Analyzing Basic Representation Choices in Oversubscribed Scheduling Problems

Laurence A. Kramer, Laura V. Barbulescu and Stephen F. Smith
The Robotics Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213, USA
{lkramer,laurabar,sfs}@cs.cmu.edu

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 is 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 making 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.

Keywords: Heuristic Search, Local Search, Meta-heuristic Search.

1 Introduction

In The Sciences of the Artificial [14], 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 search techniques to navigate a complex space, it has often proven fruitful to employ knowledge about the contours of the space in the search process. It is generally accepted in AI that choosing the right representation for a problem can result in search spaces in 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.[1] have shown a permutation-based genetic algorithm to outperform a schedule repair based technique in the context of scheduling groundstation access for satellite communications. Globus et al.[6, 5] 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.

In this paper, we consider the basic question of representation choice in the context of another oversubscribed scheduling domain, the US Air Force Air Mobility Command (AMC) scheduling problem [11]. 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 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 [9, 10]. 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 better 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 to move things around, and the permutation-based technique fails to identify them.

2 Basic Representational Choices

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 [16, 15, 6, 1]. The advantage of using such an indirect representation is that a wide range of general-purpose search algorithms (from heuristic methods to exact, tree search methods) can be employed to search the permutation space, while all the particular constraints of the domain are encapsulated in a schedule builder. As a downside, in general, one can not predict the effect of a change in the current solution (permutation) until the schedule builder computes the new schedule. Also, depending on the schedule builder, the set of 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. 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 will affect the corresponding schedule).

Searching the schedule space directly can be an attractive alternative ([7, 3, 9, 10]) to permutation space search. 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 typically exploit domain knowledge to decide how to reorganize the schedule. The challenge is in defining the right search operators.

3 The AMC Scheduling Domain

The AMC scheduling problem considered in this paper abstracts the large airlift and tanker mission scheduling problem faced by the US Air Force Air Mobility Command (USAF AMC). Drawing from [11], the problem is briefly summarized as follows:

- 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 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 de-positioning 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.

3.1 AMC Task Priority as a Hard Constraint

In the AMC domain tasks are scheduled strictly by priority. Furthermore, it is not normal procedure to "trade off" or bump a higher priority mission to get some number of lower priority missions into the schedule. There are five major priority classes: 1 through 5, with 1 the highest priority class.

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. Subsequent (lower priority) tasks can still be feasibly assigned if resources are 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 some tasks that have been forced to be excluded.

A solution (schedule) S respects the hard priority constraint if the following two rules hold:

1. There exists no feasible solution $S^{'}$ that contains a superset of the priority 1 tasks that are contained in S. In other words, there is no $S^{'}$ where all priority 1 tasks that are assigned to resources in S *and* one or more additional priority 1 tasks (unassigned in S) are assigned.[1]
2. For any $k = 1..4$, there exists no $S^{'}$ that contains exactly the same set of assigned tasks at priority levels 1 through k that are contained in S and a superset of the priority $k+1$ tasks that are contained in S.

[1]This does not imply that $S^{'}$ 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.

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 $k = 0..4$.

3.2 Enforcing the Hard Priority Constraint

In practice, given that the problem we consider is $\mathcal{NP}$-complete [4], 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 precluded by the new objective function.

More specifically, we resort to using a heuristic *scoring value* for each priority class, that emphasizes the differences between classes: priority 5 maps to 1, priority 4 to 1,000, priority 3 to 1,000,000, and so on. We define the *penalty score* for a schedule as the sum of the scoring values for all the unassignables. The new objective function minimizes the penalty score. Since the number of tasks we consider is less than 1,000, by using a factor of 1,000 between successive priority classes, we ensure that a higher priority task couldn't possibly be swapped out to make way for any number of lower priority tasks and still achieve a better (lower) penalty score.

4 Algorithms

Previous research has found *TaskSwap*, a repair-based algorithm, to perform well on the AMC scheduling problem [9, 10, 12]. 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 space have been implemented for oversubscribed scheduling domains. One stands out because of its similarity with *TaskSwap*: Squeaky Wheel Optimization (SWO) [8]. *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 groundstation access for satellite communications [2]. Hence we choose *SWO* as our representative permutation-space search procedure, defined in Section 4.3. We also implemented various hill-climbing algorithms, employing several simple swap operators as well as a "temperature descent swap" [6] operator, but found these techniques to perform rather poorly when running for a reasonable number of iterations.

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 [12]. 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 [9, 10] implements a repair-based approach to rearranging tasks in an input schedule so as to include additional, 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. The retracted tasks are reassigned subsequent to assigning the formerly unassignable task, the idea being that there might exist a new feasible schedule where retracted tasks are shifted somewhat in time or assigned to an alternate resource. The algorithm recurses on any retracted task that cannot be reassigned, and returns successfully when all visited tasks are assigned, or with failure when all tasks contending for the same set of alternate resources have been considered. For a complete algorithmic description of this basic *TaskSwap* procedure, we direct the reader to [9].

TaskSwap localizes its search for a solution that enforces the 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 task 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*. It is easy to see that the following property holds:

Property 1 *The final schedule produced by the new version of* PriorityTaskSwap *can never result in a worse (higher) penalty score than the initial schedule.*

4.3 Squeaky Wheel Optimization (SWO)

SWO [8] repeatedly iterates through a cycle composed of three phases. First, a scheduling order of the elements in the problem is defined and based on this a greedy solution is built. Then, the solution is analyzed and the elements causing "trouble" are ranked based on their contribution to the objective function. Third, the ranking of such "trouble makers" is used to move them earlier in the scheduling order. The cycle is repeated until a termination condition is met.

In our scheduling context, we build the initial greedy solution based on the priorities associated with the elements in the problem. The "trouble maker" elements are unassigned tasks. During each iteration, we examine the schedule and identify the unassignable tasks. Based on its priority, 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.

5 Experimental Design

For our experiments with *PriorityTaskSwap* and *SWO*, we used the 100-instance data set of mission scheduling problems for the AMC domain which is described in [11]. Each problem instance consists of approximately 1000 missions (tasks) that must be scheduled across 14 air wings (resources) with variable capacity. The 100 instances are divided into 5 sets of 20, each being progressively more resource constrained. For each instance an initial schedule is generated after which *PriorityTaskSwap* and *SWO* are run. For *SWO* we found empirically that moving unassignable tasks

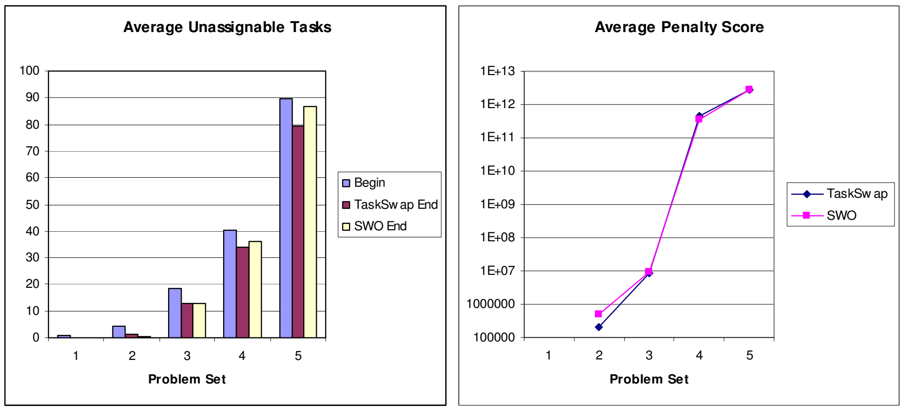

Figure 1: Solution Quality: Number of Unassignable Tasks and Penalty Score

forward for only a short distance (50 to 100 positions) did not perform well. Setting the distance to around 200 resulted in best performance. We defined the distance to move forward an unassignable u as: $200 + (10 \times \mathit{numeric\_priority}(u))$, where $\mathit{numeric\_priority}(u) = 6 - \mathit{priority\_class}(u)$.[2] For each instance, *PriorityTaskSwap* is run to completion, and *SWO* is run until an iteration limit of 50 is reached[3] or the procedure finds an optimal solution (0 unassignable tasks).

6 Experimental Results

First, we compare *SWO* and *TaskSwap* in terms of the penalty score and end number of unassignables (Figure 1). We find that the two algorithms perform similarly for the first four sets of problems. A Wilcoxon signed-rank test [13] shows that the results are not significantly different for these problems. *SWO* is able to find two new optimal solutions (zero unassignable tasks), for problems 4 and 20 in set 3 [11]. However, for the problems with high levels of oversubscription (in set 5), a Wilcoxon signed-rank 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.

To assess schedule stability, we compare initial and final schedules for each run in terms of: number of tasks that shifted in time and number of tasks that changed resource (Figure 2). 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 in more of the initial unassignables. *SWO* takes advantage of this: on average it changes the start time for as many as 700 tasks for

[2]For example, a priority-1 unassignable task will be move forward 250 positions, and a priority-2 task 240 positions.

[3]SWO typically found its best solution after a few iterations, and running more than 50 iterations produced no further improvement. We also experimented with stochastic versions of the priority to distance mapping, but were unable to achieve better results.

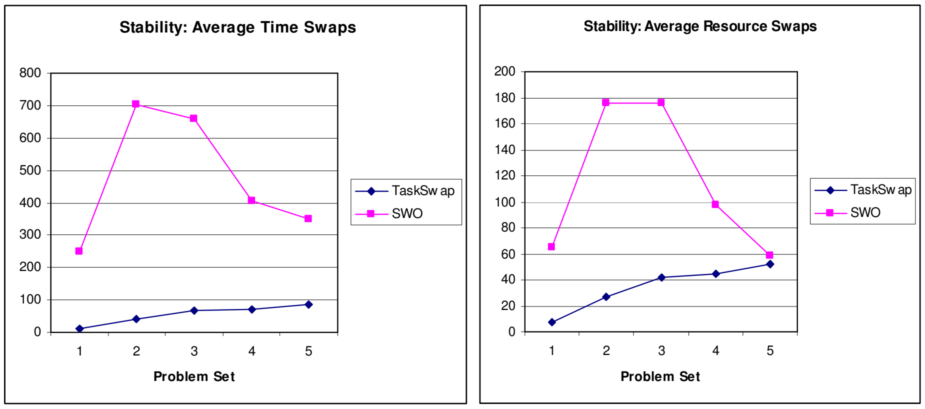

Figure 2: Solution Stability: Number of Time and Resource Swaps

the problem set 2 and reassigns resources to more than 170 tasks for that set. For high levels of oversubscription, there isn't much flexibility in rearranging tasks in the schedule (since the initial schedule is so packed). *SWO* moves a lot fewer tasks for problem sets 4 and 5 than it does for 2 and 3, and it cannot find the right set of moves to perform as well as *TaskSwap*.

7 Final Remarks

Implementing a permutation-based algorithm that respects task priority constraints is a challenging proposition. Our approach in this paper has been to define an objective function that encodes the priority constraint as a numeric sum, that can only be improved by assigning more tasks of higher priority. The difficulty though is the inherent problem of guaranteeing that priority is enforced while giving the search sufficient flexibility to progress.

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.

Acknowledgements

This research was supported in part by the USAF Air Mobility Command under Contract # 7500007485 to Northrop Grumman Corporation, by the Department of Defense Advance Research Projects Agency (DARPA) under Contract # FA8750-05-C-0033, and by the CMU Robotics Institute. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the USAF or DARPA.

References

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

[2] L. Barbulescu, L. D. Whitley, and A.E. Howe (2004), Leap before you look: An effective strategy in an oversubscribed problem. In *Proc. 19th National Artificial Intelligence Conference.*

[3] 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 (2000), ASPEN - Automating space mission operations using automated planning and scheduling. In *6th International SpaceOps Symposium (Space Operations)*, Toulouse (France).

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

[5] A. Globus, J. Crawford, J. Lohn, and R. Morris (2002), 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.

[6] A. Globus, J. Crawford, J. Lohn, and A. Pryor (2004), 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.

[7] M. D. Johnston and G. Miller (1994), Spike: Intelligent scheduling of hubble space telescope observations. In M. Zweben and M. Fox, editors, *Intelligent Scheduling.* Morgan Kaufmann.

[8] D. E. Joslin and D. P. Clements (1999), "Squeaky Wheel" Optimization. *JAIR*, 10:353–373.

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

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

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

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

[13] R. Ott and M. Longnecker (2000) *An Introduction to Statistical Methods and Data Analysis, 5th Ed.* Duxbury Pr.

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

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

[16] L. D. Whitley, T. Starkweather, and D. Fuquay (1989), 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.