DISSERTATION

ANALYSES OF ALGORITHM PERFORMANCE FOR AN OVERSUBSCRIBED SCHEDULING PROBLEM

Submitted by

Laura Barbulescu

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

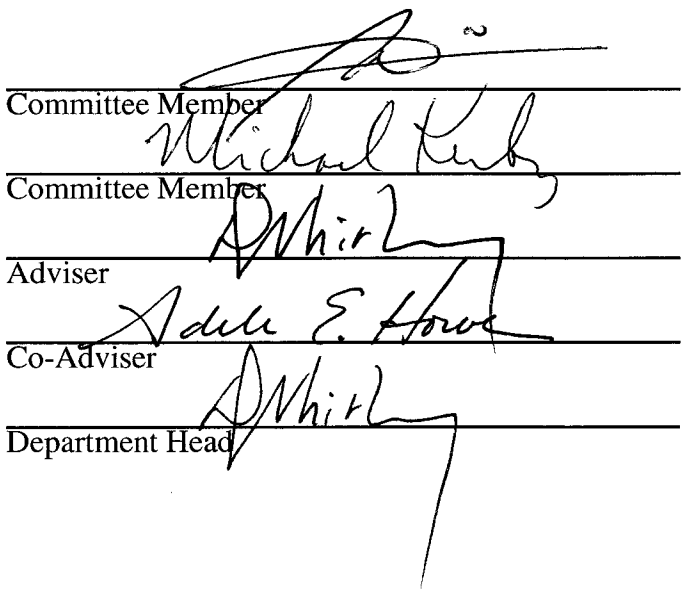
Fall 2005

COLORADO STATE UNIVERSITY

July 5, 2005

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY LAURA BARBULESCU ENTITLED ANALYSES OF ALGORITHM PERFORMANCE FOR AN OVERSUBSCRIBED SCHEDULING PROBLEM BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work



Committee Member

Committee Member

Adviser

Co-Adviser

Department Head

ABSTRACT OF DISSERTATION

ANALYSES OF ALGORITHM PERFORMANCE FOR AN OVERSUBSCRIBED SCHEDULING PROBLEM

We analyze the factors that influence algorithm performance for an oversubscribed application, scheduling for the Air Force Satellite Control Network (AFSCN). AFSCN scheduling assigns access requests to specific time slots on antennas at ground stations. The application is oversubscribed: not all tasks can be accommodated given the available resources. As a special class of scheduling problems, oversubscribed problems present additional challenges. While, in general, solutions to scheduling problems specify the start times and resources assigned to tasks, in oversubscribed scheduling the maximal subset of tasks that can be scheduled with the available resources also needs to be identified.

We implemented various algorithms for AFSCN scheduling. Some algorithms, such as a domain-specific repair-based algorithm or constraint-based scheduling heuristics, failed to identify good solutions. We have found a set of fairly simple algorithms that perform well on the AFSCN scheduling domain, for both real and synthetically generated problems. The algorithms in the set are: hill-climbing, a genetic algorithm (GA) and Squeaky Wheel Optimization (SWO). All the algorithms in the set are designed to traverse the same search space: solutions are represented as permutations of tasks; a greedy schedule builder converts the permutation into a schedule by assigning a 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. This research identifies performance factors that make each of the algorithms a good fit for AFSCN scheduling.

The AFSCN scheduling search space is dominated by plateaus, due to both the discrete nature of the objective function and to the fact that the schedule builder converts multiple permutations into identical schedules. Each algorithm handles plateaus differently. Hill-climbing randomly walks on the plateaus until it finds exits to lower plateaus: the higher the percentage of the space occupied by plateaus, the more random wandering is likely for hill-climbing. We found the ordering of the neighbors to be the main performance factor in expediting plateau traversal for hill-climbing. The GA and SWO both traverse the plateaus quickly, by making multiple changes to the solutions. The long, directed leaps across the search space are the main performance factor for the GA and SWO.

We also investigated whether initializing the search closer to the best solutions is the key to performance. We found that such initialization helps but is not by itself enough to explain algorithm performance results.

The main contributions of this research work are: 1) We performed the first coupled formal and empirical analysis of the AFSCN scheduling problem. 2) We designed techniques for analyzing algorithm performance, which could transfer to other applications. 3) We identified algorithm performance factors, which are likely to hold on other similar problems. 4) We designed a new best performing algorithm, by combining the features we found to have most influence on performance.

Laura Barbulescu
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
Fall 2005

ACKNOWLEDGEMENTS

I wish to thank my advisors, Adele Howe and Darrell Whitley for all the guidance, help and support throughout my Ph.D. years.

I also thank Dr. James T. Moore, Associate Professor, Dept. of Operational Sciences, Air Force Institute of Technology and Brian Bayless and William Szary from Schriever Air Force Base for providing the data.

I thank Dr. Jeremy Frank from the NASA Ames Research Center for, among other excellent advice, suggesting we take a second look at SWO. I thank Dr. John Bresina from NASA Ames Research Center for his insightful suggestions about limitations and possible extensions of this research.

I am grateful to my family, my Mom and Dad, my sister and my wonderful husband: without you, this dissertation would not have been possible.

Finally, I acknowledge the financial support I received from grants from the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grants number F49620-00-1-0144 and F49620-03-1-0233. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

To Stephanie

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Scheduling problems arise in a variety of domains, both military and civilian, from manufacturing to the space program. In general, scheduling problems specify a set of activities that require resources of limited capacity and need to be processed such that various constraints, primarily temporal, are satisfied. Fox (1994) defines scheduling as "assigning resources and times for each activity so that the assignments obey the temporal restrictions of activities and the capacity limitations of a set of shared resources". However, for many real-world scheduling problems, a solution satisfying all the temporal and resource restrictions may not exist. In particular, problems for which there are typically more requests than can be accommodated with the available resources have been classified as *oversubscribed* [SP92]. Oversubscribed scheduling problems can also be found in the scheduling literature under the names of *overconstrained* problems [Pem00] or problems with *overloaded resources* [BPP98].

While scheduling usually consists of assigning time slots and resources to the requests, for oversubscribed problems, the solution also identifies the best subset of the requests that can be satisfied given the resource and time constraints. The time slots and resources are then assigned to this subset An objective function is used to rate the subsets of requests to be scheduled: it could vary from a very simple "maximize the number of requests scheduled" to complex functions that use request priorities, user preferences,

weighted attributes, etc. When solving an oversubscribed problem, two questions need to be answered: which requests to schedule and what start times and resources to assign to them.

The focus application of this research is AFSCN (Air Force Satellite Control Network) access scheduling; it requires assigning access requests (communication relays to U.S.A. government satellites) to specific time slots on antennas at ground stations. For each request, multiple alternative antennas are specified, any of which can be assigned to the request. AFSCN scheduling 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. In prior research on AFSCN, the objective function maximized the number of tasks scheduled [Sch93] [Goo93] [Par94]. In our research, we also consider a new objective function that minimizes the temporal overlap of tasks on specific resources in a schedule containing *all* tasks.

We distinguish two forms of AFSCN scheduling: Single-Resource Scheduling (SiRS) and Multi-Resource Scheduling (MuRS). Single-Resource Scheduling is a simplification of the application in which there is only one resource (i.e., antenna) that is being scheduled. Multi-Resource Scheduling includes requests that can potentially be scheduled on one of several alternative resources. We start by analyzing synthetic SiRS and MuRS problems, then we focus on two data sets of real-world problems: a week of data from 1992 and five days from 2002 and 2003.

The work presented in this dissertation represents the first coupled formal and empirical analysis of the AFSCN scheduling problem. For the formal analysis of the prob-

---

[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.

lem, we present two main results. First, we show that a modified version of AFSCN scheduling where all the requests are low-altitude requests can be optimally scheduled in polynomial time. Second, we formally establish that AFSCN Scheduling is $\mathcal{NP}$-complete. For this, we show that minimizing the number of unsatisfied task requests on a single antenna (SiRS) is equivalent to minimizing the number of late jobs on a single machine. The decision version of this problem (is there a schedule with X late jobs, where X is the known minimum number of late jobs) is known to be $\mathcal{NP}$-complete. Nevertheless, finding optimal or near optimal solutions to many realistic single resource problem instances is relatively easy. For some reasonable size problems, *branch and bound* methods can be used to either find or confirm optimal solutions.

Our empirical study of AFSCN scheduling focuses on answering specific questions about algorithm performance. The first question we address is: which algorithms perform well for AFSCN scheduling? We implemented a variety of algorithms: some incorporate domain-specific heuristics, while others are more general or have been successfully used to solve other oversubscribed scheduling problems. We identify the algorithms producing the best results for each of the three types of problems: the synthetic SiRS and MuRS and the real-world problems. We also investigate how algorithm performance transfers from single resource to multiple resource problems, from synthetic to real-world problems and from the old problems (1992) to the more recent (2002 and 2003) real-world ones. Next, given a set of best performing algorithms for real-world AFSCN scheduling, we design experiments to answer the question: what algorithm features are essential to produce good performance results for real-world AFSCN scheduling? Gaining such insight into the factors that influence algorithm performance is important since it may suggest algorithm improvements to better fit the domain. Finally, are there any common characteristics that explain performance results for all the algorithms in our set? We look for such characteristics in the problem domain, more

specifically, we attempt to identify features of the problem search space that influence and so partially explain algorithm performance results for *all* the best performing algorithms for AFSCN scheduling. Finding features of the problem search space that influence algorithm performance may motivate the design of an algorithm to exploit the presence of such features.

Our results show that for both MuRS and the real-world problems a *set* of fairly simple algorithms produced the best results relative to several constraint directed and heuristic techniques; the algorithms in the set are: hill-climbing, a genetic algorithm (GA) and Squeaky Wheel Optimization (SWO). All three algorithms are designed to traverse the same search space: solutions are represented as permutations of requests, 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. Hill-climbing only applies minor changes to move from one solution to another in the search space. In contrast, both the GA and SWO usually propose large directed changes to the current solution at each iteration.

We formulate several hypotheses about why particular algorithms excel. We find that large flat regions (plateaus) appear to be the dominant feature in the search space. The plateaus are due both to the discrete nature of the objective function and the fact that the schedule builder converts multiple permutations into identical schedules (many-to-one mapping). The many-to-one mapping is a result of the way the schedule builder assigns resources to the requests: reordering requests that do not compete for resources does not affect their resource assignment in the schedule.

Each of the algorithms handles 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 find

that indeed 80% of the moves evaluated by local search are on plateaus. Additionally, we show that the ordering of the neighbors matters a great deal for expediting plateau traversal for local search. On the other hand, both the GA and SWO take long, directed leaps across the search space at each iteration. We show that these long leaps are the source of power for both the GA and SWO. We present results for modified versions of the GA and SWO where we control how far the algorithms can leap and allow only shorter leaps; we show that the shorter the leaps, the worse the performance.

We also hypothesized that the GA is discovering patterns of interaction between tasks. We show that the GA finds some simple patterns that exploit domain knowledge. However, the simple domain knowledge is not enough to account for the performance. The GA appears to be discovering fairly complex patterns of request orderings in the data. However, not all these patterns are necessary in order to produce good solutions. Subsets of solutions produced by the GA will exhibit different patterns of request orderings, and therefore, it is difficult to identify such patterns across a set of solutions.

We investigate whether starting the search closer to the best solutions is the key to performance. We find that initializing the search with good solutions as opposed to random ones helps, but is not by itself enough to explain performance results.

The plateaus in the search space appear to be the key performance barrier for all the algorithms in our set. For hill-climbing, it is essential to randomize the order of checking the shifting neighbors to effectively cross the plateaus. For the GA and SWO, we conjecture that the long leaps are a desired algorithm feature to expedite plateau traversal. In fact, we find that the performance of hill-climbing can be substantially improved by allowing multiple moves (and therefore long leaps) at each iteration.

Based on our analyses we design a new algorithm by combining the randomized order of checking the neighbors (as in hill-climbing) with the long leaps (as in SWO and the GA). The new algorithm performs better than hill-climbing, SWO and the GA.

The performance of the new algorithm offers more evidence to strengthen the conjecture that long leaps result in faster plateau traversal.

Other researchers studying different applications have also mentioned long leaps as a method to quickly move through the search space. For SAT (the satisfiability problem), large flat regions are also present. Gent and Walsh [GW95] proposed Jump-SAT, an algorithm that occasionally "jumps" through the space, to allow the search to move large distances in the search space, by making larger walk steps. For scheduling Earth Observing Satellites, which is also an oversubscribed scheduling problem, Al Globus et al.[GCLP04] note that multiple changes to the current solution are more effective than single changes. While Globus et al. do not mention the connection to the plateaus in the search space, we believe that in fact, similarly to AFSCN scheduling, the multiple changes are useful because they speed up plateau traversal.

With a few exceptions (such as job-shop scheduling), Hooker's "competitive testing paradigm" [Hoo95] is still prevalent in scheduling research when developing algorithms: when faced with solving a problem, most research focuses on comparing a set of algorithms and deciding which one works better. However, as Hooker[Hoo95] writes: "The competitive nature of most algorithmic experimentation is a source of problems that are all too familiar to the research community" and "Competitive testing tells us which algorithm is faster but not why". Part of the reason for competitive testing in scheduling is the fact that practitioners must *quickly* select an algorithm from a set when given a problem to solve; a detailed analysis of the problem and the search space is time consuming. In contrast to the competitive testing paradigm, the research presented in this dissertation is an analysis of the problem, algorithm and search space characteristics for a practical real-world application: AFSCN scheduling.

While our research targets AFSCN scheduling, the contributions of our research extend beyond this application domain. First, we designed various techniques to inves-

tigate reasons for algorithm performance. These techniques can transfer to other applications; in fact, recent work of Roberts et al.[RHW05] applies some of the techniques to another oversubscribed application: scheduling observations for fleets of Earth Observing Satellites. Second, we found that large plateaus are present in the search space. This should also be the case for other scheduling problems where a many-to-one mapping is present between the solutions in the search space and the actual schedule space. When alternative resources are present, it is likely that the alternative resources will specify sets of resources with similar properties, as needed to execute a task; multiple tasks might therefore specify the same set of alternative resources. This can also result in groups of schedules that are evaluated the same: different request assignments to alternative resources can be equivalent in terms of the objective function. Hence, it seems likely that the results we obtained are transferable to other domains. Third, we formulated the conjecture that long leaps are a desired algorithm feature when large plateaus are present in the search space. As mentioned above, there are some indications that this conjecture might in fact hold for other domains as well.

The balance of this dissertation is organized as follows. In Chapter 2 we overview the research published on oversubscribed scheduling. We include in the survey descriptions of some of the more prominent oversubscribed applications in the scheduling literature, as well as summaries of the solution methods proposed for each. A general description of the AFSCN scheduling problem is presented in Chapter 3. This chapter also includes a summary of previous research results, a proof of $\mathcal{NP}$-completeness and a description of a greedy algorithm to solve optimally a subclass of AFSCN scheduling in polynomial time. In Chapter 4 we introduce the algorithms we implemented for general AFSCN scheduling. We report the performance results for these algorithms in Chapter 5. The results show that a set of three algorithms perform best for our application. In Chapter 6 we investigate reasons for such performance results, by separately analyzing

performance details for each algorithm. Finally, in Chapter 7 we show that the AFSCN scheduling problem search space is dominated by large flat regions. We conjecture that algorithms taking large leaps in the search space perform well by moving faster across these flat regions; we show results supporting this conjecture for each of the three algorithms in our set. The last chapter of this dissertation concludes by summarizing the contribution of this work and lists possible future research directions.

Chapter 2

Background: Oversubscribed Scheduling Problems and Solution Methods

Oversubscribed scheduling problems are problems for which there are typically more requests than can be accommodated with the available resources [SP92]. While the domains in which oversubscribed scheduling applications can be found vary widely, common characteristics of such applications can be identified. First, by definition the resources are oversubscribed. Second, the requests typically specify start times, durations, deadlines, and priorities. Alternative resources and setup times are sometimes specified as well. Finally, the objective function is usually defined using the request priorities. For example, scheduling observations for telescopes is typically oversubscribed. Astronomers spend a lot of time observing objects as they set in the west. The windows of opportunity to observe such objects are usually short, and there are more observation requests than can be accommodated by the telescope. The requests specify the beginning of the visibility window for the object as the start time. The end of the visibility window is the deadline, and priorities are also assigned to the observations. Setup times are usually needed to point the telescope from one object to another. An objective function is used to decide which observations to schedule.

Smith et al.[SP92] mention the distinction between "oversubscribed" and

"constraint-relaxable" scheduling problems. Constraint-relaxable problems are those for which the resource capacity can be eventually increased if needed or for which the deadlines can be relaxed. A typical example of a constraint-relaxable problem from machine scheduling is minimizing the total tardiness. An example of a problem that is not "constraint-relaxable" is [RM99] the problem of scheduling airplane departures. The objective function is finding an optimal sequence of airplane departures from a runway. Almost all of the constraints imposed on airplane departures are safety regulations: the capacity of the runway cannot be increased and the constraints on the time intervals between successive departures cannot be modified. When not all the constraints can be satisfied, fewer airplanes will be scheduled for departure. Being imposed by safety regulations, the constraints cannot be relaxed. The applications presented in this chapter are not "constraint-relaxable".

Oversubscribed scheduling problems are frequently encountered in the scheduling literature. Both exact and approximate methods, varying from simple greedy heuristics, to complex, hybrid algorithms, have been employed to solve oversubscribed scheduling problems. This section is an overview of some of the most broadly researched oversubscribed scheduling problems. For each problem, we present a short problem description, the objective function and the solution methods. We conclude by presenting a summary of all the solution methods employed by the applications in this chapter.

2.1 Satellite Scheduling

Satellite systems are expensive limited resources. Hundreds of requests compete for antennas at ground stations, instruments, recording devices, and transmitters on the satellites. Typically, more tasks need to be scheduled than can be accommodated by the satellites and the ground stations. Frank et al.(2001) [FJMS01] note that currently, scientific activities on different satellites are scheduled independently of one another, and

thus a lot of manual coordination of the observations is needed. Automating the process of generating a schedule given a set of requests and the constraints associated with them can result in more efficient schedules. Automation also facilitates "what-if" scenarios, especially needed when negotiating with customers and re-scheduling are part of the scheduling process (as is the case of scheduling requests for military satellites, where outright rejection of a request is not an option).

A particular characteristic of satellite scheduling is the fact that the observations as well as the communication between the satellites and the ground stations depend on satellite visibility. Windows of potential visibility are defined based on the orbit of the satellite and the position of the ground station or of the object observed by the satellite. Other application specific features are the visibility conditions, the weather, the capability of the different instruments on board of the satellite, the transmission rates, the costs associated with using different communication devices, etc. All of these make satellite scheduling a very complex domain. We structure the presentation of the research done in this domain in a progression, starting with studies published for simple versions of the problem.

Burrowbridge(1999) studies one of the most simple versions of satellite scheduling: scheduling the communications between ground stations and low-altitude satellites, with the objective of maximizing the number of scheduled tasks. The visibility windows for low altitude satellites have approximately the same duration as the duration of the requested communication; in other words, for this problem, each request has a fixed start time, a fixed end time, and a duration equal to the difference between the start and end time. The well-known *greedy activity-selector* algorithm [CLR90] is used to schedule the requests since it yields a solution with the optimal (maximal) number of scheduled tasks.

A simple one-resource satellite scheduling problem is solved by Pemberton(2000).

The resource is the camera on the satellite; the requests have fixed start times and fixed durations and a priority value is associated with each request. The objective is to schedule the requests such that the sum of the priorities of the scheduled requests is maximized. Basically the difference between the scheduling problem solved by Pemberton and the low-altitude satellite scheduling investigated by Burrowbridge is the presence of priorities associated with the requests. Pemberton makes the point that applying constrained programming techniques to this problem can result in poor performance given the large number of requests to be scheduled. Therefore he proposes a "priority segmentation algorithm" which is a hybrid algorithm combining a greedy approach with branch-and-bound. The requests sorted in the order of their priority are divided into groups of n requests. Starting with the highest priority request group, for each group of n requests, an optimal schedule is identified (using branch-and-bound and constraint propagation) where the previously scheduled requests are considered as locked (cannot be moved anymore). The value of n is varied starting from 1 to 32. Pemberton applies his algorithm to one-resource problems with 100 and 1000 tasks. However, for the 100 request problems, exact optimal solutions could be obtained using branch-and-bound algorithms such as the one described by Baptiste et al. [BPP98] (see Section 4.1 for a more detailed description of this algorithm). Baptiste et al. report good algorithm performance when solving one machine scheduling problems with up to 100 requests. Also, in a review of Pemberton's paper, Verfaillie [Ver00] mentions that a simple dynamic programming algorithm can outperform the priority segmentation algorithm by optimally solving Pemberton's problems.

Wolfe et al.(2000) define a more complex one-resource problem, the "window-constrained packing problem" (WCP), which specifies for each request the earliest start time, latest finish time and the minimum and maximum duration. The objective function for the WCP is also different, based on the idea that the middle of the time win-

dow in which the request can be scheduled is to be preferred over positions toward the edges. The objective function combines request priority with the position and duration of the scheduled request in its required window and the number of requests scheduled. Two greedy heuristic approaches and a genetic algorithm are implemented to solve the window-constrained packing problem. The genetic algorithm uses job permutations as the population and is shown to improve on the results obtained with the two greedy heuristics for randomly generated problems.

A different scheduling approach is proposed for the more complex problem of scheduling communications between satellites and the Deep Space Network (DSN) 26-meter subnet [GC96] (which is a collection of antennas in California, Australia and Spain). For each satellite, a set of project requirements specifies the minimum and maximum number of communication events, and the minimum and maximum duration and time intervals between communications. Scheduling is done weekly; a problem defines the set of satellites to be scheduled, specific constraints on the satellites, and visibilities for each satellite and antenna. The solution should specify a maximum feasible set of time intervals for satellite-antenna communications such that all the project requirements are satisfied. The problem is usually oversubscribed; the goal is to obtain "adequate" solutions quickly (or prove infeasibility) in a short time, such that "what if" reasoning is facilitated in the negotiation with the customers. Multiple satellites, multiple antennas, as well as various constraints make this problem much more complex than the ones previously presented. Gratch et al.[GC96] propose a heuristic scheduling approach (LR-26) using Lagrangian relaxation combined with constraint propagation search. Basically the constraints involving more than one antenna are relaxed, and each antenna is scheduled independently. The relaxed constraints are weighted and added into the objective function. Lagrangian relaxation attempts to find a global solution by adjusting the set of weights; if Lagrangian relaxation fails to identify a solution, constraint

propagation search is used. An adaptive version of the scheduler is also described, where the scheduler is provided with a variety of heuristic methods and it chooses a particular combination that works well for a set of given problems. Experimental data show that the adaptive scheduler is learning strategies that significantly outperform randomly selected strategies and results in improved problem solving performance over the original LR-26 approach.

While all the satellite scheduling problems presented so far schedule communications from one ground station to satellites (the resource is represented by the ground station), Verfaillie et al. [VLS96] schedule requests for three instruments on a satellite (the three instruments are the resources here). Their version of the satellite scheduling problem emerged from scheduling requests for the *Spot5* satellite. The problem has been described in [VLS96, BVA+96]: scheduling a set of images, which specify priorities and which can be taken from any one of the three instruments on the satellite. Constraints are imposed on the transition times between successive images on the same instrument, and on dataflow through the satellite telemetry. The goal is to select a subset of images such that the sum of the priorities of the images is maximized. Verfaillie et al. [VLS96] introduce an algorithm called "Russian Doll Search", which is a modified branch-and-bound algorithm based on the existence of a fixed variable ordering. The algorithm iteratively solves n subproblems: first the subproblem represented by the last variable, second the subproblem containing only the last two variables, and so on. For each subproblem solved, the result is recorded and used to bound the search when solving the next subproblem. The Russian Doll Search is compared to a depth first branch-and-bound and to two approximate methods: a greedy algorithm and tabu search[1]. Russian Doll Search provides optimal solutions, faster than the depth first al-

[1]Tabu search was first defined by Glover in 1986 [Glo86]; it is a heuristic method that uses memory

gorithm and is therefore suited for small and medium size problems. It fails on large size problems, where more complex constraints are also present, and thus approximate methods are more appropriate for these problems.

A more recent, more complex version of Verfaillie et al.'s problem has been studied by Lemaître et al.(2000): scheduling the set of images for new Agile Earth observing satellites. While satellites like Spot5 have only one degree of freedom, the Agile satellites have three degrees of freedom, and therefore more opportunities are present when scheduling the images. Again, each image specification includes a priority, the earliest start time, latest finish time and duration. Additional constraints are present such as: minimal time between two successive acquisitions, images that need to be taken twice from different time windows, and images that need to always be scheduled. Branch-and-bound using constraint propagation is combined with heuristics to control the combinatorial explosion (such as dropping the low priority images from the schedule from the start, or imposing a certain order on the sequence of the images to be scheduled). Next descent local search is also used to solve the problem, where a image is added to or removed from the current feasible sequence of image; several heuristics guide the choice of the image and its position in the sequence. Seven problems are used to compare the performances of the two algorithms. Local search outperforms the systematic search when a limit of 5 minutes of CPU time is imposed on the systematic search and local search is allowed to perform 100 executions of two minutes CPU time each. Obviously, the systematic method is expensive, and the approximate method seems to result in better performance here.

Potter and Gasch [PG98] examine the problem of image scheduling for NASA's Landsat 7 satellite. Given the constraints on the spacecraft equipment, approximately

---

to prevent search from cycling and getting trapped in locally optimal regions of the space.

530 images can be acquired per day. The images have priorities associated with them; the priorities are dynamically adjusted based on factors such as cloud cover and the age of requests. The objective is to maximize the coverage of Earth's surface and the value of the acquired images, while also maximizing utilization of available resources. Their algorithm greedily schedules all the images encountered given the orbit of the satellite, no matter what their priorities are, while resources are available. When a resource is filled, the scheduler attempts to make room for new requests by unscheduling some of the requests previously scheduled, based on certain rules. An interesting feature of the algorithm is that it forces good images to be scheduled by maintaining a minimum load greater than zero for one of the resources (the solid state recorder or SSR).

Advancing toward the goal of automated planning and scheduling for complex real-world satellite operations, researchers at NASA built a complex environment called ASPEN. The ASPEN (A Scheduling and Planning Environment) framework [CRK+00] is used to model and solve real-world applications involving Earth observing satellites. ASPEN implements features commonly encountered when modeling planning and scheduling problems; constructive and repair-based algorithms are implemented for finding a valid schedule given a set of goals. As a planning and scheduling framework, ASPEN is more focused on the optimization of the command sequence [CRK+00] (the planning aspect) than on the optimization of observation preferences. Two satellite applications in which ASPEN was used are the EO-1 and CX-1 missions. EO-1 [SGY+98] is an Earth imaging satellite; it can perform only four data acquisitions per day. CX-1 [ECB+01] is a small Earth orbiting satellite. The goal of the CX-1 scheduling problems is to maximize the amount of atmospheric data downlinked. Several criteria are considered when computing the value of a schedule, such as: number of violated conflicts, amount of data downlinked, power usage, memory usage, etc.

Globus et al. [GCLM02, GCLP03, GCLP04] compared a genetic algorithm, sim-

ulated annealing (SA) and hill climbing on a complex, synthetic form of the satellite scheduling problem. They generate ten synthetic problems, using various parameters such as: the number of satellites (one, two or three), the capacity of a solid state recorder (SSR) which stores the images until they can be sent to a particular ground station, the number of observations ($2100 * n$, where n is the number of satellites), the priority of the observations. The objective function is a weighted sum of the priorities of the unscheduled requests, the mean time spent slewing and the mean image angle (which determines the resolution of the images). The schedules are represented by a permutation and a simple, greedy deterministic scheduler is used to assign times and resources to each request. The results of the study show that simulated annealing performs best for this domain. Also, for hill climbing, the best operator is a "temperature dependent swap" that performs a large number of multiple moves in the beginning of the search and a smaller number of multiple moves in the end.

Frank et al.(2001) propose a new approach to solving the problem of scheduling fleets of Earth observing satellites. It combines a constraint-based planner with a stochastic greedy search algorithm: the greedy search selects the next task to be scheduled, and the planner propagates the constraints and adds to the schedule all the activities required by the scheduled task, such as setup and postprocessing. The greedy search is based on Bresina's HBSS algorithm [Bre96] (also see Section 2.2). A heuristic approach is used to choose the next task to be scheduled and the start time for the task, based on contention measures. This approach is one of the most complex applied to the satellite scheduling problem; it models all the satellite resources and communication resources using a constraint-based language.

2.2 Telescope Scheduling

Scheduling observations for a telescope is typically oversubscribed. Astronomers spend a lot of time observing objects as they set in the west. The windows of opportunity to observe such objects are usually short, and there are more observation requests than can be accommodated by the telescope. The requests specify the beginning of the visibility window for the object as the earliest start time. The end of the visibility window is the latest finish time, and priorities are also assigned to the observations. Setup times are usually needed to point the telescope from one object to another. The activities that need to be scheduled usually specify constraints of precedence, ordering, minimum and maximum time separation, preemption, and repetition. An objective function is used to decide which observations to schedule, based on the priorities of the observations, minimizing the setup times and maximizing the quality of the observations.

Aspects of telescope scheduling make this problem very similar to satellite scheduling. For example, in both telescope and satellite scheduling, windows of visibility are defined; also, setup times are associated with the viewing instruments (telescope) and antennas and instruments on the satellite (satellite scheduling) and the weather impacts on the visibility conditions. In most cases, there are more requests than can be scheduled. We present three examples of oversubscribed telescope scheduling problems. Two refer to the Hubble Space Telescope (one is a simplified version of the problem, the other describes a complex scheduler built for the telescope). The third example refers to a telescope scheduling problem investigated by Bresina (1998).

When scheduling observations for the Hubble Space Telescope, demands for observations exceed by far the capabilities of the telescope. The candidate observations need to be scheduled on one of the six viewing instruments in a particular configuration. The observations have windows of visibility and durations. Setup times are associated with pointing the viewing instruments toward the object observed and with reconfiguring the

instruments. Power constraints impose a limit on the number of viewing instruments that can be active at the same time. Additionally, precedence constraints and priorities may be specified for the observations.

A simplified version of the problem is described by Smith et al.(1992). It is assumed that only one instrument can be active at any time; the setup time needed to reconfigure the instrument to be used is assumed to depend only on the previously used instrument. Two conflicting objectives are identified: maximizing the resource utilization (in this case telescope utilization) and rejecting as few candidate observations as possible. Two greedy scheduling heuristics are described, each addressing one of the two objectives. The "nearest neighbor look-ahead" heuristic minimizes the idle times on the telescope by choosing to schedule candidate observations with early start times; lookahead is performed to make sure that the candidate chosen is the least disruptive (resulting in the least number of observations that cannot be scheduled anymore). The "most-constrained first" heuristic maximizes the number of observations scheduled by choosing to schedule the observation with the fewest possible start times. In order to address the dual nature of the objective function, a composite strategy is then implemented. The composite strategy uses the most-constrained first heuristic to select the next observation scheduled; however, for the cases where the number of start times available is similar for all of the unscheduled candidates, the nearest neighbor look-ahead heuristic is used instead. On a set of seven randomly generated problems, the composite strategy manages to effectively combine the two heuristics.

Next, we present the complex scheduler built to solve the real problem of scheduling requests for the Hubble Space Telescope. SPIKE [JM94] is a scheduler that was built for the Hubble Space Telescope scheduling problem. SPIKE has been operational since October 1989; it can handle the broad complexity of telescope scheduling. The constraints are classified into feasibility constraints (hard constraints) and preference constraints,

which have a weight value associated. The temporal constraints have higher weights than resource constraints. The activities have priorities, and possible time assignments have preferences associated with them. Suitability functions are used to combine feasibility and preference constraints for the observations. In SPIKE, repair-based methods are used to build the schedule. Repair based methods identify conflicts in the schedule and try to repair them by moving tasks around. Such methods are very suitable for oversubscribed scheduling and have been widely used in scheduling space operations: ASPEN (described in the previous section) employs iterative repair. The repair-based method implemented in SPIKE is called "multistart stochastic repair". Three main steps compose the multistart stochastic repair technique. The three steps are repeated until a time limit is reached and the best schedule is selected:

- First, the observations are heuristically assigned such that the most constrained ones are scheduled first and the number of conflicts is minimized.
- Then, repair heuristics are applied: hill climbing is used to move the activities with most conflicts to a time where the number of conflicts is minimized.
- Finally, the conflicting activities are removed from the schedule; lower priority, higher number of violated constraints, lower preference time assignment activities are preferred for removal. If there are gaps in the schedule, the unscheduled tasks are used to fill the gaps (using a best-first approach).

When iterating through these three steps, the initial assignment and the repair heuristic can vary. The quality of the schedule is dictated by the number of observations scheduled, the total time scheduled, and the sum of preference values associated with the scheduled observations.

The third example of telescope scheduling [Bre98] is the problem of scheduling observations for the "T3" automatic photoelectric telescope at Fairborn Observatory in

Arizona (a telescope that gathers photon count information describing the brightness of different objects in the sky). The observations have priorities; also, some observations need to be repeated during the night. "Enablement intervals" are defined as the time intervals within which the observations can be executed and are determined by specific constraints such as the darkness and the distance to the moon. These enablement intervals are in fact the equivalent of the visibility windows encountered in satellite scheduling. The telescope is equipped with a controller that uses dispatch heuristics to decide which observations to schedule next. The dispatch heuristics work well for nights with a balanced load of required observations. However, there are many nights for which not all the observations required can be executed; for such nights the problem of telescope scheduling is oversubscribed, and the dispatch heuristic performs poorly. The objective function is defined as a weighted sum of three attributes of the final schedule, with the most important attribute minimizing the sum of the priorities of the missed observations; the other two attributes minimize the airmass and the telescope slewing respectively.

Bresina proposes a heuristic to improve the quality of observation scheduling, based on the dispatch heuristics built into the telescope controller. The heuristic is embedded into what was a new search technique: HBSS (Heuristic-Biased Stochastic Sampling) (Bresina published an initial description of HBSS in 1996 [Bre96]). HBSS combines heuristic guidance with random sampling; it employs a bias function as a method of balancing the exploration of the search space and the exploitation of the states ranked best by a heuristic. Basically, HBSS incrementally builds a solution; at each step the next move is chosen with a probability based on the bias function. When a strong bias function is used (for example, exponential or high-degree polynomial), the search mostly follows the heuristic: high probability is associated with the moves ranked as the best by the heuristic (in the extreme, greedy search behavior can be obtained). With a weak bias (logarithmic or linear), the search explores more of the space, in the extreme, pro-

ducing random-sampling behavior. A set of problem instances over the 251 day interval of Julian Dates [2450400,2450650] were used to test the new search algorithm. For these problems, experimental results show that HBSS outperforms the dispatch heuristic built into the telescope controller.

Since its publication in 1996, HBSS has been widely referenced and used. In the scheduling domain, Frank(2001) proposes the use of the HBSS search technique in solving the problem of scheduling Earth observing satellites. Oddi et al.(1997) extend HBSS to define the bias dynamically, based on how well the heuristic can discriminate between the alternatives. They show the increased performance of the new algorithm when solving job-shop problems with non-relaxable deadlines and complex metric constraints. Also, we used HBSS to solve flow-shop problems with various amounts of structure; HBSS was one of the best algorithms we implemented for these problems [WBHW99].

2.3 Air Mobility Command (AMC) Airlift Scheduling

AMC scheduling refers to the efficient allocation of aircraft and crews to transportation missions. The problem is oversubscribed; the objective is to assign delivery missions to wings such that higher priority missions take precedence over lower priority missions. Kramer and Smith [KS03, KS04] adopt an iterative repair approach in the context of AMC scheduling. Their algorithm starts by creating a greedy initial schedule based on mission priority. Repair search is then performed using a "task-swapping procedure": in an attempt to make room for one or more of the initially unscheduled tasks, if feasible, some of the tasks in the schedule are retracted and reassigned. To decide which tasks need to be retracted, three retraction heuristics are defined. Empirical results show one of these heuristics, "Max-Flexibility" to be the strongest performer. The flexibility of a task is defined as the average duration of the task on the alternative resources divided by the the length of the time window available to execute the task. The retracted tasks are

then rescheduled such that the most constrained ones are scheduled first, at the earliest feasible time.

In [KS05], the task-swapping procedure is extended by using a new heuristic, "max-availability" to decide where to reschedule the retracted tasks (instead of rescheduling the retracted tasks at the earliest feasible time). The new heuristic computes the resource contention over possible task assignments and chooses the assignment where resource availability is maximal. Max-availability dramatically improves the results of the task swapping procedure; also, it increases the stability of the schedules by minimizing the amount of change during task swapping.

2.4 Scheduling Shuttle Payload Operations

Rabideau et al.(1999) describe the DATA-CHASER Automated Planner and Scheduler (DCAPS) used to schedule the DATA-CHASER space shuttle payload for a solar science experiment performed in 1997. The problem consists of finding a sequence of activities related to instrument observations, data storage, communication and power systems with the goal of increasing science return (number of measurements taken and downlinked) while satisfying all the spacecraft constraints. Both planning and scheduling are employed to solve the problem; planning determines the activities needed to be performed such that specific scientific goals are satisfied, while scheduling assigns start times to these activities such that the constraints (imposed by limited availability of the resources, but also the temporal constraints) are satisfied.

Scheduling is performed in three steps:

1. An initial schedule is generated using domain specific knowledge.

2. The schedule is repaired to remove conflicts; frequently the resources are over-subscribed but other conflicts may also be present. The scheduler uses iterative repair to remove the conflicts in the schedule. Iterative repair [ZDD94] mentioned

earlier in this chapter iteratively selects a conflict in the schedule and adds, moves or deletes an activity in an attempt to solve the conflict. For example, suppose a resource is oversubscribed at time t: if an activity exists which replenishes the resource, such an activity can be added to the schedule. Also, the scheduler can move one of the activities requiring the resource at that particular time t to a different time in the schedule; in the worst case, an activity can be removed.

3. The oversubscribed schedules are optimized using a special "packing" algorithm, based on the doubleback algorithm described by Crawford(1996). The packing algorithm performs a sweep over the scheduled activities starting with the first one and moves them to their earliest possible start time; the idea is that by doing so, more room will be available toward the end of the schedule for the activities for which the resource requirements could not be satisfied in the initial schedule.

Rabideau et al. note that the use of the DATA-CHASER Automated Planner and Scheduler resulted in "significant quantitative improvements in operations efficiency" [RCWM99] over the manual generation of the schedules.

2.5 Scheduling F-14 Flight Simulators

Syswerda [Sys91] describes the problem of scheduling access to flight simulator resources in a laboratory: F-14 jet fighters, together with the radar and the support equipment are the resources to be scheduled. The tasks require extensive setup of the equipment; the optimization of the setup is important since it can take from one to two hours. Precedence relations specify that some tasks cannot be started before the completion of other tasks. Preference and priority are also specified for the tasks. When not all the tasks can be scheduled, the ones with lower priority are dropped from the schedule. The objective function is the sum of the weighted priorities for the scheduled tasks. The priorities are weighted by a 1.5 factor if for the corresponding task preference constraints

are violated; otherwise the weight factor is 2. A genetic algorithm is used to solve problems with 90 tasks and up to 30 resources.

2.6 Heterogeneous Computing Systems

Heterogeneous computing systems are ensembles of interconnected computers of various computational capabilities [Bra01]. Applications defined as collections of communicating tasks with various computational requirements need to be executed on these computers; obviously the goal is to achieve increased application performance. The tasks to be scheduled have priority values; the tasks also have arrival and due dates. Multiple versions are defined for the tasks, where only one version needs to be executed. User preferences are defined for each version. Lower preference versions of a task have reduced requirements. When lower preference versions are scheduled, more tasks can be accommodated. Precedence constraints can also be imposed on tasks (modeling the idea that a task cannot start until it receives all its input data, which is possibly produced by other tasks). The tasks need to be assigned to the machines, and then on each machine, the tasks need to be ordered. The objective function is defined as the sum of the product of preference and priority for each of the tasks completed before the deadline.

Braun [Bra01] defines two greedy heuristics for scheduling collections of tasks on heterogeneous computing systems. A generational genetic algorithm (GA) as well as a steady-state GA (*Genitor* [Whi89]) are also implemented using a special structure of the chromosome and crossover and mutation operators designed for this chromosome structure. *Genitor* performs the best, however one of the greedy heuristics also results in good performance (the genetic algorithms were seeded with the solution obtained by the greedy heuristic).

2.7 Data Staging

Data staging [TBS+00] is a data management problem for a heterogeneous network of locations. Limited data storage is available at each location, and communication links are available at specified times between the locations. Data items become available in certain data locations, and requests for data items are made from specified locations in the network. The requests include priorities and deadlines. The requests are satisfied by transferring the data from the locations where they are generated to the locations where they are requested; the transfer is made using the communication links available. The communication links have a start time and end time; the duration for transmitting a data item from one location to another through a specified communication link is also specified.

Theys et al.[TBS+00] define three greedy heuristics for scheduling data communications from multiple sources to their destination, based on Dijkstra's multiple source shortest-path algorithm for a weighted, directed graph. The objective function is represented by the sum of the priorities of the satisfied requests.

2.8 Summary of Algorithms Used

As presented so far, a variety of algorithms have been employed to solve oversubscribed scheduling problems. In Table 2.1, we summarize the algorithms developed for the problems in this section. Exact algorithms (first two columns in Table 2.1) always find the optimal solution of the problem; examples of such algorithms are various versions of tree search, such as the Russian Doll Search algorithm used by Verfaillie (1996) to schedule a set of images to be taken from a satellite (see section 2.1). We also included in this category the greedy activity selector algorithm [CLR90]. The applicability of this algorithm to oversubscribed scheduling problems is reduced (it can only solve very simple one-resource problems with fixed start times and end times); however, Burrow-

Problem	Exact Algs.		Approximate Algs.							Hybrid Algs.	
	Act. Sel.	Tree Search	Greedy	GA	Tabu Search	Hill climb.	Iter. repair	HBSS	SA	Tree Search & Lagrangian Relaxation	Tree Search & Greedy Heur.
Burrowbridge(1999)	X										
Pemberton(2000)											X
Wolfe and Sorensen(2000)			X	X							
Verfaillie et al.(1996)		X	X		X						
Lemaître et al.(2000)						X					X
Chien et al.(2000)							X				
Potter and Gasch(1998)			X								
Gratch and Chien(1996)										X	
Frank et al.(2001)								X			
Globus et al.(2004)				X		X			X		
AFIT research (1993-1996)			X	X			X				
Kramer and Smith(2005)			X				X				
Smith and Pathak(1992)			X								
Johnston and Miller(1994)							X				
Bresina(1998)								X			
Rabideau et al.(1999)							X				
Syswerda(1991)				X							
Braun(2001)			X	X							
Theys et al.(2000)			X								

Table 2.1: Summary of the algorithms used to solve the oversubscribed scheduling problems described in this section. SA stands for simulated annealing.

bridge(1999) used it to optimally schedule low-altitude satellite requests.

Most of the solutions for oversubscribed scheduling problems use approximate algorithms: greedy and search algorithms, such as hill climbing, genetic algorithms, tabu search, iterative repair, HBSS or simulated annealing (SA). Greedy algorithms (the fourth column in Table 2.1) are very frequently used. Obviously if domain specific knowledge can be translated into a greedy algorithm, solutions for the problems can be obtained very quickly, and many times such solutions prove to be competitive with the ones produced by more time-consuming methods (for example, see [Bra01]). Genetic algorithms (the fifth column in the table) have been successfully used on a variety of scheduling applications; many researchers report great success in using genetic algorithms for oversubscribed problems as well. Iterative repair (the eighth column in the table) is also successfully used to solve oversubscribed scheduling problems. Fi-

nally, hybrid algorithms (last two columns in the table) combine exact and approximate techniques (for example, tree search can be used to solve subproblems); like the exact algorithms, the hybrid ones are not very frequently used.

Most of the research overviewed in this section suggests that approximate algorithms are better suited to solve complex, large size oversubscribed scheduling problems than the exact or hybrid algorithms. For large size problems, Verfaillie(1996) and Lemaître (2000) observed a decline in the performance of the Russian Doll Search and the hybrid heuristic combined with a tree search algorithm respectively, noting that approximate methods (such as hill climbing) perform better for such problems. Also, research at AFIT [Par94] has shown a genetic algorithm to outperform a hybrid of exact methods and heuristic search when solving the satellite range scheduling problem. Pemberton(2000) successfully uses a hybrid algorithm, but he solves very simple problems. Burrowbridge also solves a very simple version of satellite scheduling. The hybrid algorithms, while not widely used, show some promising results when scheduling communications between satellites and the DSN (Deep Space Network) 26-meter subnet [GC96].

Exact tree search algorithms using constraint propagation have been proposed for overconstrained problems (overconstrained problems are problems for which not all the constraints can be satisfied) [FW92]. Oversubscribed scheduling problems can be considered a subclass of overconstrained problems. In particular, for oversubscribed scheduling problems, the constraints which cannot be satisfied are always resource constraints: the resources available are insufficient and cannot satisfy all the requests. Freuder et al.(1992) propose branch-and-bound algorithms to solve what they define as "partial constraint satisfaction problems": a solution satisfies only a subset of the constraints because the problem is overconstrained and not all the constraints can be satisfied. For oversubscribed scheduling problems, the constraints that are conflicting

are related to the capacity of the resources in the problem and the solution cannot drop constraints on resource capacity. Therefore some of the tasks will not appear in the final schedule, which in terms of partial constraint satisfaction translates to dropping all the constraints corresponding to these tasks. Baptiste et al.(1998) describe a branch-and-bound algorithm based on constraint propagation and use it to maximize the number of scheduled jobs for an overloaded[2] one-machine scheduling problem. However, real-world oversubscribed scheduling problems are usually more complex; the general opinion seems to be that for such problems exact branch-and-bound methods are very inefficient, because of the explosion of the search space [Pem00, FJMS01, BVA+96].

[2]Baptiste uses the word "overloaded" instead of oversubscribed; we decided to use the "oversubscribed" term because it appears more frequently in the scheduling literature.

Chapter 3

AFSCN Scheduling

AFSCN scheduling possesses several features that distinguish it from well studied scheduling problems such as job-shop (JSP) and yet are common in many other real applications. Unlike JSP, not all the tasks can be scheduled with the available resources; AFSCN scheduling is an oversubscribed application. Similarly to any oversubscribed scheduling application (see Chapter 2), the objective functions defined for AFSCN scheduling characterize the degree of request satisfaction. We consider two objective functions: minimizing the number of unscheduled tasks and minimizing the total overlap between tasks in an infeasible schedule.

The tasks in AFSCN scheduling are requests to reserve an antenna for a specified duration with the purpose of communicating with a certain satellite. Because of the visibility requirements between the satellites and the antennas, AFSCN scheduling specifies windows of visibility during which the task durations need to be scheduled. The visibility windows are a common feature of satellite scheduling problems in general (Section 2.1), and are also present in telescope scheduling (Section 2.2). For each task in AFSCN scheduling, multiple alternative resources are specified. The alternative resources are also present in other real-world scheduling applications, such as AMC airlift scheduling (Section 2.3).

In this chapter we give a general overview of the AFSCN scheduling domain. We

start with a description of the problem. Next, we summarize previous efforts in solving the AFSCN scheduling problem from other research groups; most of the previous work has been performed at the Air Force Institute of Technology (AFIT). Finally, we present an NP-completeness proof for the decision version of AFSCN scheduling with minimizing the number of unscheduled requests. Also, we show that a simplified version of the problem can be optimally solved in polynomial time.

3.1 Problem Description

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[1]. Figure 3.1 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 their target satellite 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 (between 10 and 25 minutes) and have the length of the visibility window equal to their duration. High altitude tasks specify requests for high altitude satellites; the durations for these requests are more varied and usually longer (anywhere between 10 and 580 minutes). The lengths of

[1]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.

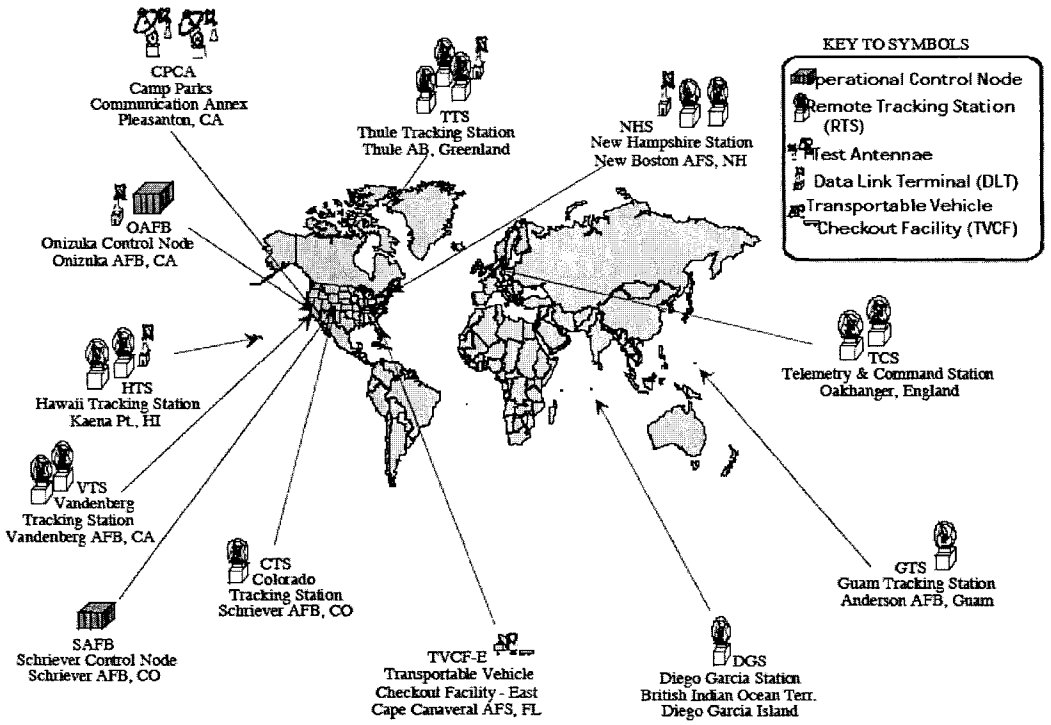


Figure 3.1: Map of the current AFSCN network including tracking stations, control and relay. The figure was produced for U.S.A. Space and Missile Systems Center (SMC).

visibility windows for high altitude tasks range from equal to the length of the task to longer than the task by up to 100 minutes.

Approximately 500 requests[2] 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[3].

---

[2]In the problems we have seen, from March 2002 and April and May 2003, between 426 and 483 requests are specified. Older problems, from October 1992 specify between 297 and 322 requests.

[3]We learned this through personal communication with Brian Bayless and William Szary from the Schriever Air Force Base.

From real problem data[4], 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 and various comments. We do not know enough about the data format (DEFT) to extract such information; therefore, we do not incorporate this in our problem specification. In previous studies of the problem, AFIT researchers Schalck [Sch93] and Gooley [Goo93] used different procedures in processing the DEFT files, resulting in different requests to be scheduled. Later, Parish [Par94] used procedures of data processing similar to Schalck's. We used Schalck's Pascal code published in [Sch93] to extract the problem definition from the DEFT data. 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. Each task request T_i, $1 \leq i \leq n$, specifies a required processing duration T_i^{Dur}. Each task request also specifies a number of $j \geq 0$ pairs of the form (R_j, T_j^{Win}), each identifying a particular alternative resource (antenna R_j) and time window T_j^{Win} for the task. The duration of the task is the same for all possible alternative resources and it needs to be allocated within the time window; we denote the lower and upper bounds of the time window by $T_j^{Win}(LB)$ and $T_j^{Win}(UB)$, respectively. For each task, only one of the alternative antennas needs to be chosen; also, the tasks can not be preempted once processing is initiated.

---

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

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. While this is not always the case in practice[5], the same assumption was made by the original research on the problem done at 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 phrased differently, 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 carry out AFSCN scheduling[6]. 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

---

[5]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. We do not know exactly what type of antennas are present at each ground station.

[6]We met with several of the schedulers at the Schriever Air Force Base 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.

practice, long maintenance tasks are often deferred until absolutely necessary.

By using an objective function that minimizes the number of request conflicts (which is the number of bumped requests), 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[7], this is the only objective function that has been applied to AFSCN scheduling.

A simple example of a problem instance is presented in Figure 3.2. 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 3.3, we show this optimal schedule, where $R8$ is conflicting with the requests scheduled.

Human schedulers of AFSCN use considerable informal domain specific knowledge about their application. For example, they know that certain requests require access to a limited subset of ground stations and so the choices for scheduling such a request are

---

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

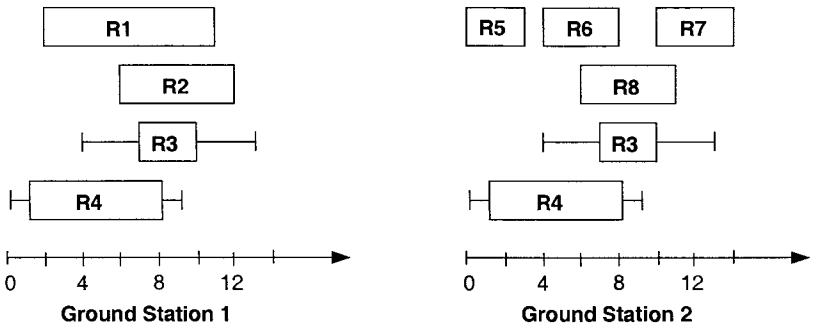


Figure 3.2: Example of a simple problem. Each tracking station has two antennas; the only high-altitude requests are $R3$ and $R4$.

more restricted. They know that the task durations sometimes can be reduced and this can be exploited in the assignment of time slots. On the other hand, no priorities are specified in the AFSCN scheduling problems. This, combined with the fact that human schedulers use domain knowledge to determine that some tasks need to be scheduled before others, results in discrepancies between the schedules we produce by minimizing the number of conflicts and the schedules produced by human schedulers. For example, because we are minimizing the number of bumped requests, requests for large time slots are bumped more often than requests for smaller time slots. Often, the large tasks are related to satellite maintenance that can be put off for some period, but after some time, it becomes critical to fit them into a schedule.

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

Based on discussions with human schedulers, it would appear that a better evaluation criterion is to minimize the sum of overlaps between conflicting tasks in a proposed schedule. In this case, all requests are scheduled before evaluation, and the evaluation measures the degree of conflict, or overlap after all requests are scheduled.

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

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

If minimizing the sum of overlaps is the objective function used, for the problem described above (see Figure 3.2), 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 $A1$, since the sum of overlaps with $R6$ and $R7$ is smaller than the sum of overlaps with $R3$ and $R4$ (see Figure 3.3). 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 $R8$.

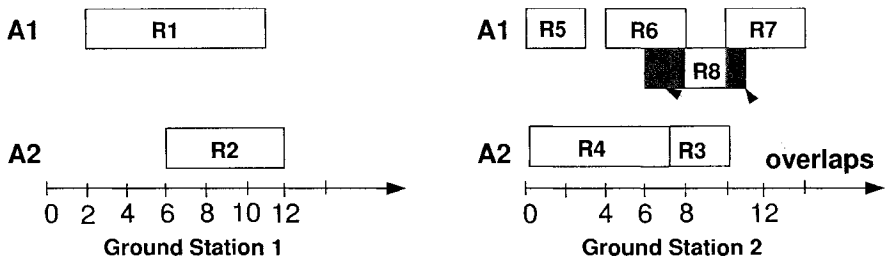


Figure 3.3: Optimizing the sum of overlaps.

In general, the schedule obtained by minimizing the sum of overlaps is different from the optimal bumped schedule with tasks added back into their least overlapping spots. The sum of overlaps for the case when the bumped tasks are added to the optimal bumped schedule is generally greater than the minimum sum of overlaps that can be achieved for the given tasks. As mentioned before, one of the problems with the optimal bumped schedules is that the long tasks (such as maintenance tasks) are not scheduled. When minimizing the total sum of overlaps, these tasks will be placed in the schedule such that the overlap between them and other tasks is minimal. The schedules obtained by minimizing the sum of overlaps look different from the schedules minimizing the number of bumps with the bumped tasks added back in. From discussions with human schedulers, it seems that the schedules minimizing the sum of overlaps look similar to the schedules they build.

3.2 Previous Research

Most of the work previously done on this problem has been published by researchers from the Air Force Institute of Technology (AFIT). Gooley (1993) developed an algorithm based on a combination of mixed-integer programming (MIP) techniques and insertion heuristics, which achieved good overall performance: 91% – 93% of all requests scheduled. Schalck (1993) used MIP techniques based on Gooley's formulation

and scheduled more than 95% of the requests. Parish (1994) tackled the problem using a genetic algorithm called *Genitor*[8], which scheduled roughly 96% of all task requests, out-performing Schalck's MIP approach. All three of these researchers tested their algorithms on a suite consisting of seven problem instances, representing actual AFSCN task request data and visibilities for seven consecutive days from October 12 to 18, 1992 and including 322, 302, 311, 318, 305, 299 and 297 requests, respectively.

Later, 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. His results showed that approximately 90% of the requests can still be scheduled using Schalck's MIP approach if 175 low altitude requests and 250 high altitude requests are present.

3.3 Problem Complexity

For purposes of analysis, we consider various abstractions of the AFSCN scheduling problem. First, we investigate the complexity of a version of the problem where only one resource (antenna) is present. The objective, same as for the original AFSCN scheduling problem, is to maximize the number of scheduled tasks. We refer to this abstraction as Single-Resource AFSCN Scheduling (SiRS) minimizing the number of conflicts.

Although not previously studied, the SiRS with the objective of minimizing the number of conflicts is equivalent to a well-known problem in the machine scheduling literature, denoted $1|r_j|\sum U_j$, in the three-field notation widely used by the scheduling community [GLJK79], where:

---

[8]For a description of *Genitor*, see section 4.4.3.

1 denotes that tasks (jobs) are to be processed on a single machine,

r_j indicates that there a release time is associated with job j, and

$\sum U_j$ denotes the number of tasks not scheduled by their due date.

Using a more detailed notation, each task T_j, $1 \leq j \leq n$ has (1) a release date T_j^{Rel}, (2) a due date T_j^{Due}, and (3) a processing duration T_j^{Dur}. A job is on time if it is scheduled between its release and due date; in this case, $U_j = 0$. Otherwise, the job is late, and $U_j = 1$. The objective is to minimize the number of late jobs, $\sum_{j=1}^{n} U_j$. Concurrency and preemption are not allowed.

$1|r_j|\sum U_j$ scheduling problems are often formulated as decision problems, where the objective is to determine whether a solution exists with L or fewer tasks (i.e., conflicts) completing after their due dates, without violating any other task or problem constraints. The decision version of the $1|r_j|\sum U_j$ scheduling problem is $\mathcal{NP}$-complete [GJ77, GJ79]. This result can formally be used to show that AFSCN scheduling is also $\mathcal{NP}$-complete. Other authors [Goo93, Par94, WS00] have suggested this is true, but have not offered a formal proof.

Theorem 1 *The decision version of AFSCN scheduling with the objective of minimizing the number of conflicts is $\mathcal{NP}$-complete.*

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

Because the set of SiRS with unit-capacity is a subset of AFSCN scheduling problems (minimizing conflicts), the general AFSCN scheduling problem with the objective of minimizing conflicts is $\mathcal{NP}$-Hard.

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

Since AFSCN scheduling is $\mathcal{NP}$-Hard and in the class $\mathcal{NP}$, AFSCN Scheduling is $\mathcal{NP}$-complete. □

AFSCN Scheduling and Minimizing Overlaps: Is It $\mathcal{NP}$-complete?

The problem of minimizing overlaps is equivalent to maximizing the scheduled durations for the tasks, for a version of the problem that would allow shorter durations than the required ones to be scheduled. This version of the problem is in many ways similar to the problem of quality maximization resource constraint project scheduling problem, for which Wang and Smith [WS04] prove it is $\mathcal{NP}$-complete. In light of this, we conjecture that AFSCN Scheduling with the objective of minimizing overlaps is also $\mathcal{NP}$-complete.

3.3.1 Polynomial Subclass of AFSCN Scheduling

While the general problem of minimizing conflicts in AFSCN Scheduling is $\mathcal{NP}$-complete, special subclasses of the problem are polynomial. Burrowbridge [Bur99] considers a simplified version of the SiRS problem where only low-altitude satellites are present and the objective is to maximize the number of scheduled tasks. Due to the orbital dynamics of low-altitude satellites, the task requests in this problem have negligible *slack*; i.e., the window size is equal to the request duration. The well-known *greedy activity-selector* algorithm [CLR90] is used to schedule the requests since it yields a solution with the maximal number of scheduled tasks.

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

We modify the greedy activity-selector algorithm for multiple resource problems: the algorithm still schedules the requests in increasing order of their due date, however it specifies that each request is scheduled on the resource for which the idle time before its start time is the minimum. Minimizing this idle time is critical to proving the optimality of the greedy solution. We call this algorithm $Greedy_{IS}$ (where *IS* stands for Interval Scheduling) and show that it is optimal for scheduling the low-altitude requests. The problem of scheduling the low-altitude requests is equivalent to an interval scheduling problem with k identical machines (for more on interval scheduling, see Bar-Noy et al. [BNGNS02], Spieksma [Spi99], Arkin et al.[AS87]). It has been proven that for the interval scheduling problem the extension of the greedy activity-selector algorithm is optimal; the proofs are based on the equivalence of the interval scheduling problem to

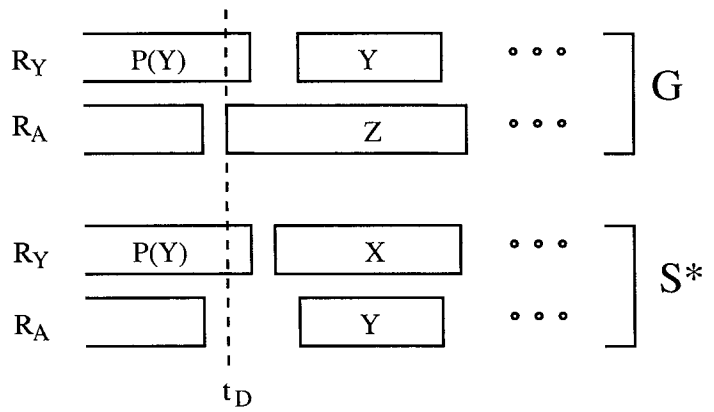


Figure 3.4: The optimal and greedy schedules are identical before time t_D. Note that Y was the next task scheduled in the greedy schedule G. In this case, Y appears on some alternative resource, R_A, in the optimal schedule $S*$, and X appears on resource R_Y. Note that the start time of request Z is irrelevant, since the transformation is performed on schedule $S*$.

the k-colorability of an interval graph [CL95]. We present a new proof, similar to the one in [CLR90] for the one-machine case.

Theorem 2 *When minimizing the number of conflicts, the* $\text{Greedy}_{\text{IS}}$ *algorithm is optimal for scheduling the low-altitude satellite requests for AFSCN Scheduling with identical resources and no slack.*

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

The transformation from $S*$ to G deals with one request at a time. We examine $S*$ starting from time 0 and compare it with G. Let t_D be the first point in time where G differs from S*. Let Y be the next request picked to be scheduled in G after time t_D. This means that Y is the next job scheduled after time t_D in G with the earliest finish time. Let the resource where request Y is scheduled in G be R_Y. Let $P(Y)$ be the

job preceding Y in the greedy schedule G. When constructing the greedy schedule G, Y is chosen to have the earliest due date *and* is placed on the resource that results in minimum idle time. If Y is not scheduled the same in $S*$ and G, then either Y does not appear at all in $S*$ (case 1), or it appears on an alternative resource R_A in $S*$ (case 2, see Figure 3.4).

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

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

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

Note that the proof does not mention what other tasks may be scheduled on G. This is because the proof only needs to convert $S*$ to G one request at a time. However, it is notable that some request Z could be encountered in schedule G at time t_D on resource R_A before either Y or X are encountered in $S*$, which is illustrated in Figure 3.4. This occurs because Z has a start time before Y, but has a finish time after Y and therefore

is **not** the next request scheduled in G. However, the proof is only concerned with the next scheduled request Y. Figure 3.4 also illustrates that it is always possible to swap *all of the subsequent requests* scheduled after the finish time of $P(Y)$ in schedule $S*$ and that the start time of Z is irrelevant. Request Z is a later greedy choice in G; the transformation will deal with Z as it moves through the greedy schedule based on the finish times.

We proved we can optimally schedule the low altitude requests when minimizing the number of conflicts. We conjecture that the minimum overlap schedule could be obtained from the minimum conflict schedule by just adding in the bumped task such that the overlap with the scheduled requests is minimized. If our conjecture holds, minimizing overlaps when only low altitude requests are present can also be optimally solved in polynomial time.

Chapter 4

Algorithms

Previous research on AFSCN scheduling performed at AFIT identified an iterative repair algorithm [Goo93] to perform well, and then found that a genetic algorithm, *Genitor* performs better [Par94] (see Section 3.2). Both iterative repair and genetic algorithms are approximate algorithms and have been successfully used in solving other oversubscribed scheduling problems, as shown in Chapter 4. In fact, the most successful algorithms for oversubscribed scheduling applications are approximate algorithms. Exact algorithms using constraint propagation are usually inefficient and impractical when solving large size real-world oversubscribed scheduling problems.

For AFSCN scheduling, we implemented *Genitor* as well as an improved version of Gooley's algorithm. We also implemented various other approaches; using the algorithm classification introduced in Table 2.1, Chapter 2, we implemented both exact and approximate algorithms for AFSCN scheduling. The algorithms we chose represent a large subset of the algorithms used when solving oversubscribed scheduling problems in general.

From the category of exact algorithms, we implemented $Greedy_{IS}$ (see Section 3.3), an extension of the greedy activity-selector used by Burrowbridge [Bur99] when solving a simplified version of the SiRS problem. We also used in our study a branch-and-bound algorithm implemented by Baptiste et al. (1998) for the $1|r_j|\sum U_j$ machine scheduling

problem.

From the category of approximate algorithms, we implemented the following algorithms:

- three greedy constructive heuristic algorithms: two greedy heuristic algorithms designed specifically for the $1|r_j|\sum U_j$ problem and a greedy heuristic based on "Max-Flexibility" defined by Kramer and Smith [KS03]
- hill-climbing
- Heuristic Biased Stochastic Sampling (HBSS)
- Gooley's iterative repair
- *Genitor*
- Squeaky Wheel Optimization (SWO).

To the best of our knowledge, SWO has not been used before for oversubscribed scheduling. *Genitor*, hill-climbing and SWO all operate in the same problem search space; we introduce in this chapter the methods we used for both solution encoding and evaluation.

Additionally, we developed constructive search algorithms for SiRS based on texture [BDSF97] and slack [SC93] constraint-based scheduling heuristics. We found that texture-based heuristics are effective when the total number of task requests is small (e.g., $n \leq 100$) for SiRS. We considered similar algorithms for the MuRS problem, but were largely unsuccessful; straightforward extensions of constraint-based technologies for multiple resources with alternatives did not appear to be effective.

4.1 Branch-and-Bound Algorithm for SiRS

Baptiste et al. [BPP98] introduce a branch and bound algorithm to solve the $1|r_j|\sum U_j$ problem[1]. The algorithm starts by computing a lower bound on the number of late tasks, v. Branch and bound is then applied to the decision problem of finding a schedule with v late tasks. If no such schedule can be found, v is incremented, and the process is repeated. The jobs are sorted in increasing order of their duration. Three dominance properties are defined to reduce the size of the search space. The first is based on the idea that "it is better to schedule smaller jobs with large time windows than larger jobs with small time windows". The second is to automatically assign a set of jobs if they can all be scheduled in a particular time interval. The third dominance property is to split the problem into two subproblems if they can be solved independently.

When solving the decision problem, at each node in the search tree, the branching scheme selects an unscheduled task and attempts to schedule the task. The choice of the task to be scheduled is made based on a heuristic that prefers small tasks with large time windows over large tasks with tight time windows. The dominance properties together with constraint propagation are applied; then the feasibility of the new one-machine schedule is checked. If the schedule is infeasible, the algorithm backtracks, and the task is considered to be late. Determining the feasibility of the one-machine schedule at each node in the search tree is also $\mathcal{NP}$-hard. However, Baptiste et al. note that for most of the cases a simple heuristic can decide feasibility; if not, a branch-and-bound algorithm is applied.

---

[1]We thank Dr. Philippe Baptiste, researcher at the French National Research Foundation and associate professor at Ecole Polytechnique, for providing executable versions of their branch-and-bound algorithm.

4.2 Constructive Heuristic Algorithms

Constructive heuristics begin with an empty schedule and iteratively add jobs to the schedule using local, myopic decision rules. These heuristics can generate solutions to even large problem instances in sub-second CPU time, but because they typically employ no backtracking, the resulting solutions are generally sub-optimal.

4.2.1 Greedy Constructive Heuristics

Machine scheduling researchers have introduced two greedy constructive heuristics for $1|r_j|\sum U_j$. Given the equivalence of SiRS to $1|r_j|\sum U_j$, we implement and test these greedy heuristics for SiRS.

Dauzère-Pérès [DP95] introduced a greedy heuristic to compute upper bounds for a branch-and-bound algorithm for $1|r_j|\sum U_j$; we denote this heuristic by $Greedy_{DP}$. The principle underlying $Greedy_{DP}$ is to schedule the jobs with little remaining slack immediately, while simultaneously minimizing the length of the partial schedule at each step. Thus, at each step, the job with the earliest due date is scheduled; if the completion time of the job is after its due date, then either one of the previously scheduled jobs or this last job is dropped from the schedule. Also, at each step, one of the jobs dropped from the schedule can replace the last job scheduled if by doing so the length of the partial schedule is minimized. By compressing the partial schedule, more jobs in later stages of the schedule can be accommodated.

Bar-Noy et al.[BNGNS02] describe a greedy heuristic, which we denote by $Greedy_{BN}$, based on a slightly different principle: schedule the job that *finishes* earliest at each step, independent of the job due date. Consequently, $Greedy_{BN}$ can schedule small jobs with significant slack before larger jobs with very little slack.

We also adapted the "Max-Flexibility" [KS03] measure for AFSCN scheduling; we define request flexibility as the duration of the request divided by the average time win-

dow on the possible alternative resources. We use the flexibility measure as a greedy heuristic for constructing solutions. We start by sorting the requests in increasing order of their flexibility. We break ties based on the number of alternative resources available: the requests with fewer alternatives available are scheduled first. For requests with equal flexibilities and number of alternative resources, the earlier request is scheduled first.

4.2.2 Heuristic Biased Stochastic Sampling (HBSS)

HBSS [Bre96] is an incremental construction algorithm in which multiple root-to-leaf paths are stochastically generated. A root-to-leaf path for our domain means starting with an empty schedule and at each step choosing which unscheduled requests to insert in the schedule, until all requests have been considered. Instead of randomly choosing a request to schedule, an acceptance probability is associated with each possible request. This acceptance probability is based on the rank of the request as assigned by the heuristic.

We used the flexibility heuristic defined by Kramer and Smith [KS03] to rank the unscheduled requests. A bias function is applied to the ranks; as noted by Bresina (1996:271), the choice of bias function "reflects the confidence one has in the heuristic's accuracy - the higher the confidence, the stronger the bias." As we'll show in Chapter 5, the flexibility heuristic is an effective greedy heuristic for constructing solutions in AF-SCN scheduling. Therefore we used a relatively strong bias function, an exponential bias. For each rank r, the bias is computed: $bias(r) = e^{-r}$. The probability to select the unscheduled request with rank r is then computed as:

$$P(r) = \frac{bias(r)}{\sum_{i \in \text{Unscheduled}} bias(rank(i))}$$

where Unscheduled represents the set of unscheduled requests.

4.3 Iterative Repair

Iterative repair methods have been successfully used to solve various oversubscribed scheduling problems, e.g., Hubble Space Telescope observations [JM94] and space shuttle payloads [ZDD94, RCWM99]. NASA's ASPEN (A Scheduling and Planning Environment) framework [CRK+00], has been used to model and solve real-world space applications such as scheduling EOS. ASPEN employs both constructive and repair-based methods [SGY+98, ECB+01]. More recently, Kramer et al. [KS03] 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 is a domain appropriate ordering heuristic to guide the repairs. For AFSCN scheduling, Gooley's algorithm [Goo93] uses domain-specific knowledge to implement a repair-based approach.

Gooley's algorithm has two phases. In the first phase, the low altitude requests are scheduled, mainly using mixed-integer programming (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 $Greedy_{IS}$ (see section 3.3). Our version accomplishes the same function as Gooley's first phase but does so with a guarantee that the optimal number of low altitude requests are scheduled. Thus, the result is guaranteed to be equal to or better than Gooley's original algorithm.

In the second phase, the high altitude requests are inserted in the schedule (without rescheduling any of the low altitude requests). The insertion of the high altitude re-

quests in the schedule is based on various domain specific heuristics. After all the high altitude requests have been considered for insertion, an interchange procedure attempts to accommodate the unscheduled requests by rescheduling some of the high altitude requests. For each unscheduled high altitude request, a list of candidate requests for rescheduling is computed (such that after a successful rescheduling operation, the unscheduled request can be placed in the spot initially occupied by such candidates.) A heuristic measure is used to determine which requests from the candidate list should be rescheduled; this favors the tasks with the largest free time blocks in their time windows. The interchange procedure is defined with two levels of recursion and is called "three satellite interchange". For the chosen candidates, the same procedure is applied to identify requests that can be rescheduled. Then the candidate requests are moved from their initial position, and finally the unscheduled requests take the spot of the candidate.

4.4 Local Search Algorithms

In contrast to constructive heuristic algorithms, local search algorithms begin with one or more complete solutions; successive modification are applied to a series of complete solution(s). All the search algorithms we consider encode solutions using a permutation π of the n task request IDs (i.e., $[1..n]$). Permutation based representations are frequently used when solving scheduling problems, e.g., [WSF89, Sys91, WS00, AD03, GCLP03]. A *greedy schedule builder* is used to generate schedules from a permutation of request IDs.

4.4.1 Schedule Builder

Uckun et al. [UBKM93] identify three basic functions for a schedule builder: to locally search for information (such as resource allocation), to enforce domain-specific constraints (for example, time windows for request execution) and to build the actual

schedule. We define a greedy schedule builder for AFSCN scheduling. Our schedule builder considers task requests in the order that they appear in π. For SiRS, the schedule builder attempts to schedule the task request within the specified time window. For the general AFSCN scheduling problem, each task request is assigned to the first available resource (from its list of alternatives) and at the earliest possible starting time[2]. 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, we allow it to overlap with the requests scheduled so far and find the resource and start time for which the overlap is minimized; we assign that resource and start time to the task. The customers do not specify preferences for the alternatives; note that this schedule builder does favor the order in which the alternative resources are specified, even though no preferences are assigned to the alternatives.

Next, we prove that it is always possible to reach an optimal schedule when minimizing the number of conflicts using the permutation representation and the greedy schedule builder. Proving optimality for the schedule builder when minimizing overlaps would be more difficult. It is not clear that the optimal overlap schedule can always be reached using the permutation representation and the greedy schedule builder.

Theorem 3 *When minimizing the number of conflicts, there exists a permutation such that the schedule builder will transform that permutation into an optimal schedule.*

Proof: We offer a constructive proof. We assigned to each of the resources in our domain an integer identifier (ID), starting with one. The proof is based on the fact that the alternative resources for each request are specified in increasing order of their IDs.

---

[2]This is the default strategy for the schedule builder. We'll also define and test other strategies in Chapter 6.

Consider an optimal schedule S minimizing the number of conflicts. We construct the permutation π by scanning the requests assigned to each resource, in increasing order of the request starting times. We start with the first resource ($ID = 1$). The first request in the permutation is the first request scheduled on the first resource, the second request in the permutation is the second request scheduled on the first resource, and so on. After finishing with the requests scheduled on the first resource, we start collecting the requests assigned to the second resource. When all the scheduled tasks are collected, we append to the current permutation the unscheduled (bumped) tasks. Note that all the scheduled tasks in S that are assigned to resource r appear in π before tasks assigned to resource $r + 1$. All tasks that appear earlier in time on resource r appear in π before tasks scheduled later in time on r. All scheduled tasks appear before unscheduled tasks. We use our schedule builder to construct the schedule $S*$ corresponding to π and prove that $S*$ is an optimal schedule as well (same number of bumped tasks). We prove that all the requests in S will be scheduled in $S*$ as well, using strong induction.

Suppose there are N_R resources. When building the schedule $S*$ starting from π, all the requests scheduled on the first resource in S will also be scheduled on the first resource in $S*$. Our schedule builder produces schedules that are front-loaded: all the requests in $S*$ that are scheduled on the first resource are assigned the earliest feasible time. Therefore the starting times for these requests might be different in S and $S*$.

Suppose that we scheduled in $S*$ all the requests that are assigned to resources 1 to $r - 1$ in S, such that there are no requests scheduled yet on resources r, $r + 1$,..., N_R in $S*$. Consider now scheduling the requests appearing on resource r in S. Let X be one such request. The schedule builder will scan the alternative resources specified for X in order. The first feasible assignment will either specify a resource r' such that $r' < r$ (if there is such a spot available in $S*$), or it will find a spot available on resource r in $S*$. Indeed, if t_X is the starting time of X in S (on resource r), there is no request scheduled

yet at time t_X in $S*$. In $S*$, all the requests appearing before X in π have either been assigned to a different resource or to starting times on r at most equal to their starting times in S. Our schedule builder will find feasible starting times for all the requests scheduled on r in S and all these requests will be scheduled either on r or on earlier resources. This concludes the second step of induction.

All the scheduled requests in S will also be scheduled in $S*$, and S is optimal with respect with the number of conflicts. Therefore, the requests not scheduled in S will also be bumped from $S*$. □

Our schedule builder for minimizing the number of conflicts is similar to the one implemented by Globus et al. [GCLP04] for scheduling Earth Observing Satellites. Both start from a permutation of requests, both discard the requests that cannot be scheduled because of previously scheduled ones (conflicts) and both are very simple, greedy and deterministic. As Globus notes [GCLP04], the most important advantage of using the greedy scheduler is that any permutation can be translated into a feasible schedule. This is in contrast with search in the space of schedules, where the search operators either have to maintain feasibility or assign values to infeasible schedules to guide the search.

4.4.2 A Next-Descent Hill-Climbing Algorithm

Perhaps the simplest local search algorithm is a hill-climber, which starts from a randomly generated solution and iteratively moves toward the best neighboring solution. A key component of any hill-climbing algorithm is the move operator. We have selected a domain-independent move operator known as the *shift* operator. Local search algorithms based on the shifting operator have been successfully applied to a number of well-known scheduling problems, for example the permutation flow-shop scheduling problem [Tai90]. From a current solution π, a neighborhood under the shifting operator is defined by considering all $(N-1)^2$ pairs (x, y) of task request ID positions in π, subject to the restriction that $y \neq x-1$. The neighbor π' correspond-

ing to the position pair (x, y) is produced by *shifting* the job at position x into the position y, while leaving all other relative job orders unchanged. If $x < y$, then $\pi' = (\pi(1), ..., \pi(x-1), \pi(x+1), ..., \pi(y), \pi(x), \pi(y+1), ..., \pi(n))$. If $x > y$, then $\pi' = (\pi(1), ..., \pi(y-1), \pi(x), \pi(y), ..., \pi(x-1), \pi(x+1), ..., \pi(n))$.

Given the relatively large neighborhood size, we use the shift operator in conjunction with next-descent (as opposed to steepest-descent) hill-climbing. 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. We call this version of hill-climbing *StructLS*. Our second implementation completely randomizes the neighborhood (choosing by random both x and y); we call this *RandLS*.

4.4.3 The *Genitor* Genetic Algorithm

The *Genitor* [Whi89] genetic algorithm was found to perform well in some early studies [Par94] of AFSCN scheduling. 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 [SMM+91], we used Syswerda's (1991) position-based crossover operator.

As in the hill-climbing algorithm, solutions are encoded as permutations of the task request IDs. In each step of the algorithm, a pair of parent solutions is selected, and a crossover operator is used to generate a single child solution, which then replaces the worst solution in the population. The result is a form of elitism, in which the best individual produced during the search is always maintained in the population. Selection

of parent solutions is based on the rank of their fitness, relative to other solutions in the population. A linear bias is used such that individuals that are above the median fitness have a rank-fitness greater than one and those below the median fitness have a rank-fitness of less than one [Whi89]. We use a population size of 200, a selection bias of 1.5 and no mutation.

Typically, genetic algorithms encode solutions using bit-strings, which enable the use of "standard" crossover operators such as one-point and two-point crossover [Gol89]. Because our solutions are encoded as permutations, a special crossover operator is required to ensure that the recombination of two parent permutations results in a child that (1) inherits good characteristics of both parents and (2) is still a permutation of the n task request IDs. Numerous crossover operators have been proposed for permutations representing scheduling problems.

Syswerda's [Sys91] order crossover and position crossover are different from other permutation crossover operators such as Goldberg's PMX operator [GL85] or Davis' order crossover [Dav85] in that there is no contiguous block which is directly passed to the offspring. Instead, several elements are randomly selected by absolute position. Syswerda's operators are largely used for scheduling applications (e.g., [Sys91, WRWH99, SP91]) and are distinct from the permutation recombination operators that have been developed for the Traveling Salesman Problem. An extensive study of crossover operators for a warehouse/shipping scheduler for the Coors brewery [SMM+91] has shown that Syswerda's position crossover performs better than PMX or Davis' order crossover.

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
```

For our implementation, we randomly choose the number of positions to be selected, such that is is larger than one third of the total number of positions and smaller than two thirds of the total number of positions. Whitley and Nam [WY95] prove that order crossover and position crossover (Syswerda's second operator) are identical in expectation when order crossover selects K position and position crossover selects L-K positions. In effect, order crossover inherits by order first, then fills the remaining slots by position. Position crossover inherits by position first, then fills the remaining slots by their relative order. Use of order or position crossover can still be found in the literature however.

The version of the *Genitor* Genetic Algorithm used here was originally developed for a warehouse scheduling application [WSF89, WSS91], but it has also been applied to problems such as job shop scheduling [VW00]. This was the same version as used by Parish [Par94].

4.4.4 Squeaky Wheel Optimization (SWO)

SWO[3] [JC99] 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 and ranked, 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.

---

[3]SWO has been patented. We wish to thank Matt Ginsberg for allowing us to use it in our studies.

We constructed the initial greedy permutation for *SWO* using the max-flexibility heuristic, as defined in 4.2.1. 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 (we set these values empirically: they seem 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 A Baseline: Random Sampling

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

Chapter 5

Algorithm Performance

As shown in Chapter 2, a wide range of algorithms have been found to perform well for oversubscribed scheduling applications. For AFSCN scheduling, researchers at AFIT found *Genitor* to perform better than MIP approaches and repair-based heuristics on a set of old problems.

The analyses in this chapter are motivated by a number of questions on algorithm performance. First and foremost, does *Genitor* still perform best when compared to other algorithms that were successful for oversubscribed scheduling applications? We compare the performance of a set of algorithms for AFSCN scheduling (as presented in Chapter 4). To understand the complexities of AFSCN scheduling, we start by studying a single-resource version of the problem (SiRS). Algorithms developed for single machine scheduling to maximize the number of scheduled jobs can be directly applied to SiRS. Second, do the results obtained for algorithms for SiRS transfer to the multiple resource version (MuRS)? Third, the data set used in AFIT research is old (from 1992); we have obtained a set of five more recent problems. How do performance results change when solving more recent problems?

The results in this chapter are obtained for both synthetic and real problems. For the synthetic problems, we use problem generators to produce two types of problems: single resource (SiRS) and multiple resource (MuRS); some of these results have also

been published in [BWWH04]. For the real problems, we use two sets of problems: one set from 1992, and the other set from 2002-2003. We have also published results for the real problems in [BHWR04, BWH04].

We present four main algorithm performance results. First, we show that a simple greedy algorithm from the machine scheduling literature performs exceptionally well for the SiRS; however, its performance does not transfer to multiple resources. Second, we identify a simple domain specific heuristic which makes the problems from 1992 easy to solve; however, we show that for the more recent problems, the simple heuristic fails to find best known solutions. Third, we identify a set of three algorithms that perform well for both the synthetic MuRS and the real problems. Fourth, we show that despite surface similarities of these three algorithms, they are very different in the way they apply modifications to the current solution (or solutions). We present evidence that the three algorithms follow very different paths through the search space to reach good solutions.

5.1 Algorithm Performance on Synthetic Problems

In general, evaluation based on real-world benchmarks is desirable. There are, however, two related but distinct drawbacks. First, obtaining real-world data is often difficult and/or costly. Second, by basing evaluation on a small set of real-world instances, we run the risk of over-fitting our algorithms to these instances. On the other hand, we also note that random problem instances can sometimes be much more difficult than real-world problems (e.g., see Taillard [Tai95] or Watson et al.[WBWH02]) and real-world problems may display "structure" that is not found in random problems. In this section we consider the performance of heuristic algorithms for AFSCN scheduling on a range of randomly generated problems, that include realistic as well as more extreme characteristics.

5.1.1 Algorithm Performance for the One Resource Problem (SiRS)

We begin our analysis by considering the SiRS problem with only the objective of minimizing the number of unscheduled requests. We defined the other objective function (minimizing task overlap) in the context of the multiple resource problems, where alternative resources are present. The alternative resources result in a large number of possible assignments for a task which cannot be scheduled: all possible positions on each of the alternative resources need to be considered to find one that minimizes the overlap. We observe that when only one resource is present, the schedules minimizing the sum of task overlap would be rather similar to the optimal bumped schedule (except for some adjustments of start times and maybe a few task swaps that might be needed).

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

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

Based on the above observations, we generate random instances of SiRS problems

using the following procedure for each of the n task requests, which takes as input the maximum slack value *MAXSLACK* allowed for any task request:

1. Sample the processing duration T_i^{Dur} uniformly from the interval $[20, 60]$.
2. Sample the slack T_i^{Slack} uniformly from the interval $[0, MAXSLACK]$.
3. Sample the window start time $T_i^{Win}(LB)$ uniformly from the interval $[1, 1440 - T_i^{Slack} - T_i^{Dur}]$.
4. Let $T_i^{Win}(UB) = T_i^{Win}(LB) + T_i^{Slack} + T_i^{Dur}$.

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

Algorithm Performance on SiRS

We can determine optimal solutions to smaller SiRS problems using the branch and bound algorithm from Baptiste et al.[BPP98][1]. We ran the branch and bound algorithm on each instance in each of our problem sets, imposing a limit for each instance of 1 hour of CPU time on a 1 GHz Pentium III running Windows XP. For all instances with 30 task requests, the algorithm computed the optimal solution within 5 seconds; 40 tasks required at most 1 minute of CPU time. Similarly, the majority of instances with 50 and 60 task requests could be solved in less than 10 minutes of CPU time. We report the number of exceptions in Table 5.1. Several of the larger problems required between

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

10 minutes and 1 hour of CPU for solution, while a small number of instances were never solved (the fourth column in Table 5.1 specifies the number of instances out of 100 which were never solved); the un-solved instances remain insoluble when the CPU limit is raised to 10 hours.

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

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

Table 5.1: The number of problem instances (out of 100) for which the Baptiste branch and bound algorithm (1) required over 10 minutes of CPU time to compute the optimal number of conflicts and (2) failed to find an optimal solution within 1 hour of CPU time.

We now analyze the performance of random sampling, $Greedy_{BN}$, $Greedy_{DP}$, *RandLS*, *StructLS* and *Genitor* on SiRS Problems. Algorithm performance is measured in terms of the total number of bumped tasks, which we denote $|Bumps|$. We can measure the *absolute* performance of a heuristic algorithm by computing the difference between the best solutions found by an algorithm and the optimal number of bumped tasks, denoted by $|Bumps_{opt}|$. In the few instances where Baptiste et al.'s algorithm failed (as

in Table 5.1) we substitute the best solution found by *any* of our heuristic algorithms as the absolute reference point (random sampling never found a higher-quality solution than any of the heuristic algorithms). It could be argued that we should have discarded the instances for which Baptiste's algorithm failed; we note that given the small number of such instances, the change in the values on the graph if we discarded those instances would be very small[2]. In both cases, we denote the difference in the number of bumped tasks by Δ_{best}.

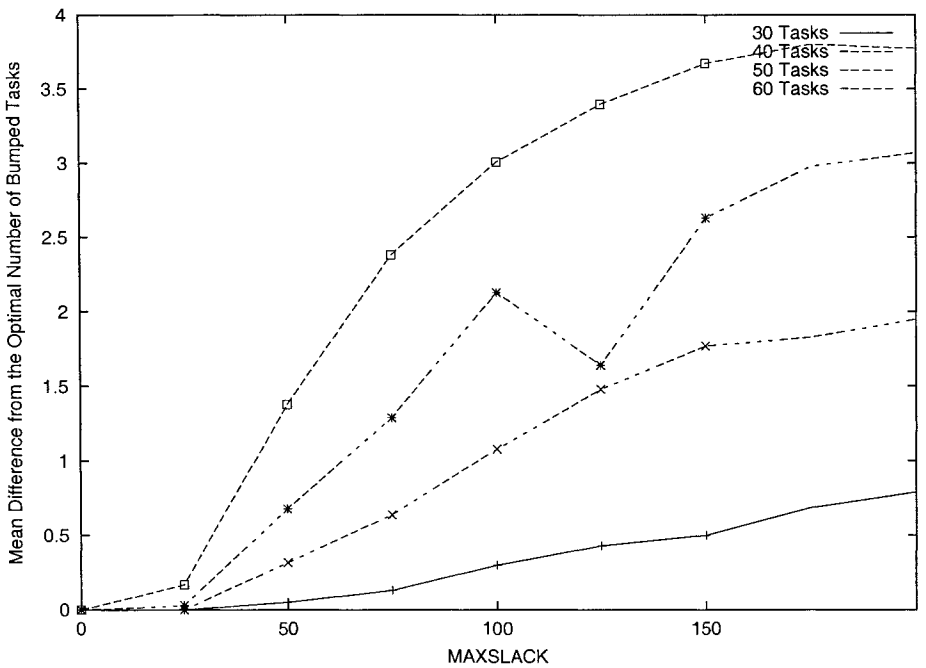


Figure 5.1: The mean difference from the optimal number of bumped tasks $\overline{\Delta_{best}}$ for random sampling.

We first consider the performance of our baseline algorithm, random sampling. We define a single trial as generating 8000 random permutations of the integers $[1..n]$ (where

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

n is the total number of task requests); each permutation is evaluated using the scheduler builder introduced in Section 4.4.1. For each of our SiRS instances, we execute 30 trials; we then take $|Bumps|$ as the minimal number of bumped tasks observed in any of the 30 trials. We report the mean Δ_{best} for random sampling in Figure 5.1. As expected (due to the growth in the size of the search space), the performance of random sampling degrades with increases in the problem size n. However, two qualitative aspects of Figure 5.1 were surprising. First, random sampling generates optimal or near-optimal solutions to test instances for which the *MAXSLACK* value is consistent with the slack of task requests in AFSCN problem instances (e.g., $MAXSLACK \leq 60$). Second, even for instances with unrealistically large *MAXSLACK*, random sampling generates solutions with on average less than 4 more bumped task requests than the optimal number in the *worst* case (in Figure 5.1, the maximum value for the mean difference from the optimal number of bumped tasks is 3.75).

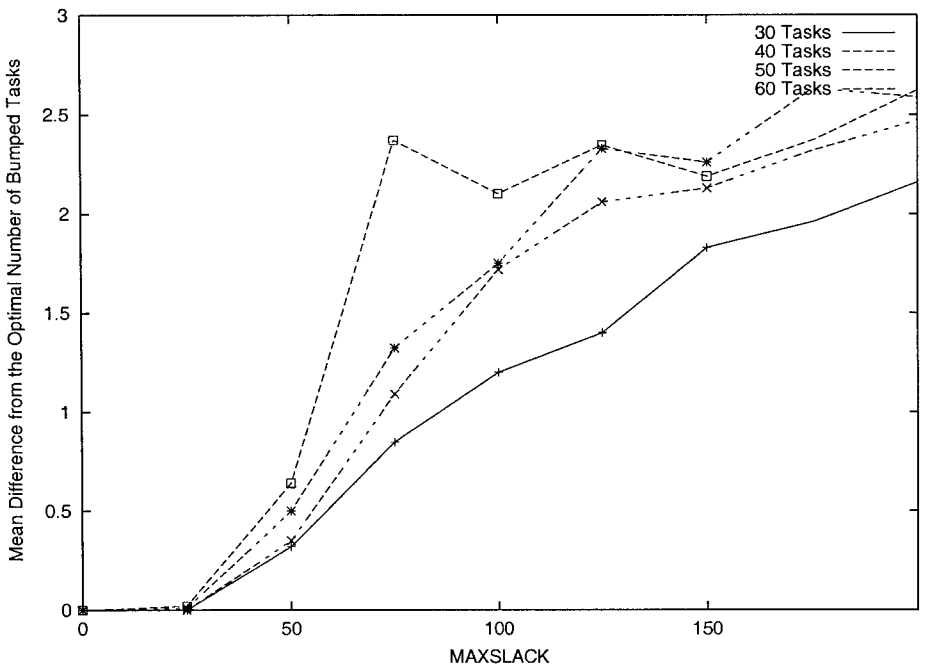


Figure 5.2: The mean difference from the optimal number of bumped tasks $\overline{\Delta_{best}}$ for the constructive heuristic $Greedy_{BN}$.

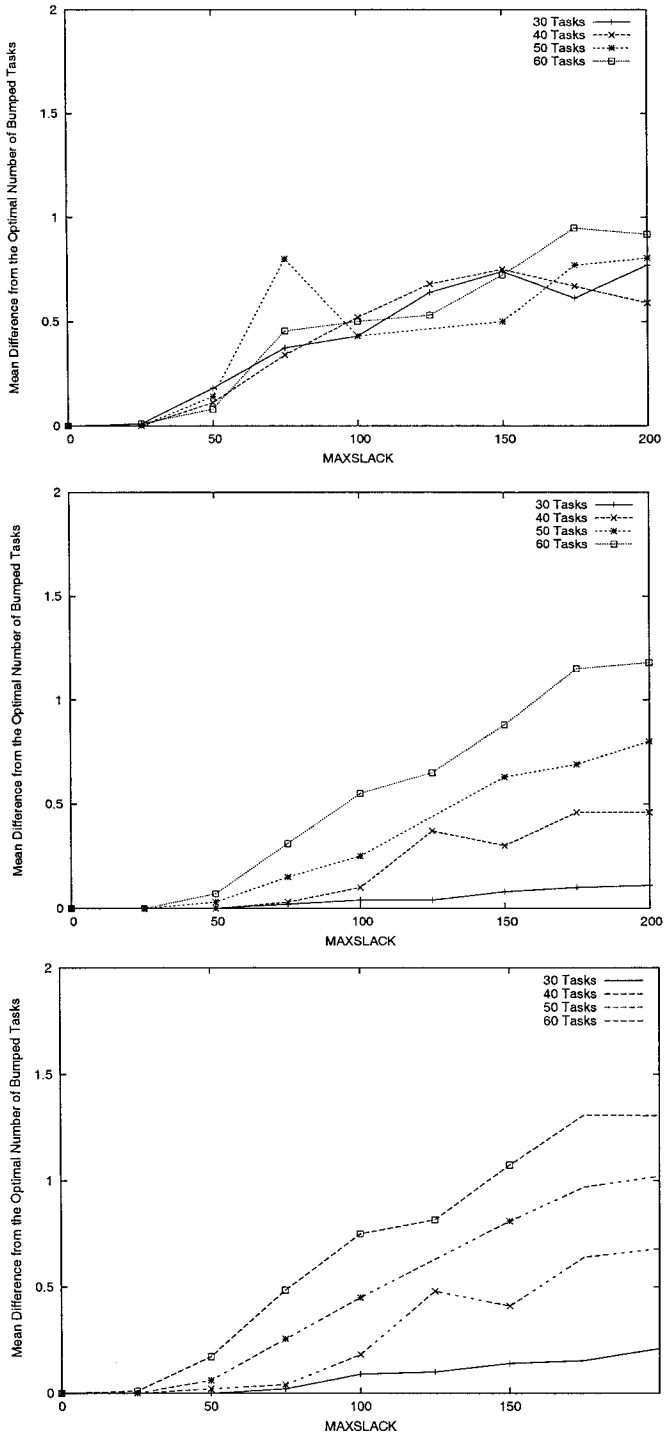


Figure 5.3: The mean difference from the optimal number of bumped tasks $\overline{\Delta_{best}}$ for $Greedy_{DP}$ (top), *RandLS* (middle) and *Genitor* (bottom).

Next, we consider the performance of the two greedy constructive heuristic algorithms for $1|r_j|\sum U_j$. As noted in Section 4.2, both $Greedy_{BN}$ and $Greedy_{DP}$ are largely deterministic; randomness is only applied to tie-break when one or more equally good alternatives are available. Consequently, we compute Δ_{best} for both algorithms using the result of a single trial on a given problem instance; we report the resulting mean Δ_{best} in Figures 5.2 and 5.3, the top graph. $Greedy_{DP}$ consistently outperforms $Greedy_{BN}$ independently of n and *MAXSLACK*. In fact, $Greedy_{DP}$ bumps less than one task request on average, obtaining optimal or near-optimal performance for all of our test instances, and with a trivial amount of computational effort (i.e., less than 1 second of CPU time). For problems with 50 and 60 requests, $Greedy_{BN}$ performs better than random sampling. For smaller problems (30 and 40 tasks), random sampling outperforms $Greedy_{BN}$.

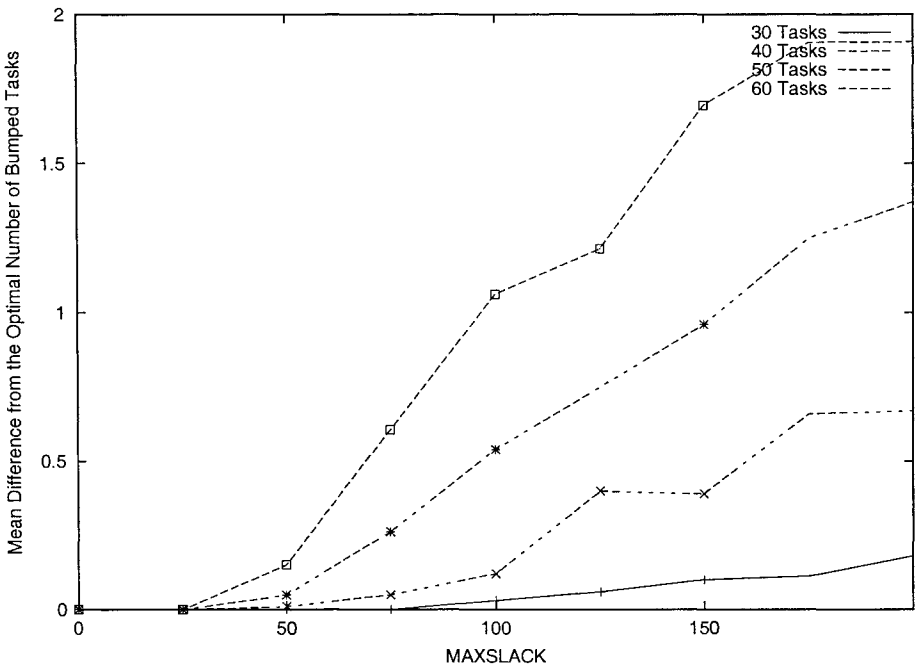


Figure 5.4: The mean difference from the optimal number of bumped tasks $\overline{\Delta_{best}}$ for StructLS.

Finally, we consider the performance of hill-climbing, *Genitor* and *SWO*. We define a single trial as an execution of the algorithm with a limit of 8000 solution evaluations.

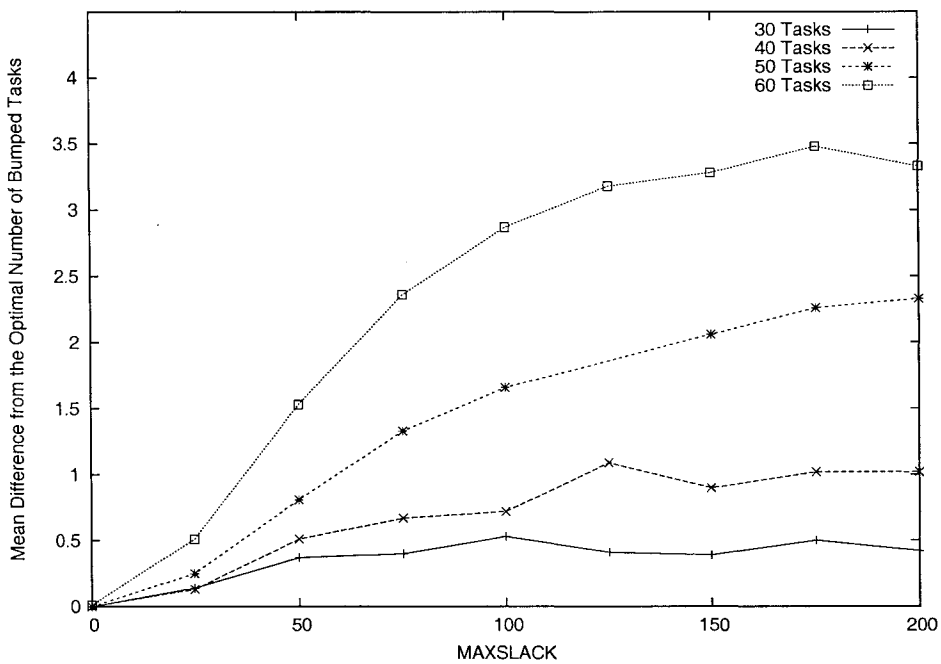


Figure 5.5: The mean difference from the optimal number of bumped tasks $\overline{\Delta_{best}}$ for *SWO*.

Due to the stochastic nature of these algorithms, we execute 30 independent trials of each algorithm on each of our test instances and define $|Bumps|$ for each algorithm as the minimal number of conflicts observed in any of the 30 trials.

The resulting means Δ_{best} are shown in Figures 5.3 (middle and bottom graphs), 5.4 and 5.5. *RandLS* performs better than *StructLS* and *Genitor*. *SWO* performs poorly, better than random sampling but worse than *StructLS*. For 30, 40 and 50 tasks, *SWO* performs better than $Greedy_{BN}$. For 60 tasks, $Greedy_{BN}$ consistently outperforms *SWO*.

Although the performance of *Genitor* and *StructLS* is indistinguishable for $n = 30$ and $n = 40$, the performance of *Genitor* scales better than that of *StructLS* for larger problem sizes. However, on these single resource problems *Genitor* under-performs the greedy heuristic ($Greedy_{DP}$), independently of n and *MAXSLACK*.

While for problem sizes 30, 40 and 50 *RandLS* performs better than $Greedy_{DP}$, for 60 requests, $Greedy_{DP}$ performs best. The performance of $Greedy_{DP}$ is surprisingly

good: $Greedy_{DP}$ being a constructive heuristic is run only once to build one schedule, while both *RandLS* and *Genitor* are allowed 30 runs and build 8000 schedules in each run.

These results demonstrate that a simple greedy heuristic, originally developed in the context of $1|r_j|\sum U_j$ out-performs, on average, *Genitor*, an algorithm more computationally expensive and previously found to be most efficient for AFSCN scheduling. *Genitor* is also outperformed by *RandLS*. While *RandLS* performs better than the simple greedy heuristic for problem sizes 30, 40 and 50, it does so at the expense of evaluating $30 * 8000$ solutions for each problem. On the other hand, $Greedy_{DP}$ produces remarkably good results in a very short time: it only builds one solution for each problem. The equivalence of SiRS to $1|r_j|\sum U_j$ scheduling enabled us to leverage algorithms for computing optimal solutions to problem instances. Our results demonstrate that SiRS problems with characteristics found in real-world AFSCN problems can often be solved to optimality, and in general, near-optimal solutions can be found.

5.1.2 Algorithm Performance for the Multiple Resource Problems (MuRS)

The SiRS problem is a simplified version of AFSCN scheduling for which the optimal solutions can be found using an exact algorithm. This allowed us not only to compare algorithm performance, but also to infer how far from optimal are the results produced by other algorithms. However, SiRS is more of a theoretical version of the AFSCN scheduling application. In practice multiple resources (antennas) are available, and alternative resources are specified for each task. MuRS more accurately models the real AFSCN scheduling problem. In this section, we analyze heuristic algorithm performance for MuRS. Because of the complexity and size of the MuRS, we were not able to use exact algorithms to find optimal solutions (this is usually the case for real-world large size scheduling problems). Therefore, all algorithm comparisons in this section

are relative to each other.

We consider two key questions to answer: Can the results for single resource problems be leveraged for multi-resource problems? and What is the best performing algorithm for AFSCN scheduling? To address these questions, we test the same algorithms as for SiRS (modified as necessary to fit the demands of the new problem) on MuRS problems.

To obtain the problems for this study, we designed a problem generator for MuRS, which preserves some of the features of the SiRS problems, while introducing new characteristics of the contemporary application. The SiRS problem generator described in section 5.1.1 requires parameters to control the task duration and the size of the time window. In the new generator, we use these parameters to model different types of requests encountered in the real-world satellite scheduling problem, such as downloading data from a satellite, transmitting information or commands from a ground station to a satellite, and checking the health and status of a satellite. Task durations include turnaround time, which is fixed at 20 minutes for low altitude tasks and 15 minutes for high altitude tasks. We classify the requests into four possible types:

- State of health (HS): short (25-30 minutes), flexible, common
- Maneuver (Mu), for example changing the orbit of a satellite: longer (35-75 minutes), inflexible, rare
- Payload download (PD), for example downloading data from the satellite: long (25-45 minutes), more flexible, common
- Payload commanding (PC): variable length (20-75 minutes), flexible, common

Our taxonomy is based on the fact that requests of a certain type share characteristics of duration, flexibility (the size of the time window versus the duration of the request) and

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

Table 5.2: Customer types.

frequency [3]. For each request type, we define the upper and lower bound on its duration T_i^{Dur} and the maximum slack *MAXSLACK* (to determine the window size). The actual tasks are generated using these parameters as described in section 5.1.1.

As a second feature, the new generator introduces models of customer behavior. We define three customer types (see Table 5.2), based on the fact that, for example, some customers will mostly generate health and status requests or payload command requests. For each customer, the number of requests to be generated is determined as a fraction of the total number of requests. We define customer patterns by specifying a time window midpoint, a resource, and the type of request. Customer patterns model the previous behavior of a customer (which could have been collected from past schedules). A database of customer patterns is used to produce the requests. For each customer type, the fraction of the requests generated for each request type and the $Predictability$ are defined. The $Predictability$ for each customer represents the fraction of requests that will be generated for this customer using customer patterns from the database. The rest of the requests corresponding to each customer will be randomly generated. The duration and window size for each request are determined using the parameters associated with the request type.

There are two types of requests in AFSCN scheduling: low-altitude and high-

[3]We do not model periodic requests and therefore do not use the frequency information.

altitude. The low-altitude requests tend to be shorter and have a duration equal to the corresponding visibility window. The high-altitude requests are more flexible, with the size of the visibility window larger than the requested duration. Similarly, in our generator the size of the visibility window is equal to the request duration for all the low-altitude requests, and it can vary for the high-altitude requests.

In our problem generator, we specify more than one ground station as possible alternatives for both low and high-altitude requests, due to the high contention for resources. Also, the time windows specified for alternative ground stations are generated based on an offset in time. The offsets represent the time needed for a satellite in its orbit to become visible from one ground station to another. We use two databases (low-altitude and high-altitude) of offsets, which were compiled[4] from data collected on the Web (http://earthobservatory.nasa.gov/MissionControl/overpass.html) about the visibilities of various satellites from the locations of the nine ground stations. For security reasons, the real AFSCN data does not specify actual satellites in an accessible form; for our modeling, we chose some satellites to be representative for the kinds of the requests encountered in AFSCN scheduling. We preserve the request duration for all the alternatives.

The process of generating the requests in the new generator is described in Figure 5.6. In AFSCN scheduling approximately half of the requests are low altitude requests. Therefore, in our generator, with a 0.5 probability, we determine for each request if it is a low-altitude or high-altitude request.

---

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

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

Figure 5.6: Pseudocode for the second generator.

Algorithm Performance for MuRS

Currently, no complete algorithm is available for computing optimal solutions for MuRS. Consequently, all comparisons of heuristic algorithms for MuRS are necessarily relative; even if one algorithm consistently outperforms another, there is no assurance that the algorithm is generating optimal, or even near-optimal, solutions.

We have previously observed that $Greedy_{DP}$ performs well for the SiRS problem. $Greedy_{DP}$ was designed for single-resource scheduling. To investigate if the performance observed for the SiRS problem transfers to the MuRS problem, we implemented a simple extension of the algorithm, to account for the presence of multiple resources and multiple alternatives for each request. For each of the resources, we use $Greedy_{DP}$ to schedule the requests that specify that resource as an alternative and are not scheduled yet; the result is an initial schedule. This means that each request will be successively considered for scheduling using $Greedy_{DP}$ on each of its alternative resources, until it is scheduled or until all alternative resources have been scheduled. Then we consider the unscheduled requests and attempt to insert them in the schedule. The unscheduled requests are considered in an arbitrary order; the request is scheduled at the earliest time on the first alternative resource available and bumped if none of the alternative resources

are available.

We did not include Gooley's algorithm in the study of the MuRS problem, because our generator produces problems that allow for antennas at multiple ground stations as alternative resources for the low altitude requests. Gooley's algorithm implementation optimally schedules the low altitude requests in its first phase, based on the fact that the alternative resources for such requests specify only the antennas present at one ground station.

We ran random sampling, *Genitor*, *RandLS*, *StructLS*, *SWO* and $\text{Greedy}_{\text{DP}}$ for problems produced by the new generator. We ran the experiments for problem sizes 300, 350, 400, 450, and 500. For each size, we generated 30 problem instances. For each algorithm, we collect the best solution obtained for each problem in 30 runs with 8000 evaluations per run.

Figure 5.7 shows the average percent of requests scheduled when minimizing conflicts. The best solutions are found by *Genitor*, *RandLS*, and *SWO*, which perform equally well on these problems. Similarly to the results observed for SiRS, *StructLS* performs worse than *RandLS*. However, the $Greedy_{DP}$ (which was one of the best performing algorithms for SiRS) results in the worst performance, except for problem size 500 where it slightly outperforms random sampling.

The results obtained when minimizing the sum of overlaps are summarized in Figure 5.8. Similarly to the results when minimizing conflicts, the best solutions are found by *Genitor*, *RandLS* and *SWO* for problem sizes 300, 350, 400 and 450. For problem size 500, *Genitor* performs slightly worse than *RandLS* and *SWO*. As with minimizing conflicts, *StructLS* performs worse than *RandLS*. Since $Greedy_{DP}$ was designed to minimize conflicts, it was not included in the set of algorithms considered when minimizing overlaps.

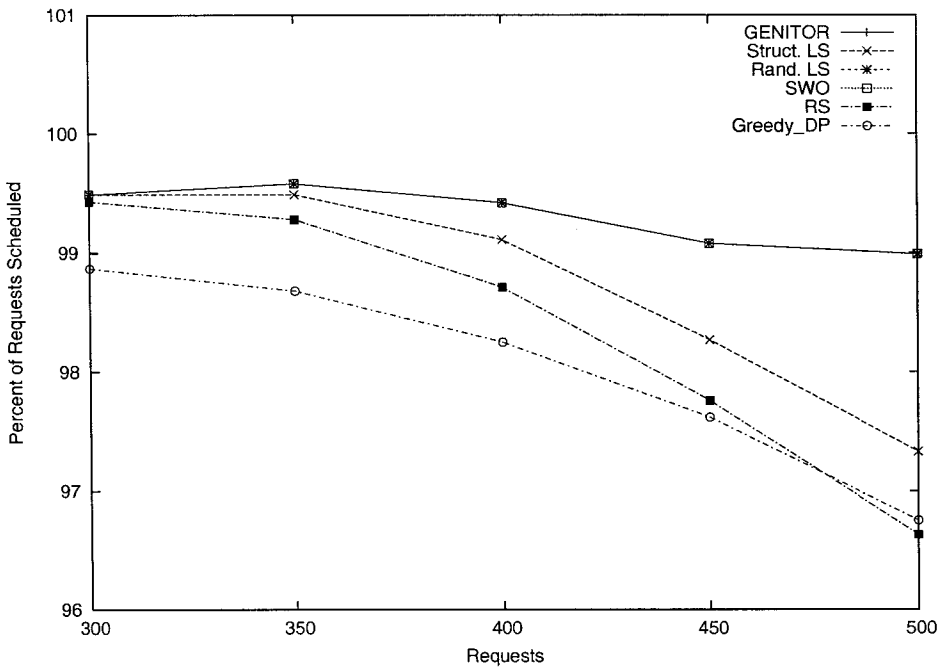


Figure 5.7: Average percent of requests scheduled by *Genitor*, the two versions of hill-climbing, *SWO* and random sampling. Note that the curves for *Genitor*, *RandLS* and *SWO* overlap.

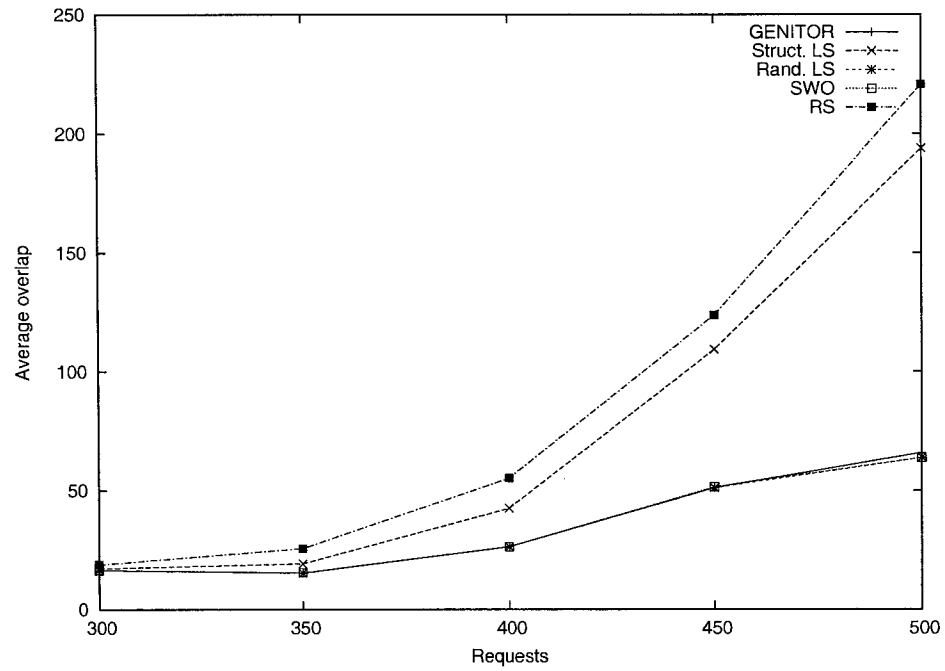


Figure 5.8: Average overlap for *Genitor*, the two versions of hill-climbing, *SWO* and random sampling. The curves for *Genitor*, *RandLS* and *SWO* overlap for 300, 350, 400 and 450 requests.

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	773
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 5.3: 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.

5.2 Algorithm Performance on Real Problems

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 5.3 summarizes the characteristics of the data. We will refer to the problems from 1992 as the A problems for AFIT, and to the more recent problems, as the R problems for recent.

Similarly to our analysis for MuRS, we report the results obtained by running random sampling, $Greedy_{DP}$, *RandLS*, *StructLS*, *Genitor* and *SWO*. In addition to these algorithms, we also include results from running Gooley's algorithm and HBSS using the flexibility heuristic as implemented for *SWO*.

Best Known Solutions

The best known solutions were obtained by performing long runs over hundreds of experiments. Using various algorithms and allowing for hundreds of thousands of evaluations, we have not found better solutions than these.

Day	CPU times		
	Genitor	Hill-climbing	*SWO*
A1	107	96	97
A2	101	92	91
A3	102	91	89
A4	107	97	95
A5	104	93	91
A6	105	93	92
A7	90	81	79
R1	189	175	178
R2	162	146	148
R3	144	133	130
R4	140	126	125
R5	127	112	109

Table 5.4: CPU times for *Genitor*, hill-climbing and *SWO* corresponding to 30 independent runs, with 8000 evaluations per run, when minimizing conflicts.

All the best known values for the sum of overlaps (see Table 5.3) can be obtained by running *Genitor* with the population size increased to 400 and allowing 50,000 evaluations per run.

When we report that an algorithm is better than *Genitor* it means that it was better than *Genitor* when both algorithms were limited to 8000 evaluations.

Results

The results of running each of the algorithms when minimizing the number of conflicts are summarized in Tables 5.5 and 5.6. For minimizing the sum of overlaps, the results are presented in Tables 5.7 and 5.8. For *Genitor*, hill-climbing and *SWO*, we report the best and mean value and the standard deviation observed over 30 runs, with 8000 evaluations per run. In addition, for *SWO* we also report the value of the initial solution (the *Init.* column). Both *Genitor* and hill-climbing were initialized from random permutations. For HBSS, the statistics are taken over 240,000 samples. With the exception of Gooley's algorithm, the CPU times are dominated by the number of evaluations and are therefore similar. On a Dell Precision 650 with 3.06 GHz Xeon running Linux,

30 runs with 8000 evaluations per run for minimizing conflicts take between 80 and 190 seconds (for more precise values, see Table 5.4). 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 *RandLS* 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. For five of the 12 problems, the initial permutation for *SWO* corresponds to a best known value (see the *Init.* column in Table 5.6); for the rest of the problems, the initial permutation is worse than the best known value by at most 2 for the A problems and at most 6 for the R problems. Therefore, it is not surprising how well *SWO* performs when minimizing the conflicts, even though we chose a very simple implementation, where all the tasks in conflict are moved forward with a fixed distance. The performance of *StructLS* 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 algorithm, finds best known valued solutions only for A7. Similarly to the results for the MuRS problem, $Greedy_{DP}$ is the worst performer; in fact, it performs worse than random sampling.

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 *StructLS* is relatively poor: worse mins/means and higher variance. However, the *RandLS* performs as well as *SWO*, and it finds two better solutions for R3 and R4. If *SWO* is allowed more evaluations, further improvements are small, unlike for *Genitor* which continues to improve the solution quality given more evaluations (see Section 5.3 for more details). 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

Day	Random Sampling			HBSS			Gooley	$Greedy_{DP}$
	Min	Mean	Stdev	Min	Mean	Stdev		
A1	21	22.7	0.87	**8**	9.76	0.46	11	21
A2	11	13.83	1.08	**4**	4.64	0.66	7	22
A3	16	17.76	0.77	**3**	3.37	0.54	5	25
A4	16	20.20	1.29	**2**	3.09	0.43	4	23
A5	15	17.86	1.16	**4**	4.27	0.45	5	18
A6	19	20.73	0.94	**6**	6.39	0.49	7	28
A7	16	16.96	0.66	**6**	7.35	0.54	**6**	22
R1	73	78.16	1.53	45	48.44	1.15	45	97
R2	52	57.6	1.67	32	35.16	1.27	36	72
R3	38	41.1	1.15	19	21.08	0.89	20	55
R4	48	50.8	0.96	**28**	31.22	1.10	29	66
R5	25	27.63	0.96	**12**	12.36	0.55	13	45

Table 5.5: Performance of random sampling, HBSS, Gooley's algorithm and $Greedy_{DP}$ in terms of number of conflicts. For HBSS, 240,000 samples are considered.

	Genitor			*StructLS*			*RandLS*			*SWO*			
Day	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev	Init.	Min	Mean	Stdev
A1	**8**	8.6	0.49	15	18.16	2.54	**8**	8.7	0.46	10	**8**	8	0.0
A2	**4**	4	0	6	10.96	2.04	**4**	4.0	0	**4**	**4**	4	0.0
A3	**3**	3.03	0.18	11	15.4	2.73	**3**	3.1	0.3	**3**	**3**	3	0.0
A4	**2**	2.06	0.25	12	17.43	2.76	**2**	2.2	0.48	3	**2**	2.06	0.25
A5	**4**	4.1	0.3	12	16.16	1.78	**4**	4.7	0.46	**4**	**4**	4	0.0
A6	**6**	6.03	0.18	15	18.16	2.05	**6**	6.16	0.37	**6**	**6**	6	0.0
A7	**6**	6	0	10	14.1	2.53	**6**	6.06	0.25	7	**6**	6	0.0
R1	**42**	43.7	0.98	68	75.3	4.9	**42**	44.0	1.25	48	43	43.3	0.46
R2	**29**	29.3	0.46	49	56.06	3.83	**29**	29.8	0.71	35	**29**	29.96	0.18
R3	**17**	17.63	0.49	34	38.63	3.74	**17**	18.0	0.69	20	18	18	0.0
R4	**28**	28.03	0.18	41	48.5	3.59	**28**	28.36	0.66	31	**28**	28.3	0.46
R5	**12**	12.03	0.18	15	17.56	1.3	**12**	12.4	0.56	**12**	**12**	12	0

Table 5.6: Performance of *Genitor*, hill-climbing and *SWO* in terms of the best and mean number of conflicts. Statistics for *Genitor*, hill-climbing and *SWO* are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

RandLS, *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 5.8; however, Gooley's algorithm was not designed to minimize overlaps. $Greedy_{DP}$ results in the worst performance (it is in fact often outperformed by random sampling).

Day	Random Sampling			HBSS			Gooley
	Min	Mean	Stdev	Min	Mean	Stdev	
A1	360	415.5	19.59	128	158.7	28.7	687
A2	134	184.93	18.99	43	70.1	31.1	535
A3	195	255.6	22.38	**28**	52.5	16.9	217
A4	222	277.3	24.51	**9**	45.7	13.0	216
A5	185	241.27	22.91	50	82.6	13.2	231
A6	275	320.9	21.68	**45**	65.5	16.8	152
A7	218	250.43	15.29	83	126.4	12.5	260
R1	1729	1816.63	35.36	1105	1242.6	42.1	1713
R2	1099	1216.73	43.14	598	681.8	27.0	1047
R3	863	934.3	28.39	416	571.0	46.0	899
R4	1194	1291	34.35	827	978.4	28.7	1288
R5	409	451.57	16.43	**146**	164.4	10.8	198

Table 5.7: Performance of $Greedy_{DP}$, HBSS and Gooley in terms of overlaps. For HBSS, 240,000 samples are considered.

	Genitor			*StructLS*			*RandLS*			*SWO*			
Day	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev	Init	Min	Mean	Stdev
A1	**104**	106.9	0.6	255	375.0	54.4	**104**	106.76	1.81	128	**104**	104	0.0
A2	**13**	13	0.0	65	174.6	50.6	**13**	13.66	2.59	44	**13**	13.4	2.0
A3	**28**	28.4	1.2	144	252.0	52.9	**28**	30.7	4.31	44	**28**	28.1	0.6
A4	**9**	9.2	0.7	153	239.6	55.4	**9**	10.16	2.39	34	**9**	13.3	7.8
A5	**30**	30.4	0.5	142	220.1	59.3	**30**	30.83	1.36	85	**30**	30	0.0
A6	**45**	45.1	0.4	190	277.4	46.7	**45**	45.13	0.5	**45**	**45**	45.1	0.3
A7	**46**	46.1	0.6	137	219.6	40.4	**46**	49.96	5.95	116	**46**	46	0.0
R1	913	987.8	40.8	1559	1830.9	143.4	798	848.66	38.42	1174	798	841.4	14.0
R2	519	540.7	13.3	1078	1235.5	92.8	494	521.9	20.28	646	491	503.8	6.5
R3	275	292.3	10.9	788	967.7	96.7	**250**	327.53	55.34	526	265	270.1	2.8
R4	738	755.4	10.3	1139	1287.1	84.2	**725**	755.46	25.42	939	731	736.2	3.0
R5	**146**	146.5	1.9	351	457.9	69.1	**146**	147.1	2.85	**146**	**146**	146.0	0.0

Table 5.8: Performance of *Genitor*, hill-climbing and *SWO* in terms of overlaps. All statistics are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

Comparing the performance results obtained for the synthetic MuRS problems with the results for the real problems, we note certain discrepancies: 1) When minimizing conflicts, between 1% and 2% of the requests are bumped for the AFIT problems. For the synthetic MuRS problems with 300 requests, on average only 0.5% of the requests are bumped (see Figure 5.7). For the R problems, between 2.8% and 8.7% of the re-

quests are bumped. Only about 1% of the requests are bumped for the similarly sized synthetic MuRS problems. 2) When minimizing overlaps, the average overlap for the AFIT problems is approximately 50; while for the synthetic problems of size 300, it is approximately 17. For the R problems, the average overlap is around 450; while for the synthetic problems of size 500, it is approximately 65.

We built the problem generator to incorporate knowledge about customers and specific patterns of requests in the synthetic problems. The AFSCN data do not include any information about the customers who generated certain requests, or about the type of the requests (other than the low/high altitude classification). The customer patterns were randomly generated; the request types were defined based on our discussion with human schedulers. Therefore, we expected to see discrepancies between the synthetic and real problems. We could increase the number of requests, such that the synthetic problems would become more oversubscribed, but this would be only an artificial method of bringing the results closer. We could also force some of the requests to be longer (this would bring the overlaps results closer). We argue that this is not necessary: even though discrepancies exist, for both the synthetic and real problems an important common characteristic emerges: the best solutions are found by *Genitor*, *RandLS* and *SWO*.

5.2.1 Old versus Current Real Problems and A Simple Domain Heuristic

The old A problems specify between 297 and 322 requests to be scheduled. For the current R problems, between 419 and 483 requests need to be scheduled. Obviously, the size of the problem increased in the past 10 years. The resources available stayed pretty much the same: the recent data include three new antennas not referenced in the AFIT problems, and the human schedulers tell us that two antennas are no longer reliable. The increase in the number of requests currently received for a day also causes an increase in the number and percentage of requests that are bumped. For the 1992 data, at most

eight task requests (or 2.5% of the tasks) are bumped in the best schedules; for the more recent data, the number increases to 42 (or 8.7%). The variance in performance for the stochastic algorithms increases somewhat for the more recent data, but not as much as does the best and mean performance.

The changes in the data appear to go beyond just a difference in scale. In this section, we show that a simple heuristic that schedules first the low altitude requests and then the high altitude requests finds best solutions quickly for all the A problems. The simple heuristic does not perform as well for the R problems.

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 [Goo93]. Also, the split heuristic is similar to the contention measures defined in [FJMS01].

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

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 5.9: 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	Random Sampling-S		
		Min	Mean	Stdev
R1	42	47	48.3	0.76
R2	29	31	32.5	0.73
R3	17	18	18.8	0.48
R4	28	29	29.9	0.45
R5	12	12	12.6	0.49

Table 5.10: Results of running random sampling with the split heuristic in 30 experiments, by generating **8000** random permutations per experiment for minimizing conflicts.

by sampling 100 random permutations are shown in Table 5.9. For the recent days of data, the performance of the split heuristic does not transfer. In Table 5.10 we present the results of 30 iterations of randomly sampling 8000 random permutations, using the split heuristic. In this case, best known valued schedules are produced only for R5.

Next, we address the question: How well does the heuristic perform when minimizing overlaps? We performed 30 iterations of randomly sampling 8000 random permutations, evaluating the permutations using the split heuristic. The results are presented in Table 5.11. With the exception of A3, A4 and A6, the split heuristic does not find best known valued schedules.

The split heuristic operates in a much smaller search space than the initial one (all the permutations in its search space specify the low altitude requests before the high

Day	Best Known	Random Sampling-S		
		Min	Mean	Stdev
A1	104	119	121	2.32
A2	13	43	43.03	0.18
A3	28	**28**	29.9	2.3
A4	9	**9**	0	9
A5	30	50	50.37	0.99
A6	45	**45**	45.03	0.18
A7	46	69	71.87	3.63
R1	773	1185	1239.2	23.06
R2	486	616	646.43	13.53
R3	250	362	401.4	17.41
R4	725	858	884.77	10.08
R5	146	148	159.07	6.2

Table 5.11: Results of running random sampling with the split heuristic in 30 experiments, by generating **8000** random permutations per experiment for minimizing overlaps.

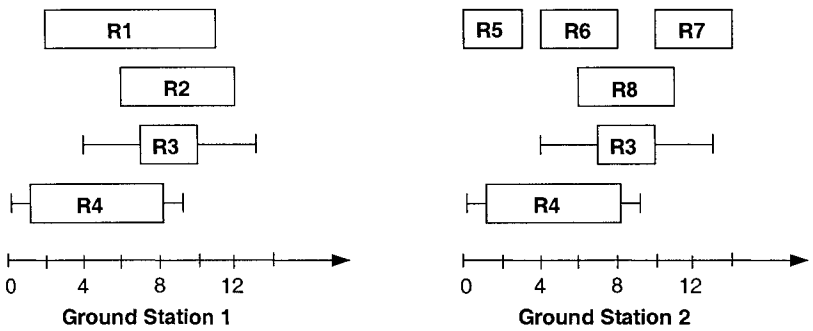


Figure 5.9: Example of a simple problem. Each tracking station has two antennas; the only high-altitude requests are $R3$ and $R4$.

altitude ones). Intuitively, if the optimal solutions were present in this search space, they might be easier to find than in the original search space. This would explain why a simple algorithm like random sampling finds best known solutions for some of the problems in the reduced search space. However, the optimal solution for the problem might not be present in this search space. For example, consider again the example problem presented in Chapter 3 (Figure 5.9). If low-altitude requests are scheduled first, then $R1$ and $R2$ are scheduled on Ground Station 1 on the two resources, and the

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

Our version of Gooley's algorithm optimally schedules the low altitude requests, then inserts the high altitude requests into the schedule. This is somewhat similar to the split heuristic. We know that: 1) the split heuristic finds best known solutions for the seven old days of data relatively easily and 2) these problems have best known solutions for which the low altitude requests are optimally scheduled. Therefore, we expected Gooley's algorithm to perform better on these problems. The performance of Gooley's algorithm on the 1992 problems is somewhat disappointing. On the other hand, the results in Table 5.5 show that its performance on the newer data is very similar to its performance on the data from 1992. Gooley's algorithm was designed based on the data from 1992. Given the differences between the 1992 data and the current data, this scale-up is surprising.

5.3 Algorithm Progression to the Solution

In the previous sections, we identified a set of algorithms that perform well for both the synthetic version (MuRS) and the real version of the problem. The algorithms in this set are: *Genitor*, *RandLS* and *SWO*. Even though all three algorithms use the same permutation search space and the same greedy schedule builder, they are very different in the way they modify the current solution (or solutions, for *Genitor*). We hypothesized

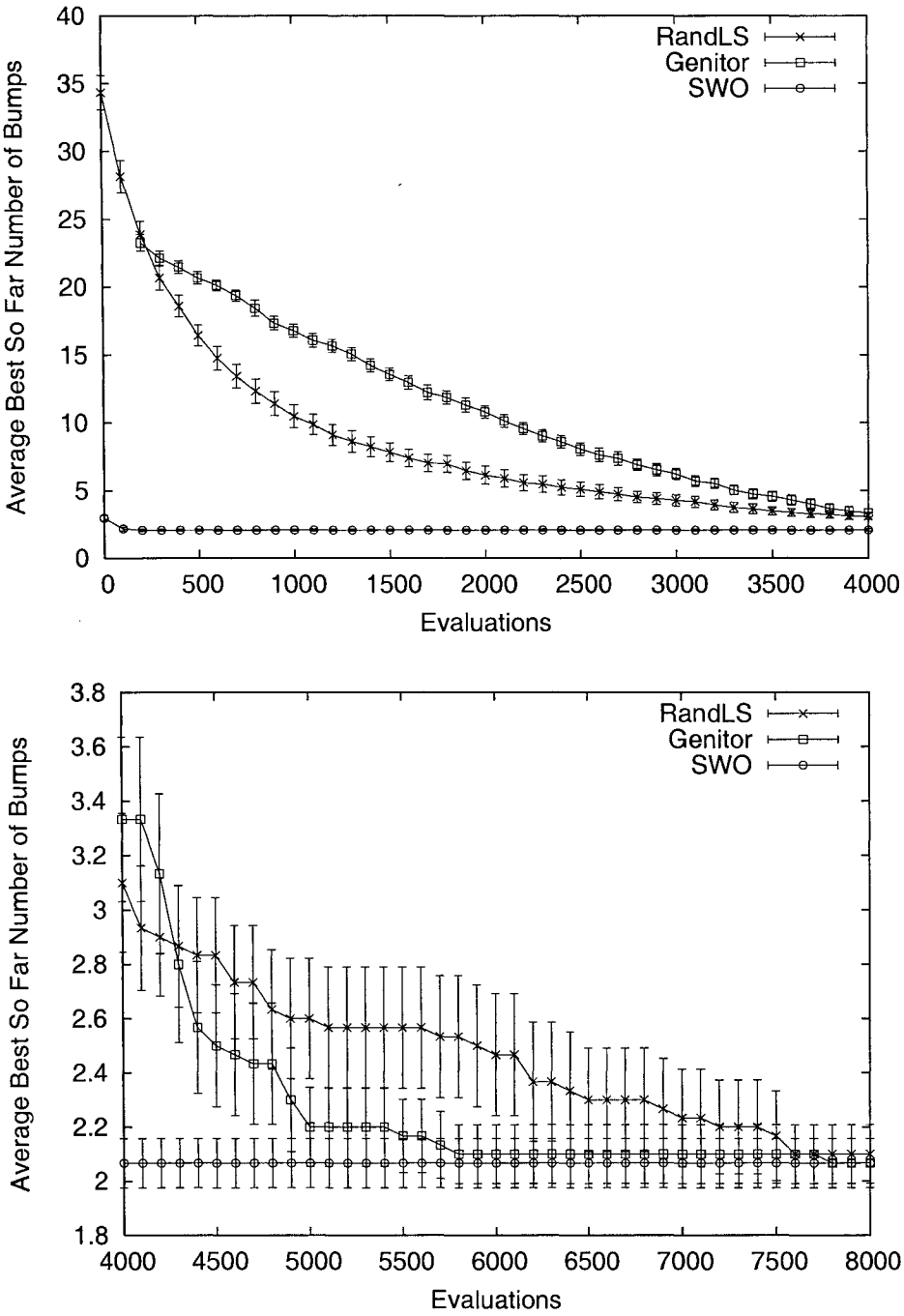


Figure 5.10: Evolutions of the average best value obtained by *RandLS*, *Genitor* and *SWO* during 8000 evaluations, over 30 runs. The improvement over the first 4000 evaluations is shown in the top figure. The last 4000 evaluations are depicted in the bottom figure; note that the scale is different on the y-axis.The graphs were obtained for $A4$; best solution value is 2.

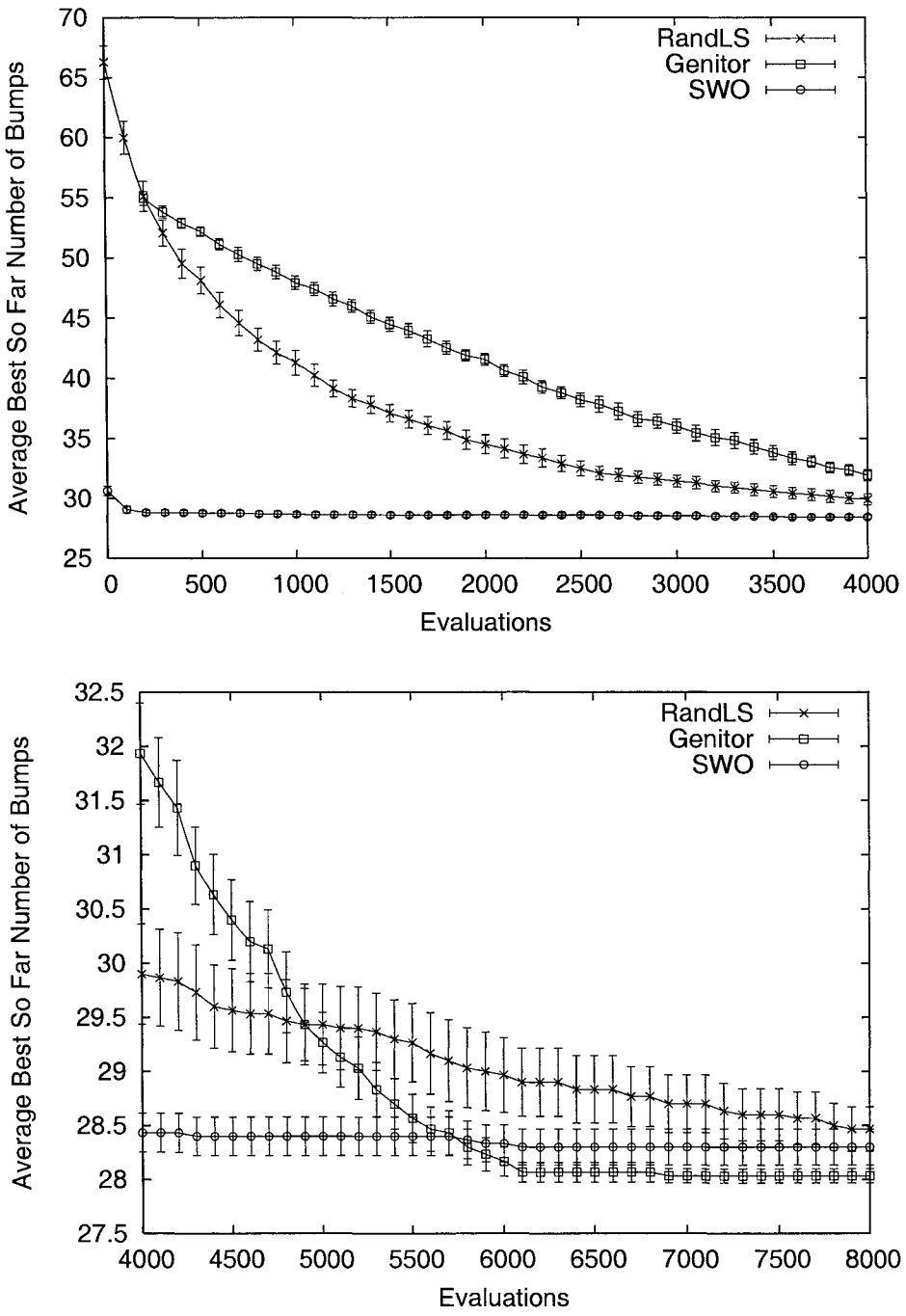


Figure 5.11: Evolutions of the average best value obtained by *RandLS*, *Genitor* and *SWO* during 8000 evaluations, over 30 runs. The improvement over the first 4000 evaluations is shown in the top figure. The last 4000 evaluations are depicted in the bottom figure; note that the scale is different on the y-axis.The graphs were obtained for $R4$; best solution value is 28.

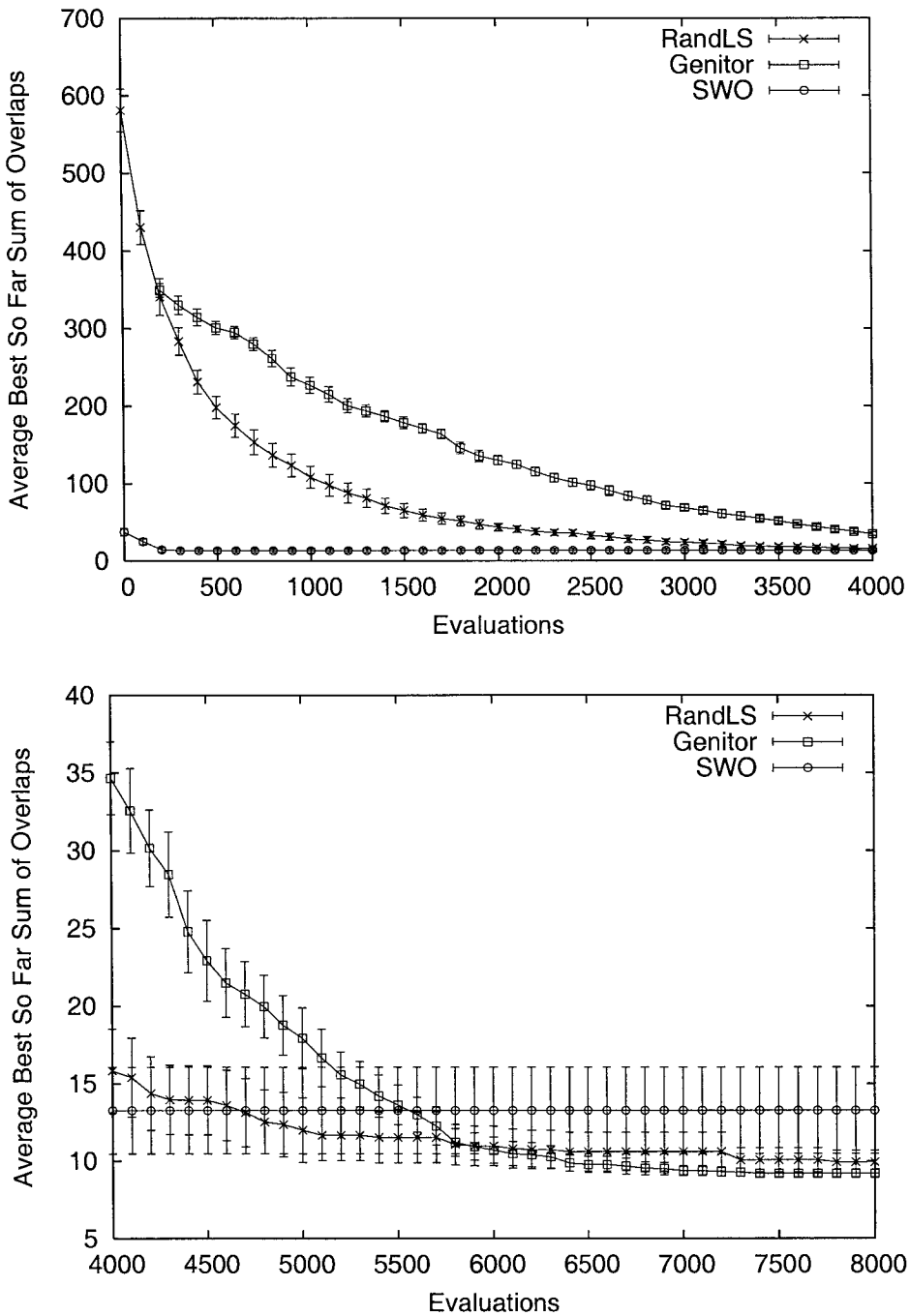


Figure 5.12: Evolutions of the average best value obtained by *RandLS*, *Genitor* and *SWO* during 8000 evaluations, over 30 runs. The improvement over the first 4000 evaluations is shown in the top figure. The last 4000 evaluations are depicted in the bottom figure; note that the scale is different on the y-axis.The graphs were obtained for $A4$; best solution value is 9.

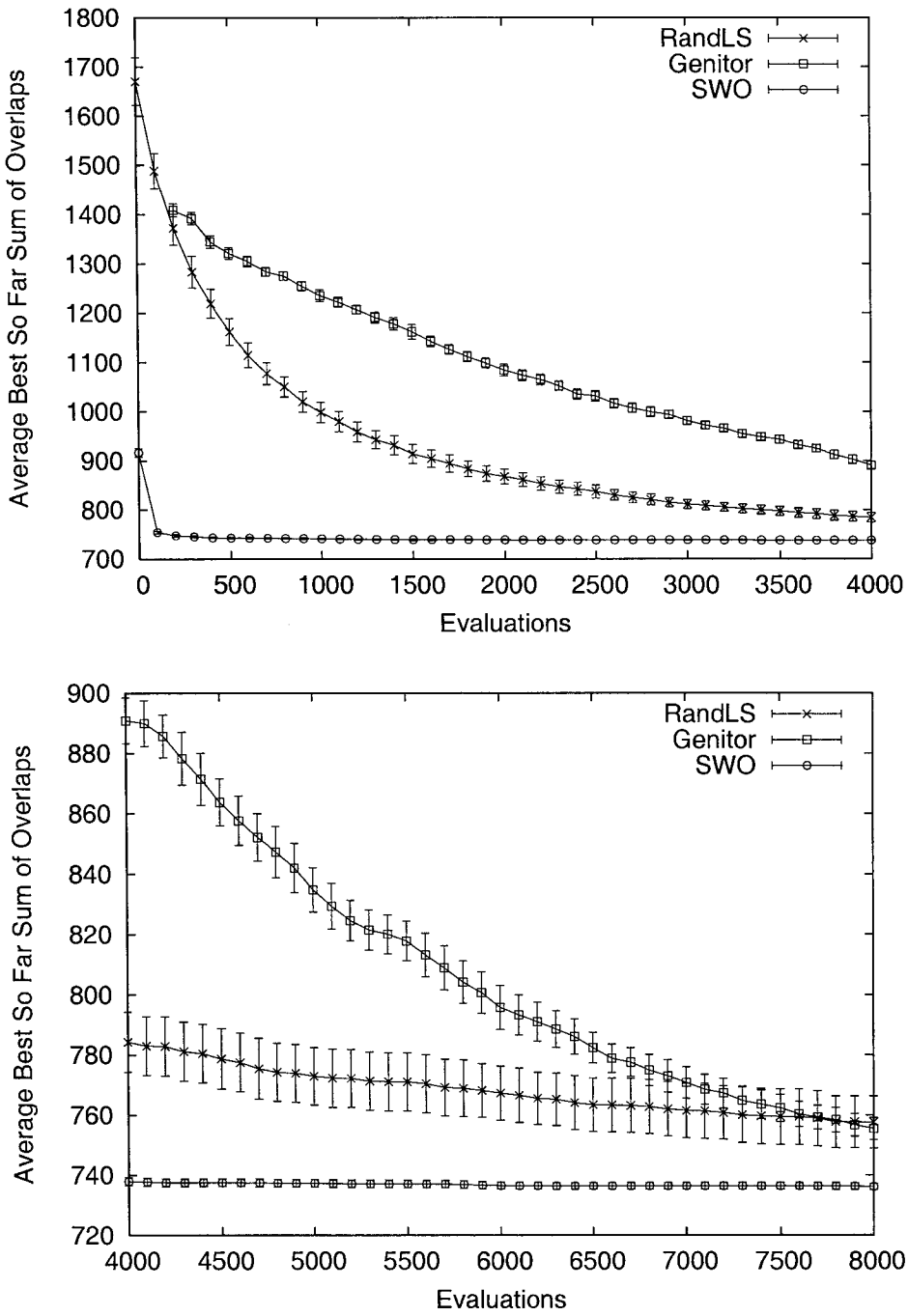


Figure 5.13: Evolutions of the average best value obtained by *RandLS*, *Genitor* and *SWO* during 8000 evaluations, over 30 runs. The improvement over the first 4000 evaluations is shown in the top figure. The last 4000 evaluations are depicted in the bottom figure; note that the scale is different on the y-axis. The graphs were obtained for $R4$; best solution value is 725.

that there are differences in the paths followed by the algorithms through the search space. To assess algorithm differences in the way they progress to the solution, for each algorithm, we collected the best value found in increments of 100 evaluations for up to 8000 evaluations. We averaged these values over 30 runs. Typical examples are presented in Figures 5.10, 5.11 when optimizing the number of conflicts and Figures 5.12, 5.13 when optimizing the overlaps. The graphs include 95% confidence interval error bars. In the beginning, *SWO* progresses very fast to a good solution, but further improvements over time are small. The initialization seems to play an important role in *SWO* performance. On the other hand, during the first half of the search, the *RandLS* progresses quickly, while *Genitor* exacts smaller improvements. In the second half of the search though, the *RandLS* takes a longer time to find better solutions while *Genitor* continues to steadily progress toward the best solution.

So far, the empirical results were obtained after 8000 evaluations. When minimizing the number of conflicts, given 8000 evaluations, *Genitor* 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*, but still appears to be improving. How do these results change if more evaluations are allowed? To answer this question, we ran 30 runs of *StructLS*, *RandLS*, *Genitor* and *SWO* allowing 500,000 evaluations per run. All four algorithms (*StructLS*, *RandLS*, *Genitor* and *SWO*) find best known values for all the problems when minimizing the number of conflicts. When minimizing the sum of overlap, all four algorithms find best known values for all the A problems, and for $R4$ and $R5$. While *RandLS* finds best known solutions for all of the problems, *Genitor* finds best known solutions for all the problems except $R1$. The best solution found by *Genitor* for $R1$ is 775. For $R1$, $R2$ and $R3$ the best solutions found by *SWO* in 30 experiments with 500,000 evaluations per experiment are 790, 489 and 265 respectively, which are worse than best known values for these problems. The

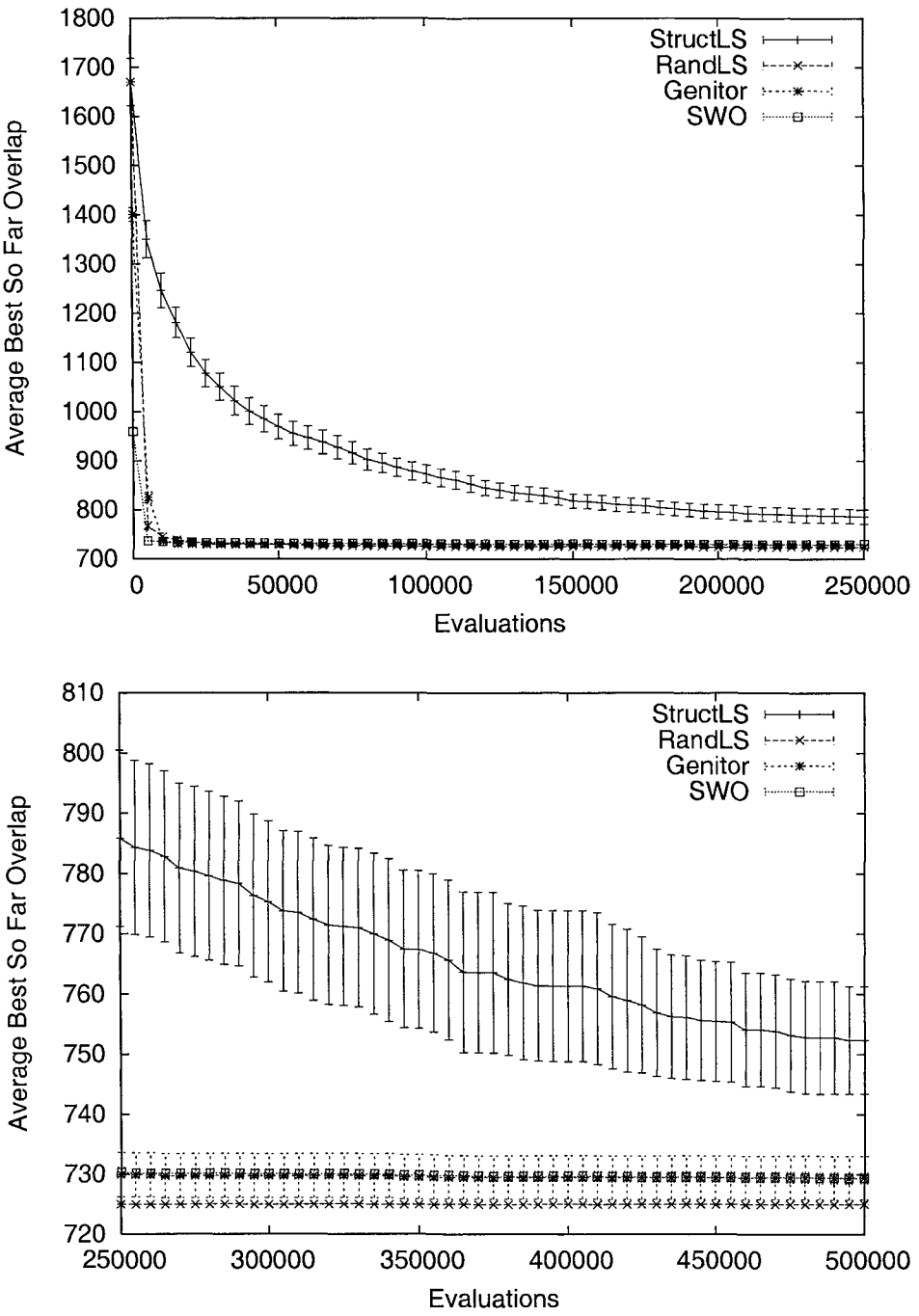


Figure 5.14: Evolutions of the average best value obtained by *StructLS*, *RandLS*, *Genitor* and *SWO* during 500,000 evaluations, over 30 runs. The graphs were obtained for $R4$; best solution value is 725.

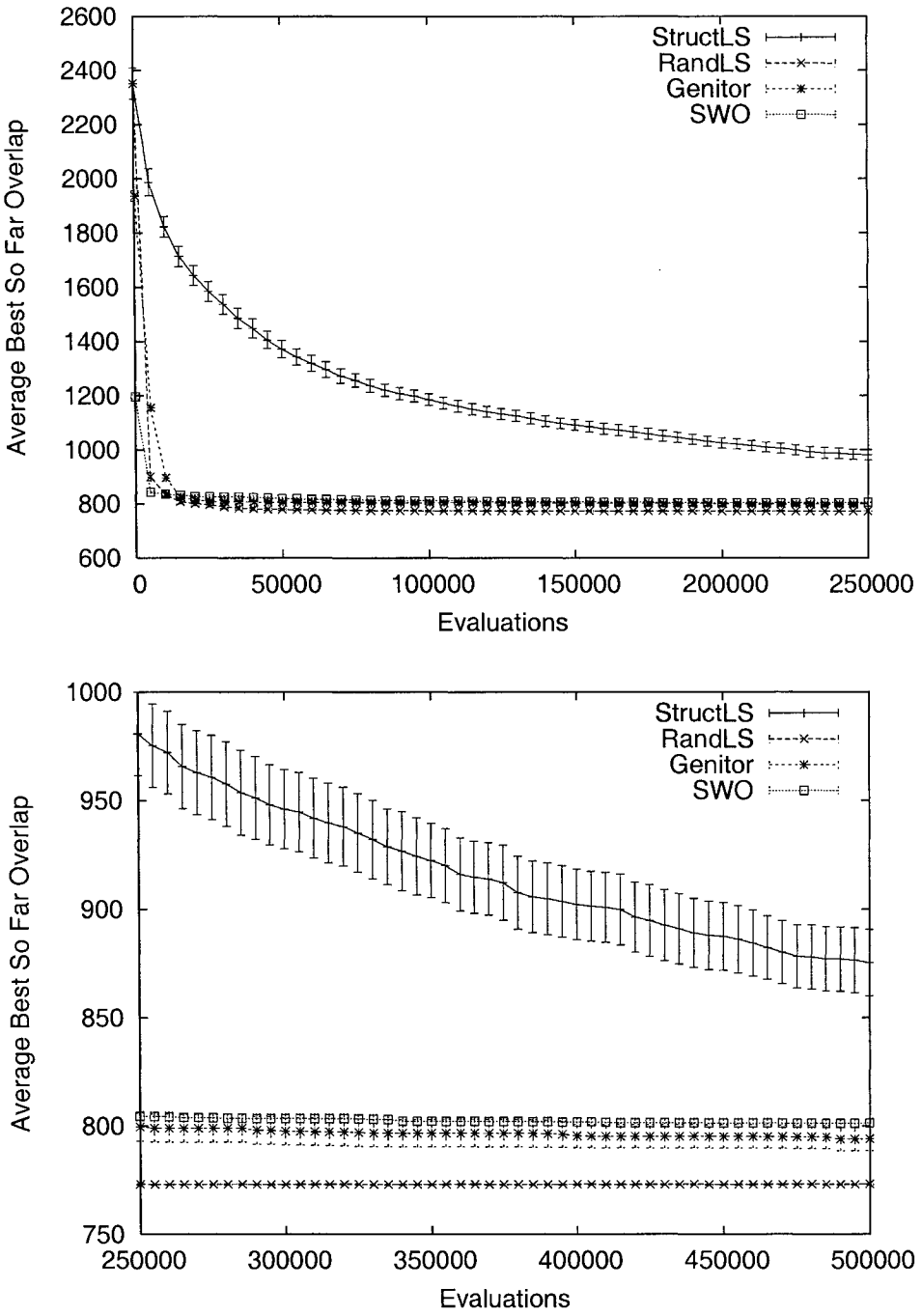


Figure 5.15: Evolutions of the average best value obtained by *StructLS*, *RandLS*, *Genitor* and *SWO* during 500,000 evaluations, over 30 runs. The graphs were obtained for $R1$; best solution value is 773.

best solutions found by *StructLS* for the same three problems are 792, 496, and 255.

In Figures 5.14 and 5.15, we show the evolution of the four algorithms over 30 runs, allowing 500,000 evaluations per run. The graphs were obtained for $R1$ and $R4$; similar behavior can be observed for the other R problems when minimizing overlaps. Three main characteristics can be observed in these graphs: 1) *StructLS* continues to creep through the search space; the average best value after 500,000 evaluations is still worse than the best known value, 2) *SWO* values show little improvement; average best final solutions are close to best known values, but still worse and 3) *RandLS* finds better final solutions than *Genitor*. Note that the graph shows *average* best so far values: for R4, *Genitor* can find the best known solution, but the average best found at the end of 500,000 evaluations is worse than the best known. With the exception of R1, considering all final solutions obtained, both *RandLS* and *Genitor* can find best known solutions.

Given more evaluations, the solutions found by *Genitor* and *RandLS* when minimizing overlaps for the R problems continue to improve. The graphs also show that best known solutions are obtained much sooner than the allowed 500,000 evaluations. In fact, the best known solutions for these problems (except for $R1$) can be obtained in 30 runs of *Genitor* with 50,000 evaluations per run. For $R1$, the best value found by *Genitor* in 30 runs with 50,000 evaluations per run is 784. Running *SWO* for 50,000 evaluations in 30 runs results in small improvements over 8000 evaluations, and only for $R2$ (from 491 in 8000 evaluations to 490) and $R4$ (from 731 to 728).

Chapter 6

Algorithm Performance Factors

The three best performing algorithms for AFSCN scheduling, *Genitor*, *RandLS* and *SWO*, appear to have little in common. All three algorithms 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 tasks in the order in which they appear in the permutation. The three algorithms project very different paths through the search space: they differ in the way the current solutions are modified to produce new solutions. *Genitor* combines two candidate solutions preserving elements of each. *SWO* creates an initial greedy solution and then attempts to improve the scheduling of all tasks known to contribute detrimentally to the current evaluation. *RandLS* makes incremental changes based on observed immediate gradients in schedule evaluation.

In this chapter we explore factors that explain the performance of each algorithm. Both the objective function and the schedule builder clearly influence algorithm performance. Therefore, we start by analyzing the differences in schedule quality for: 1) the two objective functions and 2) alternative schedule builders. Then, we formulate a number of hypotheses about the main performance factors for each of the algorithms:

- We find that the search space is dominated by large plateaus; plateaus are known to pose difficulties for hill-climbing algorithms. We hypothesize that *StructLS* per-

forms poorly because of the systematic order in which it evaluates the neighbors. For *RandLS*, we hypothesize that it performs essentially a random walk on the plateaus to find exits leading to better solutions; given the distribution of solutions and lack of gradient information, this may be as good a strategy as any.

- For *Genitor*, we hypothesize that it discovers patterns of relative task orderings. This is a classic building block hypothesis: some pattern that is present in parent solutions contributes to their evaluation in some critical way; these patterns are then recombined and inherited during genetic recombination [Gol89].

- *SWO* starts extremely close to the best solution; we hypothesize that this is the reason for its good performance. This hypothesis also implies that it is relatively easy to modify good greedy solutions to find the best known solutions. If this hypothesis holds, it has significance for *Genitor* and *RandLS* as well: *Genitor* and *RandLS* initialized with greedy solutions should also find improving solutions easily (similarly to *SWO*).

We test our hypotheses considering the greedy schedule builder as well as the objective function as part of the problem specification. We structure the analysis by focusing on each algorithm separately. To test the last hypothesis, we include a study of the effect of initializing with the greedy *SWO* initial solutions in the subsections corresponding to *Genitor* and *RandLS*.

Our analyses show that there is limited evidence for the existence of building blocks. And while *SWO* does quickly find good solutions, it cannot reliably find best known solutions. The hypotheses formulated for *Genitor* and *SWO* are somewhat supported by the data; however, they are not enough to explain the observed performance. Therefore, we formulate another hypothesis:

- *SWO* and Genitor make long leaps in the search space, which allow them to relatively quickly traverse the plateaus.

This last hypothesis appears to well explain the performance of the two methods.

6.1 Objective Function

The choice of the objective function is an important factor in the quality of the schedules produced by optimization algorithms. Minimizing the number of bumped jobs is the objective function used in previous research and development on AFSCN scheduling. A close examination of the schedules produced by this objective function shows that it introduces a bias toward leaving the longer requests unscheduled. The human schedulers need to negotiate new possible durations with the customers such that most requests are scheduled; making room in the schedule for the longer requests can be a difficult task to accomplish. We defined a second objective function, minimizing the sum of overlaps (see Chapter 3). This objective function provides useful information about the best possible way to accommodate all the requests with minimum adjustments of request durations.

The new objective also provides a richer evaluation function for the algorithms. When minimizing the number of bumped tasks, if 500 jobs are being scheduled, then the number of conflicts (unscheduled tasks) is an integer between 1 and 500. Most of the time, the number of conflicts is between 1 and 100 (and typically smaller); this objective function is a somewhat coarse metric. When using overlaps, the objective function is not just related to the number of jobs, but also to their durations. For example, if the number of conflicts is between 1 and 100, but the overlaps range from 1 to 50 time units, then the objective function ranges over 1 to 5000 and provides more distinction between alternative solutions. This means that two schedules may have exactly the same number of bumped tasks, but have very different values in terms of the number of time units that

are conflicted and overlap. Using overlaps as the objective functions, approximately 20 times more unique objective function values are observed compared to the objective of minimizing conflicts when sampling a high number of random permutations.

Next, we exemplify the differences in schedules produced when optimizing one of the two objective functions. We show: 1) the corresponding sum of overlaps if the bumped requests were allowed to overlap with the scheduled requests, in schedules minimizing conflicts and 2) the number of tasks that would be bumped instead of allowed to overlap with scheduled requests, for schedules minimizing overlaps. In Table 6.1, we present the results of running Genitor-Bumps (Genitor with the objective function of minimizing the number of conflicts) and Genitor-Overlaps (Genitor with the objective of minimizing the sum of overlaps). The statistics are collected over 30 runs, with 8000 evaluations per run. For each best schedule obtained by Genitor-Bumps in a run, we compute the corresponding sum of overlaps for the bumped requests. Statistics for both the number of conflicts and the sum of overlaps over 30 runs are shown in columns 2-7. For each permutation evaluated as best at the end of a run of Genitor-Overlaps, we apply the schedule builder that minimizes the number of conflicts. Again, we present statistics for the number of conflicts and the sum of overlaps over 30 runs in columns 8-13.

The results show clearly that optimizing the number of conflicts results on average in a larger corresponding sum of overlaps than when the overlaps are optimized, and the increase can be quite significant. Similarly, optimizing the sum of overlaps results in a number of conflicts which is usually larger than when the conflicts are optimized; the increase is significant for the R problems. When minimizing the number of conflicts, the variance in the computed sum of overlaps for the solutions is much higher than the variance corresponding to the overlaps for Genitor-Overlaps. Also, for the R problems, the variance in the corresponding number of bumps for Genitor-Overlaps is higher than the variance in the number of bumps for Genitor-Bumps. The results also suggest that

	Genitor-Bumps						Genitor-Overlaps					
	Bumps			Overlaps			Bumps			Overlaps		
Day	Min	Mean	S.D.	Min	Mean	S.D.	Min	Mean	S.D.	Min	Mean	S.D.
A1	8	8.6	0.5	192	327.7	83.7	9	9.4	0.6	104	106.9	0.5
A2	4	4	0	76	106.8	19.0	4	4.1	0.4	13	13	0
A3	3	3.03	0.2	121	176.9	29.9	3	3.2	0.4	28	28.4	1.2
A4	2	2.06	0.2	9	30	16.08	2	2.2	0.4	9	9.2	0.7
A5	4	4.1	0.3	36	123.6	34.1	4	4.4	0.5	30	30.4	0.5
A6	6	6.03	0.2	70	103.1	12.9	6	6.06	0.2	45	45.1	0.4
A7	6	6	0	130	164.8	15.7	7	7.9	0.6	46	46.1	0.6
R1	42	43.7	0.9	1441	1650.8	76.6	55	61.4	2.9	913	987.8	40.8
R2	29	29.3	0.5	803	956.23	53.9	33	39.2	1.9	519	540.7	13.3
R3	17	17.6	0.5	790	849.9	35.9	24	27.4	10.8	275	292.3	10.9
R4	28	28.03	0.18	1069	1182.3	75.3	35	38.07	1.98	738	755.43	10.26
R5	12	12.03	0.18	199	226.97	20.33	12	12.1	0.4	146	146.53	1.94

Table 6.1: The results obtained for Genitor-Bumps and Genitor-Overlaps by running 30 experiments with 8000 evaluations per experiment. Genitor-Bumps optimizes the number of conflicts. Genitor-Overlaps optimizes the sum of overlaps.

when minimizing the number of conflicts, longer tasks are bumped, thus resulting in a large sum of overlaps. Indeed, for Genitor-Bumps, the average length of the bumped tasks can be computed by dividing the mean overlap (column seven in Table 6.1) by the mean number of bumps (column four in the table). This is always greater than the average length of the tasks involved in overlaps; for Genitor-Overlaps, the mean sum of overlaps is always smaller and the mean number of bumps is always higher than for Genitor-Bumps.

6.2 Schedule Builder

With the exception of HBSS, all the algorithms we implemented for AFSCN scheduling rely on a schedule builder to translate from the permutation space to the schedule space. The design of a schedule builder heavily influences algorithm performance. Indeed, given a permutation solution, two different schedule builders will likely produce two different schedules for that same permutation. One would expect therefore the search

spaces associated with different schedule builders to be very different. This means that the performance of an algorithm changes depending on the schedule builder used to produce the schedules.

Gooley's schedule builder (see Section 4.3) implements complex domain-specific heuristics to allocate time and resources to the requests. The schedule builder described in 4.4.1 is a very simple greedy one. We investigate the question: How do the schedules produced by Gooley's scheduler compare to schedules produced by the greedy scheduler and why?

The greedy schedule builder allocates the resources using a static order, that does not depend on the context of the schedule that is built. This increases the redundancy of the search space and such redundancy is an important issue for algorithm performance. To assess the effect of the scheduler on algorithm performance, we implemented three other versions of the schedule builder. They differ in the way they assign start times and resources to the requests. We investigate the interaction effect for *Genitor*, *RandLS*, *StructLS* and *SWO* when these different schedule builders are used.

One of the main strengths of the simple greedy schedule builder is the fact that it produces schedules fast, in approximately 4 to 8 seconds per 8000 evaluations. The new schedule builders use more complex criteria to determine the starting times and resources to be assigned to the requests; therefore, we expect the new schedule builders to also be more expensive computationally. We compare these CPU times with the ones corresponding to evaluating 8000 solutions when each of the new schedule builders is used. All the CPU times are obtained from runs on a Dell Precision 650 with 3.06 GHz Xeon running Linux.

Gooley's Schedule Builder

Gooley's schedule builder is very different from our simple greedy scheduler. So how does it compare to ours?

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

Table 6.2: Comparison of the effects of the permutation generation and the schedule builder on the value of the solution for *Genitor* and Gooley's algorithm.

To assess the influence of his schedule builder, we pair *Gooley*'s initial permutations with the simple, greedy schedule builder. This study only addresses minimizing the number of conflicts, since Gooley's algorithm was designed to optimize this objective. For clarity, we call the greedy schedule builder *GreedyS*. Surprisingly, this pairing improves over the solutions found in Gooley's original configuration for most of the problems (see Table 6.2). Thus, it appears that *GreedyS* is the better schedule builder.

This result is surprising: for six out of the seven problems from 1992 (for which Gooley's algorithm was designed), the greedy schedule builder works better than the domain-specific, complex heuristics used to schedule the high altitude requests and then to repair the schedule. We conjecture that the reasons for such performance differences might be found in the heuristics used to choose the resource and start time when scheduling a high altitude request.

To support this conjecture, we analyze in more detail the way Gooley's scheduler produces schedules. Gooley's Schedule Builder is represented by the set of heuristics used to incorporate the high altitude requests in the schedule containing the low altitude

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

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

For each high altitude request, its alternative resources are ranked based on the currently available free time across the whole resource; the resource with the most free time is considered first. Note that this is an extremely coarse measure; the available free time across the slots where the request can be scheduled would be more informative. Various criteria are then used to determine the start time for the request. These criteria refer to the free time available between the end of the request to be scheduled and the next request scheduled on that resource. For example, preference is given to slots for which

less than five minutes or more than 60 minutes are available between the end of the request scheduled and the next one. Such choices are motivated by heuristic knowledge about the domain: no request can be allocated less than five minutes, but if a slot longer than 60 minutes is available, a request can most likely be scheduled.

After all the high altitude requests have been considered for insertion, an interchange procedure attempts to accommodate the unscheduled requests by rescheduling some of the high altitude requests. Gooley's algorithm uses a flexibility measure to determine which requests should be rescheduled. The flexibility measure for a task request T_i as:

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

where:

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

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

The flexibility measure is directly proportional to the average length of the three largest blocks of free time in the alternative time windows, without correlating these to the duration of the request. This results in high flexibility for requests with long blocks of free time in their alternative time windows. However, since the length of such blocks is not checked against request duration, the corresponding requests might be harder to reschedule than shorter requests with shorter free blocks of time on their alternative resources (and smaller flexibility measure as defined). Because of this flaw in the flexibility measure, sometimes the algorithm unsuccessfully attempts to reschedule the request identified by the flexibility measure, when in fact there exists a different request, with a smaller flexibility, which can be rescheduled (but won't be considered for rescheduling).

The coarse measure used to assign resources to the high altitude requests as well as the flaw in the flexibility heuristic both contribute to producing schedules that are worse than those produced by the greedy scheduler.

Earliest Start Schedule Builder

The only difference between this scheduler and the greedy scheduler is that the **earliest start** scheduler schedules each request at the earliest time available on its alternative resources. If there are multiple resources on which the task can start at the earliest time, one resource is chosen by random. The greedy scheduler schedules each request at the earliest time on the *first* available resource: it checks the alternative resources in the order in which they are specified for the request and stops as soon as it finds a resource on which the request can be scheduled. The **Earliest Start Schedule Builder** finds the earliest possible start times for *all* the resources and chooses the resource with the smallest one to schedule the request.

We show the results obtained by running *Genitor*, *RandLS*, *StructLS* and *SWO* with the earliest start schedule builder in Tables 6.3 and 6.4. With the exception of *SWO*, the performance of all the other algorithms worsens, both in terms of best solution and in terms of mean solution found, for both objective functions.

The performance of *SWO* is similar to its performance when using the simple greedy schedule builder. It is not clear why the performance of *SWO* does not degrade with this schedule builder, while the other algorithms do. The reason does not seem to be related to the initial greedy solution: the values in the *Init* column for *SWO* correspond to exactly the same initial permutation that was evaluated by the greedy scheduler in the *Init* column of Tables 5.6 and 5.8. With a few exceptions, for both objective functions, the initial permutation results in a worse schedule when the earliest start schedule builder is used. While it is not clear why this would be the case, it seems that by scheduling earliest start time first the resources end up fragmented. The alternative resources appear

Day	*Genitor*			*StructLS*			*RandLS*			*SWO*			
	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Init	Min	Mean	SD
A1	9	10.57	1.41	21	26.9	2.67	14	18.97	3.38	9	**8**	8	0
A2	**4**	7.27	1.46	13	19.03	2.54	8	12.67	3.01	6	**4**	4	0
A3	5	6.97	1.75	14	20.83	2.76	9	12.9	2.19	5	**3**	3	0
A4	3	6.57	1.81	12	19.3	2.77	7	12.7	3.45	5	**2**	2	0
A5	**4**	7.4	1.94	14	19.23	2.56	7	12.07	2.72	**4**	**4**	4	0
A6	8	9.2	1.21	12	19.53	2.96	9	14.1	2.82	7	**6**	6	0
A7	**6**	6.87	1.07	16	20.73	2.4	9	14.17	3.9	7	**6**	6	0
R1	58	64.03	3.47	86	96	4.91	62	77.97	7.68	50	43	44.5	0.57
R2	37	43.07	3.43	63	72.23	4.17	45	56.03	5.19	40	30	30.33	0.48
R3	25	30.07	2.69	45	53.53	3.94	27	39.07	6.72	19	18	18	0
R4	34	38.23	1.7	54	61.3	3.66	38	46.17	4.56	33	**28**	28.17	0.38
R5	15	19.7	2.56	32	42.03	3.58	19	27.73	5.02	15	**12**	12	0

Table 6.3: Minimizing the number of conflicts: performance of *Genitor*, local search and *SWO* when the **earliest start** schedule builder is used. Statistics for *Genitor*, local search and *SWO* are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

in the same relative order for all the requests. Therefore, when the same two alternative resources are specified for a sequence of tasks, the greedy scheduler will assign tasks to the first resource, and only when this is not possible anymore, the second resource will be assigned to the tasks. Imagine two shorter tasks and one longer task having to be executed on the two resources, such that the greedy scheduler would schedule the two shorter requests on the same resource and the longer one on the other requests. The earliest start scheduler will assign the two short requests on separate resources, if the long request appears after the short ones in the permutation, possibly resulting in not enough room left on any of the resources for the long request. In terms of CPU time, 8000 evaluations take approximately 16 seconds for the AFIT problems and 22 seconds for the current days (3 to 5 times longer than for the greedy scheduler).

Max-Availability Schedule Builder - No Update Version

Kramer and Smith [KS05] define a **max-availability** heuristic to identify the best starting time for a task; they report obtaining surprisingly good results when using the

	Genitor			*StructLS*			*RandLS*			*SWO*			
Day	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Init.	Min	Mean	SD
A1	136	181.57	25.04	497	635.73	79.2	247	400.56	92.6	128	**104**	104	0
A2	37	80.97	26.48	201	329	62.48	42	184.6	61.97	83	**13**	13	0
A3	48	110.9	25.35	221	358.3	66.8	83	189.97	54.82	66	**28**	28	0
A4	41	90.97	32.55	296	436.53	73.4	118	249.4	73.04	110	**9**	9.23	1.1
A5	37	83.23	25.93	194	324.33	67.09	85	163.43	55.25	85	**30**	30	0
A6	70	114.33	33.22	239	347.7	58.1	82	178.27	63.36	**45**	**45**	45	0
A7	57	85.77	15.34	232	356.33	58.11	90	204.63	91.95	108	**46**	46	0
R1	1378	1502.6	74.92	2140	2359.4	110.61	1275	1792.83	201.63	1219	813	834.83	11.0
R2	732	855.73	72.89	1402	1577.73	92.16	794	1112.7	179.02	653	497	507.1	6.43
R3	546	635.26	64.1	979	1168.77	100.27	456	764.83	122.25	522	265	266.8	1.69
R4	907	1026.77	56.44	1392	1544	103.96	926	1191.8	135.53	978	730	736.8	3.86
R5	240	298.1	37.07	566	689.03	66.96	263	415.63	119.9	148	**146**	146	0

Table 6.4: Minimizing the sum of overlaps: performance of *Genitor*, local search and *SWO* when the **earliest start** schedule builder is used. Statistics for *Genitor*, local search and *SWO* are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

heuristic in the context of AMC scheduling. Given an unassigned task, max-availability is based on computing the resource contention over the time window specified for the task and assigning the task to a spot with maximum resource availability.

While for the AMC scheduling problem the resources are multicapacitated (capacity greater than or equal to one), for our domain, the capacity of the resources (antennas) is one. Therefore, we adjusted the max-availability heuristic to fit the AFSCN scheduling domain. First, we defined the Total Potential Requirement for each resource. If a request specifies a resource as a possible alternative, at any point inside its time window it could require one unit of that resource. The Total Potential Requirement at a time t on a resource R is equal to the number of requests which specify as one of their alternatives a pair of the form (R, T^{Win}), with t included in T^{Win}. In other words, the requests for which a time slot containing time t on resource R is a scheduling alternative contribute one unit to the Total Potential Requirement for R. For example, consider again the toy problem in Figure 3.2. The Total Potential Requirement for any of the two antennas present at Ground Station 1 is shown in Figure 6.1 (following a notation introduced by Kramer and Smith [KS05]). Note that the durations for $R3$ and $R4$ are 3 and 7,

respectively; however, the whole time window is considered when computing the Total Potential Requirement.

Similarly to Kramer and Smith [KS05], we use a best-worst case metric when assigning the requests. Each time we build a schedule, for each request and each of the time intervals in which it can be assigned, we compute the maximum total potential requirement. In effect, we slide forward a window with the length equal to the duration over all its alternative time windows T^{Win} (on all the alternative resources). For each position of the sliding window, we compute the maximum total potential requirement during that time interval. The schedule builder chooses to schedule the request during the time interval with the smallest maximum total potential requirement. For example, based on Figure 6.1, for request $R3$, the following lists of values represent the Total Potential Requirement for each possible time interval (the duration of $R3$ is 3 and its time window starts at 4): $(3, 3, 4)$, $(3, 4, 4)$, $(4, 4, 4)$, $(4, 4, 3)$, $(4, 3, 3)$, $(3, 3, 2)$ and $(3, 2, 1)$. The smallest maximum value in these time intervals is 3, for the time intervals starting at time 9 and 10. Our schedule builder chooses the first such time interval encountered to schedule the request (for our example, $R3$ would start at time 9)[1]. Once the request is scheduled, we can update the Total Potential Requirement for all the other time units and resources specified as possible alternatives for the request (by decreasing it by one). We chose not to update the Total Potential Requirement for this first version of the schedule builder. We call this version the **Max-Availability-No-Update** schedule builder.

The results obtained using the Max-Availability-No-Update schedule builder are summarized in Tables 6.5 and 6.6. When minimizing the number of conflicts (see Table 6.5), *Genitor*, *RandLS*, and *SWO* perform well, with results similar to the ones obtained for the greedy schedule builder. In addition, *SWO* finds the best known solu-

[1] A better idea might have been to minimize the sum in the interval.

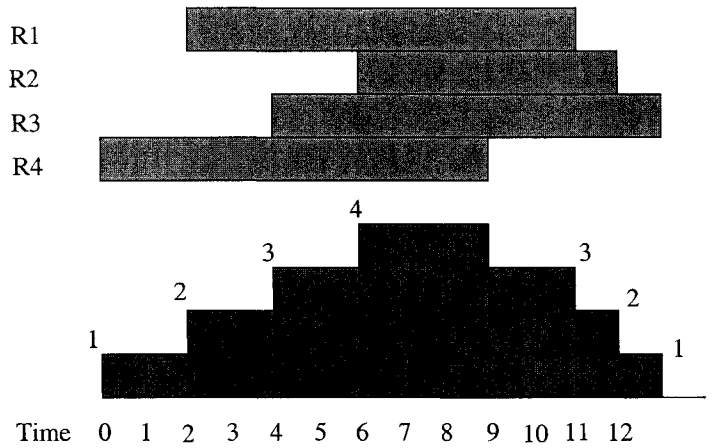


Figure 6.1: The total potential requirement for an antenna at Ground Station 1 (see the example problem in Figure 3.2). The time windows corresponding to each request are shown in the top figure. The total potential requirement is shown in the bottom figure.

	Genitor			*StructLS*			*RandLS*			*SWO*			
Day	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Init.	Min	Mean	SD
A1	**8**	8.3	0.46	10	11.9	1.73	**8**	8.53	0.51	**8**	**8**	8	0
A2	**4**	4	0	5	7.3	1.64	**4**	4	0	5	**4**	4	0
A3	**3**	3	0	7	8.97	1.73	**3**	3.03	0.18	**3**	**3**	3	0
A4	**2**	2.33	0.48	4	7.5	1.8	**2**	2.33	0.48	4	**2**	2	0
A5	**4**	4	0	4	9.37	2.0	**4**	4	0	**4**	**4**	4	0
A6	**6**	6.03	0.18	8	12	2.03	**6**	6	0	**6**	**6**	6	0
A7	**6**	6	0	**6**	7.97	1.27	**6**	6	0	**6**	**6**	6	0
R1	**42**	44.07	0.83	55	67.73	5.02	**42**	43.97	1.19	51	43	44.2	0.61
R2	**29**	30.67	0.71	43	49.47	3.08	**29**	30.63	1.07	38	**29**	30.2	0.55
R3	**17**	17.33	0.48	22	26.33	2.44	**17**	17.57	0.73	18	**17**	17	0
R4	**28**	28	0	34	37.37	2.28	**28**	28.53	0.68	30	**28**	28.23	0.43
R5	**12**	12	0	14	17.77	1.81	**12**	12.13	0.35	**12**	**12**	12	12

Table 6.5: Minimizing the number of conflicts: performance of *Genitor*, local search and *SWO* when the **Max-Availability-No-Update** schedule builder is used. Statistics for *Genitor*, local search and *SWO* are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

tion for R3 (it does not find it when using the simple schedule builder). The performance of *StructLS* improves when the new schedule builder is used; however, *StructLS* is still the worst performer in the set. When minimizing the sum of overlaps (see Table 6.6), *Genitor*, *StructLS* and *RandLS* perform better than when using the simple greedy scheduler. *SWO* performs worse for R1 and R2 and better for R3 and R4. While overall these

	Genitor			*StructLS*			*RandLS*			*SWO*			
Day	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Init.	Min	Mean	SD
A1	107	106.5	1.14	144	215.87	49.87	**104**	106	1.44	119	**104**	115.1	5.96
A2	**13**	13	0	27	78.23	38.29	**13**	13	0	48	**13**	13	0
A3	**28**	28.43	0.82	49	112.97	40.44	**28**	28.27	0.69	44	**28**	28.3	0.92
A4	**9**	9.13	0.73	29	121.2	35.24	**9**	9.2	1.1	54	**9**	12.43	6.82
A5	**30**	30.5	1.53	53	107.26	34.76	**30**	30.33	1.27	85	**30**	30	0
A6	**45**	45	0	86	167.2	46.3	**45**	46.03	2.75	**45**	**45**	45.1	0.31
A7	**46**	46	0	**46**	75.63	20.78	**46**	46	0	106	**46**	46	0
R1	858	901.86	27.11	1209	1566.03	135.87	774	813.3	22.2	1142	818	841.63	12.31
R2	507	530.83	11.7	791	1057.57	103.81	492	525.53	24.46	687	494	503.3	5.08
R3	253	261.1	4.98	453	677.83	83.4	**250**	306	61.86	491	**250**	252.43	1.96
R4	727	734.97	6.2	980	1107.9	70.93	**725**	738.97	16.6	942	726	732.83	3.82
R5	**146**	146	0	194	278.23	32.37	**146**	146.87	7.82	**146**	**146**	146	0

Table 6.6: Minimizing the sum of overlaps: performance of *Genitor*, local search and *SWO* when the **Max-Availability-No-Update** schedule builder is used. Statistics for *Genitor*, local search and *SWO* are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

performance results indicate that the Max-Availability-No-Update schedule builder results in better solutions, this improvement is obtained at the expense of much longer CPU times. Even though the Total Potential Requirement is computed before we start scheduling, every time we schedule a request, we have to examine its values for each of the possible time intervals (in order to determine the interval with the minimum maximum value). For high altitude requests, this could get quite expensive: for each alternative resource, $T_i^{Win}(UB) - T_i^{Win}(LB) - T_i^{Dur} + 1$ possible windows need to be examined (and there might exist up to 14 alternative resources). Performing 8000 evaluations with the Max-Availability-No-Update schedule builder takes approximately 63 seconds for an AFIT problem and approximately 119 seconds for a current problem (16 to 20 times longer than for the greedy scheduler). It is possible that fewer evaluations would have sufficed to obtain good results for some of the instances; we checked for a few instances and found that the last improvements actually happened close to 8000 evaluations.

The Total Potential Requirement is a rather coarse measure of the contention for the resource. A better measure is SumHeight [BDSF97], where for each request 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. With SumHeight, the potential requirement for a resource R at time t would be computed as a sum over all the requests of the probability of a request requiring R at time t. SumHeight would be computationally more expensive than max-availability. Because using max-availability with the schedule builder already takes 16 to 20 times longer than the greedy scheduler, we decided not to implement a SumHeight version.

Max-Availability Schedule Builder - Update Version

The Max-Availability-No-Update schedules are built based on the initial Total Potential Requirement. One would expect that as requests get scheduled, updating the Total Potential Requirement by removing the contribution of all the alternatives except for the one scheduled would more closely model the change in contention for resources. Therefore, we also implemented a version of the schedule builder using the max-availability heuristic for which we update the Total Potential Requirement every time a request gets scheduled, by decreasing it by one for each unit of time in each of the alternative time windows specified for that request (except for the one that got scheduled). Also, for this version of the schedule builder, we chose to break ties by random. If indeed the Total Potential Requirement is a good (albeit coarse) measure of contention, this new scheduler should perform even better than Max-Availability-No-Update.

Note that using SumHeight instead of max-availability would still be computationally more expensive. With SumHeight, when updating, the contribution of each of the possible alternatives to the aggregate curve needs to be subtracted. This contribution is expressed probabilistically for SumHeight; for max-availability it is equal to one. Even if the implementation of SumHeight caches the contribution of each request to the aggregate demand, some computation to retrieve the actual value from the cached one would still be needed.

The results obtained with the Max-Availability-Update scheduler are presented in

Day	*Genitor*			*StructLS*			*RandLS*			*SWO*			
	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Init	Min	Mean	SD
A1	**8**	8.03	0.18	11	15.23	2.3	**8**	9.27	1.56	9	**8**	8	0
A2	**4**	4	0	5	8.83	1.72	**4**	4.8	1.3	5	**4**	4	0
A3	**3**	3.07	0.25	5	10.83	2.3	**3**	4.9	1.52	4	**3**	3	0
A4	**2**	2.43	0.5	7	11.1	3.0	**2**	4.3	1.42	3	**2**	2	0
A5	**4**	4	0	6	10.97	2.31	**4**	5.47	1.3	4	**4**	4	0
A6	**6**	6.3	0.47	9	12.97	2.51	**6**	7.67	1.2	**6**	**6**	6	0
A7	**6**	6	0	**6**	7.8	1.58	**6**	6	0	**6**	**6**	6	0
R1	47	49.2	1.03	65	74.83	4.6	50	58.33	5.9	51	44	44.8	0.55
R2	30	31.73	0.87	46	52.87	4.3	34	41.33	5.03	37	**29**	30.1	0.55
R3	**17**	18.03	0.67	24	32.3	3.56	18	21.63	3.02	18	**17**	17.1	0.3
R4	**28**	28.93	0.74	33	40.57	3.22	**28**	32.67	4.07	31	**28**	28.1	0.3
R5	**12**	12.4	0.5	14	21.9	2.88	**12**	14.53	1.5	**12**	**12**	12	0

Table 6.7: Minimizing the number of conflicts: performance of *Genitor*, local search and *SWO* when the **Max-Availability-Update** schedule builder is used. Statistics for *Genitor*, local search and *SWO* are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

Day	*Genitor*			*StructLS*			*RandLS*			*SWO*			
	Min	Mean	SD	Min	Mean	SD	Min	Mean	SD	Init	Min	Mean	SD
A1	**104**	107.7	2.55	195	316.13	53.16	107	142.73	35.47	119	**104**	111.53	7.13
A2	**13**	15.53	2.06	74	131.43	40.53	**13**	27.13	16.27	104	**13**	13.37	2.01
A3	30	35.87	3.3	120	172.93	39.96	30	51.37	21.17	47	**28**	28.3	0.92
A4	**9**	16.77	5.3	80	166.87	56.13	**9**	49.93	35.22	34	**9**	12.23	6.83
A5	**30**	34.3	2.74	48	143.3	59.3	**30**	57.33	46.58	87	**30**	30	0
A6	**45**	50.2	4.24	121	204.97	54.5	**45**	86.1	26.87	**45**	**45**	45.03	0.18
A7	**46**	46	0	53	103.93	33.24	**46**	49.63	6.44	106	**46**	46	0
R1	1036	1111.93	38.55	1551	1757.27	128.75	1014	1287.07	172.78	1179	820	840.07	12.33
R2	552	596.83	20.55	1015	1166.5	87.09	684	832	84.19	689	495	509	5.28
R3	283	308.3	13.13	555	776.5	89.8	304	466.57	96.37	509	**250**	252.37	1.99
R4	742	769.3	10.55	1058	1205.73	77.58	771	871.2	86.02	947	729	735	2.9
R5	**146**	153.5	4.81	268	357.7	46.12	**146**	200.17	35.53	**146**	**146**	146	0

Table 6.8: Minimizing the sum of overlaps: performance of *Genitor*, local search and *SWO* when the **Max-Availability-Update** schedule builder is used. Statistics for *Genitor*, local search and *SWO* are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

Tables 6.7 and 6.8.

First, we compare the Max-Availability-Update schedule builder with the greedy schedule builder. When minimizing the number of conflicts, with the exception of R1 and R2, *Genitor*, *RandLS* and *SWO* show performance results similar to the ones

obtained with the greedy scheduler. For both objective functions, the performance of *StructLS* improves over the results produced with the greedy scheduler; however, *StructLS* is still the worst performer in the set. Comparing with the greedy scheduler for minimizing the overlaps, *Genitor* and *RandLS* either find equally good best solutions or worse (for R1, R2, R3 and R4). *SWO* performs worse for R1 and R2 and better for R3 and R4; similarly to the greedy scheduler, it finds best known solutions for all the old days. The Max-Availability-Update schedule builder produces worse results than the greedy schedule builder.

Second, we compare the Max-Availability-Update schedule builder with the Max-Availability-No-Update schedule builder. For most days of data (especially for the new days), the Max-Availability-Update scheduler worsens the performance of all the algorithms, for both objective functions. This is a surprising result: one would expect that updating the contention will help improve algorithm performance; the results show this is not the case. The Max-Availability-No-Update scheduler uses the max-availability heuristic only to compute an initial contention profile. With each request scheduled, the profile becomes less accurate. When enough requests are scheduled, the best-worst interval to schedule a request has been most likely already assigned; the scheduler relies less on the profile and more on the available spots in the schedule to schedule a request. This behavior should be prevented by updating the contention profile after each request is scheduled, as it is done in the Max-Availability-Update schedule builder. However, the results show that updating the contention profile results in worse performance. We conjecture that our approximation to a contention profile using the Total Potential Requirement is too coarse and therefore it is not reliable in indicating the best position to schedule a request.

Note that there are two main differences between the Max-Availability-Update and the Max-Availability-No-Update schedule builders. The Max-Availability-Update

scheduler is updating the contention every time a request is scheduled, but it also breaks ties by random. We implemented an initial version such that it would always choose the first available spot (like Max-Availability-No-Update does). We found that it performed worse than Max-Availability-No-Update; we introduced randomization in an attempt to improve the results. It turns out that choosing the first available spot for the request satisfying the best-worst max-availability measure seems to be less disruptive than choosing such a spot by random (when multiple choices are possible). We conjecture that the reason for this is similar to the reasons why the **earliest start** scheduler performs worse than the greedy scheduler: when choosing the spot to schedule a request by random, the blocks of free time on the resources are more fragmented. Choosing the first available spot is likely to schedule a request on the same resource as a previously scheduled request specifying similar alternatives. This leaves more free time on the alternative resources to schedule longer requests.

In fact, randomization when implementing a schedule builder can result in problems because of the unpredictability of the value assigned to a permutation. For example, Shaw and Fleming [SF97] argue that the use of randomization in a schedule builder can be detrimental to the performance of a genetic algorithm when an indirect representation is used (for which the chromosomes are not schedules). They support this idea by noting that in general, genetic algorithms rely on the preservation of solution patterns that contribute to good solution values. Preserving such patterns (using the crossover mechanism) depends on the schedule builder to systematically assign good values to solutions containing these patterns. Also, for *SWO*, randomization in the schedule builder changes the significance of reprioritization from one iteration to the next one. If the scheduler is randomized, the new order of requests is very likely to result in a schedule that is not the "repaired version" of the previous one. If the same permutation of requests can be transformed into multiple different schedules because of the nondeterministic nature of

the scheduler, the *SWO* mechanism will not operate as intended.

6.3 Hill-Climbing Performance Factors

Based on our earlier experiments (see Section 5.2) we know that the two versions of hill-climbing we implemented, *StructLS* and *RandLS* are dramatically different in terms of performance results: while *StructLS* performs poorly, *RandLS* is one of the best performing algorithms. In this section, we explore a number of hypotheses to explain these performance results:

1. We hypothesize that plateaus dominate the search space. We offer evidence to support this hypothesis by showing that the search space is highly redundant: given any solution S in the search space, approximately 60% of its neighbors result in schedules that are identical to the one corresponding to S.

2. The two versions of hill-climbing differ by the ordering of the neighbors: while *StructLS* imposes a certain structure over the ordering, *RandLS* checks the neighbors in a random order. Obviously the ordering of the neighbors matters. We hypothesize that the poor performance of *StructLS* is a result of checking the neighbors in a systematic order.

3. We hypothesize that it is possible to restrict the neighborhood, by eliminating some or all of the permutations corresponding to identical schedules. While one would hope that such a reduction will result in more efficient search, it is not clear that is the case. We explore possibilities of reducing the neighborhood size and investigate how this affects hill-climbing performance.

4. Both *StructLS* and *RandLS* are initialized with random solutions. We hypothesize that initializing the search with a greedy solution improves the performance for both algorithms.

6.3.1 Pairwise Interactions Between Requests

In this subsection, we address our first hypothesis in explaining the performance of *RandLS* and *StructLS*: the search space is dominated by plateaus. For both *RandLS* and *StructLS*, we analyze the shifting neighborhoods of various solutions. We find that a high number of the neighbors result in schedules identical to the current one; this offers evidence in support of our hypothesis.

The shifting neighborhood is produced by considering all possible pairs of positions (x, y) in the solutions (permutations) and moving the request from position x to position y. The size of this neighborhood is quite large ($(n-1)^2$, where n is the total number of requests). However, we expect that not all pairs of positions result in a change in the schedule. If for some pairs of positions the schedule does not change, how often does this happen?

We generate all possible shifting neighbors given a solution (permutation) and convert the resulting permutations into schedules. We count how many resulting schedules are identical to the one obtained from the original permutation. If by shifting the request in position x into position y there is no change in the schedule, then there is no interaction between that request and the request initially in position y. We call such a pair of requests a "non-interacting pair". To check how the number of non-interacting pairs varies with solution quality, we generate all possible shifting neighbors in random as well as best known solutions. The statistics are computed over 30 permutations for each problem. We compute the average number of pairs of requests for which the corresponding shift does not change the schedule (the number of non-interacting pairs).

The results of the pairwise sensitivity test are summarized in Tables 6.9 and 6.10. The second column *Total Pairs* represents the total number of shifting pairs in the permutation (for n requests, the number in this column is $(n-1)^2$). The mean of the number of non-interacting pairs of requests (from 30 random or optimal permutations)

Day	Total Pairs	Minimizing Conflicts							
		Random Perms				Optimal Perms			
		Mean	Stdev	Avg %	Stdev	Mean	Stdev	Avg %	Stdev
A1	103041	62868.3	1090.58	61.01	1.06	64117.6	920.083	62.2	0.89
A2	90601	57872.6	1174.17	63.88	1.3	58507.5	1152.03	64.58	1.27
A3	95481	58751.2	1388.61	61.53	1.45	58730.1	1103.13	61.5	1.16
A4	100489	59670.6	1149.49	59.38	1.14	60578.6	1099.98	60.28	1.09
A5	92416	55724.1	1057.66	60.3	1.14	55109.4	1144.37	59.63	1.24
A6	88804	52627.8	1169.15	59.26	1.32	52076.5	839.71	58.64	0.94
A7	87616	55082	1203.62	62.87	1.37	55650.1	1245.25	63.52	1.42
R1	232324	130865	2389.74	56.33	1.02	132192	1814.99	56.9	0.78
R2	207936	115237	1780.22	55.42	0.86	115703	1761.79	55.64	0.85
R3	180625	104296	2201.71	57.74	1.22	105592	1463.72	58.46	0.81
R4	184900	107464	2367.38	58.12	1.28	109703	1757.1	59.33	0.95
R5	174724	102994	1852.47	58.95	1.06	104960	1329.21	60.07	0.76

Table 6.9: Statistics for the number of pairs of non-interacting requests over 30 random and optimal permutations, for minimizing conflicts.

Day	Total Pairs	Minimizing Overlaps							
		Random Perms				Optimal Perms			
		Mean	Stdev	Avg %	Stdev	Mean	Stdev	Avg %	Stdev
A1	103041	61020.8	1185.87	59.2	1.15	64254.9	1130	62.36	1.1
A2	90601	56384.7	1469.37	62.23	1.62	58449.4	1001.17	64.51	1.1
A3	95481	57548.4	1332.14	60.27	1.39	58261.2	1016.59	61.01	1.06
A4	100489	57753	1013.56	57.47	1.0	59798.4	1335.21	59.5	1.33
A5	92416	54919.6	939.57	59.43	1.01	55608.1	814.544	60.17	0.88
A6	88804	50830.2	1200.2	57.24	1.35	52165.7	932.252	58.74	1.05
A7	87616	53836	1382.23	61.44	1.58	56110.7	1025.5	64.04	1.17
R1	232324	121193	2465.61	52.17	1.06	125578	2328.47	54.05	1.0
R2	207936	108313	2257.53	52.08	1.09	113781	1831.82	54.72	0.88
R3	180625	99797.4	2427.59	55.25	1.34	100751	1392.26	55.78	0.77
R4	184900	103315	2129.88	55.88	1.52	107692	1812.91	58.24	0.98
R5	174724	100766	2291.13	57.67	1.31	104097	1758.74	59.58	1.0

Table 6.10: Statistics for the number of pairs of non-interacting requests over 30 random and optimal permutations, for minimizing overlaps.

appears in the columns denoted *Mean*, with standard deviation in the following column. 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*), with

the corresponding standard deviation in the next column.

The results show that: 1) Approximately 60% of all possible shifts result in identical schedules. This means that for any permutation in the search space, approximately 60% of its neighbors correspond to schedules identical to its own. This shows that the plateaus are present in the search space. 2) The quality of the solution does not seem to affect the number of non-interacting requests. This means that not only random solutions, but good solutions as well, are located on plateaus. 3) In general, minimizing the number of conflicts results in more non-interacting pairs than minimizing the overlaps. 4) For the R problems, a smaller number of non-interacting requests is present than for the A problems.

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 suggests that we cannot reduce the search space size by simply not considering certain requests.

6.3.2 *RandLS* versus *StructLS*: The Ordering of the Neighbors

In this section we explore the hypothesis that the systematic order in which *StructLS* generates and examines the neighbors is the reason for its poor performance. We start by emphasizing the differences between *StructLS* and *RandLS*.

As described in Section 4.4.2, *StructLS* chooses by random the position x of the

request to be shifted and then evaluates the neighbors produced by shifting in *all* possible positions (1, 2, 3,...,n), accepting the new solutions if better or equally good.

Consider a permutation of $n-1$ from the total of n requests. Suppose *StructLS* chose to examine the shifting neighbors corresponding to the n-th request X. To produce the first neighbor, X is inserted in the first position in the permutation and the schedule builder is applied: a schedule S is obtained. Next, scanning the permutation of requests from left to right, X is successively inserted in the second position, then the third and so on; every time, a schedule is built. 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.

This turns out to be particularly inefficient: shifting a request into consecutive positions is likely to result in identical schedules. As long as there is no competition for resources between the shifted request and the requests in the positions where it is shifted, the schedule will not change. We define 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.

Consider the case where *StructLS* encounters a neighbor that is worse than the current value of the solution. Shifting the same request in subsequent positions will not change the schedule as long as there is no competition between the shifted request and the requests in the subsequent positions. The schedule corresponding to the next neighbors examined by *StructLS* will not change and therefore will also be worse, until a position is found that interacts with the shifted request. In effect, for *StructLS*, when 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.

On the other hand *RandLS* generates each neighbor by choosing by random *both* the shifted request and the position where it will be shifted. As shown in the previous section, at least 50% of the neighbors correspond to schedules identical to the current one. In fact, even more neighbors correspond to schedules of the same value as the current one (see Section 7.2.1). Consider the case where *RandLS* encounters a worse neighbor. Since the next neighbor is obtained by randomly selecting again a new request to be shifted and a new position, odds are that the next neighbor will have a value that is identical to the current best. The probability of having the next neighbor identical to the current worse one is small.

To better understand why *RandLS* results in such a great improvement over *StructLS*, we count 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 collect the counts for 30 runs of *RandLS* and *StructLS*, with 8000 evaluations per run (these runs are identical to the ones producing the results in Tables 5.6 and 5.8). We perform the experiment both for minimizing the number of conflicts and for minimizing overlaps. There exist approximately 20 times more possible objective function values for minimizing overlaps compared to minimizing conflicts; we expected minimizing

	% Worse				% Equal				% Better			
	RandLS		StructLS		RandLS		StructLS		RandLS		StructLS	
Day	Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev
A1	16	0.73	83.6	1.58	83.7	0.75	16	1.61	0.3	0.06	0.2	0.04
A2	14.4	0.59	82.2	2.5	85.4	0.6	17.5	2.5	0.2	0.03	0.1	0.03
A3	16.8	0.75	82.2	2.12	82.9	0.74	17.1	2.13	0.3	0.03	0.1	0.04
A4	19.3	0.88	84.8	1.8	80.4	0.87	14.8	1.8	0.3	0.04	0.1	0.04
A5	17.4	0.71	83.6	1.87	82.3	0.71	16.0	1.84	0.3	0.03	0.1	0.04
A6	18.0	0.54	84.3	1.85	81.7	0.53	15.4	1.83	0.3	0.04	0.1	0.04
A7	14.3	0.64	82.4	1.86	85.5	0.64	17.2	1.83	0.2	0.03	0.1	0.03
R1	21.9	0.83	87.2	1.94	77.5	0.83	12.5	1.96	0.6	0.06	0.2	0.05
R2	19.6	0.95	85.2	1.69	79.8	0.95	14.5	1.69	0.6	0.05	0.1	0.04
R3	17.8	0.67	85.2	2.06	81.8	0.66	14.4	2.05	0.4	0.06	0.2	0.05
R4	16.7	0.58	84.4	2.04	82.9	0.58	15.3	2.02	0.4	0.05	0.1	0.05
R5	13.8	0.707	83.0	2.19	85.9	0.7	16.6	2.22	0.3	0.04	0.1	0.05

Table 6.11: Minimizing conflicts: Average percentage of evaluations (out of 8000) resulting in worse, equally good or improving solutions over 30 runs of *RandLS* and *StructLS*.

overlaps to find a higher number of improving solutions during search than minimizing conflicts. The results are summarized in Tables 6.11 and 6.12. As expected, the number of worse neighbors evaluated during search is significantly higher for *StructLS*. Both versions of search accept most of the time equally good moves; only a small number of improving steps are taken. However, since *StructLS* spends more than 80% of the time evaluating worse moves, it is left with less than 20% of the evaluations (or even less than that when minimizing overlaps) to move through the space. *StructLS* needs more evaluations to find good solutions. *RandLS* always finds more improving neighbors (twice as many as *StructLS*, or even more).

The ordering of the neighbors for *StructLS* does not prevent it from eventually finding good solutions; while it does make the search inefficient, given enough evaluations, *StructLS* does find best known solutions. This also hints to the possibility that if one could only eliminate from the neighborhood some of the redundancy, fewer evaluations would be wasted on identical schedules and best known solutions might be found faster. We address the question of reducing the neighborhood size in the next subsection. By in-

	% Worse				% Equal				% Better			
	RandLS		StructLS		RandLS		StructLS		RandLS		StructLS	
Day	Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev
A1	20.1	0.81	87.8	1.43	79.1	0.82	11.7	1.44	0.6	0.1	0.3	0.07
A2	16.6	0.56	86.0	1.28	82.8	0.56	13.6	1.24	0.5	0.02	0.2	0.08
A3	19.3	0.55	86.8	1.62	80.0	0.57	12.7	1.55	0.6	0.06	0.3	0.1
A4	21.3	0.69	88.5	0.9	77.8	0.7	11.0	0.9	0.7	0.08	0.3	0.07
A5	19.5	0.61	87.3	1.48	79.8	0.61	12.2	1.46	0.6	0.08	0.3	0.06
A6	21.3	0.75	87.5	1.58	77.9	0.76	11.9	1.54	0.6	0.06	0.3	0.06
A7	16.8	0.61	85.7	1.41	82.6	0.64	13.8	1.3	0.5	0.07	0.3	0.06
R1	32.2	0.87	91.0	0.99	65.8	0.91	8.5	0.99	1.8	0.19	0.4	0.07
R2	26.8	0.66	90.4	1.39	71.7	0.69	9.1	1.36	1.3	0.16	0.3	0.07
R3	24.7	1.11	89.9	1.29	74.0	1.22	9.6	1.3	1.1	0.16	0.3	0.08
R4	23.6	0.74	89.5	1.4	75.2	0.73	10.0	1.4	1.0	0.1	0.3	0.07
R5	17.9	0.54	87.8	1.5	81.3	0.52	11.6	1.49	0.7	0.1	0.3	0.06

Table 6.12: Minimizing overlaps: Average percentage of evaluations (out of 8000) resulting in worse, equally good or improving solutions over 30 runs of *RandLS* and *StructLS*.

Day	Minimizing Conflicts			Minimizing Overlaps		
	Min	Mean	Stdev	Min	Mean	Stdev
A1	**8**	8.4	0.49	**104**	106.1	1.39
A2	**4**	4.0	0.0	**13**	13.0	0.0
A3	**3**	3.03	0.18	**28**	28.2	1.09
A4	**2**	2.03	0.18	**9**	9.13	0.73
A5	**4**	4.4	0.49	**30**	30.57	0.5
A6	**6**	6.0	0.0	**45**	45.03	0.18
A7	**6**	6.0	0.0	**46**	46.16	0.91
R1	**42**	43.93	1.08	792	883.53	48.87
R2	**29**	29.53	0.73	502	525.0	22.09
R3	**17**	18	0.74	**250**	309.36	52.89
R4	**28**	28.16	0.38	**725**	754.06	29.49
R5	**12**	12.46	0.5	**146**	147.16	4.41

Table 6.13: Statistics for the results obtained in 30 runs of *StructLS*, with 500,000 evaluations per run. The mean and best value from 30 runs as well as the standard deviations are shown.

creasing the number of evaluations to 500,000, 30 runs of *StructLS* produce best known solutions for all problems when minimizing conflicts and for all except R1 and R2 when minimizing sum of overlaps (see Table 6.13).

The numbers in Tables 6.11 and 6.12 also highlight the differences in the search

spaces corresponding to the two objective functions: when minimizing the sum of overlaps, the percentages of worse and also better neighbors encountered during search are slightly higher than for minimizing conflicts. This result is as expected.

The different discretization in the search spaces corresponding to the two objective functions also corresponds to two times more improving moves needed to find best known solutions when minimizing the overlaps. One might argue that the two times as many improvements found when minimizing overlaps versus minimizing conflicts are too few, given the number of unique values in each search space (20 times more values when minimizing overlaps). However, when the objective is to minimize the sum of overlaps, the improvements translate to jumping over some of the possible objective function values: there is a number of other possible but skipped-over values between the value of the current solution and the value of the new solution. We observed a higher number of skipped-over unique solution values in the beginning of the search; as search focuses, getting closer to the best solution, the number of such solution values diminishes. When minimizing conflicts, starting with the value of the initial solution, most of the possible values for the objective function are encountered during search.

6.3.3 Reduced Size Neighborhood

Reduced size neighborhoods have been successfully implemented for classical scheduling problems such as flow-shop and job-shop. By reducing the size of the neighborhood while preserving the ability to find improving solutions, high quality solutions can be found in a shorter time.

To reduce the size of the shifting neighborhood, we impose restrictions on the positions and ordering of the requests in the solutions (permutations), based on their time windows and alternative resources. This is motivated by the fact that 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). 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. Imposing restrictions on the possible positions for a request in the permutation does limit the number of shifting moves allowed for that request, resulting in a smaller number of neighbors. The question we address here is: can we define such a restricted neighborhood that will result in improved performance results for hill-climbing?

We define a restricted shifting operator by narrowing the movement of each request based on its time window. In order to define such an operator, the initial solution as well as all the solutions generated must preserve certain relative orders between the requests. Therefore, the restricted shifting 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 shifting 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 hill climbing using the restricted shifting operator both for minimizing conflicts and for minimizing overlaps in Table 6.14. We expected the restricted shifting operator to result in values at least as good as the ones found using shifting. However, experiments performed using 30 trials for each version of hill climbing and 8000 evaluations per trial resulted in poor values. More than 8000 evaluations are needed to find good solutions when using the restricted

shifting operator. All the other algorithms in our test set are allowed 30 runs of 8000 evaluations per run. To be fair in the comparison with the other algorithms while also allocating more evaluations per trial, we allocated 16000 evaluations for each trial but only allowed for 15 trials. This results in exactly the same number of total evaluations allowed ($15 * 16000 = 30 * 8000$). 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. While the hill-climber using the restricted shifting operator outperforms *StructLS*, it performs worse than *RandLS*. In fact, we ran hill climbing with the restricted shifting operator using 30 trials, with a large number of evaluations (500,000 evaluations) per trial. Local search using the restricted shifting operator did not find best known values for any of the problems (see Table 6.15).

We investigate the reason why hill climbing using the restricted shift operator doesn't seem to be able to reach the best known solutions. While ideally we would like the restricted shifting operator to only eliminate the redundant permutations from the search space, we find 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

Day	Minimizing Conflicts				Minimizing Overlaps			
	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	773	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 6.14: Results of running restricted hill climbing in 15 experiments, by evaluating 16000 permutations per experiment.

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 that does not appear to contain the best solutions.

6.3.4 The Role of Initialization

Starting closer to the best known values might help the performance of hill-climbing. Alternatively, hill-climbing might get stuck and be incapable of finding solutions as good as the ones found by starting with random permutations. Does the initialization help or hurt hill-climbing performance? Also, comparing the evolutions of *SWO* and *RandLS* (see Section 5.3), we noted that *SWO* descends very fast to good solutions in the space. Would the seeded versions of hill-climbing find good solutions at a comparable rate if they are initialized with the same permutations as *SWO*?

We investigate the changes in the performance of *RandLS* and *StructLS* when initialized with the same initial solution as *SWO*. We call these versions of hill climbing

Day	Minimizing Conflicts				Minimizing Overlaps			
	Best Known	Min	Mean	Stdev	Best Known	Min	Mean	Stdev
A1	8	10	10.53	0.5	104	139	139.0	0.0
A2	4	5	5.03	0.18	13	20	30	12.45
A3	3	5	5.8	0.4	28	39	39.7	1.51
A4	2	9	9.76	0.43	9	47	50.33	4.79
A5	4	6	6.0	0.0	30	55	72.73	10.32
A6	6	7	8.4	0.62	45	47	56.16	11.46
A7	6	7	7.0	0.0	46	78	90.76	21.74
R1	42	43	44.13	0.68	773	816	858.86	20.66
R2	29	32	33.5	0.73	486	499	526.76	24.42
R3	17	21	21	0.0	250	328	462.9	52.23
R4	28	30	32.23	0.77	725	748	761.26	9.33
R5	12	15	15.43	0.57	146	168	177.63	6.33

Table 6.15: Statistics for the results obtained in 30 runs of restricted hill climbing, with 500,000 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.

Seeded RandLS and *Seeded StructLS*.

To answer the first question, we compare the results of random initializations (Tables 5.6 and 5.8) to those of running *Seeded RandLS* and *Seeded StructLS*. The results are shown in Tables 6.16 and 6.17.

The performance of *StructLS* considerably improves when seeded with the greedy solutions. For minimizing the number of conflicts, *Seeded StructLS* finds best known solutions for all the A problems, as well as for R4 and R5. For minimizing the sum of overlaps, *Seeded StructLS* finds best known solutions for six of the A problems and for R5. For the rest of the problems, the mins, means and standard deviations are all smaller than those obtained by *StructLS*.

When minimizing the number of conflicts, *RandLS* produces best known solutions even when started from a random initial solution; as we will show next, initialization speeds up the progress of the algorithm toward the solution. When minimizing the sum of overlaps, *Seeded RandLS* finds best known solutions for all the problems (*RandLS*

Day	*Seeded StructLS*			*Seeded RandLS*		
	Min	Mean	Stdev	Min	Mean	Stdev
A1	**8**	8.63	0.67	**8**	8.43	0.5
A2	**4**	4	0.0	**4**	4	0.0
A3	**3**	3.1	0.3	**3**	3	0.0
A4	**2**	2.2	0.4	**2**	2	0.0
A5	**4**	4	0.0	**4**	4	0.0
A6	**6**	6.07	0.25	**6**	6	0.0
A7	**6**	6.73	0.45	**6**	6.03	0.18
R1	44	45.53	1.33	**42**	43.07	0.78
R2	31	32.43	0.97	**29**	29.43	0.57
R3	18	18.97	0.76	**17**	17.67	0.6
R4	**28**	29.3	0.79	**28**	28	0.0
R5	**12**	12.03	0.18	**12**	12	0.0

Table 6.16: Performance of hill climbing when initialized from greedy permutations. The objective function is minimizing the number of conflicts. All statistics are taken over 30 independent runs, with 8000 evaluations per run.

does not find best known solutions for R1 and R2). In fact, *Seeded RandLS* is the only algorithm in our set to find the best known for R1 in 30 runs, with 8000 evaluations per run.

To answer the second question, we show the progression of *RandLS* and *StructLS* and compare it with their random-start versions as well as *Genitor* and *SWO* by generating graphs similar to the ones in section 5.3. A typical example for each objective function is presented in Figures 6.2 and 6.3, respectively. For both objective functions, *Seeded StructLS* finds small improvements to the current solutions and spends a long time before such improvements are found; even though it starts with a random solution, *RandLS* quickly descends to better solutions and outperforms *Seeded StructLS*. Similarly to the evolution of *RandLS* and *StructLS*, *Seeded RandLS* finds betters solutions a lot faster than *Seeded StructLS* does. *Seeded RandLS* does not find improvements as fast as *SWO* does in the beginning, however, there is a crossover point between the two algorithms. *SWO* performance levels off while *Seeded RandLS* continues to find small

Day	*Seeded StructLS*			*Seeded RandLS*		
	Min	Mean	Stdev	Min	Mean	Stdev
A1	107	112.03	5.97	**104**	106	1.44
A2	**13**	15.33	5.96	**13**	13	0.0
A3	**28**	28.4	2.19	**28**	28	0.0
A4	**9**	9.97	3.95	**9**	9	0.0
A5	**30**	32.3	5.19	**30**	30	0.0
A6	**45**	45.97	2.41	**45**	45	0.0
A7	**46**	62.27	14.73	**46**	46	0.0
R1	857	931.37	43.97	**773**	776.67	5.58
R2	523	560.17	20.53	**486**	491.2	6.58
R3	274	367.27	49.97	**250**	251	5.48
R4	730	781.13	34.73	**725**	735.7	15.54
R5	**146**	147.07	2.61	**146**	146	0.0

Table 6.17: Performance of hill climbing when initialized from greedy permutations. The objective function is minimizing the sum of overlaps. All statistics are taken over 30 independent runs, with 8000 evaluations per run.

improvements.

To summarize the effect of initialization, we found that: 1) Initialization improves the performance of both *RandLS* and *StructLS*; in fact, *Seeded RandLS* is the only algorithm from the ones we implemented that finds best known solutions for all the problems and both objective functions in 30 runs with 8000 evaluations per run. 2) Initialization also speeds up the progress of hill-climbing toward the solution. While the *Seeded RandLS* is still not as fast as *SWO* in finding improvements in the beginning of the search, it does continue to improve average best so far solutions after *SWO* performance levels off.

6.4 Genitor Performance Factors

We hypothesize that *Genitor* performs well because it identifies patterns of request ordering: certain requests that must come before other requests. To test this, we examine

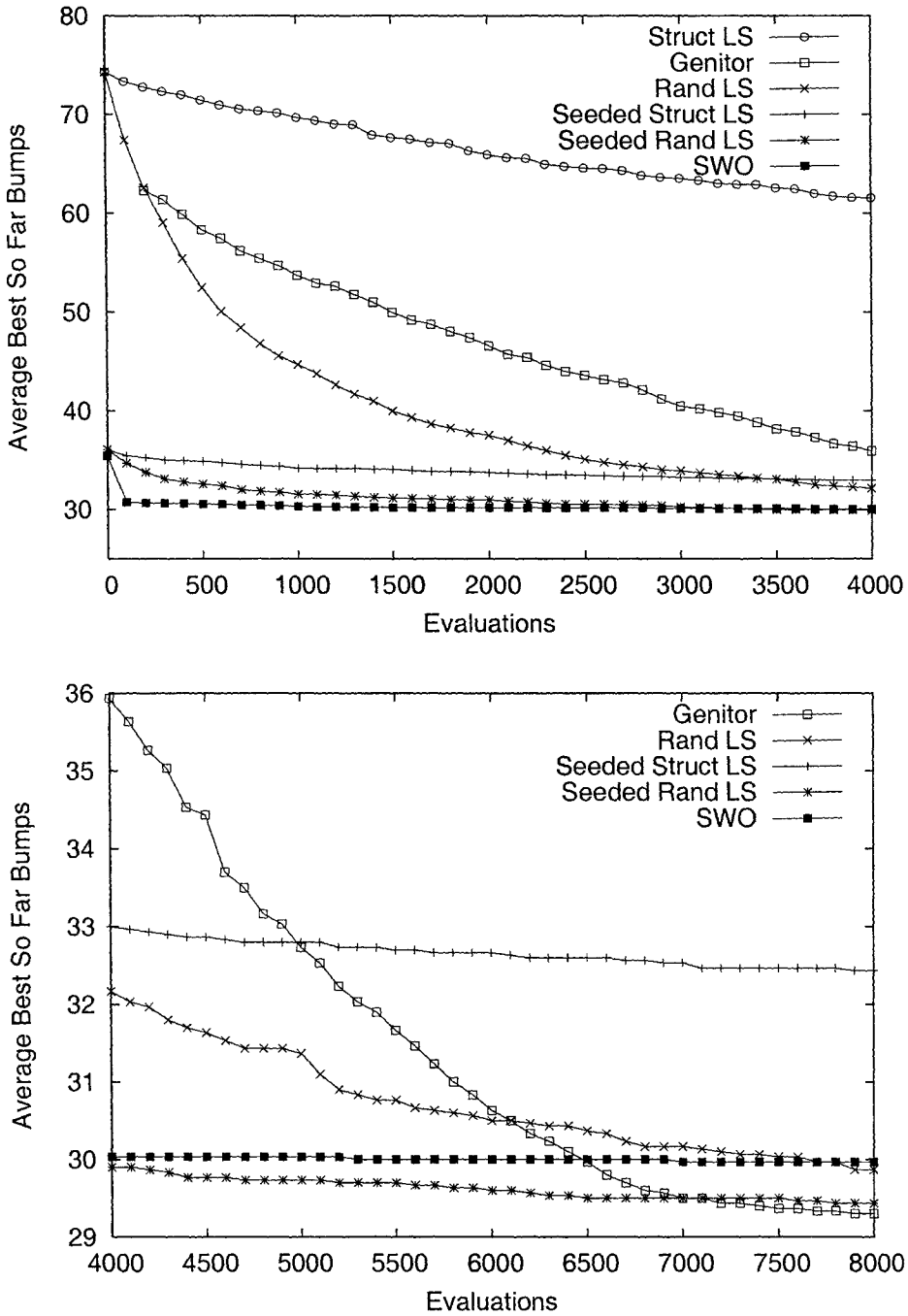


Figure 6.2: Evolutions of the average best value for conflicts obtained by *SWO*, hill-climbing initialized with random and greedy solutions and *Genitor* during 8000 evaluations, over 30 runs. Note that the scale is different on the y-axis for the last 4000 evaluations (bottom graph).The graphs were obtained for $R2$; best solution value is 29.

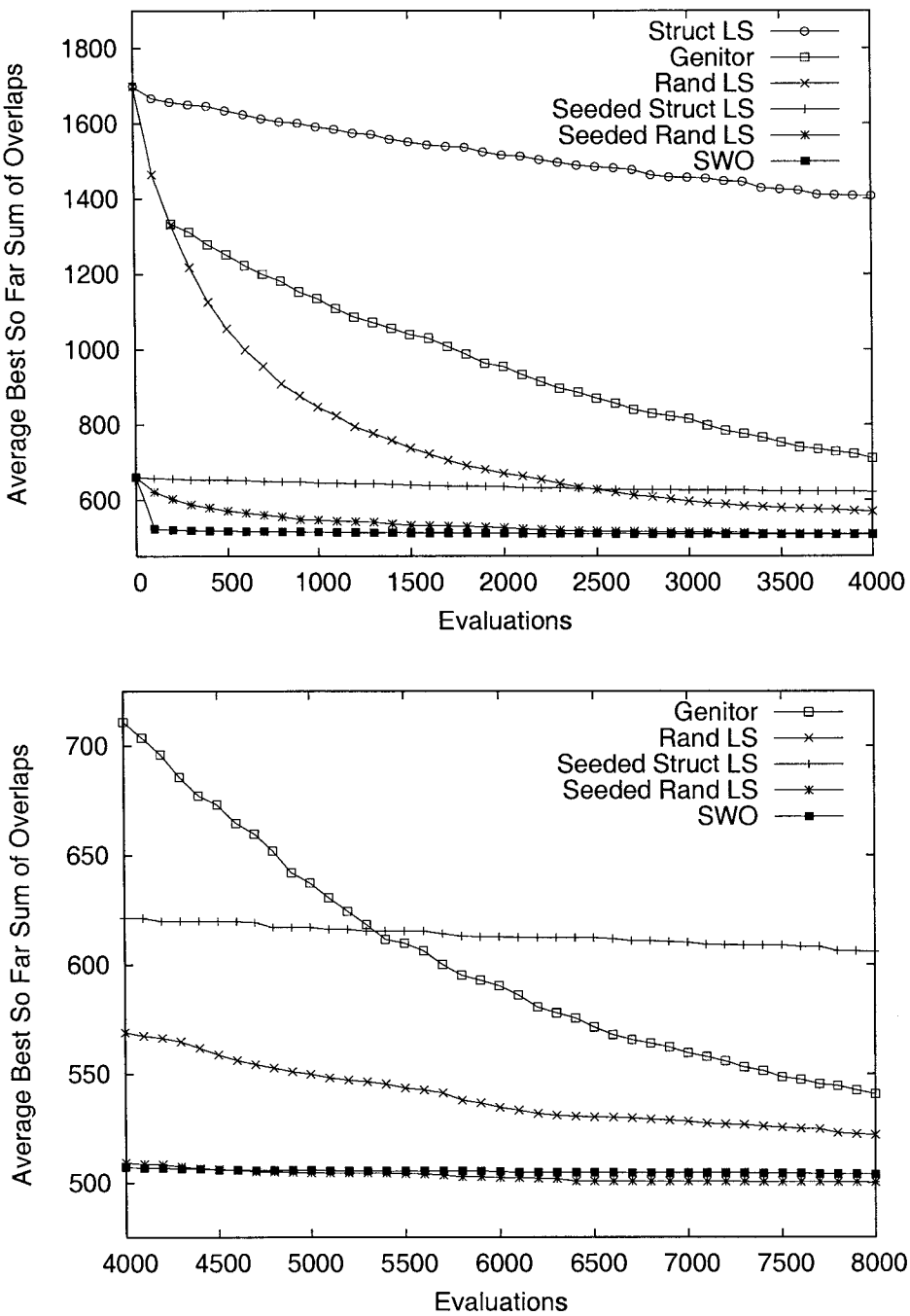


Figure 6.3: Evolutions of the average best value for sum of overlaps obtained by *SWO*, hill-climbing initialized with random and greedy solutions and *Genitor* during 8000 evaluations, over 30 runs. Note that the scale is different on the y-axis for the last 4000 evaluations (bottom graph).The graphs were obtained for $R2$; best solution value is 486.

the final solutions produced by different runs. We show that one simple pattern emerges: low altitude requests appearing before the high altitude requests. This pattern is closely related to the split heuristic defined in Section 5.2.1; we investigate the possibility of improving the performance of *Genitor* by using the split heuristic when evaluating the permutations. Finally, given the competitive advantage of initialization in *SWO*, we address the question: Does initialization also improve *Genitor*'s performance?

6.4.1 Patterns of Request Ordering

The patterns of request ordering (where certain requests must appear before others in the permutation) are closely related to the concept of *backbone* [AGKS00, SGS00] of a problem. A backbone is represented by a set of variables such that in all the optimal solutions, each variable is assigned the same value. The idea behind the backbone concept is that the majority of effort during search might be spent finding the correct assignment for backbone variables, such that once these variables are assigned, optimal solutions are easily found. For our problem, we can define backbone variables to represent precedence relations between pairs of requests (for example, given two requests A and B, precedence specifies that A is before B or B is before A). Therefore, a backbone in our domain would be a set of such precedence relations that are identical for all optimal solutions, or "patterns of request ordering".

To identify the backbone for a given problem, the set of **all** optimal solutions needs to be computed. This is not possible for our domain: in fact, the best value we report for all the problems are "best knowns"; we don't know for a fact that these are optimal values. Instead, we examine sets of final solutions that correspond to good schedules and identify chains of common request orderings in these permutations. The presence of such chains would support the hypothesis that *Genitor* is discovering patterns of request orderings.

To test for the presence of patterns of request ordering, we ran 1000 trials of *Genitor*.

From each final population, we selected one best solution. Looking at the set of 1000 solutions, we decided for which solution value we will attempt to identify patterns of request orderings. When we could not find enough best-known valued solutions in the set of 1000, we selected the next good solution value, such that more than 30 solutions of identical value are present in the set. We relax this rule for R1, for which the values vary from 880 to 1084 (when minimizing overlaps). 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 A1, we selected the solutions corresponding to best known values for the A problems. For A1, there were only two solutions corresponding to the best known value (104) and 922 solutions of value 107.

First, we checked for request orderings of the form "requestA before requestB". The results are summarized in Table 6.18. 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 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 request (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 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 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 results in Table 6.18 are heavily biased by the number of solutions considered[2]. Indeed, let s denote the number of solutions of identical value (the number in column *# of Solutions*). Also, let n denote the total number of requests. Suppose there are no preferences of orderings between the tasks in good solutions. For a request ordering A before B there is a probability of $1/2$ that it will be present in one of the solutions, and therefore, a probability of $1/2^s$ that it will be present in all s solutions. Given that there exist n^2 possible precedences, the expected number of common orderings if no preferences of orderings between tasks exist is $n^2/2^s$. For the A problems and for R5, $s >= 420$. The expected number of common orderings assuming no preferences of orderings between tasks exist is smaller than $n^2/2^{420}$, which is negligible. Therefore, the number of actually detected common precedences (approximately 30 to 125 for low before high pairs and anywhere from 1 to 17 for the others) seem to be actual request patterns. This is also the case for the other R problems. Indeed, for example, for R1, when $s = 15$, the expected number of common orderings if no preferences of orderings between tasks exist is 7.1, while the number of actually detected precedences is 2815 for low before high and 1222 for the other pairs.

The results of this study suggest that *Genitor* does discover patterns of request interaction, and most of them specify a low altitude request before a high altitude request. These results support the basic idea in Gooley's schedule builder (see Section 6.2): since most request patterns specify low-before-high pairs of request, scheduling low before high and then applying a repair mechanism seems like a good idea. The difficulty is in finding a repair mechanism capable of recognizing the requests to be rescheduled and the low-before-high pairs to be preserved.

---

[2]We wish to thank the anonymous reviewer of an earlier version of the work submitted to the *Journal of Artificial Intelligence Research* for this insightful observation; the rest of the paragraph is based on his/her comments.

Day	Minimizing Conflicts				Minimizing Overlaps			
	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**	996	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	251	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 6.18: Common pairs of request orderings found in permutations corresponding to best known/good *Genitor* solutions for both objective functions.

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 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 6.4, 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 be-

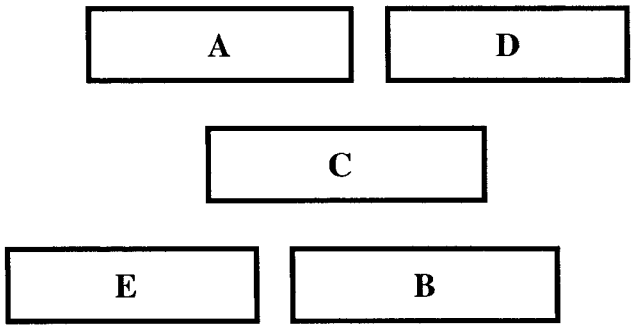


Figure 6.4: A, B, C, D and E compete for two antennas at the same tracking station.

ing 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. It seems that this is a case where the traditional building-block hypothesis [Gol89] holds. *Genitor* seems 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. This results also hint to the possibility of the existence of backdoors [RGS03]. Backdoors are subsets of variables defined for a given algorithm, for which once they are assigned values, the problem becomes easy to solve. As opposed to backbones, multiple sets of backdoor variables can exist. This would explain the fact that not all request patterns need to be encountered in all solutions.

6.4.1.1 Low before High Altitude Request Patterns

Most of the request patterns encountered in good solutions produced by *Genitor* specify low altitude requests before the high altitude requests. By examining the solutions produced by *Genitor*, it is easy to see that they don't follow a "split", where most low

altitude requests would appear before most high altitude requests. However, it might still be the case that there is no interaction between high altitude requests appearing in the solutions before low altitude requests. This would mean in effect that *Genitor* schedules low before high altitude requests. This would also offer strong support for Gooley's schedule builder (see Section 6.2), showing that there exist request orderings for which scheduling all low before high altitude requests results in a best known solution.

To test this, we design an experiment comparing schedules produced by the original schedule builder with the "split" version of it (Section 5.2.1) for permutations derived from *Genitor*. While in the final permutations some high altitude requests do appear before low altitude requests, we hypothesize that most of these high altitude requests do not interact with the low altitude requests appearing after them. If true, the evaluation of high-quality schedules should, on average, remain unchanged when the split heuristic is applied.

To test our hypothesis, we apply the split heuristic to exactly the same permutations used in the previous section to identify common pairs of request orderings. We interpret these permutations in two ways. First, each permutation is transformed into a schedule in the usual way: task are scheduled as they are encountered in the permutation, without regard to low and high-altitude requests. Second, the schedule is built with all of the low-altitude requests scheduled first, followed by the high-altitude requests; both the low and high-altitude requests are still scheduled based on the order in which they appear in the permutation.

Thus, each permutation generates two schedules, the first built normally, the second built such that the low-altitude requests are scheduled first. We calculate how often the two schedules are identical, how often they are different but with the same evaluation and how often they are different with different evaluations.

The results when minimizing conflicts are summarized in Table 6.19. The number

of times the schedules were identical ("Same Evaluation, Same Conflicts") is given in the third column. The number of times the two schedules are different, but have the same evaluation (number of bumps) is given in column "Same Evaluation, Different Conflicts". Finally, the last column shows the number of times when the evaluations are different.

For the A problems, more than 90% of the time, the same evaluation results when the same permutation is either, 1) directly mapped to a schedule (first come, first served, based on the order of the permutation) to obtain a solution, or, 2) all of the low-altitude requests from the permutation are filled first and then all of the high-altitude requests are scheduled. The numbers in the last column of the table also warn that when using the split heuristic only a subspace of the permutations is considered (the permutations that are separated into low and high-altitude requests). This subspace does not contain all the best-known solutions, and, in fact, for different instances of the problem, this subspace could be suboptimal. This suggests that for the A problems *Genitor* is indeed scheduling low-altitude requests with high priority *when appropriate.* The results are very different for the recent days. Only for R5 it seems that the low before high pattern can account for the performance of *Genitor.*

For minimizing the sum of overlaps, the results are presented in Table 6.20. With the exception of A3, A4, A6 and R5, all the solutions found translate into worse schedules when the split heuristic is applied. The results 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 [3] does not account for the performance of *Genitor* for the rest of the problems.

---

[3]The low-before-high pattern is present for the other problems as well, and there is no significant decrease from the number of patterns for A3, A4, A6 and R5, as shown in Table 6.18.

Day	Solution Value	Total Number of Solutions Found	Same Evaluation Same Conflicts	Same Evaluation Different Conflicts	Worse Evaluation
A1	**8**	420	38	373	9
A2	**4**	1000	726	106	168
A3	**3**	996	825	115	56
A4	**2**	937	733	50	154
A5	**4**	862	800	12	50
A6	**6**	967	843	56	68
A7	**6**	1000	588	408	4
R1	43	251	0	59	192
R2	**29**	573	0	0	573
R3	**17**	470	0	0	470
R4	**28**	974	4	484	486
R5	**12**	892	612	189	91

Table 6.19: The effect of applying the split heuristic when evaluating best known schedules produced by *Genitor*.

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

6.4.2 The Effect of the Split Heuristic

We showed in the previous section that most of the common patterns of request ordering in solutions produced by *Genitor* specify a low altitude request scheduled before a high

Day	Genitor-S		
	Min	Mean	Stdev
R1	**42**	42	0
R2	30	30	0
R3	18	18	0
R4	**28**	28	0
R5	**12**	12	0

Table 6.21: Minimizing conflicts: results of running *Genitor* with the split heuristic in 30 experiments, with 8000 evaluations per experiment.

altitude request. Also, for some of the problems in our data sets we showed that *Genitor* is in effect scheduling low altitude before high altitude requests. For the rest of the problems, scheduling low altitude requests first from the solutions found by *Genitor* translates into worse schedules. However, this does not preclude the existence of good solutions in which the low altitude requests are all scheduled before the high altitude requests, or the possibility of *Genitor* finding such solutions. We know that scheduling all the low altitude requests before high altitude requests *may* prevent finding the optimal solutions (see 5.2.1). However, it is not clear that this is the case for all the problems in our data sets. In fact, the split heuristic makes the A problems easy to solve when minimizing the number of conflicts; sampling a relatively small number of solutions produces best known values (see 5.2.1). In this section we investigate the following question: Does the split heuristic prevent *Genitor* to find best known solutions for our test problems?

Table 6.21 shows the results of using the split heuristic with *Genitor* on the R problems, when minimizing the number of conflicts. For the A problems, even random sampling with the split heuristic finds best known solutions (see Section 5.2.1), therefore these problems are not included here. For the R problems, the split heuristic does not always find the best schedules.

The results for minimizing the sum of overlaps are shown in Table 6.22. 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. We hypothesized that for the same problems, A3, A4, A6 and R5, Genitor (without the split heuristic) learns to schedule the low altitude requests before the high altitude ones (see 6.4.1.1); the results in Table 6.22 strengthen this conjecture.

Genitor with the split heuristic does find best known solutions for four of the problems when minimizing the sum of overlaps. It is not surprising that the split heuristic might not work well when minimizing the sum of overlaps. Intuitively, 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.

6.4.3 The Role of Initialization

While *Genitor* normally starts with a population of random permutations, *SWO* is initialized with a greedy solution. As shown in Chapter 5, the greedy solution can be quite good; it results in 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 changes in the performance of *Genitor* when initialized using greedy solutions similar to the starting one for *SWO*. Could *Genitor* too

Day	Genitor-S		
	Min	Mean	Stdev
A1	119	119	0.0
A2	43	43	0.0
A3	**28**	28	0.0
A4	**9**	9	0.0
A5	50	50	0.0
A6	**45**	45	0.0
A7	69	69	0.0
R1	907	924.33	6.01
R2	513	516.63	5.03
R3	276	276.03	0.18
R4	752	752.03	0.0
R5	**146**	146	0.0

Table 6.22: Minimizing sum of overlaps: results of running *Genitor* with the split heuristic in 30 experiments, with 8000 evaluations per experiment.

obtain competitive advantage (performance boost) from initialization?

We seed 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 compare the results of random initializations (Tables 5.6 and 5.8) to those of running *SeededGenitor*. The results are shown in Table 6.23. 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. When minimizing the sum of overlaps, initializing the *Genitor* population results in major improvements of the results 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*.

By initializing *Genitor* with permutations similar to the initial *SWO* solutions, we can directly compare the prioritization mechanism in *SWO* with Syswerda's crossover operator in *Genitor*. Here we are not trying to hybridize *Genitor* with *SWO* to obtain a

Day	Minimizing conflicts			Minimizing overlaps		
	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 6.23: Performance of Genitor when initialized from greedy permutations. All statistics are taken over 30 independent runs, with 8000 evaluations per run.

better performing algorithm. Instead, we are investigating the hypothesis that the greedy initial solutions can easily be modified to obtain best known solutions, in the context of *Genitor*. We are also addressing the question: Does *SWO* still find improvements faster than *SeededGenitor*?

We generate progression graphs similar to the ones in section 5.3. We include *Genitor* in these graphs for comparison. An example for each objective function is presented in Figures 6.5 and 6.6. For both objective functions, the curves are similar, as is relative performance. *SWO* quickly finds a good solution, then its performance levels off. *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. We also observe that for minimizing the number of conflicts, the crossover point between *SeededGenitor* and *SWO* is earlier than for minimizing the overlaps.

We find that initialization helps the performance of *Genitor*: 1) *SeededGenitor* per-

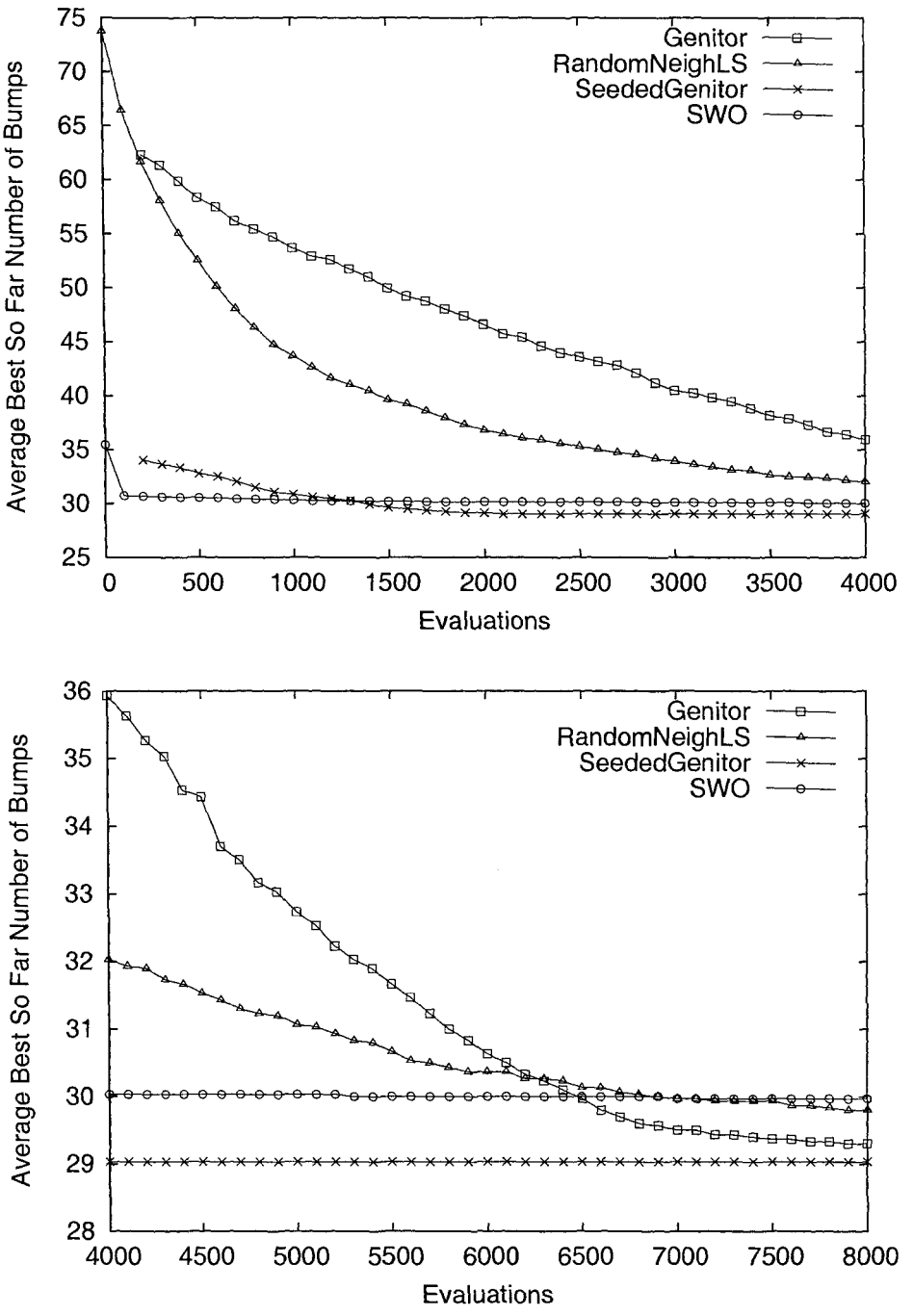


Figure 6.5: Evolutions of the average best value for conflicts obtained by *SWO*, *SeededGenitor*, local search with the randomized neighborhood and *Genitor* during 8000 evaluations, over 30 runs. Note that the scale is different on the y-axis for the last 4000 evaluations (bottom graph).The graphs were obtained for $R2$; best solution value is 29.

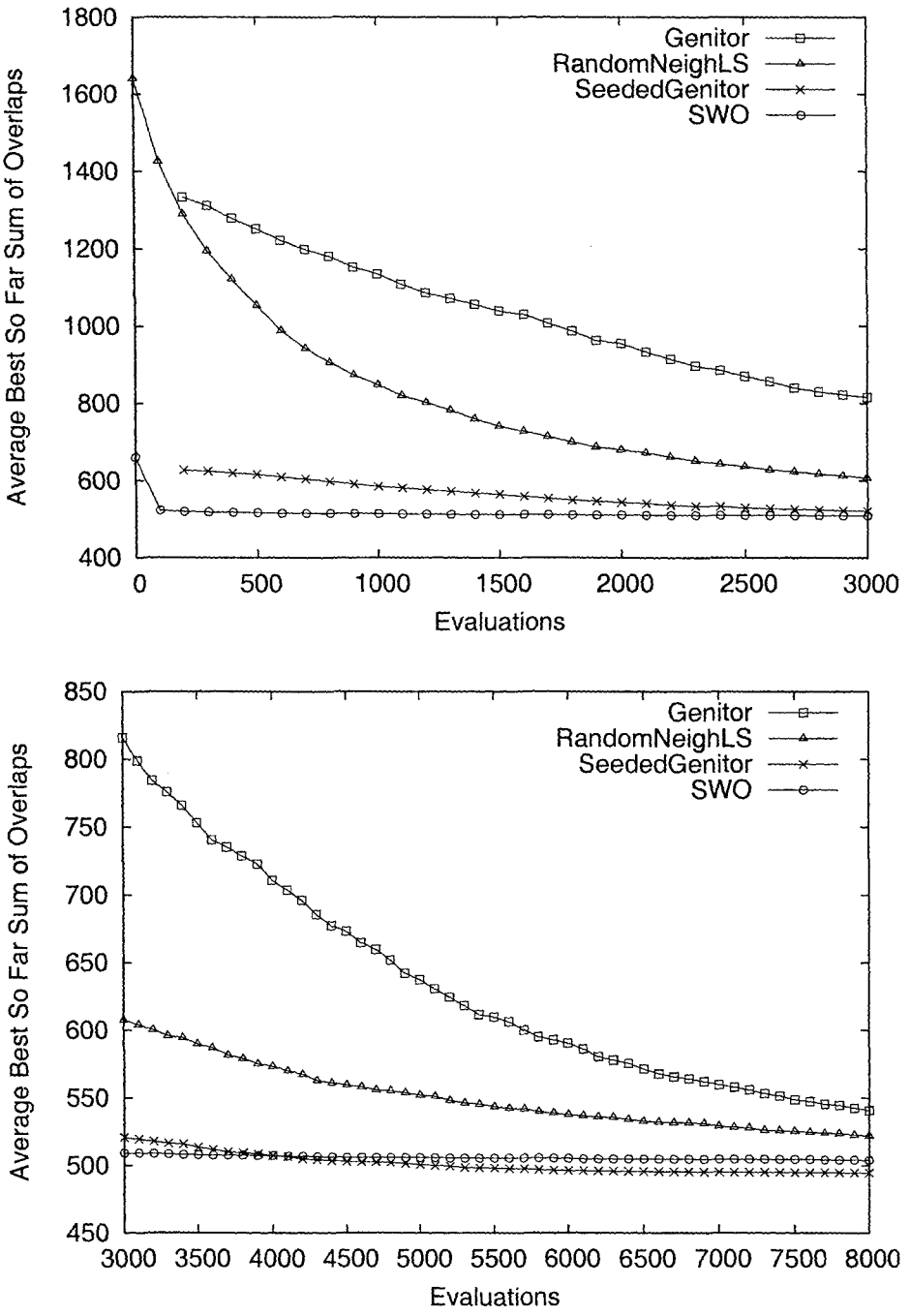


Figure 6.6: Evolutions of the average best value for sum of overlaps obtained by *SWO*, *SeededGenitor*, local search with the randomized neighborhood and *Genitor* during 8000 evaluations, over 30 runs. Note that the scale is different on the y-axis for the last 4000 evaluations (bottom graph).The graphs were obtained for *R2*; best solution value is 486.

forms much better than *Genitor*; in fact, it finds best known solutions for all problems and both objective functions, except for R1 when minimizing overlaps. 2) *SeededGenitor* does not find improvement as fast as *SWO* does in the beginning of the search; however, it outperforms *SWO* within the 8000 evaluations.

Also, we find that the only pattern we could reliably identify in all the solutions specifies a low altitude requests before a high altitude requests. However, a closer look shows that such patterns are not enough to account for *Genitor*'s performance.

6.5 SWO

For both objective functions and all the problems in our set, *SWO* finds the best solutions most of the time. A closer look at *SWO* shows that there are two independent variables that might contribute to its performance: the greedy initial permutation and the prioritization mechanism. While both Genitor and hill-climbing (*RandLS* and *StructLS*) are initialized with random solutions (by default), *SWO* starts with a greedy initial solution. As shown in Chapter 5, the greedy initial solution can be quite good; it is possible that good greedy solutions can be easily modified to find best known solutions. This motivates the question we address in this section: Is *SWO* performance explained by the simple fact that it starts closer to good solutions?

6.5.1 The Effect of the Flexibility Heuristic (The Role of Initialization)

SWO is initialized with a prioritization of the requests based on the flexibility heuristic (see Chapter 4). Given the quality of the solutions produced by simply ordering the requests based on the flexibility measure, maybe it is the case that only a few requests are out of place. Algorithms such as Limited Discrepancy Search (LDS) [HG95] or Heuristic Biased Stochastic Sampling (HBSS) [Bre96] provide mechanisms to follow a heuristic only most of the time (allowing for a number of choices that are different

from the heuristic one). LDS starts by following the heuristic, then it deviates from the heuristic just once, then two times etc. HBSS defines a heuristic bias function to control the degree of following the heuristic. We asked the question: Would such an algorithm using the flexibility heuristic result in good performance? The results in Tables 5.5 and 5.7 indicate that HBSS is outperformed by *SWO* most of the time. It seems that the flexibility heuristic is not by itself capable of generating best known solutions, even when combined with mechanisms like the ones in HBSS, which allow some departure from the heuristic choices.

The progress to the solution graphs for *SWO* show that the most dramatic improvement in the value of the current solution happens during the first 100 evaluations. How much does the initial greedy permutation change during the first 100 evaluations? To answer this question, we analyze two possible measures for the change in the initial permutation. First, we count how many tasks are found in conflict and moved forward during each evaluation. We average over 100 evaluations; the results are shown in Table 6.24.

During each evaluation, the number of tasks moved forward is relatively small. For example, for minimizing conflicts, the average number of tasks moved forward during each evaluation is very close (within two units or less) to the number of bumped tasks in best known solutions for the A problems and $R5$. There is little variance in the number of tasks that get moved during each evaluation. Second, the change in the initial permutation after 100 evaluations can be described by the total number of unique tasks that get moved forward in the first 100 iterations (see Table 6.25). Each time a task is moved forward, other tasks are moved back; for example, if 40 requests are moved forward, there are more than 40 differences (probably a lot more) between the initial and the final permutation solution. These results explain some of the values in Table 6.24. For example, for problem $R5$, during each evaluation exactly 12 tasks are

Day	Minimizing Conflicts		Minimizing Overlaps	
	Mean	Stdev	Mean	Stdev
A1	10.12	1.28	12.24	1.25
A2	4.11	0.58	4.01	0.1
A3	3	0	3.1	0.3
A4	2.67	0.49	3	0
A5	4.65	0.84	4.84	1.16
A6	6	0	6	0
A7	7.42	1.45	9.19	1.45
R1	50.46	2.37	70.78	2.83
R2	34.04	2.2	40.96	2.19
R3	21.79	2.19	29.36	4.14
R4	34.91	3.13	43.4	1.9
R5	12	0	12	0

Table 6.24: Statistics for the number of tasks moved forward by *SWO* during each of the first 100 evaluations over 30 runs.

moved forward; however, not the same tasks are moved: there are approximately 32 different tasks from which 12 are moved during each evaluation (note that 12 is a best known value for this problem).

The results show that during the first 100 evaluations the initial permutation as one would expect does change a lot; however, during each evaluation, the average number of tasks moved forward is close to the number of tasks that do not get scheduled in the best solution found by *SWO*.

The initial greedy permutation changes a lot even during the first 100 evaluations. Is it then the case that *SWO* would perform just as well if initialized from a random permutation? To answer this question, we replace the initial greedy permutation (and its variations in subsequent iterations of *SWO*) with random permutations and then use the *SWO* mechanism to iteratively move forward the requests in conflict. We call this version of *SWO RandomStartSWO*. We compare the results produced by *RandomStartSWO* with results from *SWO* to assess the effects of the initial greedy solution. We also compare the *RandomStartSWO* results with the ones from local search and *Genitor* (since

Day	Minimizing Conflicts		Minimizing Overlaps	
	Mean	Stdev	Mean	Stdev
A1	40.9	3.08	41.3	2.65
A2	14.57	1.45	18.57	2.6
A3	9	1.7	9.67	0.96
A4	8.33	1.89	10.33	1.99
A5	16.3	1.8	17	2.05
A6	18.1	1.09	19.63	0.49
A7	30.6	2.04	30.83	2.17
R1	184.8	6.05	175.5	6.27
R2	99.33	3.37	105.03	4.28
R3	95.53	6.21	89.2	6.08
R4	103.33	3.08	107.37	3.05
R5	32.1	1.29	33.43	1.3

Table 6.25: Statistics for the total number of unique tasks moved forward by *SWO* during the first 100 evaluations over 30 runs.

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[4]. The results produced by *RandomStartSWO* are presented in Table 6.26. 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; 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.

[4]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.

The numbers in Table 6.26 show that *RandomStartSWO* is capable of finding good solutions (best known solutions most of the time, with the exception of the current days when minimizing overlaps). These results offer limited support to our initial hypothesis about *SWO* performance factors: the greedy initial permutation is only partially responsible for *SWO* performance.

To further verify this, we also assess whether *RandomStartSWO* progresses at the same fast rate as *SWO* does (we showed in Chapter 5, Figures 5.11, 5.10, 5.13, 5.12 that *SWO* finds good solutions fast, then its performance levels off). The graphs in Figures 6.7, 6.8, 6.9 and 6.10 show algorithm progress to the solution. *RandLS*, *StructLS* and *RandomStartSWO* are initialized with identical random solutions. *RandomStartSWO* quickly descends to good solution values, then further improvements are very small. The improvement over the initial solutions in the beginning of the search is very dramatic; afterward, the progress through the search space is very similar to what we observed for *SWO*. The graphs also show that on average, *RandomStartSWO* levels off at a worse value than *SWO* does. In the beginning, the *RandomStartSWO* finds good solutions much faster than *RandLS*. However, in the second half of the search, *RandLS* keeps finding better solutions until it outperforms *RandomStartSWO* (in terms of average best so far). The most noticeable feature shown in the graphs is that *RandomStartSWO* progresses very fast to good solutions; the prioritization mechanism is very effective in moving quickly from random to good solutions.

6.6 Multiple Moves Hypothesis

Starting close to good solutions does help the performance of both *Genitor* and hill-climbing; this supports the hypothesis that it might be easier to modify greedy solutions to find the best known solutions. However, starting *SWO* from random solutions does not seem to impact its performance as much as one would expect if the greedy initial

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

Table 6.26: 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.

solutions were essential to guide the search in the right direction. Also, there is only limited evidence for the existence of building blocks in *Genitor* solutions. This suggests that, while there is some support for our hypotheses for algorithm factors for *Genitor* and *SWO*, these hypotheses are not enough to explain the observed performance. On the other hand, our analyses of hill-climbing identified large plateaus in the search space; in light of this, we formulate another hypothesis: *SWO* and *Genitor* make long leaps in space, which allow them to relatively quickly traverse the plateaus.

SWO is moving forward multiple requests that are known to be problematic. The position crossover mechanism in *Genitor* can be viewed as applying *multiple* consecutive shifts to the first parent, such that the requests in the selected positions of the second parent are moved into those selected position. In a sense, each time the crossover operator is applied, a multiple move is proposed for the first parent. We hypothesize that this multiple move mechanism present both for *SWO* and *Genitor* allows them to make long leaps in the space and thus reach the solutions fast.

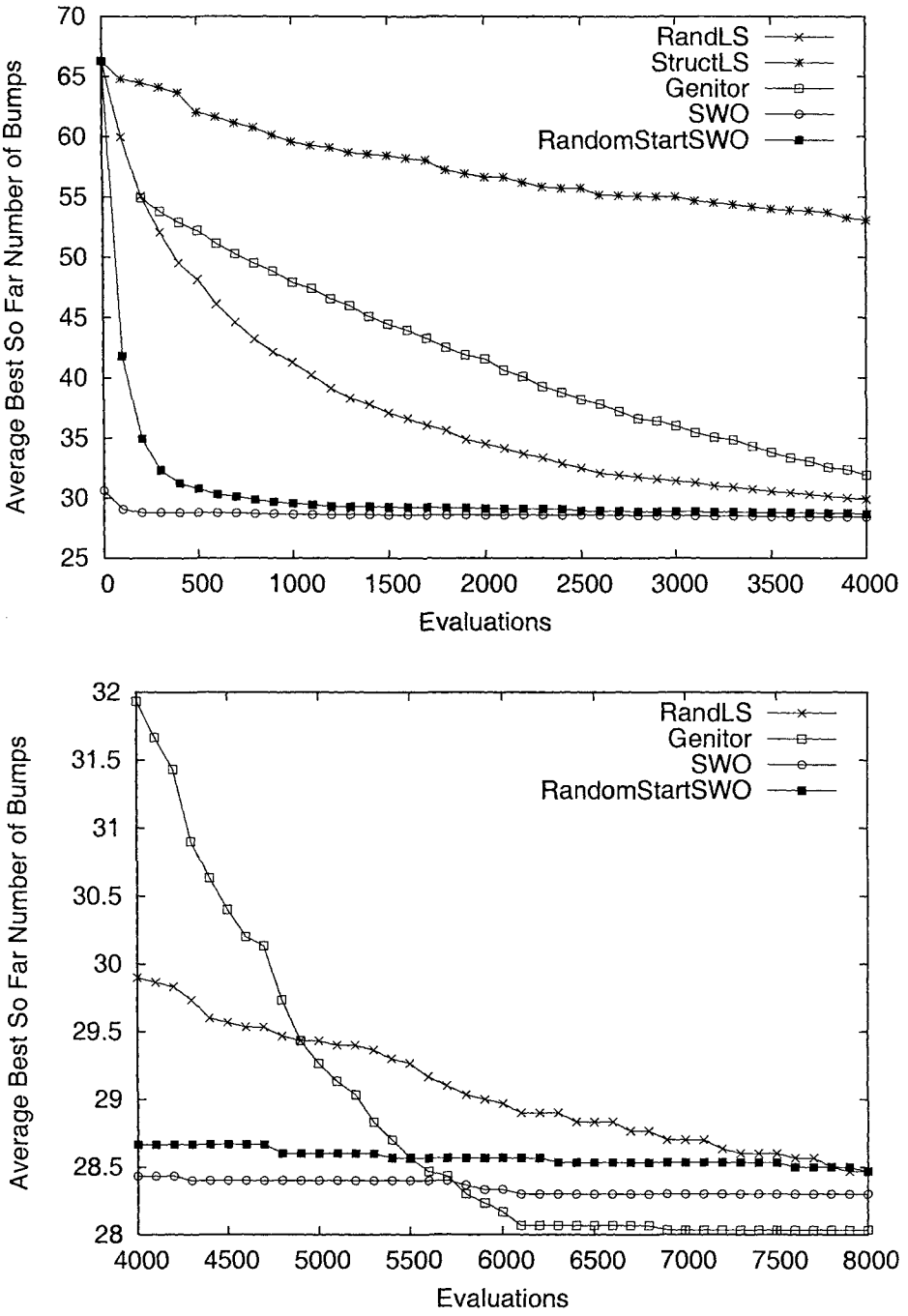


Figure 6.7: Evolutions of the average best value obtained by *RandLS*, *StructLS*, Genitor, *SWO* and *RandomStartSWO* during 8000 evaluations, over 30 runs. Note that the scale is different on the y-axis in the two graphs; also, *StructLS* is not shown in the bottom graph. The graphs were obtained for $R4$; best solution value is 28.

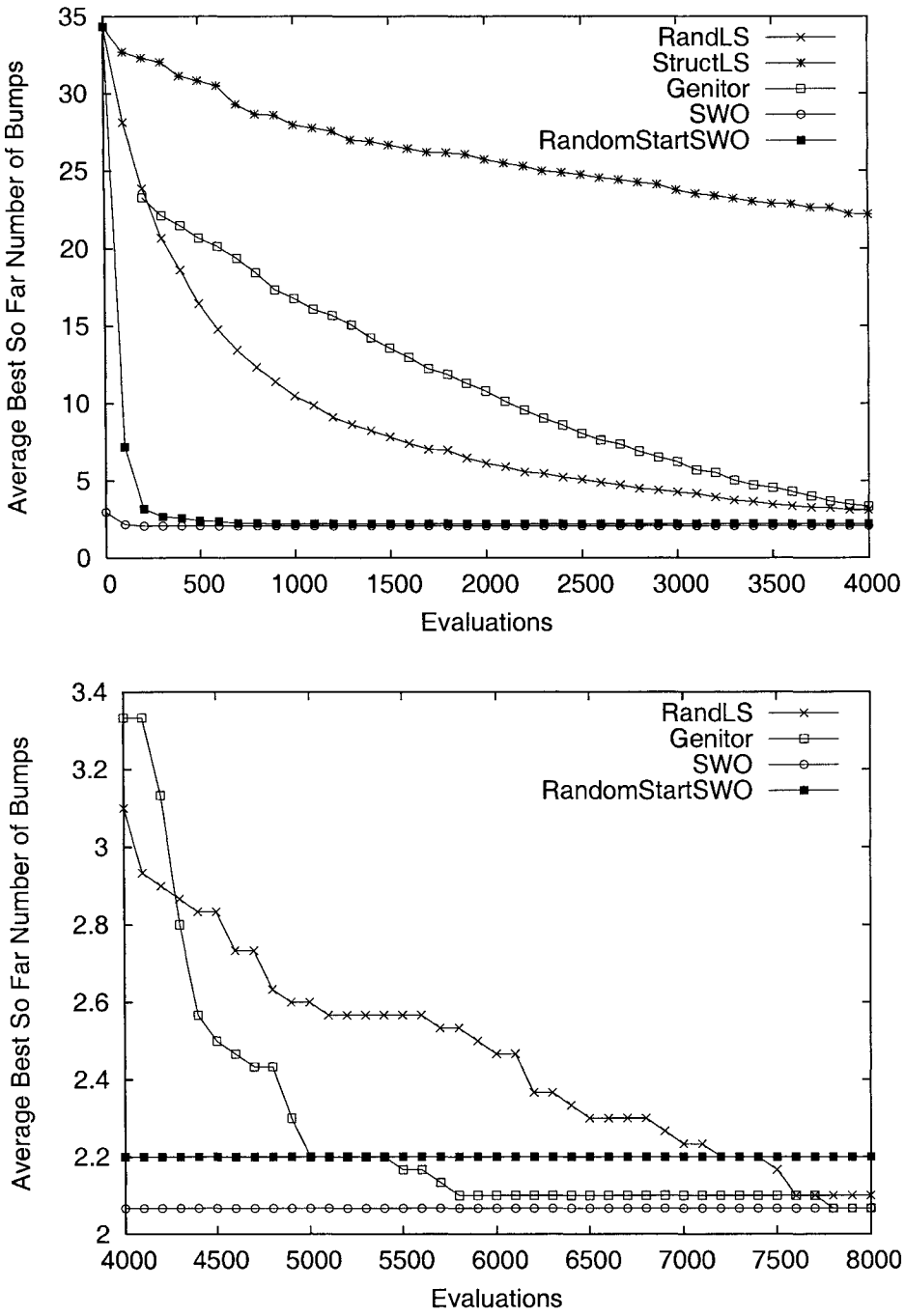


Figure 6.8: Evolutions of the average best value obtained by *RandLS*, *StructLS*, Genitor, *SWO* and *RandomStartSWO* during 8000 evaluations, over 30 runs. Note that the scale is different on the y-axis in the two graphs; also, *StructLS* is not shown in the bottom graph. The graphs were obtained for $A4$; best solution value is 2.

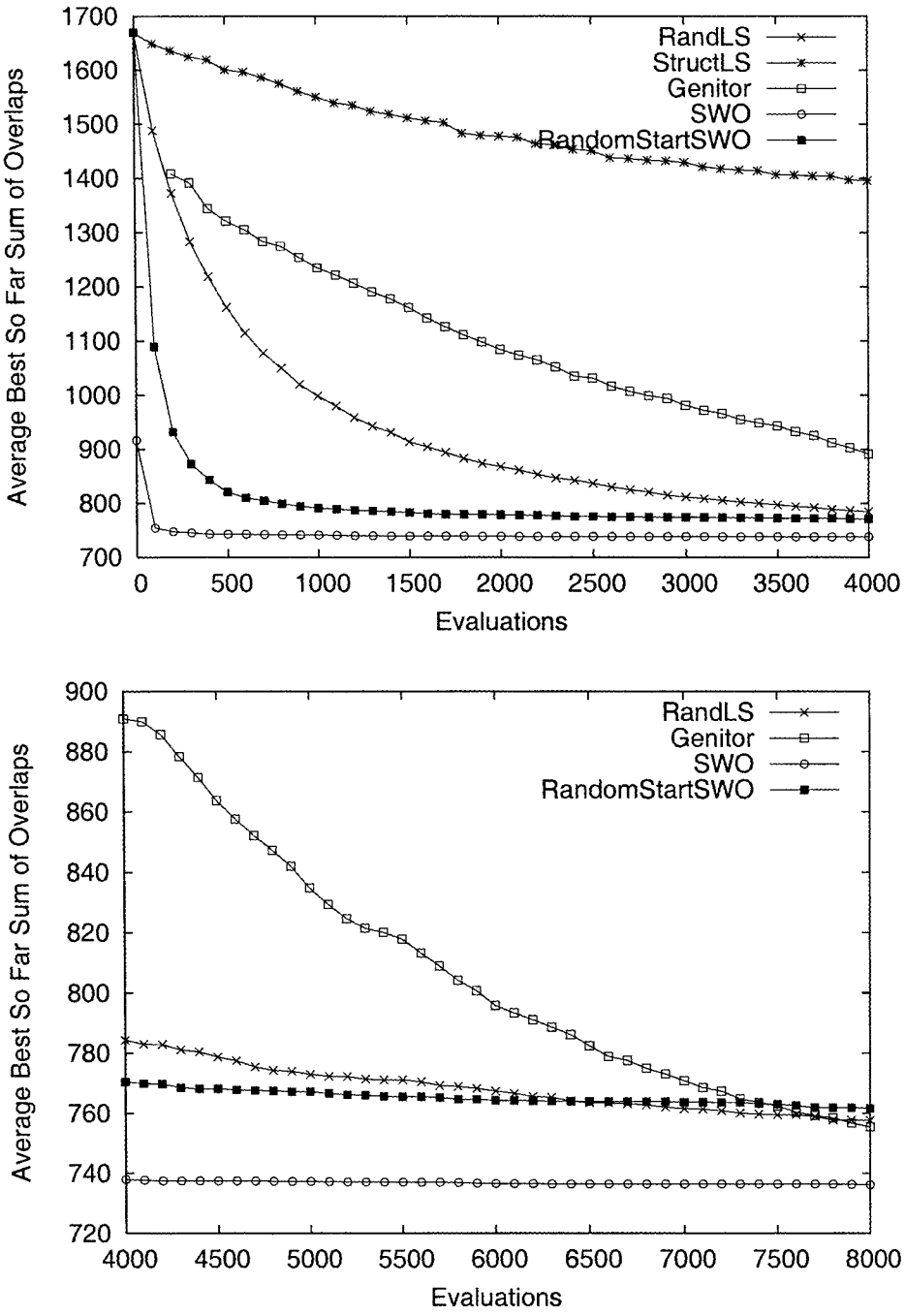


Figure 6.9: Evolutions of the average best value obtained by *RandLS*, *StructLS*, Genitor, *SWO* and *RandomStartSWO* during 8000 evaluations, over 30 runs. Note that the scale is different on the y-axis in the two graphs; also, *StructLS* is not shown in the bottom graph. The graphs were obtained for $R4$; best solution value is 725.

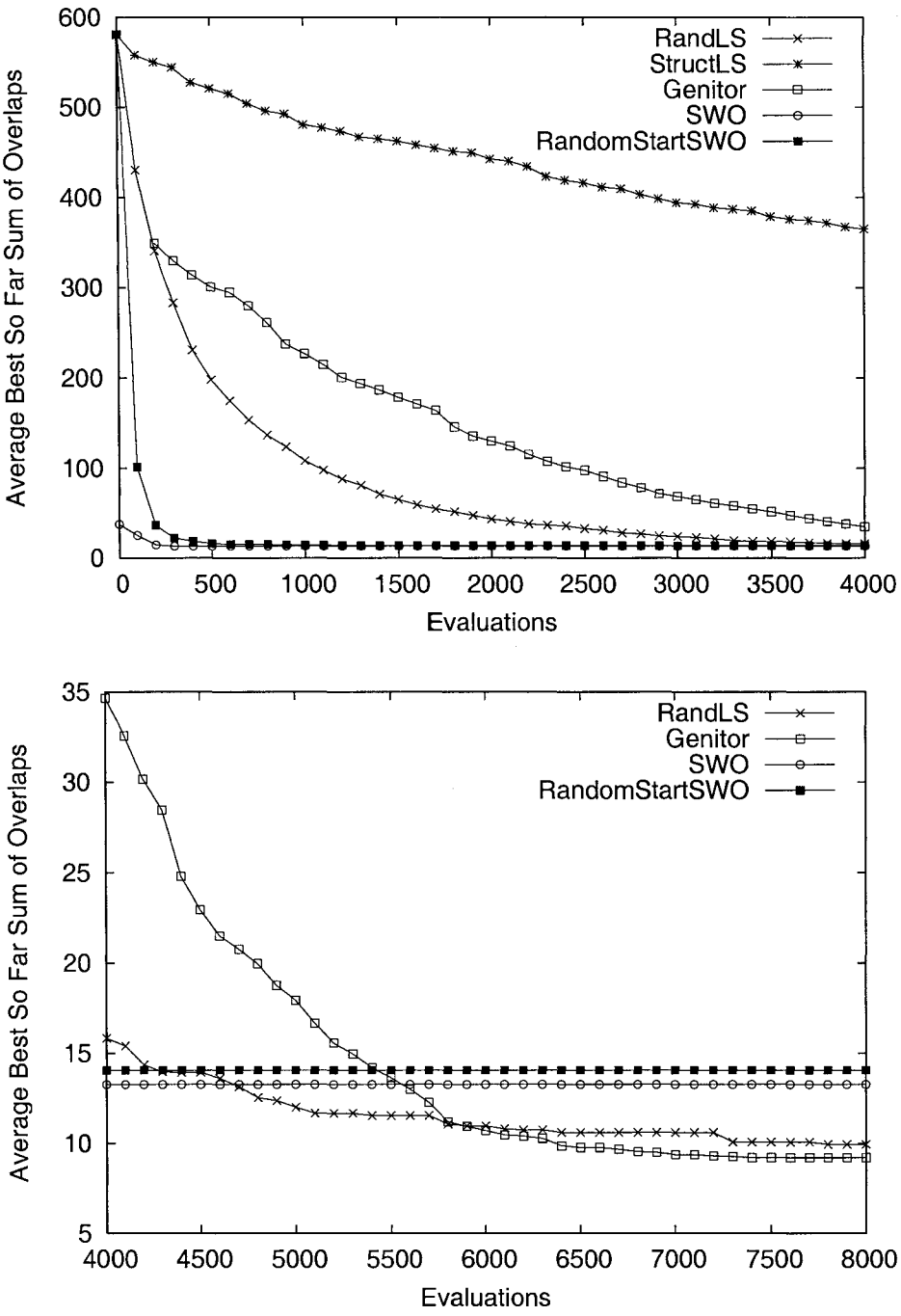


Figure 6.10: Evolutions of the average best value obtained by *RandLS*, *StructLS*, Genitor, *SWO* and *RandomStartSWO* during 8000 evaluations, over 30 runs. Note that the scale is different on the y-axis in the two graphs; also, *StructLS* is not shown in the bottom graph. The graphs were obtained for $A4$; best solution value is 9.

To test the contribution of multiple moves when traversing the search space, we perform experiments with a variable number of moves at each step for both *Genitor* and *SWO*. For *Genitor*, we vary the number of crossover positions allowed. For *SWO*, we vary the number of requests in conflict moved forward.

6.6.1 The Effect of Multiple Moves on *Genitor*

To test the effect of multiple moves on *Genitor*, we change Syswerda's position crossover by imposing a fixed number of selected positions in the second parent (see Section 4.4.3 for a description of Syswerda's position crossover). We call this implementation *Genitor-k* where k is the number of selected positions. Recall that our implementation of Syswerda's position crossover randomly selects a number of positions that is larger than one third and smaller than two thirds of the total number of positions. If multiple moves are indeed a factor in performance then increasing the number of selected positions up to a point should result in finding improvements faster. If only a few positions are selected, the offspring will be very similar to the first parent. If the number of selected positions is large, close to the number of total requests, the offspring will be very similar to the second parent. If the offspring is very similar to one of its two parents, we expect a slower rate in finding improvements to the current best solution. Therefore, both for small and for large k values, we expect *Genitor-k* to find improvements at a much slower rate than *Genitor* or *Genitor-k* with average k values (values closer to half of the number of requests).

For our study, we run *Genitor-k* with k=10, 50, 100, 150, 200, 250, 300 and 350. We allowed 8000 evaluations per run and performed 30 runs for each problem. The results are summarized in Tables 6.27 and 6.28 for minimizing the number of conflicts and in Tables 6.29 and 6.30 for minimizing the sum of overlaps. Note that for A6 and A7 there are 299 and 297 requests to schedule respectively. Therefore $k = 300$ and $k = 350$ cannot be run for these two problems. Also note that for example, $k = 200$ does not

mean that there are 200 differences in the selected positions between the two parents. The offspring is likely to be very similar to its parents, regardless of the value of k, when the parents are similar.

When minimizing the number of conflicts, the worst results are produced by $k = 10$. For $k = 50$, the results improve, best knowns are found for all the A problems; however, for R1, R2, and R3, the best knowns are not found. Starting with $k = 100$ up to $k = 250$ *Genitor-k* finds best known solutions for all the problems. The means and standard deviations are also very similar for all these k values; the smallest means and standard deviations correspond to $k = 200$ for the A problems and $k = 250$ for the R problems (with the exception of R3 for which $k = 200$ produces better results). For $k = 300$, the best knowns are not found anymore for the A problems; 300 is very close to the size of the five A problems for which it is feasible to run *Genitor-300.* The decay in performance is not as significant for the R problems: there is an increase in the means and standard deviations for $k = 300$ and $k = 350$; however, best knowns are still found for four out of the five problems. Note that $k = 400$ would have been a lot closer to the total number of requests for the R problems; we believe the performance would have degraded more for the R problems for larger k values. When minimizing overlaps, we observe trends that are very similar to the ones for minimizing the number of conflicts. $k = 10$ produces poor results, followed by $k = 50$. Similar results are produced by $k = 100, 150, 200, 250$. $k = 150$ results in the smallest means and standard deviations for the A problems, while $k = 200$ and $k = 250$ produce best results for the R problems. For $k = 300$ and $k = 350$, similarly to minimizing the number of conflicts, the means and standard deviations increase and so do the best solutions found; best knowns are only found for R5.

In terms of the evolution to the solution, we observe very similar trends for the two objective functions. We present two typical examples for minimizing bumps for the A

Day	*Genitor-10*			*Genitor-50*			*Genitor-100*			*Genitor-150*		
	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev
A1	11	1.93	1.94	**8**	9.26	0.63	**8**	8.66	0.47	**8**	8.53	0.5
A2	5	7.13	1.77	**4**	4.03	0.18	**4**	4.0	0.0	**4**	4.0	0.0
A3	6	10.4	2.12	**3**	3.36	0.55	**3**	3.0	0.0	**3**	3.0	0.0
A4	5	10.66	2.7	**2**	3.13	0.81	**2**	2.23	0.50	**2**	2.06	0.25
A5	5	9.6	2.29	**4**	4.73	0.69	**4**	4.26	0.44	**4**	4.2	0.4
A6	9	12.63	1.8	**6**	6.83	0.94	**6**	6.03	0.18	**6**	6.06	0.25
A7	8	10.6	1.75	**6**	6.1	0.30	**6**	6.0	0.0	**6**	6.0	0.0
R1	57	66.5	4.38	47	52.0	2.82	**42**	45.83	1.68	**42**	44.36	1.24
R2	42	47.16	3.59	32	34.53	1.47	**29**	30.0	0.78	**29**	29.6	0.56
R3	27	31.1	2.41	19	21.6	1.67	**17**	18.03	0.61	**17**	17.63	0.61
R4	36	41.9	2.74	**28**	30.96	2.04	**28**	28.33	0.47	**28**	28.1	0.4
R5	13	20.73	2.53	**12**	13.23	0.81	**12**	12.46	0.62	**12**	12.2	0.4

Table 6.27: Performance of *Genitor-k*, where k represents the fixed number of selected positions for Syswerda's position crossover, in terms of the best and mean number of conflicts. Statistics are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

problems in Figures 6.11, 6.12 for minimizing the number of conflicts, and Figures 6.15 and 6.16 for minimizing overlaps. $k = 10$ is slower in finding improvements than $k = 50$ and $k = 50$ is slower than $k = 100$. $k = 100$ up to $k = 250$ result in evolutions that are similar to the original *Genitor*'s evolution (not all of them appear on the graph, for readability reasons). $k = 300$ either is the worst performer or the second worst (between $k = 10$ and $k = 50$). For the R problems, two typical examples are presented in Figures 6.13 and 6.14 for minimizing the number of conflicts and Figures 6.17 and 6.18 for minimizing overlaps. Again, $k = 10$ is slower than $k = 50$ which is slower than $k = 100$. $k = 150$ up to $k = 250$ are performing similarly and also similar to the original *Genitor* implementation. $k = 300$ is still moving through the space at a rate that's similar to *Genitor*'s. Only for $k = 350$ does the performance start to decay.

The original implementation of the crossover operator (with a variable number of selected position) was shown to work well not only for our domain but also for other scheduling applications, such as [Sys91, WRWH99, SP91]. For our test problems, the results in this subsection show that the number of crossover positions influences the per-

Day	*Genitor-200*			*Genitor-250*			*Genitor-300*			*Genitor-350*		
	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev
A1	**8**	8.56	0.56	**8**	8.9	0.3	9	11.8	1.66	-	-	-
A2	**4**	4.0	0.0	**4**	4.03	0.18	9	13.66	1.76	-	-	-
A3	**3**	3.0	0.0	**3**	3.06	0.25	4	9.2	2.1	-	-	-
A4	**2**	2.0	0.0	**2**	3.13	0.81	4	8.56	1.94	-	-	-
A5	**4**	4.3	0.46	**4**	4.73	0.58	10	13.86	2.14	-	-	-
A6	**6**	6.06	0.25	**6**	6.5	0.57	-	-	-	-	-	-
A7	**6**	6.0	0.0	**6**	6.06	0.25	-	-	-	-	-	-
R1	**42**	44.03	1.15	**42**	44.03	0.85	43	44.26	1.01	43	45.46	1.22
R2	**29**	29.36	0.49	**29**	29.4	0.49	**29**	29.7	0.59	**29**	30.13	0.86
R3	**17**	17.33	0.4	**17**	17.7	0.65	**17**	17.73	0.58	**17**	18.63	0.8
R4	**28**	28.03	0.18	**28**	28	0.0	**28**	28.03	0.18	**28**	28.63	0.71
R5	**12**	12.1	0.3	**12**	12.06	0.25	**12**	12.16	0.37	**12**	12.6	0.81

Table 6.28: Performance of *Genitor-k*, where k represents the fixed number of selected positions for Syswerda's position crossover, in terms of the best and mean number of conflicts. Statistics are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values. The dashes indicate that the permutation solutions for A6 and A7 are shorter than 300 (299 and 297, respectively), and therefore can not select 300 positions in these permutations.

Day	*Genitor-10*			*Genitor-50*			*Genitor-100*			*Genitor-150*		
	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev
A1	149	221.53	38.85	107	115.76	11.53	107	107.2	0.76	107	107.1	0.54
A2	30	69.66	29.22	**13**	15.73	3.86	**13**	13.43	1.54	**13**	13.03	0.18
A3	51	122.86	36.12	**28**	36.26	8.19	**28**	28.9	1.72	**28**	28.16	0.64
A4	59	124.5	42.25	**9**	19.36	9.3	**9**	9.23	0.72	**9**	9.06	0.36
A5	43	90.7	32.01	**30**	33.06	3.62	**30**	30.36	0.96	**30**	30.43	0.5
A6	94	145.06	33.12	**45**	49.6	5.54	**45**	45.36	0.8	**45**	45.16	0.46
A7	67	115.66	27.96	**46**	51.7	7.89	**46**	46.5	2.23	**46**	47.63	3.9
R1	1321	1531.13	107.35	987	1139.5	76.57	914	991.13	38.19	915	963.96	26.89
R2	743	961.13	81.62	557	643.86	50.0	515	549.1	18.8	516	540.86	15.82
R3	480	652.5	90.37	319	391.56	47.31	268	305.3	20.63	269	291.3	13.36
R4	866	1069.23	74.65	768	840.23	38.79	735	757.43	15.95	731	752.7	14.07
R5	208	309.03	46.3	**146**	172.13	18.18	**146**	151.53	7.63	**146**	148.23	5.26

Table 6.29: Performance of *Genitor-k*, where k represents the fixed number of selected positions for Syswerda's position crossover, in terms of the best and mean number of conflicts. Statistics are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

formance of *Genitor*, both in terms of best solutions found and in terms of the rate of finding improvements. For a small number of crossover positions (10 or 50), the solutions found are not competitive, and the improvements are found at a slower rate than

Day	*Genitor-200*			*Genitor-250*			*Genitor-300*			*Genitor-350*		
	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev
A1	107	107.1	0.74	107	108.03	2.22	113	157.66	26.21	-	-	-
A2	**13**	13.2	0.92	**13**	17.0	5.8	116	185.56	33.27	-	-	-
A3	**28**	28.9	1.6	**28**	31.63	4.47	63	106.23	26.21	-	-	-
A4	**9**	9.1	0.4	**9**	10.36	3.41	37	78.06	25.78	-	-	-
A5	**30**	30.6	1.3	**30**	31.56	2.22	76	160.0	36.59	-	-	-
A6	**45**	46.33	2.7	**45**	50.96	8.82	-	-	-	-	-	-
A7	**46**	47.63	4.2	**46**	49.93	5.39	-	-	-	-	-	-
R1	878	970.1	38.38	914	968.63	31.59	935	986.7	37.9	927	1008.8	42.17
R2	512	538.43	13.94	511	538.93	12.88	526	551.63	12.27	532	559.46	19.7
R3	268	287.96	11.05	270	292.23	12.85	272	299.43	16.3	299	332.46	20.14
R4	730	752.1	12.25	734	754.53	11.89	745	764	13.36	743	785.36	26.63
R5	**146**	147.633	2.95	**146**	147.96	3.7	**146**	148.6	3.84	**146**	157	10.6

Table 6.30: Performance of *Genitor-k*, where k represents the fixed number of selected positions for Syswerda's position crossover, in terms of the best and mean number of conflicts. Statistics are taken over 30 independent runs, with 8000 evaluations per run. Min numbers in boldface indicate best known values.

in the original *Genitor* implementation. Similarity to *Genitor*'s original performance is obtained for k values between 100 and 250. Higher k values result in a decay in performance. These results also offer an empirical motivation for the choice of the number of crossover positions in the original *Genitor* implementation. Indeed, in the original implementation, the crossover uses a number of positions randomly selected between one third and two thirds of the total number of requests. This translates for the sizes of problems in our sets to a number of positions that is approximately between 100 and 300. Another interesting observation based on the results presented in this subsection is that, for our test problems, *Genitor* could perform just as well if the number of crossover positions was fixed (for example, to 150 for the A problems and 200 for the R problems) as opposed to randomly selected, so long as enough differences exist between the members of the initial population.

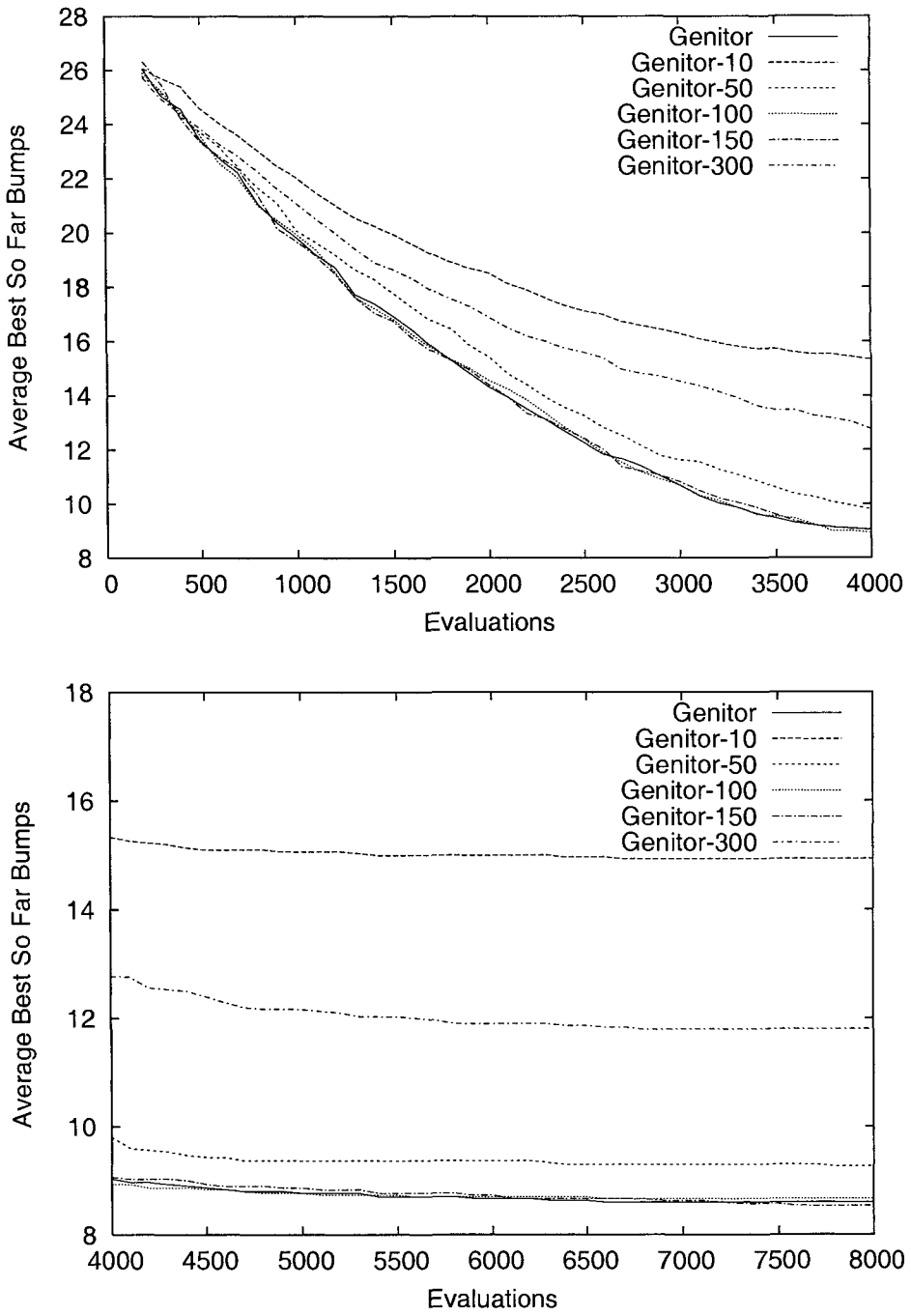


Figure 6.11: Evolutions of the average best value obtained by *Genitor* and its versions with a fixed number of selected positions for crossover. during 8000 evaluations, over 30 runs. The graphs were obtained for $A1$; best solution value is 8.

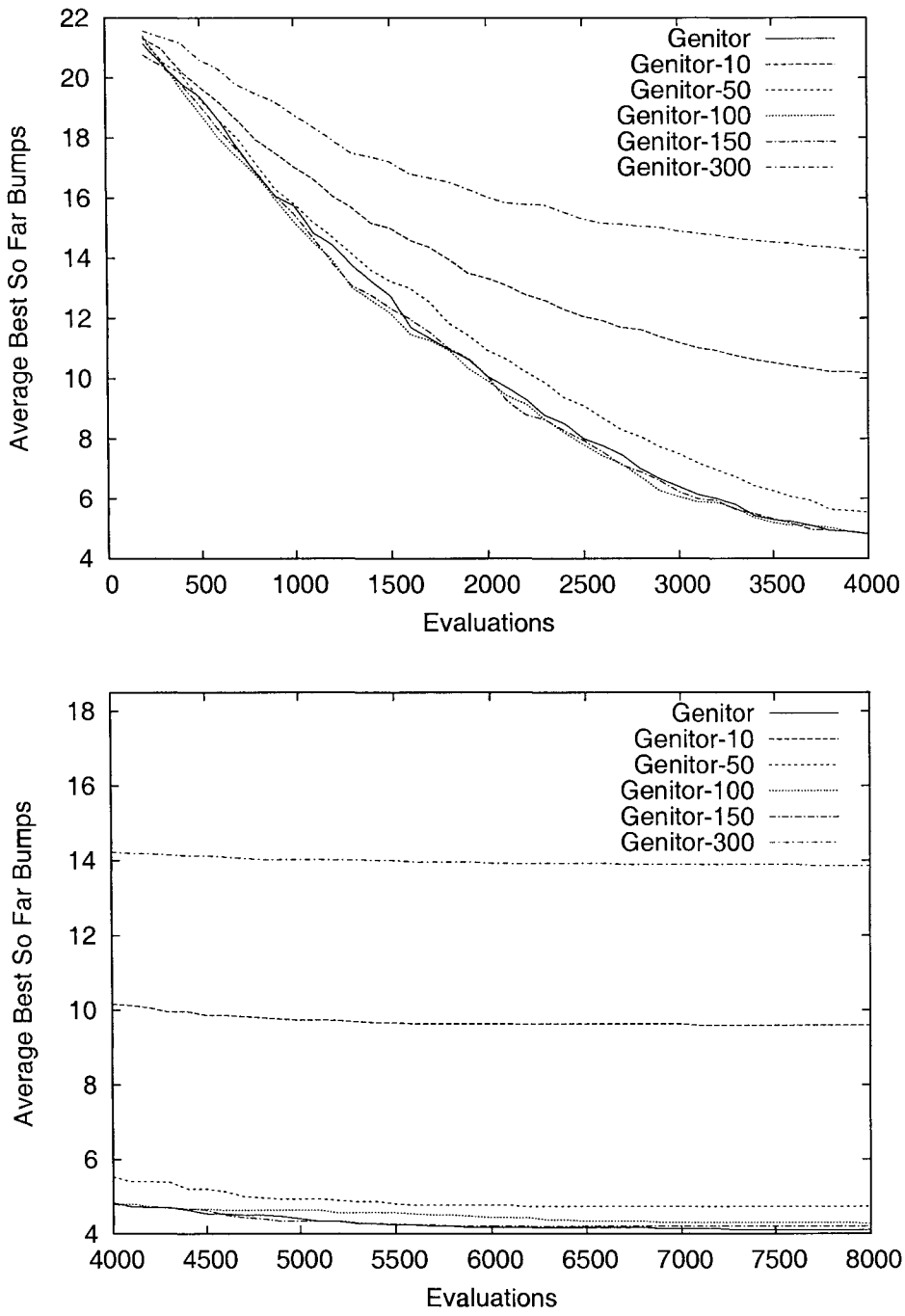


Figure 6.12: Evolutions of the average best value obtained by *Genitor* and its versions with a fixed number of selected positions for crossover. during 8000 evaluations, over 30 runs. The graphs were obtained for $A5$; best solution value is 4.

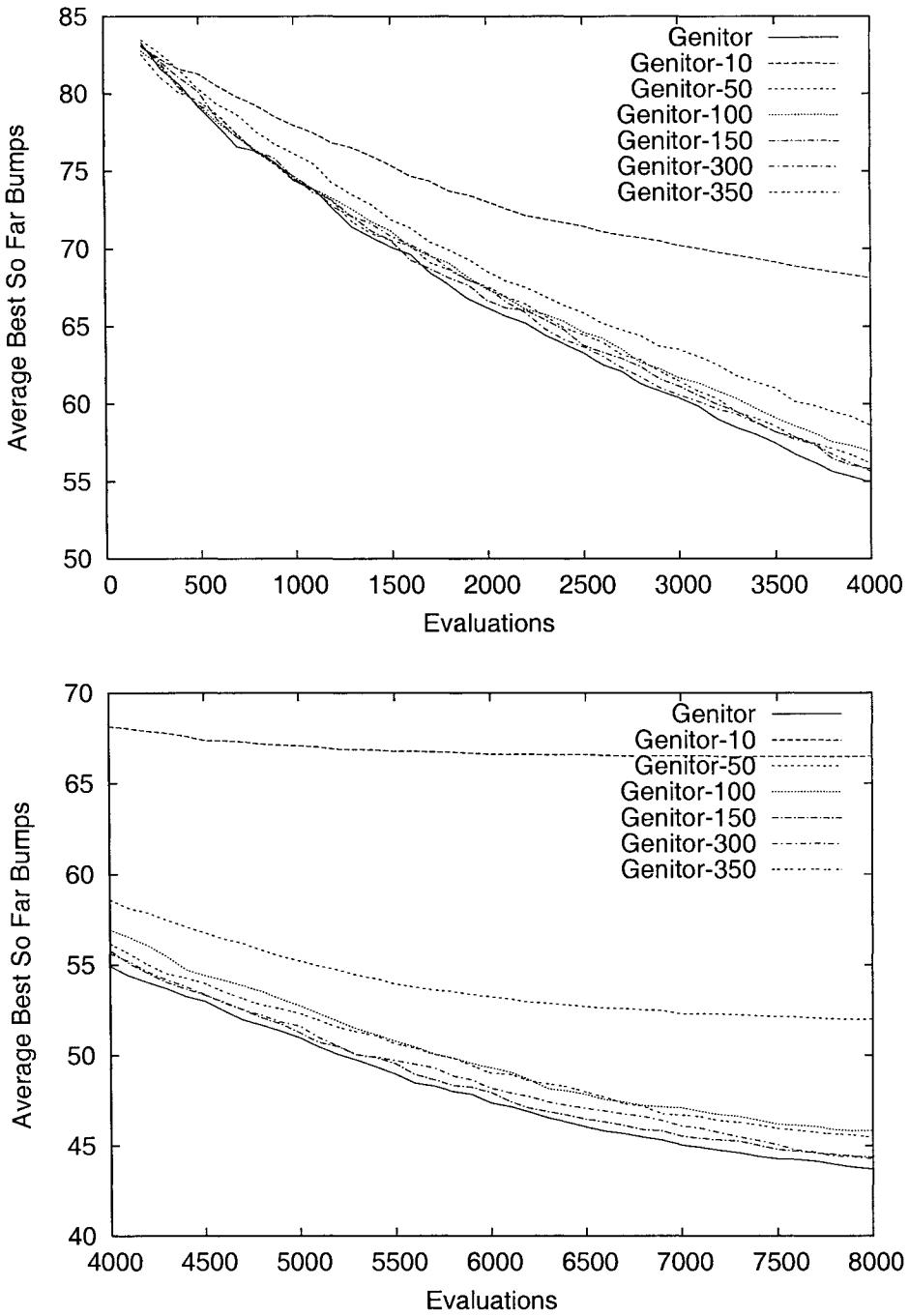


Figure 6.13: Evolutions of the average best value obtained by *Genitor* and its versions with a fixed number of selected positions for crossover. during 8000 evaluations, over 30 runs. The graphs were obtained for $R1$; best solution value is 42.

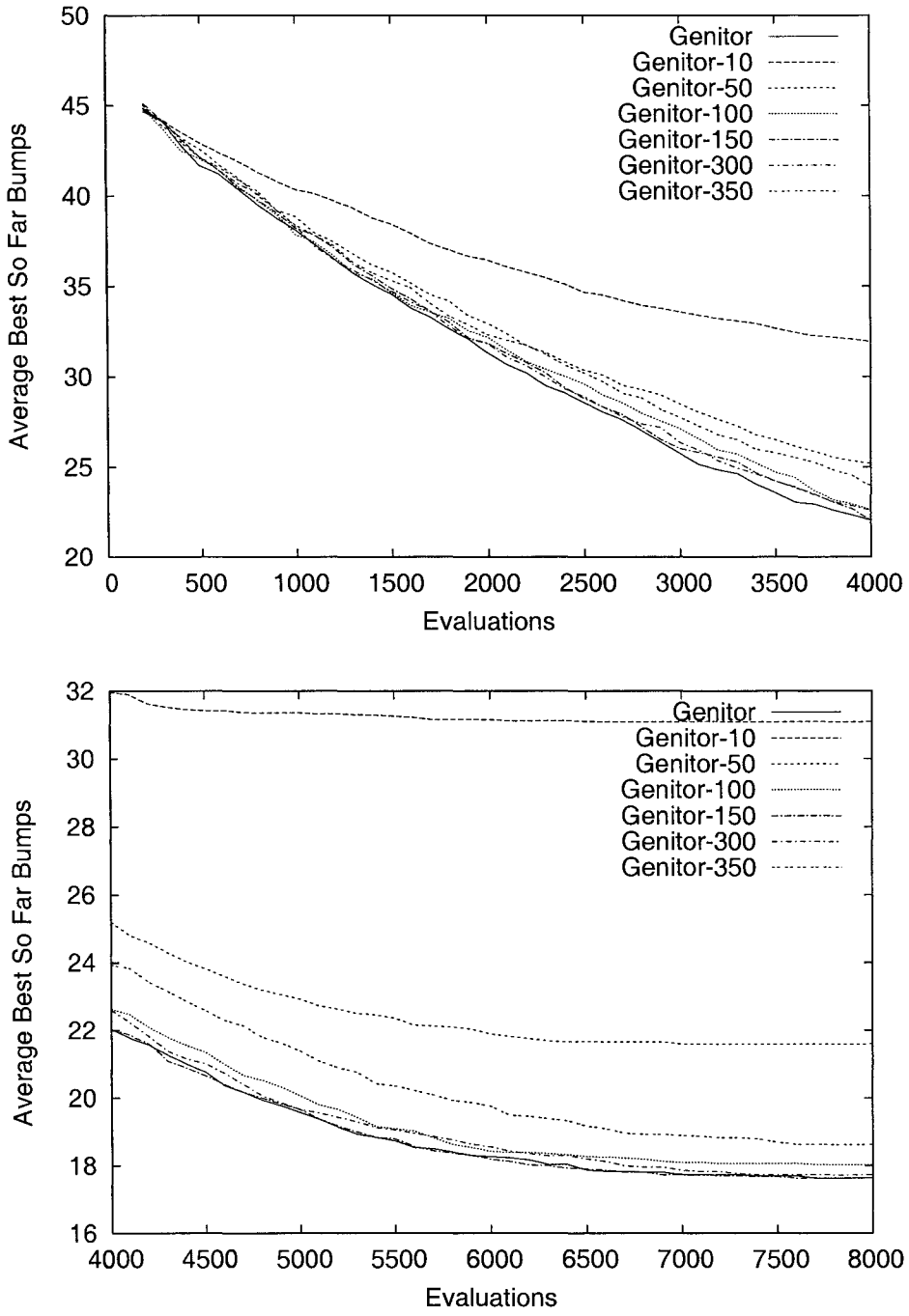


Figure 6.14: Evolutions of the average best value obtained by *Genitor* and its versions with a fixed number of selected positions for crossover. during 8000 evaluations, over 30 runs. The graphs were obtained for $R3$; best solution value is 17.

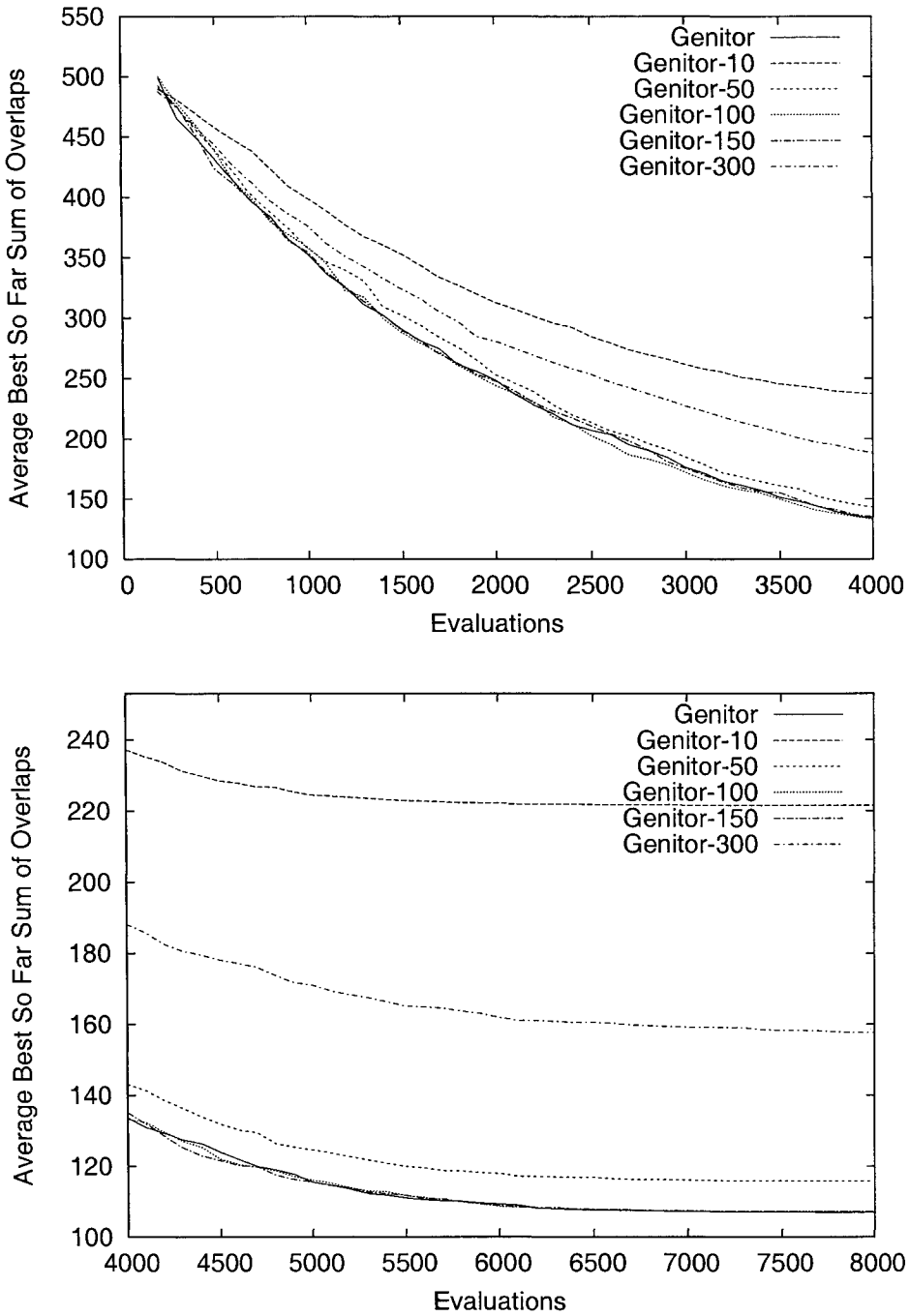


Figure 6.15: Evolutions of the average best value obtained by *Genitor* and its versions with a fixed number of selected positions for crossover. during 8000 evaluations, over 30 runs. The graphs were obtained for $A1$; best solution value is 104.

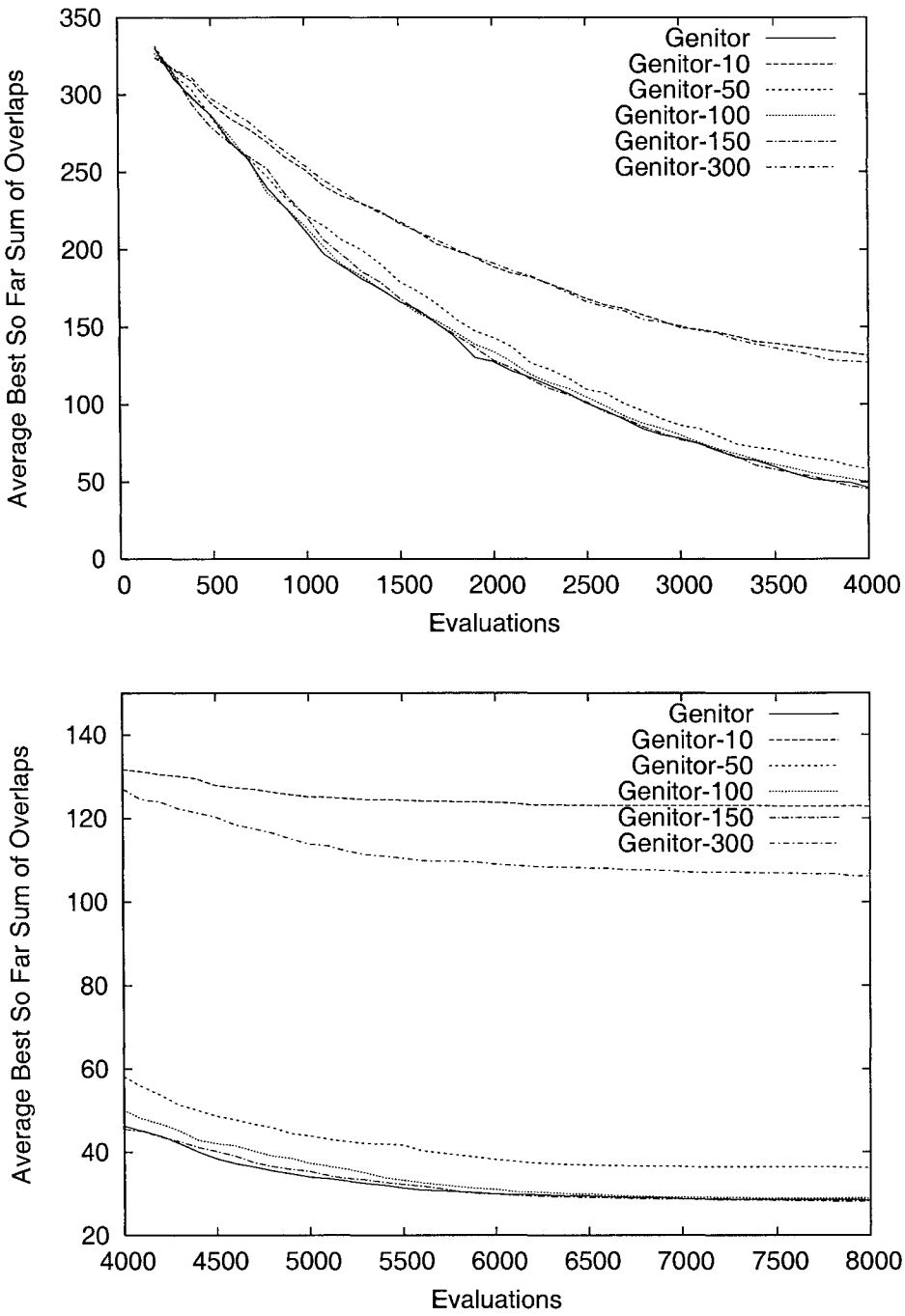


Figure 6.16: Evolutions of the average best value obtained by *Genitor* and its versions with a fixed number of selected positions for crossover. during 8000 evaluations, over 30 runs. The graphs were obtained for $A3$; best solution value is 28.

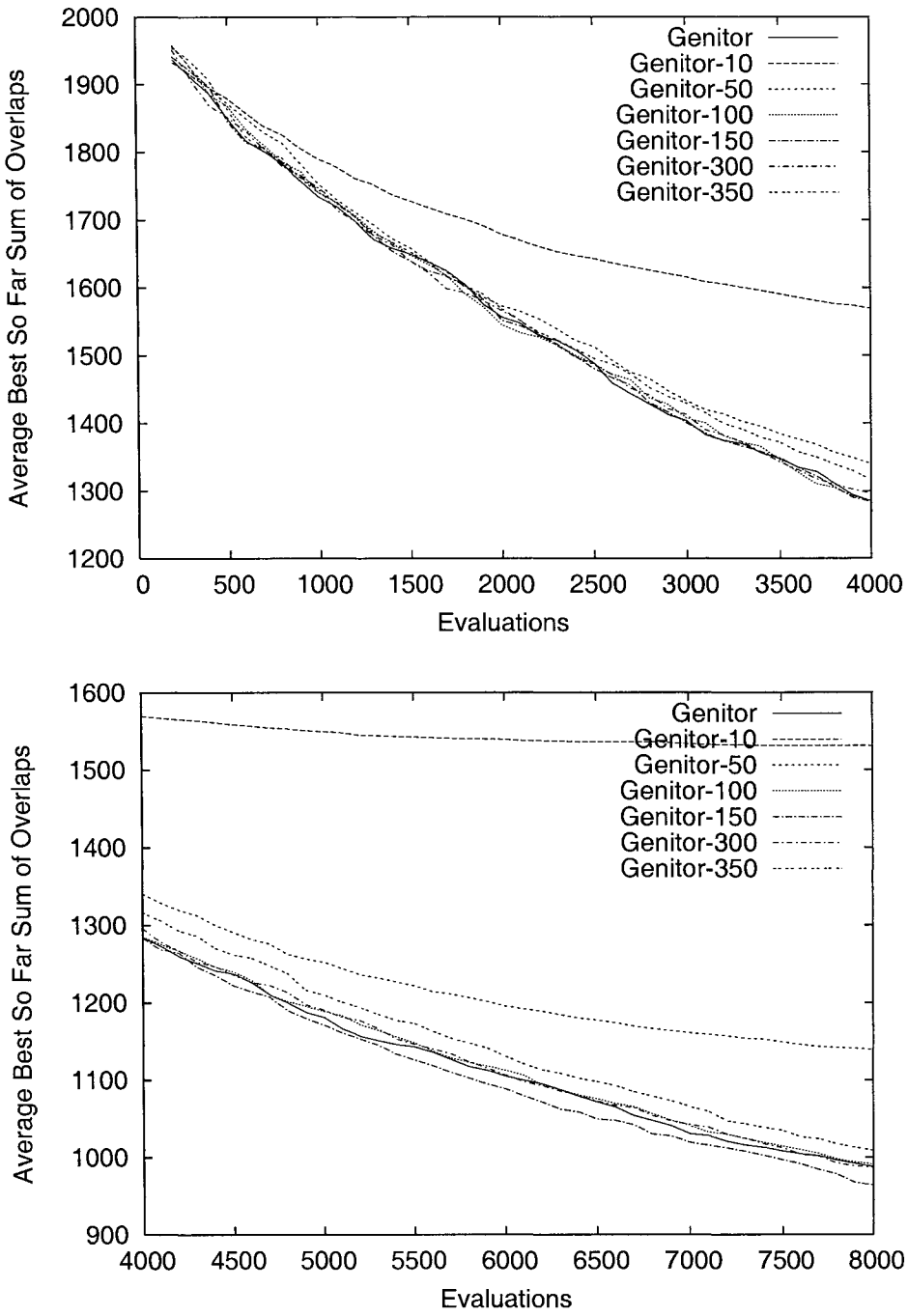


Figure 6.17: Evolutions of the average best value obtained by *Genitor* and its versions with a fixed number of selected positions for crossover. during 8000 evaluations, over 30 runs. The graphs were obtained for $R1$; best solution value is 773.

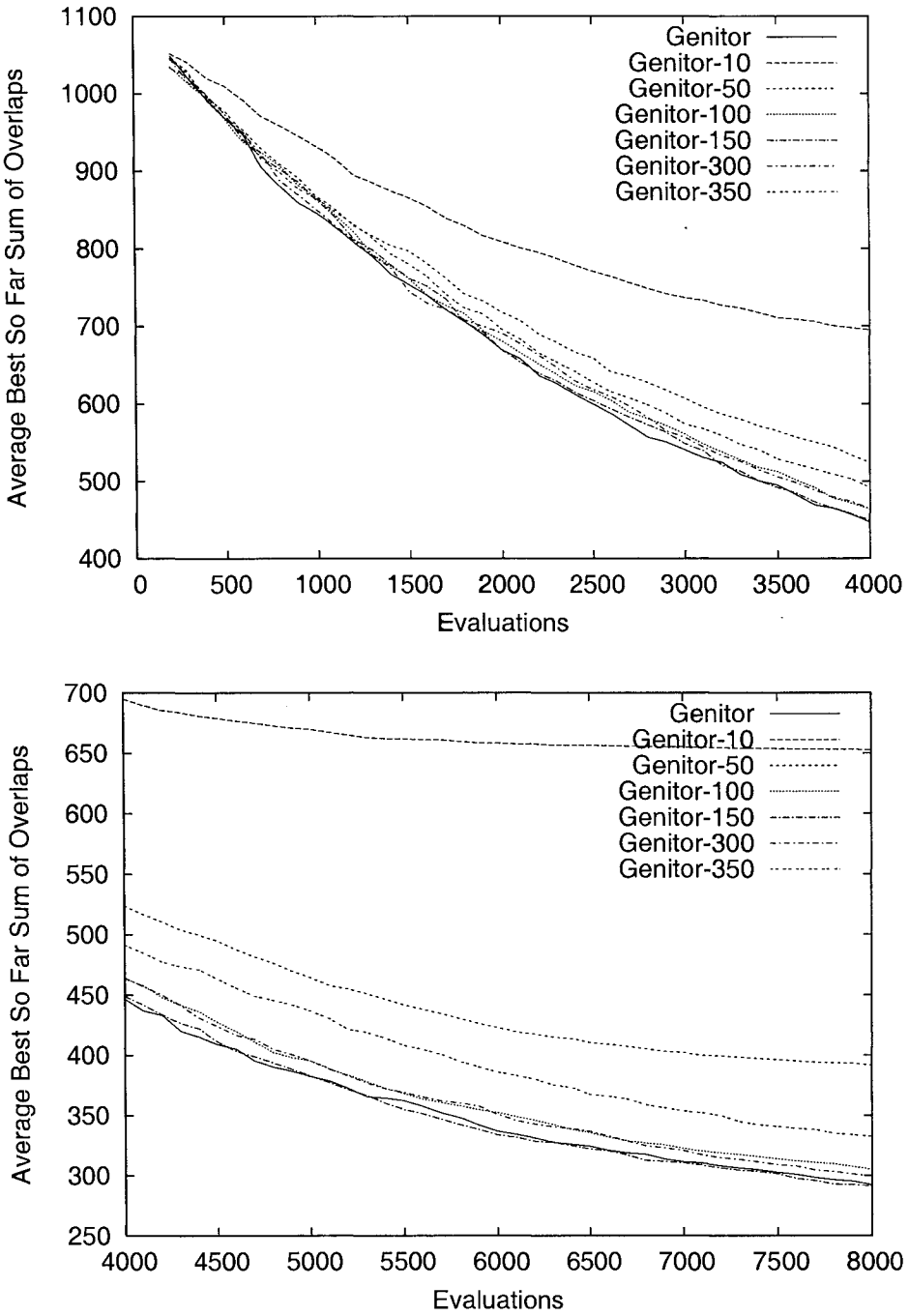


Figure 6.18: Evolutions of the average best value obtained by *Genitor* and its versions with a fixed number of selected positions for crossover. during 8000 evaluations, over 30 runs. The graphs were obtained for $R3$; best solution value is 250.

6.6.2 The Effect of Multiple Moves on *SWO*

We hypothesize that the multiple moves present in *SWO* are necessary for its performance. To test this hypothesis, we start by investigating the effect of moving forward only one request. This is somewhat similar to the shifting operator present in both *RandLS* and *StructLS*: a request is shifted forward in the permutation. However, to implement the SWO reprioritization mechanism, we restrict both the chosen request to be moved and the position where it gets moved. 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[5]. We call this new algorithm *SWO1Move*. The results obtained by running *SWO1Move* for 30 runs with 8000 evaluations per run are presented in Table 6.31. The initial solutions are identical to the solutions produced using the flexibility heuristic for initializing *SWO*.

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. Note that the initial greedy permutation produces a best known valued schedule for $R5$, and therefore we exclude $R5$ when comparing *SWO1Move* and *SWO* (it was included in Table 6.31 only for completeness). Previously, we implemented *SWO1Move* for minimizing overlaps by moving forward the request that contributes most to the total overlap [BWH04]. Randomly choosing the request to be moved forward improved the

[5]We tried other values; on average, a value of five seems to work best.

performance of *SWO1Move*[6]; 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* for $R1$, $R2$, $R3$ and $R4$, when minimizing overlaps. 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 cannot be scheduled. More fine tuning is needed when minimizing the sum of overlaps; *SWO1Move* also needs to find the positions for the overlapping 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.

For the current days of data and minimizing the overlaps, the results show that moving just one of the requests contributing to the overlaps does not produce results as good as those obtained by *SWO*. Next, we investigate the changes in performance when an increasing number of requests (from the requests contributing to the objective function) are moved forward.

We design an experiment where a constant number of the requests involved in conflicts are moved forward. To do this, we need to decide how many requests to move and which ones. Moving two or three requests forward results in small improvements over the results in Table 6.31. We determined empirically that when moving multiple requests (more than five) forward, choosing them at random as opposed to based on their contribution to the sum of overlaps hurts algorithm performance. Therefore, we

---

[6]Randomization is useful because *SWO* can become trapped in cycles [JC99]

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 6.31: 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.

Day	k=10			k=20			k=30			k=40		
	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev	Min	Mean	Stdev
R1	840	862.5	11.28	815	829.77	8.45	798	820.63	8.12	825	841.13	8.02
R2	512	530.2	9.18	498	506.53	5.25	493	508.97	5.26	508	526.26	6.25
R3	284	291.36	4.65	266	268.9	2.21	266	271.07	3.52	266	273.2	3.54
R4	764	778.57	8.45	749	757.3	6.16	740	744.47	2.6	737	747.2	5.06

Table 6.32: Performance of a modified version of *SWO* where k of the requests contributing to the sum of overlaps are moved forward for a constant distance 5. All statistics are taken over 30 independent runs, with 8000 evaluations per run.

run multiple versions of *SWO* moving k requests forward, with k = 5, 10, 20, 30, 40. To determine which requests are moved forward, at each step we sort the requests contributing to the sum of overlaps in decreasing order of their contribution and only move forward the first k (or all of them, if k is greater than the number of requests contributing to the sum of overlaps).

The results obtained for $R1$, $R2$, $R3$ and $R4$ are summarized in Table 6.32. For the A problems, all these new *SWO* versions find best known solutions. As noted above, for $R5$

the greedy initial permutation corresponds to a best known value schedule. The results show a general performance improvement as k grows from 10 to 20 and then 30. $k = 40$ results in worsening performance for $R1$ and $R2$. Note that the algorithm performance for $R3$ does not change when $k >= 20$. This is not surprising; since good solutions (in terms of overlaps) for this problem correspond to schedules with a small number of overlapping tasks (see Table 6.1), moving forward 20 requests or more means moving most of the requests in conflict once good solutions are found. The results indicate that for the problems in our set, when minimizing overlaps, if *SWO* is allowed to only move forward a constant number of requests, $k = 30$ produces best results.

To summarize, the results in this section show that the multiple moves are the source of power in both *Genitor* and *SWO*. Evidence for this is provided by: 1) The performance of *Genitor* worsens when only a small, fixed number of crossover positions is allowed. 2) For *SWO*, moving only one request forward (or a small number of requests, smaller than 30 for the R problems) results in worse performance.

Chapter 7

Plateaus in the Search Space

The AFSCN search space is dominated by plateaus. Evidence to support this is the fact that more than 50% of the neighbors in the shifting neighborhood result in exactly the same schedule for both overlaps and minimal conflicts evaluation function (see Section 6.3.1). Even more neighbors correspond to schedules with the same value as the current solution.

The three algorithms in our set handle the plateaus differently: while randomization is the most important algorithm feature for *RandLS*; for *Genitor* and *SWO*, multiple moves appear to matter most. We hypothesize that the plateaus are large. The presence of large plateaus would explain our previous results on algorithm performance factors: we conjecture that randomization and multiple moves can speed up plateau traversal.

The presence of plateaus can pose problems for search algorithms like *RandLS*, which rely only on gradient information to progress through the space. We expected *RandLS* to spend a significant number of its allowed evaluations on plateau moves; we hypothesized that *RandLS* randomly walks on each of the plateaus it encounters, until it finds an improving move to escape to a lower plateau. We cannot precisely determine how easy it is to escape the plateaus, because an exhaustive enumeration of the plateau states is impractical for our domain. Instead, we propose an empirical method to characterize the average length of plateau walks. We define a plateau walk as the suc-

cessive moves of equal value accepted during search before either an improving move is found or an upper limit on the number of such plateau moves is reached. Given a plateau, the average length of plateau walks represents a crude evaluation of the time spent by hill-climbing traversing that plateau. We collect data about the average length of plateau walks and find that the plateau walks become longer as the plateaus get lower; this suggests that the lower plateaus might be larger and/or more difficult to escape.

At each iteration, hill-climbing proposes incremental modifications to the current solution: one request is inserted in a different position in the permutation of requests. On the other hand, both *SWO* and *Genitor* propose multiple modifications to the current solution or set of solutions. Intuitively, we can view these multiple modifications as leaps through the search space. We conjecture that in the AFSCN search space, which is dominated by plateaus, algorithms taking large leaps in the space perform well, possibly because the leaps result in fast plateau traversal. In Section 6.6, we showed that the multiple moves are the source of power in both *Genitor* and *SWO*. The large leaps offer a partial explanation for these performance results. The large leaps conjecture also suggests an algorithm that should perform well for AFSCN scheduling: *RandLS* modified such that multiple moves are allowed at each iteration. We implement a version of the multiple-move hill-climbing and show that it either outperforms or performs as well as any of the three algorithms in our set.

7.1 Related Work

Large flat regions (plateaus) in the search space are typical for combinatorial optimization problems [HS05] and are known to pose problems for local search algorithms, which progress by finding and accepting improving moves from the neighborhood of the current solution. Frank et al.[FCS97] and Hoos and Stützle[HS05] formally define a number of terms that are used to characterize the plateaus. The *level* of a position in

the search space represents the value assigned by the objective function to that position. Intuitively, a plateau in the search space represents a set of connected positions such that all positions have the same level. The *border* of a plateau is defined as the set of positions in a plateau that have at least one neighbor outside the plateau. An *exit* from a plateau is a position included in the border that has at least one neighbor at a lower level than the plateau level. If a plateau does have exits, then it is called a *bench*. Plateaus without exits are *locally optimal*; if the level of a locally optimal plateau is equal with the optimal value of the objective function, the plateau is also *globally optimal*.

Plateaus in the search space have been extensively studied for the satisfiability SAT decision problems [FCS97, HS05]. Frank et al.[FCS97] characterized plateaus for 3-SAT problems (where each clause specifies 3 variables); they used GSAT to find the plateaus. GSAT is a hill-climber that always chooses to flip the variable minimizing the number of unsatisfied clauses in the current solution. Frank et al. found that most local minima tend to be very small (median size 48), while benches are typically 10-30 times larger than local minima. Most of the time there exist many exits from benches; also benches at higher levels have more exits than benches at lower levels. In fact, for many of the benches, most of the positions are on the border. However, some of the benches are very large with few exits, and can be very hard to escape.

Hoos and Stützle [HS05] defined plateau connection graphs (PCG) to model the plateaus in the search space. PCGs are built by aggregating all the states on the same plateau into a single vertex, and modeling the exits connecting the plateaus as edges. The PCGs for an easy and a hard instance show that for the easy instance the optimal solution can be easily reached from anywhere, while, for the hard instance, most searches will get stuck in a locally optimal plateau that is not globally optimal and from which it is hard to reach an optimal solution. The likelihood of "reaching" a plateau P' in the search space from a plateau P is computed as the number of exits from the current

plateau P that have neighbors in P' divided by the total number of exits from P; this also indicates how strongly two plateaus are connected. Hoos and Stützle conjecture that these differences in plateau connectivity might be correlated to the differences in problem difficulty between problems that are very similar otherwise (in terms of the number of variables, the number of clauses and the way such problems were generated). However, while PCGs can provide a good understanding of the search space structure, building them is computationally very expensive.

As noted in the introduction of this chapter, we conjecture that by applying multiple modifications to the current solution, the plateaus in the AFSCN search space can be traversed faster. Jumps through the search space by allowing for multiple modifications to the current solution have also been implemented by Gent and Walsh [GW95] for SAT in an algorithm called Jump-SAT. Jump-SAT is very similar in concept with *SWO*. It combines hill-climbing with a mechanism that flips a variable from each unsatisfied clause, causing all the previously unsatisfied clauses to become satisfied (of course, this might cause other clauses to become unsatisfied). The idea behind Jump-SAT is to allow the search to move large distances in the search space, by making larger steps. While not state of the art for SAT, JumpSat was shown to outperform greedy local search. Taking larger steps in the search space improved the performance of the original hill-climbing algorithm. This suggests some generalization potential for the multiple move conjecture we formulated for AFSCN scheduling: allowing multiple moves in a search space dominated by large plateaus might improve algorithm performance results for other combinatorial optimization problems.

Plateaus have been extensively studied for the SAT problems. In scheduling, to the best of our knowledge, analyses of the plateaus in the search space have been published only for job-shop scheduling and for a timetabling problem. For job-shop, Watson [Wat03] found that there are many plateaus in the search space. Most plateaus are very

small (10 or fewer solutions); there exist only a few extremely large plateaus. However, for almost 60% of all the plateaus examined, no exits were present. This is very different from the results reported for SAT, where most states on a plateau have exits (are border states). When exits were present in a plateau, there were only a few exits. This suggests that plateau walks (allowing equally good moves to be accepted) are unlikely to be effective for job-shop scheduling.

Chiarandini and Stützle [CS03] performed an analysis of the plateaus for the the university course timetabling problem (UCTP). The UCTP is the problem of assigning rooms and timeslots to a number of courses. The problem is overconstrained; violation of certain constraints (soft constraints) is allowed and the objective function minimizes the number of such constraint violations. The authors note that in fact solutions with zero violations are possible for all problems in their set. For the UCTP, the size of the plateaus is in most cases very large; sometimes however, the plateaus are small. Walks on plateaus are longer for the larger plateaus. The authors report that for a next descent version of hill-climbing the number of improving moves increases when equally good moves are allowed. The number of steps on plateaus is on average nine times bigger than the number of improving moves. While 90% of the accepted moves are plateau moves, the plateau walks are effective (result in an increase in the number of improving moves), probably because a high number of exits are also present.

7.2 Plateaus in the AFSCN Scheduling Domain

In Section 6.3.1 we showed that the permutation search space is highly redundant: there exists a many to one mapping from the permutation space to the space of schedules. Given an arbitrary permutation in the search space and the schedule produced for it by the greedy schedule builder, approximately 60% of all possible shifting moves leave the schedule unchanged. The many to one mapping is a direct consequence of the in-

teraction between the greedy, deterministic schedule builder and the problem domain. Changing the position of a request X in a permutation will result in a change in the schedule only if X competes for resources with at least one of the requests located in the permutation between the old and new position of X. This means that flat regions (plateaus) must be present in the search space. Also, given the large neighborhood size and the high percentage of equally valued neighbors, the plateaus must be relatively large in size.

In this section, we examine two hypotheses. First, we hypothesize that large plateaus are present in the AFSCN search space. Second, we hypothesize that lower plateaus (corresponding to better solutions) are more difficult to escape (i.e., are larger with fewer exits). The fact that lower plateaus are more difficult to escape is consistent with the observation that *SWO* performance levels off quickly and *RandLS* levels off in the second half of the search (see graphs in Section 5.3).

7.2.1 Presence of Plateaus

To assess the presence of plateaus in the search space, we examine the area around both random and optimal solutions. We collect 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 subsume the ones identical with the current solution (for identical schedules, we presented results in Section 6.3.1). In Table 7.1, 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. 2) When minimizing overlaps, more than 62% (usually around 70%) of the shifts result in same value schedules. This means that most of the accepted moves during search are non-improving moves; it appears that search ends up randomly walking on a plateau until an exit is found.

To get a more complete picture of the percentages of equal, worse and better neigh-

Day	Total Neigh.	Minimizing Conflicts				Minimizing Overlaps			
		Random Perms		Optimal Perms		Random Perms		Optimal Perms	
		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 7.1: 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

bors *during search*, we compute the entire shifting neighborhood for each of the accepted moves during one run of the RandLS, when minimizing conflicts. Each time a move is accepted, we generate the entire shifting neighborhood for the accepted move and count how many of the neighbors from this neighborhood are better, equal or worse than the accepted move. Note that the sum of these three numbers is equal to the size of the neighborhood. We transform each of the three numbers in the corresponding percentage value. For all the problems in our AFSCN data sets, we find that across the whole search 80% or more of all the neighbors have the same value as the current solution. For example, in Figure 7.1, we show the changes in the percentages of equal, worse and better neighbors during one run of *RandLS* using the randomized version of the neighborhood for A2. We saw similar behavior for the other problems. For A2, the moves accepted between evaluations 1400 and 2800 are, in fact, positions inside the plateaus (with 0 better neighbors) and their value is worse by one than the best known. The best known value for this run is found after 2949 evaluations.

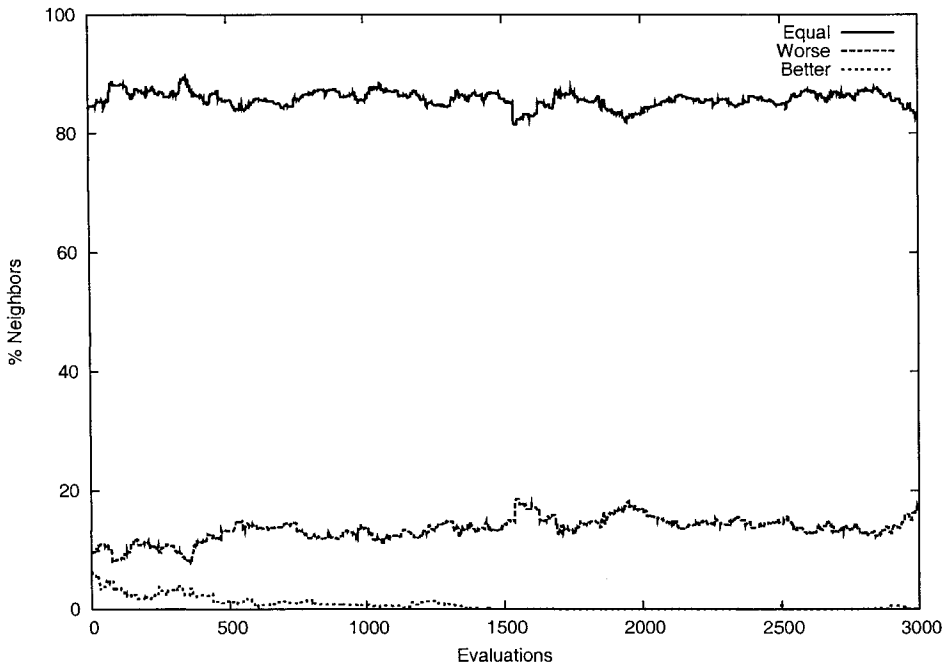


Figure 7.1: Percentage of equal, worse and better neighbors during a run of *RandLS* for A2, when minimizing the number of conflicts. The best known value for this problem is found after 2949 evaluations.

7.2.2 Impact of Plateaus on Hill-Climbing

As shown by the values in Tables 6.11 and 6.12 (Section 6.3.2), hill-climbing spends most of the time traversing plateaus in the search space by accepting non-improving moves. The progression graphs in Section 5.3 show that *RandLS* takes longer to find improvements in the second half of the search. In this subsection we provide some explanations for the evolution of *RandLS* by showing that as search progresses, the plateau walks become longer before finding an exit. This supports our hypothesis that lower plateaus (corresponding to better solutions) are more difficult to escape.

We estimate the length of plateau walks for *RandLS* by sampling a number of plateaus and computing the average number of plateau steps before a better solution is found. To do this, we consider one run of *RandLS* with 8000 evaluations allowed and identify the best solution found so far, BSF, after 500 evaluations, 1000 evaluations

and so on (in increments of 500 evaluations). Starting from each BSF, we perform 100 iterations of the *RandLS*. For each iteration, we compute the length of the plateau walk (the number of steps on the plateau before a solution better than BSF is found or a maximum number of steps is reached). We chose a limit of 1000 steps for the old days of data and 8000 steps for the more recent days. We compute the average length of the plateau walk over the 100 *RandLS* iterations performed for each BSF; this represents the average number of steps needed to find a better solution (or an exit from the plateau).

In Figure 7.2 we display the results obtained for $R4$; we observed similar behavior for the other 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 BSF solutions of value 729 end up taking the maximum number of steps allowed (8000) without finding an exit from the plateau.

The random walk steps count only equal moves; the number of evaluations needed by *RandLS* (x-axis) is considerably higher, due to needing to check detrimental moves before accepting equal ones. We chose to use the number of evaluations on the x-axis (instead of number of accepted moves) for two reasons: 1) we wanted to show how the value of the BSF changes during one run of *RandLS*, and at the same time show the average length of the random walk on the plateaus, and 2) if number of accepted moves would be used on the x-axis, then x and y would show similar information: x would show the length of a particular random walk on each plateau encountered and y, the corresponding average lengths for 100 random walks on the same plateaus.

The results show that large plateaus are present in the search space, and as search advances through the search space, the number of steps needed to exit a plateau tends to increase. Note that the states corresponding to BSF of value 30 are on the same plateau; in some cases though, there is minimal overlap between the error bars. 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 is a very small percentage of the total number of neighbors and therefore local search has a very small probability of finding an exit. A higher number of random walks on each plateau would probably be needed to more accurately show the average random walk length. Still, the graphs in Figure 7.2 show a trend: as the plateaus correspond to better values, the number of steps to find an exit increases.

Using the terminology introduced by Frank et al. [FCS97], 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. Instead, we focused on the average length of the plateau walk to show how the plateaus influence local search performance. The length of the plateau walk depends on two factors: the size of the plateau and the number of exits from the plateau. The number of improving neighbors for a solution decreases as the solution becomes better (as shown in Figure 7.1); 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, enumerating all the states on a plateau for the AFSCN domain is impractical (following a technique developed by Frank [FCS97], just the first iteration of breadth first search would result in approximately $0.8 * (n-1)^2$ or approximately 162,000 states on the same plateau; in the second iteration there would be approximately $162,000 * 0.8 * (n-1)^2$ or $162,000^2$ states etc.).

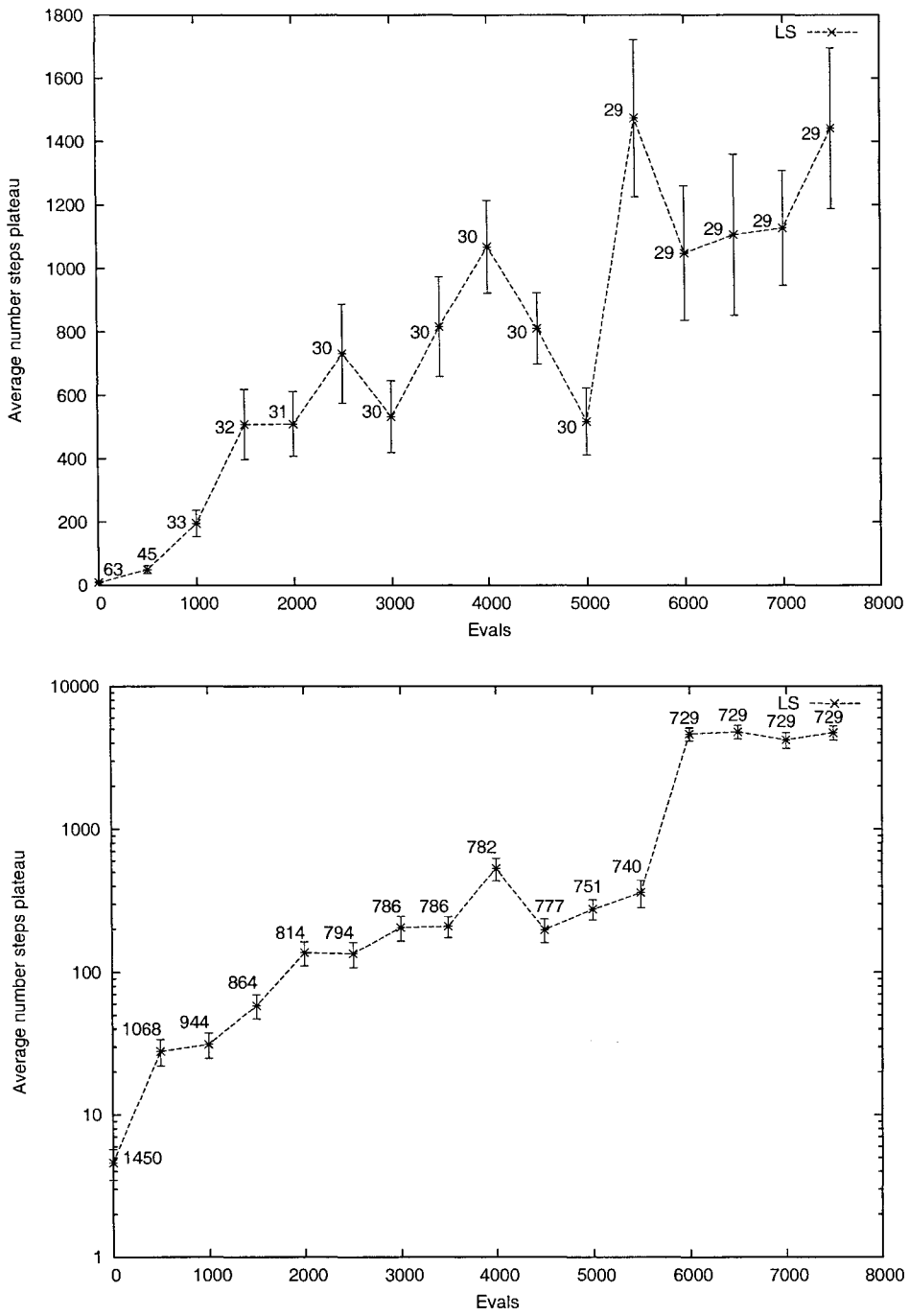


Figure 7.2: Average length of the plateau walk when minimizing conflicts (top) or overlaps (bottom) for a single *RandLS* run on R4. The labels on the graphs represent the value of the current solution. Note the *log* scale on the y axis when minimizing overlaps. The best known values are 28 when minimizing conflicts and 725 when minimizing overlaps.

7.3 Multiple Moves Hypothesis

During each iteration, hill-climbing proposes minor changes to the current solution: shifting request X from position x to position y only changes the relative orders between X and the requests appearing between x and y (including the request in the y position). All the other relative orders between requests remain unchanged. On the other hand, *SWO* and *Genitor* propose multiple modifications to the current solution/solutions at each iteration. Given the interaction mentioned in the previous section between the greedy schedule builder and the problem domain, such multiple modifications increase the likelihood of obtaining a schedule that is different from the current one. We conjecture that *SWO* and *Genitor* can traverse the plateaus faster because they perform multiple changes in the current solution at each iteration. To support this conjecture, we show that hill-climbing can traverse the plateaus significantly faster if multiple instead of incremental moves are allowed at each iteration.

The shifting operator for hill-climbing moves only one request to obtain a neighbor of the current solution. In contrast, the reprioritization mechanism in *SWO* is the equivalent of shifting forward **all** the requests involved in conflict in the current solution. This suggests a possible modification of the hill-climbing move operator: choose a number of pairs of positions and apply shifting to these pairs, one after another, without building the schedule after each shift; build the schedule only after shifting has been applied for all the pairs. We choose to perform 10 shifts for each multiple shifting move; we call this operator *10-Shift RandLS*. We choose to use 10 shifts for each move because it performs better than other values we tried: we found that allowing for a smaller number of shifts per move (five) or a larger number of shifts per move (20) produces worse performance. Note that an overly high number of shifts per move results in jumping too far in the search space and missing the solution. We do not try to find the number of shifts that would optimize the performance of the multiple-move hill-climber: the purpose of

implementing *10-Shift RandLS* is to provide additional support for the multiple move conjecture. We are investigating the questions: 1) Does *10-Shift RandLS* find improving solutions faster than *RandLS*? 2) Given the similarity with *SWO*, how does *10-Shift RandLS* compare with *RandomStartSWO*[1]?

The results of 30 runs of *10-Shift RandLS* with 8000 evaluations per run are presented in Table 7.2. Also, typical progression graphs are shown in Figures 7.3, 7.4, 7.5 and 7.6. When comparing the final solutions, *10-Shift RandLS* performs similarly to *RandLS* when minimizing the conflicts. When minimizing the overlaps, the results are similar only for the A problems; for the R problems, *RandLS* finds better solutions for R1 and R3. We conjecture that the reason for such difference in the performance of *10-Shift RandLS* and *RandLS* is the fact that a static number of the shifts to be performed during each move is probably not a good idea. As the search progresses to better solutions, the number of shifts should also decrease to allow for more precision in sampling the space around the current solution. The strength of *10-Shift RandLS*, however, is in finding improving moves fast, faster than *RandLS* and getting close to the improvement rate for *RandomStartSWO*, as shown in the progression graphs. The results of this experiment support our conjecture that large leaps through the search space speed up plateau traversal.

Similarly to these results, we showed that for *SWO*, simultaneously moving multiple contentious requests forward in the permutation produces better schedules than moving just one such request at a time (see Section 6.6). We hypothesized that multiple moves are useful in the AFSCN scheduling domain to find better solutions faster. In fact, the hypothesis might hold for other oversubscribed scheduling problems as well. Al Globus

---

[1]We compare with *RandomStartSWO* (instead of *SWO*) since it starts from the same random initial solutions as *10-Shift RandLS* does.

Day	Minimizing Conflicts			Minimizing Overlaps		
	Min	Mean	Stdev	Min	Mean	Stdev
A1	**8**	8.3	0.47	**104**	106.9	0.55
A2	**4**	4.0	0.0	**13**	13.0	0.0
A3	**3**	3.0	0.0	**28**	28.27	0.9
A4	**2**	2.0	0.0	**9**	9.4	1.22
A5	**4**	4.1	0.3	**30**	30.23	0.43
A6	**6**	6.0	0.0	**45**	45.07	0.37
A7	**6**	6.0	0.0	**46**	46.0	0.0
R1	**42**	42.9	1.03	879	973.63	52.88
R2	**29**	29.17	0.38	493	515.87	19.2
R3	**17**	17.47	0.5	252	284.83	39.01
R4	**28**	28.03	0.18	**725**	742.57	18.57
R5	**12**	12.0	0.0	**146**	146.0	0.0

Table 7.2: Statistics for the results obtained in 30 runs of *10-Shift RandLS*, with 8,000 evaluations per run. The mean and best value from 30 runs as well as the standard deviations are shown.

Day	Minimizing Conflicts			Minimizing Overlaps		
	Min	Mean	Stdev	Min	Mean	Stdev
A1	**8**	8.2	0.4	**104**	107.1	1.24
A2	**4**	4.0	0.0	**13**	13.0	0.0
A3	**3**	3.0	0.0	**28**	28.33	1.3
A4	**2**	2.03	0.18	**9**	9.13	0.73
A5	**4**	4.1	0.3	**30**	30.23	0.43
A6	**6**	6.0	0.0	**45**	45.0	0.0
A7	**6**	6.0	0.0	**46**	46.0	0.0
R1	**42**	42.63	0.72	785	817.83	27.07
R2	**29**	29.1	0.3	490	510.37	19.14
R3	**17**	17.5	0.57	**250**	273.33	43.68
R4	**28**	28.07	0.25	**725**	740.07	19.56
R5	**12**	12.0	0.0	**146**	146.03	0.19

Table 7.3: Statistics for the results obtained in 30 runs of ALLS, with 8,000 evaluations per run. The mean and best value from 30 runs as well as the standard deviations are shown.

et al.[GCLP04] found that when solving the oversubscribed problem of scheduling fleets of Earth Observing Satellites (EOS) using hill-climbing, moving only one request at a

time was inefficient. They propose a move operator where a random number (between 1 and 15) of requests are moved to generate the neighbors of the current solution. Similarly to our domain, a permutation representation and a greedy deterministic schedule builder are used. We conjecture that their schedule builder also results in multiple permutations being mapped to the same schedule, and therefore that plateaus are present in the EOS search space as well. The fact that moving more than one request improved the results suggests that our conjecture could also hold for EOS scheduling: multiple moves might speed up plateau traversal for this domain as well.

7.3.1 New Algorithm: Attenuated Leap Local Search

As we showed for *10-Shift RandLS*, the quality of the final solutions obtained is either equally good or sometimes worse than for the single move *RandLS*. We hypothesize that the reason for such performance difference is the fact that 10 shifts result in a leap that might be too long when getting closer to the optimal solution. A better idea would be to start by taking large leaps in the search space, and as the solutions improve, to lower the number of shifts performed during each move. This is similar to the idea behind the "temperature dependent" hill-climbing move operator implemented by Al Globus et al.[GCLP04], for which the number of requests to move is chosen by random but biased such that a large number of requests are moved early in the search while later only a few requests are moved[2]. Hill-climbing with the temperature dependent operator produced better results for EOS than simply choosing a random number of requests to move.

In the same spirit, we implement a multiple move hill-climber with a variable move count operator: given a decay rate, we start by shifting ten requests, then nine, eight

[2]The operator is similar to the temperature dependent behavior in simulated annealing; this explains the name of the operator.

etc. We choose to decrement the number of shifts for every 800 evaluations; we call this version of hill-climbing *Attenuated Leap Local Search* (ALLS). ALLS performs remarkably well: in 30 runs with 8000 evaluations per run, it finds better best values than all the algorithms in our set when minimizing overlaps for the R days (see Table 7.3). It also finds best known values for all the other combinations of problem/objective function, similarly to the algorithms in our set: *Genitor*, *SWO* and *RandLS*. In fact, a single tailed, two sample t-test comparing ALLS to *RandLS* shows that ALLS finds statistically significantly better solutions ($P < .023$) on both conflicts and overlaps for the five more recent days.

ALLS also progresses through the search space at a rate similar to the *10-Shift RandLS* (see Figures 7.3 7.4, 7.5 and 7.6). ALLS achieves such great performance by combining the power of finding good solutions fast using multiple moves in the beginning of the search with the accuracy of locating the best solutions using one-move shifting at the end of the search.

In this chapter, we showed that large plateaus are present in the search space and that the lower plateaus are more difficult to escape than the higher ones. We formulated the conjecture that large leaps through the search space might speed up plateau traversal, resulting in improved algorithm performance. We found evidence supporting this conjecture in the fact that both *Genitor* and *SWO* take large leaps through the search space: *Genitor* via the crossover operator and *SWO* using the reprioritization mechanism. For further evidence, we showed that a multiple move version of hill-climbing finds improvements faster than the traditional one-move version. In fact, our ALLS finds better best solutions for the R problems when minimizing overlaps in 30 runs with 8000 evaluations per run than all the three algorithms in our set: *Genitor*, *SWO* and *RandLS*. It also finds best known values for all the other combinations of problem/objective function. The multiple move mechanism allows ALLS to find improvements at a rate that is

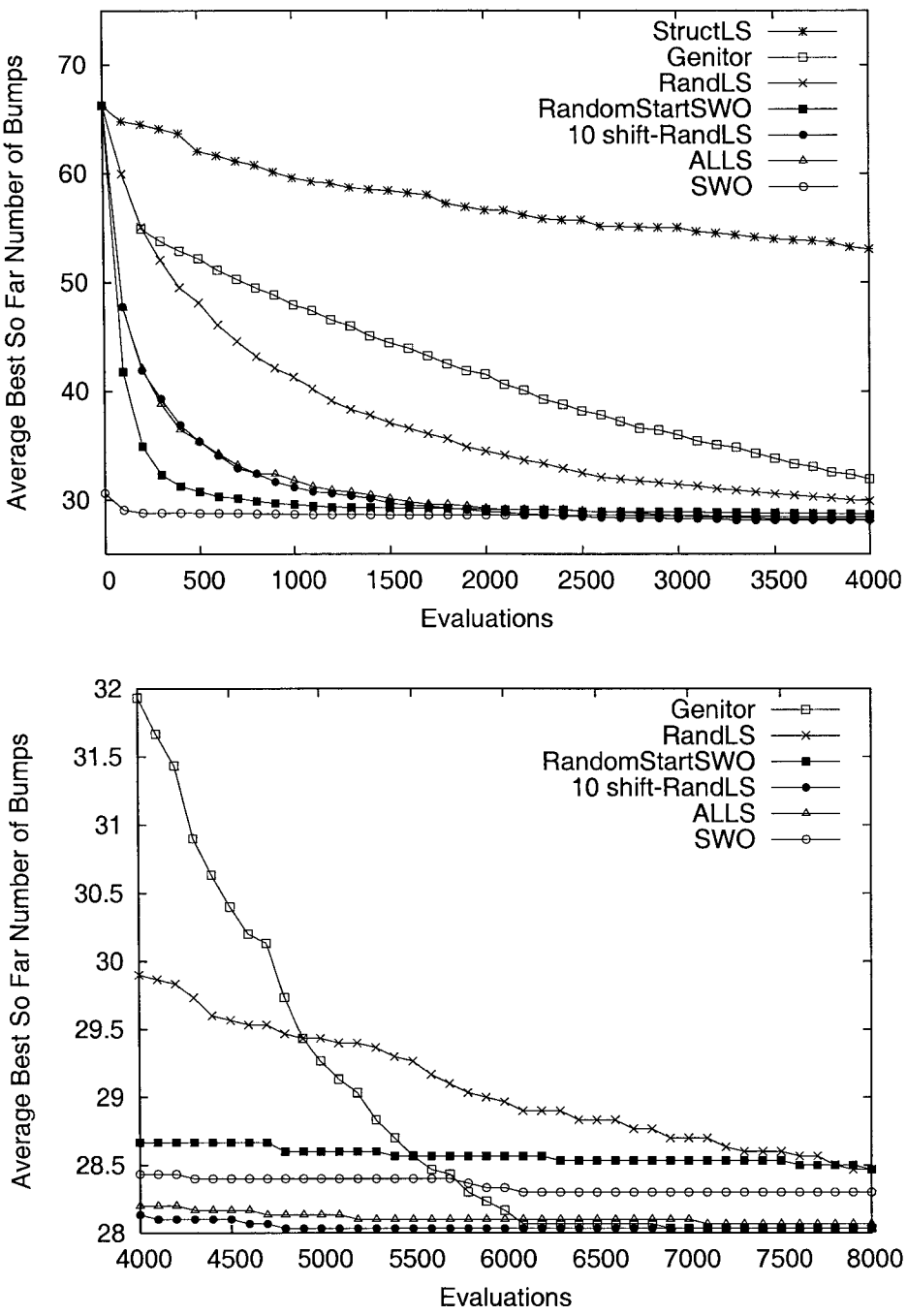


Figure 7.3: Evolutions of the average best value obtained by *10-Shift RandLS* and ALLS during 8000 evaluations, over 30 runs. The improvement over the first 4000 evaluations is shown in the top figure. The last 4000 evaluations are depicted in the bottom figure; note that the scale is different on the y-axis.The graphs were obtained for $R4$; best solution value is 28.

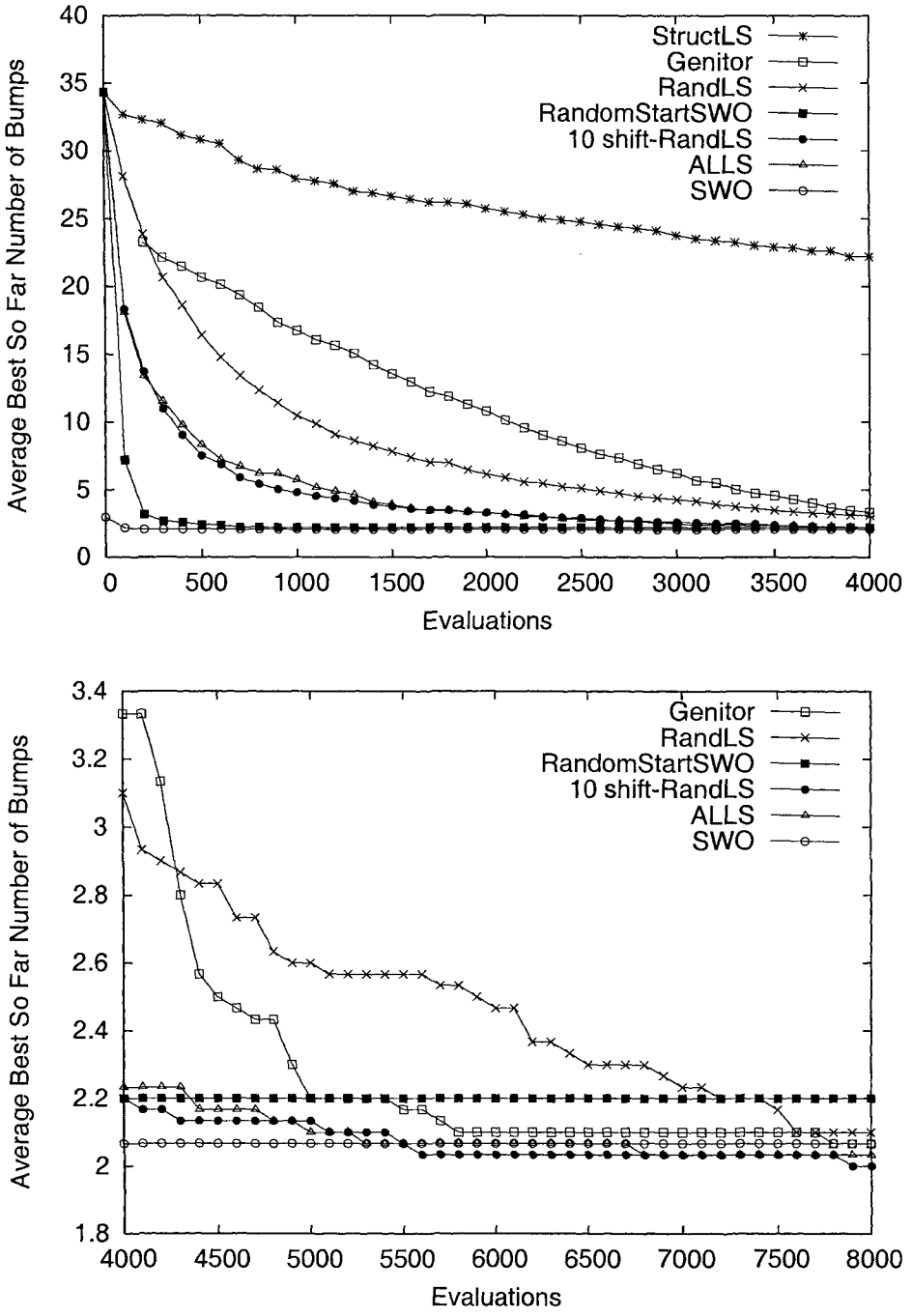


Figure 7.4: Evolutions of the average best value obtained by *10-Shift RandLS* and ALLS during 8000 evaluations, over 30 runs. The improvement over the first 4000 evaluations is shown in the top figure. The last 4000 evaluations are depicted in the bottom figure; note that the scale is different on the y-axis.The graphs were obtained for $A4$; best solution value is 2.

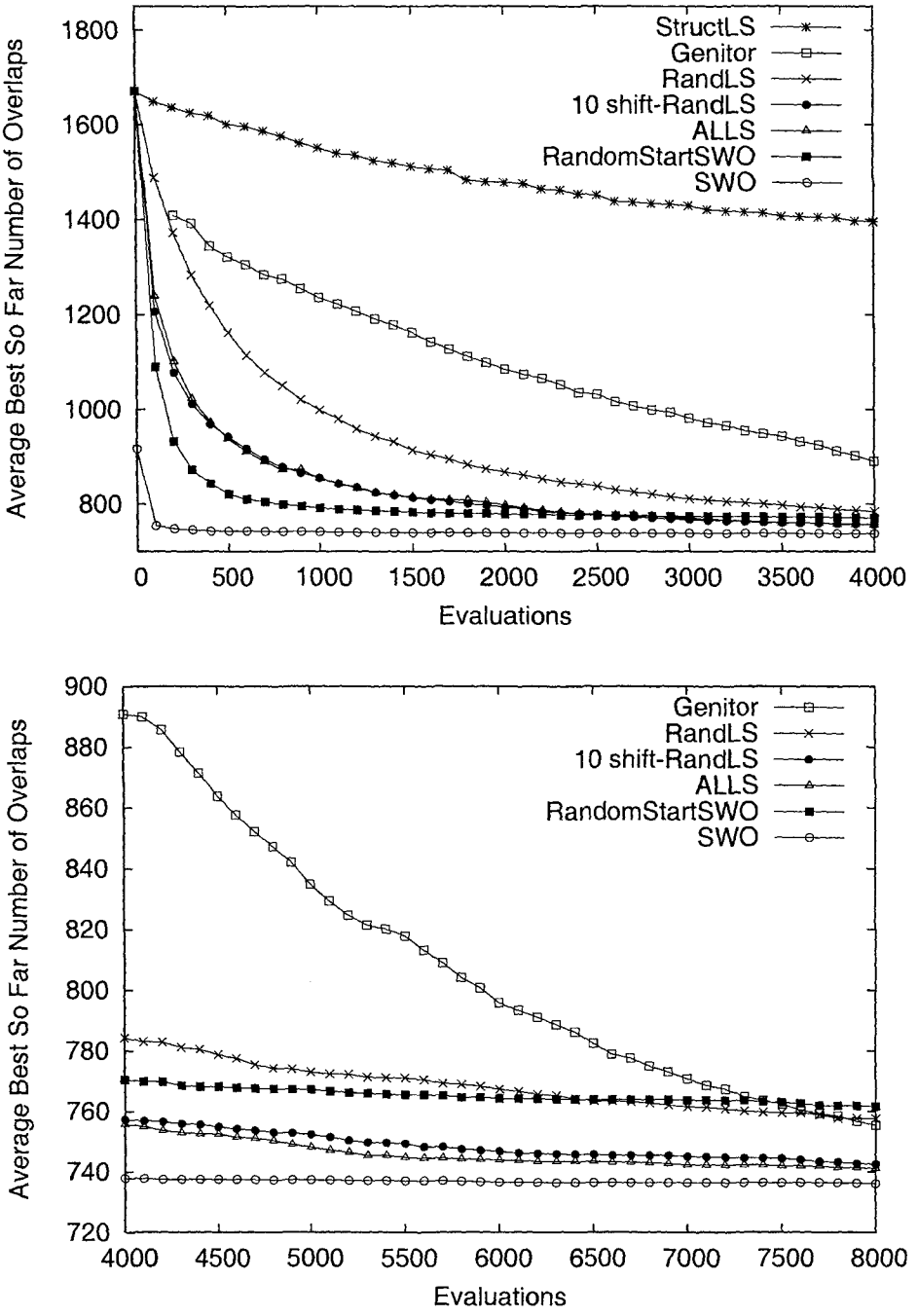


Figure 7.5: Evolutions of the average best value obtained by *10-Shift RandLS* and ALLS during 8000 evaluations, over 30 runs. The improvement over the first 4000 evaluations is shown in the top figure. The last 4000 evaluations are depicted in the bottom figure; note that the scale is different on the y-axis.The graphs were obtained for $R4$; best solution value is 725.

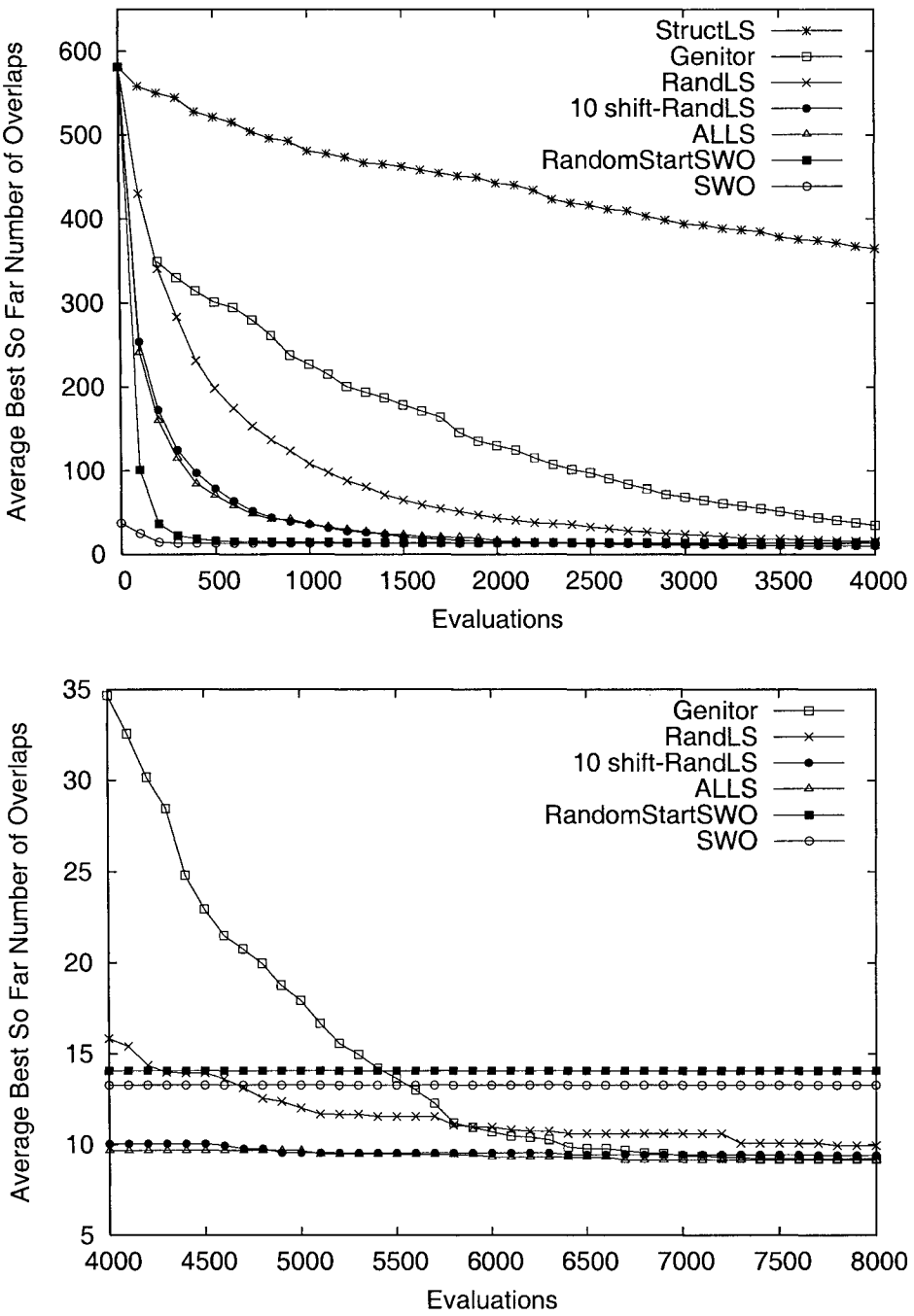


Figure 7.6: Evolutions of the average best value obtained by *10-Shift RandLS* and ALLS during 8000 evaluations, over 30 runs. The improvement over the first 4000 evaluations is shown in the top figure. The last 4000 evaluations are depicted in the bottom figure; note that the scale is different on the y-axis.The graphs were obtained for $A4$; best solution value is 9.

almost close to *SWO* in the beginning of the search. The analysis of the plateaus in the search space resulted in the implementation of ALLS with surprisingly good results.

Chapter 8

Summary, Future Work and Conclusions

For most scheduling problems of practical relevance, finding the optimal solution in a reasonably short time is not possible. Most research in solving real-world scheduling problems focuses on developing new algorithms and showing empirically that these algorithms perform better than previous attempts. Understanding why an algorithm performs well or poorly usually is not a research priority. When faced with a real-world application, the practitioners must quickly select or adapt an algorithm from a set of possible algorithms; a detailed analysis of the problem and its search space is time consuming.

However, understanding the factors that influence algorithm performance is important for at least three reasons: 1) it can demonstrate that a particular algorithm choice is appropriate for the application, 2) it may motivate algorithm improvements to better fit the domain and 3) given new problem instances, the algorithm most likely to perform best can be selected based on particular features of the new instances. The research presented in this dissertation is an analysis of the problem, algorithm and search space characteristics for a practical real-world application: AFSCN (Air Force Satellite Control Network) scheduling.

Although it is an important application for both the government (e.g., military and

NASA) and civilian applications (access to commercial satellites), the majority of previous work has been in proposing new heuristic methods without much examination of their strengths and limitations. While the application was relatively new, designing new algorithms and comparing them to previous ones was appropriate; at this stage, we need to understand the attributes of current problems and solutions to propose future directions for this dynamic application.

The main contributions of this research are:

- We proved that the decision version of AFSCN scheduling is NP-complete and that the low altitude requests can be scheduled optimally in polynomial time.

- We identified a set of simple algorithms that perform best: *Genitor*, a hill-climber and *SWO*. All these algorithms use a permutation representation and a schedule builder to transform permutations into schedules. We found that these algorithms perform better than domain-specific heuristics and repair-based approaches.

- We designed techniques to investigate reasons for algorithm performance. These techniques could transfer to other optimization applications.

- We identified algorithm features that contribute to algorithm performance for AFSCN scheduling. We found that the ordering of the neighbors is the main performance factor for hill-climbing. The multiple moves are the main performance factor for *Genitor* and *SWO*.

- We found that large plateaus are predominant in the search space. As a result, we showed that good solutions can be reached faster by taking long leaps through the search space. We conjectured that the long leaps speed up plateau traversal.

- Based on our analyses, we designed a new algorithm, ALLS, that combines what appeared to be critical elements of the best performing algorithms. ALLS, a

multiple-move hill-climbing algorithm proved to perform better than the original three algorithms in our set.

8.1 Limitations

We start by presenting some limiting assumptions in the data[1]. The real-world AFSCN scheduling problem defines setup times dependent on the antennas at the ground stations. This setup times need to be scheduled before assigning a request to an antenna. In our data, the setup times are considered to be part of the duration of the requests and are fixed: 20 minutes for low altitude requests and 15 minutes for high altitude requests. From discussions with the human schedulers, the real setup times might in fact be smaller for some of the antennas. Therefore, the problems we solve might be more constrained in terms of the setup times than they are in practice.

Also, the problems we solve consider all antennas present at a ground station as possible alternate resources for a request. This is not always the case in practice; some antennas can only handle low-altitude requests. This becomes a problem if the number of high altitude tasks scheduled at the same ground station at the same time exceeds the number of antennas that can in fact handle high altitude requests.

The real-world data were made available to us in the DEFT format. We did not find proper documentation to extract the problem specification from the DEFT format. We used code published by Schalck [Sch93] in an AFIT Master's thesis to read the DEFT format. Thus, we make the same assumptions as those made by previous research from the Air Force Institute of Technology (AFIT). It is not clear how unrealistic these assumptions are.

---

[1]I wish to thank Dr. John Bresina from NASA Ames Research Center for his insightful suggestions about limitations and possible extensions of this research. Some of the ideas discussed here are based on his suggestions.

Next, we focus on the limitations of our empirical research. With the exception of HBSS, all the algorithms we tested for the real-world problem use an "indirect representation": they operate on permutations of requests and rely on a schedule builder to produce the schedules corresponding to these permutations. The schedule builder therefore becomes an important factor in algorithm performance; different schedule builders produce different schedules for the same permutation. This also makes it harder to predict the effect of applying modifications to the current solution: while the *permutation* corresponding to the solution does change, it is not clear if the *schedule* corresponding to the solution will also change. Also, the schedules reached by using the schedule builder represent a subset of the full space of schedules. We showed in Section 4.4.1 that this subset does contain an optimal solution for the objective function of minimizing the number of conflicts and the greedy scheduler. It is not clear however that the subset also contains an optimal solution when minimizing the sum of overlaps.

Given these limitations, a better approach might be to operate in the schedule search space (or have the algorithms directly build the schedule). We implemented various domain-specific heuristics to be used to directly solve the problem, based on resource contention, overlap of requests or flexibility of scheduling the requests. The best results we obtained using such "direct" approaches were produced by HBSS with the flexibility heuristic (see Section 5.2). We could not reproduce the good performance results reported by the indirect methods. We conjecture that we did not find the best heuristic to efficiently solve the problems.

After producing an initial schedule the human schedulers need to negotiate with the customers to alter the requests until all or most of them are scheduled. To help the human schedulers in the negotiation process, we defined a new objective function for this problem: we allowed all the requests to be scheduled and minimized the sum of request overlaps. While this objective function is more informative than minimizing the number

of conflicts, it does not consider the value of the contributing individual overlaps. If conflicts are removed by negotiating new durations, it is important to also restrict the maximum overlap for any request. Otherwise, resolving the conflict could be very difficult, if not impossible. A possible way to restrict the maximum overlap for a request is to consider the duration adjustment that should be performed by the scheduler to remove the overlap. This duration adjustment could be defined as a percentage of the total duration; the maximum duration adjustment for a request would need to be minimized in addition to minimizing the sum of duration adjustments for all the requests. Currently though we do not have enough information about the criteria used by the human schedulers to fit more requests into the schedule. It is not clear therefore that managing the individual overlaps would further help in the negotiation process.

8.2 Future Work

Beyond AFSCN scheduling, the main direction to extend this work is to consider the issue of generalization. Do our results transfer to other oversubscribed scheduling applications?

Our analyses show that large plateaus are present in the search space. It is reasonable to expect that this is a characteristic for other scheduling problems, for at least two reasons. First, alternative resources are specified in many scheduling problems. The alternative resources make it possible for a set of requests to be scheduled in multiple different ways on a set of identical alternative resources, without any effect on the objective function. The most simple example would be two schedules for which the only difference is that two resources swap the requests assigned to them: all requests scheduled on resources $R1$ and $R2$ in one schedule appear in the other schedule starting at the same times, but on resources $R2$ and $R1$, respectively. Second, many solutions to scheduling problems impose orderings between requests and define a schedule builder

to produce schedules. In these cases, redundancy in the search space is very likely. For example, a request that specifies a time window toward the end of the schedule will be scheduled at the end of the schedule even if it is the first request or second request etc. assigned by the schedule builder. Different orderings of the requests are likely to produce the same schedule. Hence, it seems likely that our results are transferable to other domains. Recent work of Roberts et al.[RHW05] applies some of the techniques to analyze the permutation search space to another oversubscribed application: scheduling Earth Observing Satellites.

Our algorithms should be easy to adapt to other domains; we only need to implement a new schedule builder to account for the specific domain constraints, without the need to design special domain heuristics. Unfortunately, data for real-world problem instances in scheduling are notoriously hard to obtain, and it is difficult to build a problem generator for a scheduling application without being aware of all the particular characteristics of that problem.

A first step in the direction of generalization would be to consider the work of Al Globus et al. [GCLP04] on the problem of scheduling Earth Observing Satellites (EOS). There are strong similarities between the research of Al Globus et al. and our work, such as: they also define a greedy deterministic scheduler to transform permutation solutions into schedules, and more importantly, they also find that multiple moves are more efficient than single moves. The techniques we defined for AFSCN scheduling could be used to answer questions such as: How much redundancy is present in the search space for EOS scheduling? What is the average length of the plateau walks? Is it still the case that lower plateaus are larger and/or have fewer exits?

Another oversubscribed application is the AMC airlift scheduling. Kramer and Smith [KS04] successfully apply a repair-based method to the AMC domain. For our domain, we found that repair-based methods are less successful. On the other hand, we

transfered the flexibility heuristic from the AMC to the AFSCN domain and showed that it still performs well. We can easily adapt our algorithms to the AMC domain (and other domains as well), since we only need to implement a schedule builder for the new domain. We would like to compare the performance of our set of algorithms with the repair-based methods for the AMC domain. For both AFSCN scheduling and EOS scheduling multiple moves proved to be more efficient than single moves. Are the multiple moves (large leaps) useful for the AMC domain as well?

8.3 Final Word

In this dissertation, we present the first coupled formal and empirical analysis of the AFSCN scheduling problem. Our theoretical results show that the decision version of AFSCN scheduling is NP-complete; also, we identify a polynomial subclass of of the problem. The contribution of our empirical 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, which could transfer to other applications. 3) We identify problem characteristics that influence algorithm performance, which are likely to hold on similar problems. Based on our analyses, we design a new algorithm that combines what we found to be the critical elements of the best performing algorithms; the new algorithm performs better than the original ones.

The main limitation of this work is the fact that it targets only one application: AFSCN scheduling. In future work, we will be testing other oversubscribed scheduling applications to determine to what extent our analyses and results generalize. We conjecture that our results might transfer to other applications, for which large plateaus dominate the search space. We have some evidence suggesting that this might be the case. First, we found that randomization helps hill-climbing performance. Randomiz-

ing the order of checking the neighbors in a hill-climber was also found by Forrest and Mitchell (1993) to work better than a structured ordering of the neighbors for a staircase like function (The Royal Road), where each step in the staircase is a plateau. Second, we found that multiple moves are a key algorithm characteristic. Al Globus et al. [GCLP04] also found that when solving the oversubscribed problem of scheduling fleets of EOS using hill-climbing, moving only one request at a time was inefficient.

We also conjecture that the effectiveness of multiple moves depends on the extent to which identical valued neighbors dominate the neighborhood of a solution. For our domain, approximately 70% or more of the neighbors have identical values to the current solution, regardless of the quality of the current solution. An "aggressive" multiple move mechanism, that makes many changes to the solution before evaluating in the beginning of the search is appropriate for such a highly redundant neighborhood. However, for problems with a much smaller percentage of identical neighbors, a less aggressive multiple move mechanism might perform better. Possible extensions of this research would examine the degree to which other oversubscribed scheduling applications 1) exhibit the same characteristics as AFSCN scheduling and 2) are amenable to the same kind of solution.

REFERENCES

[AD03] U. Aickelin and K. Dowsland. An indirect genetic algorithm for a nurse scheduling problem. *Computers & Operations Research*, 31(5):761–778, 2003.

[AGKS00] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating Satisfiable Problem Instances. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 256–261. AAAI Press / MIT Press, 2000.

[AS87] E.M. Arkin and E.B. Silverberg. On the k-coloring of intervals. *Discrete Applied Mathematics*, 18:1–8, 1987.

[BDSF97] J. C. Beck, A. J. Davenport, E. M. Sitarski, and M. S. Fox. Texture-based Heuristic for Scheduling Revisited. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 241–248, Providence, RI, 1997. AAAI Press / MIT Press.

[BHWR04] L. Barbulescu, A.E. Howe, L.D. Whitley, and M. Roberts. Trading places: How to schedule more in a multi-resource oversubscribed scheduling problem. In *Proceedings of the International Conference on Planning and Scheduling*, 2004.

[BNGNS02] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. Approximating the throughput of multiple machines in real-time scheduling. *SIAM Journal on Computing*, 31(2):331–352, 2002.

[BPP98] P. Baptiste, C. Le Pape, and L. Peridy. Global constraints for partial CSPs: A case-study of resource and due date constraints. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98*, pages 87–101. Springer, 1998.

[Bra01] T.D. Braun. Heterogeneous distributed computing: Off-line mapping heuristics for independent tasks and for tasks with dependencies, priorities, deadlines, and multiple versions. In *Ph.D. Thesis*. School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, 2001.

[Bre96] J.L. Bresina. Heuristic-Biased Stochastic Sampling. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 271–278, Portland, OR, 1996.

[Bre98] J.L. Bresina. Stochastic heuristic search and evaluation methods for constrained optimization. In *Ph.D. Thesis*. Graduate School- New Brunswick, Rutgers, The State University of New Jersey, 1998.

[Bur99] S.E. Burrowbridge. Optimal Allocation of Satellite Network Resources. In *Masters Thesis*. Virginia Polytechnic Institute and State University, 1999.

[BVA+96] E. Bensana, G. Verfaillie, J. Agnese, N. Bataille, and D. Blumstein. Exact and inexact methods for the daily management of an Earth observation satellite. In *Proceedings of the Fourth International Symposium on Space Mission Operations and Ground Data*, Munich, Germany, 1996.

[BWH04] L. Barbulescu, L.D. Whitley, and A.E. Howe. Leap before you look: An effective strategy in an oversubscribed problem. In *Proceedings of the Nineteenth National Artificial Intelligence Conference*, 2004.

[BWWH04] L. Barbulescu, J.P. Watson, D.L. Whitley, and A.E. Howe. Scheduling Space-Ground Communications for the Air Force Satellite Control Network. *Journal of Scheduling*, 2004.

[CL95] M.C. Carlisle and E.L. Lloyd. On the k-coloring of intervals. *Discrete Applied Mathematics*, 59:225–235, 1995.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT press, Cambridge, MA, 1990.

[Cra96] J.M. Crawford. An approach to resource constrained project scheduling. In G.F. Luger, editor, *Proceedings of the 1996 Artificial Intelligence and Manufacturing Research Planning Workshop*. The AAAI Press, Albuquerque,NM, 1996.

[CRK+00] S. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. ASPEN - Automating space mission operations using automated planning and scheduling. In *6th International SpaceOps Symposium (Space Operations)*, Toulouse (France), 2000.

[CS03] M. Chiarandini and T. Stützle. A landscape analysis for a hybrid approximate algorithm on timetabling problem. Technical Report AIDA-03-05, FG Intellektik, FB Informatik, TU Darmstadt, Germany, 2003.

[Dav85] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the ninth international joint conference on aritificial intelligence*, pages 162–164, Los Angeles, 1985.

[DP95] S. Dauzère-Pérès. Minimizing Late Jobs in the General One Machine Scheduling Problem. *European Journal of Operational Research*, 81:131–142, 1995.

[ECB+01] B. Engelhardt, S. Chien, A. Barrett, J. Willis, and C. Wilklow. The DATACHASER and Citizen Explorer benchmark problem sets. In *European Conference on Planning*, Toledo (Spain), 2001.

[FCS97] J. Frank, P. Cheeseman, and J. Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.

[FJMS01] J. Frank, A. Jonsson, R. Morris, and D. Smith. Planning and scheduling for fleets of earth observing satellites. In *Proceedings of the Sixth International Symposium on Artificial Intelligence, Robotics, Automation and Space*, 2001.

[FM93] S. Forrest and M. Mitchell. Relative Building-Block Fitness and the Building Block Hypothesis. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 109–126. Morgan Kaufmann, 1993.

[Fox94] M.S. Fox. ISIS: a retrospective. In Michael B. Morgan, editor, *Intelligent Scheduling*, pages 391–422. Morgan Kaufmann Publishers, 1994.

[FW92] E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. In *Artificial Intelligence*, volume 58, pages 97–109, 1992.

[GC96] J. Gratch and S. Chien. Adaptive problem-solving for large-scale scheduling problems: A case study. *Journal of Artificial Intelligence Research*, 4:365–396, 1996.

[GCLM02] A. Globus, J. Crawford, J. Lohn, and R. Morris. Scheduling earth observing fleets using evolutionary algorithms: Problem description and approach. In *Proc. of the Third International NASA Workshop on Planning and Scheduling for Space*, pages 27–29, October 2002.

[GCLP03] A. Globus, J. Crawford, J. Lohn, and A. Pryor. Scheduling earth observing satellites with evolutionary agorithms. In *International Conference on Space Mission Challenges for Information Technology*, Pasadena, CA, July 2003.

[GCLP04] A. Globus, J. Crawford, J. Lohn, and A. Pryor. A comparison of techniques for scheduling earth observing satellites. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04), Sixteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-04)*, pages 836–843, July 25-29 2004.

[GJ77] M.S. Garey and D.S. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM Journal on Computing*, 6:416–426, 1977.

[GJ79] M.S. Garey and D.S. Johnson. *Computers And Intractability: A Guide To The Theory Of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[GL85] David E. Goldberg and Robert Lingle. Alleles, Loci, and the Traveling Salesman Problem. In *Proceedings of the International Conference on Genetic Algorithms and Their Applications*, pages 8–15, Pittsburgh, PA, 1985.

[GLJK79] R.L. Graham, E.L. Lawler, J.K.Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Matehmatics*, 5:287–326, 1979.

[Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 5:533–549, 1986.

[Gol89] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[Goo93] T.D. Gooley. Automating the Satellite Range Scheduling Process. In *Masters Thesis*. Air Force Institute of Technology, 1993.

[GW95] I. Gent and T. Walsh. Unsatisfied variables in local search. In *Hybrid Problems, Hybrid Solutions*, pages 73–85. IOS Press Amsterdam, 1995.

[HG95] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Ar tificial Intelligence*, 1995.

[Hoo95] J.N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1:33–42, 1995.

[HS05] Holger Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, New York, 2005.

[Jan96] Kwangho Jang. The Capacity of the Air Force Satellite Control Network. In *Masters Thesis*. Air Force Institute of Technology, 1996.

[JC99] David E. Joslin and David P. Clements. "Squeaky Wheel" Optimization. In *Journal of Artificial Intelligence Research*, volume 10, pages 353–373, 1999.

[JM94] M.D. Johnston and G.E. Miller. Spike: Intelligent scheduling of Hubble space telescope observations. In Michael B. Morgan, editor, *Intelligent Scheduling*, pages 391–422. Morgan Kaufmann Publishers, 1994.

[KS03] L.A. Kramer and S.F. Smith. Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems. In *Proceedings of 18th International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, August 2003.

[KS04] L.A. Kramer and S.F. Smith. Task swapping for schedule improvement: A broader analysis. In *Proceedings of the Fourteenth International Conference on Planning and Scheduling*, 2004.

[KS05] L.A. Kramer and S.F. Smith. Maximizing availability: A commitment heuristic for oversubscribed scheduling problems. In *Proceedings of the Fifteenth International Conference on Planning and Scheduling*, 2005.

[LVJ00] M. Lemaître, G. Verfaillie, and F. Jouhaud. How to manage the new generation of Agile Earth Observation Satellites. In *6th International SpaceOps Symposium (Space Operations)*, Toulouse, France, 2000.

[OS97] A. Oddi and S.F. Smith. Stochastic procedures for generating feasible schedules. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-93)*, pages 308–314, Providence, RI, 1997. AAAI Press / MIT Press.

[Par94] D.A. Parish. A Genetic Algorithm Approach to Automating Satellite Range Scheduling. In *Masters Thesis*. Air Force Institute of Technology, 1994.

[Pem00] J.C. Pemberton. Toward Scheduling Over-Constrained Remote-Sensing Satellites. In *Proceedings of the Second NASA International Workshop on Planning an d Scheduling for Space*, San Francisco, CA, 2000.

[PG98] W.J. Potter and J. Gasch. A photo album of earth: Scheduling landsat 7 mission daily activities. In *Proceedings of Fifth International Symposium on Space Mission Operations and Ground Data Systems (SpaceOps 98)*, 1998.

[RCWM99] G. Rabideau, S. Chien, J. Willis, and T. Mann. Using iterative repair to automate planning and scheduling of shuttle payload operations. In

Innovative Applications of Artificial Intelligence (IAAI 99), Orlando,FL, 1999.

[RGS03] R.Williams, C.P. Gomes, and B. Selman. Backdoors to typical case complexity. In *IJCAI*, pages 1173–1178, 2003.

[RHW05] M. Roberts, A.E. Howe, and L.D. Whitley. Modeling local search: A first step toward understanding hill-climbing search in oversubscribed scheduling. In J. Blythe, editor, *Poster Session of ICAPS 2005*, pages 33–36, 2005.

[RM99] H. Rudova and L. Matyska. Uniform framework for solving over-constrained and optimization problems. In *In CP'99 Post-Conference Workshop on Modeling and Solving Soft Constraints*, Alexandria, VA, 1999.

[SC93] S. Smith and C.C. Cheng. Slack-based Heuristics for Constraint Satisfaction Problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 139–144, Washington, DC, 1993. AAAI Press.

[Sch93] S.M. Schalck. Automating Satellite Range Scheduling. In *Masters Thesis*. Air Force Institute of Technology, 1993.

[SF97] K.J. Shaw and P.J. Fleming. Use of rules and preferences for schedule builders in genetic algorithms for production scheduling. *Proceedings of the AISB'97 Workshop on Evolutionary Computation. Lecture Notes in Computer Science*, 1305:237–250, 1997.

[SGS00] J. Singer, I.P. Gent, and A. Smaill. Backbone Fragility and the Local Search Cost Peak. In *Journal of Artificial Intelligence Research*, volume 12, pages 235–270, 2000.

[SGY+98] R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, and A. Fukunaga. Using ASPEN to automate EO-1 activity planning. In *Proceedings of the 1998 IEEE Aerospace Conference*, Aspen, CO, 1998.

[SMM+91] T. Starkweather, S. McDaniel, K. Mathias, L.D. Whitley, and C. Whitley. A Comparison of Genetic Sequencing Operators. In L. Booker and R. Belew, editors, *Proc. of the 4th Int'l. Conf. on GAs*, pages 69–76. Morgan Kaufmann, 1991.

[SP91] Gilbert Syswerda and Jeff Palmucci. The Application of Genetic Algorithms to Resource Scheduling. In L. Booker and R. Belew, editors, *Proc. of the 4th Int'l. Conf. on GAs*. Morgan Kaufmann, 1991.

[SP92] S.F. Smith and D.K. Pathak. Balancing antagonistic time and resource utilization constraints in over-subscribed scheduling problems. In *The Eighth IEEE Conference on Applications of Artificial Intelligence*, Monterey, CA, 1992.

[Spi99] F.C.R. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2:215–227, 1999.

[Sys91] Gilbert Syswerda. Schedule Optimization Using Genetic Algorithms. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*, chapter 21. Van Nostrand Reinhold, NY, 1991.

[Tai90] Eric D. Taillard. Some Efficient Heuristic Methods for the Flow Shop Sequencing Problem. *European Journal of Operational Research*, 47:65–74, 1990.

[Tai95] Eric D. Taillard. Comparision of Iterative Searches for the Quadratic Assignment Problem. *Location Science*, 3(2):87–105, 1995.

[TBS+00] M.D. Theys, N. Beck, H.J. Siegel, M. Jurczyk, and M. Tan. Scheduling heuristics for data requests in an oversubscribed network with priorities and deadlines. In *The 20th International Conference on Distributed Computing Systems*, pages 97–109, Taipei, Taiwan, 2000.

[UBKM93] N. S. Uckun, S. Bagchi, K. Kawamura, and Y. Miyabe. Managing genetic search in job shop scheduling. *IEEE Expert*, 8(5):15–24, 1993.

[Ver00] G. Verfaillie. Commentary on the paper entitled: Toward scheduling overconstrained remote-sensing satellites. In *Proceedings of the Second NASA International Workshop on Planning and Scheduling for Space*, pages 90–91, San Francisco, CA, 2000.

[VLS96] G. Verfaillie, M. Lemaître, and T. Schiex. Russian doll search for solving constraint optimization problems. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 181–187, Portland, OR, 1996.

[VW00] M. Vazquez and L.D. Whitley. A Comparison of Genetic Algorithms for the Static Job Shop Scheduling Problem. In Schoenauer, Deb, Rudolph, Lutton, Merelo, and Schwefel, editors, *Parallel Problem Solving from Nature, 6*, pages 303–312. Springer, 2000.

[Wat03] J.-P. Watson. *Empirical Modeling and Analysis of Local Search Algorithms for the Job-Shop Scheduling Problem.* PhD thesis, Colorado State University, Department of Computer Science, Fort Collins, CO, 2003.

[WBHW99] J.P. Watson, L. Barbulescu, A. E. Howe, and L. D. Whitley. Algorithm performance and problem structure for flow-shop scheduling. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-93)*, pages 688–695, Orlando, FL, 1999. AAAI Press / MIT Press.

[WBWH02] J.-P. Watson, L. Barbulescu, L. D. Whitley, and A.E. Howe. Contrasting Structured and Random Permutation Flow-Shop Scheduling Problems: Search Space Topology and Algorithm Performance. *INFORMS Journal on Computing*, 14(2), 2002.

[Whi89] L. D. Whitley. The GENITOR Algorithm and Selective Pressure: Why Rank Based Allocation of Reproductive Trials is Best. In J. D. Schaffer, editor, *Proc. of the 3rd Int'l. Conf. on GAs*, pages 116–121. Morgan Kaufmann, 1989.

[WRWH99] J. P. Watson, S. Rana, L.D. Whitley, and A. Howe. The Impact of Approximate Evaluation on the Performance of Search Algorithms for Warehouse Scheduling. *Journal of Scheduling*, 2(2):79–98, 1999.

[WS00] William J. Wolfe and Stephen E. Sorensen. Three Scheduling Algorithms Applied to the Earth Observing Systems Domain. In *Management Science*, volume 46(1), pages 148–168, 2000.

[WS04] X. Wang and S. Smith. Generating schedules to maximize quality. Technical Report CMU-RI-TR-04-25, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, April 2004.

[WSF89] L.D. Whitley, T. Starkweather, and D. Fuquay. Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator. In J. D. Schaffer, editor, *Proc. of the 3rd Int'l. Conf. on GAs*. Morgan Kaufmann, 1989.

[WSS91] L.D. Whitley, T. Starkweather, and D. Shaner. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*, chapter 22, pages 350–372. Van Nostrand Reinhold, 1991.

[WY95] L.D. Whitley and N.-W. Yoo. Modeling Permutation Encodings in Simple Genetic Algorithm. In L.D. Whitley and M. Vose, editors, *Foundations of Genetic Algorithms-3*. Morgan Kaufmann, 1995.

[ZDD94] M. Zweben, B. Daun, and M. Deale. Scheduling and rescheduling with iterative repair. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.