Journal of Artificial Intelligence Research X (X) X Submitted X; published X

Algorithm Performance Factors for an Oversubscribed Scheduling Application

Laura Barbulescu LAURA@CS.COLOSTATE.EDU
Adele E. Howe HOWE@CS.COLOSTATE.EDU
L. Darrell Whitley WHITLEY@CS.COLOSTATE.EDU
Mark Roberts MROBERTS@CS.COLOSTATE.EDU
Computer Science Department
Colorado State University
Fort Collins, CO 80523 USA

Abstract

We study the factors that influence algorithm performance for an oversubscribed scheduling application, allocating satellite communication time slots on ground station antennas for the U.S. Air Force. We identify a set of fairly simple algorithms that perform best relative to several constraint directed and heuristic techniques. The algorithms in our set are: local search, a genetic algorithm (GA) and Squeaky Wheel Optimization (SWO). These algorithms are very different from each other in the way they traverse the search space. We identify the problem characteristics that favor each of the algorithms in the set, such as the redundancy of the search space, the size of the search neighborhood and the proclivity of plateaus. We find that the redundancy of the search space is the key factor in local search performance. For the GA and SWO, the good performance is the result of taking long leaps in the search space, by operating multiple changes on the schedule at each step. Initializing the GA and SWO with a greedy solution is also a factor in algorithm performance, but is not as crucial as making long leaps.

1. Introduction

Although many papers show empirically that some search algorithm performs best on specific problems, one suspects that other algorithms (either ones that were not tested or better implementations of ones that were) may do just as well. Usually, the research fails to identify what makes an algorithm perform well or poorly, which would explain whether other algorithms might do so well.

With the exception of a few recently published works (e.g., (Aickelin & White, 2004) (Watson, 2003)), the understanding of *why* an algorithm is a good fit to a scheduling problem is very limited. Hooker's"competitive testing paradigm" (Hooker, 1995) is prevalent when developing algorithms for scheduling. Clearly, factors such as the size of the search space, the presence of plateaus in the search space, the distribution and size of the plateaus favor some algorithms to the detriment of others. Gaining insight into the problem features that influence algorithm performance is important for at least three reasons: 1) It may suggest algorithm improvements to better fit the domain, 2) Given new problem instances in the same domain, the algorithm most likely to perform best can be selected based on particular features of the new instances, and 3) Making decisions on the applicability of a certain algorithm to a new domain can be based on the presence of similar problem features. In this paper, in contrast to the competitive testing paradigm, we identify *a set* of algorithms that perform well for our scheduling domain and then investigate reasons for the performance results.

Our target application is an oversubscribed scheduling application with alternative resources. AFSCN (Air Force Satellite Control Network) access scheduling requires assigning access requests (communication relays to U.S.A. government satellites) to specific time slots on an antenna at a ground station. It is oversubscribed in that not all tasks can be accommodated given the available resources[1]. It is also challenging and relevant in that human schedulers perform the task every day with minimal automation at present.

While scheduling problems are hard to solve in general, oversubscribed scheduling domains present additional challenges: not only does the solution need to specify the start times and resources assigned to the tasks, it also needs to identify a subset of the tasks that can be feasibly scheduled such that an objective function is optimized. A common objective is to minimize the weighted summation of tasks that are not included in the final schedule (e.g., (Pemberton, 2000) (Wolfe & Sorensen, 2000) (Kramer & Smith, 2003)). The weight is usually a direct function of the priority of the task; we disregard the weights because AFSCN tasks do not include priorities. In prior research on AFSCN, the objective function maximizes the number of tasks scheduled (Schalck, 1993) (Gooley, 1993) (Parish, 1994). In our research, we also considered a new objective function that minimizes the overlaps of tasks in a schedule containing *all* tasks.

We implemented various algorithms for the AFSCN scheduling domain. Some rather complex algorithms, such as repair-based using domain specific knowledge or constraint-based scheduling heuristics, failed to identify good solutions in earlier research (Barbulescu, Watson, Whitley, & Howe, 2004b) (Barbulescu, Howe, Whitley, & Roberts, 2004a) (Barbulescu, Whitley, & Howe, 2004c). We have found a set of fairly simple algorithms that worked well: local search using shifting, the Genitor genetic algorithm (GA), and Squeaky Wheel Optimization (SWO). All are designed to traverse essentially the same search space: solutions are represented as permutations of tasks, which a greedy schedule builder converts into a schedule by assigning start time and resources to the requests in the order in which they appear in the permutation. However, these algorithms vary in the *way* they traverse the search space.

We formulated several hypotheses about why particular algorithms excel. First, plateaus appear to be a dominant feature of this search space due to both the discrete nature of the objective functions and the fact that the schedule builder converts multiple permutations into identical schedules (n to 1 mapping). Each of the algorithms handle plateaus differently. Local search randomly walks on the plateaus until it finds exits to lower plateaus: the higher percentage of the space occupied by plateaus, the more random wandering is likely for local search. We found that indeed 80% of the moves evaluated by local search are on plateaus. We show that the n to 1 mapping from the permutation to the schedule space is the main factor in algorithm performance for local search. Additionally, we show that the ordering of the neighbors matters a great deal for expediting plateau traversal for local search.

Second, we hypothesized that the GA is discovering patterns of interaction between tasks. We found that the GA finds some simple patterns that exploit domain knowledge.

1. To be considered to be oversubscribed, not all problem instances need overtax the available resources, but for our application, that appears to be the case.

However, the simple domain knowledge is not enough to account for the performance; the GA appears to also be discovering fairly complex relationships in the data.

Third, we hypothesized that the size of the search space and the proclivity of plateaus favor algorithms that take long, directed leaps across the search space. By leveraging this similarity between the GA and SWO, we show that their power is obtained from taking multiple steps and therefore instituting large changes to the schedules, as opposed to taking single steps, which translate into minor, incremental changes to the schedule.

Finally, we investigated whether initializing the search closer to the best solutions is the key to performance. We found that narrowing the distance to the optimal solution helps but is not by itself enough.

The contribution of our research work is threefold: 1) We identify fairly simple algorithms to solve an oversubscribed scheduling application with multiple alternative resources. 2) We present techniques for analyzing algorithm performance, and 3) We identify problem characteristics that influence algorithm performance, which are likely to hold on similar problems.

The remainder of the paper is organized as follows. In Sections 2, we describe the AFSCN scheduling domain and present some results showing the amount of task interaction present in the problems. Next, in Section 3, we review related work on other oversubscribed, multiple resource scheduling applications. We present the algorithms we chose for this study and their performance on two sets of real problems in Sections 4 and 5. In Section 6, we identify factors that influence algorithm performance, first for the local search, then for the GA and SWO. Such factors include the redundancy of the search space, the presence of the plateaus, the large size of the search neighborhoods, and the discretization of the objective function. Finally, we offer some observations from our analyses and discuss future directions of research.

2. AFSCN Scheduling

The U.S.A. Air Force Satellite Control Network (AFSCN) is currently responsible for coordinating communications between civilian and military organizations and more than 100 USAF managed satellites. Space-ground communications are performed using 16 antennas located at nine tracking stations around the globe[2]. Figure 1 shows a map of the current configuration of AFSCN; this map shows one fewer tracking station and antennae than are in our data, due to those resources apparently having been taken off-line recently. Customer organizations submit task requests to reserve an antenna at a tracking station for a specified time period based on the visibility windows between target satellites and tracking stations. Two types of task requests can be distinguished: low altitude and high altitude orbits. The low altitude tasks specify requests for access to low altitude satellites; such requests tend to be short (e.g., 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, with large visibility windows.

2. The U.S.A. government is planning to make the AFSCN the core of an Integrated Satellite Control Network for managing satellite assets for other U.S.A. government agencies as well, e.g., NASA, NOAA, other DoD affiliates. By 2011, when the system first becomes operational, the Remote Tracking Stations will be increased and enhanced to accommodate the additional load.

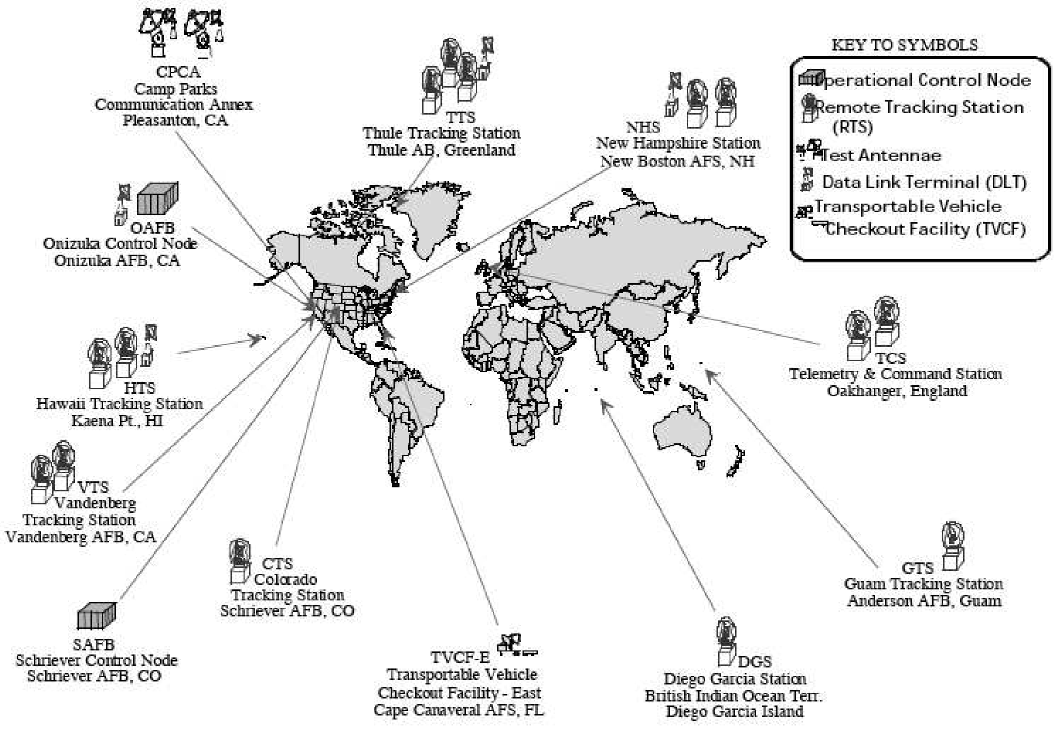


Figure 1: Map of the current AFSCN network including tracking stations, control and relay. The figure was produced for U.S.A. Space and Missile Systems Center (SMC).

Approximately 500 requests are typically received for a single day. Separate schedules are produced by a staff of human schedulers at Schriever Air Force Base for each day. Of the 500 requests, often about 120 conflicts remain after the first pass of scheduling.

From real problem data, we extract a description of the problem specification in terms of task requests to be scheduled, with their corresponding type (low or high altitude), duration, time windows and alternative resources. The real data also include information about satellite revolution numbers, optional site equipment, tracking station maintenance times (downtimes), possible loss of data due to antenna problems, various comments etc.; we do not incorporate such information in our problem specification. The information about the type of the task (low or high altitude) as well as the identifier for the satellite involved are included in the task specification. We do not know how the satellite identifier corresponds to an actual satellite and so rely on precomputed visibility information which is present in the actual requests.

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 $j \geq 0$ pairs of the form (R_i, T_i^{Win}), each identifying a particular alternative resource (antenna) and time window for the task.

ID	Date	# Requests	# High	# Low	Best Conflicts	Best Overlaps
A1	10/12/92	322	169	153	8	104
A2	10/13/92	302	165	137	4	13
A3	10/14/92	311	165	146	3	28
A4	10/15/92	318	176	142	2	9
A5	10/16/92	305	163	142	4	30
A6	10/17/92	299	155	144	6	45
A7	10/18/92	297	155	142	6	46
R1	03/07/02	483	258	225	42	774
R2	03/20/02	457	263	194	29	486
R3	03/26/03	426	243	183	17	250
R4	04/02/03	431	246	185	28	725
R5	05/02/03	419	241	178	12	146

Table 1: Problem characteristics for the 12 days of AFSCN data used in our experiments. ID is used in other tables. Best conflicts and best overlaps are the best known values for each problem for these two objective functions.

We obtained 12 days of data for the AFSCN application. The first seven days are from a week in 1992 and were given to us by Colonel James Moore at the Air Force Institute of Technology. These data were used in the first research projects on AFSCN. We obtained an additional five days of data from schedulers at Schriever Air Force Base. Table 2 summarizes the characteristics of the data. We will refer to the problems from 1992 as the A problems, and to the more recent problems, as the R problems.

While requests are made for a specific antenna, often a different antenna at the same tracking station may serve as an alternate because it has the same capabilities. We assume that all antennas at a tracking station can serve as alternate resources[3]. While this is not always the case in practice, the same assumption was made by previous research from the Air Force Institute of Technology (AFIT). A low altitude request specifies as possible resources the antennas present at a single tracking station (for visibility reasons, only one tracking station can accommodate such a request). Usually there are two or three antennas present at a tracking station, and therefore, only two or three possible resources are associated with each of these requests. High altitude requests specify all the antennas present at all the tracking stations that satisfy the visibility constraints; as many as 14 possible alternatives are specified in our data.

Previous research and development on AFSCN scheduling focused on minimizing the number of request conflicts for AFSCN scheduling, or alternatively, maximizing 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

3. In fact, large antennas are needed for high altitude requests, while smaller antennas can handle the low altitude requests. Depending on the type of antennas present at a tracking station, not all antennas can always serve as alternate resources for a request.

carry out AFSCN scheduling[4]. Satellites are valuable resources, and the AFSCN operators work to fit in every request. What this means in practice is that after negotiation with the customers, 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 or deferred to another day. In fact, in practice, long maintenance tasks are often deferred until absolutely necessary.

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[5], this is the only evaluation function that has been applied to AFSCN scheduling. However, given that all requests need to be eventually scheduled, we designed a new evaluation criterion that schedules all the requests by allowing them to overlap and minimizing the sum of overlaps between conflicting tasks.

A simple example of a problem instance is presented in Figure 2. There are two tracking stations and two resources (two antennas) at each tracking 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 tracking stations. The rest of the requests are low-altitude requests. $R1$ and $R2$ request the first tracking station, while $R5$, $R6$, $R7$, and $R8$ request the second tracking station. The low-altitude requests can be scheduled only at a specific tracking station, with a fixed start and end time, while the high-altitude requests have alternative resources and a time window specified. To minimize the number of conflicts, an optimal schedule will bump one of the requests. For example, if $R8$ is the request bumped, $R1$ and $R2$ can be scheduled at the first tracking station, and $R3$, $R4$, $R5$, $R6$ and $R7$ can be scheduled at the second tracking station. In Figure 3, we show this optimal schedule, where $R8$ is conflicting with the requests scheduled. If the new evaluation criterion, minimizing the sum of overlaps, is used, request $R8$ could either be scheduled on antenna $A1$ or antenna $A2$ at Tracking 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 that provides guidance about the fewest modifications needed to accommodate $R8$. Additionally, the objective function means that the long maintenance tasks (which are likely to be much harder to reinsert in the schedule) are not simply bumped, but rather are placed in the least contentious spot.

2.1 Task Interaction in AFSCN problems

The most distinctive characteristic of AFSCN scheduling is the availability of time/resource alternatives. When placing a task, one must consider not only where it might fit in a window, but also whether it could fit into an alternative. Taking a simplistic view, this may

4. We met with the several of the schedulers at Schriever to discuss their procedure and have them cross-check our solution. We appreciate the assistance of Brian Bayless and William Szary in setting up the meeting and giving us data.
5. Commercial software has been developed to assist the human schedulers, but the exact algorithms are proprietary.

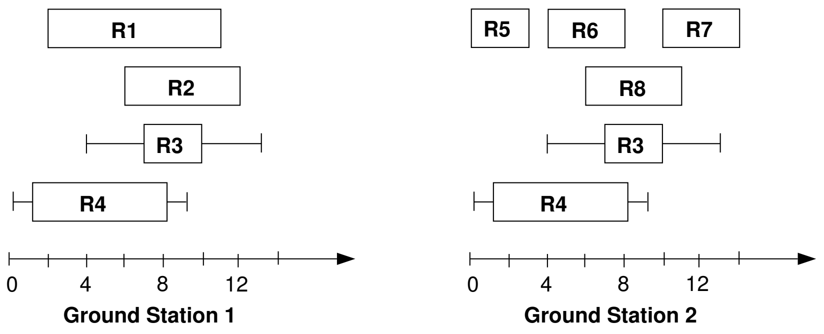


Figure 2: Example of a simple problem. Each tracking station has two antennas; the only high-altitude requests are $R3$ and $R4$.

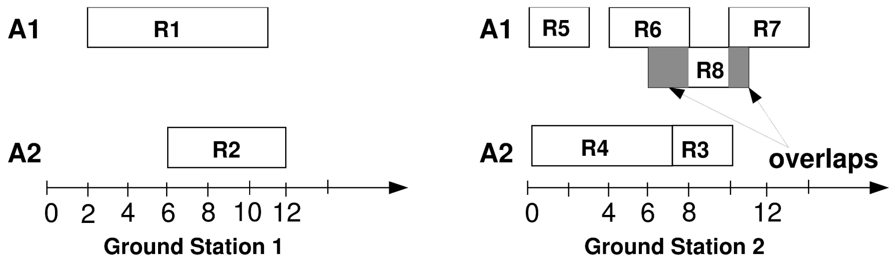


Figure 3: Optimizing the sum of overlaps.

dramatically increase the branching factor of deciding how to place tasks. Additionally, this can lead to a cascade effect of considering mutual constraints between requests. The question is how much do tasks interact and how does this affect problem difficulty?

First, we examined the nature of the contention in AFSCN. An oversubscribed problem is not oversubscribed on all resources at all times. Many heuristics focus attention on time periods or task combinations of maximal contention (e.g., CBAslack (Smith & Cheng, 1993), SumHeight (Beck, Davenport, Sitarski, & Fox, 1997)). We built contention graphs by adding all tasks into all of their possible slots. As with SumHeight, for each task, its individual demand for a resource is computed probabilistically, based on the assumption that there exists a uniform probability distribution over the possible start times. Four points characterize the individual demand:

$$\left(est, \frac{1}{|STD|}\right), \left(lst, \frac{min(|STD|, dur)}{|STD|}\right), \left(eft, \frac{min(|STD|, dur)}{|STD|}\right), (lft, 0)$$

where dur is the duration of the task (T_i^{Dur}), $|STD|$ represents the number of possible start times, est represents the earliest starting time ($T_i^{Win}(LB)$), lst is the latest possible start time, eft is the earliest final time, and lft represents the latest final time ($T_i^{Win}(UB)$).

The aggregate demand for a resource is then obtained by summing the individual demands for all the tasks that can be scheduled on that resource. For our problems, we found

Day	SumHeight	#Requests	Mean	#Alts	Overlap
A1	7.72	6	2.98	5.34	0.47
A2	8.78	8	3.63	6.34	0.82
A3	8.99	7	3.34	6.12	0.67
A4	7.37	4	3.44	6.25	1.0
A5	8.17	6	3.32	6.15	0.73
A6	8.50	7	3.34	6.16	0.76
A7	8.53	8	2.85	6.18	0.43
R1	10.74	8	3.66	4.46	0.5
R2	10.48	7	3.22	4.84	0.52
R3	8.83	7	2.55	4.42	0.33
R4	9.21	9	2.87	4.54	0.43
R5	7.59	5	2.32	4.50	0.80

Table 2: Statistics on contention in AFSCN data. Second column is the maximum SumHeight value. Third is number of requests that contribute the maximum value to the slot (with a probability of one, these requests require that particular time slot). Fourth is the average contention over all resources at all time slots. Fifth is the mean number of alternatives per request. Last column is the mean percentage of overlap in alternatives shared by pairs of requests in contention.

that demand is consistent (i.e., only a few time slots that have zero requests) and can get pretty high (more than nine tasks vying for the same time slot). Table 2 summarizes contention for the AFSCN data. The statistics in the table suggest that the most contentious slots are likely to be hard to resolve. They have high contention values that involve four to nine tasks having that slot as their required interval.

Figure 4 shows the contention profile for the resource with the most requests competing for the same resource on the day with the highest demand (a worst case scenario). The *Requests* line in the graph represents the number of requests contributing to the contention line. The third column in Table 2 represents the number of requests that *always* require that particular time slot; note that the number of requests contributing to the contention for the time slot (the *Requests* line in the graph) is always higher. The *Mean Contention* line represents the average contention for this resource, computed using the individual contention values corresponding to each minute of the day. In this case, contention is a constant factor, and the spikes include up to 18 tasks. Because a large number of requests may be vying for the same slots, any constructive algorithm that orders by contention will have a high branching factor.

Second, we examined the overlap in alternatives at the points of highest contention. We found that many requests do share alternatives. The results are summarized in Table 2. The last columns in Table 2 show that many alternatives are possible, and many (or all) tasks share the same set of alternatives. This may help reduce somewhat the branching factor as constructive search progresses; unfortunately, we also found that most of them prioritize the alternatives similarly, making it somewhat more difficult to trade-off placement.

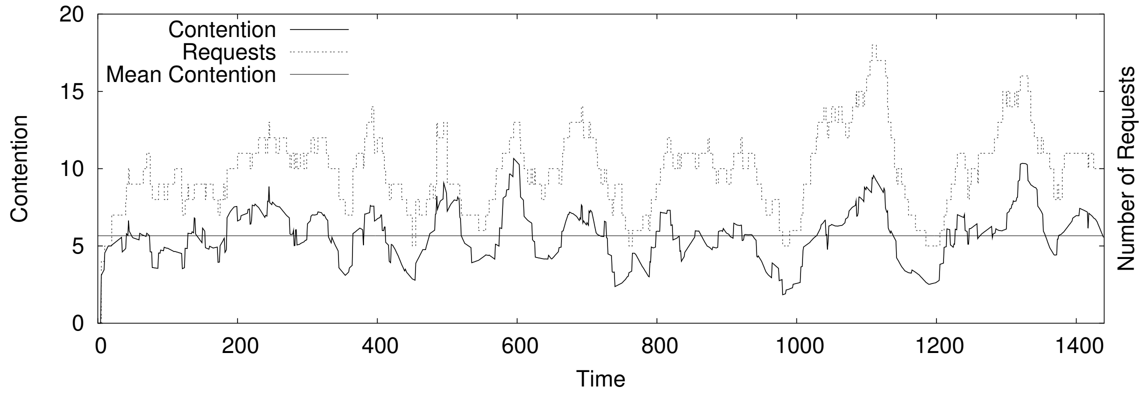


Figure 4: Example of resource contention graph for 3/7/2002. This resource had the highest number of tasks (18) contending for a single slot.

3. Related Work

The AFSCN application is a multiple resource, oversubscribed problem. Examples of other such applications are USAF Air Mobility Command (AMC) airlift scheduling (Kramer & Smith, 2003), NASA's shuttle ground processing (Deale, Yvanovich, Schnitzuius, Kautz, Carpenter, Zweben, Davis, & Daun, 1994), scheduling telescope observations (Bresina, 1996) and satellite observation scheduling (Frank, Jonsson, Morris, & Smith, 2001; Globus, Crawford, Lohn, & Pryor, 2003).

AMC scheduling assigns delivery missions to air wings (Kramer & Smith, 2003). Their system adopts an iterative repair approach by greedily creating an initial schedule ordering the tasks by priority and then attempting to insert unscheduled tasks by retracting and re-arranging conflicting tasks.

The Gerry scheduler was designed to manage the large set of tasks needed to prepare a space shuttle for its next mission (Zweben, Daun, & Deale, 1994). Tasks are described in terms of resource requirements, temporal constraints and required time windows. The original version used constructive search with dependency-directed backtracking, which was not adequate to the task; a subsequent version employed constraint-directed iterative repair.

In satellite scheduling, customer requests for data collection need to be matched with satellite and tracking station resources. The requests specify the instruments required, the window of time when the request needs to be executed, 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 tracking 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).

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 and Sorensen (2000) 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) compares a genetic algorithm, simulated annealing, Squeaky Wheel Optimization (Joslin & Clements, 1999) 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 (Bresina, 1996). HBSS (Bresina, 1996) 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) (to Manage the New Generation of Agile Earth Observation Satellites,). 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 (Gooley, 1993) and Schalck (Schalck, 1993) describe algorithms based on mixed-integer programming (MIP) and insertion heuristics, which achieved good overall performance: 91% – 95% of all requests scheduled. Parish (Parish, 1994) uses the *Genitor* (Whitley, 1989) genetic algorithm, which scheduled roughly 96% of all task requests, out-performing the MIP approaches. All three of these researchers used the AFIT benchmark suite[6] consisting of seven problem instances, representing actual AFSCN task request data and visibilities for seven consecutive days from October 12 to 18, 1992. Later, Jang (Jang, 1996) 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.

While the general problem of AFSCN Scheduling with minimal conflicts is $\mathcal{NP}$-complete, special subclasses of AFSCN Scheduling are polynomial. Burrowbridge (Burrowbridge, 1999) 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-

6. 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.

known *greedy activity-selector* algorithm (Cormen, Leiserson, & Rivest, 1990) is used to schedule the requests since it yields a solution with the maximal number of scheduled tasks. To schedule low altitude requests on one of the multiple antennas present at a particular ground station, we extended the greedy activity-selector algorithm for multiple resource problems. We proved that this extension of the greedy activity-selector optimally schedules the low altitude requests for the general problem of AFSCN Scheduling (Barbulescu et al., 2004b).

The search space for AFSCN scheduling has many large flat regions (plateaus). This is similar to MAXSAT (Gent & Walsh, 1993). In MAXSAT, these plateaus result from the fact that the evaluation function is a linear combination of subfunctions: each clause is a subfunction. A Walsh analysis shows that unless the ratio of clauses to variables is extremely high, there will exist large hyperplanes that will have constant evaluation and therefore correspond to plateaus (Heckendorn, 1999). In oversubscribed scheduling problems, moves affecting only requests that do not compete for the same resources or time windows have no impact on the evaluation function. Strategies for "walking" across these flat regions are therefore key components for successful search in both domains. One strategy used for MAXSAT, which is similar in flavor to the SWO strategy used here, is JumpSAT (Gent & Walsh, 1995). Instead of flipping *one* variable in an unsatisfied clause, one variable is flipped in *each* unsatisfied clause. While not state of the art for MAXSAT, JumpSat was shown to outperform greedy local search (the GSAT algorithm). MAXSAT and AFSCN scheduling differ in two important ways. First, the neighborhood size is $O(N)$ for MAXSAT while it is $O(N^2)$ for scheduling (where N is the number of variables or requests). Second, MAXSAT has a fast partial evaluation that allows one to quickly evaluate a move. Our scheduling domain requires that a new schedule be built from scratch and evaluated after every move.

4. Algorithms

We implemented a variety of algorithms for AFSCN scheduling: iterative repair, heuristic constructive search, local search, a genetic algorithm (GA), Squeaky Wheel Optimization (SWO). We found that local search, the GA and SWO work best for AFSCN scheduling. We also considered constructive search algorithms based on texture (Beck et al., 1997) and slack (Smith & Cheng, 1993) constraint-based scheduling heuristics. We implemented straightforward extensions of such algorithms for our application. The results were poor; the number of request tasks combined with the presence of multiple alternative resources for each task make the application of such methods impractical.

Solution Representation Permutation based representations are frequently used when solving scheduling problems, e.g., (Whitley, Starkweather, & Fuquay, 1989), (Syswerda, 1991) (Wolfe & Sorensen, 2000) (Aickelin & Dowsland, 2003) (Globus et al., 2003). All of our algorithms, except iterative-repair, 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. Each task request is assigned to the first available resource (from its list of alternatives) and at the earliest possible starting time. When minimizing the number of conflicts, if the request cannot be scheduled on any of the alternative resources, it is dropped from the schedule (i.e., bumped). When minimizing the sum of overlaps, if a request cannot be scheduled without conflict on any of the alternative resources, it overlaps; we assign such a request to the resource on which the overlap with requests scheduled so far is minimized. Note that our schedule builder does favor the order of the resources even though no preference is specified for any of the alternatives.

4.1 Iterative Repair

Iterative repair methods have been successfully used to solve various oversubscribed scheduling problems, e.g., Hubble Space Telescope observations (Johnston & Miller, 1994) and space shuttle payloads (Zweben et al., 1994; Rabideau, Chien, Willis, & Mann, 1999). NASA's ASPEN (A Scheduling and Planning Environment) framework (Chien, Rabideau, Knight, Sherwood, Engelhardt, Mutz, Estlin, Smith, Fisher, Barrett, Stebbins, & Tran, 2000), has been used to model and solve real-world space applications such as scheduling EOS. ASPEN employs both constructive and repair-based methods (Sherwood, Govindjee, Yan, Rabideau, Chien, & Fukunaga, 1998; Engelhardt, Chien, Barrett, Willis, & Wilklow, 2001). More recently, Kramer et al. (Kramer & Smith, 2003) 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 (Gooley, 1993) 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.

Gooley's algorithm has two phases. 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 from 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 a greedy algorithm (Barbulescu et al., 2004b) proven to schedule the optimal number of low altitude requests. 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 (this is similar to the flexibility measure defined by Kramer and Smith for AMC (Kramer & Smith, 2003)); ties are broken based on the number of alternative resources specified (fewer alternatives scheduled first). The insertion of the high altitude requests in the schedule is based on various domain specific heuristics.

4.2 Local Search

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 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. Our initial implementation of hill-climbing introduced a bias in the order in which we were checking the neighbors: while we chose the position x by random, we were checking in order all the neighbors obtained by shifting the job at position x into positions 0, 1, ..., n-1, accepting both equal moves and improving moves. Our second implementation completely randomizes the neighborhood (choosing by random both x and y).

4.3 Genetic Algorithm

Genetic algorithms were found to perform well in some early studies (Parish, 1994) and for an abstraction of NASA's Earth Observing Satellite (EOS) scheduling problem (Wolfe & Sorensen, 2000). For our studies, we used the version of *Genitor* originally developed for a warehouse scheduling application (Starkweather, McDaniel, Mathias, Whitley, & Whitley, 1991). Like all genetic algorithms, *Genitor* maintains a population of solutions; in our implementation, we fixed the population size to be 200. 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. Selection of parent solutions is based on the rank of their fitness, relative to other solutions in the population. Following Parish (1994) and (Starkweather et al., 1991), we used Syswerda's (1991) position-based crossover operator.

Syswerda's position-based crossover operator starts by selecting a number of random positions in the second parent. The corresponding selected elements will appear in exactly the same positions in the offspring. The remaining positions in the offspring are filled with elements from the first parent in the order in which they appear in this parent:

Parent 1:	A	B	C	D	E	F	G	H	I	J
Parent 2:	C	F	A	J	H	D	I	G	B	E
Selected Elements:		*	*			*			*	
Offspring:	C	F	A	E	G	D	H	I	B	J

4.4 Squeaky Wheel Optimization (SWO)

SWO (Joslin & Clements, 1999) repeatedly iterates through a cycle composed of three phases. First, a greedy solution is built, based on priorities associated with the elements in the problem. Then, the solution is analyzed and the elements causing "trouble" are identified, based on their contribution to the objective function. Third, the priorities of such "trouble makers" are modified, such that they will be considered earlier during the next iteration. The cycle is then repeated, until a termination condition is met.

We constructed the initial greedy permutation for SWO by sorting the requests in increasing order of their flexibility. Our flexibility measure is identical to that defined for the AMC oversubscribed scheduling application (Kramer & Smith, 2003): the duration of the request divided by the average time window on the possible alternative resources. We break ties based on the number of alternative resources available. For requests with equal flexibilities and numbers of alternative resources, the earlier request is scheduled first. For multiple runs of SWO, we restarted it from a modified permutation created by performing 20 random swaps in the initial greedy permutation.

When minimizing the sum of overlaps, we identified the overlapping requests as the "trouble spots" in the schedule. We sorted the overlapping requests in increasing order of their contribution to the sum of overlaps. We associated with each such request a distance to move forward, based on its rank in the sorted order. We fixed the minimum distance of moving forward to one and the maximum distance to five (this seems to work better than other possible values we tried). The distance values are equally distributed among the ranks. We moved the requests forward in the permutation in increasing order of their contribution to the sum of overlaps (smaller overlaps first). We tried versions of SWO where the distance to move forward is proportional with the contribution to the sum of overlaps or is fixed. However, these versions performed worse than the rank based distance implementation described above. When minimizing conflicts in the schedule, since all conflicts have an equal contribution to the objective function, we decided to move them forward for a fixed distance of five (we tried values between two and seven and five was best).

4.5 Heuristic Biased Stochastic Sampling (HBSS)

HBSS (Bresina, 1996) is an incremental construction algorithm in which multiple root-to-leaf paths are stochastically generated. Instead of randomly choosing a move, an acceptance probability is associated with each possible move. This acceptance probability is based on the rank of the move assigned by the heuristic. A bias function is then applied to the ranks, and the resulting values are normalized. As noted by Bresina (1997:271), the choice of bias function "reflects the confidence one has in the heuristic's accuracy - the higher the confidence, the stronger the bias." At each step, the HBSS algorithm needs to heuristically choose the next request to schedule from the unscheduled requests. We used the flexibility measure as described for SWO to rank the unscheduled requests and a relatively strong exponential bias function, due to the strength of the flexibility heuristic: $bias(r) = e^{-r}$, where r is the rank of a move.

5. Algorithm Performance

The results of running each of the algorithms when minimizing the number of conflicts and the sum of overlaps are summarized in Tables 3 and 4 respectively. For *Genitor*, local search and SWO, we report the best and mean value and the standard deviation observed over 30 runs, with 8000 evaluations per run. For HBSS, the statistics are taken over 240,000 samples. Both *Genitor* and local search were initialized from random permutations. The best known values for the sum of overlaps (see Table 2) were obtained by running *Genitor* with the population size increased to 400 and up to 50,000 evaluations. With the exception of Gooley's algorithm, the CPU times are dominated by the number of evaluations and therefore similar. On a Dell Precision 650 with 3.06 GHz Xeon running Linux, 30 runs with 8000 evaluations per run take between 80 and 190 seconds (for more precise values, see (Barbulescu et al., 2004a)). For HBSS, we do not re-compute the ranks of the unscheduled tasks every time we choose the next request to be scheduled; therefore, the CPU times for HBSS are similar to the CPU times for the other algorithms.

When minimizing conflicts, many of the algorithms find solutions with the best known values. *Genitor* and local search using a randomized order perform best. SWO also performs well, with the exception of R1 and R3; however, some adjusting of the parameters used to run SWO may fix this problem. It is in fact surprising how well SWO performs when minimizing the conflicts, given that we chose a very simple implementation, where all the tasks in conflict are moved forward with a fixed distance. The performance of local search with a biased order is poor. HBSS performs well for the *A* problems; however, it fails to find the best known values for R1, R2 and R3. The original solution to the problem, Gooley's, only computes a single solution; its results can be improved by a sampling variant (see Section 6.3.1).

When minimizing overlaps, *Genitor* and SWO perform equally well for the *A* problems and R5; SWO produces the best results for the remaining data. The performance of local search with a biased order is relatively poor: worse mins/means and higher variance. However, the local search using a randomized order performs as well as SWO, and it finds two better solutions for R3 and R4. If SWO is allowed up to 50,000 evaluations, it does not find better solutions, unlike *Genitor* which continues to improve the solution quality. HBSS finds best known solutions only for A3, A4, A6, and R5; however, even for these problems, the means are worse and standard deviations higher than for local search with a randomized order, *Genitor* and SWO. For comparison, we computed the overlaps corresponding to the schedules built using Gooley's algorithm and present them in the last column of Table 4; however, Gooley's algorithm was not designed to minimize overlaps.

6. Factors That Influence Algorithm Performance

When implementing a solution for a new optimization application, one is confronted with a panoply of search algorithms and heuristics. The question for the practitioner is which to use; the question for the researcher and skeptical practitioner is why a particular algorithm.

In the set of experiments that follow, we set out to understand why algorithms exhibited the performance we observed. As we tried out different algorithms and heuristics, we discovered, as have many others, that sometimes the first, second or even third attempts to use a particular algorithm and heuristic produced mediocre performance. Because of that, we will only report results on algorithms that exhibit reasonable performance. Even so, that range of algorithms is sufficient to support several hypotheses about the nature of the AFSCN problem and its relationship with algorithm performance.

Day	*Genitor*			Biased Neigh. LS			Random Neigh. LS			SWO			HBSS			Gooley
	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	
A1	**8**	8.6	0.49	15	18.16	2.54	**8**	8.7	0.46	**8**	8	0.0	**8**	9.76	0.46	11
A2	**4**	4	0	6	10.96	2.04	**4**	4.0	0	**4**	4	0.0	**4**	4.64	0.66	7
A3	**3**	3.03	0.18	11	15.4	2.73	**3**	3.1	0.3	**3**	3	0.0	**3**	3.37	0.54	5
A4	**2**	2.06	0.25	12	17.43	2.76	**2**	2.2	0.48	**2**	2.06	0.25	**2**	3.09	0.43	4
A5	**4**	4.1	0.3	12	16.16	1.78	**4**	4.7	0.46	**4**	4	0.0	**4**	4.27	0.45	5
A6	**6**	6.03	0.18	15	18.16	2.05	**6**	6.16	0.37	**6**	6	0.0	**6**	6.39	0.49	7
A7	**6**	6	0	10	14.1	2.53	**6**	6.06	0.25	**6**	6	0.0	**6**	7.35	0.54	**6**
R1	**42**	43.7	0.98	68	75.3	4.9	**42**	44.0	1.25	43	43.3	0.46	45	48.44	1.15	45
R2	**29**	29.3	0.46	49	56.06	3.83	**29**	29.8	0.71	**29**	29.96	0.18	32	35.16	1.27	36
R3	**17**	17.63	0.49	34	38.63	3.74	**17**	18.0	0.69	18	18	0.0	19	21.08	0.89	20
R4	**28**	28.03	0.18	41	48.5	3.59	**28**	28.36	0.66	**28**	28.3	0.46	**28**	31.22	1.10	29
R5	**12**	12.03	0.18	15	17.56	1.3	**12**	12.4	0.56	**12**	12	0	**12**	12.36	0.55	13

Table 3: Performance of *Genitor*, local search, SWO, HBSS and Gooley's algorithm in terms of the best and mean number of conflicts. Statistics for *Genitor*, local search and SWO are taken over 30 independent runs, with 8000 evaluations per run. For HBSS, 240,000 samples are considered. Min numbers in boldface indicate best known values.

Day	*Genitor*			Biased Neigh. LS			Random Neigh. LS			SWO			HBSS			Gooley
	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	
A1	**104**	106.9	0.6	255	375.0	54.4	**104**	106.76	1.81	**104**	104	0.0	128	158.7	28.7	687
A2	**13**	13	0.0	65	174.6	50.6	**13**	13.66	2.59	**13**	13.4	2.0	43	70.1	31.1	535
A3	**28**	28.4	1.2	144	252.0	52.9	**28**	30.7	4.31	**28**	28.1	0.6	**28**	52.5	16.9	217
A4	**9**	9.2	0.7	153	239.6	55.4	**9**	10.16	2.39	**9**	13.3	7.8	**9**	45.7	13.0	216
A5	**30**	30.4	0.5	142	220.1	59.3	**30**	30.83	1.36	**30**	30	0.0	50	82.6	13.2	231
A6	**45**	45.1	0.4	190	277.4	46.7	**45**	45.13	0.5	**45**	45.1	0.3	**45**	65.5	16.8	152
A7	**46**	46.1	0.6	137	219.6	40.4	**46**	49.96	5.95	**46**	46	0.0	83	126.4	12.5	260
R1	913	987.8	40.8	1559	1830.9	143.4	798	848.66	38.42	798	841.4	14.0	1105	1242.6	42.1	1713
R2	519	540.7	13.3	1078	1235.5	92.8	494	521.9	20.28	491	503.8	6.5	598	681.8	27.0	1047
R3	275	292.3	10.9	788	967.7	96.7	**250**	327.53	55.34	265	270.1	2.8	416	571.0	46.0	899
R4	738	755.4	10.3	1139	1287.1	84.2	**725**	755.46	25.42	731	736.2	3.0	827	978.4	28.7	1288
R5	**146**	146.5	1.9	351	457.9	69.1	**146**	147.1	2.85	**146**	146.0	0.0	**146**	164.4	10.8	198

Table 4: Performance of *Genitor*, local search, SWO, HBSS and Gooley's algorithm in terms of the best and mean sum of overlaps. All statistics are taken over 30 independent runs, with 8000 evaluations per run. For HBSS, 240,000 samples are considered. Min numbers in boldface indicate best known values.

6.1 Redundancy of the Search Space

The AFSCN search space is dominated by plateaus for three reasons. First, the schedule builder operates in a greedy fashion, scheduling a request on the first available resource from the list of possible alternatives. For example, consider a permutation of $n-1$ from the total of n requests. If the last request X is inserted in the first position in the permutation and the schedule builder is applied, a schedule S is obtained. We now scan the permutation of $n-1$ requests from left to right, successively inserting X in the second position, then the third and so on, building the corresponding schedule. As long as none of the requests appearing before X in the permutation require the particular spot occupied by X in S as their first feasible alternative to be scheduled, the same schedule S will be obtained. This happens for two reasons: 1) the requests are inserted in the schedule in the order in which they appear in the permutation and 2) the greedy schedule builder considers the possible alternatives in the order in which they are specified and accepts the first alternative for which the request can be scheduled. Let $k+1$ be the first position to insert X that will alter S; this means that the first feasible alternative to schedule the request in position k overlaps with the spot occupied by X in S. When X is inserted in position $k+1$, a new schedule $S1$ is obtained; the same schedule $S1$ will be built by inserting X in subsequent positions, until encountering a request for which its first feasible alternative overlaps with the spot occupied by X in $S1$, etc. In effect, we are shifting X in all possible positions in the permutation; when performing a shift operation, we are changing the position of a request relative to others. This example also shows that shifting in a permutation might not change the corresponding schedule.

A second reason for the plateaus in the search space is the presence of time windows. If a request X needs to be scheduled sometime at the end of the day, even if it appears in the beginning of the permutation, it will still occupy a spot in the schedule towards the end (assuming it can be scheduled) and therefore, after most of the other requests (which appeared after X in the permutation).

A third reason is the discretization of the objective function. Clearly, the range of conflicts is a small number of discrete values (with a weak upper bound of the number of tasks). The range for overlaps is still discrete but is larger than for conflicts. Using overlaps, approximately 20 times more unique values than for conflicts can be encountered during search. The effect of the discretization can be seen in the differing results using the two objective functions. Thus, one reason for including both in our studies is to show some of the effects of the discretization.

In this section, we explore the effects of plateaus on search. We start by examining pairwise interactions between requests. We examine the possibility of reducing the size of the neighborhood by either exploring knowledge about pairwise task interactions or by restricting possible positions for a task in the permutation based on its time windows.

6.1.1 Are Interactions Between Requests Mostly Pairwise?

The first reason for plateaus and the example in Figure 2 suggests that the availability of alternative resources may lead to n-way interactions between requests. Such interactions may prove difficult for algorithms that exploit local gradients and make pairwise changes because single pairwise changes may produce no discernible effect.

We tested whether this is the case by perturbing solutions (permutations) in all possible pairwise changes and converting the resulting permutation into a schedule. We noted how many resulting schedules are identical to the one obtained from the original permutation. We defined two schedules to be identical if each request is scheduled on the same resource, starting at the same time in both schedules; when minimizing overlaps, the overlapping requests also need to be scheduled on the same resource and starting at the same time in both schedules.

Day	Total	Minimizing Conflicts				Minimizing Overlaps			
		Random Perms		Optimal Perms		Random Perms		Optimal Perms	
	Pairs	Mean	Avg %	Mean	Avg %	Mean	Avg %	Mean	Avg %
A1	51681	21999.7	42.5	22911.3	44.3	20767.2	40.2	22925.8	44.3
A2	45451	20659.7	45.4	21194.6	46.6	19684.3	43.3	21169.7	46.6
A3	48205	20652.3	42.8	20655.6	42.8	19859.6	41.2	20373.5	42.3
A4	50403	20632.3	40.9	21240.4	42.1	19491.7	38.7	20729.1	41.1
A5	46360	18850.7	40.6	19019.8	41.0	18371.3	39.6	19238.9	41.5
A6	44551	17882.3	40.1	17804.4	39.9	16771.7	37.6	17811.7	39.9
A7	43956	19278.3	43.8	20001.6	45.5	18406.9	41.9	20220.7	46.0
R1	116403	44001.2	37.8	45068.8	38.7	39380.1	33.8	41285.6	35.5
R2	104196	38246.5	36.7	38775	37.2	34362.7	32.9	37805.9	36.2
R3	90525	35779.9	39.5	36464.3	40.2	33470.8	36.9	33894.3	37.4
R4	92665	36399.7	39.2	38032	41.0	34147.1	36.8	36816.9	39.7
R5	87571	35590.5	40.6	36368.3	41.5	34180.8	39.0	35891.4	40.9

Table 5: Statistics for the number of pairs of non-interacting requests over 30 random and optimal permutations, for both objective functions

We explored pairwise interactions between requests in random as well as best known solutions. The statistics were computed over 30 permutations for each problem. For each permutation and each pair of requests (A, B), we built the schedules corresponding to two shifting moves: A in the position of B and B in the position of A (all shifting moves start from the same initial permutation). We counted for how many pairs (A, B) in a permutation the corresponding schedule is identical when shifting either A in the position of B or B in the position of A; we call such pairs of requests "non-interacting requests" for that particular schedule. We computed the average number of pairs of non-interacting requests over the 30 permutations.

The results of the pairwise sensitivity test are summarized in Table 5. The second column *Total Pairs* represents the total number of request pairs in the permutation (for n requests, the number in this column is $n(n-1)/2$). The mean of the number of non-interacting pairs of requests (from 30 random or optimal permutations) appears in the columns denoted *Mean.* To normalize the results across the various problem sizes, we also report the average percentage of non-interacting request pairs in a permutation (*Average Percentage*). The results show that: 1) More than 36% of the pairs do not interact (make no difference to the schedule) when minimizing conflicts. When minimizing overlaps, more than 33% of the pairs do not interact. 2) Best known solutions have slightly more non-interacting pairs than do random solutions, except for A6 when minimizing conflicts. 3) In general, minimizing the number of conflicts results in more non-interacting pairs than minimizing the overlaps.

The fact that the schedule is more sensitive to shifting when minimizing overlaps is not surprising. Consider for example the case of a permutation $ABC...X...Y...$ where X and Y compete for the same resource and cannot be scheduled. Suppose that shifting X before Y and Y before X results in the same schedule when minimizing conflicts. When minimizing the number of overlaps, the schedule includes overlapping requests. While shifting X before Y might still leave the schedule unchanged, shifting Y before X is likely to change the schedule. Indeed, in the initial schedule, the greedy schedule builder first encounters X and places it into the partial schedule such that the overlaps are minimized. Since X and Y compete for the same resource, the placement of Y will then be influenced by the fact that X is already in the schedule. When Y is encountered first by the schedule builder, the overlaps will be minimized with respect to Y first; this is likely to result in a different placement of X and Y than in the initial schedule. In general, if the schedule builder translates two permutations into exactly the same schedule when minimizing conflicts, minimizing overlaps may result in two different schedules, because the placement of the overlapping requests may differ.

As a second aspect of task interaction, we observed that for certain requests, no matter where they are shifted in the permutation, the corresponding schedule does not change. If there exist such requests that "do not matter", we hypothesized that we could eliminate them from the permutation and reduce the size of the search space. However, further examination of examples shows that a request *seems* not to interact (when shifted) with any of the other requests because of the particular order of the rest of the requests in the permutation; we need to consider more than pairwise interactions. Consider a simple example where two requests A and B are both scheduled on the same resource, such that when any one of the two requests are scheduled, a third request C (which also specifies the same resource) cannot be scheduled on that resource. A and B may not themselves overlap in time, but C may be long enough to overlap with both of them. If the requests appear in the permutation in the order A, B, C or B, A, C (not necessarily in consecutive positions), it might appear that both A and B "do not matter" (each can be shifted anywhere in the permutation without a change in the schedule). When A is shifted, B still appears before C to prevent it from being scheduled, and similarly, when B is shifted, A still prevents C from being scheduled. However, if, for example, C appears in the initial permutation after A and before B, shifting A after C will result in a different schedule (C can be scheduled since both A and B now appear after C in the permutation). While it seemed that A "did not matter" in the initial permutation, shifting A obviously can change the schedule given a different ordering of the requests in the permutation. This fact combined with the knowledge about the contention present in the problems suggests that we cannot reduce the search space size by simply eliminating certain requests or splitting the problem into smaller size separate problems.

6.1.2 Temporal Constraints Between Pairs of Tasks

A request for a time slot near the beginning of the day will always appear in the schedule before a request with a time window near the end of the day, regardless of permutation ordering (assuming both requests are scheduled). Thus, another data pattern is the set of relative orders between tasks, based on when they can appear in the final schedule. If we know that a request must appear within a particular time window, then moving it in the permutation past another request that must appear later should have no effect on the final schedule. To test this, we defined a reduced-size search neighborhood by restricting the movement of each request based on its time window. This implicitly resulted in a reduction of the size of the search space for local search and, if successful, should indicate whether relative orders might be learned.

The restricted shift operator starts with a permutation of the requests in increasing order of their earliest starting times. Iteratively, we randomly select a position *pos* in this permutation and shift the corresponding request to the right, in positions $pos+1$, $pos+2$,..., $pos+k$, where $pos+k+1$ is the first position after *pos* corresponding to a request with a non-intersecting, later time window. For each such shift, we evaluate the new permutation. If the best schedule corresponding to a shift is better than or as good as the current one, we accept its permutation. This restricted shift operator ensures that for any pairs of requests A and B such that their time windows do not intersect and the time window for A is earlier than the time window for B, A will always appear before B.

We summarize the results obtained by running local search using the restricted shift operator both for minimizing conflicts and for minimizing overlaps in Table 6. For each version of local search, we performed 15 trials, allocating 16000 evaluations for each trial. Experiments performed using 30 trials for each version of local search and 8000 evaluations per trial resulted in worse values. For each algorithm, we report the best value obtained (*Min*), the average value (*Mean*), and the standard deviation (*Stdev*) in 15 trials. The second column *Best Known* contains the best known solutions. The hill climber using a randomized neighborhood outperforms search using the restricted shift operator. In fact, we ran local search with the restricted shift operator using 30 trials, with a large number of evaluations (500,000 evaluations) per trial. Local search using the restricted shift operator did not find best known values for any of the problems.

We investigated the reason why local search using the restricted shift operator doesn't seem to be able to reach the best known solutions. While ideally we would like the restricted shift operator to only eliminate the redundant permutations from the search space, we found that there also exist schedules which cannot be reached using the restricted shift. Consider a simple example where four low altitude requests, A, B, C and D, need to be scheduled at the same tracking station, on one of two antennas. The time windows required by A, B, C and D are $(0,7)$, $(1,3)$, $(8,10)$ and $(5,9)$ respectively. Consider a schedule such that B and C are scheduled on the first antenna, D on the second, and A is not scheduled. For example, the $BCDA$ permutation would result in such a schedule. However, no permutation produced by the restricted shift corresponds to this schedule. Indeed, any such permutation should specify C before D (otherwise D would be scheduled on the first antenna) and A after B and D, so therefore A after C. However, the restricted shift prevents A from being moved after C (their time windows do not intersect and C starts later than A). While the schedule in the example above does a poor job in accommodating the requests, the fact that there exist such schedules (which cannot be reached using the restricted shift) means that the restricted shift operates in a reduced size schedule space as well. We conjecture that this is the reason why the restricted shift operator doesn't seem to be able to reach the best known solutions.

6.1.3 Estimating the Size of Plateaus in the Permutation Search Space

Local search spends most of the time traversing plateaus in the search space (by accepting non-improving moves). In this section, we study the average length of random walks on the

	Minimizing Conflicts				Minimizing Overlaps			
Day	Best Known	Min	Mean	Stdev	Best Known	Min	Mean	Stdev
A1	8	10	10.93	0.79	104	139	139.73	1.53
A2	4	5	5.26	0.45	13	20	31.47	14.63
A3	3	5	6.4	0.63	28	39	46.47	10.78
A4	2	9	10.4	0.73	9	49	65.73	16.86
A5	4	6	7.4	0.73	30	36	64	12.11
A6	6	8	8.46	0.63	45	47	59.66	13.31
A7	6	7	7.6	0.5	46	78	85.4	6.22
R1	42	46	49.8	2.04	774	932	990.6	42.39
R2	29	35	36.26	1.16	486	549	596.4	31.1
R3	17	21	22.13	1.06	250	324	474.93	72.99
R4	28	32	34.2	1.2	725	769	805.13	32.07
R5	12	15	16.66	1.04	146	178	200.06	15.32

Table 6: Results of running restricted local search in 15 experiments, by evaluating 16000 permutations per experiment.

plateaus encountered by local search. We show that as search progresses the random walks become longer before finding an improvement.

We showed in section 6.1.1 that approximately a third of all shifting pairs of requests result in schedules identical with the current solution. However, an even larger number of neighbors result in different schedules with the same value as the current solution. This means that most of the accepted moves during search are non-improving moves; search ends up randomly walking on a plateau until an exit is found. We collected results about the number of schedules with the same *value* as the original schedule, when perturbing the solutions in all possible pairwise changes. Note that these schedules include the ones identical with the current solution. The results are summarized in Table 7. Again, we report the average percentage of neighbors identical in value with the original permutation. The results show that: 1) More than 84% of the shifts result in schedules with the same value as the original one, when minimizing conflicts. When minimizing overlaps, more than 62% (usually around 70%) of the shifts result in same value schedules. 2) Best known solutions have slightly more same-value neighbors than do random solutions.

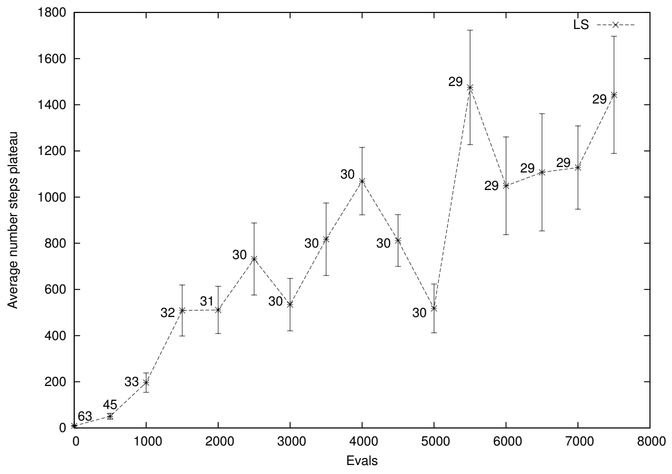
To evaluate the characteristics of the plateaus in the permutation search space, we performed the following experiment. We ran local search (using the randomized neighborhood version) with 8000 evaluations allowed; at every 500 evaluations we identified the current solution Crt. For each such Crt, we performed 100 iterations of local search starting from Crt and stopping as soon as a better solution or a maximum number of equally good solutions were encountered. For the A problems, best known solutions are often found early in the search; most of the 100 iterations of local search started from such a Crt would reach the maximum number of equally good solutions. Therefore, we chose a limit of 1000 steps on the plateau for the A problems and 8000 steps for the R problems. We averaged the number of equally good solutions encountered during the 100 trials of search performed for each Crt; this represents the average number of steps needed to find an exit from a

Day	Total	Minimizing Conflicts				Minimizing Overlaps			
		Random Perms		Optimal Perms		Random Perms		Optimal Perms	
	Neighbors	Mean	Avg %	Mean	Avg %	Mean	Avg %	Mean	Avg %
A1	103041	87581.1	84.9	91609.1	88.9	75877.4	73.6	88621.2	86.0
A2	90601	79189.3	87.4	83717.9	92.4	70440.9	77.7	81141.9	89.5
A3	96100	82937	86.8	84915.4	88.9	73073.3	76.5	82407.7	86.3
A4	100489	84759	84.3	87568.2	87.1	72767.7	72.4	85290	84.8
A5	92416	77952	84.3	82057.4	88.7	67649.3	73.2	79735.9	86.2
A6	88804	74671.5	84.0	78730.3	88.6	63667.4	71.6	75737.9	85.2
A7	87616	76489.6	87.3	79756.5	91.0	67839	77.4	77584.3	88.5
R1	232324	189566	81.5	190736	82.0	145514	62.6	160489	69.0
R2	207936	173434	83.4	177264	85.2	137568	66.1	160350	77.1
R3	180625	153207	84.8	156413	86.5	126511	70.0	139012	76.9
R4	184900	157459	85.1	162996	88.1	130684	70.6	145953	78.9
R5	174724	154347	88.3	159581	91.3	133672	76.5	152629	87.3

Table 7: Statistics for the number of neighbors resulting in schedules of the same value as the original, over 30 random and optimal permutations, for both objective functions

plateau. In Figure 5 we display the results obtained for R4; similar behavior was observed for the rest of the problems. Note that we used a *log* scale on the y axis for the graph corresponding to minimizing overlaps: most of the 100 walks performed from the current solution of value 729 end up taking the maximum number of steps allowed (8000) without finding an exit from the plateau. The results show that large plateaus are present in the search space; improving moves lead to longer walks on lower plateaus.

For the AFSCN scheduling problems, most of the states on a plateau have at least one neighbor that has a better value (this neighbor represents an exit). However, the number of such exits represents a very small percentage from the total number of neighbors and therefore local search has a very small probability of finding an exit. Using the terminology introduced by Frank (Frank, Cheeseman, & Stutz, 1997), most of the plateaus encountered by search in the AFSCN domain would be classified as benches, meaning that exits to states at lower levels are present. If there are no exits from a plateau, the plateau is a local minimum. Determining which of the plateaus are local minima (by enumerating all the states on the plateau and their neighbors) is prohibitive because of the large size of the neighborhoods and the large number of equally good neighbors present for each state in the search space. Instead, we focus on the average length of the random walk on a plateau as a factor in local search performance. The length of the random walk on the plateau depends on two factors: the size of the plateau and the number of exits from the plateau. Preliminary investigations show that the number of improving neighbors for a solution decreases as the solution becomes better - therefore we conjecture that there are more exits from higher level plateaus than from the lower level ones. This would account for the trend of needing more steps to find an exit when moving to lower plateaus (corresponding to better solutions). It is also possible that the plateaus corresponding to better solutions are larger in size; however,



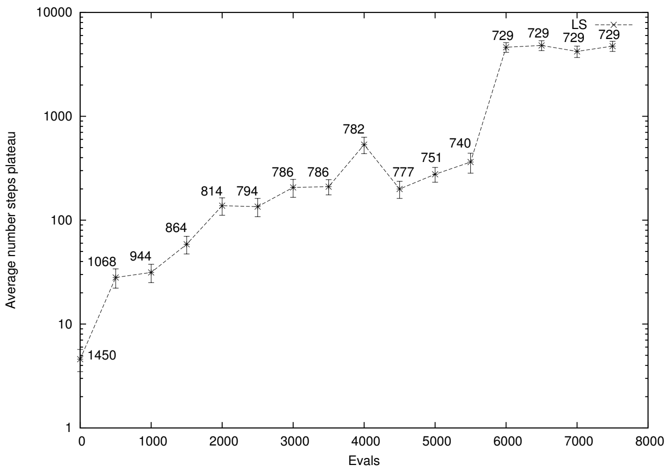


Figure 5: Average length of the random walk on plateaus when minimizing conflicts (left) or overlaps (right) for a single local search run on R4. The labels on the graphs represent the value of the current solution. Note the *log* scale on the y axis for the graph corresponding to minimizing overlaps. The best known value for this problem is 28 when minimizing conflicts and 725 when minimizing overlaps.

enumerating all the states on a plateau for the AFSCN domain is impractical (following a technique developed by Frank (Frank et al., 1997), just the first iteration of breadth first search would result in approximately $0.8 * (n-1)^2$ states on the same plateau).

6.2 The Order of the Neighbors

Our original implementation of local search introduced an ordering bias into the neighborhood checking. Its poor performance led us to conclude that the plateaus likely precluded effective local search (Barbulescu et al., 2004c). In this section, we show that by randomizing the neighborhood the performance of local search can be made competitive, and we offer an explanation of why the ordering bias was a particular problem.

Our initial implementation of the shifting search neighborhood iteratively chose by random the position x of the request to be shifted and then evaluated the neighbors produced by shifting in *all* possible positions (0, 1, 2,...,$n-1$) and accepting the new solutions if better or equally good. This turns out to be particularly inefficient; shifting a request into consecutive positions is likely to result in identical schedules (as exemplified in the beginning of the section). If shifting a request results in a neighbor that is worse than the current value of the solution, shifting the same request in subsequent positions will also be worse (until a position is found that changes the corresponding schedule). In effect, for our initial implementation, when local search encounters a worse neighbor, the probability of the next neighbor being also worse increases. The chains of worse neighbors obtained by shifting a request in consecutive positions can be quite long. For some requests, shifting them around the current position results in an identical schedule (equally good); once they are shifted

	Average % Worse		Average % Equal				Average % Better	
			Identical sch.		Non-Identical sch.			
Day	Rand	Bias	Rand	Bias	Rand	Bias	Rand	Bias
A1	16	83.6	0.02	0.05	83.5	16	0.33	0.2
A2	14.4	82.2	0.02	0.04	85.2	17.5	0.2	0.1
A3	16.8	82.6	0.02	0.03	82.7	17.1	0.3	0.1
A4	19.3	84.8	0.02	0.02	80.2	14.8	0.3	0.1
A5	17.4	83.6	0.02	0.05	82.2	16.0	0.3	0.1
A6	18.0	84.3	0.01	0.04	81.5	15.4	0.3	0.1
A7	14.3	82.4	0.02	0.1	85.3	17.2	0.2	0.1
R1	21.9	87.2	0.01	0.03	77.4	12.5	0.6	0.2
R2	19.6	85.2	0.01	0.01	79.8	14.5	0.5	0.1
R3	17.8	85.2	0.02	0.06	81.6	14.4	0.4	0.2
R4	16.7	84.4	0.01	0.01	82.7	15.3	0.4	0.1
R5	13.6	83.0	0.02	0.1	85.9	16.6	0.3	0.1

Table 8: Minimizing conflicts: Average percentage of evaluations (out of 8000) resulting in worse, equally good or improving solutions over 30 runs of local search with the randomized neighborhood.

far enough (and this distance could be only 10-20 positions) from the original position, all the rest of the schedules are worse.

To better understand why the randomized neighborhood results in such a great improvement in local search performance, we counted for each run how many of the evaluations resulted in worse solutions, equally good solutions (counting separately here the identical schedules) and better solutions. We collected the counts for 30 runs of local search, with 8000 evaluations per run (these runs are identical to the ones producing the results in Tables 3 and 4). We did the same thing for local search using the biased neighborhood. The results are summarized in Tables 8 and 9. As expected, the number of worse neighbors evaluated by local search is significantly higher for the biased version of search. Given the high percentage of equally valued neighbors (see Section 6.1.3), both versions of search spend most of the time accepting equally good moves; only a small number of improving steps are taken. However, since the biased version of search spends more than 80% of the time evaluating worse moves, it is left with only approximately 15% of the evaluations (or even less than that when minimizing overlaps) to move through the space. The biased version of search needs more evaluations to find good solutions. This is also emphasized by the number of improving neighbors found by the two versions of local search: the randomized version always finds more improving neighbors (twice as many as the biased version, or even more).

6.3 Patterns of Request Ordering or What Genitor Might Be Learning

We hypothesize that *Genitor* performs well because it discovers interactions between the requests. The first part of the section explores whether Genitor is learning a previously discovered heuristic: schedule low altitude (more constrained) requests first. We show that

	Average % Worse		Average % Equal				Average % Better	
			Identical sch.		Non-Identical sch.			
Day	Rand	Bias	Rand	Bias	Rand	Bias	Rand	Bias
A1	20.1	87.8	0.03	0.16	79.1	11.7	0.6	0.3
A2	16.6	86.0	0.02	0.04	82.8	13.6	0.5	0.2
A3	19.3	86.8	0.02	0.08	80.0	12.7	0.6	0.3
A4	21.3	88.5	0.03	0.02	77.8	11.0	0.7	0.3
A5	19.5	87.3	0.03	0.05	79.8	12.2	0.6	0.3
A6	21.3	87.5	0.01	0.07	77.9	11.9	0.6	0.3
A7	16.8	85.7	0.02	0.1	82.6	13.8	0.5	0.3
R1	32.2	91.0	0.01	0.03	65.8	8.5	1.8	0.4
R2	26.8	90.4	0.02	0.04	71.7	9.1	1.3	0.3
R3	24.7	89.9	0.03	0.09	74.0	9.6	1.1	0.3
R4	23.6	89.5	0.02	0.02	75.2	10.0	1.0	0.3
R5	17.9	87.8	0.02	0.1	81.3	11.6	0.7	0.3

Table 9: Minimizing overlaps: Average percentage of evaluations (out of 8000) resulting in worse, equally good or improving solutions over 30 runs of local search with the randomized neighborhood.

such an approach can produce good solutions for AFSCN scheduling. In the second part of this section, we examine sets of permutations that correspond to schedules with the best known values and identify chains of common request orderings in these permutations (similar in spirit to the notion of backbone in MAXSAT (Singer, Gent, & Smaill, 2000)). The presence of such chains would support the hypothesis that *Genitor* is discovering patterns of request orderings.

6.3.1 Identifying Domain Knowledge

One of the particular characteristics of the AFSCN scheduling problem is the presence of two categories of requests. The low altitude requests have fixed start times and specify only one to three alternative resources. The high altitude requests implicitly specify multiple possible start times (because their corresponding time windows are usually longer than the duration that needs to be scheduled) and up to fifteen possible alternative resources. Clearly the low altitude requests are more constrained. To exploit this, we implemented a heuristic that schedules the low altitude requests before the high altitude ones; we call this heuristic the "split heuristic". We incorporated the split heuristic in the schedule builder: given a permutation of requests, the new schedule builder first schedules only the low altitude requests, in the order in which they appear in the permutation. Without modifying the position of the low altitude requests in the schedule, the high altitude requests are then inserted in the schedule, again in the order in which they appear in the permutation. The idea of scheduling low altitude requests before high altitude requests was the basis of Gooley's heuristic (Gooley, 1993). Also, the split heuristic is similar to the contention measures defined in (Frank et al., 2001).

Day	Best Known	Random Sampling-S		
		Min	Mean	Stdev
A1	8	8	8.2	0.41
A2	4	4	4	0
A3	3	3	3.3	0.46
A4	2	2	2.43	0.51
A5	4	4	4.66	0.48
A6	6	6	6.5	0.51
A7	6	6	6	0

Table 10: Results of running random sampling with the split heuristic in 30 experiments, by generating 100 random permutations per experiment for minimizing conflicts.

Day	Best Known	Genitor-S		
		Min	Mean	Stdev
R1	42	42	42	0
R2	29	30	30	0
R3	17	18	18	0
R4	28	28	28	0
R5	12	12	12	0

Table 11: Minimizing conflicts: results of running *Genitor* with the split heuristic in 30 experiments, with 8000 evaluations per experiment.

The results we obtained using the split heuristic are somewhat surprising: when minimizing conflicts, best known valued schedules can be obtained quickly for the A problems by simply sampling a small number of random solutions. The results obtained by sampling 100 random permutations are shown in Table 10.

Given the good results produced by the split heuristic on the A problems (for minimizing conflicts), one of our first hypotheses about why *Genitor* did well was that it was learning to schedule the low altitude requests before the high altitude requests. To test this hypothesis, we designed an experiment comparing schedules produced by the original schedule builder with the heuristic version of it for permutations derived from *Genitor*[7]. We found that for more than 90% of the best known schedules found by *Genitor* for the A problems, the split heuristic does not increase the number of conflicts in the schedule (showing that either the low altitude requests were first in the permutation or they did not conflict with earlier requests). However, for the R problems, the split heuristic does not always find the best schedules. Table 11 shows the results of using the split heuristic with *Genitor* on the R problems. In fact, we can show that scheduling all the low altitude requests before high altitude requests may prevent finding the optimal solutions.

7. An earlier paper (Barbulescu et al., 2004b) covers this experiment in more detail.

We performed a similar empirical study for the objective function of minimizing the sum of overlaps to determine if the performance of the split heuristic transfers to minimizing overlaps. We ran 1000 trials of *Genitor* with 8000 evaluations per trial. We selected the solutions corresponding to best known values for the *A* problems. For the *R* problems, the solution values obtained in 1000 trials are widely spread (for example, for 03/07/02 the values vary from 880 to 1084); therefore, we selected the best solution values that occurred at least 30 times out of the 1000 runs. For each of the selected solutions, we applied the heuristic schedule builder to its corresponding permutation; we then compared the values of the two schedules. The results are presented in Table 12. For A1, there were only two solutions corresponding to the best known value (104) and 922 solutions of value 107. For R1, the solution values degrade too much if we impose the rule of finding at least 30 solutions of the same value, therefore we chose the solution value 947 which occurred 15 times. With the exception of A3, A4, A6 and R5, all the solutions found translate into worse schedules when the split heuristic is applied. We then ran *Genitor* using the split heuristic. The results are shown in Table 13. With the exception of A3, A4 and A6, *Genitor* using the split heuristic fails to find best known solutions for the *A* problems. For the *R* problems, using the split heuristic in fact improves the results obtained by *Genitor* for R1 and R2 and finds the best known solution for R5.

The results in Tables 12 and 13 suggest that *Genitor* might be learning to schedule low altitude before high altitude requests for A3, A4, A6 and R5. However, the low-before-high pattern does not account for the performance of *Genitor* for the rest of the problems. This is not a surprising result. When using the split heuristic while minimizing the sum of overlaps, many of the requests that cannot be accommodated in the schedule and overlap with other requests already scheduled are likely to be high altitude requests (since the schedule builder schedules the low altitude requests first). The high altitude requests tend to be longer, therefore the overlap corresponding to them will also be larger than the overlap corresponding to low altitude requests. This could result in a suboptimal schedule. When minimizing the sum of overlaps, the long requests are likely to be scheduled such that the overlaps are computed for shorter requests which compete with them for the same resources. This in fact means that at least some of the long (high altitude) requests will have to be scheduled before shorter (low altitude) requests.

6.3.2 Common Request Orderings

As an alternative explanation, we hypothesized that *Genitor* is discovering patterns of request ordering: certain requests that must come before other requests. To test this, we identify common request orderings present in solutions obtained from multiple runs of *Genitor*. We ran 1000 trials of *Genitor* and selected the solutions corresponding to best known values. First, we checked for request orderings of the form "requestA before requestB" which appear in all the permutations corresponding to best known solutions (for the *A* problems) and corresponding to good solutions (for the *R* problems). The results are summarized in Table 14. The *Sol. Value* columns show the value of the solutions chosen for the analysis (out of 1000 solutions). The number of solutions (out of 1000) corresponding to the chosen value is shown in the *# of Solutions* columns. When analyzing the common pairs of request orderings for minimizing the number of conflicts, we observed that most

Day	Solution Value	Total Number of Solutions Found	Same Evaluation	Worse Evaluation
A1	107	922	0	922
A2	13	959	0	959
A3	28	833	299	534
A4	9	912	612	300
A5	30	646	0	646
A6	45	817	771	46
A7	46	891	0	891
R1	947	15	0	15
R2	530	30	0	30
R3	285	37	0	37
R4	744	31	0	31
R5	146	722	494	228

Table 12: Minimizing sum of overlaps: The effect of applying the split heuristic when evaluating good schedules produced by *Genitor*.

	Best	Genitor-S		
Day	Known	Min	Mean	Stdev
A1	104	119	119	0.0
A2	13	43	43	0.0
A3	28	28	28	0.0
A4	9	9	9	0.0
A5	30	50	50	0.0
A6	45	45	45	0.0
A7	46	69	69	0.0
R1	774	907	924.33	6.01
R2	486	513	516.63	5.03
R3	250	276	276.03	0.18
R4	725	752	752.03	0.0
R5	146	146	146	0.0

Table 13: Minimizing sum of overlaps: results of running *Genitor* with the split heuristic in 30 experiments, with 8000 evaluations per experiment.

pairs specified a low altitude request appearing before a high altitude one. Therefore, we separate the pairs into two categories: pairs specifying a low altitude request before a high altitude requests (column: *(Low,High) Pair Count*) and the rest (column: *Other Pairs*). For the *A* problems, the results clearly show that most common pairs of ordering requests specify a low altitude request before a high altitude request. For the *R* problems, more "Other pairs" can be observed. In part, this might be due to the small number of solutions

	Minimizing Conflicts				Minimizing Overlaps			
Day	Sol. Value	# of Solutions	(Low,High) Pair Count	Other Pairs	Sol. Value	# of Solutions	(Low,High) Pair Count	Other Pairs
A1	8	420	77	1	107	922	78	7
A2	4	1000	29	1	13	959	50	3
A3	3	936	86	1	28	833	72	10
A4	2	937	132	3	9	912	117	5
A5	4	862	45	9	30	646	48	17
A6	6	967	101	10	45	817	124	10
A7	6	1000	43	3	46	891	57	11
R1	43	25	2166	149	947	15	2815	1222
R2	29	573	64	5	530	30	1597	308
R3	17	470	78	21	285	37	1185	400
R4	28	974	54	16	744	31	1240	347
R5	12	892	57	10	146	722	109	11

Table 14: Common pairs of request orderings found in permutations corresponding to best known/good *Genitor* solutions for both objective functions.

corresponding to the same value (only 15 out of 1000 for R1 when minimizing overlaps). The small number of solutions corresponding to the same value is also the reason for the big pair counts reported when minimizing overlaps for the *R* problems.

The permutations used to identify common pairs of request orderings are exactly the same permutations used in 6.3.1 to investigate if *Genitor* is learning to schedule the low altitude requests before the high altitude requests. We know that for the *A* problems the split heuristic results in good solutions; moreover, for exactly the same permutations, reordering the low altitude before the high altitude requests does not change the value of the schedule for more than 90% of the permutations (for details on the conflicts results, see (Barbulescu et al., 2004b)). Therefore, the results in Table 14 are somewhat surprising. We expected to see more low-before-high common pairs of requests for the *A* problems when minimizing the number of conflicts; instead, the pair counts are similar for the two objective functions. *Genitor* seems to discover patterns of request interaction, and most of them specify a low altitude request before a high altitude request. However, it is not always the case that scheduling **all** low altitude requests before the high altitude requests results in good schedules; in fact, when minimizing overlaps, such a heuristic worsens the performance of *Genitor* for most problems.

We also hypothesized that longer chains of common request orderings would be found in permutations producing solutions with identical requests bumped. Instead of examining all permutations corresponding to best known values for a problem, we partitioned these permutations in sets such that all permutations in a set result in exactly the same requests being bumped. We computed again the chains of common request orderings for each set. For sets containing a small number of permutations (less than five), we were able to identify longer chains of requests. However, for large size sets of permutations (resulting in the same conflicts), we only found chains of length two. Examination of these chains shows that

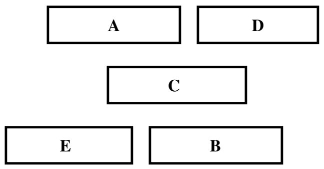


Figure 6: A, B, C, D and E compete for two antennas at the same tracking station.

sometimes the requests that caused a certain bump are missing from the common chains of requests appearing before the bump. This happens because given a group of requests, multiple orderings of the requests result in exactly the same conflicts. Consider the example in figure 6, and suppose that all requests can be scheduled on one of the two antennas present at a tracking station. Then the sequences (A, B, C, D, E), (D, B, C, A, E), (E, A, C, B, D) result in C being bumped; however, there are no common orderings of requests preceding C, nor any common ordering of the requests in the three sequences.

While *Genitor* does seem to discover patterns of request ordering, multiple different patterns of request orderings can result in the same conflicts (or even the same schedule). We could think of these patterns as building blocks. *Genitor* learns to identify good building blocks (orderings of requests resulting in good partial solutions) and propagates them into the final population (and the final solution). Such patterns are essential in building a good solution. However, the patterns are not ubiquitous (not all of them are necessary) and, therefore, attempts to identify them in solutions across different solutions produced by *Genitor* might fail.

6.4 The Role of Initialization

While *Genitor* normally starts with a population of random permutations, SWO is initialized with a greedy solution. The greedy solution can be quite good. The greedy solution translated into best known values for four problems (A2, A3, A5, A6 and R5) when minimizing the number of conflicts and for two problems (A6 and R5) when minimizing overlaps. In this section, we investigate the contribution of starting from better solutions.

We seeded the initial population of *Genitor* (size 200) with the greedy initial permutation built for SWO and 199 variations of this permutation, obtained by randomly swapping 20 pairs of low altitude requests; we called the new algorithm *SeededGenitor*.

We compared the results of random initializations (Tables 3 and 4) to those of running *SeededGenitor*. The results are shown in Table 15. When minimizing the number of conflicts, *Genitor* produces best known solutions even when started from random initial populations; initializing the *Genitor* population speeds up the progress of the algorithm toward the solution (see Section 6.6). When minimizing the sum of overlaps, initializing the *Genitor* population results in major improvements on the results obtained for the new days of data (in fact, except for R1 all the results are equal to the best known results). Also, *SeededGenitor* finds equal or better solutions than does SWO.

We repeated the empirical study identifying patterns of request orderings (from Section 6.3.2) for the solutions produced by *SeededGenitor*. Surprisingly, while the number of chains of length three increases and some chains of length four and five also appear, the chains of length two are still predominant. Most of these chains again specify a low altitude

	Minimizing conflicts			Minimizing overlaps		
Day	Min	Mean	Stdev	Min	Mean	Stdev
A1	8	8	0.0	104	107.9	3.0
A2	4	4	0.0	13	13	0.0
A3	3	3	0.0	28	28	0.0
A4	2	2	0.0	9	9	0.0
A5	4	4	0.0	30	30	0.0
A6	6	6	0.0	45	45	0.0
A7	6	6	0.0	46	46	0.0
R1	42	42	0.0	794	828.3	19.3
R2	29	29.03	0.18	486	494.5	7.7
R3	17	17.13	0.34	250	257	5.9
R4	28	28	0.0	725	730.6	6.5
R5	12	12	0.0	146	146	0.0

Table 15: Performance of Genitor when initialized from greedy permutations. All statistics are taken over 30 independent runs, with 8000 evaluations per run.

request before a high altitude request. However, the number of such common orderings is significantly higher than for *Genitor* initialized with random populations.

Initializing search from greedy permutations does significantly improve the performance of *Genitor*. But how important is the initial greedy permutation for SWO? To answer this question, we replaced the initial greedy permutation (and its variations in subsequent iterations of SWO) with random permutations and then used the SWO mechanism to iteratively move forward the requests in conflict. We call this version of SWO *RandomStartSWO*. We compared the results produced by *RandomStartSWO* with results from SWO to assess the effects of the initial greedy solution. We also compared the *RandomStartSWO* results with the ones from local search and *Genitor* (since both these algorithms are initialized with random permutations); this comparison enables us to assess the effects of the SWO mechanism of moving forward requests in conflict[8]. The results produced by *RandomStartSWO* are presented in Table 16. With the exception of R2, when minimizing the number of conflicts, best known values are obtained by *RandomStartSWO* for all the problems. In fact, for R1 and R3, the best results obtained are slightly better than the best found by SWO. When minimizing the sum of overlaps, best known values are obtained for the A problems; only for the R problems, the performance of SWO worsens when it is initialized with a random permutation. However, *RandomStartSWO* still performs better or as well as *Genitor* (with the exception of R2 when minimizing the number of conflicts and R5 for overlaps) for both objective functions.

8. In strict SWO terms, this mechanism is called prioritization. However, we choose not to use this term when initializing with a random permutation, since the initial variable ordering is not the result of associating some kind of priorities to the variables.

Day	Minimizing Conflicts				Minimizing Overlaps			
	Best Known	Min	Mean	Stdev	Best Known	Min	Mean	Stdev
A1	8	8	8.0	0.0	104	104	104.46	0.68
A2	4	4	4.0	0.0	13	13	13.83	1.89
A3	3	3	3.16	0.46	28	28	30.13	1.96
A4	2	2	2.13	0.34	9	9	11.66	1.39
A5	4	4	4.03	0.18	30	30	30.33	0.54
A6	6	6	6.23	0.63	45	45	48.3	6.63
A7	6	6	6.0	0.0	46	46	46.26	0.45
R1	42	42	43.43	0.56	774	851	889.96	31.34
R2	29	30	30.1	0.3	486	503	522.2	9.8
R3	17	17	17.73	0.44	250	268	276.4	4.19
R4	28	28	28.53	0.57	725	738	758.26	12.27
R5	12	12	13.1	0.4	146	147	151.03	2.19

Table 16: Statistics for the results obtained in 30 runs of SWO initialized with random permutations, with 8000 evaluations per run. The mean and best value from 30 runs as well as the standard deviations are shown. For each problem, the best known solution for each objective function is also included.

6.5 Multiple Moves or What Genitor and SWO Have in Common

As in other problems with many plateaus (e.g., MAXSAT), we hypothesize that long leaps in the search space are instrumental for an algorithm to perform well on AFSCN scheduling. Like *Genitor*, SWO makes multiple changes in the permutation at each step. In fact, the SWO mechanism of moving forward multiple, known to be problematic requests seems to be the dominant performance factor; by moving forward all the requests in conflict, even when starting from random permutations, most of the solutions are better than those produced by *Genitor*.

To test the effect of multiple moves in traversing the search space, we implemented a version of SWO where only one request is moved forward at each step. For minimizing the conflicts, one of the bumped requests is randomly chosen; for minimizing overlaps, one of the requests contributing to the sum of overlaps is randomly chosen. For both minimizing the conflicts and minimizing the sum of overlaps the chosen request is moved forward for a constant distance of five[9]. We called this new algorithm *SWO1Move*. The results obtained by running *SWO1Move* for 30 runs with 8000 evaluations per run are presented in Table 17.

When minimizing conflicts, *SWO1Move* performs as well as SWO (in fact, it finds the best known solution for R1 as well). When minimizing the sum of overlaps, the performance of SWO for the *R* problems worsens significantly when only one task is moved forward. Previously, we implemented *SWO1Move* for minimizing overlaps by moving forward the request that contributes most to the total overlap (Barbulescu et al., 2004c). Randomly choosing the request to be moved forward improved the performance of *SWO1Move*[10];

9. We tried other values; on average, a value of five seems to work best.
10. Randomization is useful because SWO can become trapped in cycles (Joslin & Clements, 1999)

Day	Minimizing Conflicts			Minimizing Overlaps		
	Min	Mean	Stdev	Min	Mean	Stdev
A1	8	8	0	104	104	0
A2	4	4	0	13	13	0
A3	3	3	0	28	28	0
A4	2	2	0	9	9	0
A5	4	4	0	30	30	0
A6	6	6	0	45	45	0
A7	6	6	0	46	46	0
R1	42	43.4	0.7	872	926.7	22.1
R2	29	29.9	0.3	506	522.9	8.9
R3	18	18	0	271	283.0	6.1
R4	28	28.1	0.3	745	765.2	10.7
R5	12	12	0	146	146	0

Table 17: Performance of a modified version of SWO where only one request is moved forward for a constant distance 5. For both minimizing conflicts and minimizing the sum of overlaps, the request is randomly chosen. All statistics are taken over 30 independent runs, with 8000 evaluations per run.

however, the improvement is not enough to equal the performance of SWO when minimizing overlaps for the new days of data. In fact, longer runs of *SWO1Move* with a random choice of the request to be moved (30 runs with 50000 evaluations) produce solutions that are still worse than those obtained by SWO. These results support the conjecture that the performance of SWO is due to the simultaneous moves of requests.

We attribute the discrepancy in the *SWO1Move* performance for the two objective functions to the difference in the discretization of the two search spaces. When minimizing conflicts, *SWO1Move* only needs to identify the requests that can not be scheduled. More fine tuning is needed when minimizing the sum of overlaps; besides from identifying the requests which can not be scheduled, *SWO1Move* also needs to find the positions for these requests in the permutation such that the sum of overlaps is minimized. We conjecture that this fine tuning is achieved by simultaneously moving forward multiple requests.

6.6 Progress Toward the Solution

SWO and Genitor apply different criteria to determine solution modifications. Local search randomly chooses the first shift resulting in an equally good or improving solution. To assess the effect of the differences, we tracked the best value obtained so far when running the algorithms. For each problem, we collected the best value found by SWO, *Genitor* and *SeededGenitor* and local search using the randomized neighborhood in increments of 100 evaluations, for 8000 evaluations. We averaged these values over 30 runs of SWO, local search, *Genitor* and *SeededGenitor*, respectively. For the *A* problems, SWO and *SeededGenitor* quickly find the best known solution; local search and *Genitor* take longer.

A typical example for the *R* problems for each objective function is presented in Figures 7 and 8. For both objective functions, the curves are similar, as is relative performance. SWO quickly finds a good solution, then its performance levels off. This could be explained by an increase in the size of the plateaus as search progresses (see Section 6.1.3). *SeededGenitor* steadily progresses in smaller steps toward the best solution, and while it takes longer to reach values as good as the ones produced by SWO, it outperforms SWO given enough evaluations. Using the randomized neighborhood, local search finds better solutions faster than *Genitor*.

We observe two differences in the objective functions. For minimizing the number of conflicts, the crossover point between *SeededGenitor* and SWO is earlier than for minimizing the overlaps. Also, when minimizing the number of conflicts, *Genitor* eventually equals or outperforms SWO. For minimizing overlaps, *Genitor* takes longer to find good solutions; given 8000 evaluations, *Genitor* does not find solutions as good as the ones produced by SWO.

Given more evaluations, the solutions found by *Genitor* for the *R* problems continue to improve. In fact, the best known solutions for these problems can be obtained by running *Genitor* for 50,000 evaluations in 30 runs. Running SWO for 50,000 evaluations in 30 runs results in small improvements and only for R2 (from 491 in 8000 evaluations to 490) and R4 (from 731 to 728).

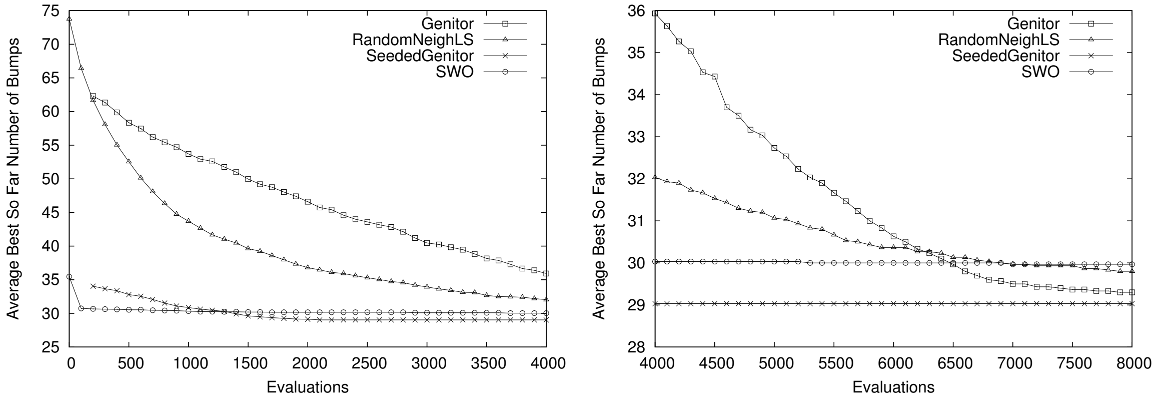


Figure 7: Evolutions of the average best value for conflicts obtained by SWO, *SeededGenitor*, local search with the randomized neighborhood and *Genitor* during 8000 evaluations, over 30 runs. The left figure depicts the improvement in the average best value over the first 4000 evaluations. The last 4000 evaluations are depicted in the right figure; note that the scale is different on the y-axis.

7. Conclusions

In scheduling, the understanding of *why* an algorithm performs well is very limited. However, identifying the factors that contribute to the performance of an algorithm is very important for: 1) identifying an algorithm that will work well given a domain, 2) demonstrating that the algorithm choice is fit for the domain and 3) motivating improvements in

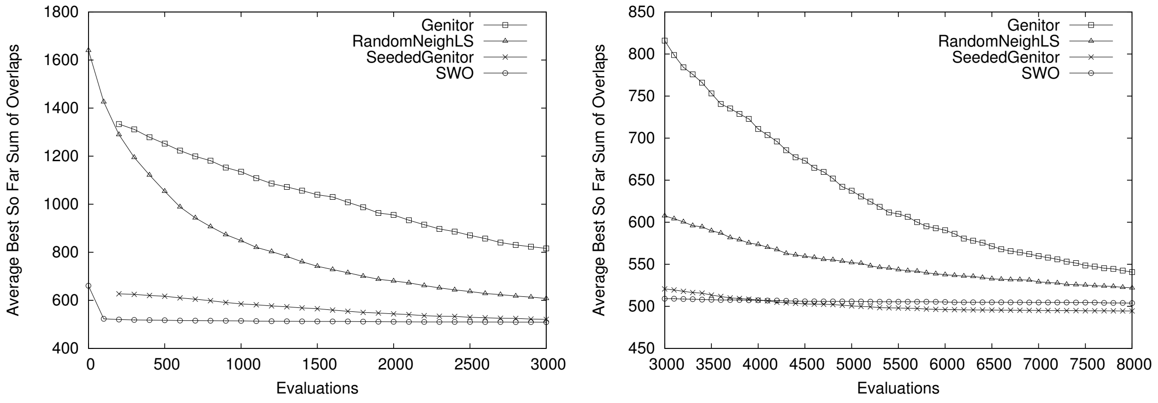


Figure 8: Evolutions of the average best value for sum of overlaps obtained by SWO, *SeededGenitor*, local search with the randomized neighborhood and *Genitor* during 8000 evaluations, over 30 runs. The left figure depicts the improvement in the average best value over the first 3000 evaluations. The last 5000 evaluations are depicted in the right figure; note that the scale is different on the y-axis.

algorithms. The main purpose of our paper is to identify factors in algorithm performance for an oversubscribed scheduling application: AFSCN access scheduling.

Instead of focusing on a competitive approach showing that some algorithm performs best, we found *a set* of fairly simple algorithms that work well for our application and investigated reasons for such performance results. The algorithms we found to excel on AFSCN scheduling are: local search using shifting, *Genitor* and SWO. All these algorithms operate in the search space of permutations of tasks; a greedy schedule builder converts the permutations into schedules. However, the paths followed by these algorithms when traversing the search space are quite different. We focused on answering the question: What makes each of the algorithms in our set a good fit for AFSCN scheduling?

For each algorithm, we formulated hypotheses about the factors in performance. We showed that plateaus are dominant in the permutation search space. For local search, 80% of the moves represent steps on plateaus. We also showed that the ordering of neighbors during next-descent local search is essential for traversing the plateaus more efficiently. For *Genitor*, we found that it discovers both some simple patterns (which exploit domain knowledge) in ordering the tasks, and some fairly complex relationships. SWO is taking long leaps across the search space; we showed that the multiple simultaneous changes to the schedule are the main factor favoring good SWO performance. We also investigated whether initializing the search closer to good solutions is a determining factor in performance. We found that such initialization helps, but is not by itself enough.

Our plans for future work follow two main directions. First, we will identify key problem factors in algorithm performance for oversubscribed applications. In particular, we investigate what about a certain heuristic makes it suitable when solving a particular oversubscribed application, but results in poor performance for a different one. For AFSCN scheduling, we designed various techniques to identify algorithm performance factors. A second direction for our future research is applying our approach beyond AFSCN access scheduling.

References

Aickelin, U., & Dowsland, K. (2003). An indirect genetic algorithm for a nurse scheduling problem. *Computers & Operations Research*, *31*(5), 761–778.

Aickelin, U., & White, P. (2004). Building better nurse scheduling algorithms. *Annals of Operations Research*, *128*, 159–177.

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

Barbulescu, L., Watson, J., Whitley, D., & Howe, A. (2004b). Scheduling Space-Ground Communications for the Air Force Satellite Control Network. *Journal of Scheduling*.

Barbulescu, L., Whitley, L., & Howe, A. (2004c). Leap before you look: An effective strategy in an oversubscribed problem. In *Proceedings of the Nineteenth National Artificial Intelligence Conference*.

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

Bresina, J. (1996). Heuristic-Biased Stochastic Sampling. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 271–278, Portland, OR.

Burrowbridge, S. E. (1999). Optimal Allocation of Satellite Network Resources. In *Masters Thesis*. Virginia Polytechnic Institute and State University.

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

Cormen, T., Leiserson, C., & Rivest, R. (1990). *Introduction to Algorithms*. MIT press, Cambridge, MA.

Deale, M., Yvanovich, M., Schnitzuius, D., Kautz, D., Carpenter, M., Zweben, M., Davis, G., & Daun, B. (1994). The Space Shuttle ground processing scheduling system. In Zweben, M., & Fox, M. (Eds.), *Intelligent Scheduling*, pp. 423–449. Morgan Kaufmann.

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

Frank, J., Cheeseman, P., & Stutz, J. (1997). When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, *7*, 249–281.

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

Gent, I., & Walsh, T. (1993). Towards an Understanding of Hill-climbing Procedures for SAT. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*. AAAI Press.

Gent, I., & Walsh, T. (1995). Unsatisfied variables in local search. In *Hybrid Problems, Hybrid Solutions*, pp. 73–85. IOS Press Amsterdam.

Globus, A., Crawford, J., Lohn, J., & Pryor, A. (2003). Scheduling earth observing satellites with evolutionary agorithms. In *International Conference on Space Mission Challenges for Information Technology*, Pasadena, CA.

Gooley, T. (1993). Automating the Satellite Range Scheduling Process. In *Masters Thesis*. Air Force Institute of Technology.

Heckendorn, R. (1999). *Patterns of Epistasis and Optimization Problem Difficulty for Evolutionary Algorithms*. Ph.D. thesis, Colorado State University, Department of Computer Science, Fort Collins, CO.

Hooker, J. (1995). Testing heuristics: We have it all wrong. *Journal of Heuristics*, *1*, 33–42.

Jang, K. (1996). The Capacity of the Air Force Satellite Control Network. In *Masters Thesis*. Air Force Institute of Technology.

Johnston, M., & Miller, G. (1994). Spike: Intelligent scheduling of Hubble space telescope observations. In Morgan, M. B. (Ed.), *Intelligent Scheduling*, pp. 391–422. Morgan Kaufmann Publishers.

Joslin, D. E., & Clements, D. P. (1999). “Squeaky Wheel” Optimization. In *Journal of Artificial Intelligence Research*, Vol. 10, pp. 353–373.

Kramer, L., & Smith, S. (2003). Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems. In *Proceedings of 18th International Joint Conference on Artificial Intelligence*, Acapulco, Mexico.

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

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

Pemberton, J. (2000). 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.

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

ROADEF03. French society of operations research and decision analisys, ROADEF Challenge 2003. http://www.prism.uvsq.fr/˜ vdc/ROADEF/CHALLENGES/2003/.

Schalck, S. (1993). Automating Satellite Range Scheduling. In *Masters Thesis*. Air Force Institute of Technology.

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

Singer, J., Gent, I., & Smaill, A. (2000). Backbone Fragility and the Local Search Cost Peak. In *Journal of Artificial Intelligence Research*, Vol. 12, pp. 235–270.

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

Starkweather, T., McDaniel, S., Mathias, K., Whitley, D., & Whitley, C. (1991). A Comparison of Genetic Sequencing Operators. In Booker, L., & Belew, R. (Eds.), *Proc. of the 4th Int'l. Conf. on GAs*, pp. 69–76. Morgan Kaufmann.

Syswerda, G. (1991). Schedule Optimization Using Genetic Algorithms. In Davis, L. (Ed.), *Handbook of Genetic Algorithms*, chap. 21. Van Nostrand Reinhold, NY.

Watson, J.-P. (2003). *Empirical Modeling and Analysis of Local Search Algorithms for the Job-Shop Scheduling Problem.* Ph.D. thesis, Colorado State University, Department of Computer Science, Fort Collins, CO.

Whitley, D., Starkweather, T., & Fuquay, D. (1989). Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator. In Schaffer, J. D. (Ed.), *Proc. of the 3rd Int'l. Conf. on GAs.* Morgan Kaufmann.

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

Wolfe, W. J., & Sorensen, S. E. (2000). Three Scheduling Algorithms Applied to the Earth Observing Systems Domain. In *Management Science*, Vol. 46(1), pp. 148–168.

Zweben, M., Daun, B., & Deale, M. (1994). Scheduling and rescheduling with iterative repair. In Zweben, M., & Fox, M. (Eds.), *Intelligent Scheduling.* Morgan Kaufmann.