Satellite Range Scheduling: A Comparison of Genetic, Heuristic and Local Search

L. Barbulescu, A.E. Howe, J.P. Watson and L.D. Whitley

Computer Science Department
Colorado State University
Fort Collins, CO 80523 USA
{laura,howe,watsonj,whitley}@cs.colostate.edu

Abstract. Three algorithms are tested on the satellite range scheduling problem, using data from the U.S. Air Force Satellite Control Network; a simple heuristic, as well as local search methods, are compared against a genetic algorithm on old benchmark problems as well as problems produced by a generator we recently developed. The simple heuristic works well on the old benchmark, but fails to scale to larger, more complex problems produced by our generator. The genetic algorithm yields the best overall performance on larger, more difficult problems.

1 Problem Description

The U.S. Air Force Satellite Control Network (AFSCN) is responsible for coordinating communications between users and satellites in space. A key mission of the AFSCN is *satellite range scheduling* (SRS), which involves scheduling communications between users on the ground and more than 100 satellites in space. All communications are performed via nine ground stations located around the globe, with an aggregate of sixteen antennas. The AFSCN scheduling center typically receives over 500 user requests for a single day.

Each user request specifies at a minimum an antenna at a particular ground station, a required duration, and a time window within which the duration must be allocated. Requests are classified as either low or high-altitude, corresponding to the orbit of the target satellite. The durations of low-altitude requests are typically equal to the visibility windows, leaving little scheduling flexibility. In contrast, high-altitude satellites are visible to more ground stations for longer periods of time. Consequently, high-altitude requests often specify alternative antennas and/or visibility windows. The objective of the SRS problem is to minimize the number of unsatisfied requests.

The SRS problem is NP-complete:[1] a reduction of the SRS problem to one resource can be shown to be equivalent to a well known NP-complete problem in the scheduling literature, denoted $1|r_j|\sum U_j$ in the three-field notation widely used by the scheduling community. The SRS problem is also over-subscribed in the sense that all requests can rarely be scheduled; to satisfy all user requests,

[1] We are currently working on a paper which presents an NP-completeness proof.

some form of arbitration process is required. Several algorithms for related over-subscribed scheduling problems have been reported in the literature (e.g., see [6] [11] [13]), but none directly address the peculiarities of the satellite range scheduling problem, including alternative resources and/or time-windows.

Researchers at the Air Force Institute of Technology (AFIT) have developed a number of algorithms for the SRS problem. Gooley and Schalck introduced an algorithm based on a combination of mixed integer programming (MIP) and insertion heuristics [4] [8], which scheduled between 91% and 95% of user requests for small problem instances. Later, Parish used a genetic algorithm called *Genitor* to solve the SRS problem [5]. *Genitor* out-performed the MIP approach, nominally scheduling 96% of user requests.

Both the MIP algorithm and the *Genitor* genetic algorithm were evaluated using the same set of seven real-world problem instances collected in 1992; we refer to this collection of instances as the "AFIT benchmark". In 1992 approximately 300 requests needed to be scheduled for a single day, compared to 500 requests per day in recent years. The need to schedule more requests has a clear impact on problem difficulty. In this paper we investigate whether the problems in the AFIT benchmarks are representative of the kinds of Range Scheduling problems that are solved in the present by AFSCN to determine whether the old results should generalize.

Currently, there is no accepted state-of-the-art algorithm for satellite range scheduling. Because it is an extremely important application, we have been engaged in a study of various algorithms for this problem. In this paper, we replicate the results reported by Parish [5] using *Genitor* to solve the AFIT benchmark problems, and investigate reasons for *Genitor*'s strong relative performance. We identify a simple heuristic that can solve all of the problems in the AFIT benchmark. Finally, we generate new problems by modeling features currently encountered by AFSCN and explore conditions where the heuristic fails. *Genitor* continues to display good results for new problems.

2 Algorithms for Satellite Range Scheduling

In this section, we document the various algorithms considered in this study. We first discuss the method of encoding solutions, and the procedure for decoding solutions into actual schedules. Next, we define the three algorithms used in our analysis: random sampling, local search under a shift neighborhood, and the *Genitor* genetic algorithm. We then conclude by briefly discussing our decision to omit two well-known families of scheduling algorithms in our analysis.

2.1 Solution Representation and Decoding

Each of the algorithms we consider represents solutions as permutations of the integers 1 through N, where N is the total number of requests to be scheduled. A permutation represents the order in which requests are given access to particular

resources. A greedy heuristic is then used to generate a schedule from a permutation, by attempting to schedule the requests in the order in which they appear in the permutation. Each request is assigned to the first available resource (from its list of alternatives), and at the earliest possible starting time. If the request cannot be scheduled on any of the alternative resources, it is dropped from the schedule (i.e., bumped). The "fitness" of a schedule is then defined as the total number of requests bumped from the schedule.

2.2 Random Sampling

Random sampling produces schedules by generating random permutations of length N. By randomly sampling a large number of schedules, we can characterize the distribution of solutions in the search space. Further, the performance of random sampling provides a baseline measure of problem difficulty.

2.3 Local Search under the Shift Neighborhood

A key component of any local search algorithm is the move operator. Because problem-specific knowledge for the SRS problem is lacking, we selected the "shift" move operator. The shift operator has been successfully applied to a number of well-known scheduling problems, such as the permutation flow-shop scheduling problem [10]. The neighborhood under the shift operator is defined by considering all $(N-1)^2$ pairs (x, y) of positions in a current solution π, subject to the restriction that $y \neq x - 1$. The neighbor π' corresponding to the position pair (x, y) is produced by *shifting* the job at position x into the position y, while leaving all other relative job orders unchanged. If $x < y$, then $\pi' = (\pi(1), ..., \pi(x-1), \pi(x+1), ..., \pi(y), \pi(x), \pi(y+1), ..., \pi(n))$. If $x > y$, then $\pi' = (\pi(1), ..., \pi(y-1), \pi(x), \pi(y), ..., \pi(x-1), \pi(x+1), ..., \pi(n))$.

Given the relatively large neighborhood size, we use the shift operator in conjunction with next-descent search. The neighbors of the current solution are examined in a random order, and the first neighbor with either a lower or equal fitness (i.e., number of bumps) is accepted. Search terminates when a pre-specified number of evaluations is exceeded.

2.4 The *Genitor* Genetic Algorithm

Genitor [12] is a "steady-state" genetic algorithm [2]. Previous studies of the SRS problem at AFIT [5] report good results when using *Genitor* in conjunction with permutation encoding of solutions. In each step of *Genitor*, a pair of solutions is selected and used to generate a single child, which then replaces the worst solution in the current population.

In *Genitor*, the parent solutions are selected based on the rank of their fitness, relative to other solutions in the population. A linear bias is used such that individuals that are above the median fitness have a rank-fitness greater than one and those below the median fitness have a rank-fitness of less than one.

The typical genetic algorithm encodes solutions as bit strings, enabling the use of standard crossover operators such as one-point and two-point crossover [3]. Because we encode solutions as permutations, a special crossover operator is required to ensure that the recombination of two parent permutations results in a child inheriting relevant characteristics of the two parents. We use Syswerda's (relative) order crossover operator [9], which preserves the relative order of the elements in the parents when constructing the child. Syswerda's operator has been successfully applied in a variety of scheduling applications.

2.5 Other Scheduling Algorithms

We also considered straightforward implementations of Tabu search for the SRS problem, but the performance of these algorithms was not competitive. With 500 requests, the number of neighbors under shift or swap-based move operators is roughly 500^2; consequently, Tabu search and other local search algorithms based on steepest descent are simply not practical. We briefly explored methods for reducing the neighborhood size, but in all cases the reduction in neighborhood size severely impacted algorithm performance.

Additionally, we developed constructive search algorithms based on texture-based [1] and slack-based [7] constraint-based scheduling heuristics that select the maximal subset of tasks that can be feasibly scheduled. We found that texture-based heuristics are highly effective when the size of the problem is small (e.g., less than 100 requests) and when alternative or backup requests are not considered. However, on larger problems, the consideration of alternative times makes the straightforward use of constraint-based methods ineffective.

3 The AFIT benchmark

The AFIT benchmark problems [2] were derived using the ASTRO system, a computer application developed to aid human schedulers. These problems represent the user requests and visibilities for seven days, from October 12 to October 18, 1992. The low-altitude requests in these problems can be scheduled only at one ground station (by assigning it to one of the antennas present at that ground station). The number of requests to be scheduled for the seven problems are 322, 302, 300, 316, 305, 298, and 297 respectively. We note that since 1992, the number of requests received during a typical day has increased substantially.

In our experimental setup we replicated the conditions and the reported results from Parish's study [5]. We ran *Genitor* on each of the seven problems in the benchmark, using the same parameters: population size 200, selective pressure 1.5, order-based crossover, and 8000 evaluations [3] for each run. We also ran

[2] We thank Dr. James T. Moore, Associate Professor of Operations Research at the Department of Operational Sciences, Graduate School of Engineering and Management, Air Force Institute of Technology for providing the data.

[3] An increase in the number of evaluations to 50k and of the population size to 400 did not improve the best solutions found for each problem.

Table 1. Performance of *Genitor*, local search, and random sampling on the AFIT benchmark problems, in terms of the best and mean number of bumped requests. All statistics are taken over 30 independent runs. The last column reports the performance of Schalck's Mixed-Integer Programming algorithm [8].

	Genitor			Local Search			Random Sampling			MIP
Day	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev	
1	8	8.6	0.49	15	18.16	2.54	21	22.7	0.87	10
2	4	4	0	6	10.96	2.04	11	13.83	1.08	6
3	3	3.03	0.18	11	15.4	2.73	16	17.76	0.77	7
4	2	2.06	0.25	12	17.43	2.76	16	20.20	1.29	7
5	4	4.1	0.3	12	16.16	1.78	15	17.86	1.16	6
6	6	6.03	0.18	15	18.16	2.05	19	20.73	0.94	7
7	6	6	0	10	14.1	2.53	16	16.96	0.66	6

random sampling and local search on each AFIT problem, with a limit of 8000 evaluations per run. For each algorithm, we performed a total of 30 independent runs on each problem. The results are summarized in Table 1. Included in the table are the results obtained by Schalck using Mixed Integer Programming [8]. As previously reported, *Genitor* yields the best overall performance.

To exploit the differences in scheduling slack and number of alternatives between low and high-altitude requests, we designed a simple greedy heuristic (which we call the "split heuristic") that first schedules all the low-altitude requests (in the order given by the permutation), followed by the high-altitude requests. We show that: (1) for more than 80% of the best known schedules found by *Genitor*, the split heuristic does not increase the number of conflicts in the schedule, and (2) the split heuristic typically produces good (and often best-known) schedules.

We hypothesized that *Genitor* may be learning to schedule the low-altitude requests before the high-altitude requests, leading to the strong overall performance. If true, the evaluation of high-quality schedules should, on average, remain unchanged when the split heuristic is applied. To test this hypothesis, we ran 1000 trials of *Genitor* on each AFIT problem. The results are summarized in Table 2. The second column (labeled "Total Number of Best Known Found") records the number of schedules (out of 1000) with an evaluation equal to the best found by *Genitor* in any run. We then applied the split heuristic to each such schedule. The schedules resulting from the split heuristic fall into three categories. First, the conflicts are identical to those found by *Genitor*; the number of schedules in this category is given in the third column ("Same Evaluation Same Conflicts"). Second, the evaluation is the same but the conflicts are different; the number of schedules in this category is given in column "Same Evaluation Different Conflicts". Third, the evaluation is different; the last column reports the number of schedules in this category. By separating the requests from the schedules produced by *Genitor* into low and high-altitude requests, the evaluation of more than 80% of the schedules remains unchanged. The numbers in

Table 2. The effect of applying the split heuristic when evaluating best known schedules produced by *Genitor*

Day	Total Number of Best Known Found	Same Evaluation Same Conflicts	Same Evaluation Different Conflicts	Worse Evaluation
1	420	38	373	9
2	1000	726	106	168
3	996	825	115	56
4	937	733	50	154
5	862	800	12	50
6	967	843	56	68
7	1000	588	408	4

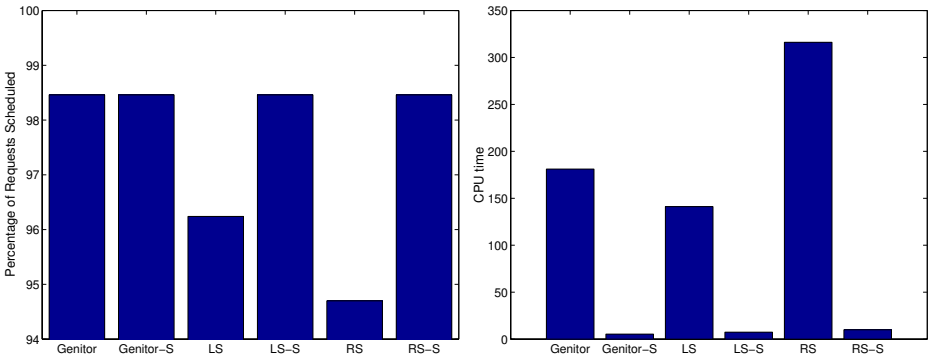

Fig. 1. Algorithm performance for the seven AFIT benchmark problems

the last column of the table also warn that when using the split heuristic only a subspace of the permutations is considered (the permutations that are separated into low and high-altitude requests); this subspace does not contain all the best-known solutions, and, in fact, for different instances of the problem this subspace could be suboptimal.

Our second hypothesis is that using the split heuristic results in solutions with a small number of conflicts. Figure 1 presents a summary of the results obtained when using the *Genitor*, Local Search and Random Sampling without the split heuristic (30 experiments, 8000 evaluations per experiment), as well as the split versions denoted by *Genitor*-S, Local Search-S and Random Sampling-S. The split versions of the three algorithms were run in 30 experiments with 100 evaluations per experiment. The minimum number of bumps in 30 experiments is recorded for each problem as the percent of requests scheduled. The left half of Figure 1 presents the average percentage of requests scheduled for the seven problems by each algorithm. The corresponding average CPU times (in seconds) appear in the right half of the figure.

Table 3. Results of running random sampling in 30 experiments, by generating 100 random permutations per experiment. A problem-specific heuristic is used in the evaluation function, where the low-altitude requests are evaluated first.

Day	Best Known	Random Sampling-S		
		Min	Mean	Stdev
1	8	8	8.2	0.41
2	4	4	4	0
3	3	3	3.3	0.46
4	2	2	2.43	0.51
5	4	4	4.66	0.48
6	6	6	6.5	0.51
7	6	6	6	0

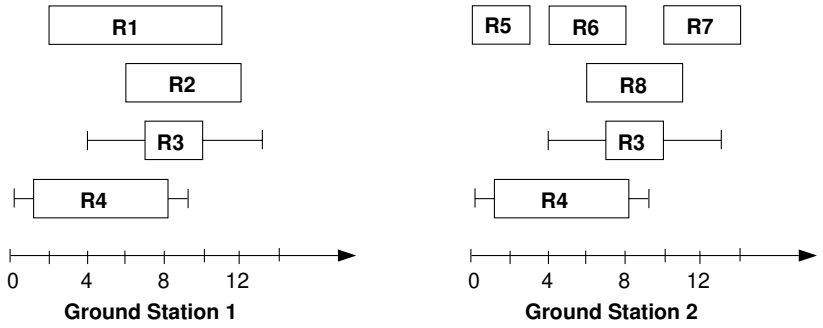

Fig. 2. Problem for which the split heuristic can not result in an optimal solution. Each ground station has two antennas; the only high-altitude requests are $R3$ and $R4$.

For all the problems, Random Sampling-S finds the best known solutions, as illustrated in Table 3. Since the best known solutions were obtained by randomly sampling a small number of permutations, solving the problems in the AFIT benchmark is easy using the split heuristic.

However, we can build a simple problem instance for which the optimal solution cannot be found using the split heuristic. Consider the problem represented in Figure 2. There are only two ground stations, and each ground station has two antennas (meaning that at each ground station at most two requests can be scheduled at the same time). There are two high-altitude requests, $R3$ and $R4$, with durations 3 and 7 respectively. $R3$ can be scheduled between start time 4 and end time 13; $R4$ can be scheduled between 0 and 9. Both $R3$ and $R4$ can be scheduled at either of the two ground stations. The rest of the requests are low-altitude requests. $R1$ and $R2$ request the first ground station, while $R5$, $R6$, $R7$, and $R8$ request the second ground station. This problem fits the description of the SRS problems in the AFIT benchmark: the low-altitude requests can be scheduled only at a specific ground station, with a fixed start and end time, while the high-altitude requests have alternative resources and a time window

specified. For all the permutation schedules, if the split heuristic is used, $R3$ and $R4$ cannot be scheduled. However, it is possible to find schedules where both $R3$ and $R4$ get scheduled, and only one request ($R1$, $R2$, or $R8$) gets bumped. The subspace containing the permutations with all the low-altitude requests before the high-altitude requests is suboptimal - the global optimum is not necessarily contained in this subspace. The example shows the potential for failure to generate optimal solutions using the split heuristic.

4 Generalizing the AFIT problems

Does the algorithm performance obtained for the AFIT benchmark transfer to larger sets of similar problems? To explore this question, we built a problem generator which produces problems similar to the AFIT benchmark but also including features encountered in the present-day real-world problems. Then we compare the results of running *Genitor*, local search and random sampling on problems produced by the problem generator to the results reported for the AFIT problems. We show that: (1) *Genitor* consistently results in the smallest number of unscheduled requests, and (2) the performance of the split heuristic on the seven AFIT problems does not transfer to the problems produced by our generator.

Two main features characterize our problem generator. First, it models different types of requests encountered in the real-world satellite scheduling problem, such as downloading data from a satellite, transmitting information or commands from a ground station to a satellite, checking the health and status of a satellite. Second, the problem generator uses models for customer behavior. The generator produces a predefined number of requests for each customer and each request type. With a 0.5 probability we determine if a request is a low-altitude or high-altitude one. For low-altitude requests, we decided to preserve the AFIT definition by assigning the duration equal to the size of the time window. However, we define alternative ground stations for both low and high-altitude requests.

To generate alternatives for a request, we collected data on the Web about the visibilities of various satellites [4] from the locations of the nine ground stations.

We repeat the experiments described for the AFIT problems by running *Genitor*, local search and random sampling for problems produced by our generator. To compare our results to the ones reported for the AFIT problems, but also to generate realistic problems, we ran the experiments for problem sizes 300, 350, 400, 450, and 500. For each size, we generated 30 problem instances.

We again ran *Genitor*, local search and random sampling, with and without the split heuristic, performing 30 runs with 8000 evaluations per run for each problem. An increase in the number of evaluations to 50k and of the population size to 400 did not improve the best solutions found for each problem. We record the number of unscheduled requests for each run. Figure 3 shows that *Genitor* on average outperforms *Genitor*-S and both versions of local search and random

[4] See: http://earthobservatory.nasa.gov/MissionControl/overpass.html for visibilities; thanks to Ester Gubbrud for helping us to compile the databases.

Table 4. The difference between the minimum number of bumps reported by an algorithm and the minimum number of bumps found by any of the six algorithms (with or without the split heuristic) is averaged over the 30 instances for each problem size

	Genitor		Local Search		Random Sampling	
Size	Mean	Stdev	Mean	Stdev	Mean	Stdev
300	0.000	0.000	0.000	0.000	0.167	0.213
350	0.000	0.000	0.333	0.368	1.067	1.099
400	0.000	0.000	1.233	1.702	2.833	3.523
450	0.000	0.000	3.667	3.678	5.967	6.240
500	0.000	0.000	8.300	3.941	11.767	7.840
	Genitor-S		Local Search-S		Random Sampling-S	
Size	Mean	Stdev	Mean	Stdev	Mean	Stdev
300	0.767	0.737	0.767	0.737	0.867	0.671
350	0.667	0.851	0.967	1.551	1.367	2.033
400	1.100	1.128	2.167	2.626	2.933	3.168
450	1.467	1.223	3.967	4.309	5.200	6.717
500	2.200	2.097	8.700	8.907	10.667	10.161

sampling. In fact *Genitor* (without the split heuristic) always outperforms all the other algorithms. In Table 4 we first subtract the minimum number of bumped requests for each problem from the minimum number of bumped requests reported by each of the algorithms (with or without the split heuristic) for that problem in 30 runs. Then we average these differences over the 30 instances generated for each size. From both Figure 3 and Table 4, it is clear that the split heuristic always results in an average decrease in performance.

5 Conclusions

Satellite Range Scheduling is an important real world problem that impacts the use of expensive and limited resources. We first considered a version of the problem studied at AFIT. For planning and experimental control purposes, we also built a problem generator that introduces new realistic features, currently encountered by the AFSCN. We show that the seven problems in the AFIT benchmark are trivial to solve when a simple heuristic is used. But, when applied to more realistic problems, the split heuristic results in poor-quality solutions. Finally, our results indicate that a genetic algorithm, *Genitor*, using a permutation representation yields the best overall performance and does so in a modest amount of time. The results also reinforce the notion that benchmarks need to be constructed or chosen to be representative for actual target applications.

References

1. J. C. Beck, A. J. Davenport, E. M. Sitarski, and M. S. Fox.: Texture-based heuristic for scheduling revisited. Proceedings of the Fourteenth National Conference on

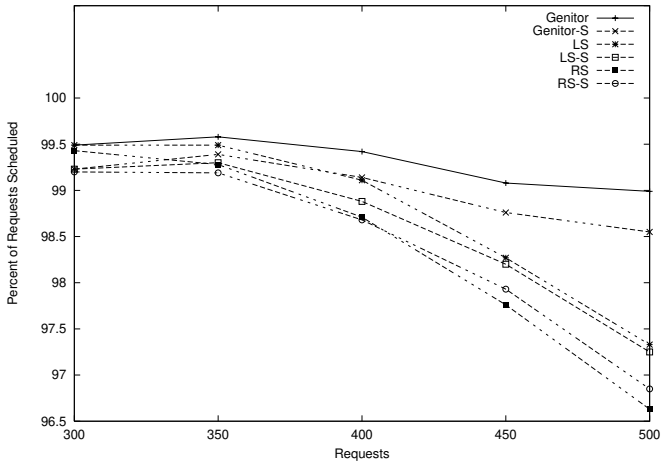

Fig. 3. Average percent of requests scheduled by the no-split and split versions of each of the algorithms.

Artificial Intelligence (AAAI-97), Providence, RI (1997) 241–248
2. Lawrence Davis: Handbook of Genetic Algorithms. Van Nostrand Reinhold, New York (1991)
3. David Goldberg: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, MA (1989)
4. T.D. Gooley: Automating the satellite range scheduling process. Master's thesis, Air Force Institute of Technology (1993)
5. D.A. Parish. A genetic algorithm approach to automating satellite range scheduling. Master's thesis, Air Force Institute of Technology (1994)
6. J.C. Pemberton: Toward scheduling over-constrained remote-sensing satellites. Proceedings of the Second NASA International Workshop on Planning and Scheduling for Space, San Francisco, CA (2000)
7. Steve Smith and C.C. Cheng: Slack-based heuristics for constraint satisfaction problems. Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93), Washington, DC (1993) 139–144
8. S.M. Schalck: Automating satellite range scheduling. Master's thesis, Air Force Institute of Technology (1993)
9. Gilbert Syswerda: Schedule Optimization Using Genetic Algorithms. In Lawrence Davis, editor, Handbook of Genetic Algorithms, chapter 21. Van Nostrand Reinhold, New York (1991)
10. E. Taillard: Some efficient heuristic methods for the flow shop sequencing problem. European Journal of Operations Research **47** (1990) 65–74
11. Gérard Verfaillie, Michel Lemaitre, and Thomas Schiex: Russian doll search for solving constraint optimization problems. Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), Portland, OR (1996) 181–187
12. L. Darrell Whitley: The *Genitor* Algorithm and Selective Pressure: Why Rank Based Allocation of Reproductive Trials is Best. J. D. Schaffer, editor, Proc. of the 3rd Int'l. Conf. on GAs, 116–121. Morgan Kaufmann (1989)
13. William J. Wolfe and Stephen E. Sorensen: Three scheduling algorithms applied to the earth observing systems domain. Management Science **46(1)** (2000) 148–166