Three Scheduling Algorithms Applied to the Earth Observing Systems Domain

William J. Wolfe • Stephen E. Sorensen
Department of Computer Science and Engineering, University of Colorado at Denver, Denver, Colorado 80217
Hughes Information Technology Systems, Aurora, Colorado 80011
wwolfe@cse.cudenver.edu • sesoren@uswest.net

This paper describes three approaches to assigning tasks to earth observing satellites (EOS). A fast and simple priority dispatch method is described and shown to produce acceptable schedules most of the time. A look ahead algorithm is then introduced that outperforms the dispatcher by about 12% with only a small increase in run time. These algorithms set the stage for the introduction of a genetic algorithm that uses job permutations as the population. The genetic approach presented here is novel in that it uses two additional binary variables, one to allow the dispatcher to occasionally *skip* a job in the queue and another to allow the dispatcher to occasionally allocate the *worst* position to the job. These variables are included in the recombination step in a natural way. The resulting schedules improve on the look ahead by as much as 15% at times and 3% on average. We define and use the "window-constrained packing" problem to model the bare bones of the EOS scheduling problem.
(*Scheduling*; *Algorithms*; *Genetic*)

Introduction

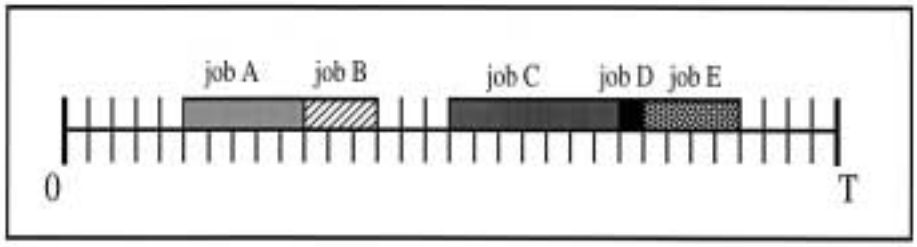
Scheduling problems are notoriously difficult. Foremost is the fact that exhaustive enumeration of the possible schedules is usually impractical. But further complications arise because of the different types of constraints that plague a real-world application (Fox 1994). This makes it difficult, if not impossible, to define an accurate objective function. The "weighting factors" that are often introduced create dubious measures as they attempt the proverbial combination of "apples" and "oranges." Rounding off the list of difficulties is the fact that the constraints are often changing in real time, resulting in a continuous stream of "rescheduling" events (see Morton and Pentico 1993, Zweben and Fox 1994). In this paper we emphasize the first difficulty: dealing with a large number of scheduling options (given a specific objective function and static constraints). To do this we introduce three algorithms that progress in complexity and accuracy: a *dispatch* algorithm, a *look ahead* algorithm, and a *genetic* algorithm. We use these as a platform for addressing the problem of multiple constraint types.

Our research begins with a particular scheduling problem that arises in NASA's Earth Observing Systems (EOS) domain (Wolfe 1995a, 1994, Short et al. 1995, MTPE/EOS Handbook 1995). In this case potentially hundreds of orbiting satellites are used to observe the earth and transmit data to the ground for further processing and dissemination. A "window of opportunity" arises when a satellite passes through positions where specific observational or scientific data can be collected. At that time the satellite needs a link to a relay satellite for transmission to the ground (very rarely do these satellites store data). This is often a bottleneck since there are very few relay satellites and they do not have enough capacity to guarantee a link (not unlike the telephone system on New Year's Eve). Additionally, although a satellite may be in the proper orbital position, the conditions may not be conducive to *quality data* due to weather, occlusion,

Management Science © 2000 INFORMS
Vol. 46, No. 1, January 2000 pp. 148–168

0025-1909/00/4601/0148$05.00
1526-5501 electronic ISSN

Figure 1 Discrete Time Steps; A Job Can Start Exactly Where Another Job Stops



sun angle, glare, calibration, or other problems. A single observation usually requires a sequence of several smaller steps to assure successful data collection: telemetry links, power, sensor calibration, pointing, tracking, etc. Each observation is usually part of a larger scientific experiment (e.g., rain forest depletion) and may have to be synchronized with other space-based observations and/or earth-based measurements. What emerges is a system that in a typical week must handle thousands of "requests for data" in support of hundreds of ongoing scientific/military/commercial observations. There are many more requests than can be serviced in any given week, and each request may have hundreds of opportunities (i.e., *over-subscribed* and *under-constrained*). The priority of the experiments and their observations is set by a NASA science committee (Asrar and Dozier 1994). This helps resolve satellite usage conflicts. Further complications arise due to emergency events, such as war, fire, flood, volcano, etc. The goal of the scheduling system, automated or manual, is to allocate tasks to satellite resources so that: 1) all constraints are satisfied; 2) preference is given to the higher priority requests; 3) the number of requests serviced is as large as possible; and 4) the collected data is high quality.

Several scheduling systems have been proposed over the years, the most relevant ones are described in Syswerda and Palmucci (1991), Uckun et al. (1993), Sadeh (1994), Muscettolla (1994), Johnson and Miller (1994), and Biefeld and Cooper (1989). Our approach most closely matches that of Syswerda and Palmucci (1991) and Uckun et al. (1993). Some of these references focus on problems unique to *factories* as opposed to *space* (Fox 1994, Sadeh 1994), and the rest cover a wide range of approaches, from the integration of planning and scheduling (Muscettola 1994) to neural modeling (Johnson and Miller 1994).

Window-Constrained Packing Problem (WCP)

To model the bare bones of this problem we define the "window-constrained packing problem" (WCP). The problem of assigning observations to satellites is very similar to packing jobs onto a time-line in a factory schedule, and/or assigning delivery times to truck routes. For simplicity assume that there is one resource (i.e., 1 satellite) and that the time span involved is the interval $[0, T]$. In this ideal model the time line is divided into discrete steps and an activity can stop at the same time as the start of another activity (see Figure 1).

There are n jobs $\{\text{job}_i | i = 1, \ldots, n\}$ and job_i must be done within its given *window of opportunity*: $[w_{0i}, w_{fi}]$ (for now we assume that there is only one window of opportunity for each job). A job is scheduled by assigning to it a continuous duration (d_i) (i.e., nonpreemptive) that satisfies: $d_{\text{min}i} \leq d_i \leq d_{\text{max}i}$, where $d_{\text{max}i}$ and $d_{\text{min}i}$ are given for each job. That is, each job has a minimum and maximum duration (see Figure 2).

In the EOS domain there are *preferences* for placements within the window of opportunity (e.g., as a function of aspect angle to the specific area of interest). A typical assumption is that the *middle* of the window is preferred over the edges, but the preferences can be much more complicated than that due to orbital dynamics and the nature of the specific target of interest. This preference is specified by a *suitability function*.[1] (See Figure 3.)

[1] Sometimes called a *score profile*, or *utility function*.

Figure 2 **Each Job Has a Priority and must Be Scheduled (i.e., Become an Activity) Within Its Window and Duration Constraints**

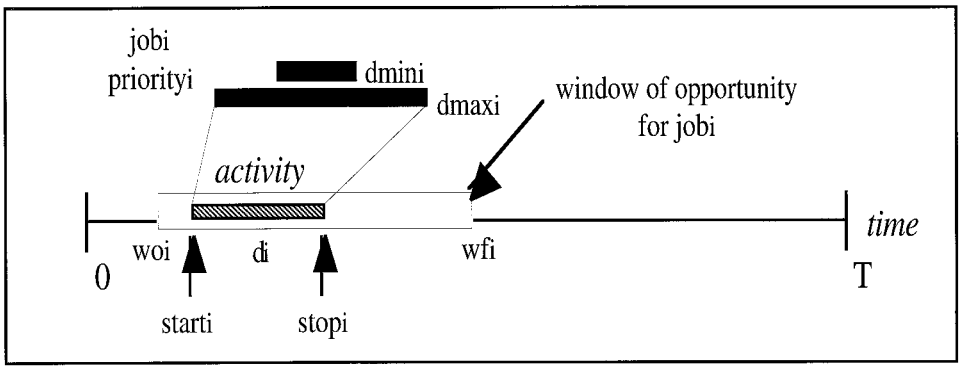
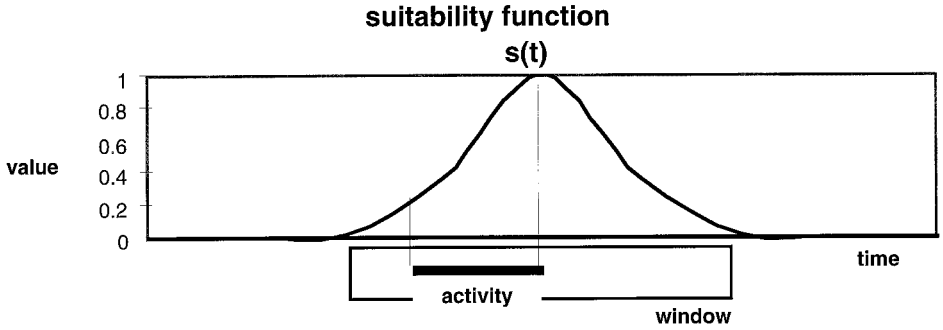


Figure 3 **A Job Is Defined by a Minimum (d_{min}) and a Maximum (d_{max}) Duration that must Be Scheduled Within a Window of Opportunity.**



Note. The suitability function expresses the preference for placement within the window (for the bell curve in this sketch the preference favors the center of the window).

Each job has a priority: $1 \leq p_i \leq 10$. (Note: In military parlance "priority 1" is the highest priority, but for our simulations we use larger numbers to indicate higher priority.) The priority measures the relative *worth* of the job, which in the EOS case measures the relative *scientific worth* of the observation. This must be consistent with the priorities defined by the NASA science board. When there is competition for the resource the *scheduler* must determine which jobs get on the schedule and specify the start and stop times that satisfy the window limits and duration constraints. Putting this all together we can formally define the window-constrained packing problem.

Window-Constrained Packing Problem (WCP)
Given n objects (requests), each having:

- a window of opportunity $[w_o, w_f]$;
- a suitability function with domain = $[w_o, w_f]$: $\{s(t) | t \in [w_o, w_f]\}$;
- a minimum length (d_{min}) and a maximum length (d_{max}); and
- a priority p.

$$\text{Maximize:} \quad Q = \sum_{i=1}^{n} x_i \cdot p_i \cdot \text{area}_i$$

$$\text{where:} \quad \text{area}_i = \sum_{t=\text{start}i}^{\text{stop}i} s_i(t)$$

subject to the constraints:

$$d_{\text{min}i} \leq d_i \leq d_{\text{max}i},$$

$$d_{\text{max}i} \leq w = w_{fi} - w_{oi},$$

$$w_{oi} \le \text{start}_i \le w_{fi} - d_i,$$

$$x_i \in \{0, 1\}$$

(note: $x_i = 1$ means that job_i is *on* the schedule),

$$[\text{start}_i, \text{stop}_i) \cap [\text{start}_j, \text{stop}_j) = \varnothing, \; i \neq j.$$

The WCP has the following *constraints*: job duration, window limits, exclusive use of the resource (i.e., capacity = 1), and nonpreemptive activities. An extended model would have multiple resources, multiple windows of opportunity for the same job, multiple *observations* for the same job, precedence constraints among the jobs/observations, etc. In what follows we stick with the simple WCP, but we have simulated more complex scenarios to verify the reasonableness of our extrapolations (see Wolfe 1995a).

The Evaluation Function (Q)

The evaluation function (Q) for the WCP is a priority-weighted sum of the "areas" under the suitability functions for each scheduled job. It expresses the preference for higher priority jobs, at longer durations, placed in the best spot in their windows of opportunity ("best spot" is defined by the suitability function). Notice that we are assuming that the *only* criteria is Q. That is, there is no score directly related to other possible criteria, such as:

$$\% \text{ Jobs} = \frac{\text{\# jobs scheduled}}{\text{\# jobs competing}} \cdot 100,$$

or

$$\% \text{ Utilization} = \frac{\text{time allocated}}{\text{total time}} \cdot 100.$$

There are several other criteria that might play a role in evaluating a schedule (e.g., set up time, etc.), but for now these are good examples of alternatives (or additions) to the Q defined above. For now we stick with our definition of Q, but later we consider the inclusion of % jobs. Notice that Q is generally larger when more jobs are scheduled but Q does not directly measure the number of jobs. There are many situations where a higher value of Q can be obtained with fewer jobs. For example, a single high priority job with a long duration may bump two or three low priority jobs. Later we introduce three scheduling *algorithms* and then evaluate their performance on the WCP. We randomly generate test problems, apply the algorithms, and compare the resulting scores (Q).

Complexity

One way to find an *optimal* solution is to exhaustively search the possibilities and compute the scores of all possible schedules. But this can be very time consuming since the average job can have many scheduling options. The number of scheduling options for a job with window width w, and duration limits $d_{\min}$, $d_{\max}$ is:

$$\begin{aligned} \#\text{ options} &= 1 + \sum_{q=d\min}^{d\max} [w - q + 1] \\ &= 1 + (d_{\max} - d_{\min} + 1) \\ &\quad \cdot (w + 1 - (d_{\min} + d_{\max})/2). \end{aligned}$$

The "1" occurring after the equals sign accounts for the option of *not* scheduling the job. A critical aspect of this problem is that we do not know ahead of time which jobs make the schedule and which ones do not. The largest number of options occurs when the window is the largest, the minimum time is the smallest, and the maximum duration is the largest.

Let $\mathbf{w} = w_{\max}$, and suppose $d_{\min} = 1$ and $d_{\max} = \mathbf{w}$:

$$\begin{aligned} &(\mathbf{w}) \cdot (\mathbf{w} + 1 - (w + 1)/2) + 1 \\ &\quad = (\mathbf{w}) \cdot (\mathbf{w} + 1)/2 + 1 \\ &\quad = (\mathbf{w})^2/2 + w/2 + 1 = O(\mathbf{w}^2). \end{aligned}$$

In the worst case, for n jobs, there are $\mathbf{w}^{2n}$ scheduling options. So, roughly speaking, each job has about $\mathbf{w}$ scheduling options, and there are about $\mathbf{w}^n$ possible schedules. Although there are several "special cases" it is clear that the general problem has an exponentially growing number of scheduling options to consider. Finally, notice that the classical Knapsack Problem (Garey and Johnson 1979) is a special case: Windows are all the length of the full time interval (i.e., the "knapsack" = $[0, T]$); each job had one duration: $d_{\max} = d_{\min}$; and suitability functions are flat (i.e., no preferences within a window).

Priority Dispatch (PD) and Look Ahead (LA) Algorithms

Having defined the WCP we now propose algorithms for creating good schedules (i.e., high Q). Analysis of the WCP reveals three critical pieces: 1) *Which* jobs make the schedule? 2) What *order* should they be in (order in *time*)? 3) What is the exact *start* and *stop* time for each scheduled job? (The last two questions are referred to as the "sequencing" and "timing" problems in traditional job shop scheduling.) First of all, if we knew ahead of time *which* of the contending jobs were going to make the schedule (i.e., a *subset* of the total set of competing jobs), finding the optimal solution would be significantly easier. If we also knew the *ordering* of those jobs (in time) on the resource the problem would be even easier (the only thing then left to do is to slip/slide/grow/shrink the jobs until we get the highest score we can get).[2]

We will discuss three algorithms in this paper: Priority Dispatch (PD), Look Ahead (LA), and Genetic (GA). It is insightful to note how each of the algorithms addresses these three pieces of the problem. The Priority Dispatch and the Look Ahead algorithms are *constructive* algorithms: they *build* a schedule from scratch. They use rules for *selecting* jobs and rules for *allocating* resources to the jobs. They can also be used to "add" to an existing schedule, usually treating the existing commitments as *hard* constraints. They are fast algorithms primarily because there is no backtracking. They are easy to understand because they use three simple phases: *Selection*, *Allocation*, and *Optimization*. The LA is similar to the PD but it uses a smarter allocation step. The LA algorithm "looks ahead" in the job queue (i.e., considers the unscheduled jobs) and tries to place the current job so as to cause the least conflict with the remaining jobs. We begin by describing the PD algorithm.

Priority Dispatch (PD)

This method begins with a list of unscheduled jobs and uses three phases: *Selection*, *Allocation*, and *Optimization*, to construct a schedule. There are many possible variations on these steps, and here we present a representative version.

Phase I: Selection. Rank the contending, unscheduled, jobs according to: rank(job_i) = priority_i. This is just one way to rank the jobs. Other job features could be used, such as *duration* or *slack*. *Slack* is a measure of the number of scheduling options available for a given job. For the WCP slack is the difference between $d_{\min}$ and the available, unscheduled, time in the window of opportunity. Jobs with high slack have a large number of ways to be scheduled, and conversely jobs with low slack have very few ways to be scheduled. Sorting by priority and then breaking the ties with slack (low slack is ranked higher) is the best sorting strategy that we have found. This is consistent with the established result that the WSPT algorithm (Morton and Pentico 1993) is optimal for certain machine sequencing problems (i.e., the "weighted-tardy" problem where all jobs must be late). It too uses a combination of priority and slack. In that case the slack is measured by: rank = priority/duration. In the simulations presented below, however, we use only the priority sort. This does not adversely effect the comparisons because all the algorithms would benefit equally from adding the slack variation.

Phase II: Allocation. Take the job at the top of the queue from phase 1 and *consider* scheduling the minimum duration ($d_{\min}$). If there is room in its window of opportunity (i.e., that does not conflict with previously scheduled jobs) consider the *best* spot, that is, where the area under the suitability function is largest. If there is no room for the job skip to the next job in the queue.[3] By choosing to place $d_{\min}$, as opposed to longer durations (such as $d_{\max}$), we get the job on the schedule while saving some room for subsequent jobs.[4] This mixes two extreme strategies: *greedy* (best spot) and *altruistic* (least duration). Continue through the job

[2] We are assuming that the suitability functions are simple (e.g., unimodal) and that a simple Hill Climbing approach would be very effective; if the functions are complex then this step is much more difficult.

[3] We do not consider *bumping* a scheduled task here. Such a variation can be easily added, but we have found this to be effective only when the job queue is poorly sorted.

[4] This favors getting *more jobs* on the schedule at the expense of some *worth*(Q). Getting more jobs on the schedule is not always mentioned as a criteria but it is usually desirable.

queue until all jobs have had one chance to be scheduled.[5]

Phase III: Optimization. After phases I and II are complete, examine the scheduled jobs and *expand* them into any unused space. Ranked by priority, *grow* each scheduled job to the left and right until one of the following happens: 1) it meets another activity; 2) it reaches its maximum duration ($d^i_{\max}$); or 3) it reaches the window limits. This is a form of Hill Climbing, and in fact, any hill climbing algorithm could be plugged in at this step, but the theme of the PD is to keep it simple.

The advantages of the PD approach are that it is simple and fast, and produces acceptable schedules most of the time. The simplicity supports the addition of many variations that may prove useful for a specific application. The PD produces optimal solutions in the simple cases (i.e., when there is little conflict between the jobs). The accuracy of the PD, however, can be poor. If the PD does not provide good solutions we recommend the Look Ahead method, which sacrifices some speed and simplicity for more accuracy.

PD's worst case run time is a function of the number of jobs (n) and the maximum window size ($w_{\max}$). The *Sort* phase requires $O(n \log n)$ time to sort the jobs. The *Allocation* phase, in the worst case, must compare each job to each of its possible window placements, giving it a run time of $O(nw_{\max})$. The *Optimization* phase must consider expanding each job to the size of the window, also giving a run time of $O(nw_{\max})$. Putting these together, we get a worst case run time of $O(n \log n) + O(nw_{\max})$. Therefore, if $w_{\max}$ is a constant as n grows, then the PD is $O(n \log n)$.

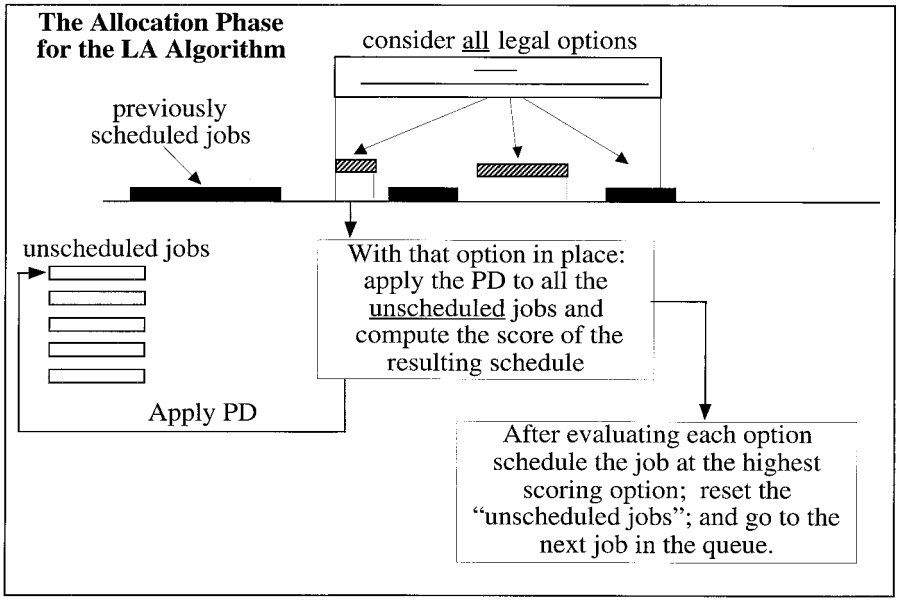
Look Ahead Algorithm (LA)

A powerful modification can be made to the PD method at the allocation step, when the job is placed at the *best* spot (as determined by area under the suitability function). This can be greatly improved by redefining *best* by *looking ahead* in the queue of unscheduled jobs. For each possible duration and position of the job apply the PD to the remaining jobs in the queue and compute Q for the resulting schedule, then unschedule all those jobs and move to the next legal duration and position. Schedule the job at the duration and position that scores the highest and move to the next job in the queue, with no backtracking. This allows the placement of a job to be sensitive to *downstream* jobs (see Figure 4.) The result is a dramatic improvement in schedule quality over the PD approach, but with an increase in run time. The LA algorithm is an excellent trade-off between speed and accuracy. The LA algorithm is near optimal for small job sets, and for larger job sets it outperforms the PD by as much as 20%, or 12% on average. The LA algorithm is difficult to beat, but the genetic algorithm described in the next section manages to improve on it by about 10% on some problems and 3% on average, but at the expense of significant increases in run times. Since the LA uses the PD ($O(nw_{\max})$) for its allocation step it is easy to see that the LA has a worst case run time of $O((nw_{\max})^2)$. Thus, if $w_{\max}$ is a constant as n grows, the LA has a worst case run time of $O(n^2)$.

We investigated several other algorithmic approaches, such as simulated annealing, genetic, and backtracking. We found that simulated annealing and backtracking algorithms provided improvements but with excessive run times. For example, we could not find a better way to control the backtracking (exhaustive search is prohibitive) than the small amount of backtracking in the LA method. That is, the combination of a good sort and an allocation based on an a look-ahead produced excellent results with minimal run times. Simulated annealing and backtracking algorithms may be useful in more complex situations, and possibly in cases where there is plenty of time available and where small improvements translate into large savings, but for our application we concluded that they were impractical. We did find, however, that the genetic approach provided a reasonable improvement for the amount of increased run time, as we discuss in the next sections, but it too suffers from long, and unpredictable, run times. It is our conclusion that the LA algorithm is the most powerful approach when run time, accuracy, and simplicity are considered.

[5] There are cases where we resort the queue after each allocation (e.g., when the allocation of a job changes the rankings of the jobs remaining in the queue, such as when *slack* is used in the sort).

Figure 4 **The Look Ahead Algorithm Scores Each Scheduling Option by Hypothetically Scheduling the Rest of the Queue (Using the PD) and Computing the Score of the Hypothetical Schedule**



Note. The option that scores the highest is the option that is actually scheduled and then the next job is considered. The look ahead algorithm considers all the legal scheduling options for each job, and is not restricted to considering the minimum duration as is the PD.

The algorithms presented in this paper are very similar to those described in Syswerda and Palmucci (1991). The major differences are in the addition of a look ahead method and in our expansion of the genetic parameters. Our look ahead method is similar to the look ahead method presented in Fox (1994). The look ahead algorithm in Fox "looks ahead" by considering different *orderings* of the job queue as allocation proceeds, with limited backtracking, whereas our look ahead never changes the order of the job queue but instead, carefully considers the "value" of each legal allocation by using the PD to "look ahead" at the potential conflicts that might arise with as yet unscheduled jobs. We have expanded on the genetic approach described in Syswerda and Palmucci (1991) by adding two more variables, giving the GA the opportunity to explore a wider search space.

Genetic Algorithm (GA)

In its purest form the genetic approach treats full schedules as members of a population and produces new members (children) by combining pieces of the higher scoring members. The process is allowed to evolve for many *generations* while the best schedules encountered are recorded. After evaluating many genetic variations we decided that using *raw* schedules as members of the population was impractical for a few reasons, the main one being that producing a new schedule from two parent schedules was unduly complicated. We found that it is easier to use job *permutations* as the population, with each permutation implicitly defining the schedule that we would get by feeding that job queue into the PD (or LA) algorithms (similar to the way "genes" are "expressed" by various enzymes). Using permutations has an additional benefit in that there are many applicable "crossover" techniques described in the literature (Whitley et al. 1989, Goldberg 1989). In particular we focused on the PMX crossover method for the simulations presented in this paper, but we explored other crossover strategies and mutation techniques with similar results. The downside of using permutations is that it restricts the search space to only those schedules that can be created by the PD (or LA). We compensate for this by

adding more genetic variables. The simulations presented in this paper use the GA-PD approach but we explored the use of the GA-LA as well. Our conclusion was that the GA-LA approach took too much run time for only a small increase in quality over the GA-PD. For the simulations below the initial population (size = # jobs) is chosen randomly. Each member of the population is *scored* (Q) based on how well the permutation did in constructing a schedule, then the population is *sorted* (highest Q on top). To create a "child" one parent is chosen uniformly from the *top half* of the population and another parent is chosen uniformly from the *whole* population. The parents are used in a *crossover* operation to create a child that is inserted at the bottom of the population (the lowest member is thrown out). Before the next generation begins the *mutation* operator has a chance to act: If any two members of the population have the same Q then one of them has two random entries swapped. The GA procedure is summarized as follows:

Initialize

- $m = N$ pop = population size; $n = N$ jobs = # jobs.
- Initialize a population of random permutations of integers $0, \ldots, n-1$: $\pi_0, \pi_1, \ldots, \pi_m$.
- Score (Q) each member of the population using PD.

Cycle

- **Sort** the population (highest Q on top), and record the current GA best score.
- **Recombination**: Pick two members of the population (in the simulations below one parent is chosen from the top half, and one from the whole population, with uniform distributions). Generate two random crossover points and apply the PMX crossover. Replace the last member of the population with the child.
- **Mutation:** If two members of the population have the same score (Q), swap two random entries in the permutation of one of them.

Augmented GA

After simulating the GA-PD we were not impressed with the results (it was doing approximately as good as the LA, but at the expense of about 100 times the run time). This is partly a tribute to the effectiveness of the LA algorithm, but we thought we could do better with a GA. After analyzing the permutation approach we decided that the GA needed a little more flexibility in exploring the scheduling options. One weakness is traceable to the sequential nature of the job queue (i.e., cannot "skip" a job). This led us to add another variable to the permutation representation: A "0" or a "1" representing whether the job is to be skipped or not when its turn comes up in the queue. We feed the job ordering to the PD as usual but jobs marked "0" are skipped. We include this variable in the recombination step as follows: If both parents have a "1" then the child is randomly assigned a "1" or "0," but with a strong bias toward "1." Similarly, if both parents have a "0" then the child is given a strong bias toward a "0." If the parents are ambiguous then we randomly assign a "1" or a "0" to the child (see Figure 5).

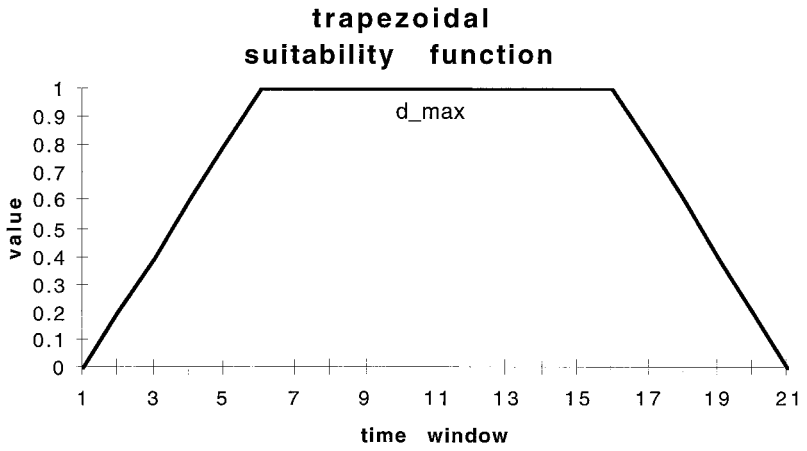
This variation in the GA works well, producing schedules that are noticeably better than the LA, but not by much. Since this approach was successful we added another binary variable to represent whether a job, if it is to be scheduled, should seek its *best* ("1") or *worst* ("0") spot in its window of opportunity. That is, skipping a job represents one end of the spectrum

Figure 5 **A "0" Indicates that a Job is to be Skipped, and a "1" Indicates that the Job Will Be Given a Chance to Make the Schedule**

	0	**1**
0	P(0) = 0.9 P(1) = 0.1	P(0) = 0.5 P(1) = 0.5
1	P(0) = 0.5 P(1) = 0.5	P(0) = 0.1 P(1) = 0.9

Note. A child "inherits" this parameter from its parents in a probabilistic way: If both parents have a "0" (or both "1") then the child gets a "0" (or "1") with a high probability (0.9 in the chart); if the parents have both a "0" and a "1" then the child gets a "0" or a "1" with equal probability.

Figure 6 **The Trapezoidal Suitability Function Penalizes Jobs that Are Placed Near Window Edges, but there Is No Penalty for Placing Job Near the Middle of the Window**



between its worst and best possible score. Stopping short of skipping the job, this new variable allows the job to be placed on the schedule but not at its best spot. This might allow for an optimal placement of a subsequent job without sacrificing (i.e., skipping) the current job. One of the reasons we chose this variable was that it was very easy to "reverse" the local evaluation function so the job would seek its "worst" as opposed to its "best" allocation. However, it is clear that a full spectrum of possibilities could be considered at each allocation. That is, it might be fruitful to allow placing the job at a randomly chosen "level" of score, from best to worst, but we did not implement this variation. The same probabilistic reasoning as above was used to assign values to the children at the recombination step. This last addition made the GA work extremely well, providing as much as 10% improvement over the LA at times and 3% on average. Thus, the GA produced excellent results but at the expense of speed.

Simulations

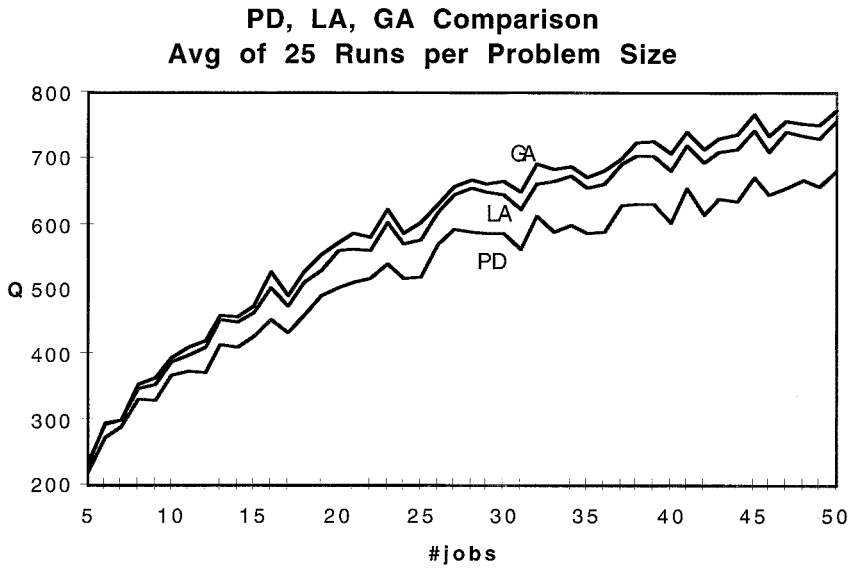
In this section we present the results of a specific simulation applying the three algorithms (PD, LA, GA) to randomly generated WCP problems. The parameters for the simulation are: number of jobs: $1 \leq n \leq 50$; time units to be scheduled: 100; maximum window size: 25; minimum job duration: 1; maximum job duration: 25; and priority: $1 \leq p \leq 10$.

For each run n random jobs are generated, giving each job uniformly random values for: $d_{\min}$, $d_{\max}$, w_0, w_f, and p. For this simulation we chose a *trapezoidal* suitability function. The *top* of the trapezoid is centered in the window of opportunity and has length equal to $d_{\max}$ for that job. This function penalizes job placements near window edges, but there is no penalty if the job duration is smaller than its maximum duration and placed *near* the center. The trapezoidal approach makes it possible for any job to get its "full score" ($p*d_{\max}$) when there are no conflicts (see Figure 6).

For each job set we applied the PD, LA, and GA, computed the scores of the resulting schedules, and averaged the results of 25 independent runs for each problem size (see Figure 7). For the GA we used a population size equal to the # jobs, and a number of generations equal to 300 times the number of jobs. The selection, recombination, and mutation strategies were described earlier in this paper, and we provide more details in the next section.

The results show that the GA averages about 12% better than the PD, a major achievement since the "guts" of the GA are based on the PD. This confirms that the GA benefits from the ability to skip jobs and to sometimes place jobs at their worst position in the window. The GA also reliably improves the LA by about 3%. This may seem small but the GA often beats the LA by a much larger margin and almost always gets a score at least equal to the LA (see Figure 8).

Figure 7 **Comparison of the Priority Dispatch (PD), the Look Ahead (LA), and the Genetic Algorithm (GA) (For Each Problem Size (# Jobs), the Results of 25 Independent Runs Were Averaged; the GA Algorithm Produces the Best Schedules)**



Keep in mind that the LA, in our experience, is a difficult algorithm to beat. It cannot be overlooked, however, that the GA takes about 200 times the run time of the LA (see Table 1). Thus, the LA is more likely to be practical for extremely large job sets.

The GA has the advantage that we can run it for longer times and possibly get better results. Figure 9 shows a case were the solution suddenly gets better after an extended time of no improvement. This can be frustrating, however, since there is no way to predict when such an event will occur.

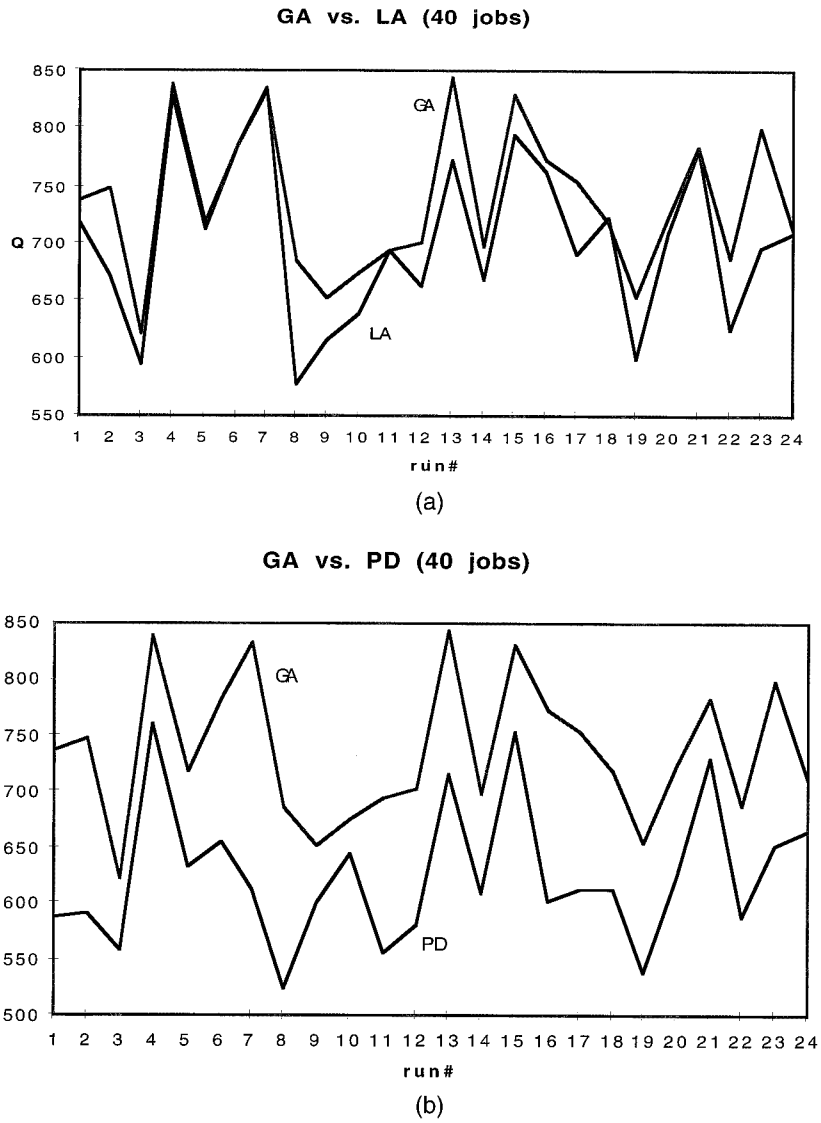
Additional GA Parameter Analysis

In this section we discuss the various choices we made in our GA implementation. In particular we show how we chose the population size, parent selection method, mutation operator, and crossover operator. We show that the larger the population the better the results but with diminishing returns beyond N pop = # jobs, and with increasing run time. We also show that the selection of the parents by our "top half" method is just as effective as the more popular "roulette wheel" approach for this problem. And we show that the mutation operator significantly increases the diversity of the population as the generations evolve. Finally, we used a PMX crossover operator, a fairly standard operator when sequences, as opposed to binary strings, are involved, but we do not compare this to other crossover methods. We did, however, do some testing that convinced us that alternate crossover methods (e.g., *order crossover* (Goldberg 1989)) were not significantly better than PMX for this application.

Population Size

We evaluated the GA's performance as a function of population size and found that although larger population sizes tend to produce better results, the improvements come at the expense of increased generations. Figure 10 shows a comparison using population sizes of # jobs and # job/2. The top of Figure 10 shows that when N pop = # jobs the GA produces slightly better scores, but the bottom shows that when N pop = # jobs/2 the GA requires significantly fewer generations. This is explained by the fact that smaller population size tends to concentrate on higher quality chromosomes relatively quickly in the evolution, and then tends to suffer from a mild case of premature convergence. In the simulations we presented in pre-

Figure 8 **The GA Results for 24 Runs of 40 Jobs per Run**



Note. The bottom figure shows that the GA achieves a reliable 12% better than the PD. The top figure shows that the GA usually matches the LA and sometimes beats it by 15%.

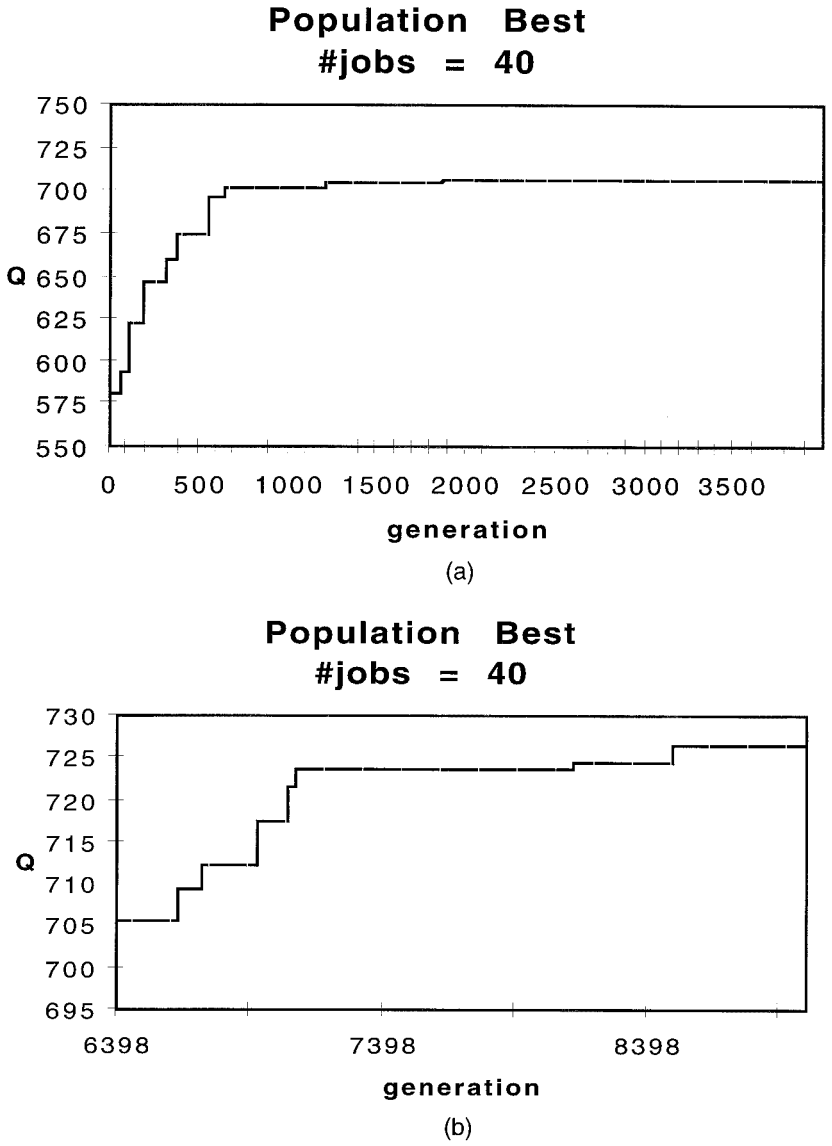
vious sections we set N pop = # jobs and therefore benefited with slightly higher scores, but the number of generations (N gen = 300$*N$ pop) was larger than might have been needed. Finally, setting N pop > # jobs did not show any significant improvement in score, while increasing the generations and run time, so we settled on N pop = # jobs for the algorithm comparisons.

Table 1 **Estimated Run Times for Each of the Algorithms.**

n	20	40	60	80	100
PD	4	8	12	25	30
LA	60	190	380	920	1,400
GA	18E3	84E3	9E5	19E5	39E5

Note. The numbers in the table are estimated seconds (×1,000), for one run of a random problem of size *n*.

Figure 9 **The Best *Q* in the GA Population at Each Generation**

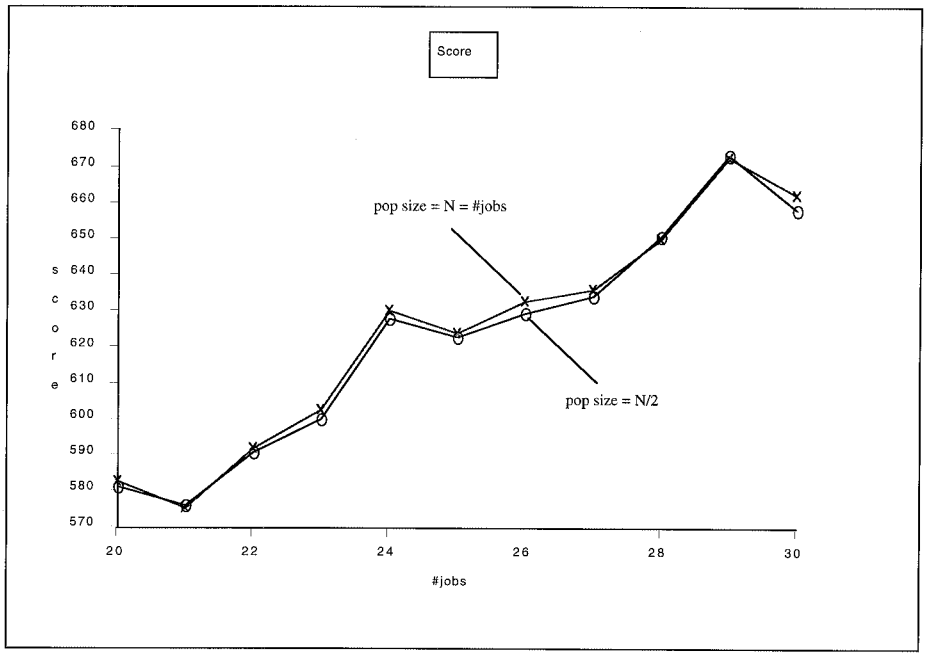


Note. In this particular run the population best was stable from about generation 2,000 to 6,000, and then around the 6,400th generation better solutions were suddenly introduced.

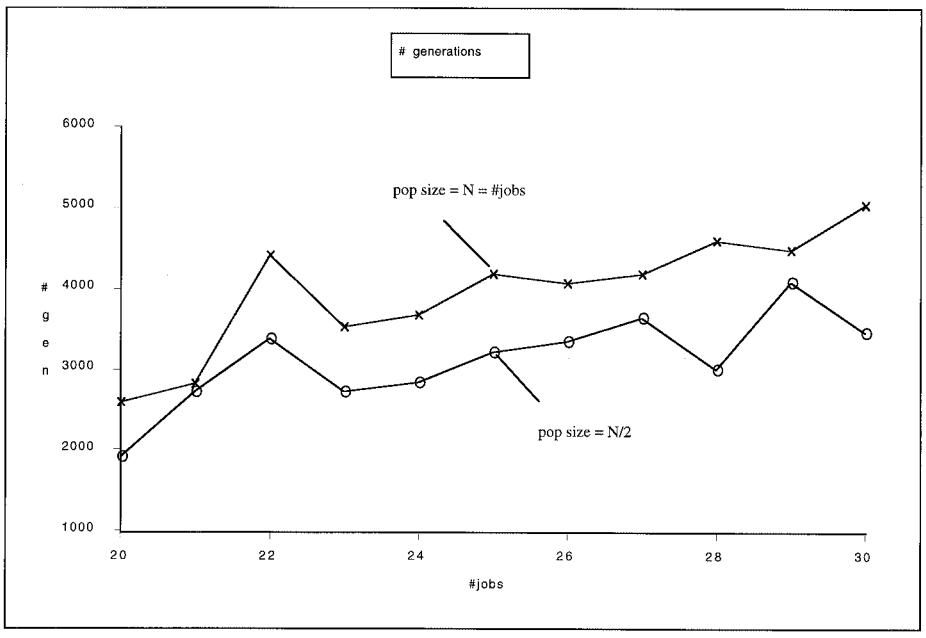
Parent Selection

In simulations we used a simple way to choose the parents for the recombination step. We choose the first parent uniformly randomly from the *top half* of the population, and the second parent uniformly randomly from the *whole* population. We refer to this method as the "top half" method. A more popular approach is the *roulette wheel* method (Goldberg 1989), in which case the population members are given a probability of selection in proportion to the percent of the total score that they represent. In comparison, we have found that the "top half" method is simpler and just as effective for our application. For example, Figure 11 shows a direct comparison of the two methods. The top of Figure 11 shows that the two methods produced approximately the same GA scores

Figure 10 **An Average of 25 Runs per Problem Size, Comparing Two Different Population Sizes: *N* pop = # jobs, and *N* pop = # jobs/2**



(a)



(b)

Note. The top figure shows that the larger population performs slightly better in terms of score, but the bottom figure shows that smaller population requires significantly fewer generations.

on an average of 25 runs for each problem size. The bottom half of Figure 11 shows that the number of generations was a little less stable, but the "top half" approach was slightly better (i.e., fewer generations). The similarity of the methods is explained, we feel, by the fact that both methods give a bias toward the higher scoring members of the population (while not ignoring the lower members), and after thousands of generations the distinctions in the particular probability distributions is washed out.

Mutations

The role of the mutation operator is to give the GA a wider space to explore, with an emphasis on random directions. Our mutation operator is invoked only when two members of the population have the same score. For this problem, after a few generations there is a relatively high probability of two members having the same score, and if nothing is done about this the GA suffers a severe case of premature convergence. The mutation operator creates a "pressure" to diversify the population. For example, Figure 12 shows a comparison of "with" and "without" mutations for the same 30 job problem. The top of the figure shows the range of scores in the population at each generation when mutations are used. The bottom half shows the range when there are no mutations. Without mutations the GA quickly converges on a homogeneous population, and the final score is significantly less than the best score with mutations.

Multiple Criteria

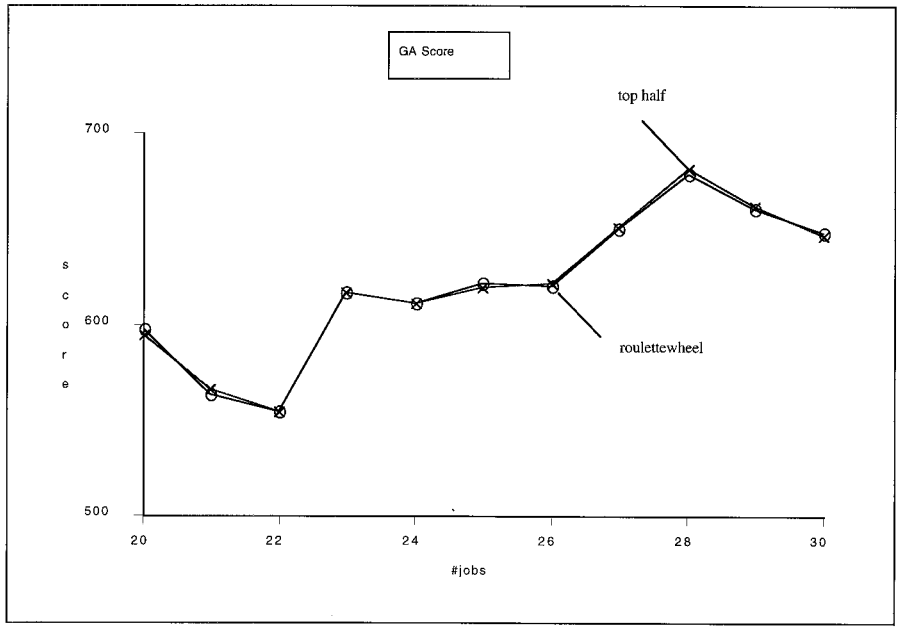
It is often the case that the objective function is a poor measure of quality because it does not take into account *all* of the possible criteria. Q, as defined earlier, encourages longer, centered, high priority jobs. In particular it does not directly measure the *total number* of jobs that make the schedule. In the EOS domain it is often expressed that the more users involved the better, even if they are not fully satisfied (e.g., less data). This would keep the lower priority experiments "alive," while the higher priority experiments would still get more preferences satisfied. It is interesting to note that the PD approach, because of its use of $d_{\min}$ in the allocation step, has a tendency to get more jobs on the schedule than the other algorithms (see Figure 13).

When new criteria are being considered (e.g., % jobs) it would then make sense to add an additional term to the evaluation function, such as: $Q = a_1Q_1 + a_2Q_2$; where Q_1 is the original Q, and Q_2 is a measure of % jobs. We performed experiments on just such cases and the results are in Wolfe (1995b). For the purposes of this paper we simply state that several factors work against the effectiveness of such an approach. First, the relative strength of the weighting factors, a_1 and a_2, is not easy to establish. Second, the different criteria can be highly correlated, making it difficult to favor one exclusively. Third, Q_1 and Q_2 are on different *scales* and need to be normalized so that the weighting factors make intuitive sense (this is usually a nonlinear process that makes it difficult to interpret the weighting factors). Fourth, it is already difficult to sort the job queue to favor a particular criteria, and this becomes even more complex when multiple criteria are involved. For practical reasons it is best to provide a score for *each* criteria and let the user interpret the combinations.

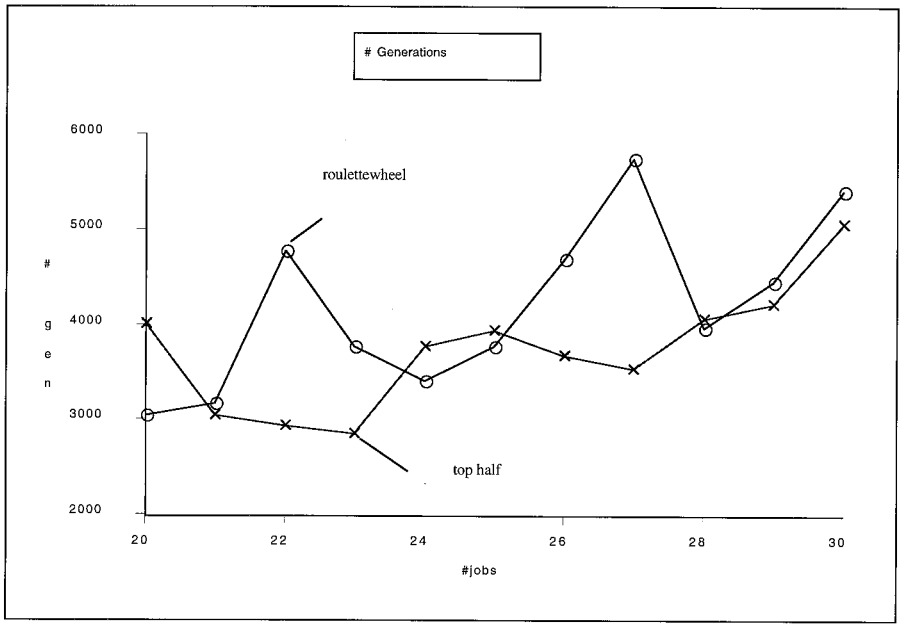
Further Analysis of the EOS Domain

The algorithms described above were tested on a "bare bones" representation of the EOS domain, using randomly generated job sets. In order to apply these algorithms to the *real* problem several additional factors must be taken into consideration. First of all, the EOS domain is complicated by the coordination of *long* and *short* term scheduling events. A long term schedule is first created, projecting out a week or more, and continually adjusted as the day of operations approaches. This process is often called "progressive refinement" (Biefeld and Cooper 1989), and is a common feature in almost all complex scheduling systems. For example, the Tracking Data Relay Satellite System (TDRSS) provides the communications link, through geostationary satellites, for several "users," including: Space Shuttle, Hubble Space Telescope, Landsat, Gamma Ray Observatory, and EOS. TDRSS schedules (managed by the NASA Goddard Spaceflight Center) are made at 24 hour increments at

Figure 11 An Average of 25 Runs per Problem Size, Comparing the Roulette Wheel and "Top Half" Methods



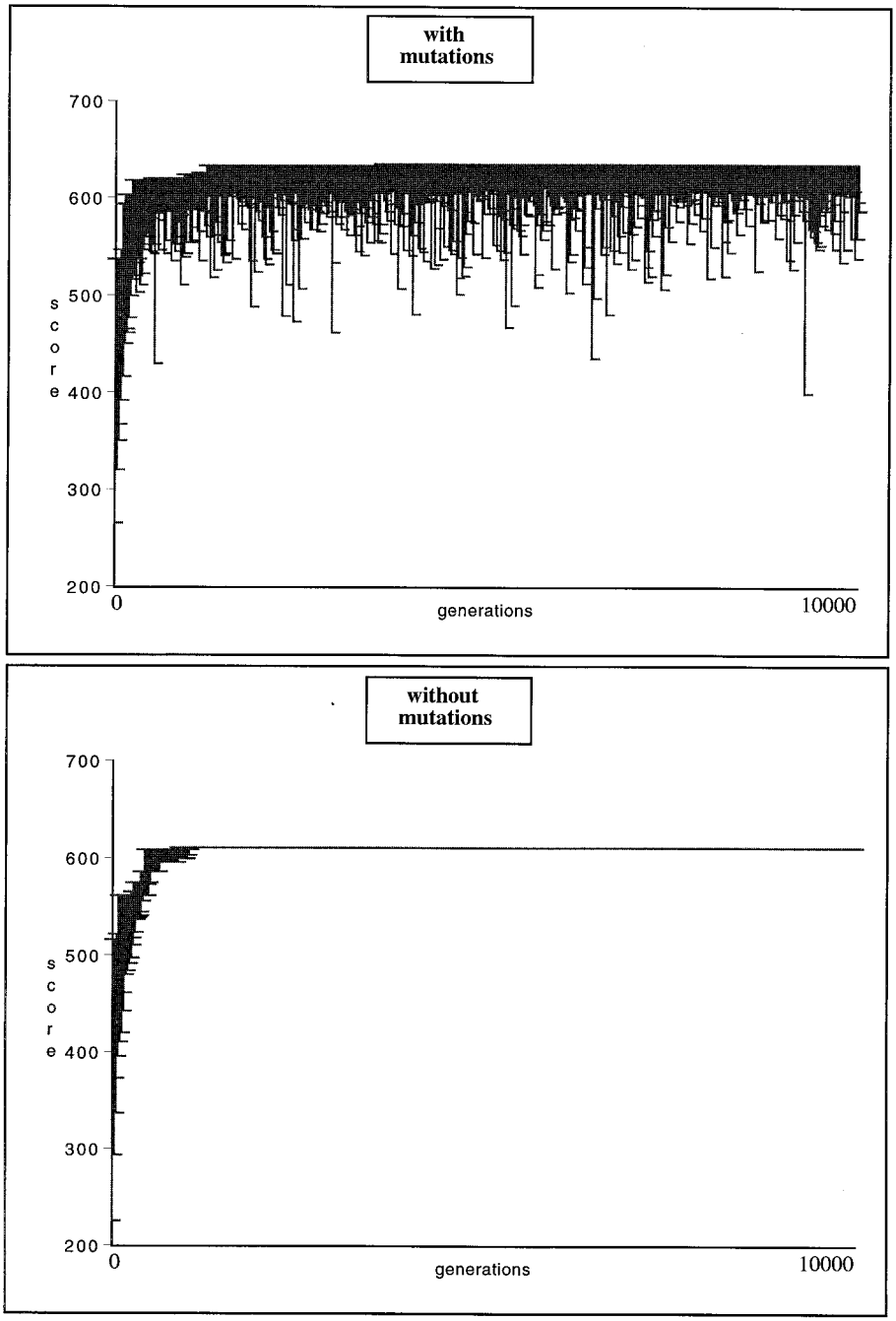
(a)



(b)

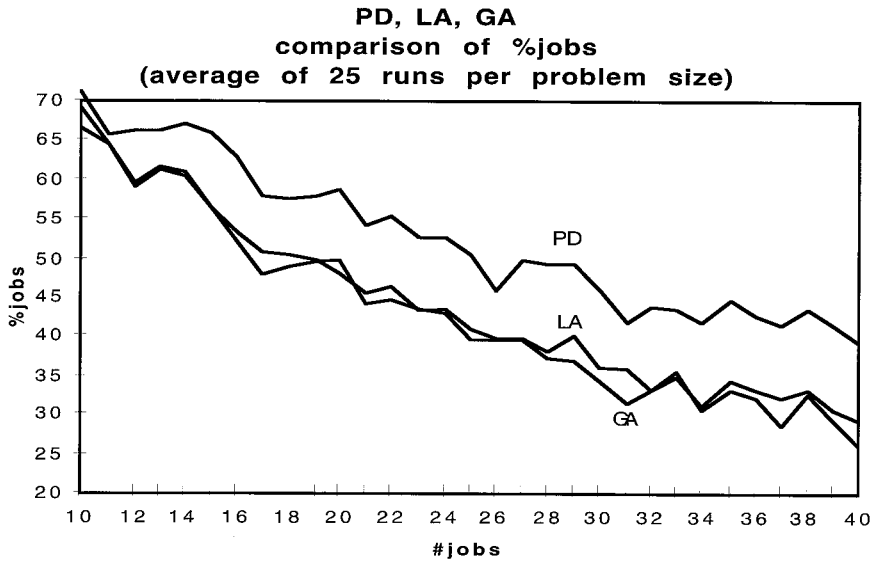
Note. The top figure shows that the GA scores produced by both are almost identical, while the bottom figure shows that the "top half" method gets its results in slightly fewer generations in most cases.

Figure 12 A Comparison of "With" and "Without" Mutations for a 30-Job Problem



Note. The figures show the ranges of scores in the population at each generation for the two cases. The mutations increase the diversity of the population, without which the GA suffers from premature convergence.

Figure 13 **The PD Algorithm Schedules a Higher Percentage of Jobs Than the LA or GA**



a 1 minute time granularity. There is a "rolling" 7 day schedule that goes through the following phases: *Outday* (more than 1 week ahead); *Stabilization* (7–2 days ahead); *Near Real Time* (2–1 days ahead); *Real Time* (the day of operations). It is expected that EOS will continue to conform to the TDRSS approach. The initial EOS schedule (i.e., the "out-day" schedule) accounts for only the most critical constraints, such as the availability of the necessary sensors and communication links, and leaves the finer details, such as sensor and antenna pointing, for later (very similar to locking in your airline fare long before you decide on hotel or taxi accommodations). A basic strategy is typically applied to the set of competing jobs: high priority, complex, jobs are scheduled first, and low priority, simple, jobs are scheduled last. In the initial scheduling phase a job is usually given its *minimum* resource requirements (minimum power, observation time, etc.). These constraints may be *relaxed* in subsequent refinements if there is not competition for a resource from a higher priority user.[6]

During the refinement phase several things may happen: high priority requests may suddenly arrive, jobs may be canceled, a satellite may malfunction, special earth events may occur (e.g., volcano eruption, flood, etc.). For example, changes in Space Shuttle plans can affect the EOS schedule. Due to manned flight, Space Shuttle is TDRSS' highest priority, and takes up a continuous link to the ground. If the Space Shuttle, for some reason, does not launch on schedule, or stays up longer than planned, TDRSS' schedule is dramatically affected. Such events may cause EOS rescheduling right up to the day of operation. In addition, since satellite orbital position and attitude are not accurately known until hours before operations, it is difficult to precisely predict a window of opportunity far ahead of time. Thus, as the day of operations approaches the tasks and their associated constraints sharpen and various adjustments are made. Although there may be many changes to the schedule during the refinement phase, caution must be exercised, however, to avoid what might be called "nervous scheduling." It is unwise to be changing too many things from day to day, for fear of confusing the humans that review the schedules. These scheduling experts tend to be balancing subtle constraints, often estimating the degree of spacecraft *feasibility* and *safety* involved in a complex sequence of operations. Such experts cannot tolerate complete rewrites of the sched-

[6] Our scheduling algorithms reflected this strategy by allocating only the minimum duration ($d_{\min}$) to a task, and then "growing" the scheduled tasks after all tasks had a chance at making the schedule.

ule from day to day. In practice, a certain amount of "stability" is enforced, even if the schedule may be deemed suboptimal from the standpoint of certain objective functions.

The primary purpose of EOS is the study of earth science. This requires it to support scientific experiments involving periodic observations, with periods ranging from hours to years. Additionally, a single observation may require simultaneous earth-based measurements (i.e., in situ). For example, it may be necessary to measure chemical content at a specific ocean site at the same time that a satellite observes it. In some cases there is a requirement for several coordinated events on the land, and in the air and sea, as a satellite observation is made (possibly involving multiple satellites). When NASA schedules an experiment (out-day) the responsible scientist makes arrangements for all supporting activities. If NASA cancels the task during the refinement phase (e.g., in favor of a higher priority experiment) then the supporting investment is lost. NASA accounts for this with a concept called "priority aging," meaning that the priority of a task is artificially increased the longer it remains on the out-day schedule. That is, it gets harder to bump a task the longer it sits on the schedule. In effect, the cost of modifications forces the schedule to stabilize over time. With these complexities in mind it is clear that our algorithms can only be applied to a limited, static, part of the problem. For example, at any time in the refinement process the competing requests, current constraints, and relaxation levels, can be "frozen" and one of the algorithms applied to produce a momentary solution. To do this, the bare bones model must be extended to include multiple windows of opportunity, multiple resources, and additional constraints (e.g., power level, precedence). These extensions will, at the least, cause run times to rise, making the GA an unlikely candidate (unless sufficient run time is available, such as might be the case for outday schedules). We have performed limited tests with extended models, and currently have no reason to doubt the final conclusions: the PD is fast but not very accurate; the LA is a little slower but much more accurate; and the GA can outperform the LA but at the expense of extensive run times.

Our analysis is based on uniformly randomly generated jobs, and although we feel that this is a reasonable representation of the real situation (since there are usually hundreds of competing jobs for most sections of the time line), special distributions are worth further study. In particular, periodic distributions are certainly of interest due to the 90 minute (on average) orbital time of a typical low earth orbit (LEO) satellite. Various bottlenecks are sure to arise when a special event occurs, such as flood or fire, attracting the attention of otherwise uninterested scientists. The adaptation of our algorithms to these situations will rely on isolating a small part of the problem, in which case the competition may then appear random. The chosen algorithm would act like a "calculator" that assists a scheduling expert. For example, the expert would decide how much of the schedule is open to rescheduling (i.e., within the stability constraints), and assign the appropriate priorities to the tasks (considering priority aging). The expert would also have to decide how much weight to give to each of the possible criteria (see the section above on "Multiple Criteria"). A graphical user interface would be almost certainly required to assist the expert in making these assignments. The expert would then run one of the algorithms and evaluate the schedule that was produced and decide if it should be declared the "current" schedule. The following day, new information would be integrated to define new constraints, competing jobs, relaxation levels, etc., and a new schedule created by the algorithm.

Conclusions

We have given an overview of the EOS scheduling problem and introduced a "bare bones" model, the "window-constrained packing problem." This model can be used for several other problems, especially routing problems. The Priority Dispatch algorithm is fast and simple and creates acceptable schedules most of the time. The Look Ahead algorithm runs a little slower but is still relatively simple and creates excellent schedules. The Genetic Algorithm is a lot slower but appears to create near-optimal schedules. Our genetic approach is novel in its use of parameters to allow the dispatcher the flexibility to *skip* jobs, and to allocate them to their *worst* spot. These parameters allow the GA to explore a wider search space. The PD

has the extra advantage of producing schedules with *more jobs*, generally at the expense of shorter durations for higher priority jobs. We have simulated many other scheduling problems and have drawn the general conclusion that what we see in the WCP is what will be seen in more complex problems. That is, a *dispatcher* can usually be defined and it will be fast/simple but produce poor schedules at times. A *look ahead* algorithm (no backtracking) can usually be defined that will run a little slower but give significant improvements in quality. And, a *genetic* algorithm can usually be introduced that takes a significantly longer run time, but provides significant improvements on some cases. The choice between these algorithms for a particular application would be a function of the answers to the following questions. 1) How important is a small percentage improvement in schedule quality? In many real-world applications the constraints are changing so rapidly that any current schedule is soon to be invalid so it makes little sense to be refining it to a high degree of (imagined) accuracy. 2) How important is it for the users or operators to understand the logic of the algorithm? In many cases it is advantageous to keep the algorithm simple so that users can modify or interpret the results effectively, and this is especially true in domains where the constraints are rapidly changing. Furthermore, the changes make it difficult for an accurate objective function to be clearly defined. These conditions favor dispatch algorithms because they are fast, simple, and reasonably accurate. 3) How long can you wait for a solution? In domains where the constraints are relatively stable, and where small schedule improvements translate into great savings, it is worth waiting for the better solutions produced by genetic algorithms.

References

Asrar, G., J. Dozier. 1994. EOS science strategy for the earth observing system. American Institute of Physics, available at ⟨http://eospso.gsfc.nasa.gov/sci_strategy/Contents.html⟩

Biefeld, E., L. Cooper. 1989. Scheduling with chronology-directed search. AIAA Computers in Aerospace VII, October, Monterey, CA, pp. 3–5.

Fox, M. 1994. ISIS: A retrospective. M. Zweben, M. Fox, eds. *Intelligent Scheduling*, Morgan Kaufmann Publishers, San Francisco, CA.

Garey, M., D. Johnson. 1979. *Computers and Intractability*. W. H. Freeman and Company, New York.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, New York.

Johnson, M. D., G. E. Miller. 1994. Spike: intelligent scheduling of hubble space telescope observations. M. Zweben, M. Fox, eds. *Intelligent Scheduling*, Morgan Kaufmann Publishers, San Francisco, CA.

Morton, T. E., D. W. Pentico. 1993. *Heuristic Scheduling Systems*. John Wiley and Sons, New York.

MTPE/EOS Handbook. 1995. NASA publication, available at: ⟨http://eospso.gsfc.nasa.gov/eos_homepage/misc_html/ref_book.html⟩

Muscettola, N. 1994. Integrating planning and scheduling. M. Zweben, M. Fox, eds. *Intelligent Scheduling*, Morgan Kaufmann Publishers, San Francisco, CA.

Sadeh, N. 1994. Micro-opportunistic scheduling: the micro-boss factory scheduler. M. Zweben, M. Fox, eds. *Intelligent Scheduling*, Morgan Kaufmann Publishers, San Francisco, CA.

Short, N. et al. 1995. Notes for the workshop for planning and scheduling of earth science data processing. NASA Goddard SFC Conference on Space Applications of AI, May 8–9, .

Syswerda, G., J. Palmucci. 1991. The application of genetic algorithms to resource scheduling. Fourth International Conference on Genetic Algorithms.

Uckun, S., S. Bagchi, K. Kawamura, Y. Miyabe. 1993. Managing genetic search in job shop scheduling. *IEEE Expert* October.

Whitley, D., T. Starkweather, D. Fuquay. 1989. Scheduling problems and traveling salesman: the genetic edge recombination operator. ICGA'89, San Mateo, CA.

Wolfe, W. J. 1994. Spacebased resource planning—Phase I. Summary Report for Hughes Information Technology Systems research grant #SG-258799SJP, January, Aurora, CO.

——. 1995a. Spacebased resource planning—Phase II. Summary Report for Hughes Information Technology Systems research grant #SG-258799SJP, January, Aurora, CO.

——. 1995b. Scheduling with distribution criteria. Report for Hughes Information Technology Systems research grant #SG-258799SJP, November, Aurora, CO.

Zweben, M., M. Fox. 1994. *Intelligent Scheduling*. Morgan Kaufmann Publishers, San Francisco, CA.

Accepted by Thomas M. Liebling; received January 1998. *This paper has been with the authors* 3 *months for* 2 *revisions.*