed. We compute as many random weight solutions as are generated by the genetic algorithm. For `net-1` and `net-2`, where 200 generations with a population of size 50 are computed for each random seed by the GA, we compute 7500 random weight solutions. For `net-3` and `net-4`, where 100 GA generations are run, we examine 3750 random weight solutions. For each of the five independent runs, the best random weight solution is returned. The average random solution reported is the average of these five solutions. We compute the network costs for the four combinations of arc and router failure/no-failure. We also compute a lower bound (LB) on the network cost as follows. For each target node, we determine a shortest path graph using lengths as arc weights. Demands are routed on shortest paths assuming unit multiplicities and arc loads are computed. New multiplicities are computed as in Section 3, except that their values are not rounded up to an integer, but instead are taken as real numbers. The arc cost is the product of its real-valued multiplicity and its length. A lower bound on the network design cost is the sum of all arc costs. The lower bound is the maximum network cost found repeating this procedure for each failure scenario. The results are summarized in Table 3, where for each instance, four sets of costs are shown: no arc or router failure, single-router failure and no arc failure, single-arc failure and no router failure, and both single-arc and single-router failure. Average CPU times for the genetic algorithm (in 1.5-GHz Itanium-2 seconds) are also listed in the table. Figures 11 and 12 show plots of the data in Table 3. We recall that the unit weight solution is one of the initial solutions generated by the GA. Thus, the final solution found by the GA can never cost more than the design using unit weights.

In almost all cases, the genetic algorithm produced designs where network cost increased when additional failures were considered. The only exception was on `net-1`, where the average cost for single-arc failure and no router failure was 2773.0, while it decreased to 2766.8 when single-router failures were also considered. This is probably due to the fact that demand originating or terminating at failed routers is discarded in the cost computation.

Table 4 lists ratios of average network costs of random weight and genetic algorithm solutions and of average network costs of unit weight and genetic algorithm solutions. The table also lists ratios of average genetic algorithm solutions and the lower bounds. The table considers only runs with no failure or single-arc failure. With respect to random weight solutions, the GA solutions were from 15% up to almost a factor of three smaller. Likewise, with respect to unit weight solutions, GA solutions were up to a factor of three smaller.

TABLE 3. Average network cost for unit, random, and genetic algorithm arc weight solutions, lower bound cost, and genetic algorithm running times (seconds on an 1.5-GHz Itanium-2 processor) for the experiments with no failures, and all single-failure combinations.

	Failure		Avg network cost				Avg time
Network	Arc	Router	Unit	Random	GA	LB	GA
net-1	no	no	4162.4	3089.6	1597.1	690.7	1.48
	no	yes	6591.5	5231.3	2512.8	690.7	7.75
	yes	no	6443.4	5389.0	2773.0	841.7	34.70
	yes	yes	6964.4	5520.6	2766.8	841.7	42.04
net-2	no	no	1805.4	1385.9	558.8	209.4	1.76
	no	yes	2770.8	2361.3	1032.2	209.4	9.48
	yes	no	2782.8	2319.6	1041.4	223.6	46.92
	yes	yes	2801.1	2419.5	1081.4	223.6	54.77
net-3	no	no	183195.2	165115.4	120708.1	51850.2	5.26
	no	yes	264176.4	314559.4	182941.6	83775.6	116.60
	yes	no	257112.1	303168.1	174780.8	80040.0	381.70
	yes	yes	260940.4	321332.1	185379.1	84178.2	470.29
net-4	no	no	655428.0	660958.5	566632.0	171642.1	26.58
	no	yes	1040761.7	1168607.7	941359.4	224954.4	651.96
	yes	no	876604.1	1065181.9	818249.3	222062.7	2035.37
	yes	yes	1076690.4	1254047.5	987499.6	245750.0	3017.41

Values are averaged over five independent runs.

The lower bounds do not appear to be very strong with the GA solution reaching almost a factor of 5 of the lower bound. Recall, however, that these runs were limited to only 100 or 200 generations, and used a small population of size 50. In contrast, the 1000 generation run with a 500 element population of Figure 8 produced a solution with cost 100,880 for `net-3` with no failures. This increases the ratio rand/GA from 1.37 to 1.64 and the ratio unit/GA from 1.52 to 1.82 and decreases the ratio GA/LB from 2.33 to 1.94.

We conclude this section with an empirical examination of the number of times the algorithms described in Sections 3 and 4 execute for each solution evaluation. We call *convergence loops* the set of `while` loops that are executed in the multiplicity setting heuristic. For the purpose of our analysis, we call the convergence loops $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$, and $\mathcal{L}_4$. Loop $\mathcal{L}_1$ is the `while` loop in lines 9 to 21 of procedure `EvaluateSolution`; loop $\mathcal{L}_2$ ($\mathcal{L}_3$) is the loop in lines 2 to 15 of procedure `SimulateFail` for arc (router) failure; and $\mathcal{L}_4$ is the loop in lines 1 to 22 of procedure `UpdateSolution`.

Figure 13 shows how the loops are related. Loop $\mathcal{L}_1$ calls all the other loops. It calls loop $\mathcal{L}_4$ in all four combinations of arc and router failure and no failure. For the case of no arc or router failure, loop $\mathcal{L}_1$ and loop $\mathcal{L}_4$ are executed only once during solution evaluation. In the case of arc or router failure, loop $\mathcal{L}_1$ is executed until one round of no failure, arc failure, and/or router failure is computed without any change in the solution. Because arc failures are simulated before router failures, loop $\mathcal{L}_1$ can terminate between simulations. Therefore, the number of calls to loop $\mathcal{L}_2$ is always at least as large as the number of calls to loop $\mathcal{L}_3$.

Loop $\mathcal{L}_4$ is called from the other loops only when at least one multiplicity has changed. For example, in the last round of each arc or router failure simulation, the multiplicities do not change. Therefore, the number of calls to loop $\mathcal{L}_4$ is always smaller than the sum of calls of the three other loops.

Table 5 presents the number of times loops $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$, and $\mathcal{L}_4$ are called and executed for instances `net-1` and `net-4`. The four combinations of arc and router failure and no failure are considered. These counts were made in the experiments reported in Table 3. For each failure combination, the table lists counts for each possible combination of loops. For each loop and instance, there are three columns. Column No. `called` is the average number of times the loop was called; No. `passes` is the number of times the loop was executed (with an exception in the case of loop $\mathcal{L}_1$), and `max` No. is the maximum number of times the loop was executed by a call. The field No. `passes` for loop $\mathcal{L}_1$ is the number of times that it calls the other convergence loops. For example, as shown in the table, in the case of no arc or router failure, loop $\mathcal{L}_1$ will execute only once per call, and consequently, `max` No. is equal to one.

Because the genetic algorithm was run for 200 generations on the smaller instance (`net-1`), the No. `called` entry for loop $\mathcal{L}_1$ is larger than it is for the larger instance (`net-4`), which was run for only 100 generations. Because there are 50 individuals per population, then three random solutions and 34 solutions originated from crossover are evaluated per generation, which, together with the evaluation of the initial solutions (in the GA the initial population is counted as the first generation of solutions), sum up the total number of calls to loop $\mathcal{L}_1$.

Loop $\mathcal{L}_2$ is executed more times than loop $\mathcal{L}_3$, because there are more arcs than routers in the networks studied.

The maximum number of times a loop is executed gives an idea about the convergence of these loops. For example,

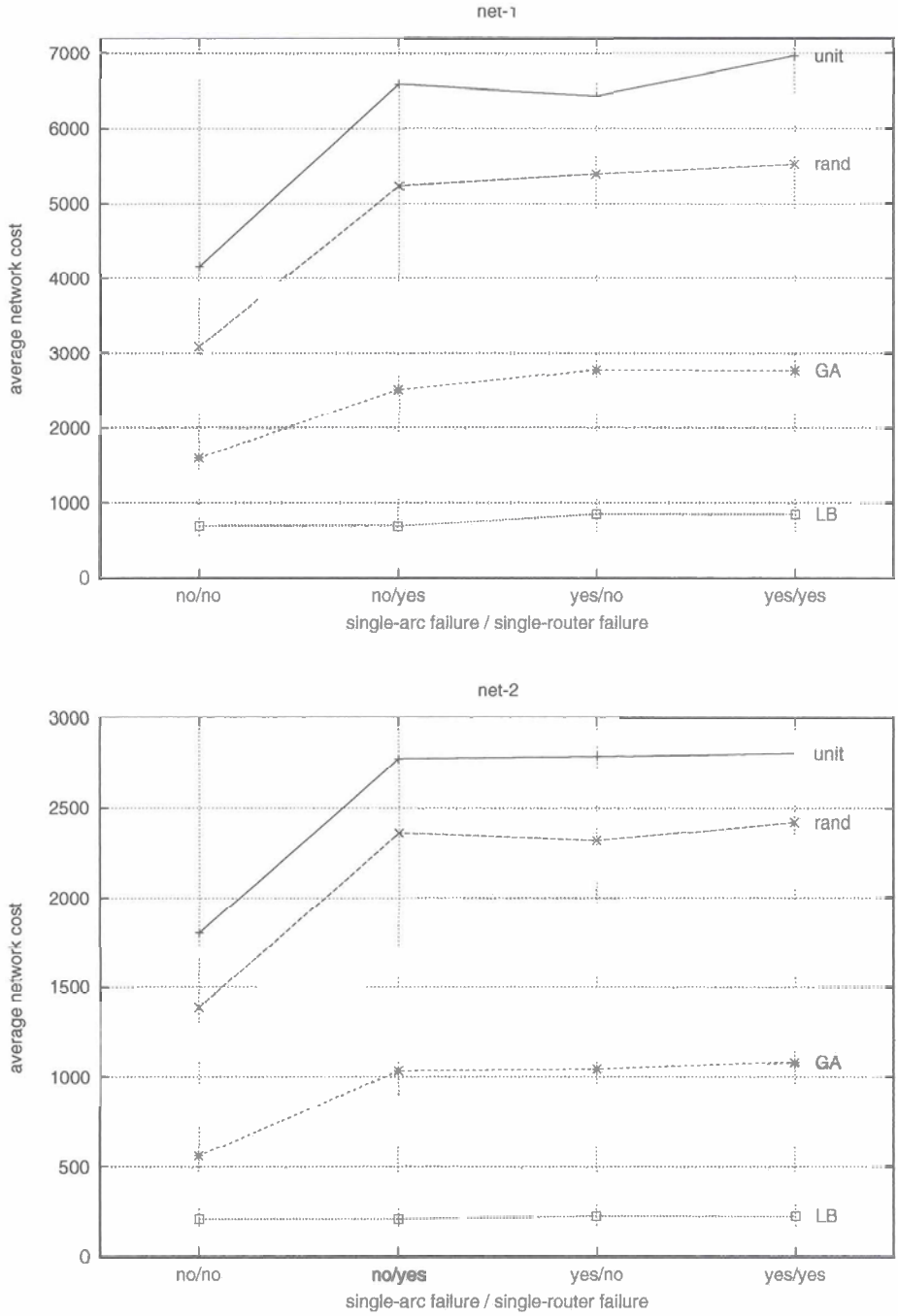

FIG. 11. Average network costs for instances `net-1` and `net-2` for random weights, unit weights, and weights determined by GA. A lower bound on network cost is also shown.

on instance `net-4`, the maximum number of times loop $\mathcal{L}_2$ was executed occurred for the case of arc and router failure. In this, case max No. = 1872.6, which means that the arc failure simulation ran up to 5.35 (=1872.6/350) times for each arc. Also, for this case, loop $\mathcal{L}_1$ was executed up to 3.33 times per call, loop $\mathcal{L}_3$ was called up to 4.75 (=337.0/71) times per call, and loop $\mathcal{L}_4$ was called up to 5.2 times per call. Note that the above maximum numbers of calls are averaged over five runs each and therefore can be real-valued.

The number of times the loops were executed is related to the instance size and kind of experiment performed. Furthermore, it is an indicator of how long the complete experiment took.

6. CONCLUDING REMARKS

In this article, we described a new evolutionary algorithm for survivable IP network design with OSPF routing. The algorithm works with a set of p solutions. Each solution

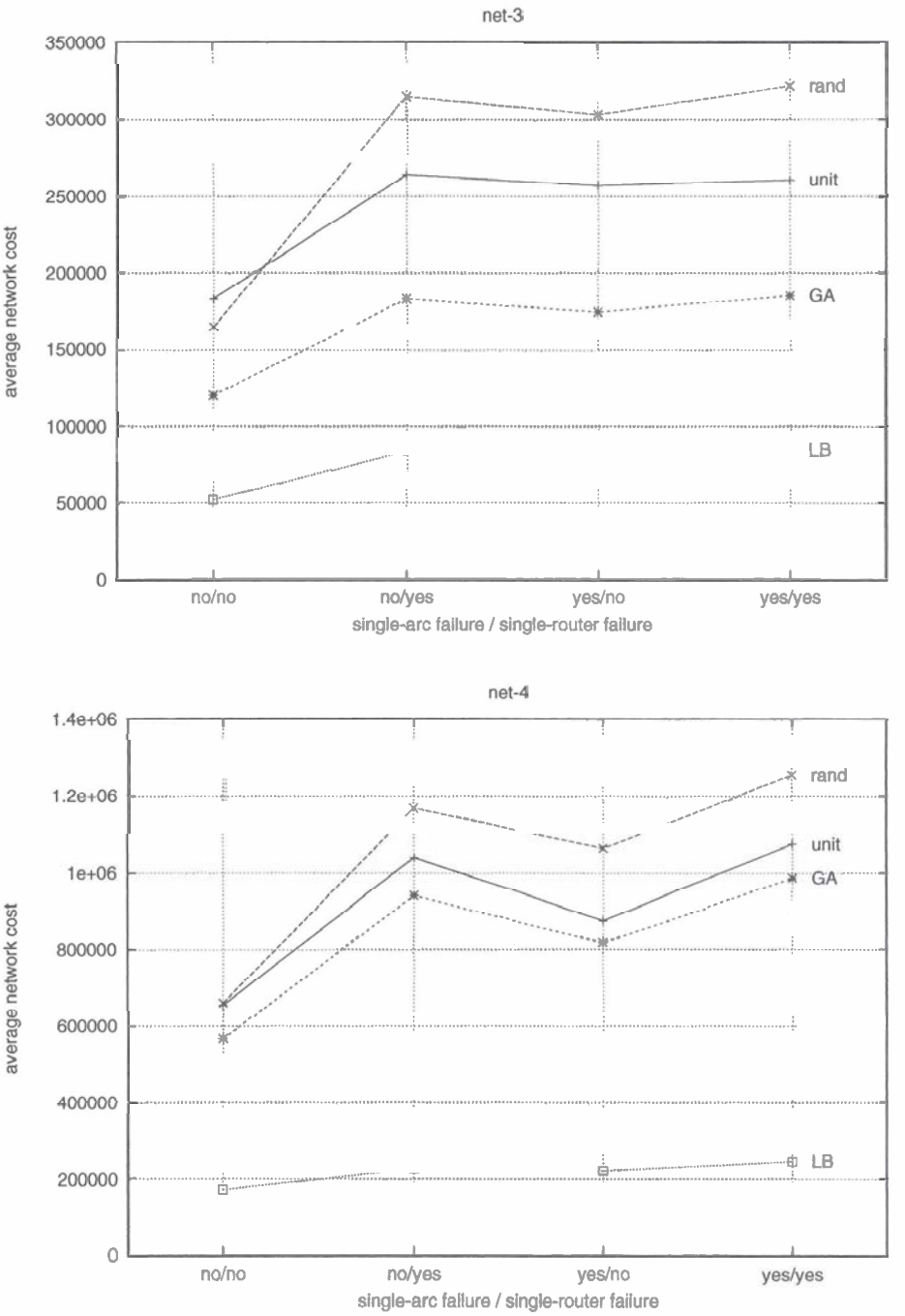

FIG. 12. Average network costs for instances `net-3` and `net-4` for random weights, unit weights, and weights determined by GA. A lower bound on network cost is also shown.

is characterized by an integer arc weight vector. The algorithm starts with a unit weight solution and $p - 1$ random weight solutions and evolves this population over N generations, combining solutions according to a specific recipe. To associate a design with each weight vector, a multiplicity setting heuristic determines the number of links that each arc must have so that all demand can be feasibly routed in the network using OSPF routing with route splitting. To achieve an efficient implementation, instead of recomputing shortest paths and loads from scratch, dynamic shortest path algorithms were used whenever possible.

To simplify the description of the algorithm, we omit from the article many extensions that we have incorporated into the algorithm. Most are simple to incorporate into a genetic algorithm. They include:

- In addition to the network cost described in the article, we also allow minimization of the number of links, as well as a fixed charge cost model, where one pays K_a to use arc a and an additional k_a per link of arc a deployed.
- Besides arc and router failures, the algorithm also allows single *span* failure. A span is a set of arcs that use a common network structure and are likely to fail simultaneously.

TABLE 4. Average network cost ratios of random and unit weight solutions to genetic algorithm network cost and ratios between mean genetic algorithm network costs and lower bound.

Network	Arc failure	unit/GA	rand/GA	GA/LB
net-1	no	2.61	1.93	2.31
	yes	2.32	1.94	3.29
net-2	no	3.23	2.48	2.67
	yes	2.67	2.23	4.66
net-3	no	1.52	1.37	2.33
	yes	1.47	1.73	2.18
net-4	no	1.16	1.17	3.30
	yes	1.07	1.30	3.68

We assume that routers do not fail.

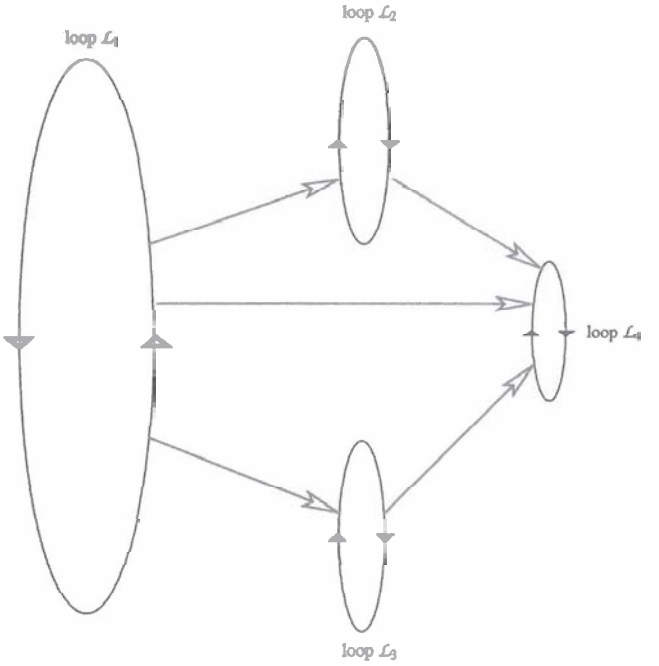

FIG. 13. The relationship between loops in the multiplicity setting heuristics. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

Single-span failure is a generalization of single-arc and single-router failures.

- Often one wishes to impose a maximum latency restriction on demand. The algorithm described in the article does not restrict any demand from following a long and winding route. It is simple to maintain the latency of each shortest path graph and thus compute the latency of each demand pair. A penalty function with the sum of excess latencies can be added to the objective function to disfavor solutions that violate the latency restriction.
- Two arcs are said to be symmetric if the tail of one is the head of the other and vice versa. Often one wishes to impose that OSPF weights be equal on symmetric arcs. Minor modifications of the random weight generation and mating/mutation procedures can achieve this.
- Likewise, multiplicities may be required to be the same on symmetric arcs. A simple modification of the multiplicity setting heuristic allows this type of restriction.
- The implementation allows the input of fixed OSPF weights for a subset $S \subseteq A$ of the arcs.
- The implementation allows for input of an initial set of multiplicities for all solutions to allow for network expansion studies.
- Offspring solutions resulting from mating of two solutions from the population may not be locally optimal with respect to the neighborhoods defined by single link removals and single arc weight increases (as in [5]). We implemented such local search procedures. Although they can reduce the design cost of some offspring, we feel that the additional computational effort does not justify their use.
- The weights produced by the best solution can be further adjusted to optimize a measure of network congestion. We allow the OSPF weights of the set of best solutions in the final population to be optimized with the weight setting algorithm of Buriol et al. [5].

TABLE 5. Average number of calls, runs, and maximum runs per call of the convergence loops of networks net-1 and net-4.

Failure			Net-1			Net-4		
Arc	Router	Loop	No. called	No. passes	max no.	no. called	No. passes	max no.
no	no	$\mathcal{L}_1$	7413.0	7413.0	1.0	3713.0	3713.0	1.0
		$\mathcal{L}_4$	7410.6	7924.4	3.2	3714.0	9213.0	5.0
no	yes	$\mathcal{L}_1$	7413.0	27132.8	5.0	3713.0	14994.2	5.4
		$\mathcal{L}_3$	12298.2	153786.0	32.0	7427.4	1032683.4	369.2
		$\mathcal{L}_4$	15748.0	16001.2	3.2	131439.8	137829.2	5.40
yes	no	$\mathcal{L}_1$	7413.0	28939.8	5.0	3713.0	15009.6	5.6
		$\mathcal{L}_2$	14106.2	1605306.8	276.2	7427.8	5259311.4	1877.2
		$\mathcal{L}_4$	24732.6	25010.8	3.2	431627.4	438850.6	5.2
yes	yes	$\mathcal{L}_1$	7413.0	37727.0	8.0	3713.0	26311.8	10.0
		$\mathcal{L}_2$	14528.2	1666501.6	262.8	9410.2	6225702.2	1872.6
		$\mathcal{L}_3$	8354.0	89004.4	25.8	7471.8	976921.8	337.0
		$\mathcal{L}_4$	31501.2	32202.6	3.2	492128.8	499017.8	5.2

Acknowledgments

The first author acknowledges discussions with Paulo Morelato França, which helped improve an earlier version of the article. We acknowledge the constructive remarks of two anonymous referees. We are also thankful to Doug Shier for carefully proofreading the manuscript and suggesting improvements.

REFERENCES

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

[2] A. Bley, "A Lagrangian approach for integrated network design and routing in IP Networks," Proceedings of International Network Optimization Conference (INOC 2003), 2003, pp. 107–113.

[3] A. Bley, M. Grötschel, and R. Wessäly, "Design of broadband virtual private networks: Model and heuristics for the B-WiN," Robust communication networks: Interconnection and survivability, volume 53 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, N. Dean, D.F. Hsu, and R. Ravi (Editors), American Mathematical Society, New Providence, RI, 1998, pp. 1–16.

[4] P. Broström and K. Holmberg, Multiobjective design of survivable IP networks, Technical Report LiTH-MAT-R-2004-03, Division of Optimization, Linköping Institute of Technology, 2004.

[5] L.S. Buriol, M.G.C. Resende, C.C. Ribeiro, and M. Thorup, A hybrid genetic algorithm for the weight setting problem in OSPF/IS-IS routing, Networks 46 (2005), 36–56.

[6] L.S. Buriol, M.G.C. Resende, and M. Thorup, Speeding up dynamic shortest path algorithms, Technical Report TD-5RJ8B, AT&T Labs Research, 2003.

[7] E. Dijkstra, A note on two problems in connexion with graphs, Numer Math 1 (1959), 269–271.

[8] 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.

[9] B. Fortz and M. Thorup, Increasing internet capacity using local search, Comput Optimizat Appl 29 (2004), 13–48. (Preliminary short version of this paper published as "Internet Traffic Engineering by Optimizing OSPF weights," Proc. 19th IEEE Conf. on Computer Communications (INFOCOM), 2000, pp. 519–528.

[10] D.E. Goldberg, Genetic algorithms in search, optimization, and machine learning, Addison-Wesley, Reading, MA, 1989.

[11] J.H. Holland, Adaptation in natural and artificial systems, MIT Press, Cambridge, MA, 1975.

[12] K. Holmberg and D. Yuan, Optimization of internet protocol network design and routing, Networks 43 (2004), 39–53.