Analyzing Basic Representation Choices in Oversubscribed Scheduling Problems

Content Areas: scheduling, search, heuristics

Abstract

Both direct schedule representations as well as indirect permutation-based representations in conjunction with schedule builders are successfully employed in oversubscribed scheduling research. Recent work has indicated that in some domains, searching the space of permutations as opposed to the schedule space itself can be more productive in maximizing the number of scheduled tasks. On the other hand, research in domains where task priority must be treated as a hard constraint has shown the effectiveness of local repair methods that operate directly on the schedule representation. In this paper, we investigate the comparative leverage provided by techniques that exploit these alternative representations (and search spaces) in this latter oversubscribed scheduling context. We find that an inherent difficulty in specifying a permutation-based search procedure is the tradeoff in guaranteeing that priority is enforced while giving the search sufficient flexibility to progress. Nonetheless, with some effort spent in tuning the move operator, we show that a permutation-space technique can perform quite well on this class of problem in cases of low oversubscription and in fact was able to find new optimal solutions to a few previously published benchmark problems. Not surprisingly, the permutation-space search does not perform as well as the schedule-space search in terms of maintaining schedule stability.

1 Introduction

In The Sciences of the Artificial[Simon, 1981], Herbert Simon recounts the example of an ant making its way toward its eventual goal over a wave-molded beach. The ant's path seems somewhat random and somewhat directed, with a complexity that is not easily decipherable. Simon makes the point that the ant itself does not navigate by some complex algorithm, but that the apparent complexity arises from the features of the terrain it traverses. In a like vein, as we craft AI search techniques to navigate a complex space to a goal state, it has often proven fruitful to employ knowledge about the contours of the space in the search process. The search space itself appears very different depending on how the search is structured. By choosing a solution representation for a given problem and defining neighboring states, we are specifying one of potentially many possible search landscapes. It is generally accepted in AI that choosing the right representation for a problem can result in search spaces for which the solutions are easier to find.

Our specific focus in this paper is on oversubscribed scheduling problems - scheduling problems for which there is typically more to do than available resources will allow and the search problem is to determine which subset of activities to include. For this class of problem, scheduling research has successfully exploited the use of both direct schedule representations and indirect, usually permutation-based representations that can be expanded into direct representations as a basis for solution. However, very little research has directly analyzed the implications and tradeoffs of this representational choice for solving various oversubscribed scheduling problems. [Barbulescu *et al.*, 2006] has shown a permutation-based genetic algorithm to outperform a schedule repair based technique in the context of scheduling satellite downlink capacity. [Globus *et al.*, 2004; 2002] similarly argue the relative superiority of a permutation-based representation over a direct representation based on Gantt charts for scheduling requests for time on Earth observing satellites and demonstrate strong performance.

In this paper, we consider the basic question of representation choice in the context of another oversubscribed scheduling domain whose description together with a set of benchmark problems has recently made available to the research community: the AMC scheduling problem [Kramer and Smith, 2005a]. The AMC problem is distinguishable from most other oversubscribed scheduling problems studied in the literature in one major respect: task priority is treated as a hard constraint. This characteristic models a range of real-life situations, especially in the military operations, where the priority of a task trumps all other considerations and is not subject to tradeoff. It can be contrasted with the more typical treatment of priority in oversubscribed scheduling, where the search problem is formulated as one of optimizing a weighted sum of the priorities of all scheduled tasks and it is possible to trade one higher priority task for some number of lower priority tasks. Repair-based algorithms working in the schedule space can be designed to enforce this hard constraint. In fact, the *TaskSwap* procedure originally developed for solving the AMC problem is such a schedule-space search algorithm [Kramer and Smith, 2003; 2004]. The question we consider in this paper is whether the relative superiority of permutation-based techniques in other oversubscribed domains carries over to domains like the AMC problem where task priority is a hard constraint.

We hypothesized that it would be difficult to design a competitive search algorithm for the AMC domain using a permutation-based representation, mainly because of the inherent difficulties of enforcing the task priority constraint. The hypothesis was based on what we thought to be a reasonable idea: searching for solutions directly in the schedule space, as opposed to using an indirect representation which needs to be translated into a schedule, should be able to adhere to the hard task priority constraint, while at the same time focusing the search to produce good solutions faster. We found that, after a certain amount of tuning, a permutation-based search technique can perform comparably with the repair-based technique for low to moderate levels of oversubscription. However, as the problems become more oversubscribed, the permutation-based technique is outperformed by the schedule-space technique. For problems with high levels of oversubscription there are very few opportunities of moving things around, and the permutation-based technique fails to identify them.

Before specifying AMC problem in more detail and the pair of solution techniques used in our analysis, we briefly summarize the general strengths and weaknesses of both permutation-space and schedule-space search procedures.

2 Related Work

For many scheduling applications in general and oversubscribed scheduling applications in particular, the representation of choice is a permutation of the tasks to be scheduled, together with a schedule builder to transform the permutation into a schedule [Whitley *et al.*, 1989] [Syswerda, 1991] [Globus *et al.*, 2004] [Barbulescu *et al.*, 2006]. The advantage of using an indirect representation is that a wide range of general-purpose search algorithms (from heuristic methods, such as hill climbing, simulated annealing, tabu search, genetic algorithms, to exact, tree search methods) can be employed to search the permutation space, while all the particularities of the domain are encapsulated in a schedule builder. As a downside, in general, one can not predict how a change in the current solution (permutation) affects the schedule until the schedule builder computes the new schedule. Also, depending on the schedule builder, the space of all schedules that can be reached from the permutation space is usually a proper subset of all possible schedules. It is not clear if this subset contains the optimal solution, or if not, how suboptimal it is. Finally, the permutation search space/schedule builder combination is not well suited for preserving schedule stability (since it is difficult to predict how a permutation change affects the corresponding schedule).

Searching the schedule space directly is an attractive alternative ([Johnston and Miller, 1994], [Chien *et al.*, 2000], [Kramer and Smith, 2003; 2004]). Powerful domain-specific heuristics are usually available (for example, various resource contention measures), and such measures can drive the search. Also, when schedule stability is important, search operators for the schedule space can be defined such that stability is preserved (by minimally changing the current schedule). While general-purpose search operators can still be defined, efficient search algorithms in the schedule space will most of the time use domain knowledge to decide how to reorganize the schedule. The challenge is in defining the right search operators given a particular scheduling application.

3 The AMC Scheduling Domain

The AMC scheduling problem described in abstracts the large airlift and tanker mission scheduling problem faced by the US Air Force Air Mobility Command (USAF AMC). [Kramer and Smith, 2005a] provide the following concise description of this problem:

- A set T of tasks (or missions) are submitted for execution. Each task $i \in T$ has an earliest pickup time est_i, a latest delivery time lft_t, a pickup location $orig_i$, a dropoff location $dest_i$, a duration d_i (determined by $orig_i$ and $dest_i$) and a priority pr_i
- A set Res of resources (or air wings) are available for assignment to missions. Each resource $r \in Res$ has capacity $cap_r \geq 1$ (corresponding to the number of aircraft for that wing).
- Each task i has an associated set Res_i of feasible resources (or air wings), any of which can be assigned to carry out i. Any given task i requires 1 unit of capacity (i.e., 1 aircraft) of the resource r that is assigned to perform it.
- Each resource r has a designated location $home_r$. For a given task i, each resource $r \in Res_i$ requires a positioning time $pos_{r,i}$ to travel from $home_r$ to $orig_i$, and a depositioning time $depos_{r,i}$ to travel from $dest_i$ back to $home_r$.

A schedule is a *feasible* assignment of missions to wings. To be feasible, each task i must be scheduled to execute within its $[est_i, lft_i]$ interval, and for each resource r and time point t, $\textit{assigned-cap}_{r,t} \leq cap_r$. Typically, the problem is over-subscribed and only a subset of tasks in T can be feasibly accommodated. If all tasks cannot be scheduled, preference is given to higher priority tasks. Tasks that cannot be placed in the schedule are designated as *unassignable*.

3.1 AMC Task Priority

In the AMC domain tasks (missions) are scheduled strictly by priority. There are five priority classes, each of which is further subdivided into 25 subclasses (giving a total of 125 distinct priorities). Priorities are denoted by three-character alphanumeric strings. The first digit ranges from 1 to 5 and defines the priority class: 1, 2, 3, 4 or 5. The second digit ranges from A to E, and the third from 1 to 5. 1 and A denote the highest (most important) priorities, and the first digit is the most significant. For instance a 1C3 priority is higher than 2A1. The highest priority is a 1A1, and the lowest a 5E5. In terms of operational procedure, lower priority missions are not traded off for higher priority missions. It is not normal procedure to ‘‘trade off” or bump one priority 1– mission to get some number of priority 2– missions into the schedule.

3.2 Priority as a Hard Constraint

A simple way of producing schedules that satisfy the task priority constraint is to assign prospective tasks in priority order. Unless the schedule is extremely oversubscribed, tasks in the highest priority classes will tend to be feasibly assigned. Assuming that the problem is oversubscribed though, at some point resource unavailability will not allow the next pending task to be inserted. This task will be marked '*unassignable*, and our simple procedure then moves to the next highest priority task that might be scheduled. Since subsequent (lower priority) tasks that are considered may have different constraints (e.g., request resources over different time intervals), it is possible that they can still be feasibly assigned, if requisite resources are still available for the time periods required. Hence it is possible for a schedule satisfying the priority constraint to include tasks of lower priority than those of tasks that have been forced to be excluded.

More precisely, we say that a solution respects the hard priority constraint if it adheres to the following two rules:

1. There is no other solution which includes additional tasks in the highest priority class.[1]
2. For all solutions which include the same tasks of priority (class) k, there exists no solution which includes additional tasks of priority $k + 1$ (next lower priority class), for any k starting at the highest priority class.

An optimal solution then is a solution satisfying the 2 rules above that also minimizes the number of unassigned tasks at each level. We define S to be *hard-priority optimal* if there is no solution with exactly the same number of unassignables in the first k highest priority classes and fewer unassignables in the $k + 1$th (next lower) priority class, for any k.

3.3 Approximating the Hard Priority Constraint

In practice, given that the problem we consider is$\mathcal{NP}$-complete, it is not in general feasibly possible to prove that a solution is hard-priority optimal, let alone that the above two rules for enforcing the task priority are respected. Instead, we choose to define a new objective function that amplifies the differences between priority classes. Given a schedule, any change that would insert into the schedule a lower priority unassignable by unscheduling (bumping) a higher priority task is made very unattractive by the new objective function.

More specifically, we resort to using a heuristic *scoring value* for each priority class, that emphasizes the differences between different classes. First we translate the alphanumeric priority values into numeric values, such that '1' and 'A' correspond to 5, '2' and 'B' correspond to 4, '3' and 'C' to 3, '4' and 'D' to 2 and '5' and 'E' to 1. For example, 1A1 corresponds to 555 and 2B3 to 443. We then amplify the differences between classes by adding to this numeric value an increment $10^{7-\text{class}}$, where $class$ is the priority class[2]. For example, for 2B3 the assigned score is 100,443.

[1]This does not imply that such other solution should schedule exactly the same tasks at the other lower priority levels; in fact it is likely that the scheduled lower priority tasks would be different.

[2]The value of the increment results in a one order of magnitude difference in scoring values between successive priority classes, where the smallest increment is of the same magnitude with the priority numeric values.

We define the *priority score* for a schedule as the sum of the scoring values for all the unassignables. The new objective function minimizes the priority score.

4 Algorithms

Previous research has found *TaskSwap*, a repair-based algorithm, to perform well on the AMC scheduling problem.[Kramer and Smith, 2003; 2004; 2005b] Conceptually, the *TaskSwap* procedure proceeds by temporarily relaxing the priority constraint, retracting one or more scheduled tasks (regardless of priority) to allow insertion of a previously unassignable task, and then recursively attempting to reintroduce retracted tasks elsewhere into the schedule. We define a variant of this base procedure in Section 4.2 as our representative schedule-space search procedure for the experimental analysis to follow.

A wide range of algorithms searching in permutation-based space have been implemented for oversubscribed scheduling domains. One stands out because of its similarity with *TaskSwap*: Squeaky Wheel Optimization (SWO) [Joslin and Clements, 1999]. SWO temporarily assigns higher priority to unscheduled tasks and attempts to schedule them earlier. Also, *SWO* has been shown to be one of the best performing algorithms for another oversubscribed domain, scheduling satellite downlink capacity[Barbulescu *et al.*, 2004]. Hence we choose SWO as our representative permutation-space search procedure, defined in Section 4.3 below.[3]

4.1 The Schedule Builder

The schedule builder produces an initial schedule for both *TaskSwap* and *SWO*; also, it translates from the permutation search space to the schedule space while running *SWO*.Given a permutation of tasks, the schedule builder considers the tasks in the order in which they appear in the permutation and uses a look-ahead heuristic to assign start times and resources to tasks, based on predicted resource contention. This heuristic, *max-availability*, is described in some detail in [Kramer and Smith, 2005b]. To produce the initial schedule, the schedule builder is applied to the tasks sorted in priority order.

4.2 TaskSwap

As indicated above, the *TaskSwap* scheduling procedure of [Kramer and Smith, 2003; 2004] implements a repair-based approach to rearranging tasks in an input schedule so as to include additional (e.g., previously unassignable) tasks. The algorithm considers unassignable tasks one by one (in priority order) and attempts to insert them in the existing schedule by retracting some number of conflicting tasks, which are reassigned subsequent to assigning the formerly unassignable task, the idea being that there might exist a new feasible assignment for the retracted tasks where they are shifted somewhat in time or assigned to an alternate resource. The algorithm recurses on any retracted task that cannot be nondisruptively reassigned, and returns successfully when all visited tasks are assigned, or with failure when all tasks con-

[3]We also implemented various hill-climbing algorithms, employing several simple swap operators as well as a "temperature descent swap" [Globus *et al.*, 2004] operator, but found these techniques to perform rather poorly when running for a limited number of iterations (50).

tending for the same set of alternate resources have been considered.[4] For a complete algorithmic description of this basic *TaskSwap* procedure, we direct the reader to [Kramer and Smith, 2003], where the procedure is named *MissionSwap*.

TaskSwap localizes its search for a solution that enforces the two rules with respect to hard priority constraints, by inserting new unassignable tasks into the schedule without unscheduling any higher priority tasks in the process. In fact, *TaskSwap* can be seen to take an overly conservative approach to enforcing the priority constraint; since it will never unschedule a task that is in its input schedule, it can actually miss opportunities for an improved solution that might result from substituting one task of a given priority for another task of the same priority.

Noticing this deficiency, and also to take advantage of the more general objective function approximation of the priority constraint defined earlier, we define a less constraining version of *TaskSwap*, where an unassignable u is considered successfully inserted and the new solution is accepted even when some initially scheduled tasks of lower priority than u are not reinserted back into the schedule, if the sum of the scoring values for these tasks is lower than the scoring value of u. We call this version *PriorityTaskSwap*.

Property 1 *If the initial schedule does not violate the two hard priority constraint rules, then the solutions found by* PriorityTaskSwap *also preserve the rules.*

Proof: The rules specify that a solution satisfies the hard priority constraint if no more tasks of a certain priority class can be inserted in the schedule (even at the expense of bumping out some set of lower priority tasks), starting with the highest priority class. Like the original *TaskSwap* procedure, *PriorityTaskSwap* inserts unassignables into the schedule in decreasing priority order, and by definition a higher priority task is never unscheduled to fit in lower priority tasks. Note that *PriorityTaskSwap* only preserves the rules, but it can not prove that the rules are enforced in the initial schedule.[5]

Property 2 *The final schedule produced by the new version of* PriorityTaskSwap *can never result in a worse (higher) priority score than the initial schedule.*

Proof: Inserting a high priority unassignable into the initial schedule might result in bumping some of the lower priority scheduled tasks. Comparing the initial and final set of unassigned tasks, there are some common tasks, and then there are tasks in the initial set which got inserted into the schedule and others that were bumped and are present in the final set. The sum of the scoring values for the inserted unassignables is always higher than the sum of the scoring values of the bumped (unscheduled) lower priority tasks. This implies that at the end of *PriorityTaskSwap* the priority score must be lower or at most equal to the priority score of the initial input schedule.

4.3 Squeaky Wheel Optimization (SWO)

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

In our scheduling context, the "trouble maker" elements are unassigned tasks. During each iteration, we examine the schedule and identify the unassignable tasks. We associate a move distance with each unassignable task, which indicates how far forward in the scheduling order this task will move for the next iteration. A task's move distance is based on the numerical value of its priority. More precisely, for unassignable u we define $dist(u) = lower\_bound + positions(u)$ where $lower\_bound$ is a lower bound on the move and $positions(u)$ is a function that defines the specific number of positions in the schedule order to move u beyond the $lower\_bound$.

5 Experimental Design

For our experiments with *PriorityTaskSwap* and *SWO*, we used the 100-problem data set of mission scheduling problems for the AMC domain generated in [Kramer and Smith, 2005a]. Each problem instance consists of 1000 missions (tasks) that must be scheduled across 14 airwings with variable capacity. The 100 problems are divided into 5 sets of 20 problems, each set being progressively more resource constrained. For each problem an initial schedule is generated. From this initial schedule, *Priority TaskSwap* and *SWO* are applied with the set of remaining unassigned tasks to improve the schedule with respect to *priority score*.

For *SWO* we found empirically that moving unassignable tasks forward for only a short distance (50 to 100 positions) did not perform well. Setting the distance to around 200 resulted in best performance. Therefore, we define $lower\_bound$ to be 200 and instantiate $dist(u)$ as $200 + \text{numeric\_priority(u)}/10 + \text{numeric\_priority(u)} \bmod 10$, where numeric_priority(u) is the numeric priority we associate with the alphanumeric priority of u (e.g., 555 for 1A1 and so on). We also experimented with stochastic versions of the priority to distance mapping, but were unable to achieve better results.

For each problem, *PriorityTaskSwap* is run to completion, and *SWO* is run until an iteration limit of 50 is reached or the procedure finds an optimal solution (0 unassignable tasks). Run-times in seconds, the best *priority score*, and the corresponding number of unassigned tasks for that score are recorded.

6 Experimental Results

A first performance indicator we examine is the runtime (see Figure 1). *SWO* takes significantly longer to run than *TaskSwap* because building the schedule in the AMC domain is expensive (it takes around 5 seconds to build the schedule given its scale and complexity)[6]. What turned out to still be comparable with the *TaskSwap* runtime is the time until the best solution is found by *SWO*. Most of the time *SWO*

[4]It is not hard to prove then that the worst-case time to consider each unassignable task is n^2 in the number of contending tasks.

[5]We believe that an exact algorithm to actually prove that the rules are enforced would be too expensive for AMC scheduling.

[6]We rebuild the whole schedule in each iteration; a possible way to speed up *SWO* is to start each iteration with a partial schedule containing the tasks that are not affected by the current move.

finds improvements early on in the search and then levels off. Longer runs of *SWO* (200 iterations and more) for selected problems showed that no significant improvement can be obtained beyond 50 iterations. This is very similar to results reported in [Barbulescu, 2006] for satellite scheduling, where *SWO* also found most of the improvements to the initial solution early in the run.

Next, we compare *SWO* and *TaskSwap* in terms of the priority score and number of unassignables. We find that the two algorithms perform similarly for the first four sets of problems. A Wilcoxon matched-pairs signed-ranks test shows that the results are not significantly different for these problems. *SWO* is able to find three new optimal feasible solutions, for problem 7 in set 2 and problems 4 and 20 in set 3. However, for the problems with high levels of oversubscription (in set 5), a Wilcoxon matched-pairs signed-ranks test shows that *TaskSwap* finds significantly better solutions in terms of number of unassignables. *TaskSwap* is able to make progress on the harder problems by selectively focusing on a relatively small subset of tasks, for which it is productive to temporarily relax the priority constraint, and concentrating on moving these tasks. In contrast, *SWO* is unable to find the appropriate combination of tasks to move in order to achieve comparable results.

While *TaskSwap* focuses on which tasks to move around in order to improve the solution, *SWO* is unconstrained in terms of what it can move around in the schedule. We conjecture that this flexibility of *SWO* is the reason it finds new optimal solutions for problems with low levels of oversubscription and at the same time, this is what hurts its performance for moderate to high levels of oversubscription. This conjecture is supported by our schedule stability results.

Finally, to assess schedule stability, we compare initial and final schedules for each run in terms of: number of tasks that shifted in time (Figure 5) and number of tasks that changed resource (Figure 4). As expected, *SWO* performs poorly in terms of schedule stability. When the level of oversubscription is low, there is also more freedom in rearranging the schedule to fit more of the initial unassignables in. *SWO* takes advantage of this: it changes the start time for as many as 700 tasks on average for the problem set 2 and reassigns resources to more than 160 tasks for the same problem set (see Figures 4 and 5). For high levels of oversubscription, there isn't much flexibility in rearranging tasks in the schedule (since the initial schedule is a pretty good one). *SWO* moves a lot fewer tasks for problem sets 4 and 5 then it does for 2 and 3, and it can not find the right set of moves to perform as well as *TaskSwap*.

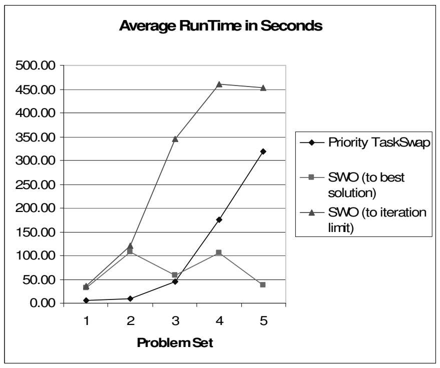

Figure 1: Computational Cost

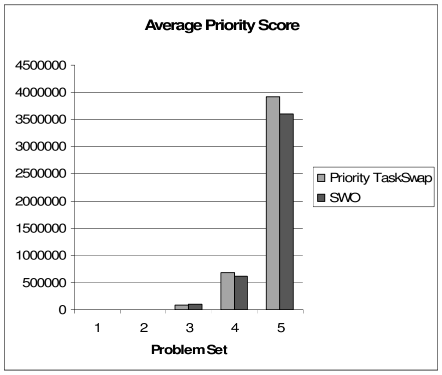

Figure 2: Solution Quality: Priority Score

7 Final Remarks

Implementing a permutation-based algorithm that respects the task priority constraints is a challenging proposition. Our approach in this paper has been to define an objective function that encodes the priority constraint - the difficulty though is the inherent problem in guaranteeing that priority is enforced while giving the search sufficient flexibility to progress. If the priority constraint is de-emphasized, SWO finds solutions with fewer unassignables but at the expense of greater violation of the task priority rules. Alternatively, the greater the emphasis on enforcing priority (by emphasizing differences between priority classes), the poorer the performance of *SWO*. Another approach might be to enforce priority in the schedule builder - e.g., processing the permutation in multiple passes by priority class. However, it is not really clear how this approach would improve the flexibility of the search.

Overall, we find two main results with respect to choice of representation for solving oversubscribed scheduling problems that require task priority to be enforced as a hard constraint. First, for low levels of oversubscription, a permutation-based search technique is able to take advantage of the many possibilities that exist to rearrange currently scheduled tasks to incorporate new unassigned tasks into the schedule; for such problems, a permutation-based representation can work as well (or even outperform) a repair-based technique. Second, for moderate to high levels of oversubscription, there is little room to move tasks around in the schedule; a technique like *TaskSwap* that focuses exactly on which tasks would need to be reassigned has a much better chance of finding improvements than a permutation-based technique, which lacks such guidance.

References

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

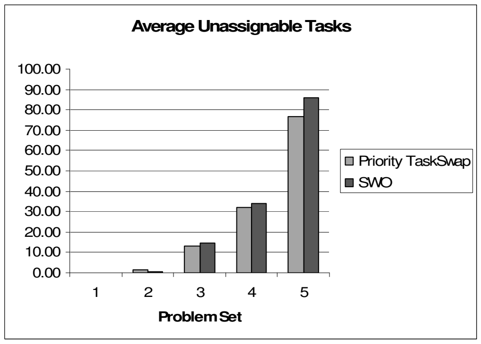

Figure 3: Solution Quality: End Number of Unassignable Tasks

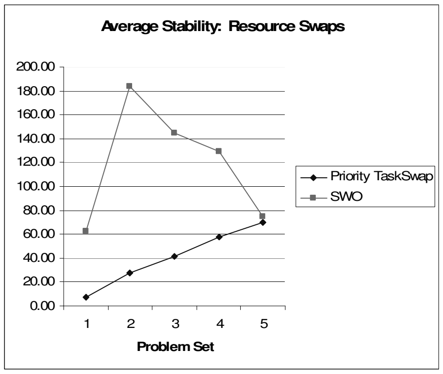

Figure 4: Solution Stability: Number of Resource Swaps

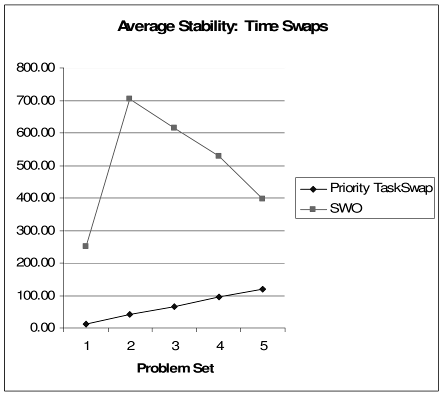

Figure 5: Solution Stability: Number of Time Swaps

[Barbulescu *et al.*, 2006] L. Barbulescu, A. Howe, and L.D. Whitley. AFSCN scheduling: How the problem and solution have evolved. *MCM*, 43:1023–1037, 2006.

[Barbulescu, 2006] L. Barbulescu. Understanding algorithm performance on an oversubscribed scheduling application. *To Appear in JAIR*, 2006.

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

[Globus *et al.*, 2002] A. Globus, J. Crawford, J. Lohn, and R. Morris. Scheduling earth observing fleets using evolutionary algorithms: Problem description and approach. In *Proc. of the 3rd Int'l NASA Workshop on Planning and Scheduling for Space*, pages 27–29, October 2002.

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

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

[Joslin and Clements, 1999] David E. Joslin and David P. Clements. "Squeaky Wheel" Optimization. In *JAIR*, volume 10, pages 353–373, 1999.

[Kramer and Smith, 2003] L.A. Kramer and S.F. Smith. Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems. In *Proc. 18th Int'l Joint Conf. on AI*, Acapulco Mexico, August 2003.

[Kramer and Smith, 2004] L. A. Kramer and S. F. Smith. Task swapping for schedule improvement, a broader analysis. In *Proc. 14th Int'l Conference on Automated Planning and Scheduling (ICAPS-04)*, Whistler BC, June 2004.

[Kramer and Smith, 2005a] L. A. Kramer and S. F. Smith. The amc scheduling problem: A description for reproducibility. Technical Report CMU-RI-TR-05-75, Robotics Institute, Carnegie Mellon University, 2005.

[Kramer and Smith, 2005b] L. A. Kramer and S. F. Smith. Maximizing availability: A commitment heuristic for oversubscribed scheduling problems. In *Proc. 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, Monterey CA, June 2005.

[Simon, 1981] H.A. Simon. *The Sciences of the Artificial, 2nd Edition*. The MIT Press, 1981.

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

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