Trading Places: How to Schedule More in a Multi-Resource Oversubscribed Scheduling Problem[*]

Laura Barbulescu and **Adele E. Howe** and **L. Darrell Whitley** and **Mark Roberts**
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
email:{laura,howe,whitley,mroberts}@cs.colostate.edu

Abstract

Oversubscribed scheduling problems require removing tasks when enough resources are not available. Prior AI approaches have mostly been constructive or repair-based heuristic search. In contrast, we have found a genetic algorithm (GA) to be the best approach to the overconstrained problem of Air Force Satellite Control Network scheduling. We present empirical results that elucidate sources of difficulty in the application and partially explain why the GA is well suited to this problem. We show that the task interaction compels changes involving many tasks simultaneously and the GA appears to be learning domain specific patterns in the data.

Introduction

Most scheduling research has studied applications in which all tasks will get scheduled ... eventually. These applications, e.g., job shop scheduling, exploit evaluation metrics such as makespan where the value degrades as the finish time increases. In contrast, typical problems in oversubscribed scheduling applications make more demands on some of the resources than can be accommodated; consequently, some tasks must be discarded. The most critical evaluation metric to minimize is number of tasks bumped: a metric with a smaller range and correspondingly less discriminatory power between solutions.

In AI, constraint-directed search procedures are commonly applied to scheduling. Constructive approaches incrementally and heuristically add tasks until no more can be accommodated, backtracking or re-starting search to explore alternative choices (e.g., (Bresina 1996)). Repair-based approaches start with a schedule and iteratively modify it to accommodate more tasks (e.g., (Kramer & Smith 2003)).

Constraint-directed methods have not proven effective for our oversubscribed application: Air Force Satellite Control Network access scheduling (AFSCN). In fact, we have found that a Genetic Algorithm (GA) has the best performance relative to several constraint directed techniques, local search and tailored heuristic techniques. The GA is similar to the repair-based techniques in that it manipulates populations of full solutions to iteratively improve the solution, but differs in how it goes about the improvement.

In this paper, we present those results and describe a set of experiments designed to identify the complexities in this application and partially explain why a GA might excel on it. We show that the improvement technique (the crossover operator) may be superior because it makes more than pairwise changes. We describe a domain specific pattern that the GA appears to be learning and present results of a heuristic based on it. We investigate whether the search space might be reduced by considering only limited movement of tasks within a schedule. Finally, we examine whether the GA might be learning the general placement of critical tasks.

Experiments show that the GA is learning some simple patterns which exploit domain knowledge; however, that domain knowledge is not enough to account for the performance. The GA appears to be discovering fairly complex relationships in the data that are not easily found through simple repair operations. We hypothesize that several characteristics of our application and other oversubscribed applications make them well suited to an adaptive, less local solution: 1) tasks may be placed on alternative resources within certain constraints, 2) domain knowledge, especially relating to prioritization, may be learned, and 3) the placement of tasks may cause cascade effects in the schedule.

AFSCN Scheduling

The U.S. AFSCN is responsible for coordinating communications between users on the ground and satellites in space. Communications to more than 100 satellites are performed through 16 antennas at nine ground stations located around the globe. To reserve a particular anntena for a period of time, users submit a task request which includes a required duration, a time window within which the duration must occur and a desired resource. Alternate time windows and antennas may also be specified. Over 500 requests are typically received for one day. Separate schedules are produced for each day. The generic problem of scheduling task requests for communication antennas is referred to as the Satellite Range Scheduling Problem (SRSP) (Schalck 1993).

Naturally, some communication antennas are typically oversubscribed. After human schedulers attempt to fit all tasks into the schedule, often about 120 conflicts, or requests that could not be accommodated, remain. Because satellites are extremely expensive resources and the tasks may be mission critical for organizations in the Defense Department, absolute rejection of requests is not an option. Rather, human schedulers must engage in a complex, time-consuming arbitration process to create a conflict-free schedule.

Human schedulers use and balance many (hard to quantify) criteria to develop a conflict-free schedule. We focus on a single, although crucial, aspect of the problem: minimizing the number of conflicts before the arbitration process. Human schedulers do not consider any conflict worse than any other conflict (Gooley 1993; Schalck 1993; Parish 1994), and the human schedulers state that reducing the number of conflicts up-front reduces 1) the work-load of human schedulers, 2) communication with outside agencies, and 3) the time required to produce a feasible schedule.

Approaches to Oversubscribed Scheduling

The AFSCN application is a multiple resource, multi-capacity oversubscribed problem. Examples of other such applications are USAF Air Mobility Command (AMC) airlift scheduling (Kramer & Smith 2003), NASA's shuttle ground processing (Deale *et al.* 1994), scheduling telescope observations (Bresina 1996) and satellite observation scheduling (Frank *et al.* 2001; Globus *et al.* 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 by priority order and then attempting to insert unscheduled tasks by retracting and re-arranging conflicting tasks. AMC differs from the AFSCN problem in one critical way: each delivery must occur within a particular time window and does not include alternative time/resource combinations.

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.

Satellite observation scheduling requires matching customer requests (e.g., instrument required, locations and times for the sensing event) for data collection to appropriate satellite resources. Frank et al.(2001) proposed a constraint-based planner with a stochastic greedy search algorithm based on Bresina's HBSS algorithm (Bresina 1996). Globus et al.(2003) compared a genetic algorithm, simulated annealing, squeaky wheel optimization 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.

The AFSCN application was previously studied by researchers from the Air Force Institute of Technology (AFIT). Gooley (1993) and Schalck (1993) developed algorithms based on mixed-integer programming (MIP) and insertion heuristics. Parish (1994) applied the *Genitor* genetic algorithm (Davis 1991), which scheduled roughly 96% of all task requests, out-performing the MIP approaches. In previous studies (Barbulescu *et al.* 2004), we have also found *Genitor* to perform best for this problem.

What Does Work: Comparison of Approaches

We considered a variety of algorithms: constructive heuristics, genetic algorithm, repair-based and local search. Constructive heuristics begin with an empty schedule and iteratively add jobs to the schedule using local decision rules. We implemented a greedy constructive heuristic $Greedy_{DP}$, which is an extension of a one-machine greedy heuristic defined by Dauère-Pérès (1995) . The one-machine heuristic schedules the jobs with the earliest due date immediately, while simultaneously minimizing the length of the partial schedule at each step. For each of the resources, we use the one-machine heuristic to schedule the requests that specify that resource as an alternative and are not scheduled yet; the result is an initial schedule. Then we consider the unscheduled requests and attempt to insert them in the schedule. The unscheduled requests are considered in an arbitrary order; the request is scheduled at the earliest time on the first alternative resource available and bumped if none of the alternative resources are available.

As the repair-based algorithm, we implemented Gooley's (1993) algorithm, which uses domain specific heuristics to construct an initial schedule and then repair it to fit in more requests. Complex rules are used to compute flexibility measures for the requests, which indicate which requests can be moved in order to accommodate unscheduled requests. These measures are based on the ratio between the duration of a request and its time window, the number of alternative resources, and the blocks of free time on resources.

The remaining algorithms 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. If the request cannot be scheduled on any of the alternative resources, it is dropped from the schedule (i.e., bumped).

Genetic algorithms were found to perform well in the AFIT studies and for an abstraction of NASA's Earth Observing Satellite (EOS) scheduling problem, denoted the Window Constrained Packing Problem (WCPP) (Wolfe & Sorensen 2000), which considers a single resource. For our studies, we used the version of *Genitor* originally developed for a warehouse scheduling application (Starkweather *et al.* 1991). Like all genetic algorithms, *Genitor* maintains a population of solutions. In each step of the algorithm, a pair of parent solutions is selected, and a crossover operator is used to generate a single child solution, which then replaces the worst solution in the population. Selection of parent solutions is based on the rank of their fitness, relative to other solutions in the population. A linear bias is used such that indi-

Day	Size	Genitor			Hill Climbing			Random Sampling			Greedy$_{DP}$	Gooley
		Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev		
10/12/92	322	8	8.6	0.49	15	18.16	2.54	21	22.7	0.87	21	11
10/13/92	302	4	4	0	6	10.96	2.04	11	13.83	1.08	22	7
10/14/92	311	3	3.03	0.18	11	15.4	2.73	16	17.76	0.77	25	5
10/15/92	318	2	2.06	0.25	12	17.43	2.76	16	20.20	1.29	23	4
10/16/92	305	4	4.1	0.3	12	16.16	1.78	15	17.86	1.16	18	5
10/17/92	299	6	6.03	0.18	15	18.16	2.05	19	20.73	0.94	28	7
10/18/92	297	6	6	0	10	14.1	2.53	16	16.96	0.66	22	6
03/07/02	483	42	43.7	0.98	68	75.3	4.9	73	78.16	1.53	97	45
03/20/02	457	29	29.3	0.46	49	56.06	3.83	52	57.6	1.67	72	36
03/26/03	426	17	17.63	0.49	34	38.63	3.74	38	41.1	1.15	55	20
04/02/03	431	28	28.03	0.18	41	48.5	3.59	48	50.8	0.96	66	29
05/02/03	419	12	12.03	0.18	15	17.56	1.3	25	27.63	0.96	45	13

Table 1: Performance of *Genitor*, hill climbing, and random sampling in terms of the best and mean number of bumped requests. All statistics are taken over 30 independent runs, with 8000 evaluations per run. The results of running the greedy heuristic *Greedy$_{DP}$* and Gooley's algorithm are also included.

viduals that are above the median fitness have a rank-fitness greater than one and those below the median fitness have a rank-fitness of less than one. Following Parish (1994), we use Syswerda's (1991) position-based crossover operator.

As the local search algorithm, we implemented a hill-climber. Because it has been successfully applied to a number of well-known scheduling problems, we selected a domain-independent move operator, the *shift* operator. From a current solution π, a neighborhood is defined by considering all $(N-1)^2$ pairs (x, y) of task request ID positions in π, subject to the restriction that $y \neq x - 1$. The neighbor π' corresponding to the position pair (x, y) is produced by *shifting* the job at position x into the position y, while leaving all other relative job orders unchanged. If $x < y$, then $\pi' = (\pi(1), ..., \pi(x-1), \pi(x+1), ..., \pi(y), \pi(x), \pi(y+1), ..., \pi(n))$. If $x > y$, then $\pi' = (\pi(1), ..., \pi(y-1), \pi(x), \pi(y), ..., \pi(x-1), \pi(x+1), ..., \pi(n))$.

Given the large neighborhood size, we use the shift operator in conjunction with next-descent hill-climbing: the neighbors of the current solution are examined in a random order, and the first neighbor with either a lower or equal number of discarded tasks is accepted. Search is initiated from a random permutation and terminates when a pre-specified number of solution evaluations is exceeded.

We compared the four algorithms on 12 days of actual data from the AFSCN: seven consecutive days from 1992 and five days from 2002-2003[1]. We also built a generator to support experiments but have found that the real data encompass complexities and subtle interactions that are not easily synthesized. The results of the comparison are summarized in Table 1. *Genitor* yields the best performance. *Greedy$_{DP}$* results in the worst performance; it is often outperformed by random sampling. Note also that the problems have become

significantly larger, leading to more bumped requests.

How Much Do Tasks Interact

Our results on AFSCN scheduling with Genitor run counter to most others' experiences. As shown in a previous section, research on oversubscribed scheduling favors iterative repair strategies embedded in search algorithms. Thus, what is it about AFSCN that makes it most amenable to solution with a genetic algorithm? In this section, we consider whether the GA may be learning and exploiting task interactions.

How Much Contention Is There?

The most distinctive characteristic of AFSCN scheduling is the availablity 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. This can lead to a cascade effect of considering mutual constraints between requests. First, we examine 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 *et al.* 1997)). As with SumHeight, we build contention graphs by adding all tasks into all of their possible slots. For our problems, we found that demand is consistent (i.e., only a few time slots that have zero requests) and can get pretty high. Table 2 summarizes contention for the AFSCN data. Figure 1 shows the contention profile for the resource with the most requests vying for the same resource on the day with the highest demand. 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 examine the overlap in alternatives at the points of highest contention. We found that many requests do share alternatives. This may help reduce somewhat the branching factor as constructive search progresses; unfor-

Date	SH	Req.	Mean	#Alts	Overlap
10/12/92	7.72	6	2.98	5.34	.47
10/13/92	8.78	8	3.63	6.34	.82
10/14/92	8.99	7	3.34	6.12	.67
10/15/92	7.37	4	3.44	6.25	1.0
10/16/92	8.17	6	3.32	6.15	.73
10/17/92	8.50	7	3.34	6.16	.76
10/28/92	8.53	8	2.85	6.18	.43
03/07/02	10.74	8	3.66	4.46	.5
03/20/02	10.48	7	3.22	4.84	.52
03/26/03	8.83	7	2.55	4.42	.33
04/02/03	9.21	9	2.87	4.54	.43
05/02/03	7.59	5	2.32	4.50	.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. Fourth is the average contention over all resources. 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.
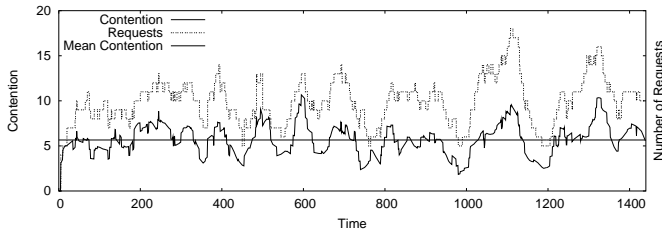

Figure 1: Example of resource contention graph for 3/7/2002. This resource had the highest number of tasks (18) contending for a single slot.

tunately, we also found that most of them prioritize the alternatives similarly, making it somewhat more difficult to trade-off placement. The last columns in Table 2 show that many alternatives are possible and many (or all) tasks share the same set of alternatives.

Are Pairwise Interactions Enough?

A key issue when modifying solutions is how much change is required to improve the evaluation. Some algorithms (e.g., hill climbing with the shift operator or ordering using min-slack), modify schedules by forcing a task's position relative to others. Such operations directly address pairwise interactions and *possibly* indirectly address n-way interactions. A problem characteristic that clearly affects the success of these operations is the degree of interactions between tasks.

We investigate two aspects of task interaction in AFSCN. First, we test how sensitive solutions are to task shifts; in particular, does the schedule produced by the permutation change if one job is shifted in it? Second, we test whether some tasks do not interact with any others.

The permutation space is clearly highly redundant. Multiple permutations can be converted by the schedule builder into identical schedules. To test schedule sensitivity to per-

mutation perturbations, we define two schedules to be identical if each request is scheduled on the same resource, starting at the same time in both schedules. We explore pairwise interactions between requests in random permutations. For each problem, we create 30 random permutations of the requests. For each such permutation and each pair of requests (A, B), we build 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 count for how many pairs (A, B) in a permutation the corresponding schedule does not change 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". We compute the average number of pairs of non-interacting requests over the 30 random permutations. We also compute the average number of pairs of non-interacting requests over 30 permutations evaluated as best known values by *Genitor*.

The results of the pairwise sensitivity test are summarized in Table 3. The second column *Total Pairs* represents the total number of request pairs in the permutation (if there are n requests, the number in this column is $n(n-1)/2$). The mean and standard deviation of the number of non-interacting pairs of requests (from 30 random or optimal permutations) appear in the columns denoted *Mean* and *Stdev*. 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 numbers in the table emphasize three main points: 1) During each step of the search, the number of moves under the shifting operator is huge (if there are n requests, there are $(n-1)^2$ possible moves). 2) Almost 40% of all the pairs of requests result in shifting moves which do not change the schedule corresponding to the current permutation. 3) The percentage of non-interacting pairs of requests does not seem to be sensitive to the quality of the solution corresponding to the permutation (it is still approximately 40% for best known solutions obtained by *Genitor*).

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, we constructed some heuristic methods to exploit this and found they had rather poor performance.

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 permu-

Day	Total Pairs	Non-interacting Pairs					
		Random Permutations			Optimal Permutations		
		Mean	Stdev	Average %	Mean	Stdev	Average %
10/12/92	51681	21999.7	724.959	42.5	22911.3	636.95	44.3
10/13/92	45451	20659.7	697.661	45.4	21194.6	814.349	46.6
10/14/92	48205	20652.3	680.159	42.8	20655.6	663.784	42.8
10/15/92	50403	20632.3	650.973	40.9	21240.4	697.542	42.1
10/16/92	46360	18850.7	676.609	40.6	19019.8	639.223	41.0
10/17/92	44551	17882.3	642.91	40.1	17804.4	574.492	39.9
10/18/92	43956	19278.3	836.376	43.8	20001.6	755.789	45.5
03/07/02	116403	44001.2	1356.4	37.8	45068.8	1208.93	38.7
03/20/02	104196	38246.5	977.8	36.7	38775	1149.02	37.2
03/26/03	90525	35779.9	854.12	39.5	36464.3	930.87	40.2
04/02/03	92665	36399.7	1115.37	39.2	38032	988.082	41.0
05/02/03	87571	35590.5	1308.93	40.6	36368.3	839.215	41.5

Table 3: Statistics for the number of pairs of non-interacting requests over 30 random and optimal permutations.

tation 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.

These examinations show that simple shifts may not effectively change a schedule and that pairwise interactions do not capture well the complexities of the AFSCN problems. Given a permutation, almost half of all possible shifting moves result in *identical* schedules, making local search based on a shift operator highly inefficient. Moreover, operators based on pairwise movements may also fail to account for the higher level interactions that we have observed.

Is it Learning a Simple Domain Heuristic?

One of our early hypotheses about why *Genitor* did well was that it was learning a simple domain specific heuristic. Requests to access low altitude satellites tend to be of short duration with no slack. High altitude requests are more variable, but in general, have larger time windows in which they fit. We hypothesized that *Genitor* may be learning to schedule the low-altitude, more tightly constrained requests before the high-altitude requests with which they interact.

To test the hypothesis we designed a simple greedy "split heuristic" for the schedule builder that first schedules all the low-altitude requests in the order given by the permutation, followed by the high-altitude requests. (See (Barbulescu *et al.* 2004) for details and a description of the 1992 and recent data.) We compared the schedules from the original schedule builder and the heuristic version for permutations derived from *Genitor*. We found that: (1) for $> 90\%$ of the

Day	Best Known	Random Sampling-S		
		Min	Mean	Stdev
10/12/92	8	8	8.2	0.41
10/13/92	4	4	4	0
10/14/92	3	3	3.3	0.46
10/15/92	2	2	2.43	0.51
10/16/92	4	4	4.66	0.48
10/17/92	6	6	6.5	0.51
10/18/92	6	6	6	0

Table 4: Results of running random sampling with the split heuristic in 30 experiments, by generating 100 random permutations per experiment.

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

Table 5: Results of running *Genitor* with the split heuristic in 30 experiments, with 8000 evaluations per experiment.

best known schedules found by *Genitor* for the 1992 data, 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), (2) the split heuristic typically produces the best-known schedules for the 1992 data, even under random sampling, and (3) for the recent data, the split heuristic fails to find the best schedules. Tables 4 and 5 show the results of using the split heuristic with random sampling on the 1992 data and with *Genitor* on the recent data.

Is Learning Temporal Constraints Enough?

Clearly, a request for a time slot near the beginning of the day will always appear in the schedule before a request with

Day	Best Known	LocalSearch			ModifiedLocalSearch			CombinedLocalSearch		
		Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev
10/12/92	8	12	15.46	2.55	10	10.93	0.79	9	10.133	0.74
10/13/92	4	7	8.26	1.03	5	5.26	0.45	4	4.86	0.74
10/14/92	3	10	12.26	10	5	6.4	0.63	5	6.33	0.97
10/15/92	2	11	13.2	1.56	9	10.4	0.73	7	9.46	1.45
10/16/92	4	7	11.53	2.16	6	7.4	0.73	6	7.2	0.67
10/17/92	6	12	14.4	1.45	8	8.46	0.63	7	8.53	0.83
10/18/92	6	8	11.46	2.55	7	7.6	0.5	6	7.46	0.63
03/07/02	42	61	68.73	4.39	46	49.8	2.04	48	50.4	1.4
03/20/02	29	46	50.86	2.94	35	36.26	1.16	35	37.13	1.45
03/26/03	17	29	32.73	2.43	21	22.13	1.06	21	23.13	0.99
04/02/03	28	38	44.06	3.59	32	34.2	1.2	33	35.2	1.2
05/02/03	12	17	21.73	2.4	15	16.66	1.04	15	16.46	1.12

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

a time window near the end of the day, no matter in which order the requests appear in the permutation (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 define a reduced-size search neighborhood by restricting the movement of each request based on its time window. This implicitly results 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 modified 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 these shifts is better than or as good as the current one, we accept its permutation. This modified 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 in Table 6. For each version of local search, we performed 15 trials, allocating 16000 evaluations for each trial. The total number of evaluations performed for each version of local search is therefore identical to the total number of evaluations allocated when *Genitor* finds the best known solutions (30 trials of 8000 evaluations each). 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 results observed for the modified shift operator appear in the columns *ModifiedLocalSearch*. The

modified shift operator clearly results in better performance than shift, for each of the test problems.

Given enough evaluations, is local search using modified shift capable of producing best known solutions? To investigate this, we performed 30 trials of running both versions of local search with a large number of evaluations (500,000 evaluations). The results are presented in Table 7. Local search using shift resulted in best known values for each of the test problems. However, local search using the modified shift operator did not find best known values for any of the problems. To exploit this result, we also ran a version of local search combining the two operators: we allocated a percentage of the total number of evaluations (75%) to local search using the modified shift operator and used the best solution obtained as the starting point for local search with shift. The results obtained for this version of local search are reported in the last columns of Table 6 (*CombinedLocalSearch*). For most of the 1992 problems, further improvement of the best solutions obtained can be observed. However, it seems that for the new problems, the number of evaluations allocated is too small. Running the modified shift operator for 75% of the total number of evaluations does not yield the values reported when all the evaluations were allocated. Also, the 25% of the total number of evaluations allocated to the shift operator are insufficient to significantly improve over the starting values (obtained by running the modified shift operator).

Not all possible schedules can be reached given the reduced permutation search space. This suggests that pairwise relative orders are not enough to capture the interactions.

Is the Placement of Critical Tasks the Key?

We have shown that pairwise changes are not effective in finding good solutions. The position-based crossover operator used by *Genitor* produces more than just pairwise changes in the permutation. 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

Day	Best Known	LocalSearch			ModifiedLocalSearch		
		Min	Mean	Stdev	Min	Mean	Stdev
10/12/92	8	8	8.4	0.49	10	10.53	0.5
10/13/92	4	4	4.0	0.0	5	5.03	0.18
10/14/92	3	3	3.03	0.18	5	5.8	0.4
10/15/92	2	2	2.03	0.18	9	9.76	0.43
10/16/92	4	4	4.4	0.49	6	6.0	0.0
10/17/92	6	6	6.0	0.0	7	8.4	0.62
10/18/92	6	6	6.0	0.0	7	7.0	0.0
03/07/02	42	42	43.93	1.08	43	44.13	0.68
03/20/02	29	29	29.53	0.73	32	33.5	0.73
03/26/03	17	17	18.0	0.74	21	21	0.0
04/02/03	28	28	28.16	0.37	30	32.23	0.77
05/02/03	12	12	12.46	0.5	15	15.43	0.56

Table 7: Results of running local search in 30 experiments, by evaluating $500K$ permutations per experiment.

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
```

We hypothesize that complex interactions exist between the requests and that *Genitor* discovers such interactions. We focus on answering two questions: 1) Is *Genitor* learning patterns of request ordering? and 2) Is *Genitor* learning where to place requests in the permutation?

To answer the first question, we identified 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 value solutions for each problem. We found that almost all the common request orderings specify a low altitude request appearing before a high altitude request. For some of the problems, a few request ordering pairs between high altitude request are also present. We could not identify longer chains of requests, of the type "RequestA before RequestB before RequestC...". Second, we grouped the selected permutations based on the requests bumped in their corresponding solutions. Given that the objective function used by *Genitor* to obtain these solutions is the number of bumps, the relative order of the requests preceding the requests that got bumped is important. We found longer chains of ordered requests when the number of permutations resulting in the same bumps was small (smaller than 5). However, for large size groups of permutations (resulting in the same bumps), we only found chains of requests of length 2. Examination of these chains shows that sometimes the requests which 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 bumps. Consider the example in figure 2, and suppose that all requests can be
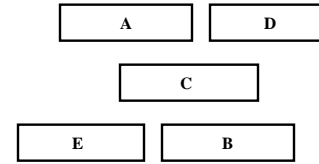

Figure 2: A, B, C, D, and E compete for two antennas at the same ground station.

scheduled on one of the two antennas present at a ground 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.

When we examined the permutations in final populations produced by *Genitor*, we found hundreds of chains of requests of varying lengths (from 2 to 15-20). While *Genitor* does seem to learn patterns of request ordering, multiple different patterns of request orderings can result in the same bumps (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 populations produced by *Genitor* might fail.

To answer the second question, we compared multiple permutations obtained by *Genitor* over multiple runs. We observed that for certain requests we can identify a portion of the permutation where those requests will be placed in each of the permutations. Also, examining the permutations in a final population produced by *Genitor* showed that some requests are localized in a certain region of the permutation. This is not true for all the requests; in fact, for most requests, their positions vary widely. The requests can appear anywhere from the first to the last position.

To test if indeed *Genitor* is learning positions of requests in the permutations, we build a "representative" permutation starting from a group of permutations as follows. We define the range of positions for a request as the difference between the rightmost position and the leftmost position in which the request appears in the considered group of permutations. Then, for each request with a range of positions smaller than a certain limit, we compute the average position (from its positions in the permutation group). We insert the request in the "representative" permutation in its computed average position (truncated to integer) if nothing is yet placed in that position; else it appears in the nearest next available position. Once all requests with a range of positions smaller than the imposed limit have been considered, we start inserting the rest of the requests. For each of the remaining requests, we compute the position in which the request appears most often (the mode) and insert the request in the closest available position to the computed one. The requests which seem localized in the group of permuta-

Day	Best Known	Final Population	*Genitor* Optima	LocalSearch Optima
10/12/92	8	10	9	11
10/13/92	4	4	5	6
10/14/92	3	3	3	5
10/15/92	2	3	2	5
10/16/92	4	4	4	7
10/17/92	6	6	6	8
10/18/92	6	6	6	7
03/07/02	42	42	45	51
03/20/02	29	29	29	36
03/26/03	17	17	19	22
04/02/03	28	28	28	36
05/02/03	12	13	12	17

Table 8: Values of "representative" permutations for permutations in: a final population, a group of 1000 *Genitor* solutions and a group of 30 local search solutions.

tions appear in the "representative" permutation in a position close to their average position in the group (and localized in the same portion of the permutation). The rest of the requests appear in a position close to their preferred position in the group. If *Genitor* is learning positions of requests in the permutations, the "representative" permutation corresponding to multiple permutations (from multiple runs) or to a final population produced by *Genitor* should be similar to the represented permutations (bumps approximately the same number of requests).

To test this, we considered the following groups of permutations for each test problem: a final population (size 200) and 1000 solutions obtained from 1000 runs of *Genitor*. Also, for each problem, we built "representative" permutations for 30 solutions obtained by running 30 trials of local search for 500K evaluations. We define the limit for the range of positions as one third of the permutation size. The results are summarized in Table 8. The "representative" permutations for permutations produced by *Genitor* evaluate close to best known values. However, for the permutations corresponding to local search optima, the values of the "representative" permutations are always worse than the best known values.

Conclusion

The AFSCN problem poses some unusual challenges to well established AI scheduling methods. For constructive algorithms, we have found that the branching factor can be large, due to the availability of many alternatives for each request. For local search algorithms, we have found barriers to defining operators with manageable sized neighborhoods; apparently, non-overlapping requests still appear interleaved in best known permutation solutions. However, a genetic algorithm appears to be able to identify and exploit patterns in the data, specifically prioritizing low-altitude requests before high-altitude requests (most of the time) and identifying approximate positions for critical tasks.

References

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

Beck, J. C.; Davenport, A. J.; Sitarski, E. M.; and Fox, M. S. 1997. Texture-based Heuristic for Scheduling Revisited. In *AAAI-97*, 241–248.

Bresina, J. 1996. Heuristic-Biased Stochastic Sampling. In *AAAI-96*, 271–278.

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

Davis, L. 1991. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.

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

Frank, J.; Jonsson, A.; Morris, R.; and 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*.

Globus, A.; Crawford, J.; Lohn, J.; and Pryor, A. 2003. Scheduling earth observing satellites with evolutionary agorithms. In *International Conference on Space Mission Challenges for Information Technology*.

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

Kramer, L., and Smith, S. 2003. Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems. In *IJCAI-03*.

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

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

Smith, S., and Cheng, C. 1993. Slack-based Heuristics for Constraint Satisfaction Problems. In *AAAI-93*, 139–144. Washington, DC: AAAI Press.

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

Syswerda, G. 1991. Schedule Optimization Using Genetic Algorithms. In Davis, L., ed., *Handbook of Genetic Algorithms*. NY: Van Nostrand Reinhold. chapter 21.

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

Zweben, M.; Daun, B.; and Deale, M. 1994. Scheduling and rescheduling with iterative repair. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. Morgan Kaufmann.