Scheduling Multi-Resource Satellites using Genetic Algorithms and Permutation Based Representations

O. Quevedo De Carvalho and D. Whitley
Colorado State University
Fort Collins, Colorado, USA

V. Shetty and P. Jampathom and M. Roberts
Naval Research Lab
Washington, D.C., USA

ABSTRACT

The U.S. Navy currently deploys Genetic Algorithms to schedule multi-resource satellites. We document this real-world application and also introduce a new synthetic test problem generator. A permutation is used as the representation. A greedy scheduler then converts the permutation into a schedule which can be displayed as a Gantt chart. Surprisingly, there have been few careful comparisons of standard generational Genetic Algorithms and Steady State Genetic Algorithms for these types of problems. In addition, this paper compares different crossover operators for the multi-resource satellite scheduling problem. Finally, we look at two ways of mapping the permutation to a schedule in the form of a Gantt chart. One method gives priority to reducing conflicts, while the other gives priority to reducing overlaps of conflicting tasks. This can produce very different results, even when the evaluation function stays exactly the same.

CCS CONCEPTS

• **Mathematics of computing** → **Graph algorithms**; *Combinatorial algorithms*; • **Theory of computation** → **Evolutionary algorithms**.

KEYWORDS

Scheduling, Steady State Genetic Algorithm, Order Crossover

ACM Reference Format:
O. Quevedo De Carvalho and D. Whitley and V. Shetty and P. Jampathom and M. Roberts. 2023. Scheduling Multi-Resource Satellites using Genetic Algorithms and Permutation Based Representations . In *Genetic and Evolutionary Computation Conference (GECCO '23), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583131.3590387

1 INTRODUCTION

Resource scheduling is perhaps one of the most commercially successful applications of Genetic Algorithms. In the 1990s the company Optimax, founded by Gil Syswerda, developed and sold permutation-based resource scheduling solutions using Genetic Algorithms [14, 15]. The key to the success of this technology was the permutation-based indirect representation of schedules and specialized recombination operators to generate permutations which are then mapped to schedules.

GECCO '23, July 15–19, 2023, Lisbon, Portugal

ACM ISBN 979-8-4007-0119-1/23/07.
https://doi.org/10.1145/3583131.3590387

There have been continued successful applications of this technology. One specific application area where Genetic Algorithms continue to be deployed is satellite scheduling. Early work by Parish [12] applied Syswerda's methods to the Air Force satellite contact scheduling problem. Parish maximized the number of scheduled tasks. Barbulescu et al. [1] extended this research and changed the evaluation function to minimize schedule time conflicts, which ensures that all tasks are eventually introduced into a viable schedule.

In this paper, we present a new real-world resource scheduling application using Genetic Algorithms. The U.S. Navy currently deploys Genetic Algorithms to schedule multi-resource satellites. In some cases, only one resource can be utilized on a satellite. But under favorable circumstances, two or more resources available on a single satellite can be used in parallel. This problem has similarities to scheduling the U.S. Air Force Satellite Control Network (AFSCN), but it is also a different problem. Scheduling multi-resource satellites also appears to be a more highly constrained problem.

Solutions are represented as permutations of the task IDs. This permutation acts as a priority queue. Each task also has a list of alternative satellite resources (as well as time windows) where that task could be placed. This permutation-based priority queue is converted into a schedule using a greedy scheduler. The tasks in the permutation are scheduled at the first requested location that is still available. This approach hides the details of the scheduling domain from the Genetic Algorithm, which in turn allows the Genetic Algorithm to be used as a general-purpose resource scheduling tool. The completeness of this approach relies on proofs that the number of schedules is smaller than the number of permutations and that every schedule is reachable from at least one permutation.

We compare a generational standard Genetic Algorithm (SGA) to a Steady State Genetic Algorithm (SSGA). Most applications for resource scheduling problems used a Steady State Genetic Algorithm. We find that either the standard GA or the Steady State GA might be best depending on how long the algorithm is allowed to run and depending on the population size.

This paper revisits the question of what recombination operator should be used for resource scheduling using a permutation-based representation. Our empirical results are consistent with previous studies [13] which suggest that Syswerda's "Order Crossover" [15] consistently produces significantly better results compared to Goldberg's "Partially Matched Crossover" (PMX) [10]. This is not surprising since PMX was designed as an operator for the Traveling Salesman Problem. Nevertheless, PMX has continued to be popular as a general-purpose permutation crossover operator.

Finally, we also vary the way that the Greedy Scheduler converts a permutation into a schedule. Our Genetic Algorithms all minimize "overlaps" but we show that different results are obtained depending on which Greedy Scheduler is used. The Greedy Scheduler can be

configured to strictly use the order in the permutation as a priority queue, and if a task cannot be scheduled without conflict, then it is placed in the schedule so as to minimize overlaps with other tasks. This approach is helpful to human schedulers. Computer-based schedulers generally do not operate autonomously; a human will review and perhaps change the schedule. A human might "trim" the time allocated to tasks in order to fit more tasks into the schedule. For example, if two tasks have a two-minute overlap in the schedule, it might be desirable to schedule both tasks by reducing the time allocated to each task by 1 minute instead of rejecting one of them. Minimizing the total overlaps in a schedule can give a human scheduler better information about how to create a feasible schedule when conflicts are present.

An alternative approach is to schedule as many tasks as possible without conflict, skipping tasks that result in conflict. Then at the end, tasks with conflicts are scheduled (again using the permutation as a priority queue) so as to minimize total overlaps between conflicting tasks. We look at how these two different approaches produce very different results, even while using exactly the same objective function. A key result of this paper is to show that the greedy schedule builder can have an extremely strong influence on the types of schedules that are generated. Practitioners who use Genetic Algorithm to build schedulers with evaluation functions that use complex indirect mappings **cannot** automatically assume that they are generating near optimal solutions relative to the evaluation function. How an indirect mapping is constructed can greatly bias optimization results for better or for worse.

2 BACKGROUND AND PROBLEM

Figure 1 represents a set of three resource task requests. In this example we assume all three tasks are competing for time on the same resource. The same task request may be called a "Task" or a "Request." Each task request indicates 1) a satellite-based resource where that task might be carried out, 2) a task duration, 3) the earliest start time (EST), and 4) the latest finish time (LFT).

The earliest start time and the latest finish time generally depend on the orbit of the satellite and when the satellite is passing over a particular location on earth. The earliest start time and latest finish time determine "a window of visibility" during which the task request needs to be executed. A task might fill the entire window, or the window might be larger than the task duration (see Figure 1.)

Some satellites rotate around the earth in approximately 90 minutes at a relatively constant speed. Other satellites rotate around the earth much less frequently, and they might have elliptical orbits. Assume that we need to use a satellite-based resource to collect data while passing over Lisbon, Portugal. There might be a satellite that passes over Lisbon at 10 am today with a 90-minute rotation around the earth. There might be a second satellite that passes over Lisbon at 11 am today, but with a 4-hour rotation. If the request requires a 10-minute contact, there might be a 15-minute window on the satellite with the fast rotation, but there might be a 40-minute window on the satellite with the slower rotation. Longer windows of visibility usually provide more flexibility in terms of where a task can be placed in a schedule. A single task request will often indicate multiple resources on different satellites which could carry out that request at different times. We will assume that these

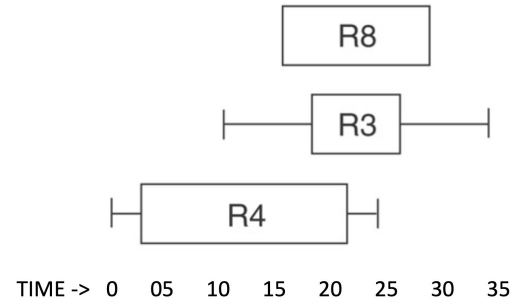

Figure 1: This illustration shows three tasks that are requesting time on the same resource. Request 8 has a time window with no flexibility: Task 8 must start at the earliest possible start time and must end at the latest finish time. On the other hand, Request 3 and Request 4 have a duration such that Task 3 and Task 4 can be moved in time, as indicated by the whisker plots. Both Task 3 and Task 4 can be scheduled without conflict on this resource, but Task 8 conflicts with both Task 3 and Task 4.

alternative options occur during a single day, but in some domains, the alternative time slots might span a week (for example in the domain of personnel roster scheduling), or some other interval.

For satellite resource scheduling (as well as for radar task scheduling for space based tracking), the solution quality can also depend on where a task is placed in the window of visibility. Under ideal conditions the task is always placed in the center of the window where the target to satellite angle is 90 degrees (i.e., nadir). This can minimize atmospheric interference, for example. However, this often does not yield a compact schedule; in general, this placement does not exploit the scheduling flexibility provided by the window of visibility. Typically more tasks can be scheduled by sliding some tasks just off of 90 degrees to fit in another task.

We can think of the time window as also having an associated Gaussian distribution reflecting the desirability of where the task is scheduled in the window. For earth observing systems this approximates the quality of the observation; for other systems (e.g. radar based systems) this Gaussian distribution might reflect the likelihood of a successful (or unsuccessful) observation. Wolfe and Sorensen [18] used a trapezoidal distribution to estimate this underlying Gaussian distribution for earth observing systems: the middle of a time window is uniformly good, but the edges of the time window are less desirable. In addition, Wolfe and Sorensen [18] use a permutation based representation to schedule Earth Observing Systems, but also added a binary vector to the representation which indicates if a task should be placed in the "best" location (i.e., nadir) or the "worst" location. The current NAVY deployment system scores a solution based on where a task is scheduled in the visibility window. In the scheduler described in the next section, we place a task in the earliest time possible in the time window; after the schedule is complete it is possible to move tasks toward the nadir position if this does not cause a conflict.

An example of a set of requests is displayed in Figure 2 as a Gantt chart. This figure shows the requests made to 10 different resources,

with all of the multiple alternatives and conflicting requests shown. Thus, one request appears multiple times.

An example of a final schedule is shown in Figure 3 as a Gantt chart. There are still some conflicts, but most task requests are scheduled without conflict.

One can prove that optimal schedules can be generated for low-altitude satellites in Polynomial time under certain conditions. The proof assumes that only one task can be scheduled in any given window of visibility for every satellite resource. This can be a reasonable assumption for satellites that consistently have small windows of visibility (e.g. 15 minutes). The proof poses the low-altitude satellite scheduling problem as an Activity Selection problem; a greedy algorithm that exactly solves this problem in Polynomial time is given by Cormen et al. [5]. Scheduling low-altitude satellites continues to have polynomial time complexity even if the tasks may be serviced by one of several satellite resources. The resulting multi-resource Activity Scheduling problem also corresponds to a variant of the interval scheduling problem, which is equivalent to the k-colorability of an interval graph [4].

When high-orbit satellites are introduced, the complexity changes dramatically. If multiple tasks can potentially be scheduled in the visibility window of a satellite, the problem becomes NP-Hard [2].

3 THE VMOC SCHEDULER

VMOC stands for the "Virtual Missions Operations Center." The VMOC scheduler was developed by the US Naval Research Lab. The VMOC scheduler uses a Steady State Genetic Algorithm, as well as a permutation representation to schedule satellite resources; Earth Observing Systems (EOS) is a primary example.

The deployed VMOC scheduler uses a permutation representation and a "greedy scheduler" which maps the permutation to a schedule. The permutation is used as a priority queue, which places tasks in the schedule in a "first come first serve" fashion.

The VMOC GA uses a multi-objective function which is integrated into a single evaluation cost. The number of conflict free tasks which can be scheduled is one important factor. But users can also provide a "priority score" for each task to indicate its urgency. Finally VMOC can also evaluate the placement of a task in the available time window. By default the VMOC GA scheduler considers 1) a total priority score (sum of the priorities for the scheduled tasks), 2) the total number of tasks scheduled, and 3) the placement of the tasks in a preferred position in the time window. But users have the option to weight these three objectives differently.

There is currently a soft fixed-time budget for scheduling. This means that the Genetic Algorithm limits the number of evaluations used to construct a schedule. Currently, multi-objective optimization methods are not used because the fixed-time budget available for producing a schedule prohibits the use of methods that do not work well with fixed budget optimization strategies. But this is also an area for potential future work.

The original VMOC GA deployment borrows key ideas from the work of Wolfe and Sorensen [18], including the use of the PMX operator. Selection is implemented by picking one parent from the best half of the population and the second parent randomly. Syswerda's Order Crossover has also been added to the current VMOC GA implementation.

The VMOC scheduler can also be "seeded" with good initial solutions. For example, if priority scores are used, one can sort all of the tasks by the priority score to create a permutation. As the permutation is constructed, a schedule is also constructed. There can be ties in the priority scores. In this case, tasks are compared to see how many windows are still available for each task to be scheduled. The task with the fewest remaining windows (the fewest placement options) is placed next in the permutation. Any remaining ties are broken randomly. The complete permutation can be added to the initial random population of the VMOC GA.

In this paper we do not present real world data, which is typically difficult to provide for many space based systems. Instead we have built a problem generator which captures key characteristics of the VMOC application. This has the advantage that we can provide a public resource for benchmarking and comparative studies.

In the typical VMOC application, satellite resources are over-constrained. And, as we have noted, there is also a fixed budget constraint which limits the amount of time which can be spent on generating and evaluating a schedule. This also means that solutions are generally not a "global optimum," and that solutions with different evaluations are seen across multiple runs.

We stress that this type of satellite scheduling application virtually always involves a human scheduler in the loop who inspects (and often changes) a schedule before it is deployed. Thus, how the schedule is presented to the human is also important.

In this paper, we do not look at all of the possible objectives of VMOC scheduling. Our experiments are designed to better isolate and understand different aspects of the Genetic Algorithms being used in our study.

3.1 Problem Generator

The problem generator takes three input parameters and yields random resource scheduling problems. The parameters are the number of each type of satellite, like Low Earth Orbit (LEO) and Medium Earth Orbit (ME) satellites, the number of tasks, and the task density. The task density controls how many alternatives, or options, a task has. For example, a given task can be executed in multiple windows in one or more satellites. The task options appear in preferential order in the dataset, meaning that the first listed location is the first preference. The problem generator also assumes the unit time is one minute. Each schedule spans 1440 minutes (24 hours), and tasks windows start and end on the same day.

The visibility windows are randomly generated according to predefined ranges and for a random subset of satellites. LEO visibility windows vary from 10 to 15 minutes, while MEO visibility windows vary from 60 to 120 minutes, which is consistent with visibility windows of real-world satellite orbits. The task duration is also randomly defined within the task visibility window.

The problem generator does not model orbital elements, like period, eccentricity, and inclination. So the problems might not have the same structure observed in real satellite resource scheduling problems. The experimental results suggest that the generator yields highly constrained problems. For the experiments, we generated 9 problems varying the number of tasks for a fixed set of 5 LEO and 10 MEO satellites. Since the resource set is fixed, the problem difficulty increases with the number of tasks.

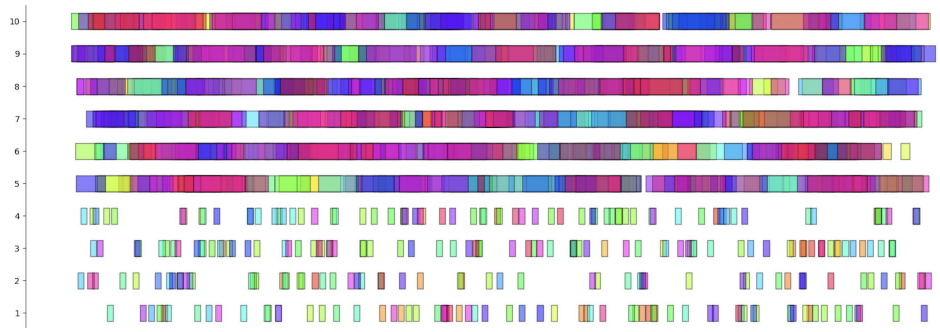

Figure 2: This Gantt Chart shows 10 resources to be scheduled. Each row is a resource. Time is on the x-axis. All requests are shown in all possible locations. The goal is to schedule each request in one location, while minimizing conflicts and overlaps.

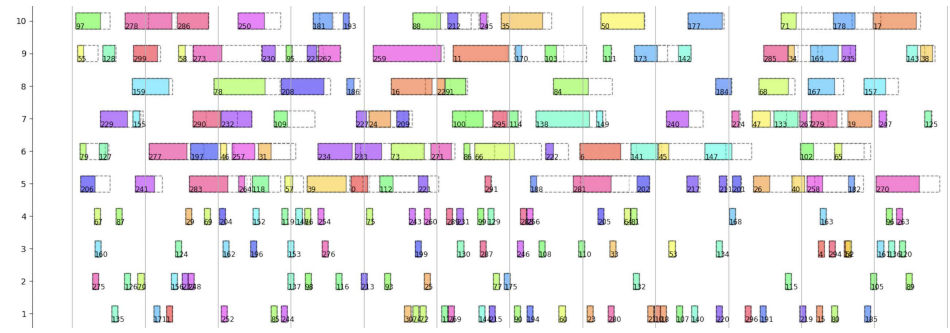

Figure 3: This Gantt Chart shows an optimized schedule, where each request is scheduled in exactly one location. Windows of visibility are still displayed. This example is taken from a larger scheduling problem with 15 resources.

4 THE GREEDY SCHEDULER

An ideal schedule has no conflicting tasks and makes optimal use of satellite visibility. However, a conflict-free schedule is almost never possible. In some domains (such as AFSCN) human schedulers might want to assign all tasks; in this case, minimizing overlaping conflict time can be better than maximizing the total number of conflict free tasks. It is easier for a human operator to fix conflicts by making small adjustments to a schedule where overlaps have already been minimized.

The greedy scheduler takes a permutation of tasks as input and tries to schedule them in such a way as to minimize the total overlap time. There are multiple ways this might be done, and these alternatives have not been highlighted and evaluated in the published GA scheduling literature.

Assume there is a permutation of tasks $[9, 2, 8, 7, 5, 4, 1, 3, 0, 6]$. At first the schedule is empty and tasks are directly placed in the schedule. In our example, assume that $[9, 2, 8, 7, 5]$ are directly placed in the schedule without conflict. But then task 4 produces a conflict in every possible schedule option. What should the scheduler do at this point? We explore two alternatives.

Alternative 1: Skip over tasks that produce a conflict. Continue to add tasks to the schedule that do not produce a conflict. After all of the tasks in the permutation have been considered, schedule the remaining tasks in the order in which they appear in the permutation. Place the conflicted task in the schedule so as to minimize overlap. Assume $[9, 2, 8, 7, 5]$ have been scheduled. Assume we skip task 4, then schedule task 1, skip task 3 and 0 (due to conflicts) and schedule task 6. After all conflict free tasks are scheduled, the remaining tasks $[4, 3, 0]$ are scheduled so as to reduce overlaps.

Alternative 2: When a task is encountered in the permutation that produces a conflict, the task is placed in the schedule immediately so as to minimize overlap.

The objective function is to minimize total conflict overlaps. However, results of the Alternative 1 and Alternative 2 decisions in the Greedy Scheduler will produce different results. Alternative 1 will tend to produce fewer conflicts, but with larger overlaps. Alternative 2 will tend to produce more conflicts, but with smaller overlaps. This demonstrates that it is not just the evaluation function that impacts the quality of the solutions that are generated.

5 PERMUTATION OPERATORS

We use a representation which is a permutation of tasks. The first use of a permutation based representation and Genetic Algorithm for scheduling problems was by Whitley et al. [16]. However, this work used the "edge recombination" operator that was better suited to the Traveling Salesman Problem than for scheduling problems. Davis introduced the first "Order Crossover" operator [7], but Syswerda later introduced a different "Order Crossover" operator [14, 15]. In this paper we compare Syswerda's Order Crossover and Partially Matched Crossover (PMX).

5.1 Order Crossover

Syswerda developed some of the most effective recombination operators for scheduling using permutation-based representations. Two of these are Syswerda's "Order Crossover" and Syswerda's "Position Crossover." These are different from the Order Crossover operator developed by Davis [6] and the PMX operator developed by Goldberg [8]. Syswerda applied a combination of Order Crossover and Steady State Genetic Algorithms to several real-world problems. Syswerda developed applications that involved personnel scheduling and resource scheduling [14, 15]. Early work by Parish [12] applied Syswerda's order crossover operator to the Air Force Satellite Control Network (AFSCN) scheduling problem. Wolfe and Sorensen [18] applied the PMX operator to an Earth Observing Systems scheduler. They reported finding no difference between PMX and Davis' Order Crossover.

Syswerda's order crossover operator randomly selects Z randomly selected tasks in the permutation corresponding to Parent 2. The tasks selected from Parent 2 are then located in Parent 1. The selected tasks are then reordered in Parent 1 so that they appear in the same relative order in which they appeared in Parent 2; these reordered Z tasks are copied to the child such that the positions come from Parent 1, the order comes from Parent 2. Those tasks in Parent 1 that do not correspond to the randomly selected task drawn from Parent 2 are passed directly to the offspring. Thus, the tasks that are directly passed from Parent 1 to the offspring are inherited by absolute position. The following is an example

```
Parent 1: A B C D E F G H I J
Parent 2: C F J E H B A D I G
```

Assume the tasks F, B, A and G are selected from Parent 2 (in that order). These same tasks are now reordered in Parent 1 using the ordering from Parent 2. We will use the # symbol to indicate a task that is directly inherited from Parent 1 and thus not reordered:

```
A B # # #  F G # # #   =>   F B # # # A G # # #
```

All other tasks (those in the # positions) are directly copied from Parent 1 to the offspring. Thus the final offspring is

```
Child: F B C D E A G H I J
```

Syswerda's crossover differs from Davis's order crossover because it uniform randomly selects the tasks to be exchanged. Davis' order crossover selects a continuous block of the permutation. The PMX operator also selects a continuous block of the permutation.

Syswerda's order crossover and position crossover are identical when correctly parameterized. Let n denote permutation length and the number of tasks to be scheduled. Note that in the above example that $n - Z$ tasks are directly inherited (copied) from Parent 1 by both position and task ID. This is also what happens with "position crossover." Thus, order crossover inherits task (and their relative order) from Parent 2, and fills in the rest of the child by copying tasks from Parent 1 in exactly the same positions. But, position crossover first copies tasks from Parent 1 in exactly the same positions, and then fills the empty positions by copying tasks from Parent 2 in relative order. The results are identical if order crossover selects Z tasks from Parent 1 to reorder, and position crossover selects the remaining $n - Z$ tasks by position [17].
We next prove a complexity result for Syswerda's crossover.

Theorem 1: Syswerda's Order Crossover can be implemented in $O(n)$ time, with a constant bound of at most $4n$ write operations and n compare operations.

Proof: The proof implements Position Crossover, which also implements Order Crossover. Let π denote a permutation of tasks. Let $Touch$ be an auxiliary bit vector of length n. The Touch vector is initialized to 0 bits and is indexed by the task ID denoted by π_i. The Child permutation is initialized to be an empty vector. Copy $n - Z$ randomly selected tasks directly from Parent 1 to the Child by position. For each task π_i copied to the Child, set $Touch(\pi_i) = 1$.

Next place a pointer at the beginning of the Child and another pointer at the beginning of Parent 2. Advanced the pointer in the Child vector until an empty position is found. Starting in Parent 2, advance the pointer until a task π_j is found in Parent 2 where $Touch(\pi_j) = 0$. Copy π_j to the current position in the Child. There is no update of $Touch(\pi_j)$ since π_j is visited only once.

This process is repeated as tasks are transferred from Parent 2 to the Child. There is one compare to the Touch vector for the n tasks in Parent 2. There are exactly $2n$ write operations to initialize and generate the Child. There are at most $2n$ write operations to initialize and generate the $Touch$ vector. □

This proof ignores calls to the random number generator. All of the experiments in this paper select half of the task requests during Order Crossover. It is possible to make 1 call to a random number generator, and then convert the resulting number into a bit string. With one (large) bit string we can select half of the task requests randomly in expectation. With two bit strings we can select a quarter of the task requests.

5.2 PMX Crossover

Partially Mapped Crossover (PMX) [10] first selects two crossover points and copies a contiguous block of tasks from Parent 1 to the Child. This first step is illustrated in this example.

```
Parent 1: A B C * D E F G * H I J
Parent 2: C F J * E H B A * D I G
   Child: _ _ _ * D E F G * _ _ _
```

Examining Parent 2, tasks not present in the block from Parent 1 are next copied from Parent 2 to the Child (in this case C, J, I).

```
Child: C _ J * D E F G * _ I _
```

Finally a chain of substitutions occurs using permutation *cycles*. Two tasks which appear in the same position in two permutations are members of a cycle. We compute the closure of a cycle using this membership test. Thus, $((F, B))$ form a closed cycle. Cycles are ordered. For example $((D, E), (E, H), (H, D))$ implies the ordering $D \rightarrow E \rightarrow H \rightarrow D$. Thus H replaces D in Parent 2 and is passed to the Child. The cycle $G \rightarrow A \rightarrow C \rightarrow J \rightarrow G$ implies that A replaces G in Parent 2. This process results in the following offspring:

```
Child: C B J D E F G H I A
```

The chain of substitutions make PMX more disruptive than Order Crossover. Both Order Crossover and PMX can be implemented in $O(n)$ time.

Table 1: Comparisons of Order Crossover (OX) and PMX after 100,000 evaluations: mean ± s.d. A t-test finds that every comparison of OX and PMX is significant at the 0.0001 level

Size	PopSize	SGA-OX	SGA-PMX	SSGA-OX	SSGA-PMX
200	256	227 ± 45	300 ± 51	309 ± 58	374 ± 58
	512	189 ± 29	244 ± 41	263 ± 50	313 ± 52
	1024	**172 ± 27**	221 ± 36	234 ± 49	293 ± 53
400	256	4913 ± 149	5060 ± 102	5105 ± 152	5203 ± 93
	512	4826 ± 116	5005 ± 104	4962 ± 121	5117 ± 80
	1024	**4741 ± 103**	4916 ± 92	4962 ± 166	5034 ± 78
600	256	14699 ± 109	14805 ± 70	14800 ± 106	14896 ± 54
	512	14622 ± 49	14745 ± 50	14759 ± 92	14841 ± 55
	1024	**14582 ± 68**	14690 ± 73	14678 ± 97	14796 ± 58
800	256	18800 ± 22	18849 ± 26	18862 ± 21	18914 ± 31
	512	18774 ± 10	18827 ± 26	18819 ± 23	18877 ± 26
	1024	**18763 ± 10**	18799 ± 20	18802 ± 21	18845 ± 27
1000	256	26499 ± 11	26537 ± 20	26543 ± 21	26579 ± 26
	512	26488 ± 13	26524 ± 14	26522 ± 14	26559 ± 21
	1024	**26479 ± 7**	26507 ± 17	26501 ± 13	26535 ± 18

5.3 The Problem with Local Search

Why not just apply Iterated Local Search to this satellite scheduling problem? The problem with local search is that the "obvious" local search neighborhood is $O(n^2)$ in size. Most neighborhoods involve selecting two tasks in the schedule. Then some change is made to the permutation based on the location of these two tasks. For example, the two tasks might be exchanged (also known as a "Swap" neighborhood) or one task might be moved in front of the other. Technically these neighborhoods are of size $n(n-1)/2$. Thus, if n=1000, it would require 499,500 function evaluations to enumerate one neighborhood. This is obviously not efficient in terms of function evaluations. On the AFSCN scheduling problem Barbulescu et al. [3] found it was better to randomize the local search neighborhood instead of enumerating the local search neighborhood; if an improving move is found, more moves of the same task request are then explored. However, a Steady State Genetic Algorithm (SSGA) still produced better results than randomized local search [3].

6 EXPERIMENTS AND RESULTS

All of our experiments used 30 runs. Most of our experiments used 100,000 evaluations. For the VMOC application, the number of evaluations may be more limited due to time constraints. Thus, we also examined a soft-budget using 30,000 evaluations.

The standard Genetic Algorithm (SGA) is taken from the books of Goldberg [8] and Holland [11]. One modification is the use of Tournament Selection. Let PopSize denote the population size.

6.1 Syswerda's Order Crossover vs PMX.

Table 1 presents a comparison of the PMX operator to Syswerda's Order Crossover. We include both the SGA and SSGA in the comparison. The experiments used Alternative 1 of the Greedy Schedule described in Section 4. A *t-test* shows that every single comparison of PMX against Order Crossover is statistically significant at the $p = 0.0001$ level using a two-tailed test.

The results in Table 1 here were generated using 100,000 evaluations. Nevertheless, run time graphs (like those in Figure 4) show that PMX is consistently poorer than Syswerda's Order Crossover no matter how many evaluations are used or when search is terminated. Table 1 presents only 5 or the 10 instances (instances 200, 400, 600, 800, and 1000), but we ran all 10 instances. Table 1 shows population sizes of 256, 512, and 1024.

It is possible that some variation on earth observing system scheduling or other multi-resource scheduling problem might favor the use of PMX. However we do not find this in our experiments. We also confirmed that Order crossover is better than PMX for the actual VMOC implementation on real world data.

6.2 Steady State GAs vs. Standard GAs

The standard GA (SGA) generates a random population. The population is then evaluated. Next Tournament Selection is used to select two parents, which are recombined to create a child. The child is passed to the next generation. We use a basic version of Tournament Selection where three individuals are selected, but only the best individual becomes a parent. This continues until the entire new population is generated.

Children replace the parents in an SGA. Our standard GA also uses elitism; the best member of the population is guaranteed to survive from one generation to the next [8]. After the new population is evaluated, the process is repeated. For scheduling problems, it is common to not use mutation. No mutation was used in our experiments. In the VMOC GA, mutation is used only if a solution has the same evaluation as another solution already in the population.

The Steady State Genetic Algorithm (SSGA) is monotonically improving, and does not have a well-defined notion of "generation." The Steady State Genetic Algorithm also starts with a random population. But after the population is evaluated, it is sorted based on the evaluation function. Tournament Selection is applied twice in order to select two parents. These two parents generate a child by crossover. The child is then evaluated. If it is better than the worst member of the population, it replaces the worst member of the population immediately. The new member of the population is then "bubbled" up into the correct position so that the population is maintained in sorted order.

Goldberg and Deb [9] proved that Steady State Genetic Algorithms generate much higher selection pressure than standard Genetic Algorithms. Steady State Genetic Algorithm uses a form of truncation selection. This results in "double selection," once from Tournament Selection and a second time from truncation selection.

There is a fundamental trade-off when using a standard GA and the Steady State GA. Generally, the Steady State GA improves solutions faster, but also runs out of diversity faster. At the same time, the Steady State GA can intensify the search around the best solutions in the current population. A standard GA improves solutions more slowly but maintains diversity longer. The standard GA does not retain whole solutions but instead retains fragments of solutions. Which is better? That depends on the problem. If a Steady State GA can generate an optimal solution or an acceptable solution quickly, then the Steady State GA is perhaps "better." But sometimes a standard GA produces a better solution given more time using a smaller population. The Steady State GA can be "slowed down" by using a larger population size. But then, the Steady State GA also converges more slowly using a larger population.

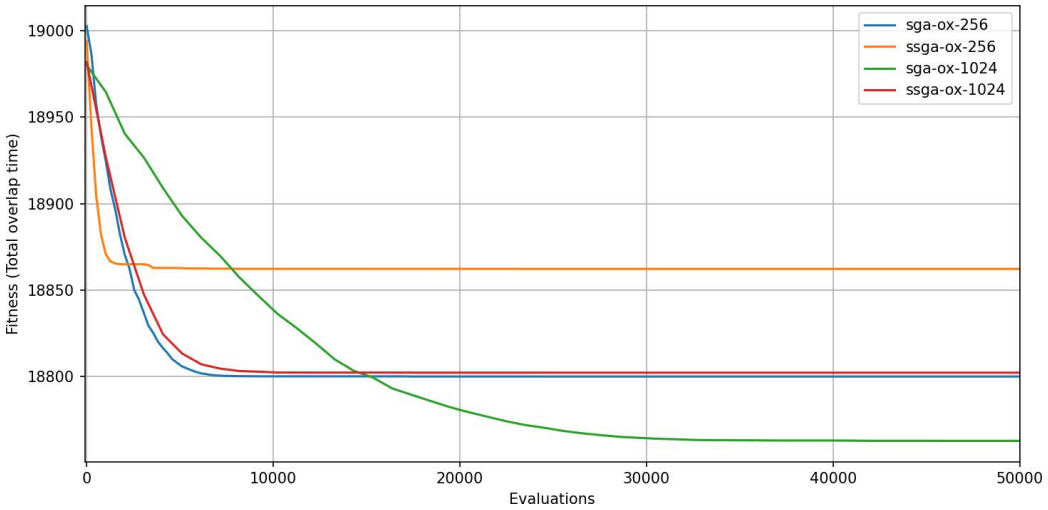

Figure 4: Progress over time is plotted using an SGA and SSGA with populations of 256 and 1024. The best configurations depends on the PopSize and the runtime budget. This problem instance has 800 tasks to be scheduled.

Figure 4 shows the kind of trade-off we have seen repeatedly with a standard generational Genetic Algorithm (SGA) and a Steady State Genetic Algorithm (SSGA). Using a smaller population (e.g. 256) the Steady State Genetic Algorithm improves more rapidly, but can prematurely converge. Given enough time the SGA using a small population overtakes the SSGA to produce better results. However, when the population size is increased to 1024, the SSGA yields the best results after 40,000 evaluations in Figure 4. This is the average of 30 runs for one problem. This general behavior was observed on all problem instances. Depending on the population size and the number of evalutions, either the SGA or the SSGA might be best. The Steady State GA requires larger populations and more evaluations to produce better results.

We first present results using only 30,000 evaluations. This is based on the constraint that each evaluation of the VMOC scheduler is expensive and a fixed-budget of 30,000 evaluations is a reasonable sample budget. Also, keep in mind that in Figure 4 we see that the SGA is better than SSGA in the range of 10,000 to 30,000 evaluations, depending on the population size.

The results in Table 2 show that sometimes the Steady State GA yields the best average results, and often the standard GA yields the best average results. But the results produced by the two algorithms are extremely similar, and the differences between the standard GA and the Steady State GA are not statistically significant. This would seem to be surprising given the wide-spread use of the Steady State GA for resource scheduling problems. We also note that the standard SGA consistently performs well with a population size of 256. On the other hand, the Steady State GA is usually best using a population of 512.

We have run experiments using 100,000 evaluations which can increase performance (see Table 3). We also ran experiments using 1 million evaluations. However, using 1 million evaluations did not greatly change the solution quality. The choice of the Greedy Schedule is far more important than the number of evaluations, which we document next.

Table 2: Results using 30,000 evaluations. These results reflect the effect of using a fixed-budget. For these results the SGA with a small population (256) are almost identical to the results of using a Steady State GA with a larger population.

Size	PopSize	Best Solution		Mean ± SD	
		SGA	SSGA	SGA	SSGA
200	256	**134**	191	199 ± 42	286 ± 57
	512	138	183	192 ± 42	263 ± 54
	1024	135	147	**153 ± 11**	204 ± 43
300	256	1686	1956	1883 ± 144	2106 ± 120
	512	1617	1785	1756 ± 108	1955 ± 83
	1024	**1561**	1685	**1734 ± 66**	1893 ± 136
400	256	4348	4554	4545 ± 136	4752 ± 143
	512	4249	4364	4456 ± 132	4635 ± 164
	1024	**4218**	4306	**4437 ± 125**	4586 ± 164
500	256	7882	8035	8046 ± 118	8269 ± 126
	512	7796	7940	**7911 ± 92**	8097 ± 93
	1024	**7783**	7907	7912 ± 46	8024 ± 88
600	256	14280	14379	14371 ± 47	14513 ± 96
	512	14250	14336	14336 ± 58	14431 ± 62
	1024	**14241**	14316	**14311 ± 32**	14396 ± 66
700	256	16747	16800	16796 ± 48	16870 ± 49
	512	16721	16745	16763 ± 34	16822 ± 54
	1024	**16718**	16728	**16743 ± 17**	16786 ± 40
800	256	18646	18679	18673 ± 19	18716 ± 23
	512	18645	18660	18656 ± 8	18689 ± 29
	1024	**18642**	18648	**18649 ± 6**	18670 ± 13
900	256	24583	24588	24591 ± 7	24615 ± 13
	512	**24581**	24584	24584 ± 2	24598 ± 11
	1024	**24581**	24583	**24583 ± 2**	24591 ± 4
1000	256	26454	26480	26471 ± 11	26501 ± 15
	512	26452	26462	26461 ± 8	26484 ± 13
	1024	**26451**	26457	**26456 ± 4**	26470 ± 9

Table 3: Comparing two Greedy Schedulers using 100,000 evaluations. "Bumps" refer to the number of conflicted tasks. "Conflict Overlap" sums the overlap times in conflicting tasks. Scheduler Alternative 1 does a better job of reducing "Bumps" and Alternative 2 does a better job of reducing "Conflict Overlap." Each entry reports mean and ± standard deviation.

		Scheduler Alternative 1				Scheduler Alternative 2			
Size	PopSize	Bumps		Conflict Overlap		Bumps		Conflict Overlap	
		SGA	SSGA	SGA	SSGA	SGA	SSGA	SGA	SSGA
200	256	13 ± 2	16 ± 3	227 ± 45	309 ± 58	13 ± 3	18 ± 3	199 ± 42	285 ± 57
	512	11 ± 2	15 ± 2	189 ± 29	263 ± 50	13 ± 3	16 ± 2	191 ± 42	263 ± 54
	1024	10 ± 2	13 ± 2	172 ± 27	234 ± 49	10 ± 1	13 ± 3	153 ± 11	203 ± 43
300	256	55 ± 4	58 ± 4	2103 ± 120	2348 ± 173	65 ± 3	69 ± 3	1883 ± 144	2106 ± 120
	512	53 ± 4	56 ± 3	2050 ± 99	2171 ± 107	63 ± 3	66 ± 2	1756 ± 109	1955 ± 83
	1024	51 ± 3	54 ± 3	1966 ± 116	2105 ± 90	62 ± 3	65 ± 3	1696 ± 80	1892 ± 136
400	256	101 ± 3	103 ± 3	4913 ± 149	5105 ± 152	119 ± 3	122 ± 4	4545 ± 136	4752 ± 143
	512	100 ± 3	102 ± 3	4826 ± 116	4962 ± 121	119 ± 3	121 ± 3	4456 ± 133	4634 ± 165
	1024	100 ± 2	102 ± 3	4741 ± 103	4962 ± 166	118 ± 2	121 ± 3	4384 ± 149	4585 ± 164
500	256	156 ± 3	159 ± 3	8467 ± 135	8607 ± 127	191 ± 3	196 ± 5	8046 ± 118	8268 ± 126
	512	154 ± 2	157 ± 4	8344 ± 86	8476 ± 119	192 ± 4	193 ± 4	7911 ± 92	8097 ± 93
	1024	154 ± 3	157 ± 3	8273 ± 114	8421 ± 110	192 ± 4	195 ± 4	7875 ± 49	8024 ± 88
600	256	241 ± 4	242 ± 4	14699 ± 109	14800 ± 106	297 ± 6	301 ± 5	14371 ± 47	14512 ± 96
	512	240 ± 3	241 ± 3	14622 ± 49	14759 ± 92	297 ± 5	300 ± 5	14336 ± 58	14431 ± 62
	1024	239 ± 3	241 ± 4	14582 ± 68	14678 ± 97	296 ± 5	300 ± 5	14300 ± 36	14396 ± 66
700	256	291 ± 5	295 ± 6	16990 ± 69	17062 ± 62	375 ± 5	377 ± 5	16796 ± 48	16870 ± 49
	512	291 ± 5	293 ± 5	16933 ± 32	17015 ± 52	373 ± 5	378 ± 6	16763 ± 34	16822 ± 54
	1024	290 ± 4	294 ± 4	16899 ± 36	16972 ± 47	373 ± 6	375 ± 5	16741 ± 17	16786 ± 41
800	256	357 ± 4	363 ± 5	18800 ± 22	18862 ± 21	451 ± 6	453 ± 7	18673 ± 19	18715 ± 23
	512	358 ± 4	362 ± 6	18774 ± 10	18819 ± 23	449 ± 5	451 ± 7	18656 ± 8	18689 ± 29
	1024	359 ± 5	360 ± 4	18763 ± 10	18802 ± 21	448 ± 5	452 ± 6	18648 ± 6	18670 ± 13
900	256	430 ± 7	433 ± 7	24680 ± 22	24751 ± 37	547 ± 6	550 ± 7	24591 ± 7	24614 ± 13
	512	429 ± 6	434 ± 6	24660 ± 15	24705 ± 31	547 ± 6	551 ± 7	24584 ± 2	24598 ± 11
	1024	429 ± 6	430 ± 6	24648 ± 13	24675 ± 19	547 ± 5	552 ± 5	24583 ± 2	24591 ± 4

6.3 Alternative Greedy Schedulers

Section 4 described two alternative versions of the greedy scheduler. **Alternative 1** delays placing tasks in the schedule which are in conflict, only placing them after conflict free tasks are scheduled. **Alternative 2** immediately schedules tasks which are in conflict so as to minimize overlap time. In Table 3 we present results using the two alternative greedy schedulers.

Alternative 1 can be used to first generate a conflict free schedule by "Bumping" tasks out of the schedule if they cause a conflict. It is desirable to minimize "Bumps". However, because we are also minimizing overlaps, it is possible to look for conflicted tasks that can be "trimmed" and placed in the schedule. In practice, getting 90% of the requested time for a task on a satellite resource is usually better than being bumped from the schedule entirely, but this can of course depend on the nature of the task request.

It is important to stress that the *objective function for all of the results in Table 3* was the total "Conflict Overlap" time. Thus the difference between **Alternative 1** and **Alternative 2** results are only due to how the permutation is mapped to a schedule in the form of a Gantt Chart. This is perhaps surprising since the "Conflict Overlap" results are extremely different for **Alternative 1** and **Alternative 2**. If we were only interested in minimizing overlaps, then **Alternative 2** would obviously be preferred.

7 CONCLUSIONS

The VMOC Genetic Algorithm scheduler is currently deployed and routinely used by the U.S. Navy to schedule satellite resources. We present a problem generator which provides a sandbox for testing schedulers for VMOC scheduling applications. We isolate specific questions about scheduler performance in our experiments. We evaluate the use of Syswerda's Order Crossover vs PMX in Table 1.

While Steady State Genetic Algorithms (SSGA) have regularly been used for scheduling applications which use an embedded a greedy scheduler, our results suggest that standard generational Genetic Algorithms (SGA) are highly competitive, and can use a smaller population size as well as avoiding the need to maintain the population in sorted order. Smaller populations can yield faster convergence, as documented in Table 2 and Figure 4.

The results in Table 3 dramatically illustrates the impact that the Greedy Scheduler can have on the optimization process. This is extremely important to document. Practitioners who use Genetic Algorithm to build schedulers with evaluation functions that use complex indirect mappings **cannot** automatically assume that they are generating near optimal solutions relative to the evaluation function. How an indirect mapping is constructed can greatly bias optimization results for better or for worse.

REFERENCES

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

[2] Laura Barbulescu, Jean-Paul Watson, L. Darrell Whitley, and Adele E. Howe. 2004. Scheduling Space-Ground Communications for the Air Force Satellite Control Network. *J. Scheduling* 7, 1 (2004), 7–34.

[3] Laura Barbulescu, L. Darrell Whitley, and Adele E. Howe. 2004. Leap Before You Look: An Effective Strategy in an Oversubscribed Scheduling Problem.. In *Proceedings of the 19th conference on Artificial Intelligence*. 143–148.

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

[5] T. Cormen, C. Leiserson, and R. Rivest. 1990. *Introduction to Algorithms*. McGraw Hill, New York.

[6] Lawrence Davis. 1985. Applying Adaptive Algorithms to Epistatic Domains. In *Proc. IJCAI-85*.

[7] Lawrence Davis. 1985. Job Shop Scheduling with Genetic Algorithms. In *International Conference on Genetic Algorithms and their Applications*.

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

[9] D. Goldberg and K. Deb. 1991. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In *Foundations of Genetic Algorithms*, G. Rawlins (Ed.). Vol. 1. Morgan Kaufmann, 69–93.

[10] David Goldberg and Jr. Robert Lingle. 1985. Alleles, Loci, and the Traveling Salesman Problem. In *Int'l. Conf. on GAs and Their Applications*, John Grefenstette (Ed.). 154–159.

[11] J. Holland. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

[12] D. Parish. 1994. *A Genetic algorithm approach to automating satellite range scheduling*. Master's thesis. AFIT, Air Force Institute of Technology.

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

[14] Gilbert Syswerda. 1991. Schedule Optimization Using Genetic Algorithms. In *Handbook of Genetic Algorithms*, Lawrence Davis (Ed.). Van Nostrand Reinhold, New York, Chapter 21.

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

[16] Darrell Whitley, Timothy Starkweather, and D'ann Fuquay. 1989. Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator. In *Proc. of the 3rd Int'l. Conf. on GAs*, J. D. Schaffer (Ed.). Morgan Kaufmann.

[17] L. Darrell Whitley and Nam-Wook Yoo. 1995. Modeling simple genetic algorithms for permutation problems. In *Foundations of Genetic Algorithms (FOGA 3)*, Darrell Whitley and Michael Vose (Eds.). Morgan Kaufmann, 163–184.

[18] William J. Wolfe and Stephen E. Sorensen. 2000. Three Scheduling Algorithms Applied to the Earth Observing Systems Domain. *Management Science* 46, 1 (2000), 148–166. http://www.jstor.org/stable/2634914