Scheduling Space-Ground Communications for the Air Force Satellite Control Network*

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

Abstract

We present the first coupled formal and empirical analysis of the Satellite Range Scheduling application. We structure our study as a progression; we start by studying a simplified version of the problem in which only one resource is present. We show that the simplified version of the problem is equivalent to a well-known machine scheduling problem and use this result to prove that Satellite Range Scheduling is NP-complete. We also show that for the one-resource version of the problem, algorithms from the machine scheduling domain outperform a genetic algorithm previously identified as one of the best algorithms for Satellite Range Scheduling. Next, we investigate if these performance results generalize for the problem with multiple resources. We exploit two sources of data: actual request data from the U.S. Air Force Satellite Control Network (AFSCN) circa 1992 and data created by our problem generator, which is designed to produce problems similar to the ones currently solved by AFSCN. Three main results emerge from our empirical study of algorithm performance for multiple-resource problems. First, the performance results obtained for the single-resource version of the problem do not generalize: the algorithms from the machine scheduling domain perform poorly for the multiple-resource problems. Second, a simple heuristic is shown to perform well on the old problems from 1992; however it fails to scale to larger, more complex generated problems. Finally, a genetic algorithm is found to yield the best overall performance on the larger, more difficult problems produced by our generator.

*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. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We also wish to thank the anonymous reviewers for their thoroughness and insight.

1 Introduction

The U.S. Air Force Satellite Control Network (AFSCN) is responsible for coordinating communications between numerous civilian and military organizations and more than 100 satellites. Space-ground communications are performed via 16 antennas located at nine ground stations positioned around the globe. Customer organizations submit task requests to reserve an antenna at a ground station for a specific time period based on the windows of visibility between target satellites and available ground stations and the needs of the task. Alternate time windows and ground stations may also be specified. Over 500 task requests are received by the AFSCN scheduling center for a typical day. The communication antennas are over-subscribed, in that many more task requests are received than can be accommodated.

Currently, human scheduling experts construct an initial schedule that typically leaves about 120 conflicts representing task requests that are unsatisfiable (because of resource unavailability). However, satellites are extremely expensive resources, and the AFSCN is expected to maximize satellite utilization; out-right rejection of task requests is not a viable option. To resolve a conflict, human schedulers need to contact the organizations which generated the conflicting task requests; these organizations are given an explanation of the reason for the conflict and are presented with possible alternatives. Only in a worst case scenario is a task request canceled. This is a complex, time-consuming arbitration process between various organizations to ensure that all conflicts present in the initial schedule are resolved and that customers can be notified 24 hours in advance of their slot scheduling. The generic problem of scheduling task requests for communication antennas is referred to as the Satellite Range Scheduling Problem [1].

In reality, the processes and criteria that human schedulers use to develop conflict-free schedules are generally implicit, sometimes sensitive (due to rank, mission and security classification of the customers), and difficult to quantify, making automation extremely difficult. Instead, we focus on a single, although crucial, aspect of the problem: minimizing the number of conflicts in the initial schedule. Human schedulers do not consider any conflict or group of conflicts worse than any other conflict [2, 1, 3]. Therefore, in general, the human schedulers themselves state that minimizing the number of conflicts up-front reduces

(1) their work-load, (2) communication with outside agencies, and (3) the time required to produce a conflict-free schedule. We consider the objective of conflict-minimization for Satellite Range Scheduling because it is a core capability, the process lends itself to automation, and we can show the relationship to existing research.

This paper presents the first coupled formal and empirical analysis of the Satellite Range Scheduling application. Although it is an important application for both the government (e.g., military and NASA) and civilian applications (access to commercial satellites), the majority of work has been in proposing new heuristic methods without much examination of their strengths and limitations. While the applications were relatively new, this was appropriate, but at this stage, we need to understand the attributes of current problems and solutions to propose future directions for this dynamic application.

To put the application in context, we first relate Satellite Range Scheduling to other well-known and studied scheduling problems. To do this, we distinguish a progression of two forms of Satellite Range Scheduling (or, for brevity, "Range Scheduling"): Single-Resource Range Scheduling (SiRRS) and Multi-Resource Range Scheduling (MuRRS). In Single-Resource Range Scheduling, there is only one resource (i.e., antenna) that is being scheduled; Multi-Resource Range Scheduling includes tasks that can potentially be scheduled on one of several alternative resources. A special case of MuRRS occurs when the various tasks to be scheduled can be decomposed into multiple, but independent single resource range scheduling problems where there are no interactions between resources. Generally however, we assume that alternatives are actually on different resources.

The problem of minimizing the number of unsatisfied task requests on a single antenna (SiRRS) is equivalent to minimizing the number of late jobs on a single machine. The decision version of this problem (is there a schedule with X late jobs, where X is the known minimum number of late jobs) is known to be $\mathcal{NP}$-complete. We use this result to formally establish that Range Scheduling is also $\mathcal{NP}$-complete. Nevertheless, finding optimal or near optimal solutions to many realistic single resource problem instances is relatively easy. For some reasonable size problems, *branch and bound* methods can be used to either find or confirm optimal solutions.

To the best of our knowledge, the connection between SiRRS and scheduling a single machine has not previously been explored. We studied both exact methods (e.g., branch and bound methods) and heuristic methods taken from the single machine scheduling literature to address the problem of SiRRS. These methods are also compared to scheduling algorithms found in the satellite scheduling literature. The Range Scheduling specific methods were largely designed for MuRRS, and in fact, we found that the single machine scheduling heuristics generally out-perform heuristics which proved effective for Range Scheduling when compared on SiRRS problems.

The remainder of the paper discusses the MuRRS problem. Our results on SiRRS problems raise several questions about what methods are most appropriate for MuRRS. Do the results from SiRRS generalize to MuRRS? How good are the previously developed heuristic solutions to the MuRRS problem?

To answer these questions, we studied the performance of a number of algorithms on a suite of problems. As it happens, MuRRS is not a single well defined problem, but rather an on-going application in a dynamic environment. Previous researchers at the Air Force Institute of Technology (AFIT) tested their algorithms on actual data from 1992; we refer to these data as the "AFIT benchmark". An important issue is whether the results from those data hold for current application conditions. For example, in 1992, approximately 300 requests needed to be scheduled for a single day, compared to 500 requests per day in recent years. More requests for the same (actually somewhat fewer) resources have a clear impact on problem difficulty. Thus, we studied the algorithms with both AFIT (from 1992) and simulated current data and investigated whether the problems in the AFIT benchmarks are representative of the kinds of Range Scheduling problems that are currently encountered by AFSCN.

We can show that the AFIT benchmarks are simple in the sense that a heuristic can be used to quickly find the best known solutions to these problems. The heuristic splits the tasks to be scheduled into "low-altitude satellite" requests and "high-altitude satellite" requests. Because low-altitude satellite requests are highly constrained, these requests are scheduled first. A theorem and proof are presented showing that when a set of low-altitude requests is restricted to specific time slots, a greedy scheduling method exists that is optimal. The proof allows tasks to be scheduled on multiple alternative resources as long as the resources have identical capabilities.

However, we can also show that MuRRS problems exist where simple heuristics do not yield optimal results. In order to better study larger satellite range scheduling problems, we developed a system for generating realistic problem instances. With larger problem instances from the new problem generator, we find that the heuristic of splitting tasks into low and high altitude requests is no longer a good strategy. Additionally, in contrast to our results on the SiRRS problem, our results on the new problems are consistent with the findings of Parish [3]: the *Genitor* algorithm outperforms other heuristics on the new problem instances. Thus, it appears that although the algorithms previously tested for the Satellite Range Scheduling problem do not excel on the SiRRS problem, these algorithms do generalize to more modern versions on the MuRRS problem.

2 Satellite Range Scheduling: Definitions and Variants

For purposes of analysis, we consider two abstractions of the Satellite Range Scheduling problem. In both cases, a problem instance consists of n task requests where the objective is to minimize the number of unscheduled tasks. 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. We assume that time windows do not overlap day boundaries; consequently, all times are constrained to the interval $[1, 1440]$, where 1440 is the number of minutes in a 24-hour period. Tasks cannot be preempted once processing is initiated, and concurrency is not allowed, i.e., the resources are unit-capacity. In the first abstraction, there are n task requests to be scheduled on a *single* resource (i.e., communications antenna). Clearly, if alternative resources for task requests are not specified, there are no resource interactions; thus, the tasks can be scheduled on each resource independently. We refer to this abstraction as Single-Resource Range Scheduling (SiRRS), which we investigate in Section 4. In the second abstraction, each task request T_i additionally specifies a resource $R_i \in [1..m]$, where m is the total number of resources available. Further, T_i may optionally specify $j \geq 0$ additional (R_i, T_i^{Win}) pairs, each identifying a particular alternative resource and time window for the task; we refer to this formulation as Multi-Resource Range Scheduling (MuRRS).

A number of researchers from the Air Force Institute of Technology (AFIT) have developed algorithms for MuRRS. Gooley [2] and Schalck [1] 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*, which scheduled roughly 96% of all task requests, outperforming 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[1]. Later, Jang [4] 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.

Although not previously studied, the SiRRS is equivalent 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 [5], where:

> **1** denotes that tasks (jobs) are to be processed on a single machine,
>
> r_j indicates that there a release time is associated with job j, and
>
> $\sum U_j$ denotes the number of tasks not scheduled by their due date.

Using a more detailed notation, each task T_j, $1 \leq j \leq n$ has (1) a release date T_j^{Rel}, (2) a due date T_j^{Due}, and (3) a processing duration T_j^{Dur}. A job is on time if it is scheduled between its release and due date; 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.

[1]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 problems.

$1|r_j|\sum U_j$ scheduling problems are often formulated as decision problems, where the objective is to determine whether a solution exists with L or fewer tasks (i.e., conflicts) completing after their due dates, without violating any other task or problem constraints. The decision version of the $1|r_j|\sum U_j$ scheduling problem is $\mathcal{NP}$-complete [6] [7]. This result can formally be used to show that Satellite Range Scheduling is also $\mathcal{NP}$-complete. Other authors ([2, 3, 8]) have suggested this is true, but have not offered a formal proof.

Theorem 1 *The decision version of Satellite Range Scheduling is $\mathcal{NP}$-complete.*

Proof: $\mathcal{NP}$-Hardness is first established. We assume the total amount of time to be scheduled and the number of tasks to be scheduled are unbounded. (Limiting either would produce a large, but enumerable search space.) The $1|r_j|\sum U_j$ scheduling problem is $\mathcal{NP}$-complete and can be reduced to a SiRRS problem with unit-capacity as follows. Both have a duration T_j^{Dur}. The release date for the $1|r_j|\sum U_j$ problem is equivalent to the lower bound of the scheduling window: $T_j^{Rel} = T_j^{Win}(LB)$. The due date is equivalent to the upper bound of the scheduling window: $T_j^{Due} = T_j^{Win}(UB)$. Both problems count the number of unscheduled tasks, $\sum U_j$. This completes the reduction.

Because the set of SiRRS with unit-capacity is a subset of Satellite Range Scheduling problems, the general Satellite Range Scheduling problem is $\mathcal{NP}$-Hard.

To show Satellite Range Scheduling is $\mathcal{NP}$-complete, we must also establish it is in the class $\mathcal{NP}$. Assume there are n requests to be scheduled on m resources. Let $S*$ be an optimal solution and denote a resource by r. For every Satellite Range Scheduling instance, there exists a permutation such that all the scheduled tasks in $S*$ that are assigned to resource r appear before tasks assigned to resource $r+1$. All tasks that appear earlier in time on resource r appear before tasks scheduled later in time on r. All scheduled tasks appear before unscheduled tasks. Unscheduled tasks are sorted (based on identifier, for example) to create a unique sub-permutation. Thus, every schedule corresponds to a unique permutation. The decision problem is whether there exists a schedule $S*$ that is able to schedule a specific number of tasks. A nondeterministic Turing machine can search to find the permutation representing $S*$ in $O(n)$ time, and the solution can be verified in time proportional to n. Thus, Satellite Range Scheduling is in the class NP.

Since Satellite Range Scheduling is $\mathcal{NP}$-Hard and in the class $\mathcal{NP}$, Satellite Range Scheduling is $\mathcal{NP}$-complete. $\square$

2.1 Variants of Range Scheduling with Differing Complexity

While the general problem of Satellite Range Scheduling is $\mathcal{NP}$-complete, special subclasses of Range Scheduling are polynomial. Burrowbridge [9] considers a simplified version of the SiRRS problem where only low-altitude satellites are present and 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. The well-known *greedy activity-selector* algorithm [10] is used to schedule the requests since it yields a solution with the maximal number of scheduled tasks.

We next prove that Range Scheduling for low-altitude satellites continues to have polynomial time complexity even if jobs may be serviced by one of several resources. In particular, this occurs in the case of scheduling low-altitude satellite requests on one of the k antennas present at a particular ground station. For our proof, the k antennas must represent equivalent resources. We will view the problem as one of scheduling multiple, but identical resources.

We modify 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. Minimizing this idle time is critical to proving the optimality of the greedy solution. We call this algorithm $Greedy_{IS}$ (where *IS* stands for Interval Scheduling) and show that it is optimal for scheduling the low-altitude requests. 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. [11], Spieksma [12], Arkin et al.[13]). It has been proven that for the interval scheduling problem the 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 [14]. We present a

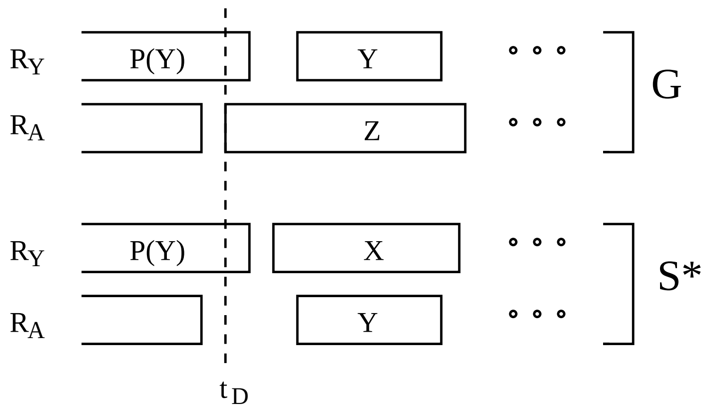

Figure 1: The optimal and greedy schedules are identical before time t_D. Note that Y was the next task scheduled in the greedy schedule G. In this case, Y appears on some alternative resource, R_A, in the optimal schedule $S*$, and X appears on resource R_Y. Note that the start time of request Z is irrelevant, since the transformation is performed on schedule $S*$.

new proof, similar to the one in [10] for the one-machine case.

Theorem 2 *The* Greedy$_{\text{IS}}$ *algorithm is optimal for scheduling the low-altitude satellite requests for Multiple-Resource Range Scheduling with identical resources and no slack.*

Proof: Let $S*$ be an optimal schedule. Let G be the greedy schedule. Assume $S*$ differs from G. We show that we can replace all the non-greedy choices in $S*$ with the greedy tasks scheduled in G; during the transformation, the schedule remains feasible and the number of scheduled requests does not change and therefore remains optimal. This transformation also proves that G is optimal.

The transformation from $S*$ to G deals with one request at a time. We examine $S*$ starting from time 0 and compare it with G. Let t_D be the first point in time where G differs from S*. Let Y be the next request picked to be scheduled in G after time t_D. This means that Y is the next job scheduled after time t_D in G with the earliest finish time. Let the resource where request Y is scheduled in G be R_Y. Let $P(Y)$ be the job preceding Y in the greedy schedule G. When constructing the greedy schedule G, Y is chosen to have the earliest due date *and* is placed on the resource that results in minimum idle time. If Y is not scheduled the same in $S*$ and G, then either Y does not appear at all in $S*$ (case 1), or it appears on an alternative resource R_A in $S*$ (case 2, see Figure 1).

Case 1: The greedy choice Y is not present in $S*$. Instead, some request X appears on R_Y. Then Y can replace X in the optimal schedule because the due date of Y is earlier than or at most equal to the due date of X (since $S*$ and G are identical up to the point in time t_D, X did not appear in G before time t_D, and we know that Y is the next greedy choice after time t_D). Also, there is no conflict in the start time, since in both schedules the requests X and Y follow $P(Y)$ in S and G, respectively. The number of scheduled tasks does not change.

Case 2: The greedy choice Y is present in $S*$ on resource R_A and some other request X follows $P(Y)$ on resource R_Y in $S*$. We can then swap *all of the subsequent requests* scheduled after the finish time of $P(Y)$ on resources R_A and R_Y in schedule $S*$. Again, there is no conflict in the start times, since in the two schedules the requests X and Y follow $P(Y)$ in S and G respectfully. This places Y on the same resource on which it appears in G. The number of scheduled tasks does not change.

The transformation continues until $S*$ is transformed into G. One of the two cases always applies. At no point does the number of scheduled tasks change. Therefore, G is also optimal. $\square$

Note that the proof does not mention what other tasks may be scheduled on G. This is because the proof only needs to convert $S*$ to G one request at a time. However, it is notable that some request Z could be encountered in schedule G at time t_D on resource R_A before either Y or X are encountered in $S*$, which is illustrated in Figure 1. This occurs because Z has a start time before Y, but has a finish time after Y and therefore is **not** the next request scheduled in G. However, the proof is only concerned with the next scheduled request Y. Figure 1 also illustrates that it is always possible to swap *all of the subsequent requests* scheduled after the finish time of $P(Y)$ in schedule $S*$ and that the start time of Z is irrelevant. Request Z is a later greedy choice in G; the transformation will deal with Z as it moves through the greedy schedule based on the finish times.

3 Algorithms for the Satellite Range Scheduling Problem

In this section, we document the various heuristic algorithms that we use in our analysis of both SiRRS and MuRRS. We begin by describing a branch-and-bound algorithm designed by Baptiste et al. (1998) for the $1|r_j|\sum U_j$ machine scheduling problem. Next, we briefly discuss two greedy heuristic algorithms designed specifically for the $1|r_j|\sum U_j$ problem. We introduce the methods for both solution encoding and evaluation in local search algorithms for both formulations of the Satellite Range Scheduling, and define the core algorithms used in our analyses: the *Genitor* genetic algorithm, a hill-climbing algorithm, and random sampling. Finally, we discuss our decision to omit some algorithms that use well-known scheduling technologies.

3.1 Branch-and-Bound Algorithm for Single-Resource Range Scheduling

Baptiste et al. [15] introduce a branch and bound algorithm to solve the $1|r_j|\sum U_j$ problem; this algorithm also applies to Single Resource Range Scheduling problems. The algorithm starts by computing a lower bound on the number of late tasks, v. Branch and bound is then applied to the decision problem of finding a schedule with v late tasks. If no such schedule can be found, v is incremented, and the process is repeated. When solving the decision problem, at each node in the search tree, the branching scheme selects an unscheduled task and attempts to schedule the task. The choice of the task to be scheduled is made based on a heuristic that prefers small tasks with large time windows over large tasks with tight time windows. Dominance properties and constraint propagation are applied; then the feasibility of the new one-machine schedule is checked. If the schedule is infeasible, the algorithm backtracks, and the task is considered late. Determining the feasibility of the one-machine schedule at each node in the search tree is also $\mathcal{NP}$-hard. However, Baptiste et al. note that for most of the cases a simple heuristic can decide feasibility; if not, a branch and bound algorithm is applied.

3.2 Constructive Heuristic Algorithms

Constructive heuristics begin with an empty schedule and iteratively add jobs to the schedule using local, myopic decision rules. These heuristics can generate solutions to even large problem instances in sub-second CPU time, but because they typically employ no backtracking, the resulting solutions are generally sub-optimal. Machine scheduling researchers have introduced two such greedy constructive heuristics for $1|r_j|\sum U_j$.

Dauzère-Pérès [16] introduced a greedy heuristic to compute upper bounds for a branch-and-bound algorithm for $1|r_j|\sum U_j$; we denote this heuristic by $Greedy_{DP}$. The principle underlying $Greedy_{DP}$ is to schedule the jobs with little remaining slack immediately, while simultaneously minimizing the length of the partial schedule at each step. Thus, at each step, the job with the earliest due date is scheduled; if the completion time of the job is after its due date, then either one of the previously scheduled jobs or this last job is dropped from the schedule. Also, at each step, one of the jobs dropped from the schedule can replace the last job scheduled if by doing so the length of the partial schedule is minimized. By compressing the partial schedule, more jobs in later stages of the schedule can be accommodated.

Bar-Noy et al.[11] introduced a greedy heuristic, which we denote by $Greedy_{BN}$, based on a slightly different principle: schedule the job that *finishes* earliest at each step, independent of the job due date. Consequently, $Greedy_{BN}$ can schedule small jobs with significant slack before larger jobs with very little slack. Both $Greedy_{DP}$ and $Greedy_{BN}$ are directly applicable only to SiRRS; simple extensions of these heuristics to MuRRS are discussed in Section 6.

3.3 Local Search Algorithms

In contrast to constructive heuristic algorithms, local search algorithms begin with one or more complete solutions; search proceeds via successive modification to a series of complete solution(s). All local search algorithms we consider encode solutions using a permutation π of the n task request IDs (i.e., $[1..n]$); a *schedule builder* is used to generate solutions from a permutation of request IDs. The schedule builder considers task requests in the order that they appear in π. For SiRRS, the schedule builder attempts to schedule the task request within the specified time window. For the MuRRS, 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 minimizing).

3.3.1 A Next-Descent Hill-Climbing Algorithm

Perhaps the simplest local search algorithm is a hill-climber, which starts from a randomly generated solution and iteratively moves toward the best neighboring solution. A key component of any hill-climbing algorithm is the move operator. We have selected a domain-independent move operator known as the *shift* operator. Local search algorithms based on the shift operator have been successfully applied to a number of well-known scheduling problems, for example the permutation flow-shop scheduling problem [17]. From a current solution π, a neighborhood under the shift operator 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 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 (as opposed to steepest-descent) hill-climbing. 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 is initiated from a random permutation and terminates when a pre-specified number of solution evaluations is exceeded.

In practice, neighborhood search algorithms fail to be competitive because the neighborhood size is so large. With 500 task requests, the number of neighbors under both shift and other well-known problem-independent move operators is approximately 500^2. The best algorithms for this problem produce solutions using fewer than 10,000 evaluations.

3.3.2 The *Genitor* Genetic Algorithm

Previous studies of Range Scheduling by AFIT researchers indicate that the *Genitor* genetic algorithm [18] provides superior overall performance [3].

Genetic algorithms have been successfully used to solve various scheduling problems, including problems with similar characteristics to Satellite Range Scheduling. For example, genetic algorithms were found to perform well for an abstraction of NASA's Earth Observing Satellite (EOS) scheduling problem, denoted the Window Constrained Packing Problem (WCPP) [8]. While WCPP is in many ways different from our SiRRS, both problems are oversubscribed (more requests need to be scheduled than can be accommodated with the available resources), 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 the quality of the allocated time-slots. The performance of two simple constructive algorithms and a genetic algorithm is studied on randomly generated instances of the WCPP; the genetic algorithm out-performed the constructive algorithms, but at the expense of larger run-times.

As in the hill-climbing algorithm, 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 [19].

Typically, genetic algorithms encode solutions using bit-strings, which enable the use of "standard" crossover operators such as one-point and two-point crossover [20]. 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. Following Parish [3], we use Syswerda's (relative) order crossover operator, which preserves the relative order of the elements in the parent solutions in the child solution. Syswerda's crossover operator has been successfully applied in a variety of scheduling applications [21] [22] [23].

The version of the *Genitor* Genetic Algorithm used here was originally developed for a warehouse scheduling application [24] [25], but it has also been applied to problems such as job shop scheduling [26].

3.4 A Baseline: Random Sampling

Random sampling produces schedules by generating a random permutation of the task request IDs and evaluating the resulting permutation using the scheduler builder introduced earlier in this section. 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.

3.5 Scheduling Algorithms Not Included

We also considered straightforward implementations of Tabu search [27] for MuRRS, but the resulting algorithms were not competitive with *Genitor*. Again, the size of the local search neighborhood using shift and other well-known problem-independent move operators is approximately 500^2. With such a large neighborhood, tabu search and other forms of neighborhood local search are simply not practical. We also combined tabu search with next-descent type search, but the performance was poor.

Additionally, we developed constructive search algorithms for SiRRS based on texture [28] and slack [29] constraint-based scheduling heuristics. We found that texture-based heuristics are effective when the total number of task requests is small (e.g., $n \leq 100$) for SiRRS.

Pemberton [30] applied an algorithm based on constraint programming combined with a greedy constructive heuristic to a remote-sensing problem similar to Wolfe and Sorenson's WCPP, with the exception that the start times of task requests are fixed. We considered similar algorithms for MuRRS, but were largely unsuccessful; straightforward extensions of constraint-based technologies for multiple resources with alternatives did not appear to be effective.

Heuristic-Biased Stochastic Sampling (HBSS) [31] has been used to schedule astronomy observations for telescopes [32] and has been also applied to EOS observation scheduling [33]. Early on, we studied HBSS and LDS [34] on this problem. Because we had difficulty designing an adequate heuristic, we could not produce performance competitive with the other algorithms. As future work, we will be exploring alternative heuristics with the heuristic constructive search algorithms.

We are also considering implementing "Squeaky Wheel" Optimization (SWO) [35] for Satellite Range Scheduling by following the methodology described for solving a scheduling problem in fiber-optic cable manufacturing [35]. A greedy algorithm constructs an initial permutation; a schedule builder is used to convert this permutation into a solution. The solution is analyzed in order to assign blame to the elements which cause "trouble" in the solution, and this information is used to modify the order in which the greedy algorithm builds the new solution. At present, our objective function is inadequate to support SWO for MuRRS; we will need to define an objective function that better quantifies the presence of conflicts in the schedule, such that given a solution we can assign blame of a certain magnitude to each of the conflicts.

4 Problem Difficulty and Single-Resource Range Scheduling

We begin our analysis by considering SiRRS. Our motivation is two-fold. Given the equivalence of the SiRRS with the $1|r_j|\sum U_j$ machine scheduling problem, our first goal is to analyze the performance of heuristic algorithms designed specifically for $1|r_j|\sum U_j$; if these algorithms are competitive, extensions may provide good performance on the more complex MuRRS problem. Secondly, SiRRS is much easier to analyze, giving us a starting point for understanding the more complex version of the problem.

In general, evaluation based on real-world benchmarks is clearly desirable. There are, however, two related but distinct drawbacks. First, obtaining real-world data is often difficult and/or costly. Second, by basing evaluation on a small set of real-world instances, we run the risk of over-fitting our algorithms to these instances. On the other hand, we also note that random problem instances can be much more difficult than real-world problems (e.g., see Taillard [36] or Watson et al.[37]) and real-world problems may display "structure" that is not found in random problems. We address these concerns by considering the performance of heuristic algorithms for the SiRRS on a range of generated problems that include realistic as well as more extreme characteristics.

4.1 The Problem Generator

We implemented a simple problem generator for the SiRRS problem. In section 6.1 we will also present a more complex problem generator which includes more realistic classes of service requests as well as realistic satellite pass times.

Our test problems for SiRRS are produced by a problem generator that we developed based on the characteristics of the AFSCN application. The AFSCN schedules task requests on a per-day basis; consequently, we restrict the lower and upper bounds of the task request time windows T_i^{Win} to the interval $[1, 1440]$, or the number of minutes in a 24-hour period. In the AFIT benchmark problems, the overwhelming majority (more than 90%) of task durations T_i^{Dur} fall in the interval $[20, 60]$. We denote the *slack* T_i^{Slack} associated with a request T_i by $T_i^{Win}(UB) - T_i^{Dur} - T_i^{Win}(LB)$; the slacks of task requests in the AFIT benchmark problems generally range from 0 to 100. Finally, no communications antennas in any AFIT benchmark problem is assigned more than 50 task requests, and typically far fewer.

Based on the above observations, we generate random instances of SiRRS problems using the following procedure for each of the n task requests, which takes as input the maximum slack value *MAXSLACK* allowed for any task request:

1. Sample the processing duration T_i^{Dur} uniformly from the interval $[20, 60]$.

2. Sample the slack T_i^{Slack} uniformly from the interval $[0, \mathit{MAXSLACK}]$.

3. Sample the window start time $T_i^{Win}(\mathit{LB})$ uniformly from the interval $[1, 1440 - T_i^{Slack} - T_i^{Dur}]$.

4. Let $T_i^{Win}(\mathit{UB}) = T_i^{Win}(\mathit{LB}) + T_i^{Slack} + T_i^{Dur}$.

Given particular values of n and *MAXSLACK*, a problem *set* consists of 100 randomly generated instances. We consider 36 problem sets in our analysis, one for each combination of $n \in [30, 40, 50, 60]$ and $\mathit{MAXSLACK} \in [0, 25, 50, 75, 100, 125, 150, 175, 200]$. In the context of the AFSCN scheduling problem, problem sets with small-to-moderate n and *MAXSLACK* correspond to realistic problem instances; instances with larger values of n and *MAXSLACK* are generally unrealistic, but are included to bracket performance.

4.2 Identifying Optimal Solutions: Algorithms and Expense

We can determine optimal solutions to SiRRS problems using a branch and bound algorithm. Baptiste and colleagues developed two branch-and-bound algorithms for the $1|r_j|\sum U_j$. The algorithm we use to compute optimal solutions to our test instances was introduced by Baptiste et al.[15][2]; a more recent extension to this algorithm is reported in [38].

We ran the branch and bound algorithm of Baptiste et al. on each instance in each of our problem sets, imposing a limit for each instance of 1 hour of CPU time on a 1 GHz Pentium III running Windows XP. For all instances with 30 task requests, the algorithm computed the optimal solution within 5 seconds; 40 requests required at most 1 minute of CPU time. Similarly, the majority of instances with 50 and 60 task requests could be solved in less than 10 minutes of CPU time. We report the number of exceptions in Table 1. Several of the larger problems required between 10 minutes and 1 hour of CPU for solution, while a small number of instances were never solved (the fourth column in Table 1 specifies the number of instances out of 100 which were never solved); the un-solved instances remain insoluble when the CPU limit is raised to 10 hours. Baptiste et al. indicate that their algorithm is able to

[2]We thank Dr. Philippe Baptiste for providing executable versions of their branch-and-bound algorithm.

compute optimal solutions for all instances with 60 or fewer task requests; we attribute this discrepancy with our results to the significant differences between our problem generator and the generator introduced by Baptiste et al. For their problem generator, Baptiste et al. use a mixture of normal and uniform distributions to select the values for processing times, release dates and slack; they also model the "load" of the machine, computed as a ratio between the total demand for processing time and the time available between the minimum release time and maximum deadline of all the jobs.

# of Tasks	*MAXSLACK*	# of instances with 10 minutes $< CPU <$ 1 hour	# of instances with $CPU >$ 1 hour
50	200	0	3
60	75	1	1
	100	3	3
	125	5	3
	150	1	5
	175	5	4
	200	7	2

Table 1: The number of problem instances (out of 100) for which the Baptiste branch and bound algorithm (1) required over 10 minutes of CPU time to compute the optimal number of bumps and (2) failed to find an optimal solution within 1 hour of CPU time.

4.3 The Performance of Heuristic Algorithms on SiRRS

We now analyze the performance of the four heuristic algorithms introduced in Sections 3.2 and 3.3 on SiRRS Problems. Algorithm performance is measured in terms of the total number of bumped tasks, which we denote $|Bumps|$. We can measure the *absolute* performance of a heuristic algorithm by computing the difference between the best solutions found by an algorithm and the optimal number of bumped tasks, denoted by $|Bumps_{opt}|$. In the few instances where Baptiste et al.'s algorithm failed (as in Table 1) we substitute the best solution found by *any* of our heuristic algorithms as the absolute reference point (random sampling never found a higher-quality solution than any of the heuristic algorithms)[3]. In

[3] In the worst case, for one set of 100 problems there were 5 instances for which Baptiste's algorithm failed to compute the optimum. For these 5 instances, in the worst case, the difference contributing to the mean was 0. If eliminated, $newMean = oldMean * (100/95) = oldMean * 1.05$, which only slightly changes the value on the graph

both cases, we denote the difference in the number of bumped tasks by Δ_{best}.

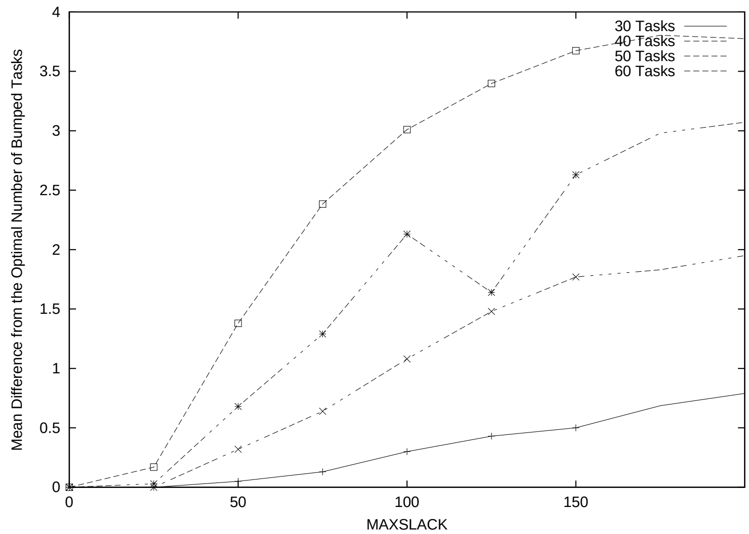

Figure 2: The mean difference from the optimal number of bumped tasks $\overline{\Delta_{best}}$ for random sampling.

We first consider the performance of our baseline algorithm, random sampling. We define a single trial as generating 8000 random permutations of the integers $[1..n]$ (where n is the total number of task requests); each permutation is evaluated using the scheduler builder introduced in Section 3.3. For each of our $1|r_j|\sum U_j$ instances, we execute 30 trials; we then take $|Bumps|$ as the minimal number of bumped tasks observed in any of the 30 trials. We report the mean Δ_{best} for random sampling in Figure 2. As expected (due to the growth in the size of the search space), the performance of random sampling degrades with increases in the problem size n. However, two qualitative aspects of Figure 2 were surprising. First, random sampling generates optimal or near-optimal solutions to test instances for which the *MAXSLACK* value is consistent with the slack of task requests in AFSCN problem instances (e.g., $\mathit{MAXSLACK} \leq 60$). Second, even for instances with unrealistically large *MAXSLACK*, random sampling generates solutions with on average less than 4 more bumped task requests than the optimal number in the *worst* case.

Next, we consider the performance of the two greedy constructive heuristic algorithms for $1|r_j|\sum U_j$. As noted in Section 3.2, both $Greedy_{BN}$ and $Greedy_{DP}$ are largely deterministic; randomness is only applied to tie-break when one or more equally good alternatives are

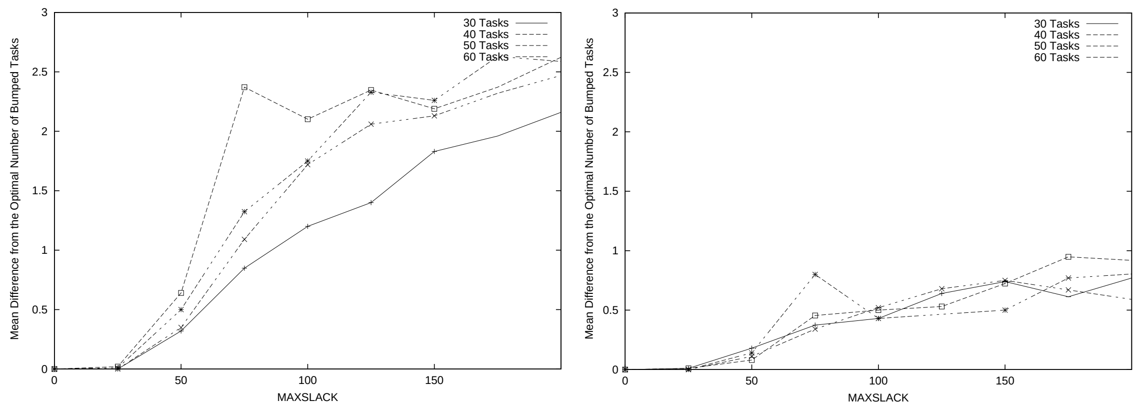

Figure 3: The mean difference from the optimal number of bumped tasks $\overline{\Delta_{best}}$ for the constructive heuristics $Greedy_{BN}$ (left figure) and $Greedy_{DP}$ (right figure).

available. Consequently, we take Δ_{best} for both algorithms as the result of a single trial on a given problem instance; we report the resulting mean Δ_{best} in Figure 3. First, we observe that $Greedy_{DP}$ consistently outperforms $Greedy_{BN}$ independently of n and *MAXSLACK*. Second, $Greedy_{DP}$ bumps less than one task request on average, obtaining optimal or near-optimal performance for all of our test instances, and with a trivial amount of computational effort (i.e., less than 1 second of CPU time).

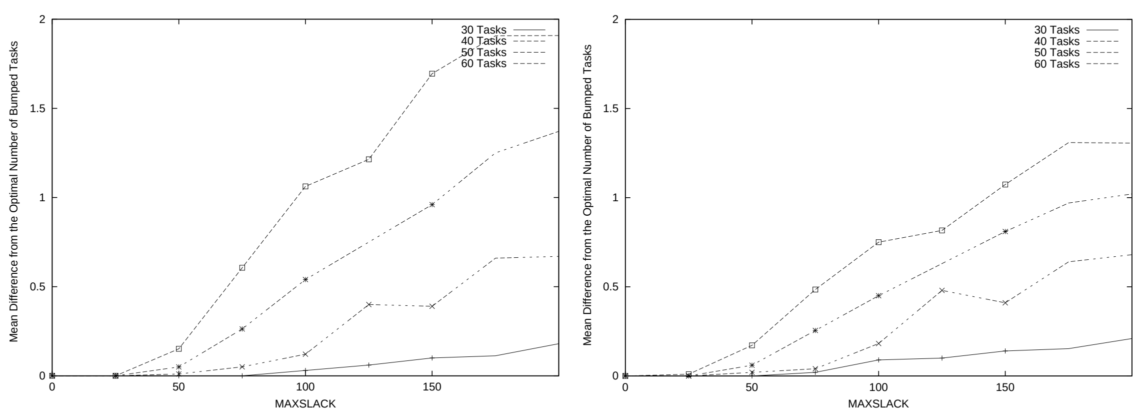

Figure 4: The mean difference from the optimal number of bumped tasks $\overline{\Delta_{best}}$ for the hill-climbing (left figure) and *Genitor* (right figure).

Finally, we consider the performance of hill-climbing and *Genitor*. For both algorithms, we define a single trial as an execution of the algorithm with a limit of 8000 solution evaluations. Due to the stochastic nature of these algorithms, we execute 30 independent trials of each algorithm on each of our test instances and define $|Bumps|$ for each algorithm as the minimal number of bumps observed in any of the 30 trials. The resulting means Δ_{best} for both algorithms are shown in Figure 4 (note: the scale on the y-axis is smaller than in previous graphs). The results indicate that both hill climbing and *Genitor* significantly outperform random sampling. Although the performance of *Genitor* and hill climbing is indistinguishable for $n = 30$ and $n = 40$, the performance of *Genitor* scales better than that of hill climbing for larger problem sizes. However, on these single resource problems *Genitor* still under-performs the best greedy heuristic ($Greedy_{DP}$), independently of n and *MAXSLACK*, and the run-time of $Greedy_{DP}$ is notably lower.

Distribution Variance Although $Greedy_{DP}$ provides excellent overall performance, the results presented in Figure 3 and 4 provide little indication of either worst-case performance or the frequency of optimal versus sub-optimal solutions.

In Figure 5 we present counts of the number of times that a particular algorithm found a solution at a distance of Δ_{best} away from the best known (and often optimal) solution. Results are presented for a problem with $n = 40, maxslack = 100$ and another with $n = 60, maxslack = 100$. The histograms show that $Greedy_{DP}$ is superior to $Greedy_{BN}$; the variance of $Greedy_{BN}$ is clearly higher. This is typical for other size problems and for different values of $maxslack$. $Greedy_{DP}$ out-performs *Genitor* on average, but the frequency of instances with large Δ_{best} is actually larger for $Greedy_{DP}$ than *Genitor*. For low n, *Genitor* exhibits less variance, while $Greedy_{DP}$ can occasionally generate poor-quality solutions. Hill climbing was slightly better than *Genitor* on a few smaller problems ($n = 30, n = 40$), but inferior on larger problems ($n = 50, n = 60$).

Discussion and Implications These results demonstrate that a simple greedy heuristic, originally developed in the context of $1|r_j|\sum U_j$ out-performs, on average, *Genitor*, an algorithm more computationally expensive and which was previously found to be most efficient for MuRRS. The equivalence of SiRRS to $1|r_j|\sum U_j$ scheduling enabled us to leverage algorithms for computing optimal solutions to problem instances. Our results demonstrate that SiRRS problems with characteristics found in real-world AFSCN problems can often be solved to optimality, and in general, near-optimal solutions can be found. We next look

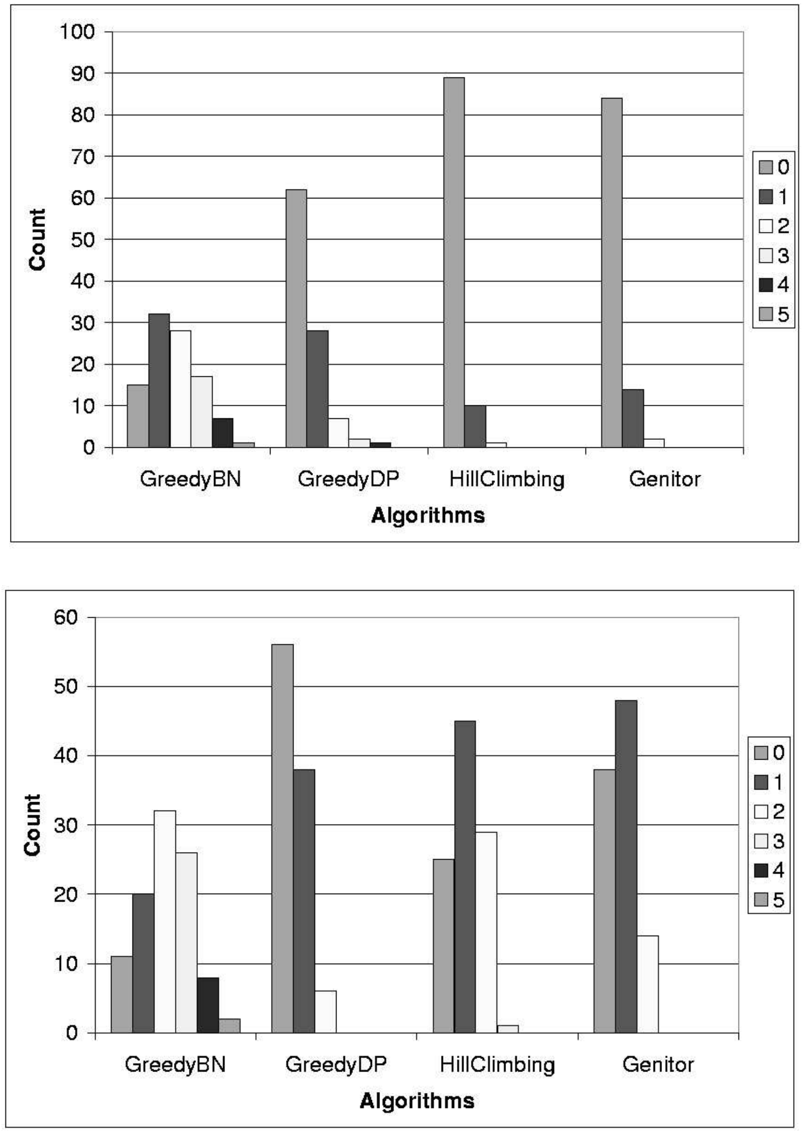

Figure 5: Histogram of counts of different numbers of bumps for the four algorithms. The top figure shows $n = 40, maxslack = 100$; the lower figure shows $n = 60, maxslack = 100$.

at multi-resource problems.

5 The Multi-Resource Range Scheduling with Alternatives: Algorithm Analysis and the AFIT Benchmark Problems

We now turn to an analysis of heuristic algorithm performance for MuRRS, which more accurately models the real AFSCN satellite range scheduling problem. At this stage, we have three key questions to answer: Can the results for single resource problems be leveraged for multi-resource problems? What is the best performing algorithm for satellite range scheduling? Why? To address these questions, we test the same algorithms as for SiRRS (modified as necessary to fit the demands of the new problem) on MuRRS problems. We then hypothesize and test an explanation for the observed performance.

As discussed in Section 2, heuristics for satellite range scheduling have previously been evaluated using only the seven problem instances in the AFIT benchmark. Although the set includes both high and low-altitude satellite requests along with alternatives, 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 total number of requests (low altitude and high altitude requests) to be scheduled for the seven problems are 322, 302, 300, 316, 305, 298, and 297, respectively. The number of low altitude requests for the seven problems are: 153, 137, 146, 142, 142, 144, and 142, respectively. Since 1992, the number of requests received during a typical day has increased substantially (to more than 500 each day), while the resources have remained more or less constant.

In our experimental setup, we replicated the conditions and the reported results from Parish's study [3]. 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 for each run. (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 also ran random sampling and hill-climbing 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.

Currently, no complete algorithm is available for computing optimal solutions for MuRRS. Consequently, all comparisons of heuristic algorithms for MuRRS are necessarily relative;

	Genitor			Hill Climbing			Random Sampling			$\text{Greedy}_{\text{DP}}$	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	21	10
2	4	4	0	6	10.96	2.04	11	13.83	1.08	22	6
3	3	3.03	0.18	11	15.4	2.73	16	17.76	0.77	25	7
4	2	2.06	0.25	12	17.43	2.76	16	20.20	1.29	23	7
5	4	4.1	0.3	12	16.16	1.78	15	17.86	1.16	18	6
6	6	6.03	0.18	15	18.16	2.05	19	20.73	0.94	28	7
7	6	6	0	10	14.1	2.53	16	16.96	0.66	22	6

Table 2: Performance of *Genitor*, hill climbing, 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 results of running the greedy heuristic $Greedy_{DP}$ are also included. The last column reports the performance of Schalck's Mixed-Integer Programming algorithm [1].

even if one algorithm consistently outperforms another, there is no assurance that the algorithm is generating optimal, or even near-optimal, solutions. To baseline the performance of the search algorithms, we also implemented a greedy constructive heuristic based on the $Greedy_{DP}$ as follows. For each of the resources, we use $Greedy_{DP}$ to schedule the requests that specify that resource as an alternative and are not scheduled yet; the result is an initial schedule. This means that each request will be successively considered for scheduling using $Greedy_{DP}$ on each of its alternative resources, until it is scheduled or until all alternative resources have been scheduled. Then we consider the unscheduled requests and attempt to insert them in the schedule. The unscheduled requests are considered in an arbitrary order; the request is scheduled at the earliest time on the first alternative resource available and bumped if none of the alternative resources are available.

The results are summarized in Table 2. Included in the table are the results obtained by Schalck using Mixed Integer Programming [1]. As previously reported, *Genitor* yields the best overall performance. $Greedy_{DP}$ results in the worst performance (it is often outperformed by random sampling).

5.1 Explaining the Performance on the Benchmarks

To exploit the differences in scheduling slack and the 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 now show that: (1) for more than 90% 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 most of the low-altitude requests before the high-altitude requests with which they interact, leading to the strong overall performance. While in the final permutations some high altitude requests do appear before low altitude requests, our hypothesis implies that most of these high altitude requests do not interact with the low altitude requests appearing after them. 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. We then used only those solutions that matched the best known solution. The resulting permutations found by *Genitor* are then interpreted in two ways. First, each permutation is decoded in the usual way: task are scheduled as they are encountered in the permutation, without regard to low and high-altitude requests. Second, it is decoded with all of the low-altitude requests scheduled first, followed by the high-altitude requests; both the low and high-altitude requests are still scheduled based on the order in which they appear in the permutation.

Thus, each permutation generates two schedules, the first decoded normally, the second decoded with low-altitude requests scheduled first. We then calculated how often the two schedules were identical, how often they were different but with the same evaluation and how often they were different with different evaluations.

The results are summarized in Table 3. 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 solution found by *Genitor* in any run. The number of times the schedules were

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
TOTAL	6182	4553	1120	509

Table 3: The effect of applying the split heuristic when evaluating best known schedules produced by *Genitor*.

identical ("Same Evaluation, Same Conflicts") is given in the third column. The number of times the two schedules are different, but have the same evaluation (number of bumps) is given in column "Same Evaluation, Different Conflicts". Finally, when the evaluations are different (the last column), the schedules produced by scheduling low-altitude requests first always result in worse performance.

By separating the requests from the permutations produced by *Genitor* into low and high-altitude requests, the evaluation of more than 80% of the schedules remains unchanged. The numbers in 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. But more than 90% of the time (i.e., in $(4553 + 1120)/6182$ cases), the same evaluation results when the same permutation is either, A) directly mapped to a schedule (first come, first served, based on the order of the permutation) to obtain a solution, or, B) all of the low-altitude requests from the permutation are filled first and then all of the high-altitude requests are scheduled. This suggests that *Genitor* is indeed scheduling low-altitude requests with high priority when appropriate.

Our second hypothesis is that using the split heuristic results in solutions with a small number of conflicts for the AFIT benchmarks. Figure 6 presents a summary of the results obtained when using *Genitor*, hill climbing, and Random Sampling without the split

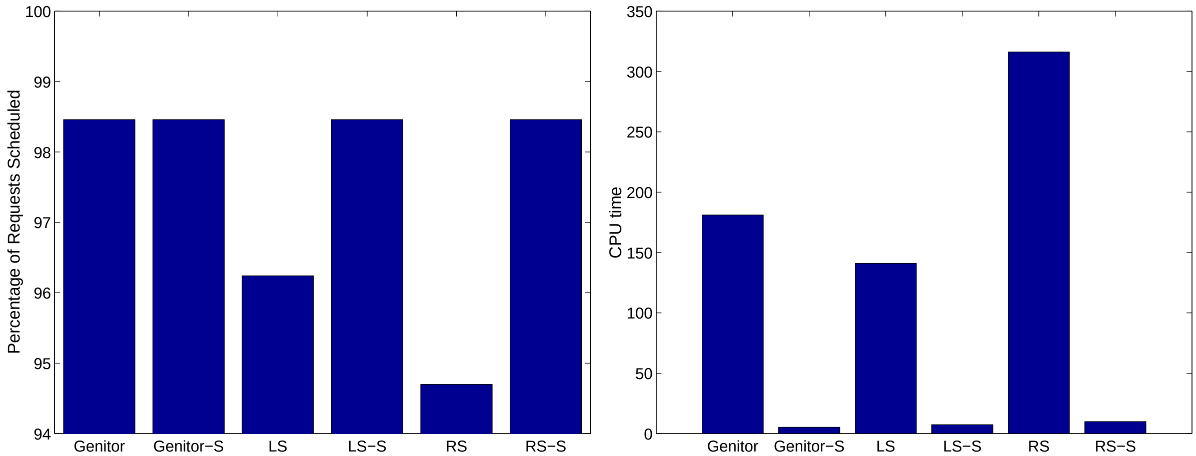

Figure 6: Algorithm performance for the seven AFIT benchmark problems.

heuristic (30 experiments, 8000 evaluations per experiment), as well as the split versions denoted by *Genitor*-S, Hill Climbing-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 percentage of requests scheduled. The left half of Figure 6 presents the average percentage of requests scheduled for the seven AFIT problems by each algorithm. The corresponding average CPU times (in seconds) appear in the right half of the figure. For all the problems, even the simplest algorithm, Random Sampling-S, finds the best known solutions, as illustrated in Table 4.

We can prove that *the low-altitude requests are scheduled optimally.* Since the split heuristic divides the low and high-altitude requests, the low-altitude requests (with no slack and no alternative windows for scheduling) correspond to multiple instances of SiRRS problems with no slack and no time windows; therefore, the low-altitude requests are optimally solved by the $Greedy_{IS}$ algorithm (the $Greedy_{IS}$ algorithm is defined in Section 2.1). We implemented $Greedy_{IS}$ and used it to schedule the low-altitude requests. In every case, the best-known solutions found by *Genitor* have sub-solutions over the subset of low-altitude requests that are identical in evaluation to those found by $Greedy_{IS}$, and therefore optimal.

This also suggests that another way to solve these problems is to use $Greedy_{IS}$ to schedule the low-altitude requests, and then use *Genitor* to schedule the remaining high-altitude requests.

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

Table 4: 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.

However, the 1992 AFIT benchmark problems are relatively easy, and this approach may not be best for other more difficult MuRRS problems.

5.1.1 When does the "split heuristic" fail?

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 7. 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. This problem fits the description of the Satellite Range Scheduling 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.

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

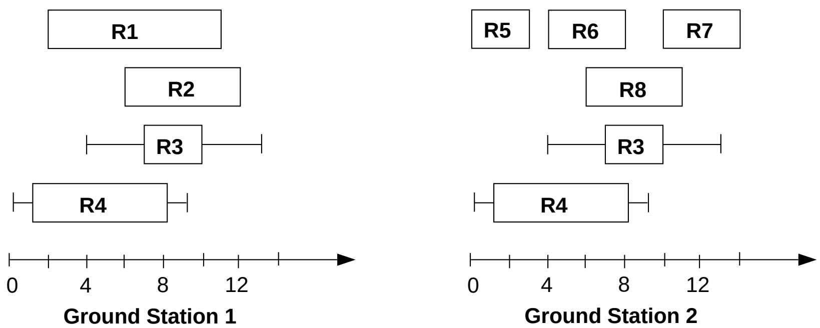

Figure 7: Example of a problem for which the split heuristic can not result in an optimal solution. Each ground station has two antennas; the only high-altitude requests are $R3$ and $R4$.

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.

6 Generalizing the AFIT Problems

Does the algorithm performance obtained for the AFIT benchmark transfer to contemporary problems with larger numbers of tasks? Do the results obtained by Parish showing that *Genitor* outperforms other algorithms also hold on other, larger problems? To explore these questions, we built a problem generator that produces problems by modeling features encountered in the real-world problems. Then we compared the results of running *Genitor*, hill climbing, and random sampling on problems produced by the problem generator to the results reported for the AFIT problems. We show that: (1) the performance of the split heuristic on the seven AFIT problems does not transfer to the problems produced by our generator but that (2) *Genitor* still consistently results in the smallest number of unscheduled requests.

6.1 New Problem Generator

We have designed a new, more realistic problem generator for MuRRS, which preserves some of the features of both SiRRS problems as well as the AFIT benchmark problems, while introducing new characteristics of the contemporary application.

The SiRRS problem generator described in section 4 requires parameters to control the request duration and the size of the time window. In the new generator, we use these parameters to model 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, and checking the health and status of a satellite. We classify the requests into four possible types:

- State of health (HS): short (5-15 minutes), flexible, common
- Maneuver (Mu), for example changing the orbit of a satellite: longer (20-60 minutes), inflexible, rare
- Payload download (PD), for example downloading data from the satellite: long (10-30 minutes), more flexible, common
- Payload commanding (PC): variable length (5-60 minutes), flexible, common

Our taxonomy is based on the fact that requests of a certain type share characteristics of duration, flexibility (the size of the time window versus the duration of the request) and frequency [4]. For each request type, we define the upper and lower bound on its duration T_i^{Dur} and the maximum slack $MAXSLACK$ (to determine the window size). The actual requests are generated using these parameters as described in section 4.

As a second feature, the new generator introduces models of customer behavior. We define three customer types (see Table 5), based on the fact that, for example, some customers will mostly generate health and status requests or payload command requests. For each customer, the number of requests to be generated is determined as a fraction of the total

[4] For now, we do not model periodic requests and therefore do not use the frequency information.

Customer Type	Predictability	Fract. of Reqs. for Each Request Type			
		HS	PD	PC	Mu
Operations and Maintenance	0.9	0.85	0.0	0.1	0.05
Image Intelligence	0.3	0.0	0.5	0.5	0.0
Signal Intelligence	0.7	0.05	0.1	0.85	0.0

Table 5: Customer types.

number of requests. We define customer patterns by specifying a time window midpoint, a resource, and the type of request. Customer patterns model the previous behavior of a customer (which could have been collected from past schedules). A database of customer patterns is used to produce the requests. For each customer type, the fraction of the requests generated for each request type and the *Predictability* are defined. The *Predictability* for each customer represents the fraction of requests that will be generated for this customer using customer patterns from the database. The rest of the requests corresponding to each customer will be randomly generated. The duration and window size for each request are determined using the parameters associated with the request type.

The AFIT problems separate the requests into low-altitude and high-altitude. The low-altitude requests tend to be shorter and have a duration equal to the corresponding visibility window. The high-altitude requests are more flexible, with the size of the visibility window larger than the requested duration. The AFIT problems assume that all the low-altitude requests have to be scheduled at the same ground station. In our new generator, we specify more than one ground station as possible alternatives for both low and high-altitude requests due to the higher contention for resources now. Also, the time windows specified for alternative ground stations are generated based on an offset in time. The offsets represent the time needed for a satellite in its orbit to become visible from one ground station to another. We use two databases (low-altitude and high-altitude) of offsets, which were compiled[5] from data collected on the Web (http://earthobservatory.nasa.gov/MissionControl/overpass.html) about the visibilities of various satellites from the locations of the nine ground stations. For security reasons, requests do not specify actual satellites in an accessible form; we chose

[5]Thanks to Ester Gubbrud, a senior undergraduate who spent part of the summer of 2001 working with Adele Howe, for helping us compile the databases.

```
For each customer
    For each request type
        Determine the number of requests NR to be generated
        Using the predictability for this customer, p, the number
          of requests generated using customer patterns is NR * p
        For each of the requests generated using customer patterns
                Choose without replacement a pattern
                Generate the request using the chosen pattern
        For each of the random requests (no patterns used)
                Generate the request using the request parameters
```

Figure 8: Pseudocode for the new generator.

some satellites to be representative for the kinds of the requests encountered in the Satellite Range Scheduling problem. We preserve the request duration for all the alternatives. The visibility window size is equal to the request duration for all the low-altitude requests, and it can vary for the high-altitude requests.

The process of generating the requests in the new generator is described in Figure 8. With a 0.5 probability, we determine for each request if it is a low-altitude or high-altitude request.

6.2 Algorithm Performance

We have previously observed that $Greedy_{DP}$ did not work well on the multi-resource problems contained in the AFIT benchmark. For completeness, we first compare $Greedy_{DP}$ to random sampling. As shown in Table 6, $Greedy_{DP}$ is again inferior to random sampling on the problems of sizes 300, 350, 400, and 450 and only slightly better than random sampling for problems of size 500 produced by the new problem generator.

We next repeat the experiments described for the AFIT problems (section 5) by running Genitor, hill climbing, and random sampling for problems produced by the new 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. For each algorithm, we performed 30

	Random Sampling		Greedy$_{DP}$	
Size	Mean	Variance	Mean	Variance
300	0.167	0.213	1.867	2.051
350	1.067	1.099	3.167	2.695
400	2.833	3.523	4.667	3.126
450	5.967	6.240	6.567	7.633
500	11.767	7.840	11.167	7.937

Table 6: The difference between the minimum number of bumps reported by the Random Sampling algorithm and Greedy$_{DP}$ 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		Hill Climbing		Random Sampling	
Size	Mean	Variance	Mean	Variance	Mean	Variance
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		Hill Climbing-S		Random Sampling-S	
Size	Mean	Variance	Mean	Variance	Mean	Variance
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

Table 7: 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.

runs with 8000 evaluations per run for each problem. Increasing the number of evaluations to 50k and of the population size to 400 did not improve the best solutions found for each problem. We recorded the number of unscheduled requests for each run.

Figure 9 shows that *Genitor* on average outperforms *Genitor*-S and both versions of hill climbing and random sampling, as well as the greedy constructive heuristic. In fact, as

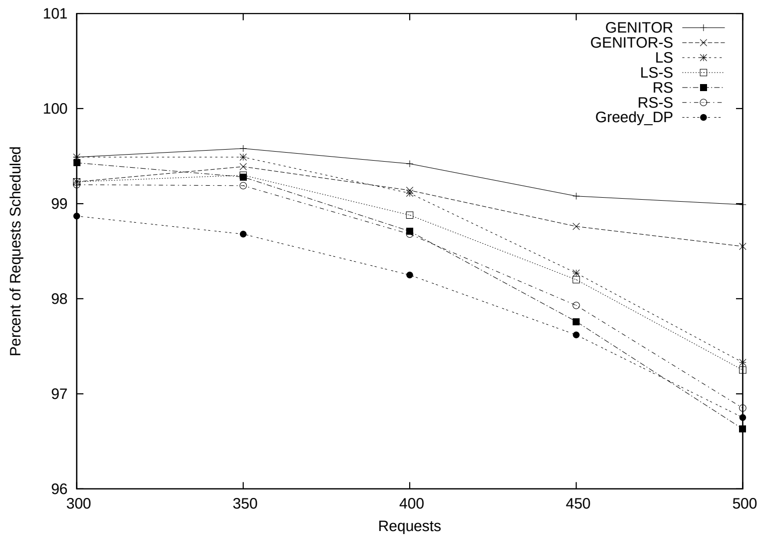

Figure 9: Average percent of requests scheduled by the no-split and split versions of each of the algorithms.

Table 7 shows, *Genitor* (without the split heuristic) always outperforms all the other algorithms. In Table 7, 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 for that problem in 30 runs. Then we average these differences over the 30 instances generated for each size; we also compute the variance. From both Figure 9 and Table 7, it is clear that the split heuristic always results in an average decrease in performance. The $Greedy_{DP}$ results in the worst performance except for problem size 500 where it slightly outperforms the no-split version of random sampling.

7 Conclusions

The Satellite Range Scheduling Problem is an important real world problem that impacts the use of expensive and limited resources. We structured our study as a progression. We start with a simple version of the problem, where only one resource is present. We follow by looking at a version of the problem studied at AFIT. Finally, for planning and experimental control purposes, we build a new problem generator that generalizes features found in the initial two versions of the problem studied and introduces new realistic features.

Although optimal algorithms have been developed for the SiRRS problem, we have not been able to modify them for MuRRS. Additionally, the simple heuristics that proved so effective in the SiRRS do not scale well to MuRRS, especially for the larger, newly generated problems discussed in Section 6.

We were able to show that Parish's earlier results with *Genitor* on AFIT's 1992 benchmarks scale-up to the more realistic contemporary problems. We also showed that *Genitor* performed so well on the AFIT problems because it effectively learned to split the problems into low and high-altitude requests, scheduling them separately. In fact, the seven problems in the AFIT benchmark are trivial to solve when a simple heuristic based on this separation is used.

However, when applied to more realistic problems, the split heuristic results in poor-quality solutions. Consequently, *Genitor* must be learning a different strategy for solving the contemporary problems. In future work, we plan on analyzing *Genitor*'s superior performance on the new problems and identifying what makes it work well. Additionally, we have recently obtained several days of recent actual data and are in the process of reverse engineering their characteristics to better tune our new generator. We are working on defining new objective functions, to provide the human schedulers with information about minimal changes in request specification such that more requests can be satisfied. We are also planning on implementing other algorithms such as HBSS and "Squeaky Wheel" Optimization to test their efficiency for Satellite Range Scheduling.

References

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

[2] T.D. Gooley. Automating the Satellite Range Scheduling Process. 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] Kwangho Jang. The Capacity of the Air Force Satellite Control Network. In *Masters Thesis*. Air Force Institute of Technology, 1996.

[5] 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.

[6] Michael S. Garey and David S. Johnson. Two-Processor Scheduling with Start-Times and Deadlines. *SIAM Journal on Computing*, 6:416–426, 1977.

[7] Michael S. Garey and David S. Johnson. *Computers And Intractability: A Guide To The Theory Of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

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

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

[11] 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.

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

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

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

[15] Philippe Baptiste, Claude Le Pape, and Laurent Peridy. Global Constraints for Partial CSPs: A Case-Study of Resource and Due Date Constraints. In Michael Maher and

Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98*, pages 87–101. Springer, 1998.

[16] S. Dauzère-Pérès. Minimizing Late Jobs in the General One Machine Scheduling Problem. *European Journal of Operational Research*, 81:131–142, 1995.

[17] Eric D. Taillard. Some Efficient Heuristic Methods for the Flow Shop Sequencing Problem. *European Journal of Operational Research*, 47:65–74, 1990.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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 / MIT Press.

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

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

[32] J.L. Bresina. Stochastic Heuristic Search and Evaluation Methods for Constrained Optimization. In *Ph.D. Thesis.* Graduate School- New Brunswick, Rutgers, The State University of New Jersey, 1998.

[33] J. Frank, A. Jonsson, R. Morris, and D.E. Smith. Planning and Scheduling for Fleets of Earth Observing Satellites. *International Symposium on Artificial Intelligence, Robotics, Automation and Space*, 2001.

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

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

[36] Eric D. Taillard. Comparision of Iterative Searches for the Quadratic Assignment Problem. *Location Science*, 3(2):87–105, 1995.

[37] Jean-Paul Watson, Laura Barbulescu, L. Darrell Whitley, and Adele E. Howe. Contrasting Structured and Random Permutation Flow-Shop Scheduling Problems: Search Space Topology and Algorithm Performance. *INFORMS Journal on Computing*, 14(2), 2002.

[38] Philippe Baptiste, Laurent Peridy, and Eric Pinson. A Branch and Bound To Minimize the Number of Late Jobs on a Single Machine with Release Time Constraints. *European Journal of Operational Research*, 144(1):1–11, 2003.