Understanding Performance Tradeoffs in Algorithms for Solving Oversubscribed Scheduling

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

Abstract

In recent years, planning and scheduling research has paid increasing attention to problems that involve resource oversubscription, where cumulative demand for resources outstrips their availability and some subset of goals or tasks must be excluded. Two basic classes of techniques to solve oversubscribed scheduling problems have emerged: searching directly in the space of possible schedules and searching in an alternative space of task permutations (by relying on a schedule builder to provide a mapping to schedule space). In some problem contexts, permutation-based search methods have been shown to outperform schedule-space search methods, while in others the opposite has been shown to be the case. We consider two techniques for which this behavior has been observed: TaskSwap (*TS*), a schedule-space repair search procedure, and Squeaky Wheel Optimization (*SWO*), a permutation-space scheduling procedure. We analyze the circumstances under which one can be expected to dominate the other. Starting from a real-world scheduling problem where *SWO* has been shown to outperform *TS*, we construct a series of problem instances that increasingly incorporate characteristics of a second real-world scheduling problem, where *TS* has been found to outperform *SWO*. Experimental results provide insights into when schedule-space methods and permutation-based methods may be most appropriate.

Introduction and Motivation

As research in automated planning and scheduling has moved into problem domains that more accurately model real-world concerns, one issue that has garnered increasing interest has been that of oversubscription (Kramer & Smith 2004; Barbulescu *et al.* 2006; Smith 2004; Nigenda & Kambhampati 2005). Generally speaking, an oversubscribed problem is one in which the resources available (e.g., time, capacity) are not sufficient to permit accomplishment of all stated tasks or goals, and hence the problem solver must decide which subset of tasks or goals to carry out. The basic objective is to maximize the number of tasks accommodated or goals satisfied, subject in some cases to associated task or goal priorities. Oversubscribed problems arise in a broad range of application domains, including rover task planning (Smith 2004; Joslin *et al.* 2005), satellite and telescope scheduling (Bresina 1996; Frank *et al.* 2001; Barbulescu *et al.* 2006) and military airlift allocation (Kramer & Smith 2004).

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

With respect to solving oversubscribed scheduling problems, two basic classes of solution techniques have emerged: those that search directly in the space of possible schedules, and those that search in an alternative space of task permutations (in which case a schedule builder is used to provide a mapping to schedule space). Both permutation-space and schedule-space methods have been shown to perform effectively in specific problem domains. This raises the question of whether there are problem characteristics that might suggest the appropriateness of one over the other.

In this paper, we attempt to gain insight into this general question by analyzing the performance tradeoffs between two specific methods on a common set of problem instances. Our starting point is two oversubscribed scheduling problems that are quite similar in character, the USAF Satellite Control Network (AFSCN) problem previously studied in (Barbulescu *et al.* 2006) and the USAF Air Mobility Command (AMC) airlift scheduling problem described in (Kramer & Smith 2005a). Prior research with the AFSCN problem has shown permutation-space scheduling procedures such as Squeaky Wheel Optimization (*SWO*) to dominate schedule-space methods. Other prior work (Kramer & Smith 2004; 2005b) has demonstrated the effectiveness of a schedule-space method called TaskSwap (*TS*) in solving the AMC scheduling problem, and in fact *TS* can be shown to outperform *SWO* in this domain. Given these results, we attempt to understand what problem characteristics set these domains and solution techniques apart. We define a series of problem sets which generalize from the AFSCN problem and increasingly incorporate characteristics of the AMC problem. Our experimental results indicate that problem hardness and the presence or absence of task priorities are two distinguishing performance factors.

Before describing the AFSCN and AMC domains and presenting our experimental analysis, we briefly summarize prior approaches to oversubscribed scheduling problems and the two specific methods of interest to our study.

Permutation Space vs. Schedule Space Search

As indicated above, we can distinguish two basic classes of approaches to solving oversubscribed scheduling problems based on the search space representation that is used.

Permutation-based methods emphasize search in the space of task permutations, where a given permutation specifies a scheduling order and is transformed into an actual schedule by a "schedule builder". Search in the permutation space has been effectively employed in many scheduling applications (Whitley, Starkweather, & Fuquay 1989; Syswerda 1991; Globus *et al.* 2004; Barbulescu *et al.* 2006). The main advantage of a permutation representation is that general-purpose search algorithms can be employed while all the particular constraints of the domain are encapsulated in a schedule builder. The main disadvantage is that in general it is not possible to predict the effect of permutation changes until the schedule builder computes the new schedule. Also, the schedules reachable via the schedule builder might represent a suboptimal subset of all possible schedules.

A second class of techniques, referred to generally as repair-based search techniques, operate directly in the space of possible schedules (Johnston & Miller 1994; Rabideau *et al.* 1999; Kramer & Smith 2004). Searching the schedule space directly is sometimes an attractive alternative. Powerful domain-specific heuristics are usually available (e.g., various resource contention measures), and such measures can direct the search in an efficient but effective manner. In continuous domains where schedule stability is important, search operators for the schedule space can be defined to generate new solutions in a controlled manner (and minimize changes to 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.

Squeaky Wheel Optimization

The permutation-based method we consider in this paper incorporates Squeaky Wheel Optimization (*SWO*) (Joslin & Clements 1999) as the core search procedure. *SWO* has been used effectively in a number of oversubscribed domains (Globus *et al.* 2004; Joslin *et al.* 2005; Frank & Kürklü 2005; Barbulescu *et al.* 2006). Once an initial permutation (scheduling order) of the input tasks to be scheduled is established, *SWO* proceeds by repeatedly iterating through a three step cycle. In the first step, a greedy constructive technique (the schedule builder) uses the permutation to produce an actual solution (schedule). The permutation represents a prioritization of the tasks, since the earlier tasks are considered earlier by the schedule builder. In the second step, the solution is analyzed and those tasks causing "trouble" (i.e., those tasks that the schedule builder was not able to get on the schedule) are ranked according to their contribution to the objective function. Finally in the third step, the ranking of current "trouble makers" is used to modify the permutation, by moving them earlier in the scheduling order. This cycle is repeated until a termination condition is met.

For the analysis performed in this paper, the schedule builder used by *SWO* places tasks into the schedule one by one (in the order specified by the current permutation), using a look-ahead heuristic based on predicted resource contention to assign a specific start time and resource to each task. This heuristic, *max-availability*, is described in some detail in (Kramer & Smith 2005b). If a given task cannot be feasibly added to the schedule at the time it is considered, it is marked as unassignable and the schedule builder simply moves on to the next task. This schedule builder is also employed to generate initial seed solutions for the other method analyzed in this paper, TaskSwap (see below).

TaskSwap

The specific schedule-space search method we consider in this paper is TaskSwap (*TS*) (Kramer & Smith 2004). *TS* implements a repair-based search aimed at rearranging tasks in an input schedule so as to include additional, previously unassignable tasks. The algorithm considers unassignable tasks one by one (according to some ordering criteria) and attempts to insert them into the existing schedule by (temporarily) retracting some number of conflicting tasks. Any retracted tasks are reassigned after assigning the formerly unassignable task, the idea being that there might exist a new feasible schedule where these retracted tasks are shifted somewhat in time and/or assigned to an alternate resource. The algorithm recurses on any retracted task that cannot be reassigned, and returns successfully when all visited tasks have been assigned, or with failure when all tasks contending for the same set of alternate resources have been considered. In the event of failure, the schedule in place prior to the attempted introduction of the new task is restored, and the next unassignable task is considered.

A stochastic neighborhood search – Value Biased Stochastic Sampling (VBSS) (Cicirello & Smith 2002) – applied to the retraction heuristic was shown to boost *TS* performance for the AMC problem sets (Kramer & Smith 2004). The results presented for *TS* in this paper were obtained by running *TS* with multiple iterations of VBSS.

Comparative Performance in Two Domains

We first contrast the performance of *SWO* and *TS* on problems drawn from two real-world oversubscribed scheduling domains, the USAF Satellite Control Network (AFSCN) scheduling problem (Barbulescu *et al.* 2006) and the USAF Air Mobility Command (AMC) airlift scheduling problem (Kramer & Smith 2005a). In the AFSCN domain, input communication requests for Earth orbiting satellites must be scheduled on a total of 16 antennas spread across 9 ground-based tracking stations. In the AMC domain, aircraft capacity from 15 geographically distributed air wings must be allocated to support an input set of airlift missions.

Despite the application differences, these two domains share a common core problem structure:

- A problem instance consists of n tasks. In AFSCN, the tasks are communication requests; in AMC they are mission requests.
- Each task T_i, $1 \leq i \leq n$, specifies a required processing duration T_i^{Dur}.[1]
- A set Res of resources are available for assignment to tasks. Each resource $r \in Res$ has capacity $cap_r \geq 1$. The

[1] Although, for AMC, the actual durations are resource-dependent.

resources are air wings for AMC and ground stations for AFSCN. The capacity in AMC corresponds to the number of aircraft for that wing; in AFSCN it represents the number of antennas present at the ground station.

- Each task T_i has an associated set Res_i of feasible resources, any of which can be assigned to carry out T_i. Any given task T_i requires 1 unit of capacity (i.e., one aircraft in AMC or one antenna in AFSCN) of the resource r that is assigned to perform it.
- Each of the feasible alternative resources $r_j \in Res_i$ specified for a task T_i defines a time window within which the duration of the task needs to be allocated. This time window corresponds to satellite visibility in AFSCN and mission requirements for AMC.
- The basic objective is to minimize the number of unassigned tasks.

One principal difference between the domains is the issue of task priority. In the AFSCN domain there is no explicit notion of priority and all tasks are weighted equally. In the AMC domain, alternatively, tasks (missions) are categorized into one of five major priority classes, and task priorities must be respected whenever scheduling tradeoffs are considered - i.e., it is not possible to substitute a lower priority task for a higher priority task even if this choice enables additional lower priority tasks to be inserted into the schedule. This places an additional constraint on the basic objective of minimizing the number of unassigned tasks.

Consideration of the benchmark problem sets that have been published for each of these domains reveals a few additional differences:

- The size of the AFSCN instances varies between 419 and 483 tasks, while the size of the AMC problem instances is more than double (983 missions).
- Resource capacity for AFSCN varies between 1 and 3; for AMC, it varies between 4 and 37.
- Degree of temporal flexibility (measured as task duration relative to the size of the resource time windows): for AFSCN, approximately one half of the requests in a given problem instance have no temporal flexibility (these are communication requests for low altitude satellites); for the AMC benchmark problems, temporal flexibility is present for all tasks.

Even though the two domains are similar in many ways, their differences somehow have an impact on solving performance. In the subsections below we show that what works well for one domain does not work as well on the other.

AFSCN Empirical Results: *SWO* Outperforms *TS*

Previous work has shown that permutation space search techniques (including *SWO*) clearly outperform repair-based search in solving AFSCN problem instances; for our comparative AFSCN experiments we closely follow the methodology reported in (Barbulescu *et al.* 2006). Considering only the five days of data in the more difficult R1 through R5 problems, we build an initial schedule starting with a task order based on most constrained (least available slack) first, sub-sorted by earliest start time first, sub-sorted by smaller number of resource alternatives first.

Problem	Initial Unassign.	End SWO	End TS
R1	58	45	49
R2	38	30	34
R3	27	18	20
R4	37	28	32
R5	19	13	15

Table 1: Performance of *SWO* and *TS* on AFSCN scheduling. The second column indicates the number of unassignable tasks in the initial schedule.

During each *SWO* iteration, we examine the schedule and identify the unassignable tasks. We move the unassignable tasks forward in the permutation by a distance of five (this is consistent with the *SWO* setup described in (Barbulescu *et al.* 2006), for which the best *SWO* performance on AFSCN has been reported).

The *TaskSwap* procedure is brought to bear on the same initial schedule for each problem as *SWO*, and it is run to completion. As *TS* attempts (in the permutation order since there is no notion of priority) to assign each unassignable task from the initial schedule, it makes only moves that maintain the state of already assigned tasks. That is, it is not free to terminate in a state where one task is de-assigned in order to assign two others. This can be seen as an unfair restriction on *TS*, but is fairly central to its design, which emphasizes schedule stability.

The results of running *SWO* for 500 iterations, and *TS* for one iteration show (Table 1) that for each problem *SWO* is able to assign more tasks than *TS*[2].

AMC Empirical Results: *TS* Outperforms *SWO*

For the AMC scheduling problem, *TS* has been applied quite effectively. *TS* by definition enforces the domain's priority constraint (since a lower priority unassignable task can never be substituted for an assigned higher priority task). To ensure that the priority constraint is also enforced by *SWO*, we define a new objective function. We first specify 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 then define the *penalty score* for a given schedule to be the sum of the scoring values for all unassignables. The new objective function minimizes the penalty score. Since the number of tasks in the AMC instances is less than 1,000, the objective function ensures that the substitution of any number of lower priority tasks for a higher priority task will result in a schedule with a greater penalty score.

We build the initial greedy solution for both *SWO* and

[2]Note that our schedule builder is slightly different from the greedy scheduler in (Barbulescu *et al.* 2006). While the values produced by SWO with this schedule builder are somewhat worse than the ones reported in (Barbulescu *et al.* 2006), it is still the case that *S*WO outperforms *T*S for these problems.

TS based on a priority sorted task permutation. We found empirically that moving unassignable tasks forward in the permutation only a short distance (50-100 positions) did not perform well, and that setting the move distance to around 200 resulted in best performance. Biasing this base distance according to priority class of the unassignable u, move distance was defined as $md(u) = 200 + (10 \times (6 - Pr(u)))$.[3]

To fully take advantage of the new objective function, we loosen the requirement for accepting schedule repairs in *TS*. Specifically, 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 into the schedule, if the sum of the scoring values for these tasks is lower than the scoring value of u.

Figure 1 compares the penalty scores obtained running *TS* (3 iterations of VBSS) with *SWO* (50 iterations) on each of the 5 sets of AMC benchmark problems; the average end number of unassignable tasks in the same runs are shown in Figure 2. A Wilcoxon signed-rank test (Ott & Longnecker 2000) shows that the average penalty scores are not significantly different for the first three sets of problems. However, at higher levels of oversubscription (problem sets 4 and 5) *TS* is seen to outperform *SWO*. With respect to penalty scores, a significant difference ($p = 0.0152$) is found for problem set 4.[4] With respect to average unassignables, significantly fewer are obtained for both problem sets 4 and 5 ($p < 0.01$).

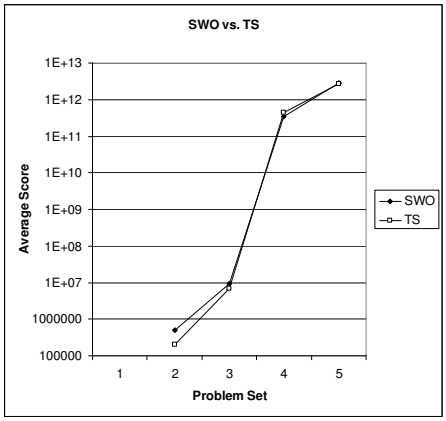

Figure 1: Average Penalty Score.

Exploring the AFSCN/AMC Problem Space

To better understand why *SWO* outperforms *TS* for AFSCN, but not for AMC, we identify problem features that are different in the two domains; starting with AFSCN-like problems we vary these features and generate new problems which sample the common AFSCN/AMC problem space. We design experiments to test two basic hypotheses:

1. Increasing the capacity and/or slack in AFSCN-like problems with no priority specified will result in *TS* performing better than *SWO*.
2. Priority constraints are better handled by *TS* than *SWO*, especially as the level of oversubscription increases.

[3]The 5 AMC priority classes range from 1 (highest) to 5.

[4]This difference is not apparent from the graph, due to the presence of outliers in the computed average scores.

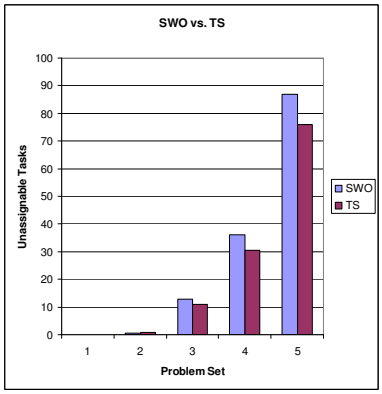

Figure 2: Average Unassignables.

Experimental Design

The instances in the AMC benchmark sets are larger in size and have more slack and resource capacity available than the AFSCN benchmark instances. Also, task priority is only present in the AMC problems. To investigate how these differences account for the observed difference in the performance of *TS* and *SWO*, we design a problem generator that produces new AFSCN-like instances with varying degrees of slack and resource capacity; AMC-like task priorities can also be included in the new instances.

For each of the five AFSCN benchmark problems (R1 to R5), the generator produces new problems based on its parameter settings, as follows:

- Problem size: For our experiments, we decided to either keep constant, double or triple the size of the initial AFSCN benchmark problems. When the problem size is kept constant, new problems are produced by moving each task's time window later in time by a uniform random choice over an hour time interval. When the size is doubled (or tripled), two (or three) new tasks are generated for each task in the original problem. The new tasks vary from the initial one in terms of time window and possibly duration.
- Slack (temporal flexibility): A duration factor df is used to determine the durations for each new task. Given a task $T_i, 1 \leq i \leq n$ with an initial duration T_i^{Dur}, the new duration is computed as: $T_i^{Dur} * (1 - random(df, 0))$, where $random(df, 0)$ produces a random number between df and zero. For example, if $df = 0.9$, the new task durations can vary anywhere between the initial duration and 10% of the value of the initial duration.
- Resource capacity: Given a resource r with capacity cap_r (in the initial AFSCN benchmark set), a capacity factor cf is used to compute the new capacity of r as: $cap_r + random(cf, 0)$.

Prob. Set	Avg. Size	Slack df	Capac. cf	Init.Sched.Unassignables $pf = false$	$pf = true$
1.1	443	0	0	34.1	71.2
1.2	886	0	3	127.7	195.6
1.3	1329	0	9	94.8	170.3
2.1	443	0.5	0	25.1	44.3
2.2	886	0.5	3	81.6	121.6
2.3	1329	0.5	9	56.12	106.6
3.1	443	0.5	3	7.4	15.7
3.2	886	0.5	6	27.3	48.9
3.3	1329	0.5	12	47	65.4
4.1	443	0.9	0	11.6	22.6
4.2	886	0.9	3	37.9	65.3
4.3	1329	0.9	9	32.3	45.4
5.1	443	0	5	4.04	13.5
5.2	886	0	8	34.9	69.0
5.3	1329	0	15	47.8	80.5
6.1	443	0.5	5	3.48	6.8
6.2	886	0.5	8	19.7	29.4
6.3	1329	0.5	15	36.8	44.7

Table 2: Description of the problem sets: the size is either similar to the initial AFSCN problems (*.1 sets), doubled (*.2 sets) or tripled (*.3 sets); df is the duration factor, cf is the capacity factor, and pf the priority flag. The average number of unassignables in a greedy initial solution computed for the 50 instances in each problem set is also recorded.

- Priority: A priority flag pf determines if task priorities are present in the problem. When pf is true, task priorities are uniformly sampled from 1..5 (following the five priority classes in AMC).

We generate 36 sets of problems, with 50 instances each. 18 of the sets are produced with no task priorities ($pf = false$), and the other 18 are identical but for the addition of task priorities ($pf = true$). The parameters used to generate the sets are shown in Table 2: the second column represents the average size of the problem instance, while the third and fourth columns represent the value of df and cf respectively. Note that problem set 1.1 with $pf = false$ contains the five initial AFSCN benchmark problems plus 45 similar instances (same size, slack and resource capacity, varying the time windows for each task). As a measure of the level of oversubscription in the instances for each set, we use the greedy constructor to build an initial schedule for the 50 instances in each set and record the average number of unassignables in columns five and six.

Experimental Results

For this newly generated problem set we conduct experiments with 500 iterations of *SWO* and 30 iterations of *TS/VBSS*. To investigate our first hypothesis, we focus on the results in terms of average number of unassignables for the problem sets without priorities (see Figure 3). We ordered the problems sets on the x axis in terms of the average initial number of unassignables (column five in Table 2), as a rough measure of the oversubscription level in each problem set. With a few exceptions we see that the two algorithms result

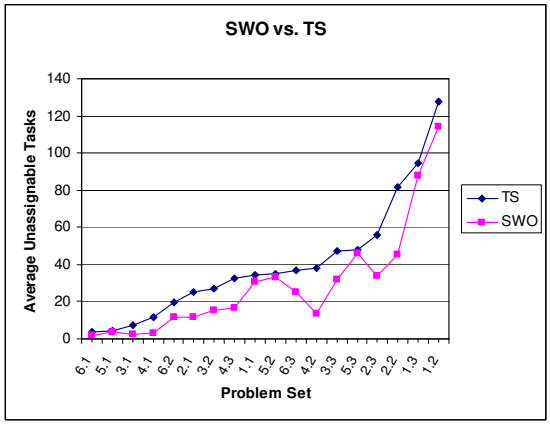

Figure 3: Average Penalty Score (lower values better) for problems without task priorities

Prob. Set	Score Diff.	Prob. Set	Score Diff.	Prob. Set	Score Diff.
6.1	**0**	6.3	0	**5.3**	**-3514**
5.1	0	2.1	0	1.1	-1003500
3.1	0	3.2	-3.5	2.3	-6009
4.1	**1**	3.3	0	2.2	-999880
6.2	0	4.2	500.5	**1.3**	**-9815700**
4.3	0.5	**5.2**	**-899500**	1.2	-1003805000

Table 3: Comparative Performance *SWO* vs. *TS*: the columns labeled Score Diff. are the median of the differences $(TS - SWO)$ of the penalty scores for all problems in the set. Negative values indicate problem sets where *TS* outperforms *SWO*. Bold numbers are statistically significant.

on average in a similar number of final unassignables for all problem sets. Our initial hypothesis was that *TS* would begin to outperform *SWO* as capacity and/or slack are increased. A Wilcoxon signed-rank test shows that *SWO* outperforms *TS* ($p < 0.01$) for all problem sets except 5.1 and 5.3. While these results do not confirm our hypothesis, there is some evidence (for example, sets 3.1, 5.1, 5.2, and 5.3) showing that *TS* begins to perform comparably with *SWO*[5] as slack is held constant and capacity is increased.

For our second hypothesis, we compare results in terms of penalty scores when task priorities are present. We present the results in tabular form (Table 3) since a graph of either average or median scores obscures the results due to a few outliers of very significant magnitude. Again, we ordered the problem sets based on the average initial number of unassignables (column six in Table 2). Columns two, four, and six in Table 3 show the median of the differences $(TS - SWO)$ of the penalty scores for all problems in the corresponding problem set. Negative numbers, then, indicate that *TS* outperforms *SWO* for that set. The results show that for moderate levels of oversubscription *SWO* and

[5]As measured by relative difference in end unassignable tasks normalized by initial number of unassignables.

TS perform similarly well, with *SWO* slightly outperforming *TS* for a few sets. In terms of statistical significance, a Wilcoxon signed-rank test shows that for the sets 4.1 and 6.1 *SWO* significantly outperforms *TS* (with $p \leq 0.002$); for the rest of the sets up to 5.2 there are no significant differences. As the problems become more oversubscribed, there is a crossover point (after problem set 5.2) and *TS* on average finds better solutions than *SWO*. The difference is significant ($p < 0.0001$) for problem sets 5.2, 5.3, and 1.3.

A Tale of Two Searches: What Works When and Why

Extrapolating from the above experimental results, we can draw a few conclusions relevant to the choice of permutation-space and schedule-space methods for solving oversubscribed scheduling problems. For problems that do not incorporate task priority, the search space is less constrained and the broader (and more disruptive) search process conducted by *SWO* provides greater opportunity to find better solutions. In contrast, repair-based searches such as *TS* suffer from a lack of strong heuristic guidance in this context and hence the more localized search that they carry out is not as effective.

In problems where task priority is in play, the situation is different. If problems tend to be only moderately oversubscribed, the additional constraint imposed on the search space by priority still leaves sufficient flexibility for techniques like *SWO* to reasonably find better solutions. However, as priority-based problems become more severely oversubscribed, the search space becomes increasingly constrained and rearrangement of task permutations become less productive. In this context, repair-based methods like *TS* benefit from the additional search space structure imposed by priority and gain in performance.

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

Barbulescu, L.; Howe, A.; Whitley, L.; and Roberts, M. 2006. Understanding algorithm performance on an oversubscribed scheduling application. *JAIR* 27:577–615.

Bresina, J. 1996. Heuristic-Biased Stochastic Sampling. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 271–278.

Cicirello, V., and Smith, S. 2002. Amplification of search performance through randomization of heuristics. In *Proc. 8th Int. Conf. on Principles and Practice of Constraint Programming*. Ithaca NY: Springer-Verlag.

Frank, J., and Kürklü, E. 2005. Mixed discrete and continuous algorithms for scheduling airborne astronomy observations. In *CPAIOR*, 183–200.

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

Globus, A.; Crawford, J.; Lohn, J.; and Pryor, A. 2004. A comparison of techniques for scheduling earth observing satellites. In *Proc. of the 19th National Conference on Artificial Intelligence (AAAI-04), 16th Conference on Innovative Applications of AI (IAAI-04)*, 836–843.

Johnston, M., and Miller, G. 1994. Spike: Intelligent scheduling of hubble space telescope observations. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. Morgan Kaufmann.

Joslin, D., and Clements, D. 1999. "Squeaky Wheel" Optimization. *JAIR* 10:353–373.

Joslin, D.; Frank, J.; Jonsson, A.; and Smith, D. 2005. Simulation-based planning for planetary rover experiments. In *Proc. of the 2005 Winter Simulation Conference*, 1049–1058. M.E. Kuhl, N.M. Steiger, F.B. Armstrong, and J.A. Joines, eds.

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

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

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

Nigenda, R., and Kambhampati, S. 2005. Planning graph heuristics for selecting objectives in over-subscription planning problems. In *Proceedings 15th Int'l Conference on Automated Plannign and Scheduling*, 192–201.

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

Rabideau, G.; Knight, R.; Chien, S.; Fukanaga, A.; and Govindjee, A. 1999. Iterative planning for spacecraft operations using the aspen system. In *Proc. 5th Int. Sym. on AI, Robotics and Automation for Space*.

Smith, D. 2004. Choosing objectives in over-subscription planning. In *Proceedings 14th International Conference on Automated Planning and Scheduling*, 393–401.

Syswerda, G. 1991. Schedule Optimization Using Genetic Algorithms. In Davis, L., ed., *Handbook of Genetic Algorithms*. NY: Van Nostrand Reinhold. chapter 21.

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