AFSCN Scheduling: How the Problem and Solution Have Evolved*

Laura Barbulescu, Adele Howe and Darrell Whitley
Colorado State University
Fort Collins, CO 80523-1873 USA
{laura,howe,whitley}@cs.colostate.edu
tele: 01-970-491-7589, fax: 01-970-491-2466

Keywords: scheduling, genetic algorithm, objective function

Abstract

The Air Force Satellite Control Network (AFSCN) coordinates communications to more than 100 satellites via nine ground stations positioned around the globe. Customers request an antenna at a ground station for a specific time window along with possible alternative slots. Typically, 500 requests per day result in more than 100 conflicts, which are requests that cannot be satisfied because other tasks need the same slot. Scheduling access requests is referred to as the Satellite Range Scheduling Problem (SRSP).

This paper presents an overview of three key issues: 1) how has the problem changed over the last 10 years, 2) what algorithms work best and 3) what objective function is appropriate for AFSCN. We compared data sets from 1992 and from 2002/2003 and found significant differences in the problems. Our evaluation of solutions focus on three algorithms: local search, Gooley's algorithm from AFIT, and the *Genitor* genetic algorithm. It can be shown that local search (and therefore metaheuristics based on local search) fail to compete with Gooley's algorithm and Genitor. Finally, while all prior work on AFSCN minimizes request conflicts, we explore an alternative objective function. Because human schedulers must eventually schedule all requests, it might be better to optimize schedules for "repairability." Our results suggest that minimizing schedule overlaps makes it easier to fit larger requests into the schedule.

*This research was partially supported by a grant from the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grants number F49620-00-1-0144 and F49620-03-1-0233. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Darrell Whitley was also supported by the National Science Foundation under Grant No. 0117209. Adele Howe was also supported by the National Science Foundation under Grant No. IIS-0138690.

1 Introduction

Although particular scheduling problems have been studied for decades (e.g., job shop scheduling studies started in the early 1960s), rarely have the changes in their application been considered. The earliest studies of the AFSCN problem by researchers at the Air Force Institute of Technology [1, 2, 3] addressed data collected in 1992. Our group has been studying this problem since 2000 and have at different points (2002 and 2003) obtained additional actual data sets.

The Air Force Satellite Control Network (AFSCN) coordinates communications to more than 100 satellites via nine ground stations positioned around the globe. Customers request an antenna at a ground station for a specific time window along with possible alternative slots. Typically, the problem is oversubscribed, meaning that more requests vie for certain time/resource slots than can be accommodated. Currently, human schedulers with some computer assistance construct the daily schedule of accesses and are forced to arbitrate conflicts between approximately 20% of the daily task requests. Scheduling access requests is referred to as the Satellite Range Scheduling Problem (SRSP).

We have found that the problem is not only challenging, but it also has changed in the intervening decade between the original studies and now. Some of the changes are minor, e.g., differences in availability of resources. Some changes are more significant but obvious, e.g., an increase in number of tasks that must be scheduled each day. Others changes are harder to identify, e.g., the relative mix of different types of requests have changed, or at least, the different types of requests now interact in more complex ways. In short, the problem is qualitatively different today from that of ten years ago. Not surprisingly, scheduling the Air Force Satellite Control Network has become more complex and difficult.

In this paper, we describe the AFSCN application and some of the changes that we have identified in the application based on the data we have collected and based on conversations with human schedulers. We show how the performance of solutions varies depending on whether the new or old data are being solved and discuss how the earlier solutions were well suited to the older problems. Finally, based on our recent conversations with the human schedulers, we propose a revision to the original application. We propose a new objective function that allows the search to locate schedules that are more easily repaired. We also show how the change in objective function impacts the performance of the solutions.

2 AFSCN Scheduling: Problem Definition

AFSCN scheduling is the problem of coordinating communications between civilian and military organizations and more than 100 satellites. Space-ground communications are performed using 16 antennas located at nine ground stations around the globe. Customer organizations submit task requests to reserve an antenna at a ground station for a specified

time period based on the visibility windows between target satellites and ground stations. Two types of task requests can be distinguished: low altitude and high altitude. The low altitude tasks specify requests for access to low altitude satellites; such requests tend to be short (15 minutes) and have a tight visibility window. High altitude tasks specify requests for high altitude satellites; the durations for these requests are more varied and usually longer, and the visibility windows are large.

A problem instance consists of n task requests. A task request T_i, $1 \leq i \leq n$, specifies both a required processing duration T_i^{Dur} and a time window T_i^{Win} within which the duration must be allocated; we denote the lower and upper bounds of the time window by $T_i^{Win}(LB)$ and $T_i^{Win}(UB)$, respectively. Tasks cannot be preempted once processing is initiated. Each task request T_i specifies a resource (antenna) $R_i \in [1..m]$, where m is the total number of resources available. Unlike some other SRSPs, these tasks do not include priorities.

T_i may optionally specify $j \geq 0$ additional (R_i, T_i^{Win}) pairs, each identifying a particular alternative resource (antenna) and time window for the task. While requests are made for a specific antenna, often a different antenna at the same ground station may serve as an alternate because it has the same capabilities. Consequently, the problem may be viewed as multi-capacitated, although for our studies we have not found any significant advantage or justification for this perspective.

We begin with the assumption that we wish to minimize the number of request conflicts for AFSCN scheduling, or alternatively we wish to maximize the number of requests that can be scheduled without conflict. Those requests that cannot be scheduled without conflict are bumped out of the schedule. This is not what happens when humans carry out AFSCN scheduling. Satellites are valuable resources, and the AFSCN operators work to fit in every request; however, the scheduling conflicts are real and numerous. What this means in practice is that some requests are given less time than requested, or shifted to less desirable, but still usable time slots. In effect, the requests are altered until all requests are at least partially satisfied.

By using an evaluation function that minimizes the number of request conflicts, an assumption is being made that we should fit in as many requests as possible before requiring human schedulers to figure out how to place those requests that have been bumped. To the best of our knowledge[1], this is the only evaluation function that has been applied to AFSCN scheduling. In the first part of this paper, this evaluation function is also used. However, it may not be the most appropriate evaluation function, as will be discussed later in this paper.

Minimizing the number of request conflicts for a simplified version of AFSCN scheduling (have available only a single resource) is similar to a well-known problem in the machine scheduling literature, denoted $1|r_j|\sum U_j$ (in the three-field notation widely used by the scheduling community [4]). The first field specifies this is a one-machine scheduling problem.

[1] Commercial software has been developed to assist the human schedulers, but the exact algorithms are proprietary.

The second field indicates that for each request a release date and due date are specified. The third field defines the objective function. A job is on time if it is scheduled between its release and due dates; in this case, $U_j = 0$. Otherwise, the job is late, and $U_j = 1$. The objective is to minimize the number of late jobs, $\sum_{j=1}^{n} U_j$. Concurrency and preemption are not allowed. Based on AFSCN's equivalence to $1|r_j|\sum U_j$, we have shown that the decision version of the simplified one resource AFSCN Scheduling problem is $\mathcal{NP}$-complete [5].

3 Previous Research on SRSP

In satellite scheduling, customer requests for data collection need to be matched with satellite and ground station resources. The requests specify the instruments required, the window of time when the request needs to be executed and the location of the sensing/communication event. These task constraints need to be coordinated with resource constraints: the windows of visibility for the satellites, maintenance periods and downtimes for the ground stations, etc. Typically, more requests need to be scheduled than can be accommodated by the available resources. A general description of the satellite scheduling domain is provided by Jeremy Frank et al.(2001) [6].

Pemberton(2000) solves a simple one-resource satellite scheduling problem in which the requests have priorities, fixed start times and fixed durations. The objective function maximizes the sum of the priorities of the scheduled requests. A *priority segmentation algorithm* is proposed, which is a hybrid algorithm combining a greedy approach with branch-and-bound. Wolfe et al.(2000) [8] define a more complex one-resource problem, the *window-constrained packing problem* (WCP), which specifies for each request the earliest start time, latest final time and the minimum and maximum duration. The objective function is complex, combining request priority with the position of the scheduled request in its required window and the number of requests scheduled. Two greedy heuristic approaches and a genetic algorithm are implemented; the genetic algorithm is found to perform best.

Globus et al.(2003) compared a genetic algorithm, simulated annealing, Squeaky Wheel Optimization [10] and hill climbing on a simplified, synthetic form of the satellite scheduling problem (two satellites with a single instrument) and found that simulated annealing excelled and that the genetic algorithm performed relatively poorly. For a general version of satellite scheduling (EOS observation scheduling), Frank et al.(2001) proposed a constraint-based planner with a stochastic greedy search algorithm based on Bresina's Heuristic-Biased Stochastic Sampling (HBSS) algorithm. HBSS [11] had been originally applied to scheduling astronomy observations for telescopes.

Lemaître et al.(2000) research the problem of scheduling the set of photographs for Agile Earth Observing Satellites (EOS) [13]. Task constraints include the minimal time between two successive acquisitions, pairings of requests such that images are acquired twice in different time windows, and hard requirement that certain images must always be acquired. They found that a local search approach performs better than a hybrid algorithm combining

branch-and-bound with various domain-specific heuristics.

The AFSCN application was previously studied by researchers from the Air Force Institute of Technology (AFIT). Gooley [1] and Schalck [2] developed algorithms based on mixed-integer programming (MIP) and insertion heuristics, which achieved good overall performance: 91% – 95% of all requests scheduled. Parish [3] tackled the problem using a genetic algorithm called *Genitor* [14], which scheduled roughly 96% of all task requests, out-performing the MIP approaches. All three of these researchers used the AFIT benchmark suite consisting of seven problem instances, representing actual AFSCN task request data and visibilities for seven consecutive days from October 12 to 18, 1992[2]. Later, Jang [15] introduced a problem generator employing a bootstrap mechanism to produce additional test problems that are qualitatively similar to the AFIT benchmark problems. Jang then used this generator to analyze the maximum capacity of the AFSCN, as measured by the aggregate number of task requests that can be satisfied in a single-day. His results showed that approximately 90% of the requests can still be scheduled when there are 175 low altitude requests and 250 high altitude requests are present.

While the general problem of AFSCN Scheduling with minimal conflicts is $\mathcal{NP}$-complete, special subclasses of AFSCN Scheduling are polynomial. Burrowbridge [16] considers a simplified version of AFSCN scheduling, where each task specifies only one resource (antenna) and only low-altitude satellites are present. The objective is to maximize the number of scheduled tasks. Due to the orbital dynamics of low-altitude satellites, the task requests in this problem have negligible *slack*; i.e., the window size is equal to the request duration. Assuming that only one task can be scheduled per time window, the well-known *greedy activity-selector* algorithm [17] is used to schedule the requests since it yields a solution with the maximal number of scheduled tasks.

4 Algorithms for AFSCN

Based on previous research on AFSCN, SRSP and other oversubscribed scheduling applications, we implemented a variety of algorithms for AFSCN scheduling: local search, heuristic constructive search, iterative repair and genetic algorithms. We considered local search algorithms, using shifting and other problem-independent move operators. Unfortunately, the size of the neighborhood with these operators is of the order of n^2, where n is the total number of request tasks. We found that because of the size of the neighborhood, both a simple hill-climber and straightforward implementations of Tabu Search [18] perform poorly, even when combined with next-descent type search.

We also considered constructive search algorithms based on texture [19] and slack [20] constraint-based scheduling heuristics. We implemented straightforward extensions of such

[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 us with the benchmark suite.

algorithms for our application. The results were again poor; the number of request tasks combined with the presence of multiple alternative resources for each task make the application of such methods impractical. Early on, we tested HBSS and Limited Discrepancy Search (LDS) [21] on AFSCN scheduling. Because we had difficulty designing an adequate heuristic, we could not produce performance competitive with the other algorithms. We are currently exploring alternative heuristics for the heuristic constructive search algorithms.

Iterative repair methods have been successfully used to solve various oversubscribed scheduling problems, e.g., Hubble Space Telescope observations [22] and space shuttle payloads [23, 24]. NASA's ASPEN (A Scheduling and Planning Environment) framework [25], which has been used to model and solve real-world space applications such as scheduling EOS, employs both constructive and repair-based methods [26, 27]. More recently, Kramer et al. [28] used repair-based methods to solve the airlift scheduling problem for the USAF Air Mobility Command. In each case, a key component to the implementation was a domain appropriate ordering heuristic to guide the repairs. For AFSCN scheduling, Gooley's algorithm [1] uses domain-specific knowledge to implement a repair-based approach. We implement an improvement to Gooley's algorithm that is guaranteed to yield results at least as good as those produced by the original version.

While not as popular as repair-based techniques, genetic algorithms (GAs) have also been used in solving oversubscribed scheduling problems. Syswerda [29] uses a genetic algorithm to solve the problem of scheduling access to flight simulator resources in a laboratory. Also, a genetic algorithm is used to solve an abstraction of NASA's EOS scheduling problem, denoted the Window Constrained Packing Problem (WCPP) [8]. Parish [3] reported better results using *Genitor* on the AFIT AFSCN problems than the ones produced by Gooley's original algorithm.

Given the prevalence of repair-based and constructive techniques in the oversubscribed scheduling literature and Parish's results for the old AFSCN scheduling data, in our study, we focus on answering three questions: 1) How does the algorithm performance change from the old days of data to the current days? 2) What are the reasons for the performance differences? and 3) What works best for AFSCN scheduling? We investigate these questions by comparing three algorithms: Gooley's algorithm as an iterative-repair approach, *Genitor* as a genetic algorithm, a simple hill-climbing algorithm, and a random sampling as a baseline.

All of the algorithms include a phase in which a partial representation of the schedule is converted into an explicit assignment of time slots onto resources. Local search and the genetic algorithm encode the solutions as permutations of the requests; the search is defined over the permutation space. A greedy, deterministic "schedule builder" translates the permutations into schedules. The evaluation of the permutation is the number of bumped tasks in the corresponding schedule. Gooley's algorithm also constructs an ordering of the high altitude requests which can also be seen as a permutation. Complex, domain specific heuristics are then used to convert the permutation into an actual schedule; we consider these to be the "schedule builder" in Gooley's algorithm.

The rest of this section is organized as follows. First, we introduce a greedy algorithm that optimally schedules the low altitude requests. We use this algorithm when implementing our version of Gooley's algorithm. Gooley's original algorithm uses mixed integer programming to schedule the low altitude requests; however, this is no longer needed given that our method is provably optimal. Second, we describe the modified Gooley's algorithm and its schedule builder. Third, we describe *Genitor* and its schedule builder. Finally, we describe the remaining two algorithms.

We include random sampling in our study as a baseline for the performance. While generally not considered a viable method for scheduling, in the current study, random sampling helps to illustrate how problem difficulty has changed over the last decade.

4.1 Optimal Scheduling of Low Altitude Requests

In [5], we proved that Range Scheduling for low-altitude satellites continues to have polynomial time complexity even if the tasks may be serviced by one of several resources. For general AFSCN scheduling, the resources R_i specified in the (R_i, T_i^{Win}) pairs are antennas at the same ground station, and the time windows (T_i^{Win}) corresponding to each antenna are identical. The problem of scheduling the low-altitude requests is equivalent to an interval scheduling problem with k identical machines (for more on interval scheduling, see Bar-Noy et al. [30], Spieksma [31], Arkin et al.[32]). It has been proven that for the interval scheduling problem, an extension of the greedy activity-selector algorithm is optimal; the proofs are based on the equivalence of the interval scheduling problem to the k-colorability of an interval graph [33]. Our proof is based on a modified version of the greedy activity-selector algorithm for multiple resource problems. The algorithm still schedules the requests in increasing order of their due date; however, it specifies that each request is scheduled on the resource for which the idle time before its start time is the minimum. We call this algorithm $Greedy_{IS}$ (where *IS* stands for Interval Scheduling). We use this result to improve Gooley's original AFSCN scheduling algorithm.

4.2 Gooley's Algorithm

Gooley [1] developed a two phase algorithm to solve the AFSCN scheduling problem. In the first phase, the low altitude requests are scheduled, mainly using MIP. Because there are a large number of low altitude requests, the requests are divided into two blocks. MIP procedures are first used to schedule the requests in the first block. Then MIP is used to schedule the requests in the second block, which are inserted in the schedule around the requests in the first block. Finally, an interchange procedure attempts to optimize the total number of low altitude requests scheduled. This is needed because the low altitude requests are scheduled in disjoint blocks. Once the low altitude requests are scheduled, their start time and assigned resources remain fixed.

In our implementation, we replaced this first phase with $Greedy_{IS}$. Our version accomplishes the same function as Gooley's first phase but does so with a guarantee that the optimal number of low altitude requests are scheduled. Thus, the result is guaranteed to be equal to or better than Gooley's original algorithm.

In the second phase, the high altitude requests are inserted in the schedule (without rescheduling any of the low altitude requests). First, an order of insertion for the high altitude requests is computed. The requests are sorted in decreasing order of the ratio of the duration of the request to the average length of its time windows; ties are broken based on the number of alternative resources specified (fewer alternatives scheduled first). This in effect creates a permutation over the high altitude requests representing an ordered priority list. The insertion of the high altitude requests in the schedule is based on various domain specific heuristics; these heuristics function as a schedule builder, which we denote by *GooleyS*.

We separated the tasks of creating an ordered permutation of the high altitude requests and the construction of the actual schedule using the schedule builder for two reasons. First, during this second phase, the low altitude requests are not touched. This phase basically solves a separate problem: schedule the high altitude requests given that some blocks of time on the resources are marked as unavailable (because these blocks of time are reserved for the scheduled low altitude requests). Second, by explicitly noting the use of a permutation, Gooley's algorithm is seen to be similar to other methods that also use permutation representations. However, in Gooley's algorithm, the permutation contains only the high altitude requests, and some repair operations are also performed once all requests have been considered for insertion.

4.2.1 Gooley's Schedule Builder

GooleyS is represented by the set of heuristics used to incorporate the high altitude requests in the schedule containing the low altitude requests. For each high altitude request, its alternative resources are ranked based on the available free time; the resource with the most free time is considered first. Various criteria are then used to determine the start time for the request. These criteria refer to the free time available between the end of the request to be scheduled and the next request scheduled on that resource. For example, preference is given to slots for which less than five minutes or more than 60 minutes are available between the end of the request scheduled and the next one.

After all the high altitude requests have been considered for insertion, an interchange procedure attempts to accommodate the unscheduled requests by rescheduling some of the high altitude requests. A flexibility measure is used to determine which requests should be rescheduled.

Gooley defines the flexibility measure for a task request T_i as:

$$\frac{\sum_{j \in AltSet} FreeBlocksAvg_j}{|AltSet|} * \frac{VisAvg}{T_i^{Dur}}$$

where:

$$VisAvg = \frac{\sum_{j \in AltSet}(T_i^{Win}(UB) - T_i^{Win}(LB))}{|AltSet|}$$

$FreeBlocksAvg_j$ represents the average of the lengths of the three largest free time blocks in a time window for the task, and $|AltSet|$ is the number of possible resources specified for the task request.

The flexibility measure is directly proportional to the average length of the three largest blocks of free time in the alternative time windows, without correlating these to the duration of the request. This results in high flexibility for requests with long blocks of free time in their alternative time windows. However, since the length of such blocks is not checked against request duration, the corresponding requests might be harder to reschedule than shorter requests with shorter free blocks of time on their alternative resources (and smaller flexibility measure as defined).

4.3 The *Genitor* Genetic Algorithm

Previous studies of AFSCN by AFIT researchers indicate that *Genitor* provides superior overall performance [3]. The version of the *Genitor* used here was originally developed for a manufacturing scheduling application [34] [35], but it has also been applied to problems such as job shop scheduling [36].

Genetic algorithms have been successfully used to solve various scheduling problems, including problems with similar characteristics to SRSP. As mentioned earlier, genetic algorithms were found to perform well for WCPP (an abstraction of EOS). While WCPP differs in many ways from the AFSCN scheduling problem, both problems are oversubscribed, both model a unit capacity resource, and the requests in both problems define time windows based on satellite visibility. EOS scheduling has a number of competing objectives, including satisfaction of the largest possible number of task requests and consideration of the quality of the allocated time-slots. When the performance of two simple constructive algorithms and a genetic algorithm was compared on randomly generated instances of the WCPP, the genetic algorithm out-performed the constructive algorithms, but at the expense of larger run-times [8].

The solutions are encoded as permutations of the task request IDs. Like all genetic algorithms, *Genitor* maintains a population of solutions. In each step of the algorithm, a pair

of parent solutions is selected, and a crossover operator is used to generate a single child solution, which then replaces the worst solution in the population. The result is a form of elitism, in which the best individual produced during the search is always maintained in the population. Selection of parent solutions is 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 [37]. We use a population size of 200, a selection bias of 1.5, no mutation.

Typically, genetic algorithms encode solutions using bit-strings, which enable the use of "standard" crossover operators such as one-point and two-point crossover [38]. Because solutions in *Genitor* are encoded as permutations, a special crossover operator is required to ensure that the recombination of two parent permutations results in a child that (1) inherits good characteristics of both parents and (2) is still a permutation of the n task request IDs. Numerous crossover operators have been proposed for permutations representing scheduling problems.

Syswerda's [29] order crossover and position crossover are different from other permutation crossover operators such as Goldberg's PMX operator [39] or Davis' order crossover [40] in that there is no contiguous block which is directly passed to the offspring. Instead, several elements are randomly selected by absolute position. These operators are largely used for scheduling applications (e.g., [29, 41, 42] for Syswerda's operator) and are distinct from the permutation recombination operators that have been developed for the Traveling Salesman Problem.

Syswerda's order crossover operator starts by selecting K random positions in Parent 2. The corresponding elements from Parent 2 are then located in Parent 1 and reordered so that they appear in the same relative order as they appear in Parent 2. Elements in Parent 1 that do not correspond to selected elements in Parent 2 are passed directly to the offspring.

```
         Parent 1:  A B C D E F G
         Parent 2:  C F E B A D G
Selected Elements:    *   * *
```

The selected elements in Parent 2 are F B and A. Thus, the relevant elements are reordered in Parent 1.

```
   A B _ _ _ F _   =>   F B _ _ _ A _
```

All other elements are copied directly from Parent 1.

```
 (F B _ _ _ A _)  &  (_ _ C D E _ G)  =>  F B C D E A G
```

Syswerda's order crossover operator was applied to AFSCN scheduling by Parish [3]. Whitley and Nam [43] prove that order crossover and position crossover (Syswerda's second operator) are identical in expectation when order crossover selects K position and position crossover selects L-K positions. In effect, order crossover inherits by order first, then fills the remaining slots by position. Position crossover inherits by position first, then fills the remaining slots by their relative order. Use of order or position crossover can still be found in the literature however. An extensive study of crossover operators for a warehouse/shipping scheduler for the Coors brewery [44] has shown that Syswerda's position crossover performs better than PMX or Davis' order crossover. We chose to use Syswerda's position crossover for our study of AFSCN scheduling.

4.3.1 Genitor's Schedule Builder: GenitorS

GenitorS considers task requests in the order that they appear in the permutations. Each task 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 evaluation of a schedule is then defined as the total number of requests that are scheduled (for maximization) or inversely, the number of requests bumped from the schedule (for minimization).

4.4 Local Search

As the local search algorithm, we implemented a hill-climber. Because it has been successfully applied to a number of well-known scheduling problems, we selected a domain-independent move operator, the *shift* operator. From a current solution π, a neighborhood is defined by considering all $(N-1)^2$ pairs (x, y) of task request ID positions in π, subject to the restriction that $y \neq x-1$. The neighbor $\pi^{'}$ 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 large neighborhood size, we use the shift operator in conjunction with next-descent hill-climbing. We choose the position x by random and check the neighbors obtained by shifting the job at position x into all the positions, starting with position 0. The first neighbor with either a lower or equal number of bumped tasks is accepted. Search is initiated from a random permutation and terminates when a pre-specified number of solution evaluations is exceeded.

The local search algorithm also operates on permutations. To evaluate a solution corresponding to a particular permutation, we use GenitorS.

4.5 A Baseline: Random Sampling

Random sampling produces schedules by generating a random permutation of the task request IDs and evaluating the resulting permutation using GenitorS. Randomly sampling a large number of permutations provides information about the distribution of solutions in the search space, as well as a baseline measure of problem difficulty for heuristic algorithms.

5 Changes in the Application: New versus Old Problems

As discussed in Section 3, algorithms for AFSCN have previously been evaluated using only the AFIT benchmark suite from the 1992 data. The number of requests (low altitude and high altitude requests) to be scheduled for each of the seven problems is approximately 300 (see table 1 for specifics). We obtained five days of more recent data[3] for the dates: 3/7/2002, 3/20/2002, 3/26/2003, 4/2/2003 and 5/2/2003.

The most significant change since 1992 is that the number of requests received during a typical day has increased substantially to approximately 500 each day. The resources have remained more or less constant: the recent data include three new antennas not referenced in the AFIT problems, and the human schedulers tell us that two antennas are no longer reliable.

The increase in the number of requests currently received for a day also causes an increase in the number and percentage of requests that are bumped. In Table 1, we present the results obtained by running the four algorithms considered in this study. The statistics for *Genitor*, hill climbing and random sampling were obtained over 30 runs, with 8000 evaluations per run. For the 1992 data, at most eight task requests (or 2.5% of the tasks) are bumped in the best schedules; for the more recent data, the number increases to 42 (or 8.7%). The variance in performance for the stochastic algorithms increases somewhat for the more recent data, but not as much as does the best and mean performance.

The changes in the data appear to go beyond just a difference in scale. In a previous study [5], we showed that a simple heuristic that schedules first the low altitude requests and then the high altitude requests finds best solutions quickly for all the 1992 problems. We called this *the split heuristic*. In Table 2, we show the results obtained by randomly sampling permutations such that all the low altitude requests are scheduled before the high altitude ones. Even with a very small number of permutations sampled during each experiment (100 permutations), the best known values are found for the 1992 problems. However, this is not the case for the recent days of data. Although the results using the split heuristic are much better than random sampling (as shown in Table 1), the split heuristic fails to find the best schedules for two of the recent problems, even when using *Genitor* with 8000 or more evaluations allocated per run (as shown in Table 3).

[3] We thank William Szary and Brian Bayless at Schriever Air Force Base for providing us with these data.

Day	Size	*Genitor*			Hill Climbing			Random Sampling			Gooley
		Min	Mean	S.D.	Min	Mean	S.D.	Min	Mean	S.D.	
10/12/92	322	8	8.6	0.49	15	18.16	2.54	21	22.7	0.87	11
10/13/92	302	4	4	0	6	10.96	2.04	11	13.83	1.08	7
10/14/92	311	3	3.03	0.18	11	15.4	2.73	16	17.76	0.77	5
10/15/92	318	2	2.06	0.25	12	17.43	2.76	16	20.20	1.29	4
10/16/92	305	4	4.1	0.3	12	16.16	1.78	15	17.86	1.16	5
10/17/92	299	6	6.03	0.18	15	18.16	2.05	19	20.73	0.94	7
10/18/92	297	6	6	0	10	14.1	2.53	16	16.96	0.66	6
03/07/02	483	42	43.7	0.98	68	75.3	4.9	73	78.16	1.53	45
03/20/02	457	29	29.3	0.46	49	56.06	3.83	52	57.6	1.67	36
03/26/03	426	17	17.63	0.49	34	38.63	3.74	38	41.1	1.15	20
04/02/03	431	28	28.03	0.18	41	48.5	3.59	48	50.8	0.96	29
05/02/03	419	12	12.03	0.18	15	17.56	1.3	25	27.63	0.96	13

Table 1: Performance of *Genitor*, hill climbing and random sampling in terms of the best and mean number of bumped requests (with standard deviation as S.D.). All statistics are taken over 30 independent runs, with 8000 evaluations per run. The results of running Gooley's algorithm are included in the last column.

Day	Best Known	Random Sampling-Split		
		Min	Mean	S.D.
10/12/92	8	8	8.2	0.41
10/13/92	4	4	4	0
10/14/92	3	3	3.3	0.46
10/15/92	2	2	2.43	0.51
10/16/92	4	4	4.66	0.48
10/17/92	6	6	6.5	0.51
10/18/92	6	6	6	0
03/07/02	42	49	50.96	0.72
03/20/02	29	33	34.66	0.84
03/26/03	17	20	20.93	0.74
04/02/03	28	31	32.06	0.78
05/02/03	12	13	14.1	0.6

Table 2: 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.

By examining the data in the recent problems, we identified situations similar to the one in Figure 1, for which scheduling low-altitude requests first results in suboptimal solutions.

Day	Best Known	Genitor-Split		
		Min	Mean	Stdev
03/07/02	42	42	42	0
03/20/02	29	30	30	0
03/26/03	17	18	18	0
04/02/03	28	28	28	0
05/02/03	12	12	12	0

Table 3: Results of running *Genitor* with the split heuristic over 30 experiments, with 8000 evaluations per experiment.

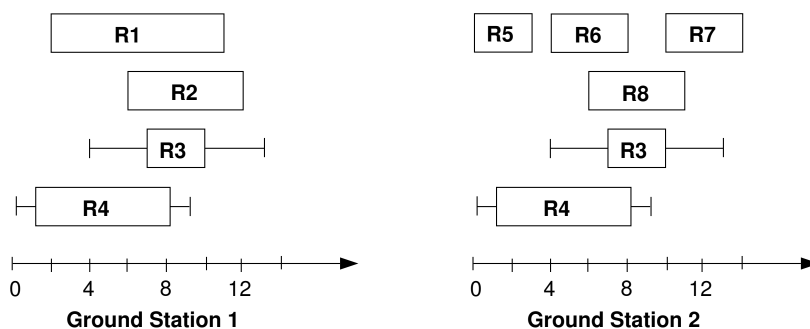

Figure 1: Example of a problem for which the split heuristic can not result in an optimal solution. Each ground station has two antennas; the only high-altitude requests are $R3$ and $R4$.

For the chosen example, there are two ground stations and two resources (two antennas) at each ground station. Two high-altitude requests, $R3$ and $R4$, have durations three and seven, 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.

If low-altitude requests are scheduled first, then $R1$ and $R2$ are scheduled on Ground Station 1 on the two resources, and the two high-altitude requests are bumped. Likewise, on Ground Station 2, the low-altitude requests are scheduled on the two resources, and the high-altitude requests are bumped. By scheduling low-altitude requests first, the two high-altitude requests are bumped. However, it is possible to schedule both of the high-altitude requests such that only one request ($R1$, $R2$ or $R8$) gets bumped. A possible solution schedules $R3$ and $R4$ on one resource at Ground Station 1 and bumps either $R1$ or $R2$; $R5$, $R6$, $R7$, and $R8$ can all be scheduled on Ground Station 2. For this example, a global optimum is not possible when all of the low-altitude requests are scheduled before the high-altitude requests.

We also designed a problem generator [5] that preserves characteristics found in the two problem data sets and also models features encountered in the real-world problems, such as types of requests (state of health, maneuver, payload download, payload commanding) and customer behavior. We used the problem generator to produce problems of sizes 300, 350, 400, 450, and 500. For each size, we generated 30 problem instances. We found that for these synthetic problems the split heuristic does not work well and its performance worsens as the problem size increases. This is consistent with our results for the real data. Also, the scale-up from 300 to 500 requests causes additional conflicts and competition for the resources, which translates into more interactions between the requests. It is not surprising therefore that the performance of the split heuristic (which is based on separating the low and high altitude requests) deteriorates as the problem size increases.

6 Algorithm Comparison: What Works Well on the Different Problems

In this section, we investigate two main questions: 1) What works well for AFSCN scheduling? and 2) What are the reasons for performance differences? Although we can provide a fairly clear answer to the first question, at this time, we only have a partial answer to the second.

To answer the first question, we present the results of running Genitor, hill-climbing and Gooley's algorithm in Table 1. Hill-climbing performs worse than both Genitor and Gooley's algorithm. We have recently investigated reasons for the poor performance of hill-climbing [45]. We found that the shifting operator is highly inefficient: given a current permutation, almost half of all possible moves result in no change in the corresponding schedule. In fact, by dramatically increasing the number of total evaluations allowed per run, from 8000 to 500,000 evaluations, hill-climbing finds best known value solutions for all the problems. The best performance for both the 1992 and the 2002/2003 data is obtained using Genitor. An increase in the number of evaluations to 800k and of the population size to 2000 did not improve the best solutions found for each problem.

Gooley's algorithm performs worse than Genitor for all the problems, except 10/18/92. The performance of Gooley's algorithm on the 1992 problems is somewhat disappointing. Gooley's algorithm optimally schedules the low altitude requests, then inserts the high altitude requests into the schedule. This is somewhat similar to the split heuristic. The split heuristic finds best known solutions for the seven old days of data relatively easily; also, we know that these problems have best known solutions for which the low altitude requests are optimally scheduled. Therefore, we expected Gooley's algorithm to perform better on these problems. On the other hand, the results in Table 1 show that its performance on the newer data is very similar to its performance on the data from 1992. Gooley's algorithm was designed based on the data from 1992. Given the differences between the 1992 data and the current data, this scale-up is surprising.

Genitor and local search are allocated the same number of evaluations (and therefore approximately the same CPU time); Gooley's algorithm is much faster. Running Genitor for 30 experiments, with 8000 evaluations per experiment, takes approximately 100 seconds CPU time for the 1992 problems and between 120 and 180 seconds for the 2002/2003 problems (on a Dell Precision 650, 3.06 GHz Xeon, running Linux). Gooley's algorithm only builds the schedule once and then it attempts to repair it by swapping tasks; it takes less than a second to run. We decided against implementing a possible extension of Gooley's algorithm that would run for a longer time (comparable to Genitor) for two reasons. First, we know that the heuristics used by Gooley to repair the schedule can be improved: Gooley's algorithm identifies the most flexible request to be rescheduled based on a flawed heuristic measure (see section 4). Second, as we show next, for the same permutation, the schedule builder in Genitor seems to result in a better schedule than the one in Gooley's algorithm for most of the problems (in fact, using GenitorS with Gooley's algorithm improves the solutions for six out of the seven 1992 problems).

6.1 Effect of the Schedule Builder on Performance

One factor that might contribute to the differences in performance is the schedule builder. The algorithms construct permutations (phase one) that are translated into schedules (phase two). We isolate the effect of each of these stages in Genitor and Gooley by comparing the effect on solution quality of 1) the permutation that gets translated into the solution (phase 1) and 2) the schedule builder (phase 2). To do this, we separate the two phases in each algorithm such that the best permutation can be used in either schedule builder. This produces four combinations of components: *Genitor*, *Gooley*, *Genitor+GooleyS* and *Gooley+GenitorS*. To control for the effect of low altitude requests, in all cases, the low altitude requests are scheduled optimally using $Greedy_{IS}$.

The results are presented in Table 4. To study the effect of the initial permutation, we hold constant the schedule builder and allow the permutation to vary which produces two comparisons: *Genitor* versus *Gooley+GenitorS* (both permutations are translated by GenitorS) and *Gooley* versus *Genitor+GooleyS* (both permutations use *GooleyS*). In both cases, the best results are usually obtained when the permutation is matched with its original schedule builder. Genitor's results dominate the results for Gooley+GenitorS. In all but two cases, Gooley dominates Genitor+GooleyS. This result is not surprising: Genitor's permutation is the result of evolving a population of permutations for which the fitnesses were computed using *GenitorS*.

To study the effect of the schedule builder on the quality of the solution obtained, we hold constant the permutation source which produces two comparisons: *Genitor* versus *Genitor+GooleyS* and *Gooley* versus *Gooley+GenitorS*. Applying *GooleyS* to the permutations from *Genitor* results in schedules that are worse than the ones obtained by *GenitorS* for the same permutations. Interestingly, pairing *Gooley*'s initial permutations with *GenitorS* improves over the solutions found in Gooley's original configuration for most of the problems.

Day	Size	Genitor	Gooley	Genitor+GooleyS	Gooley+GenitorS
10/12/92	322	8	11	9	9
10/13/92	302	4	7	6	5
10/14/92	311	3	5	8	4
10/15/92	318	2	4	6	3
10/16/92	305	4	5	8	4
10/17/92	299	6	7	9	6
10/18/92	297	6	6	6	7
03/07/02	483	42	45	49	48
03/20/02	457	29	36	36	35
03/26/03	426	17	20	22	20
04/02/03	431	28	29	30	31
05/02/03	419	12	13	16	12

Table 4: Comparison of the effects of the permutation generation and the schedule builder on the value of the solution for *Genitor* and Gooley's algorithm.

Thus, it appears that *GenitorS* is the better schedule builder.

The results suggest that Genitor's schedule builder may contribute to its relative success. This result is surprising: for six out of the seven problems from 1992 (for which Gooley's algorithm was designed), a simple greedy scheduler works better than the domain-specific, complex heuristics used to schedule the high altitude requests and then to repair the schedule.

To obtain the results just presented, the phases were decoupled so that the output of the first was available for the second. Consequently, the separation was not as clean as it could be: Genitor still used GenitorS for its objective function. An alternative is to excise GenitorS completely from Genitor for the comparison. GooleyS starts with a schedule containing the low altitude requests (optimally scheduled) in which it inserts the high altitude requests. In order to use GooleyS with Genitor, the population in Genitor would only contain permutations of the high altitude requests. However, using GooleyS as the scheduler most likely will not produce better results than Genitor using the split heuristic (since Gooley's algorithm always schedules the low altitude requests first). As for the split heuristic, this new version of Genitor would probably work well for the 1992 data; however, because of the existence of situations as the one depicted in Figure 1, it will perform worse than Genitor (with GenitorS) on the 2002/2003 data.

7 New Objective Function

Human schedulers of AFSCN use considerable, informal domain specific knowledge about their application. For example, they know that certain requests require access to a limited subset of ground stations and so the choices for scheduling such a request are more restricted. They know that the task durations sometimes can be reduced and this can be exploited in the assignment of time slots[4]. They also know that certain large requests can be split. Considering that our automated schedulers do not possess this additional knowledge, Genitor is doing quite well, managing to schedule 90-93% of the tasks on the days we have analyzed.

Nevertheless, minimizing the number of unscheduled task produces schedules that are unlike those that humans produce. For example, because we are minimizing the number of bumped requests, requests for large time slots are bumped more often than requests for smaller time slots. Often, the large tasks are related to satellite maintenance that can be put off for some period, but after some time, it becomes critical to fit them into a schedule.

Human schedulers also indicate that they often negotiate changes in *duration* of tasks to fit in additional tasks–and that everything must eventually be scheduled, even if that means modifying the requests. The problem with minimizing the number of bumped tasks is that this evaluation function does not include any evaluation of how difficult it will be to modify the schedule and fit in those requests that have been bumped.

Based on discussions with human schedulers, it would appear that a better evaluation criterion is to minimize the sum of overlaps between conflicting tasks in a proposed schedule. In this case, all requests are scheduled before evaluation, and the evaluation measures the degree of conflict, or overlap after all requests are scheduled. If the overlaps are small, it may be possible to simply trim time from the requested tasks that overlap without actually moving and rescheduling requests.

This is not the case when requests are bumped. There is no indication as to where the bumped requests might be reinserted into the schedule, and in fact, reinserting larger requests into the schedule may mean moving many other scheduled tasks that conflict with the bumped request. Consequently, a schedule that has been optimized to minimize the number of bumped requests may in fact look very different than the schedule a human eventually produces because many requests have to be rescheduled in order to fit in the bumped requests, which means that the schedule produced by the automated system offers little utility to the human scheduler. A schedule that minimizes conflicts also gives no indication as to what requests might be "trimmed" and fit into the schedule with only minor modifications, but nevertheless which will not fit without modifications.

[4]The real data we used to extract our problem instances are specified in a DEFT format. In this format, task durations are exactly specified; given the character of our application, most of these durations are strictly enforced.

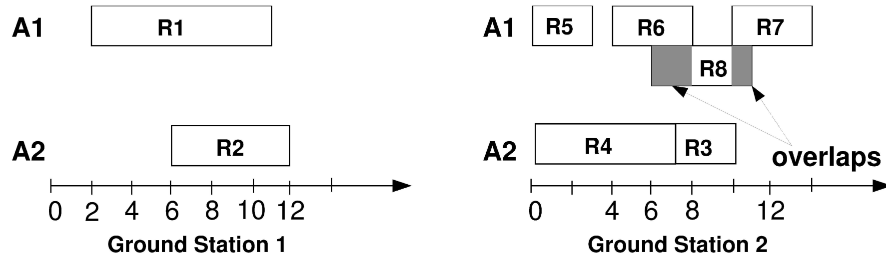

Figure 2: Optimizing the sum of overlaps.

A schedule that allocates time to all requests and then minimizes the total overlap provides human schedulers with a potential solution that is much closer to the conflict-free schedule they must eventually produce. In order to modify a request, human schedulers must negotiate with the individual who made the original request. If a request is reduced or allocated a less desirable time slot, the negotiated compromise must be honored after that. Having a solution that is closer to the final solution provides a much better starting point for negotiations.

Additionally, the new objective function provides a richer evaluation function for the algorithm. When minimizing the number of bumped tasks, if 500 jobs are being scheduled, then the number bumped is always an integer between 1 and 500. If most of the time, the number of bumps is between 1 and 100, then most of the time the evaluation function is an integer between 1 and 100. Thus, the evaluation function is a somewhat coarse metric. When using overlaps as an evaluation, the evaluation function is not just related to the number of jobs, but also to their durations. If the number of conflicts is between 1 and 100, but the overlaps range from 1 to 50 time units, then the evaluation function ranges over 1 to 5000 and provides more distinction between alternative solutions.

An example of how the sum of overlaps is computed is presented in Figure 2. Note that this is a solution for the example problem in Figure 1. R8 could either be scheduled on antenna A1 or antenna A2 at Ground Station 2. In order to minimize the overlaps, we schedule R8 on A1 (the sum of overlaps with R6 and R7 is smaller than the sum of overlaps with R3 and R4). While this is a trivial example, it illustrates the fact that instead of just reporting R8 as bumped, the new objective function results in a schedule which provides guidance about the fewest modifications needed to accommodate R8.

What is really being done is that the new evaluation function calculates conflicts in terms of individual time units instead of individual requests. This also means that two schedules many have exactly the same number of conflicts, but have very different evaluations in terms of the number of time units that are conflicted and overlap.

We designed a new schedule builder for Genitor to schedule all tasks (including conflicting tasks) and compute the sum of the overlaps. As with minimizing the number of bumps,

Day	Genitor						GenitorOverlaps					
	Bumps			Overlaps			Bumps			Overlaps		
	Min	Mean	S.D.	Min	Mean	S.D.	Min	Mean	S.D.	Min	Mean	S.D.
10/12/92	8	8.6	0.5	192	327.7	83.7	9	9.4	0.6	104	106.9	0.5
10/13/92	4	4	0	76	106.8	19.0	4	4.1	0.4	13	13	0
10/14/92	3	3.03	0.2	121	176.9	29.9	3	3.2	0.4	28	28.4	1.2
10/15/92	2	2.06	0.2	9	30	16.08	2	2.2	0.4	9	9.2	0.7
10/16/92	4	4.1	0.3	36	123.6	34.1	4	4.4	0.5	30	30.4	0.5
10/17/92	6	6.03	0.2	70	103.1	12.9	6	6.06	0.2	45	45.1	0.4
10/18/92	6	6	0	130	164.8	15.7	7	7.9	0.6	46	46.1	0.6
03/07/02	42	43.7	0.9	1441	1650.8	76.6	55	61.4	2.9	913	987.8	40.8
03/20/02	29	29.3	0.5	803	956.23	53.9	33	39.2	1.9	519	540.7	13.3
03/26/03	17	17.6	0.5	790	849.9	35.9	24	27.4	10.8	275	292.3	10.9
04/02/03	28	28.03	0.18	1069	1182.3	75.3	35	38.07	1.98	738	755.43	10.26
05/02/03	12	12.03	0.18	199	226.97	20.33	12	12.1	0.4	146	146.53	1.94

Table 5: The results obtained for Genitor and GenitorOverlaps by running 30 experiments with 8000 evaluations per experiment. Genitor optimizes the number of bumps. GenitorOverlaps optimizes the sum of overlaps.

in the order defined by the permutation, each task request is assigned to the first available resource from its list of alternatives and at its earliest possible starting time. If a request cannot be scheduled without conflict on any of the alternative resources, it overlaps; we assign such a request to the alternative resource on which the overlap with requests scheduled so far is minimized. We call this version *GenitorOverlaps*.

In Table 5, we present the results of running Genitor and GenitorOverlaps for 30 runs, with 8000 evaluations per run. Genitor optimizes the number of bumps; we compute for each best schedule obtained in a run, the corresponding sum of overlaps for the bumped requests. Statistics for both the number of bumps and the sum of overlaps over 30 runs are shown in columns 2-7. GenitorOverlaps optimizes the sum of overlaps; for each permutation evaluated as best at the end of a run, we apply the GenitorS schedule builder to compute the corresponding number of bumps. Again, we present statistics for the number of bumps and the sum of overlaps over 30 runs in columns 8-13.

The results show clearly that optimizing the number of bumps results on average in a larger corresponding sum of overlaps than when the overlaps are optimized, and the increase can be quite significant. On the other hand, optimizing the sum of overlaps results in a number of bumps which is usually larger than when the bumps are optimized; the increase is significant for the 2002/2003 data. These results also suggest that when minimizing the number of bumps, longer tasks are bumped, thus resulting in a large sum of overlaps.

8 Conclusions

Given how difficult it is to obtain actual data from real applications, it is compelling to assume that the application is well represented by what data have been obtained. Such an assumption is safe sometimes, but not always. In this paper, we present an overview of the AFSCN scheduling problem with a focus on how the application has changed over time and how the performance of the solutions have been affected.

First, we analyze the changes in the problem over the last ten years by comparing data sets from 1992 and 2002-2003. Some changes are obvious: the number of task requests significantly increased, while the resources remained more or less the same. From this, other changes emerge: the contention for the resources is higher, and therefore more requests cannot be accommodated in the schedule. We also found changes in the way the tasks interact. While a simple heuristic made the 1992 problems easy to solve, this is not the case for the new problems. The task requests in the current data interact in more complex ways, and the simple heuristic cannot always find the best solution.

Second, we study what algorithms work best for the scheduling application circa 1992 and 2002/2003. Previous research has shown that a genetic algorithm, Genitor, performs best for the 1992 problems. We found that Genitor also performs best for the 2002/2003 data, outperforming both local search and Gooley's algorithm, a domain-specific repair-based algorithm. Our results also show that the performance of Gooley's algorithm does scale-up when solving the later problems; this is somewhat surprising given that Gooley's algorithm was designed based on the 1992 data set. We also investigate possible reasons for the performance difference between Gooley's algorithm and Genitor by comparing the schedule builders in the two algorithms. We show that by replacing Gooley's schedule builder with the schedule builder from Genitor, the solutions obtained by Gooley's algorithm improve for six out of the seven problems from 1992.

Finally, we introduce a new objective function: minimizing the sum of the overlaps. The new objective function is motivated by the fact that human schedulers must eventually schedule all the requests by negotiating with the customers possible shorter durations or alternative time windows. This provides human schedulers with a potential solution that is much closer to the conflict-free schedule they must eventually produce.

The AFSCN scheduling application changed over a decade (from 1992 to 2002-2003), becoming more complex and difficult. Yet, surprisingly, the best solutions developed for the 1992 data have been shown to be robust to the changes. We are currently working on explaining why Genitor seems well suited to this application [45, 46], accommodating even the changes in complexity that have been observed over more than a decade.

References

[1] T.D. Gooley. Automating the Satellite Range Scheduling Process. In *Masters Thesis*. Air Force Institute of Technology, 1993.

[2] S.M. Schalck. Automating Satellite Range Scheduling. In *Masters Thesis*. Air Force Institute of Technology, 1993.

[3] D.A. Parish. A Genetic Algorithm Approach to Automating Satellite Range Scheduling. In *Masters Thesis*. Air Force Institute of Technology, 1994.

[4] R.L.Graham, E.L.Lawler, J.K.Lenstra, and A.H.G.Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. *Annals of Discrete Matehmatics*, 5:287–326, 1979.

[5] L. Barbulescu, J.P. Watson, L.D. Whitley, and A.E. Howe. Scheduling space-ground communications for the Air Force satellite control network. *Journal of Scheduling*, 2004. to appear.

[6] J. Frank, A. Jonsson, R. Morris, and D. Smith. Planning and scheduling for fleets of earth observing satellites. In *Proceedings of the Sixth International Symposium on Artificial Intelligence, Robotics, Automation and Space*, 2001.

[7] J.C. Pemberton. Toward Scheduling Over-Constrained Remote-Sensing Satellites. In *Proceedings of the Second NASA International Workshop on Planning an d Scheduling for Space*, San Francisco, CA, 2000.

[8] William J. Wolfe and Stephen E. Sorensen. Three Scheduling Algorithms Applied to the Earth Observing Systems Domain. In *Management Science*, volume 46(1), pages 148–168, 2000.

[9] A. Globus, J. Crawford, J. Lohn, and A. Pryor. Scheduling Earth observing satellites with evolutionary agorithms. In *International Conference on Space Mission Challenges for Information Technology*, Pasadena, CA, July 2003.

[10] David E. Joslin and David P. Clements. "Squeaky Wheel" Optimization. In *Journal of Artificial Intelligence Research*, volume 10, pages 353–373, 1999.

[11] J.L. Bresina. Heuristic-Biased Stochastic Sampling. In *Proceedings of the Thirteenth National Conference on Artificial I ntelligence*, pages 271–278, Portland, OR, 1996.

[12] M. Lemaître, G. Verfaillie, and F. Jouhaud. How to manage the new generation of Agile Earth Observation Satellites. In *6th International SpaceOps Symposium (Space Operations)*, Toulouse (France), 2000.

[13] French Society of Operations Research and Decision Analisys. ROADEF challenge 2003. http://www.prism.uvsq.fr/ vdc/ROADEF/CHALLENGES/2003/.

[14] Lawrence Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[15] Kwangho Jang. The Capacity of the Air Force Satellite Control Network. In *Masters Thesis*. Air Force Institute of Technology, 1996.

[16] Sarah Elizabeth Burrowbridge. Optimal Allocation of Satellite Network Resources. In *Masters Thesis*. Virginia Polytechnic Institute and State University, 1999.

[17] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT press, Cambridge, MA, 1990.

[18] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, MA, 1997.

[19] J. C. Beck, A. J. Davenport, E. M. Sitarski, and M. S. Fox. Texture-based Heuristic for Scheduling Revisited. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 241–248, Providence, RI, 1997. AAAI Press / MIT Press.

[20] S. Smith and C.C. Cheng. Slack-based Heuristics for Constraint Satisfaction Problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 139–144, Washington, DC, 1993. AAAI Press.

[21] William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *Proc. of the 14th Intl. Joint Conf. on A.I*, 1995.

[22] M.D. Johnston and G.E. Miller. Spike: Intelligent scheduling of Hubble space telescope observations. In Michael B. Morgan, editor, *Intelligent Scheduling*, pages 391–422. Morgan Kaufmann Publishers, 1994.

[23] M. Zweben, B. Daun, and M. Deale. Scheduling and rescheduling with iterative repair. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.

[24] G. Rabideau, S. Chien, J. Willis, and T. Mann. Using iterative repair to automate planning and scheduling of shuttle payload operations. In *Innovative Applications of Artificial Intelligence (IAAI 99)*, Orlando,FL, 1999.

[25] S. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. ASPEN - automating space mission operations using automated planning and scheduling. In *6th International SpaceOps Symposium (Space Operations)*, Toulouse (France), 2000.

[26] R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, and A. Fukunaga. Using ASPEN to automate EO-1 activity planning. In *Proceedings of the 1998 IEEE Aerospace Conference*, Aspen, CO, 1998.

[27] B. Engelhardt, S. Chien, A. Barrett, J. Willis, and C. Wilklow. The DATA-CHASER and Citizen Explorer benchmark problem sets. In *European Conference on Planning*, Toledo (Spain), 2001.

[28] L.A. Kramer and S.F. Smith. Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems. In *Proceedings of 18th International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, August 2003.

[29] Gilbert Syswerda. Schedule Optimization Using Genetic Algorithms. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*, chapter 21. Van Nostrand Reinhold, NY, 1991.

[30] Amotz Bar-Noy, Sudipto Guha, Joseph (Seffi) Naor, and Baruch Schieber. Approximating the Throughput of Multiple Machines in Real-Time Scheduling. *SIAM Journal on Computing*, 31(2):331–352, 2002.

[31] Frits C.R. Spieksma. On the Approximability of an Interval Scheduling Problem. *Journal of Scheduling*, 2:215–227, 1999.

[32] Esther M. Arkin and Ellen B. Silverberg. Scheduling Jobs with Fixed Start and End Times. *Discrete Applied Mathematics*, 18:1–8, 1987.

[33] Martin C. Carlisle and Errol L. Lloyd. On the k-coloring of Intervals. *Discrete Applied Mathematics*, 59:225–235, 1995.

[34] Darrell Whitley, Timothy Starkweather, and D'ann Fuquay. Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator. In J. D. Schaffer, editor, *Proc. of the 3rd Int'l. Conf. on GAs*. Morgan Kaufmann, 1989.

[35] L. Darrell Whitley, Timothy Starkweather, and Daniel Shaner. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*, chapter 22, pages 350–372. Van Nostrand Reinhold, 1991.

[36] M. Vazquez and D. Whitley. A Comparison of Genetic Algorithms for the Static Job Shop Scheduling Problem. In Schoenauer, Deb, Rudolph, Lutton, Merelo, and Schwefel, editors, *Parallel Problem Solving from Nature, 6*, pages 303–312. Springer, 2000.

[37] L. Darrell Whitley. The GENITOR Algorithm and Selective Pressure: Why Rank Based Allocation of Reproductive Trials is Best. In J. D. Schaffer, editor, *Proc. of the 3rd Int'l. Conf. on GAs*, pages 116–121. Morgan Kaufmann, 1989.

[38] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[39] David E. Goldberg and Robert Lingle. Alleles, Loci, and the Traveling Salesman Problem. In *Proceedings of the International Conference on Genetic Algorithms and Their Applications*, pages 8–15, Pittsburgh, PA, 1985.

[40] L. Davis. Applying adaptive algoirthms to epistatic domains. In *Proceedings of the ninth international joint conference on aritificial intelligence*, pages 162–164, Los Angeles, 1985.

[41] J. P. Watson, S. Rana, D. Whitley, and A. Howe. The Impact of Approximate Evaluation on the Performance of Search Algorithms for Warehouse Scheduling. *Journal of Scheduling*, 2(2):79–98, 1999.

[42] Gilbert Syswerda and Jeff Palmucci. The Application of Genetic Algorithms to Resource Scheduling. In L. Booker and R. Belew, editors, *Proc. of the 4th Int'l. Conf. on GAs*. Morgan Kaufmann, 1991.

[43] Darrell Whitley and Nam-Wook Yoo. Modeling Permutation Encodings in Simple Genetic Algorithm. In D. Whitley and M. Vose, editors, *Foundations of Genetic Algorithms-3*. Morgan Kaufmann, 1995.

[44] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A Comparison of Genetic Sequencing Operators. In L. Booker and R. Belew, editors, *Proc. of the 4th Int'l. Conf. on GAs*, pages 69–76. Morgan Kaufmann, 1991.

[45] L. Barbulescu, A.E. Howe, L.D. Whitley, and M. Roberts. Trading places: How to schedule more in a multi-resource oversubscribed scheduling problem. In *Proceedings of the International Conference on Planning and Scheduling*, 2004. to appear.

[46] L. Barbulescu, L.D. Whitley, and A.E. Howe. Leap before you look: An effective strategy in an oversubscribed scheduling problem. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, 2004. to appear.