SONET Ring Sizing with Genetic Algorithms[†]

Nachimuthu Karunanithi[‡]
Bellcore, Room 2Q–265
445 South Street
Morristown, NJ 07960
Email: karun@faline.bellcore.com

Tamra Carpenter[§¶]
Bellcore, Room 2L–335
445 South Street
Morristown, NJ 07960
Email: tcar@bellcore.com

Scope and Purpose—Genetic algorithms are becoming recognized as a robust method for generating solutions to certain "hard" optimization problems. In this study, we examine the suitability of a genetic algorithm for solving an optimization problem that arises in sizing SONET rings in a telecommunications network. We consider applying genetic algorithms to this problem because it is a computationally difficult problem whose solutions have clear economic impacts and very straightforward encodings in genetic algorithms.

Abstract—We describe an optimization problem that arises in SONET ring sizing. We compare solutions obtained by the genetic algorithm to both optimal solutions obtained by the CPLEX mixed integer program solver and heuristic solutions generated by the algorithm that is incorporated in the SONET Toolkit – a decision support system for planning SONET networks.

[†]A condensed version of this paper was presented at ACM Symposium on Applied Computing [1].

[‡]Nachimuthu Karunanithi is a Research Scientist at Bellcore in Morristown, NJ. He received Ph.D in Computer Science from Colorado State University in 1992. His research interests are personal communications systems, neural networks, genetic algorithms, and software reliability.

[§]Tamra Carpenter is a Research Scientist at Bellcore in Morristown, NJ. She obtained her Ph.D in Operations Research from Princeton University in 1992. Her research interests are in optimization and mathematical programming – particularly as they apply to telecommunications network design.

[¶]Author for correspondence.

SONET Ring Sizing with Genetic Algorithms[†]

Nachimuthu Karunanithi
Bellcore, Room 2Q–265
445 South Street
Morristown, NJ 07960
Email: karun@faline.bellcore.com

Tamra Carpenter
Bellcore, Room 2L–335
445 South Street
Morristown, NJ 07960
Email: tcar@bellcore.com

1 Introduction

Synchronous Optical Network (SONET) technology allows today's telecommunications networks to concentrate large traffic volumes onto a relatively small number of nodes and links. As a result, the amount of traffic that could potentially be interrupted by the failure of one of these high-capacity network elements is also large. Thus, ensuring survivability through fault-tolerant network design is becoming increasingly important. SONET [2] rings are incorporated into telecommunications network architectures to provide immediate recovery from certain equipment failures.

Structurally, a SONET ring consists of a set of nodes connected by high-capacity optical links to form a cycle that nowhere overlaps itself. (See Figure 1.) A telecommunication network may include several embedded rings to afford built-in failure protection. Ring configurations afford high failure survivability because every pair of ring nodes is connected by two physically diverse paths. As a consequence, no single node or link failure disconnects the ring.

In the event of a failure, all traffic, except that which terminates at a failed node, can be

[†]A condensed version of this paper was presented at ACM Symposium on Applied Computing [1].

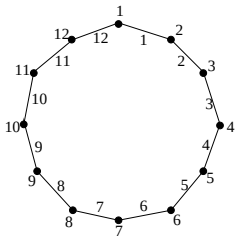


Figure 1: A 12 node ring.

diverted around the failure as long as the links have enough capacity. In practice, rings are available in only a few discrete sizes with all links of the same capacity. Thus, when we refer to a ring's capacity, we are implicitly referring to the capacity of each of its links. Not surprisingly, larger rings cost more.

In designing a network that includes rings, we'd naturally choose the cheapest rings that afford full survivability from single failures. To do this, we first determine the capacity required to assure survivability, then select the smallest available ring size that meets or exceeds this requirement. Determining the capacity required for survivability depends on the particular type of ring being constructed. In the rings that we consider[1], the capacity required for survivability depends on the routing of the demands around the ring. Given a particular routing for each demand (clockwise or counter-clockwise), the smallest capacity required to assure survivability is linearly related to the maximum amount of traffic routed on any link. Thus, the question becomes: how do we route the demands so as to minimize the maximum amount of traffic on a link? The process of assigning each demand to one of two possible routes around the ring is known as *ring loading*. The *ring loading problem* is the related optimization problem that minimizes the maximum traffic on a link. The ring loading problem and its application to SONET ring sizing are presented by Cosares and Saniee [3]. (See also [4].) Cosares and Saniee [3] prove that ring loading is NP-complete and propose several heuristics for obtaining approximate solutions.

The fact that the ring loading problem is NP-complete makes it a logical candidate for a heuristic search methods like genetic algorithms. Moreover, the binary nature of the decision provides an immediate encoding for a genetic algorithm. As a result, GAs can be implemented in a very natural way. To our knowledge, ring loading is a new application for genetic algorithms, which can be very successful on realistic-sized instances. Goldberg [5] notes that evolutionary algorithms have already been incorporated into SONET network design at US West to assist in determining which nodes to group together on a ring.

This paper is organized into three main parts: (1) a mathematical description of the ring loading problem; (2) outlines of the heuristic methods we apply; and (3) computational results comparing GA solutions with optimal solutions obtained by CPLEX [6] and approximate solutions obtained by a heuristic implemented in the SONET Toolkit [7] – a decision support system used by transport planners in the Bell regional operating companies.

[1] Bidirectional rings with time-slot interchange capability.

2 The Ring Loading Problem

The ring loading problem is defined on a network whose nodes are connected to form a ring, as shown in Figure 1. We consider the nodes to be indexed from 1 to n going clockwise around the ring. Similarly, the links are indexed from 1 to n so that index i corresponds to the link on the clockwise side of node i.

Given a set $\mathcal{D}$ of demands between nodes on the ring, the ring loading problem seeks a routing for each demand – either clockwise or counter-clockwise – such that the maximum load on a link is minimized. A demand $k \in \mathcal{D}$ is defined by a pair of nodes i_k and j_k and some amount d_k of traffic between them. We assume that d_k is nonnegative and integer. A particular demand k, partitions the links of the ring into two sets: those that are traversed in a clockwise routing of demand k and those that are traversed in a counter-clockwise routing. Let $C(k)$ be the clockwise links, and let $CC(k)$ be the counter-clockwise links. Given these definitions, the ring loading problem is

$$\text{[RL]:} \quad \text{minimize} \quad z$$

$$\begin{aligned}
\text{subject to:} \quad & x_k + \tilde{x}_k = 1, \quad \forall k \in \mathcal{D} && (1)\\
\sum_{l \in C(k)} d_k x_k + & \sum_{l \in CC(k)} d_k \tilde{x}_k \leq z, \quad \forall l = 1, \ldots, n && (2)\\
& x_k,\ \tilde{x}_k \in \{0,1\} \quad \forall k \in \mathcal{D}. && (3)
\end{aligned}$$

The constraints (1) ensure that each demand is routed. The left-hand side of each constraint (2) computes the traffic on a particular link l. Hence, the objective value z will be the maximum amount of traffic on any link of the ring.

Since [RL] is formulated using binary variables, a particular demand is routed either entirely clockwise or entirely counter-clockwise. By relaxing this "all-one-way" requirement, we could formulate the ring loading problem to allow some portion of a demand to route clockwise and the remainder to route counter-clockwise. Frank, Nishizeki, Saito, Suzuki, and Tardos [8] analyze the ring loading problem when the demands may be split in an "integer way". Earlier results of Frank [9] (for a more general problem) provide a polynomial time algorithm for this version of ring loading. If the demands may be split arbitrarily, then ring loading is equivalent to solving a linear program and, therefore, polynomially solvable. Shulman, Vachani, Ward, and Kubat [4] have proposed an algorithm that exploits the special structure of the underlying network to solve the continuous-variable ring loading problem extremely efficiently. So, although ring loading may be viewed as a mathematical program in binary, integer, or continuous variables, the binary ring loading problem is the only one that is computationally difficult. Hence, it is this problem that we consider in our study.

3 Heuristics for Binary Ring Loading

Cosares and Saniee [3] consider the (binary) ring loading problem in the context of an interactive planning tool for SONET networks. Thus, they advocate heuristic approaches for two primary reasons: 1) the binary ring loading problem is computationally intractable; and 2) a ring loading subroutine could, potentially, be called a large number of times during a network planning session. Cosares and Saniee [3] propose several heuristics, but we include only the two-phase greedy heuristic our study because it is the basis of the one currently implemented in the SONET Toolkit [7].

The two-phase greedy heuristic is most easily described by first presenting a basic greedy (one-phase) approach. The basic greedy algorithm is: given a set of demands sorted in non-increasing order of demand size, route them one at a time so that the traffic on the busiest link is increased the least. The two-phase greedy algorithm is simply the basic greedy algorithm calling itself to break ties. Specifically, the algorithm is stated as:

- For each demand, route in the direction that causes the current maximum link load to increase the least.

  If there is a tie, do the following:

  - Temporarily route the current demand clockwise, then route all other demands according to the basic greedy algorithm, breaking ties arbitrarily.
  - Temporarily route the current demand counter-clockwise, then route all other demands according to the basic greedy algorithm, breaking ties arbitrarily.
  - Permanently route the current demand in the direction that yielded the best final link load.

Our results verify that this algorithm is extremely fast and that its solutions are relatively good. However, the variability in the quality of the solutions that it produces suggests that there may also be a place for alternative methods such as genetic algorithms.

3.1 Genetic Algorithm

Genetic algorithms, developed by John Holland [10], are search procedures based on the mechanics of natural evolution. Genetic algorithms try to strike a balance between exploration and exploitation during search. However, the fitness proportionate selection criterion used in standard genetic algorithms may sometimes lead to premature convergence [11]. One way to reduce premature convergence is to use a variant of the standard genetic algorithm called the steady-state genetic algorithm. In this study, we employ a steady-state genetic algorithm called GENITOR[2] that was developed by Whitley [11].

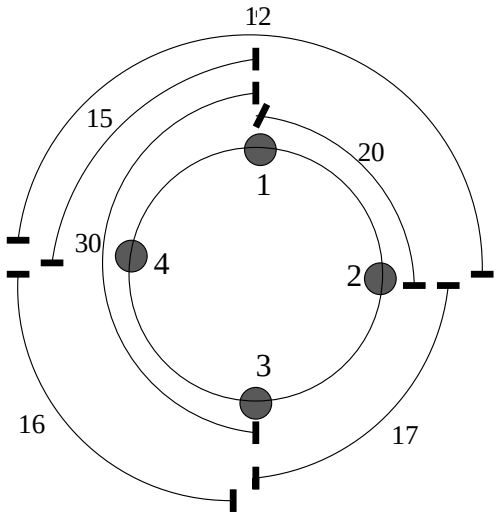
[2]GENITOR is available though the Computer Science Department at Colorado State University.

The objective for the ring loading problem is to minimize the maximum load on the links of a ring. Thus, for each individual in the population, a fitness value is assigned based on the maximum link load it induces. The particular fitness function we use is

$$\frac{1}{\max_l \{\text{load on link } l\}}.$$

In this study, we use a standard binary encoding because it maps directly to the routing of the demands. Here, the number of bits in a string is equal to the number of demands to be routed. A value of **1** in position k can be interpreted as routing load d_k in the clockwise (**C**) direction, and a value of **0** as routing in the counter-clockwise (**CC**) direction. This is illustrated in Figure 2.

Endpoints	Amount	Direction
(1,2)	20	C
(1,3)	30	CC
(1,4)	15	CC
(2,3)	17	C
(2,4)	12	CC
(3,4)	16	C



GA Encoding: 1 0 0 1 0 1
Maximum load is 57 on link 1-4.

Figure 2: An example of a routing with its GA encoding.

4 Test Cases

We evaluate the utility of the genetic algorithm by examining its performance on eight different sets of test problems. The test sets arise by considering two different ring sizes with four different demand distributions.

The rings in our tests contain either 10 or 25 nodes. These sizes are selected to examine both "ordinary" and "extreme" cases. A ring in a telecommunications network will typically contain between 5 and 15 nodes. Thus, we consider the 10 node rings to be ordinary-sized rings and the 25 node rings to be extremely large rings.

We generate the demand amounts d_k randomly from either a uniform or a bimodal distribution. The uniformly distributed tests are sampled from either a low variance range of 1-100 or a higher variance range of 1-500. The bimodal case is constructed by sampling from two separate intervals within the 1-500 range. The bimodal sample has 80% of its demands between 1 and 50 and the remaining 20% between 400 and 500.

Three of our demand cases include positive demand between all possible pairs of nodes. For a ring with n nodes, there are $n * (n-1)/2$ possible source and destination pairs. We consider a set of demands to be *complete* if it has a positive demand between every possible pair of nodes, and *partial* if it has positive demand between only a subset of the possible pairs. Thus, three of our test cases have complete sets of demands. These cases are formed by sampling from one of the three demand distributions described above – uniform low variance, uniform high variance, and bimodal.

We expect that many rings in telecommunication networks will not carry traffic between all possible pairs of nodes. For this reason, our remaining demand case includes only a partial set of demands. Partial sets of demands are generated by randomly fixing some of the possible demands to zero. In this case, we fix half of the possible demands to zero and sample the remaining demands uniformly from the low variance range.

To summarize, the four demand cases we consider are:

- **Case 1**: Complete set of demands between 1 and 100 with uniform distribution (C1_).
- **Case 2**: Half of the demands in Case 1 set to zero (C2_).
- **Case 3**: Complete set of demands between 1 and 500 with uniform distribution (C3_).
- **Case 4**: Complete set of demands between 1 and 500 but with a bimodal distribution (C4_).

Note that cases 1, 3 and 4 have complete demand sets, while case 2 has a partial set.

We generate 10 different problem instances for each case. This yields 40 instances for each ring size. For convenience, they are labeled Ci_j, where $1 \leq i \leq 4$ represents the demand case and $1 \leq j \leq 10$ represents the instance within a case. For example, C1_10 represents the 10th instance of the test case 1.

5 Computational Results

Tables 1 and 2 summarize results obtained by applying the two-pass algorithm, CPLEX, and GENITOR to 10 and 25 node problems on a Sun4m workstation. Since the two-pass algorithm is by far the fastest of the three methods, our intent is mainly to evaluate the *quality* of the solutions obtained by the two heuristics. To do this, we compare two-pass

and GENITOR solutions to optimal solutions obtained by CPLEX [6] – a widely-used commercial solver for linear and mixed integer linear programs.

5.1 A Discussion of CPLEX Settings

The ring loading problem includes integer variables, so we use CPLEX's mixed integer solver, which employs a branch and bound algorithm. We invoke CPLEX from within our own program via the CPLEX callable library. We use all default options with the exception that we specify the *absolute mipgap* to be 0.99 and the *relative mipgap* to be 10^{-9}. These settings save CPLEX needless search but do not omit any potentially optimal solutions.

To clarify how these settings are used, we briefly describe the branch and bound algorithm. (A more detailed discussion appears in [12].) The branch and bound method solves integer programs by solving a sequence of linear programs. The first linear program (LP) is obtained by relaxing all of the integrality constraints to allow fractional solutions. For [RL] the constraints (3) are replaced by the simple bound constraints:

$$0 \leq x_k,\ \tilde{x}_k \leq 1 \qquad \forall k \in \mathcal{D}.$$

Thus, the solution for this LP may contain fractional variables. In trying to solve [RL], we create the next LP by fixing one of the fractional variables to either 0 or 1. Thus, each fractional variable can potentially yield two linear subproblems. The optimal solutions for each of these LPs may also include fractional variables which can, in turn, be fixed. This process of identifying and fixing fractional variables is called branching. One can now see that the linear subproblems form a tree with the original linear relaxation at the root. Branch and bound proceeds in this way – fixing variables and solving linear programs – until some LP yields an all-integer solution.

When an integer solution is discovered, it provides an upper bound on the optimal integer solution. Alternatively, the optimal value of an LP subproblem provides a lower bound on the value of any integer solution that can be its descendent. So, if the least upper bound is lower than an LP subproblem solution, the LP cannot possibly yield a better integer solution, so branch and bound explores this branch no further. In this way, branch and bound can avoid searching fruitless branches. When there remain no branches that can potentially yield a better integer solution, branch and bound will have either found the optimal integer solution or determined that the problem is infeasible. (Note that [RL] is never infeasible.)

This description illustrates the fact that branch and bound is an exponential search algorithm. Thus, naively applying branch and bound to large integer programs may result in extremely long solution times.

We specify the absolute mipgap parameter in CPLEX to further reduce needless search. At any iteration of branch and bound, the absolute mipgap is the difference between the best upper bound and the best lower bound. The relative mipgap is the absolute gap divided by the lower bound plus 1.0. Both of these values provide stopping criteria for CPLEX.

We set the relative gap at its lowest setting to prevent stopping at a suboptimal solution. Alternatively, we set a high (.99) value for the absolute gap because we know that the objective value for the ring loading problem must be integer. Thus, we know that CPLEX has discovered an optimal solution as soon as it finds an integer solution that is within 1.0 of its lower bound.

5.2 GENITOR Settings

We use the *Reduced Surrogate Crossover* operator and the *Adaptive Mutation* operator implemented in the GENITOR package. For all our experiments, the mutation rate is 0.15, and the crossover probability is 0.20.

For 10 node rings the results are generated using a population of 500; for 25 node rings the population is 1000. Since the final solution produced by the GENITOR algorithm is influenced by the randomly generated initial population, we perform the search 50 times for each instance. Thus, the results reported for the GENITOR algorithm are based on summary statistics from these experiments.

The stopping criterion for GENITOR is that a pre-specified number of evaluations be performed. For the 10 node problems, GENITOR performs 10,000 evaluations. For 25 node problems it does 100,000 evaluations. These limits are selected based upon preliminary observation of several different values. Usually, it is the case that the best solution is observed well before reaching this limit. The timings reported in Tables 1 and 2 provide insight into the average time it takes to reach the best solution. Since it is also of interest to know worst-case timings, we note that the maximum time required to reach the evaluation limit in the 10 node problems is 9.00 seconds, while the maximum time required in the 25 node problems is 542.78 seconds.

5.3 Observations

In each of the 10 node problems, the genetic algorithm finds an optimal solution at least once. The objective mean values and standard deviations indicate that there is little variability in the quality of the GA solution from one trial to the next. For several problems, the genetic algorithm obtains an optimal solution in every trial. In most cases, the two-pass algorithm is within 10% of the optimum. However, it is off by as much as 25% for example *C4_3*. In many of the 10 node instances, we observe that the worst solution obtained by GENITOR is at least as good as the two-pass solution.

For some of the 25 node problems, CPLEX reaches its branch and bound node limit before it discovers an optimal solution. When this happens, we are sometimes able to generate optimal solutions by adjusting additional CPLEX parameters, or by adding some constraints to the basic formulation. When the optimal solution is obtained by one of these alternate means, we provide only the solution but no CPU time in Table 2. In a few cases, none

of these strategies yields a confirmed optimal solution. When no solution can be proven optimal, the value reported in Table 2 is simply the best solution observed, which is an upper bound on the true optimal value.

Once again, the best genetic algorithm solution often matches the best solution obtained. Although the GA solutions still appear to be quite robust, there is more variability between its best and worst solutions. In contrast, the two-pass algorithm performs more consistently on large problems than it does on the small ones. In every case, it is within 11% of the best solution observed.

Figures 3 and 4 illustrate the relative quality of the best values obtained by each of the three methods. Each of the figures includes four different problem groups corresponding to the four different demand cases. For each demand case there are ten observations corresponding to the ten different instances. The graphs depict the relative deviations from the best solution observed for each instance. There is little discernible difference between the CPLEX and GENITOR best solutions, but there is often a noticeable difference between the one-pass solution and the best solution observed. We note, however, that the best GENITOR solution is based upon 50 trials, and the CPLEX solution may also be based upon several trials.

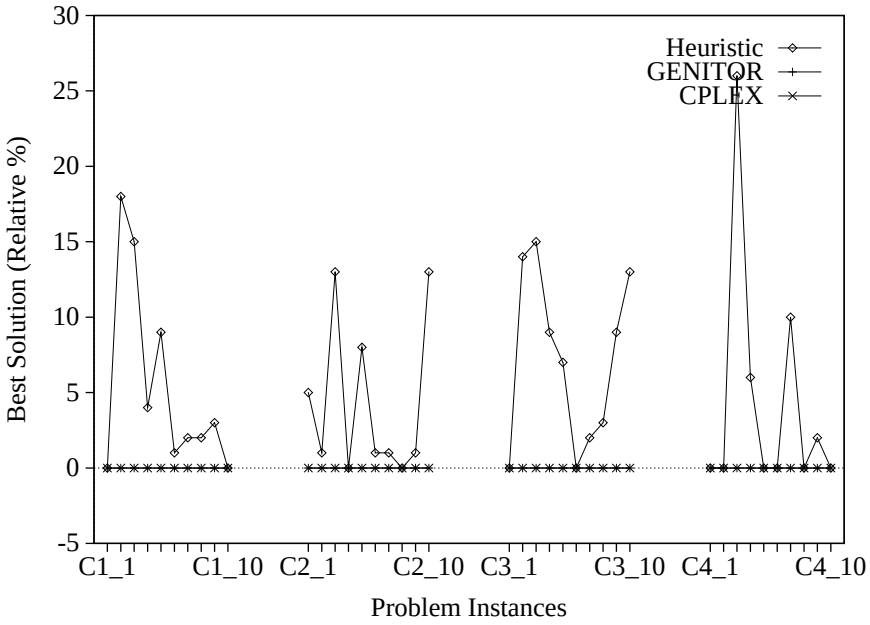


Figure 3: Comparative summary of best results for 10 node examples.

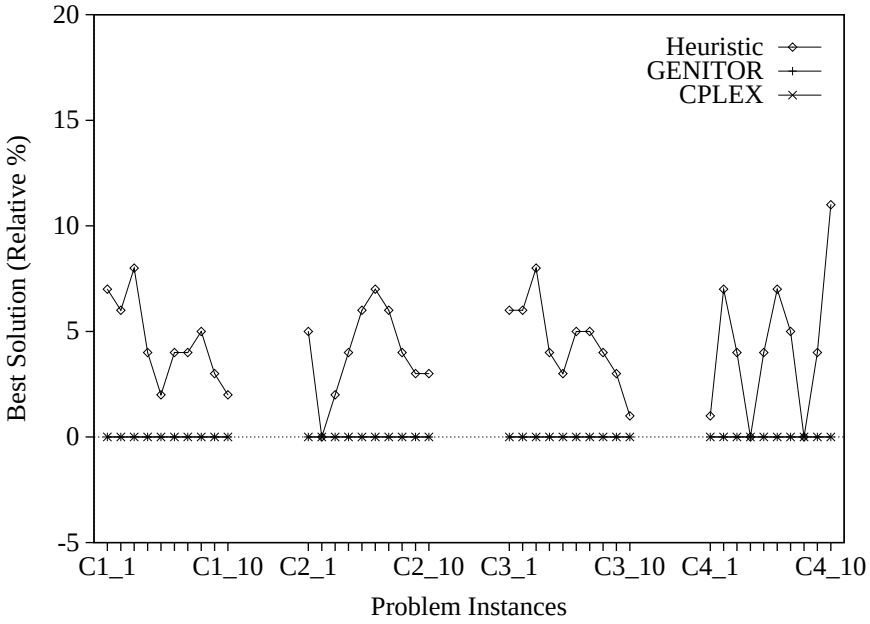


Figure 4: Comparative summary of best results for 25 node examples.

5.3.1 More Detailed Observations with CPLEX

Four problems are not solved using any of the CPLEX options specified because they reach the branch and bound node limit of 20,000 before any solution can be proved optimal. For these problems more detailed information is given in Tables 3 and 4. One of these problems – problem $C3\_4$ – is solved using a different formulation. Thus, its optimal value is given in Table 2.

Table 3 contains the objective value obtained by CPLEX, with different combinations of options, after examining 20,000 branch and bound nodes. We allow alternative settings for three different options. We consider turning the CPLEX aggregator "on" and "off". We also consider changing the branch direction to preferentially branch up the branch and bound tree in the hope of limiting tree size, and we alter the node selection strategy to use a "best-estimate" rather than a "best-bound" strategy. By default, the node selection strategy is "best-bound" and the branching direction is based on the magnitude of the branching variable's integer infeasibility, which is always 0.5 in ring loading. For more detail on these and other CPLEX options, we refer the reader to the CPLEX documentation [6].

While we are sometimes able to solve different subsets of our test set by adjusting the CPLEX parameters, no one set of parameters performs consistently best. Therefore, we report the results in Tables 1 and 2 based on the default settings. We consider resetting either the branching direction or the node selection strategy with the aggregator "on" and

with the aggregator "off". In all, there are six cases that correspond to the following sets of options:

Option Set 1: Aggregator = on; branching direction = default;
node selection = best-bound.

Option Set 2: Aggregator = on; branching direction = up;
node selection = best-bound.

Option Set 3: Aggregator = on; branching direction = default;
node selection = best-estimate.

Option Set 4: Aggregator = off; branching direction = default;
node selection = best-bound.

Option Set 5: Aggregator = off; branching direction = up;
node selection = best-bound.

Option Set 6: Aggregator = off; branching direction = default;
node selection = best-estimate.

The objective value of the best integer solution is provided in Table 3 and the corresponding CPU time is given in Table 4. The *lower bound* provided in Table 3 is the tightest lower bound across the cases considered.

We note that any instance (including those in Table 2) that reaches the branch and bound node limit of 20,000 takes a long time to solve because CPLEX must solve 20,000 linear programs. The times provided in Table 4 give a sense for how long it takes to reach this limit. Tables 3 and 4 together illustrate the main drawback in naively applying standard optimization software to solve these types of problems: spending a long time and examining many thousands of branch and bound nodes does not necessarily ensure a "good" solution. Branch and bound is an exponential search algorithm. Although it is assured of finding an optimal solution if it terminates with an absolute mipgap less than one, it may not find a good or even feasible solution if it stops because it reaches a node or time limit.

6 Conclusions

Our results emphasize the fact that GENITOR is robust not only in the quality of it's solutions, but also in the time it takes to obtain them. Its running time is sensitive to the number of demands populating the ring, but it is relatively insensitive to the specific problem instance and the underlying demand distribution. The low variability in both solution time and quality are important features for software that is to be used interactively.

In our study, we do not attempt to tune the CPLEX parameters for our particular problem structure. We also do not attempt to provide CPLEX a stronger formulation by adding

constraints to the basic formulation in Section 2. These strategies often yield large improvements in the performance of integer program solvers, and might provide an alternate avenue for research.

Our main goal has been to demonstrate the suitability of genetic algorithms for solving the binary ring loading problem. We feel that such algorithms have merit in this context because ring loading yields an immediate binary encoding and because these algorithms produce solutions of consistently high quality. Thus, applying a genetic algorithm to this problem is simple to do and yields good results.

Our comparisons with the two-pass algorithm show that there is a tradeoff between time and solution quality. Currently, the SONET Toolkit opts for fast solutions because it evaluates many potential rings. Once a ring has been selected it may be advantageous to size it based on a higher quality solution. For a 10-node ring with a complete set of demands, the cost of computing such a solution is roughly five seconds. Thus, we can easily imagine using GA in the context of a two-phased planning process.

It may still be possible to reduce the time required by the genetic algorithm by adjusting some of its internal parameters. Reducing the size of the population is one mechanism to reduce the solution time. Another option is to introduce an additional termination criterion based on the LP lower bound for [RL]. The intent of such a criterion is to stop the genetic algorithm as soon as it discovers a solution that is within a prescribed tolerance of the lower bound. The special structure of rings makes the LP lower bound extremely easy to compute [4]. Thus, such a termination strategy is easy to incorporate and is likely to reduce computation time but still provide high-quality solutions. Genetic algorithms will become an attractive solution alternative if computing times can be reduced without impacting the quality of the solution.

Finally, we note that in practice rings are built in a few standard sizes that correspond to the SONET standard transmission rates. Thus, we cannot assume that we can build rings in *any* discrete size. The implication of this is that solution inaccuracies may have little effect on cost if they do not require moving to a larger-sized ring, but are extremely costly when a larger ring than needed is built. Further studies with real data are needed to determine the cost associated with solution inaccuracies.

Test Cases	Heuristics		CPLEX		GENITOR				
	Link Capacity	CPU Time Sec	Link Capacity	CPU Time Sec	Link Capacity			CPU Time Sec	
					Mean	Best	Std	Mean	Std
C1_1	584	0.03	584	0.77	584.0	584	0.0	4.13	0.49
C1_2	881	0.02	749	0.55	749.7	749	3.3	4.49	0.61
C1_3	759	0.03	661	5.37	662.3	661	1.8	5.86	1.05
C1_4	726	0.03	701	1.62	701.4	701	1.6	4.73	0.74
C1_5	721	0.02	660	2.77	660.8	660	1.3	5.26	0.97
C1_6	733	0.03	726	0.67	726.2	726	0.7	5.50	0.96
C1_7	729	0.02	712	2.48	712.5	712	1.1	5.06	0.69
C1_8	715	0.02	702	1.53	702.6	702	2.6	5.15	1.08
C1_9	666	0.03	644	0.70	645.6	644	4.8	5.31	0.89
C1_10	715	0.03	712	1.02	712.8	712	2.0	4.94	0.87
C2_1	329	0.02	312	0.28	312.2	312	0.4	2.77	0.71
C2_2	402	0.00	399	0.90	400.2	399	1.5	2.63	1.10
C2_3	434	0.00	383	0.28	384.4	383	3.1	2.79	0.97
C2_4	346	0.00	346	0.45	346.6	346	1.8	2.62	0.85
C2_5	487	0.00	449	0.15	449.1	449	0.3	2.31	0.54
C2_6	429	0.00	424	0.20	425.3	424	3.6	2.98	1.02
C2_7	409	0.00	403	0.42	403.7	403	1.8	2.97	1.17
C2_8	474	0.02	473	0.80	474.0	473	2.7	2.99	1.17
C2_9	350	0.00	347	0.85	347.0	347	0.4	2.29	0.78
C2_10	517	0.02	456	0.62	456.6	456	2.9	2.53	0.57
C3_1	2895	0.03	2893	0.95	2893.4	2893	0.0	4.65	0.75
C3_2	4236	0.03	3722	0.92	3726.9	3722	19.8	4.51	0.66
C3_3	3771	0.03	3277	4.98	3284.8	3277	9.9	5.88	1.13
C3_4	3786	0.02	3469	1.57	3469.7	3469	4.1	4.87	0.89
C3_5	3504	0.03	3273	2.65	3277.5	3273	6.8	5.11	0.92
C3_6	3615	0.03	3605	4.10	3607.4	3605	4.6	5.73	1.21
C3_7	3611	0.02	3530	3.65	3533.1	3530	4.9	5.28	0.84
C3_8	3575	0.02	3486	3.53	3488.8	3486	5.9	5.39	0.90
C3_9	3472	0.03	3196	1.20	3199.8	3196	17.4	5.05	1.05
C3_10	3982	0.02	3530	4.10	3534.9	3530	8.4	5.03	0.88
C4_1	1865	0.03	1863	0.87	1863.2	1863	0.0	4.73	0.73
C4_2	1859	0.03	1858	1.47	1858.0	1858	0.0	4.48	0.52
C4_3	2036	0.03	1620	0.38	1620.0	1620	0.0	4.45	0.64
C4_4	2120	0.03	2000	1.25	2000.0	2000	0.0	4.69	0.78
C4_5	1939	0.02	1932	0.75	1932.1	1932	1.0	4.74	1.07
C4_6	1659	0.02	1658	0.57	1658.0	1658	0.0	4.92	0.88
C4_7	2102	0.03	1907	0.67	1912.7	1907	19.3	5.37	0.92
C4_8	1858	0.00	1857	0.87	1868.8	1857	12.2	4.88	0.79
C4_9	2048	0.02	2009	4.35	2009.0	2009	0.0	5.10	0.99
C4_10	1777	0.02	1770	1.78	1770.0	1770	0.0	4.59	0.61

Table 1: A Summary of Results for the Ring with 10 Nodes.

Test Cases	Heuristics		CPLEX		GENITOR				
	Link Capacity	CPU Time Sec	Link Capacity	CPU Time Sec	Link Capacity			CPU Time Sec	
					Mean	Best	Std	Mean	Std
C1_1	4391	1.98	4110	19.78	4161.5	4119	28.8	447.64	39.67
C1_2	4150	2.02	3911	22.60	3933.5	3911	19.1	454.10	44.19
C1_3	4476	1.93	4136	19.12	4207.8	4153	31.1	448.11	48.57
C1_4	4353	2.03	4202	-	4234.9	4204	25.7	457.50	37.23
C1_5	4206	2.15	4122	29.73	4137.3	4122	15.3	427.31	42.73
C1_6	4297	2.20	4128	22.18	4181.9	4141	30.6	452.74	39.39
C1_7	4182	2.05	4018	-	4078.0	4023	22.7	454.57	36.95
C1_8	4335	2.03	4142	46.30	4196.4	4156	30.6	448.99	36.50
C1_9	4182	2.00	4075	331.98	4124.9	4082	25.2	456.40	39.89
C1_10	4093	1.93	4010	23.72	4020.9	4010	15.6	430.00	45.93
C2_1	2246	0.40	2147	-	2147.1	2147	1.6	118.92	13.86
C2_2	1946	0.52	1946	-	1947.9	1946	3.9	111.63	14.77
C2_3	2381	0.57	2323	31.93	2325.6	2323	4.5	137.38	17.75
C2_4	2134	0.58	2050	122.85	2062.2	2052	5.8	126.71	15.46
C2_5	2366	0.45	2235†	-	2242.9	2235	14.7	138.90	18.91
C2_6	2199	0.45	2049	58.58	2051.8	2049	5.1	128.11	14.92
C2_7	2164	0.47	2051	5.72	2051.2	2051	1.4	118.26	21.96
C2_8	2110	0.47	2026	551.27	2032.3	2026	10.2	141.44	17.97
C2_9	2237	0.45	2179	13.28	2181.4	2179	5.2	134.08	16.77
C2_10	2201	0.52	2136	22.70	2136.3	2136	0.8	123.28	16.42
C3_1	21594	2.05	20391	202.60	20631.5	20401	157.3	451.10	42.39
C3_2	20559	1.95	19399	38.65	19494.4	19399	81.8	472.95	35.09
C3_3	22059	1.72	20516	38.35	20912.0	20543	179.4	450.67	39.99
C3_4	21765	1.73	20853	-	21049.9	20854	109.9	459.15	38.22
C3_5	21061	2.10	20453	558.07	20549.6	20453	93.8	449.98	40.70
C3_6	21562	2.22	20477	54.58	20768.1	20552	138.6	465.62	38.97
C3_7	20847	2.02	19922	35.82	20186.7	19930	145.2	459.93	35.97
C3_8	21358	1.98	20547	879.58	20801.0	20551	155.7	458.11	45.53
C3_9	20906	1.95	20216	-	20422.4	20216	141.8	473.11	31.57
C3_10	20047	2.07	19886	29.53	19931.7	19886	64.5	459.75	36.17
C4_1	9598	1.87	9523	16.72	9553.9	9523	22.5	472.89	39.18
C4_2	10155	2.00	9495	11.90	9517.6	9495	43.6	490.26	37.86
C4_3	9769	1.97	9354	16.13	9435.9	9362	84.1	484.48	44.67
C4_4	10224	1.75	10195	24.82	10204.7	10195	19.6	481.91	46.75
C4_5	9993	1.90	9594	24.72	9616.5	9594	41.0	460.86	38.79
C4_6	10827	2.13	10094	118.77	10096.4	10094	3.4	488.70	40.54
C4_7	9522	1.88	9061	19.43	9070.0	9061	15.3	489.26	31.02
C4_8	9476	1.85	9466†	-	9478.7	9466	20.9	479.25	41.29
C4_9	10512	2.02	10089†	-	10128.6	10089	40.5	487.00	40.84
C4_10	11676	2.02	10532	21.57	10535.3	10532	11.6	458.96	53.21

Table 2: A Summary of Results for the Ring with 25 Nodes. †: Solution not confirmed optimal.

Test Cases	Lower Bound	CPLEX Options Set					
		1	2	3	4	5	6
C2_5	2231	2324	2364	2235	2236	2438	2236
C3_4	20853	21220	23986	20943	21220	25582	20943
C4_8	9415	9467	13176	9466	9502	11497	9467
C4_9	10066	10141	12324	10089	10264	12433	10089

Table 3: Objective values for problems not solved.

Test Cases	CPLEX Options Set					
	1	2	3	4	5	6
C2_5	1423.58	1328.45	1180.47	1134.85	1324.08	1240.65
C3_4	2875.30	2910.98	2771.02	3011.58	2833.93	2870.33
C4_8	2762.55	2819.13	2852.77	2632.48	2955.73	2566.93
C4_9	2362.68	2366.08	2209.40	2358.22	2668.75	2493.60

Table 4: CPU times for problems not solved.

Acknowledgement.

We thank Steve Cosares and Iraj Saniee for introducing us to the ring sizing problem, and we thank Clyde Monma, Joshua Alspector, Stu Personick, and David Ackley for their comments on an earlier draft of this paper. The second author thanks Leslie Hall for pointing out references [8] and [9] and for many fun and elucidating discussions on the topic of rings.

References

[1] N. Karunanithi and T. Carpenter. A Ring Loading Application of Genetic Algorithms, *Proceedings of the ACM Symposium on Applied Computing*, pp. 227–231, 1994.

[2] R. H. Cardwell and J. O. Eaves. SONET: A ten-year perspective. *Bellcore Exchange*, 12–17, April, 1994.

[3] S. Cosares and I. Saniee. An optimization problem related to balancing loads on SONET rings. *Telecommunication Systems*, 3:165–181, 1994.

[4] A. Shulman, R. Vachani, J. Ward, and P. Kubat. Multicommodity flows in ring networks. Technical Report. GTE Laboratories, Waltham, MA, 02254, 1991.

[5] D. E. Goldberg. Genetic and evolutionary algorithms come of age, *Communications of the ACM*, Vol. 37, No. 3, pp. 113–119, 1994.

[6] CPLEX Optimization, Inc. *Using the CPLEX Callable Library and CPLEX Mixed Integer Library*, Version 2.1, 1993.

[7] S. Cosares, I. Saniee and O. Wasem. Network planning with the SONET Toolkit. *Bellcore Exchange*, 8–13, September/October, 1992.

[8] A. Frank, T. Nishizeki, N. Saito, H. Suzuki, and E. Tardos. Algorithms for routing around a rectangle. *Discrete Applied Mathematics*, 40:363–378, 1992.

[9] A. Frank. Edge-disjoint paths in planar graphs. *Journal of Combinatorial Theory, Series B*, 39:164–178, 1985.

[10] J. H. Holland. *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, 1975.

[11] D. Whitley. The GENITOR Algorithm and Selective Pressure: Why Rank-based Allocation of Reproductive Trials is Best. *Proceedings of the Third Conference on Genetic Algorithms, 1989.*, Washington, D. C., pp. 116-121, Morgan Kaufmann, Publishers.

[12] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*, John Wiley and Sons, Inc., 1988.