# Searching Alternate Spaces to Solve Oversubscribed Scheduling Problems

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

May 15, 2007

**Abstract**

Oversubscribed scheduling problems have been approached using both direct representations of the solution space and indirect, permutation-based representations (coupled with a schedule builder to produce the corresponding schedule). 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. Finally, we consider opportunities for improving performance by integrating the two approaches into a hybrid approach that exploits both search spaces.[1]

## 1 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 and Smith, 2004, Barbulescu et al., 2006, Smith, 2004, Nigenda and 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

[1] An abbreviated version of this paper appeared in the Proceedings of AAAI 2007 [Kramer et al., 2007a].

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 [Smith et al., 2004].

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 and 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 and 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. In particular, on smaller problem instances with lower levels of capacity *SWO* quite often outperforms *TS*. However, on larger instances and as capacity increases, *TS* becomes quite competitive. For problems where task priority is a hard constraint, *TS* begins to outperform *SWO*, particularly as the instances become more oversubscribed.

Finally, we consider the opportunities of improving performance by combining the two approaches. Both *SWO* and *TS* can often become trapped in states from which either no progress can be made or no improving moves can be found within a relatively large number of iterations. We investigate the question: when one technique appears to be stuck, is it productive to switch to the other technique and search in an alternate space? We show that a hybrid algorithm that switches from *SWO* to *TS* when *SWO* gets stuck is able to either match or outperform the best result of both algorithms on all the problem sets.

## 2 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 et al., 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 and Miller, 1994, Rabideau et al., 1999, Kramer and 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.

### 2.1 TaskSwap

The specific schedule-space search method we consider in this paper is TaskSwap (*TS*)[Kramer and 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 *InsertUnassignableTasks* procedure (Figure 1) considers unassignable tasks one by one, according to priority or some other ordering criteria, and attempts to insert them into the existing schedule by calling *TaskSwap* (line 4). *TaskSwap* (Figure 2) temporarily retracts some number of conflicting tasks (Figure 2, line 3). Any retracted tasks are reassigned after assigning the formerly unassignable task (Figure 2, lines 5 and 7); 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 core of the *RetractTasks* procedure is an heuristic, *ChooseTaskToRetract*, which selects those tasks for retraction that have the best "fitness" for reassignment. In prior research [Kramer and Smith, 2003], the heuristic *max-flexibility* was shown to be an effective metric for *ChooseTaskToRetract* and very fast to execute. We use it throughout the experiments in this paper.

A look-ahead heuristic is used to select the specific start time and resource for each task in the procedure *ScheduleTasks*. This heuristic, *max-availability* [Kramer and

**InsertUnassignableTasks**($Unassignables$)
1. $Protected \leftarrow \emptyset$
2. **loop for** $(task \in Unassignables)$ **do**
3. SaveScheduleState
4. $Result \leftarrow$ TaskSwap($task, Protected$)
5. **if** $Result \neq \emptyset$
6. **then** $Protected \leftarrow Result$
7. **else** RestoreScheduleState
8. **end-loop**
9. **loop for** $(i \in Unassignables \wedge status_i = unassigned)$ **do**
10. ScheduleTask($i$)
11. **end-loop**
12. **end**

Figure 1: InsertUnassignableTasks procedure

**TaskSwap**($task, Protected$)
1. $Protected \leftarrow Protected \cup \{task\}$
2. $ConflictSet \leftarrow$ ComputeTaskConflicts (task)
3. $Retracted \leftarrow$ RetractTasks ($ConflictSet, Protected$)
4. **if** $Retracted = \emptyset$ **then** Return($\emptyset$) ; failure
5. ScheduleTasks($task$)
6. $Retracted \leftarrow$ PrioritizeTasks($Retracted$,least-flexible-first)
7. ScheduleTasks($Retracted$)
8. **loop for** $(i \in Retracted \wedge status_i = unassigned)$ **do**
9. $Protected \leftarrow$ TaskSwap($i, Protected$)
10. **if** $Protected = \emptyset$ **then** Return($\emptyset$) ; failure
11. **end-loop**
12. Return($Protected$) ; success
13. **end**

**RetractTasks**($Conflicts, Protected$)
1. $Retracted \leftarrow \emptyset$
2. **loop for** $(OpSet \in Conflicts)$ **do**
3. **if** $(OpSet - Protected) = \emptyset$ **then** Return($\emptyset$)
4. $t \leftarrow$ ChooseTaskToRetract($OpSet - Protected$)
5. UnscheduleTask($t$)
6. $Retracted \leftarrow Retracted \cup \{t\}$
7. **end-loop**
8. Return($Retracted$)
9. **end**

Figure 2: Basic TaskSwap Search Procedure

Smith, 2005b], considers existing and potential resource contention for an unassigned task and places it where availability is predicted to be maximal over the range of that task.

*TaskSwap* 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 (Figure 2, lines 8-11). 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 (Figure 1, line 7). A final scheduling pass is made on remaining unassignable tasks in case any tasks can be synergistically assigned due to freed up space in the schedule (Figure 1, lines 9-11).

Since *TS* relies so heavily on the ChooseTaskToRetract heuristic making good choices, we've found that a stochastic search in the neighborhood of the heuristic is often effective in assigning additional tasks. In particular, Value Biased Stochastic Sampling (VBSS) [Kramer and Smith, 2004, Cicirello and Smith, 2002], applied to the retraction heuristic was shown to boost *TS* performance for the AMC problem sets [Kramer and Smith, 2004]. The results presented for *TS* in this paper were obtained by running *TS* with multiple additional iterations of VBSS.

## 2.2 Squeaky Wheel Optimization

The permutation-based method we consider in this paper incorporates Squeaky Wheel Optimization (*SWO*) [Joslin and 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 and Kürklü, 2005, Barbulescu et al., 2006]. The algorithm starts by establishing an initial permutation (scheduling order) of the input tasks to be scheduled, based on some priority heuristic (Figure 3, line 1), then it 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 schedule (Figure 3, line 3). The permutation represents a prioritization of the tasks, since the

**Squeaky Wheel Optimization**($TaskList$, $PrHeuristic$, $MaxIterations$)
1. $TaskOrdering \leftarrow PrioritizeTasks(TaskList, PrHeuristic)$
2. **loop for** $i \leftarrow 1$ **to** $MaxIterations$ **do**
3. $\quad ScheduleTasks(TaskOrdering)$
4. $\quad$ **loop for** $(task \in Unassignables)$ **do**
5. $\quad\quad MoveDistance \leftarrow ComputeMoveDistance(task, TaskOrdering)$
6. $\quad\quad TaskOrdering \leftarrow MoveForward(task, TaskOrdering, MoveDistance)$
7. $\quad$ **end-loop**
8. **end-loop**
9. **end**

Figure 3: Basic SWO Search Procedure

earlier tasks are considered earlier by the schedule builder. In the second step, the solution is analyzed and for each of the tasks causing "trouble" (i.e., those tasks that the schedule builder was not able to get on the schedule) and a "move distance" is computed (Figure 3, line 5). The move distance represents the number of positions by which to move the task forward in the permutation. The heuristics used to compute the move distance can vary anywhere from defining a fixed number of positions to computing the number of positions as a function of the contribution of the task to the objective function (where the tasks contributing more would be moved for a longer distance). Finally in the third step, the "trouble makers" are moved earlier in the scheduling order (Figure 3, line 6). 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 the same *max-availability* look-ahead heuristic as *TS*. 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. The schedule builder is also employed to generate initial seed solutions for both *TS* and *SWO*.

# 3 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 [Smith et al., 2004]. 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 $T_i^{Dur}$.[2]
- A set $Res$ of resources are available for assignment to tasks. Each resource $r \in Res$ has capacity $cap_r \geq 1$. The 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.

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

- 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 two domains of interest 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:

- *Problem Size*: 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*: capacity for AFSCN varies between 1 and 3 units; for AMC, it varies between 4 and 37.

- *Degree of temporal flexibility*[3]: 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.

## 3.1 Minimizing Unassignable Tasks in AFSCN

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.

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).

[3] measured as task duration relative to the size of the resource time windows.

| 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.

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*. Running *TS* using *VBSS* for more than one iteration did not result in improvement for the AFSCN problem instances, not surprising given the limited resource capacity, and thus a dearth of good alternative task retraction choices.[4]

## 3.2 Minimizing Weighted Unassignables in AMC

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 *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

[4]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.

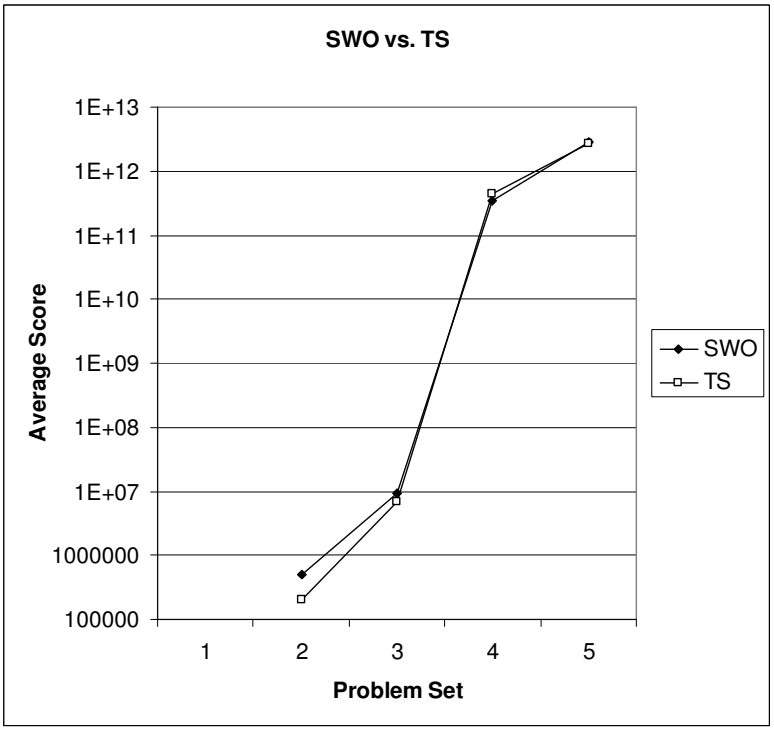


Figure 4: AMC Problems: Average Penalty Score.

base distance according to priority class, $Pr$, of the unassignable $u$, move distance was defined as $md(u) = 200 + (10 \times (6 - Pr(u)))$.[5]

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 4 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 5.[6] A Wilcoxon signed-rank test [Ott and 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.[7] With respect to average unassignables, significantly fewer are obtained

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

[6]These results were originally reported in [Kramer et al., 2007b].

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

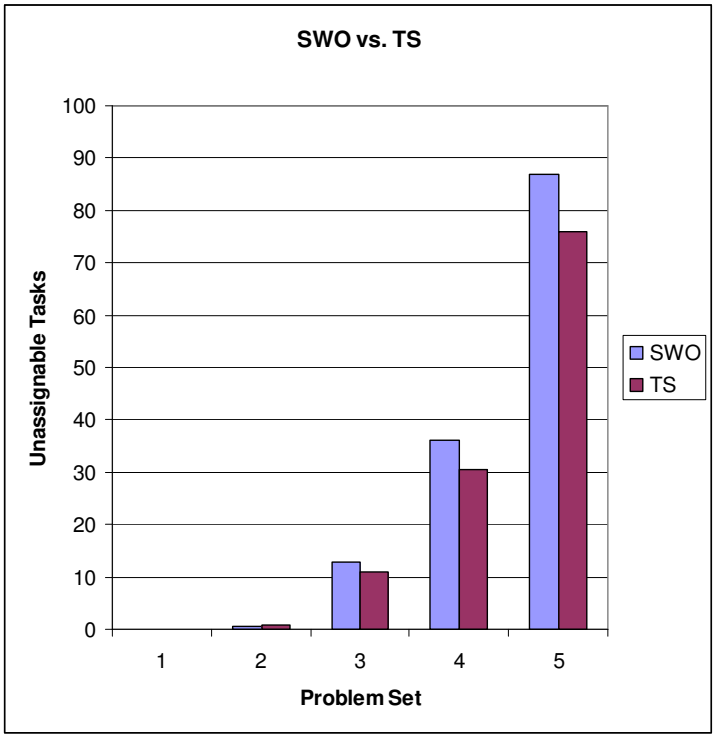


Figure 5: AMC Problems: Average Unassignables.

for both problem sets 4 and 5 ($p < 0.01$).

# 4 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 problem instances 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.

scores.

## 4.1 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 instances 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), one (or two) 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: $cap_r + random(cf, 0)$.

- 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. For this newly generated problem set we conduct experiments with 500 iterations of *SWO* and 30 iterations of *TS/VBSS*.

| 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.

## 4.2 Experimental Results

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 6). 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 on average in a similar number of final unassignables for all problem sets. Our initial hypothesis was that *TS* would begin to outperform *SWO* and vice versa 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*[8] 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 data 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

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

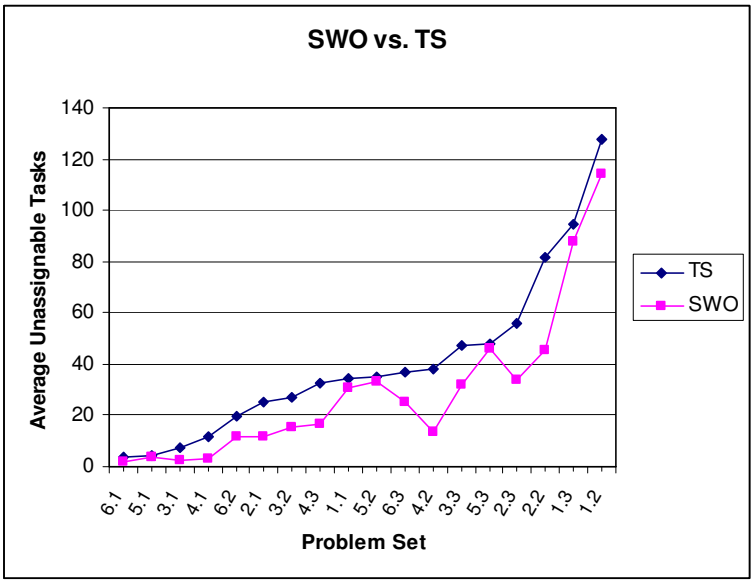


Figure 6: Average Penalty Score (lower values better) for problems without task priorities

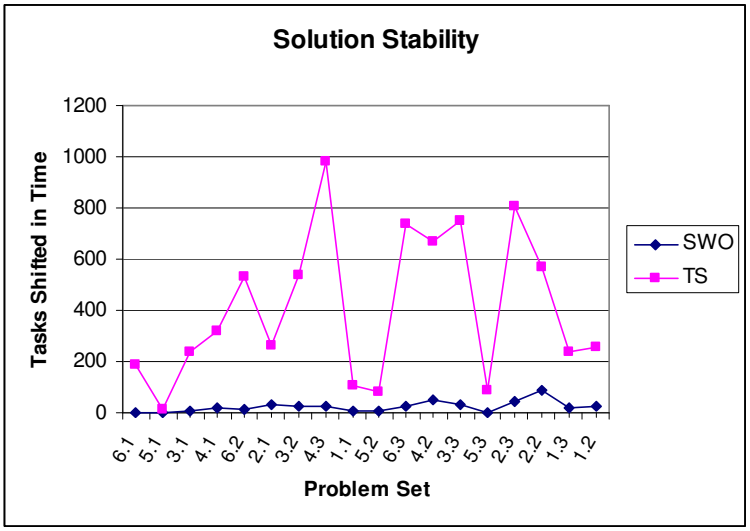


Figure 7: Stability: Number of Tasks that Shifted in Time

| 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 problem instances in the set. Negative values indicate problem sets where *TS* outperforms *SWO*. Bold numbers are statistically significant.

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 problem instances 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 *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 at 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.

We also track schedule stability. We compare the initial and final schedules for each run in terms of number of tasks that shifted in time (Figure 7)[9]. Clearly *SWO* is not well suited to preserve the initial schedule; on the other hand, *TS* is designed with schedule stability in mind.

# 5 Searching Alternate Spaces

Our research has demonstrated on a common data set that both Squeaky Wheel Optimization and TaskSwap are effective methods for solving oversubscribed scheduling problems. Given these two good methods with relative strengths and weaknesses we pose the question: is it productive to combine the strengths of the two in a hybrid fashion to produce a new the technique which outperforms both methods individually? Before answering that question we first outline the strengths and weaknesses and the differing search spaces of *TS* and *SWO*.

[9]This result is for the $pf$ = false problems. As before, the x-axis is ordered in increasing order of difficulty. Similar results are obtained when $pf$ = true.

## 5.1 Two Techniques: Strengths and Weaknesses

Both *TS* and *SWO* are designed to be repair procedures that start from an initial good solution generated by a greedy scheduler. SWO repeatedly generates new schedules – which may or may not be better than the current best – by generating new task permutation orderings for rescheduling. The new permutation orderings are constructed by identifying the "squeaky wheel" tasks and moving them forward a distance which is typically dependent on features of the task. For instance, for the problems that incorporate task priority, the squeaky wheels are the unassignable tasks, and they are moved forward by a function proportional to their priority.

The characteristic that *SWO* is fairly blind to the current schedule state contributes to both its strengths and weaknesses. The fact that *SWO* often schedules and unschedules numerous tasks that may not be central to a good solution allows it jump to problem spaces that might otherwise not be found by a more directed method. Given a good initial schedule and a good move operator, this can often allow *SWO* to converge fairly quickly to a very good solution. After this very good solution is achieved, though, *SWO* continues on blindly, often wasting numerous iterations in the space of bad solutions. In our experiments we regularly see that after *SWO* has stopped generating improving solutions it gets stuck in oscillatory behavior, repeatedly trading off one set of unassignable tasks for another distinct set of unassignable tasks.

In contrast to *SWO*'s disruptive, though effective, behavior, *TS* was designed with schedule stability in mind, and aims at assigning more tasks by examining in detail the current schedule state as each unassignable task is attended to in turn. Already assigned tasks are temporarily de-assigned on the basis of their fitness for reassignment, and each assignment is made by heuristics that examine the schedule state both at the time of assignment and in look-ahead to future task assignments. The strong focus of the *TS* algorithm, somewhat tempered by successive searches (e.g., *VBSS*) in the neighborhood recommended by the heuristics, is both its strength and weakness. It is a strength in the sense that *TS* is able to move inexorably from one good solution to a better one. It is a weakness in the sense that *TS* may miss the *best* solution that requires undoing of a good one to get there.

## 5.2 TS and SWO: Combining the Strengths in a Hybrid

Like most local search algorithms *SWO* and *TS* often become stuck in a state from which no improving moves can be found over a relatively large number of iterations. There are a number of approaches that can be considered for hybridizing *TS* and *SWO* to address this problem. We study two TS/SWO hybrids that operate at a very coarse grain. Since both techniques seem to get stuck somewhat in a good solution, we run *TS* to focus on the search for a better solution after *SWO* gets stuck generating new solutions in a somewhat undirected way. We also experiment with running *SWO* after TS gets stuck at a good solution, with the idea that *SWO*'s disruptiveness may jump to an even better point.

For our previous experiments with the problems described in Table 2 we ran *TS* with VBSS for 30 iterations. The value 30 is somewhat arbitrary, but reasonable given that the *TS* algorithm is very fast and in practice rarely showed improvement after 20

iterations. *SWO*, which has no "natural" stopping criterion other than discovery of an optimal solution – rare for the problems we studied – was run for a maximum of 500 iterations. In this case 500 is fairly dependent on problem-type, and is based on the observation that allowing for 1,000 or 2,000 iterations only very rarely produced additional improvement.

The question of whether *TS* can improve upon *SWO* after 500 iterations or whether *SWO* can improve on *TS* after 30 iterations is in itself only somewhat interesting, as for any given problem the primary search technique itself may not have finished improving. In designing our experiments then we ask a different question: When one method is "stuck," can the other improve on that solution? Answering this question depends on how we determine a technique to be "stuck," and for that we again rely on close observation of numerous runs of the problems in our test data set.

## 5.3 Hybrid Experimental Setup

We define a parameter, $\xi$, iterations without improvement, which we set to 50 for *SWO* and 5 for *TS*. If *SWO* makes no improvement within 50 iterations since its last improvement (with the maximum iterations still 500) then *TS* is run for 5 iterations. We will refer to this algorithm, *SWO* followed by *TS*, as *HybridSWO*. Similarly for *TS* we set $\xi = 5$. If *TS* doesn't improve within 5 iterations to a maximum of 30, *SWO* is executed for 50 iterations. We call this algorithm, *TS* followed by *SWO*, *HybridTS*.

We run *HybridSWO* and *HybridTS* on the 1,800 problem instances (900 with $pf = false$, 900 with $pf = true$) from the 18 sets and report on the results analogous to those described in Section 4.2.

## 5.4 Experimental Results for Hybrid Runs (Without Task Priority)

Table 4 summarizes the results of running *HybridTS* and *HybridSWO* for the 18 problem sets without task priority ($pf = false$). The second column shows the improvement that *HybridTS* (*TS* followed by *SWO*) was able to achieve over *TS* alone. It is computed as the median of the difference in end number of unassignable tasks between *HybridTS* and *TS* for each individual problem. I.e., for all problem instances, $p_i, 1 \leq i \leq 50$, with end number of unassignables, $u$, *HybridTS* Improvement $= median(HybridTS_{i,u} - TS_{i,u})$. A negative value, then, indicates some measure of improvement for that problem set. Similarly, column three displays the improvement of *HybridSWO* over *SWO* alone (*HybridSWO* Improvement = $median(HybridSWO_{i,u} - SWO_{i,u})$). As can be seen, *HybridTS* is able to improve somewhat on *TS* in four of the problem sets; *HybridSWO* improves on *SWO* alone in one problem set.

Columns four and five indicate the performance of *HybridTS* and *HybridSWO* versus the best of *TS* and *SWO* alone. Column four, *HybridTS* vs. Best, is computed for all problem instances, $p_i, 1 \leq i \leq 50$, with end number of unassignables, $u$, as $median(HybridTS_{i,u} - min(TS_{i,u}, SWO_{i,u}))$ and column 5, *HybridSWO* vs. Best, as $median(HybridSWO_{i,u} - min(TS_{i,u}, SWO_{i,u}))$. These results are not encouraging for the Hybrid methods as *HybridTS* performs worse than the best in 11 of 18 problem sets and can improve on the best in none. *HybridSWO* doesn't fare as poorly,

| Prob. Set | HybridTS Improvement | HybridSWO Improvement | HybridTS vs. Best | HybridSWO vs. Best |
|---|---|---|---|---|
| 6.1 | 0 | 0 | 0 | 0 |
| 5.1 | 0 | 0 | 0 | 0 |
| 3.1 | 0 | 0 | 0 | 0 |
| 4.1 | **-1** | 0 | 0 | 0 |
| 6.2 | 0 | 0 | 1 | 0 |
| 4.3 | 0 | 0 | 3 | 0 |
| 3.2 | 0 | 0 | 2 | 0 |
| 2.1 | **-1** | 0 | 2 | 0 |
| 6.3 | 0 | 0 | 1.5 | 0 |
| 5.2 | 0 | 0 | 0 | 0 |
| 4.2 | 0 | 0 | 4 | 0 |
| 1.1 | **-5** | 0 | 0 | 0 |
| 5.3 | 0 | 0 | 0 | 0 |
| 3.3 | 0 | 0 | 3 | 0 |
| 2.3 | 0 | **-0.5** | 3 | 0 |
| 2.2 | 0 | 0 | 9 | 0 |
| 1.3 | 0 | 1 | 1 | 1 |
| 1.2 | **-3.5** | 1 | 4 | 1 |

Table 4: Problems without Task Priority: Relative Performance of *HybridTS* and *HybridSWO* compared to *TS*, *SWO* and the better of *TS* and *SWO*, respectively. Negative values reflect improvement.

matching the best in 18 of the 20 sets, but in general it, too, is unable to improve on the best solution.

## 5.5 Experimental Results for Hybrid Runs (With Task Priority)

The results for $pf = true$ are summarized in Table 5 and are computed identically to the ones in Table 4, but now the values represent the median difference in penalty scores, not unassignable tasks. The results show (column five of Table 5) that *HybridSWO* is able to match or improve upon the best results of *TS* or *SWO* alone for all problem sets, significantly so for four of them. This behavior is not surprising, as *TS* was shown (Table 3) to outperform *SWO* for the "harder" problems incorporating task priority. Clearly running *TS* to improve upon *SWO* in *HybridSWO* produces a significant benefit, but what was not intuitively obvious is that it would improve on *TS* by itself. Apparently starting from the "better plateau" that *SWO* provides leverages *TS* more than starting from a good, greedy solution.

The data also show that *HybridTS* (*TS* followed by *SWO*) is only able to improve significantly on *TS* in one case – problem set 6.3. *HybridSWO* matches or improves on the performance of *SWO* in all problem sets, and is significantly better in 13 if 18. Compared to the best results *HybridTS* is able to improve in only problem set 6.3, and produces worse values in half the problem sets.

| Prob. Set | HybridTS Improvement | HybridSWO Improvement | HybridTS vs. Best | HybridSWO vs. Best |
|---|---|---|---|---|
| 6.1 | 0 | 0 | 0 | 0 |
| 5.1 | 0 | 0 | 0 | 0 |
| 3.1 | 0 | 0 | 0 | 0 |
| 4.1 | 0 | 0 | 1 | 0 |
| 6.2 | 0 | **-0.5** | 0 | 0 |
| 4.3 | 0 | **-3** | 1000 | 0 |
| 6.3 | **-1503072** | **-2002.5** | **-1004071** | 0 |
| 2.1 | 0 | **-998.5** | 0 | **0** |
| 3.2 | 0 | **-2003.5** | 0 | 0 |
| 3.3 | 0 | **-1000450** | 599.5 | -1000 |
| 4.2 | 0 | **-1998.5** | 1001.5 | **-963.5** |
| 5.2 | 0 | **-999475** | 0 | 0 |
| 5.3 | 0 | -651500 | 0 | 0 |
| 1.1 | 0 | **-4006422.5** | 50.5 | **0** |
| 2.3 | 0 | **-3016008.5** | 400.5 | -3154 |
| 2.2 | 0 | **-6057765** | 1165 | **-2502395** |
| 1.3 | 0 | **-1005870000** | 300 | -999500 |
| 1.2 | -1200 | **-1509085150** | 592500 | -2496700 |

Table 5: Problems with Task Priority: Relative Performance of *HybridTS* and *HybridSWO* compared to *TS*, *SWO* and the better of *TS* and *SWO*, respectively. Negative values reflect improvement, with those in bold statistically significant ($p \leq 0.009$).

## 5.6 Revisiting Problem Set 1.2

An interesting question that arises is why *HybridSWO* is so effective for the harder problem sets with task priority, but is ineffective for the same problem sets without task priority? A quick answer to this might be that in general for problems without task priority the performance of *TS* often lags behind that of *SWO*, and thus it might be somewhat surprising had *TS* been able to improve on *SWO* when it got "stuck." This turns out not to be the case, though. The problem turns out to be the difficulty in selecting an appropriate value for $\xi$, iterations without improvement. For all problem sets without task priority, and for the "easier" problem sets with task priority, *SWO* converges to its best solution fairly quickly, so the choice of $\xi = 50$ is a good one. Some of the problem instances in the harder problem sets without task priority do not follow this trend.

We can see this by closely tracing the runtime behavior of Problem Set 1.2, the hardest we tested. For this test we ran *SWO* 1,000 iterations instead of 500, and for each of the 50 problem instances recorded at which iteration an improving move occurred. The results are recorded graphically in Figure 8 where each line in the graph is one solution instance, the x-axis is the move number, and the y-axis the iteration corresponding to that move. The first thing we note is that for half of the problem instances *some* improvement was made after 500 iterations. This was generally not the case for most of the other problem sets. The second thing is that most problem instances make

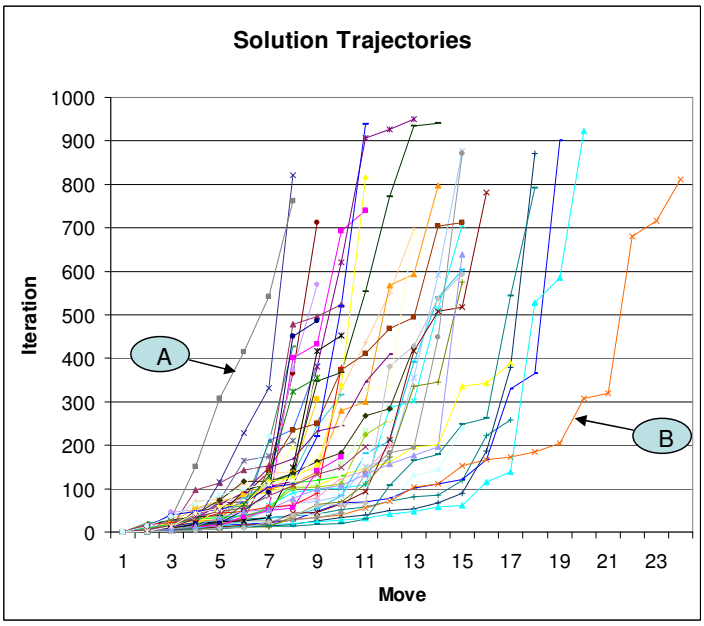


Figure 8: Solution Trajectories for Problem Set 1.2 ($pf = false$)

most of their improvement early and often, clustering in the lower left hand quadrant of the graph. For these instances, the choice of $\xi = 50$ is a good one, as it picks up the fact that *SWO* has made a number of moves, but appears to be stuck.

Problem instances like those labeled $A$ and $B$ occur fairly frequently in this graph, though, and represent a problem for choosing a common value of $\xi$. In case $A$ we see that *SWO* makes a few quick improving moves before iteration 50, but then does not improve until iteration 151, and then at iteration 307. A few more improvements are found up until iteration 761. Increasing $\xi$ to even 100 in this case would not have helped, so by declaring *SWO* to be stuck at iteration 31, we are asking *TS* to do a great deal of work from that point. Example $B$ is somewhat different, but equally problematic. In this case *SWO* has made 19 improving moves by iteration 205. There is a then a large gap to a better state at iteration 307. Suppose we had increased $\xi$ to 150. This would get us to iteration 319 at move 21. But then the next move is not until iteration 680, and we would miss it and the succeeding two improving moves.

To summarize, then, for most problem sets we can select a common, relatively small value for $\xi$ and be assured that it reflects the fact that the solver is stuck. For a few problem sets without task priority, though, a reasonable a priori value for $\xi$ is difficult to find. One final point we should make is that for problem set 1.2, whether for 500 max iterations of *SWO* or 1,000, if we set $\xi$ to be sufficiently large, say between 300 and 400, *HybridSWO is* able to improve on the prior best solution in 32 of 50 instances.

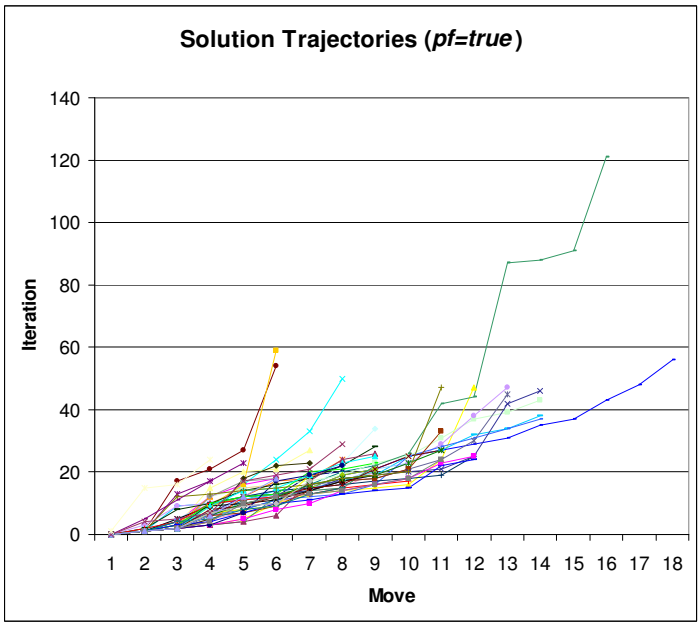


Figure 9: Solution Trajectories for Problem Set 1.2 ($pf = true$)

## 5.7 A Radically Different Search Space

We repeat the same experiment for the problem instances with random task priority ($pf = true$) for set 1.2, first running 1,000 iterations of *SWO*, followed by 5 iterations of *TS*. The resulting solution trajectories are plotted in Figure 9. While the solutions improve over only a slightly smaller number of moves than the problems without task priority, the striking thing is that all instances except one cease making moves within 60 iterations, with most tightly clustered in the 15 to 25 move range. It is clear from this experiment that *SWO* operates in a very different space for the problems with random task priority. It converges much more quickly to a less than optimal solution, and it turns out that *TS* is able to improve on the *SWO* solution for *all* the 50 problem instances in this set.

## 5.8 Runtime Behavior

Finally, we compare the runtime performance for *TS*, *SWO*, and the hybrid methods. One would hope that we could see some improvement in run times in the hybrid methods due to the significantly reduced number of iterations that are executed. This turns out not to be the case, though, as is apparent in Figure 10. The problem is that for all methods except *TS*, the runtime behavior of *SWO* dominates the overall runtime. Certainly both *HybridSWO* and *HybridTS* are faster than *SWO* alone, but neither of them is able to match the runtime of *TS*. As can be seen from the graph, *SWO* runtime is strictly

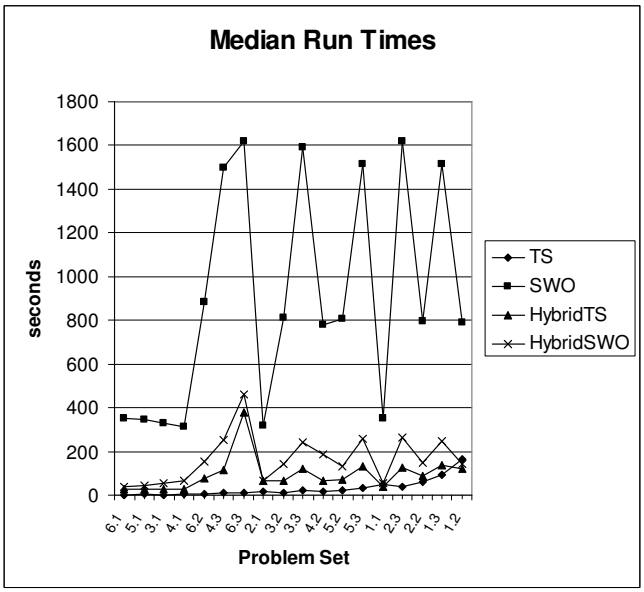


Figure 10: Median Run Times, Hybrid vs. Non-hybrid Algorithms

a function of problem size and number of iterations – the problem sets with approximately 450 tasks taking close to 400 seconds to run, those with double the number of tasks running in approximately 800 seconds, and those with three times the number of tasks consuming 1,500 to 1,600 seconds. In contrast, the runtime of *TS* depends only on the initial number of unassignables (it is $O(n^2)$ in $n$, the number of unassignable tasks) and the number of neighborhood iterations it is run. Looking at the graph from left to right, we see that its runtime scales very gradually as the problem sets become harder. The runtimes for both *HybridSWO* and HybridTS incorporate some number of *SWO* runs, and suffer for it. On the other hand, their run time is always much better than *SWO* alone, and with only one exception on average have an upper bound of about 300 seconds. As a matter of fact, with some reduction of $\xi$, iterations without improvement, it may be possible for the problems with task priority for the runtime performance of *HybridSWO* to approach that of *TS*, since *SWO* converges very quickly (generally in less than 30 iterations) for these problems.

We tested this conjecture for problem set 1.2 ($pf = true$) by resetting $\xi$ from 50 to 10, and the maximum iterations to run *SWO* from 500 to 50. Doing this only minimally affects the performance of *HybridSWO*: the median penalty score increased less than 0.00005%. Its median runtime improved from 144 to 82 seconds (as compared to 122 seconds for *HybridTS*, 166 seconds for *TS*, and 791 seconds for *SWO*).

# 6 Related Work

One of the limitations of simple search algorithms is that they often get trapped in a locally optimal state. Simulated annealing, tabu search, iterated local search, and random restarts are just a few examples of widely-studied metaheuristics commonly used as strategies for escaping or avoiding local optima. These strategies do not alter the initial search space of the problem; instead, they escape local optima by accepting non-improving moves. A different, less common strategy is to change the representation of the problem, under the assumption that a local optimum under one representation will not remain a local optimum under a different representation. Our two hybrid algorithms *HybridTS* and *HybridSWO* employ this latter strategy, by changing the representation between the permutation space and the schedule space. Representation change has also been successfully employed by bit climbing and genetic algorithms for binary search spaces: for example, in [Barbulescu et al., 2000], dynamically changing the Gray encoding is shown to boost algorithm performance on a set of benchmark problems traditionally used to test evolutionary algorithm performance.

The idea of hybridizing SWO to improve its performance has also been studied by Terada et al. [Terada et al., 2006] for a resource constrained "car-sequencing" set of benchmark problems. Their results show that hybridizing by directly incorporating genetic algorithm features in SWO significantly improves over the performance of both. While these results are interesting and support the idea that SWO is a good candidate for hybridizing with another algorithm, their approach is very different and therefore not directly comparable to ours.

# 7 Summary and Future Research

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.

Due to its schedule-space search and use of informed heuristics *TS* is very quick and non-disruptive of overall schedule state. *SWO*, on the other hand is quite disruptive as it makes its way through a search of schedule permutation space, but this "schedule

blindness" often allows it to stumble upon solutions that *TS* in its relative myopia is unable to find.

We extend the analysis of the two individual techniques on a common problem set to the study of two hybrid techniques: *HybridTS* and *HybridSWO*. The former employs *SWO* to diversify after *TS* appears to be stuck at its best solution. The latter uses *TS* to focus on even better solutions after *SWO* appears to have quiesced to at least a local optimum. Our experiments tend to show that *HybridTS* is not much of an improvement on *TS* alone, and generally does not improve on the best solution of *TS* or *SWO* individually. *HybridSWO*, on the other hand, turns out to be very promising, particularly for problems with task priority. It is quite often able to improve on the results of *SWO* alone, and for the hardest problems regularly results in a better penalty score than the best achieved by either of *TS* or *SWO*. Furthermore, with proper parameter tuning, these better scores may actually be achieved in less time.

Our results, while interesting, raise a number of questions for future research. Among them:

- Could further improvement be gained by hybridizing at a finer grain, employing facets of *SWO* within the *TS* algorithm, or alternating between the two techniques repeatedly?
- Would it be productive for the problems we've studied, as some research has suggested, to hybridize *SWO* or *TS* with other techniques?
- Are there problem features that our test sets did not encompass that would tend to favor different techniques and different hybridizations?
- How useful would online and offline learning methods be in improving the performance of the hybrid methods?

# 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

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

L. V. Barbulescu, J.-P. Watson, and D. Whitley. Dynamic representations and escaping local optima: Improving genetic algorithms and local search. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 879–884, 2000.

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

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

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

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

A. Globus, J. Crawford, J. Lohn, and A. Pryor. 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)*, pages 836–843, July 25-29 2004.

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

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

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

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, 2005a.

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 2005b.

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

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.

L. A. Kramer, L. V. Barbulescu, and S. F. Smith. Understanding performance tradeoffs in algorithms for solving oversubscribed scheduling. In *Proceedings of the 22nd Conference on Artificial Intelligence*, Vancouver, BC, July 2007a. The AAAI Press.

L. A. Kramer, L. V. Barbulescu, and S. F. Smith. Analyzing basic representation choices in oversubscribed scheduling problems. In *Proceedings of the 3rd Multidisciplinary International Scheduling Conference: Theory and Applications (MISTA-07)*, Paris, France, August 2007b.

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

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

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

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

S. F. Smith, M. B. Becker, and L. A. Kramer. Continuous management of airlift and tanker resources: A constraint-based approach. *Mathematical and Computer Modeling – Special Issue on Defense Transportation: Algorithms, Models and Applications for the 21st Century*, 39(6-8):581–598, 2004.

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

J. Terada, H. Vo, and D. Joslin. Combining genetic algorithms with squeaky-wheel optimization. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO'06)*, Seattle, WA, July 2006.

L. 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.