Task Swapping for Schedule Improvement: A Broader Analysis

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

Abstract

In this paper we analyze and extend a recently developed "task-swapping procedure" for improving schedules in oversubscribed situations. In such situations, there are tasks which cannot be directly added to the current schedule without introducing capacity conflicts. A schedule is improved if one or more of these tasks can be feasibly included, and the goal of task swapping is to rearrange some portion of the current schedule to make this possible. Key to effective task swapping is an ability to exploit the scheduling flexibility inherent in the constraints associated with various scheduled tasks, and previous work has shown that the use of retraction heuristics that favor tasks with greater rescheduling flexibility can give rise to strong schedule improvement capabilities. We extend this work by developing and evaluating several improvements to the core task swapping procedure. We introduce three pruning techniques and show that each significantly improves computational efficiency while maintaining solution quality. We then investigate the possibility of improving the "end" solutions by stochastically exploring the "neighborhood" around them, and demonstrate that improved solutions are possible given the ability to spend additional time.

Introduction

Scheduling in a continuous plan-schedule-execute-reschedule environment requires more than just the ability to generate good schedules. In such an environment we take as given that the schedules generated will be good, but not optimal, as even if generating an optimal schedule were possible, that schedule would quickly become obsolete as requirements are revised and feedback from execution is taken into account. This view of things puts a premimum on the ability to revise and improve an existing schedule quickly, while at the same time maintaining stability in decisions wherever possible.

In (Kramer & Smith 2003), a task swapping algorithm (referred to as **MissionSwap**) is introduced as a mechanism for improving schedules in oversubscribed problem domains. It was developed for specific application to the USAF Air Mobility Command (AMC) mission scheduling problem (Becker & Smith 2000). In this context, it was used to overcome the bias of a greedy priority-driven schedule generation procedure and squeeze additional, lower priority missions into the airlift schedule. However, the procedure appears equally applicable in schedule repair settings, such as minimizing the number of missions that must be dropped from the schedule when resource capacity is unexpectedly lost.

Recognizing the more general applicability of this task swapping search procedure, this paper studies the **MissionSwap** procedure in more detail. On one hand, we analyze original search design decisions and consider opportunities for efficiency improvements. On the other, we propose extensions aimed at boosting the performance of task retraction heuristics and producing better schedule improvement results. We conduct experiments using a mission scheduling data set comparable to that used in the original study, and quantify performance tradeoffs associated with different algorithmic variants. Interestingly, some of the improvements considered differentially affect the performance of different retraction heuristics, and lead to a somewhat different picture of relative strengths than was obtained in (Kramer & Smith 2003). Overall, our experiments show that **MissionSwap** can be made significantly faster without degrading quality, and that given a reasonable amount of time to devote, iterated stochastic search can lead to additional solution quality. These characteristic make MissionSwap a particularly attractive method for anytime schedule repair and improvement.

We begin by briefly summarizing the AMC scheduling domain and the previously developed **MissionSwap** procedure that we take as our starting point.

The AMC Scheduling Problem

Without loss of generality the AMC scheduling problem can be characterized abstractly as follows:

- A set T of tasks (or missions) are submitted for execution. Each task $i \in T$ has an earliest pickup time est_i, a latest delivery time lft_t, a pickup location $orig_i$, a dropoff location $dest_i$, a duration d_i (determined by $orig_i$ and $dest_i$) and a priority pr_i
- A set Res of resources (or air wings) are available for assignment to missions. Each resource $r \in Res$ has capacity $cap_r \geq 1$ (corresponding to the number of contracted

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

aircraft for that wing).

- Each task i has an associated set Res_i of feasible resources (or air wings), any of which can be assigned to carry out i. Any given task i requires 1 unit of capacity (i.e., one aircraft) of the resource r that is assigned to perform it.
- Each resource r has a designated location $home_r$. For a given task i, each resource $r \in Res_i$ requires a positioning time $pos_{r,i}$ to travel from $home_r$ to $orig_i$, and a de-positioning time $depos_{r,i}$ to travel from $dest_i$ back to $home_r$.

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

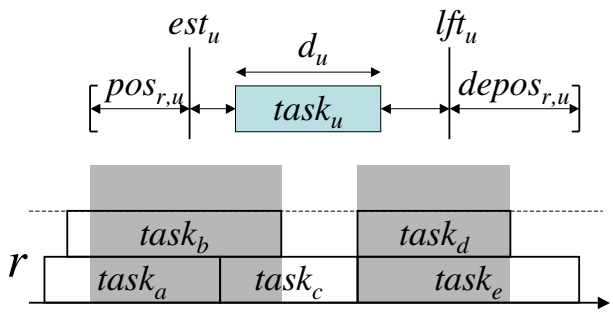
Both the scale and continuous, dynamic nature of the AMC scheduling problem effectively preclude the use of systematic solution procedures that can guarantee any sort of maximal accommodation of the tasks in T. The approach adopted within the AMC Allocator application instead focuses on quickly obtaining a good baseline solution via a greedy priority-driven allocation procedure, and then providing a number of tools for selectively relaxing problem constraints and incorporating as many additional tasks as possible (Becker & Smith 2000). The task swapping procedure of (Kramer & Smith 2003) is one such schedule improvement tool.

The Basic Task Swapping Procedure

The task swapping procedure summarized below takes the solution improvement perspective of iterative repair methods (Minton *et al.* 1992; Zweben *et al.* 1994) as a starting point, but manages solution change in a more systematic, globally constrained manner. Starting with an initial baseline solution and a set U of unassignable tasks, the basic idea is to spend some amount of iterative repair search around the "footprint" of each unassignable task's feasible execution window in the schedule. Within the repair search for a given $u \in U$, criteria other than task priority are used to determine which task(s) to retract next, and higher priority tasks can be displaced by a lower priority task. If the repair search carried out for a given task u can find a feasible rearrangement of currently scheduled tasks that allows u to be incorporated, then this solution is accepted, and we move on to the next unconsidered task $u^{'} \in U$. If, alternatively, the repair search for a given task u is not able to feasibly reassign all tasks displaced by the insertion of u into the schedule, then the state of the schedule prior to consideration of u is restored, and u remains unassignable. Conceptually, the approach can be seen as successively relaxing and reasserting the global constraint that higher priority missions must take precedence over lower priority missions, temporarily creating "infeasible" solutions in hopes of arriving at a better feasible solution.

In the subsections below, we describe this task swapping procedure, and the heuristics that drive it, in more detail.

Task Swapping



ConflictSet$_u$ = {{a,b}, {b,c}, {d,e}}

Figure 1: An unassignable task u

Figure 1 depicts a simple example of a task u that is unassignable due to prior scheduling commitments. In this case, u requires capacity on a particular resource r, and the time interval $ReqInt_{r,u} = [est_u - pos_{r,u}, lft_u + depos_{r,u}]$ defines the "footprint" of u's allocation requirement. Within $ReqInt_{r,u}$, an allocation duration *alloc-dur*$_{r,u} = pos_{r,u} + d_u + depos_{r,u}$ is required. Thus, to accommodate u, a subinterval of capacity within $ReqInt_{r,u}$ of at least *alloc-dur*$_{r,u}$ must be freed up.

To free up capacity for u, one or more currently scheduled tasks must be retracted. We define a conflict $Conflict_{r,int}$ on a resource r as a set of tasks of size Cap_r that simultaneously use capacity over interval int. Intuitively, this is an interval where resource r is currently booked to capacity. We define the confict set $ConflictSet_u$ of an unassignable task u to be the set of all distinct conflicts over $ReqInt_{r,u}$ on all $r \in R_u$. In Figure 1, for example, $ConflictSet_u = \{\{a,b\},\{b,c\},\{d,e\}\}$.

Given these preliminaries, the basic repair search procedure for inserting an unassignable task, referred to as **MissionSwap**, is outlined in Figure 2. It proceeds by computing $ConflictSet_{task}$ (line 2), and then retracting one conflicting task for each $Conflict_{r,int} \in ConflictSet_{task}$ (line3). This frees up capacity for inserting $task$ (line 5), and once this is done, an attempt is made to feasibly reassign each retracted task (line 6). For those retracted tasks that remain unassignable, **MissionSwap** is recursively applied (lines 7-10). As a given task is inserted by **MissionSwap**, it is marked as protected, which prevents subsequent retraction by any later calls to **MissionSwap**.

In Figure 3, top-level **InsertUnassignableTasks** procedure is shown. Once **MissionSwap** has been applied to all unassignable tasks, one last attempt is made to schedule any remaining tasks. This step attempts to capitalize on any op-

MissionSwap($task, Protected$)
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. ScheduleTask($task$)
6. ScheduleInPriorityOrder($Retracted$, least-flexible-first)
7. **loop for** $(i \in Retracted \wedge status_i = unassigned)$ **do**
8. $Protected \leftarrow$ MissionSwap($i, Protected$)
9. **if** $Protected = \emptyset$ **then** Return($\emptyset$) ; failure
10. **end-loop**
11. Return($Protected$) ; success
12. **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 MissionSwap Search Procedure

portunities that have emerged as a side-effect of **MissionSwap's** schedule re-arrangement.

Retraction Heuristics

The driver of above repair process is the retraction heuristic **ChooseTaskToRetract**. In (Kramer & Smith 2003), three candidate retraction heuristics are defined and anlayzed, each motivated by the goal of retracting the task assignment that possesses the greatest potential for reassignment:

- **Max-Flexibility** - One simple estimate of this potential is the scheduling flexibility provided by a task's feasible execution interval. An overall measure of task i's temporal flexibility is defined as

$$Flex_i = \frac{\sum_{r \in Res_i} \textit{alloc-dur}_{r,i}}{(lft_i - est_i) \times |Res_i|}$$

leading to the following retraction heuristic:

$$MaxFlex = i \in C : Flex_i \leq Flex_j \forall j \neq i$$

where $C \in ConflictSet_u$ for some unassignable task u.

- **Min-Conflicts** - Another measure of rescheduling potential of a task i is the number of conflicts within its feasible execution interval, i.e. $|ConflictSet_i|$. This gives the following heuristic:

$$MinConf = i \in C : |ConflictSet_i| \leq |ConflictSet_j| \quad \forall j \neq i$$

where $C \in ConflictSet_u$ for some unassignable task u.

InsertUnassignableTasks($Unassignables$)
1. $Protected \leftarrow \emptyset$
2. **loop for** $(task \in Unassignables)$ **do**
3. SaveScheduleState
4. $Result \leftarrow$ MissionSwap($task, Protected$)
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 3: InsertUnassignableTasks procedure

- **Min-Contention** - A more informed, contention based measure is one that consider's the portion of a task's execution interval that is in conflict. Assuming that dur_C designates the duration of conflict C, task i's overall contention level is defined as

$$Cont_i = \frac{\sum_{C \in ConflictSet_i} dur_C}{\sum_{r \in Res_i} ReqInt_{r,i}}$$

leading to the following heuristic:

$$MinContention = i \in C : Cont_i \leq Cont_j, \forall j \neq i$$

Prior results

The original experiments of (Kramer & Smith 2003) (carried out on a suite of 100 problems) demonstrated the efficacy of the **MissionSwap** procedure in the target domain. In this study, max-flexibility was shown to be the strongest performer; its application enabled scheduling, on average, of 42% of the initial set of unassignable missions. Min-contention, scheduled 38%, but was almost three times slower. Min-conflicts proved less effective, scheduling only 30% on average.

Improving Task Swapping Performance

The **MissionSwap** task swapping procedure is clearly a general mechanism for schedule repair in the presence of multi-capacity resources. While the AMC Allocator performs well employing a max-flexibility or min-contention heuristic, even random choice can be used although with higher cost and worse results. The max-flexibility heuristic, it should be noted, turns out to be well informed because of a reasonable variance in slack (more specifically the ratio of runtime to feasible window) across the set of input tasks. In domains where slack is not very variable, it is likely that a contention-based heuristic would be the better informed.

Independent of heuristic, one can identify two different factors that can impact the effectiveness of a schedule repair procedure like task swapping:

1. *Extent of solution change* - How much space should be created in the schedule (how many tasks should be retracted) when attempting to insert a new task? In continuous and reactive contexts, it is important to minimize

change. Also, retraction of more tasks implies that more tasks need to be put back, and hence minimizing the number of tasks retracted can also be expected to have a positive impact on computational efficiency. On the other hand, increasing the amount of change may increase opportunities for incorporation of additional tasks.

2. *How to manage search efficiency?* The scale and complexity of the space searched by the task swapping procedure prohibits systematic exploration of all retraction and rescheduling options. The task swapping procedure gains leverage from its retraction heuristic. At the same time, this heuristic may offer more or less guidance in different decision contexts, and this can lead to unproductive search decisions and unnecessary search paths.

In the task-swapping procedure summarized in the previous section specific design decisions are made relative to each of these issues. With respect to retraction, procedure **RetractTasks** (in conjunction with the retraction heuristic that is employed) identifies a set of tasks that may be larger than necessary and may cover more of the resource timeline than is strictly necessary to allow insertion of the new task.

With regard to search control, the **MissionSwap** procedure will, in the worst case, try to retract and reschedule all tasks involved in a given conflict before abandoning its attempt to insert a new task, and each unassignable task is considered only once in the overarching **InsertUnassignableTasks** procedure.

In the following sections, we reconsider these design decisions and evaluate several variations of the task-swapping procedure. Our experimental design follows that of the original paper; using the same "AMC Tutorial Data Set" (Kramer & Smith 2003) as a seed, five new data sets of twenty problems each were generated, with resource capacities randomly reduced from 0 to 10%, 0 to 20%, 0 to 30%, 0 to 40%, and 0 to 50% to obtain increasing levels of capacity constrainedness. For each alteration in design space we consider, this suite of 100 problems is used to evaluate solution quality and runtime. In conducting our analysis, we make use of the same retraction heuristics considered previously: max-flexibility, min-conflicts, min-contention, and random choice (as a baseline). All experiments were run on a 1.8Ghz Pentium IV PC with 1Gb of RAM, running Windows 2000. The scheduling engine is implemented in Allegro Common Lisp 6.2.

Minimizing the Number of Tasks Retracted

The first two alterations to MissionSwap that we investigate narrow the search space by reducing the number of tasks that are retracted. The intuition is that by retracting less, fewer branches will be need to be searched to produce a final solution. We identify these two methods as Task Pruning and Interval Pruning.

Task Pruning

In the original **MissionSwap** task swapping method, the **RetractTasks** procedure frees up resource capacity for an Unassignable task by retracting one task that is currently scheduled in each conflict interval. In the naive implementation that is implied, these decisions are based strictly on the advice of the retraction heuristic. It turns out that in the actual *implementation* of the algorithm, though, a fairly straightforward optimization was employed: if a task is retracted in one interval and it happens to free up other intervals, no further tasks need be retracted in those intervals. Consider for example, the set of conflicts preventing the insertion of unassignable $task_u$ in Figure 4. If $task_b$ is selected by the retraction heuristic to resolve conflict $\{a, b\}$ then conflict $\{b, c\}$ is also resolved and there is no need for further retraction, even if the retraction heuristic prefers $task_c$ to $task_b$.

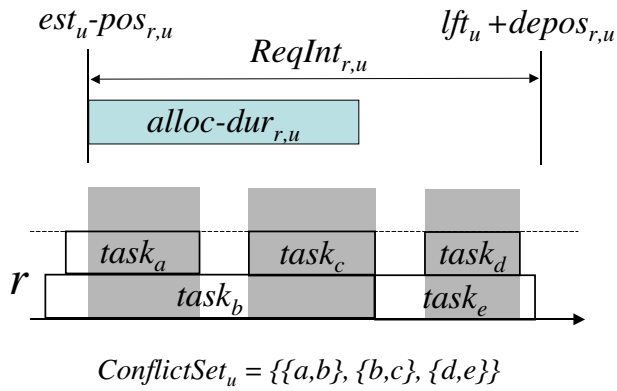


Figure 4: Another unassignable task u

Intuitively it seems that minimizing the number of tasks retracted in this manner will reduce the amount of search required and thus make the base algorithm more efficient. However, it not as obvious whether this optimization might hurt or help solution quality. For instance, it might be the case that task pruning reduces search, but in the process reduces quality by affording unassignable tasks less "room" in which to reschedule. In general, it might be the case that additional disruption of the current solution can create additional opportunities for solution improvement.

Figures 5 and 6 depict the average quality and cost performance of various heuristics on the test problem suite with task pruning enabled. The comparative performance results (shown in Table 1) confirm that task pruning *does* improve average runtime by 71% in the worst case (random choice retraction heuristic) and by 85% (over a six-fold speed-up) in the best case (max-flexibility retraction heuristic).

The interesting thing is that these significant speed-ups are achieved with an accompanying *increase* in solution quality for all of the retraction heuristics. Max-flexibility improves the least at 2.63% and min-contention the most at 11.48%.

Given the win-win nature of this result we incorporate task pruning as part of the base configuration for all subsequent experiments reported below. It's possible – though unlikely – that some other tweak to the algorithm could mitigate against task pruning, but we feel that to be unlikely.

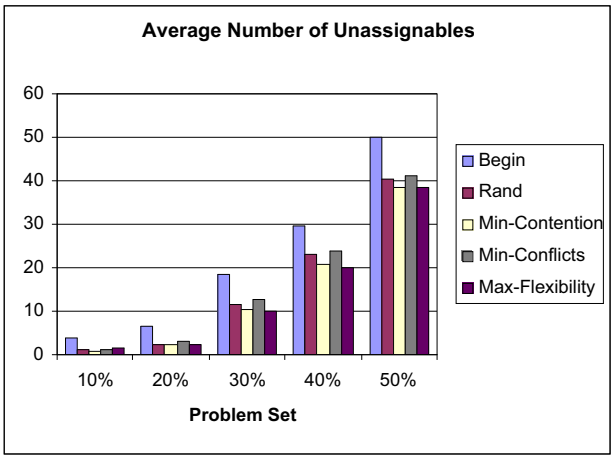


Figure 5: Solution Quality

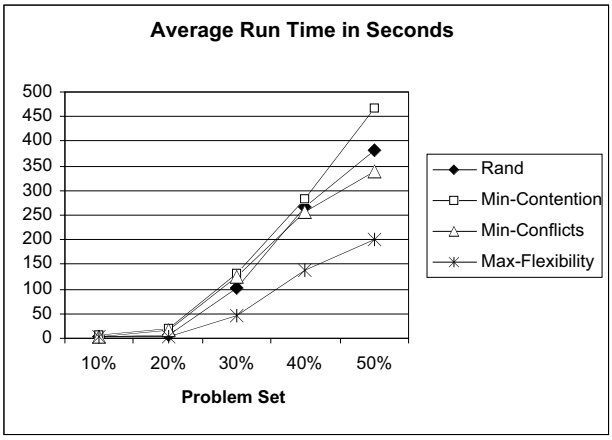


Figure 6: Computational Cost

Interval Pruning

A second way in which **RetractTasks** is non-minimal is in the amount of space that it clears along the timelines of resources required by a given unassignable task. Consider again the potential insertion of $task_u$ in Figure 4, where three conflict intervals are identified over the required interval $ReqInt_{r,u}$ of $task_u$ on resource r. **RetractTasks** (with or without task pruning) will retract sufficient tasks to resolve *all* conflicts and free up the entirety of $ReqInt_{r,u}$; even though $task_u$ could be feasibly scheduled on r in this case without retraction of $task_d$ or $task_e$. Moreover, **RetractTasks** will clear $ReqInt_{r,u}$ for each $r \in R_u$.

While it doesn't seem productive to rule out one resource over another a priori, it may be profitable to limit the number of conflicts processed on a given resource r to a subset that is strictly necessary to allow placement of the unassignable task at hand.

Given that the default allocation strategy used to insert new tasks schedules in an earliest-first manner, a natural subset of conflicts to focus on are those are obtained via a "left-to-right" scan of $ReqInt_{r,u}$. Specifically, we augment **RetractTasks** to terminate at the point where a sub-interval of $ReqInt_{r,u}$ that is greater than or equal to *alloc-dur*$_{r,i}$ has been obtained. We refer to this extension to **RetractTasks** procedure as *interval pruning*.

Running our base algorithm with interval pruning activated results in another significant cost performance gain for all retraction heuristics, in general around a two-fold speed-up (See Table 1 below). The results in terms of quality are somewhat mixed, though. The quality of solutions obtained with Max-flexibilty degrades slightly (by 1.66% on average), while the quality of min-contention solutions increases by 5.45% on average.

It is understandable that the solution quality of max-flexibility suffers somewhat as the interval that is freed up is pruned in breadth. Basically we have given the newly retracted tasks less room to schedule in, while the heuristic is computed based on as task's full feasible window. The contention-based heuristics, alternatively, appear to gain leverage as the extent of cleared space on resource timelines is narrowed.

Depth-Bounded Search Cutoff

Depth-limited depth-first search is a well known technique (Russell & Norvig 1995) for limiting search in many problem areas where a depth bound is known, to prune branches and still produce a complete, but possibly non-optimal result.

In the case of task-swapping, the use of a depth bound makes particular sense; as search proceeds further down in the tree resulting from an attempt to resolve a given conflict, the retraction heuristic can be expected to become less and less useful. For example, if max-flexibility is the heuristic, then choices high in the search tree will correspond to those operations that have the largest slack. As the search descends to lower plies in the tree, the tasks selected for retraction have increasingly less slack and hence will likely become more and more difficult to reschedule. A similar progressive weakening of search leverage as the search descends deeper into the tree can be expected from any retraction heuristic.

In practice, our observation of many hundreds of runs of the **MissionSwap** algorithm indicates that if the search proceeds much past 8 to 10 levels deep, it is almost always doomed to fail. There is very occassionally a successful path as much as 20 levels deep, but that is very rare.

Accordingly, we examine the performance tradeoff of running **MissionSwap** in truncated mode with a fixed depth cutoff. Our expectation is that this depth cutoff should save a good deal of time by failing earlier in the case of eventual failure, while only affecting the quality of the results slightly, if we have select a good depth bound. Following the above observations we resolve our suite of test problems using a cutoff value of 8.

Our comparative experimental results (See Table 1) prove this conjecture to be true. By limiting search to eight levels deep, we see run-times that are two to three times faster,

	Average Unassignables by Pruning Technique				
	None	Task	Task+Interval	Task+Depth	T+I+D
Begin	21.83	21.83	21.83	21.83	21.83
Random	16.53	15.69	16	16.4	16.19
Min-Cont.	16.38	14.5	13.71	15.73	13.85
Min-Conf.	17.64	16.36	15.77	16.51	15.82
Max-Flex.	14.81	14.42	14.66	14.58	14.72
	Average Runtime by Pruning Technique				
	None	Task	Task+Interval	Task+Depth	T+I+D
Random	518.85	151.91	56.79	45.71	46.8
Min-Cont.	729.43	181.14	79.11	74.07	75.19
Min-Conf.	750.88	148.23	93.57	63.75	78.99
Max-Flex.	528.8	78.37	43.7	37.34	37.36

Table 1: Overall Performance Results

depending on heuristic. Degredation in solution quality is seen to vary across heuristics; max-flexibilitty suffers only a 1.11% decrease in solution quality, while Min-Contention incurs an 8.48% decrease. Min-contention clearly needs to occasionally search deeper than eight plies to achieve its best results. At the same time, this quality loss can be weighed against a 2.4 times speedup in runtime.

Composite Pruning Results

Our next experiment in design space of the MissionSwap algorithm combines all pruning techniques – task, interval, and depth. The results are summarized in Table 1. As can be seen, this combination generally seems to be a good trade off between the speed gains of depth pruning and the quality gains of interval pruning.

The best that can be said is, that outside of task pruning, which was an unmitigated success, improvements due to interval and depth pruning were somewhat heuristic dependent, and these results would likely vary somewhat depending on the characteristics of the application domain. Figure 7 shows the variation between the two best heuristics, max-flexibility and min-contention, using the various pruning techniques. Figure 8 compares their runtime performance across those techniques.

While it is clear that max-flexibility is still far and away the best heuristic in terms of runtime, several of the new pruning techniques give min-contention an edge in terms of solution quality. If the desire is for best solution quality, using min-contention as the retraction heuristic with task and interval pruning is the best choice, with a final unassignable average value of 13.71 at a 79.11 average runtime. If the goal is to achieve the fastest runtime, it is best to use max-flexibility using task and depth pruning, achieving an average final unassignable value of 14.58 in 37.34 seconds.

Iterated Task-Swapping

To this point, our investigation of tradeoffs in the design space of task swapping search procedures has aimed at extensions that improve run time performance. In some cases, these extensions have had a synergistic positive effect on solution quality; in others, runtime improvement has come at the expense of some decrease in solution quality.

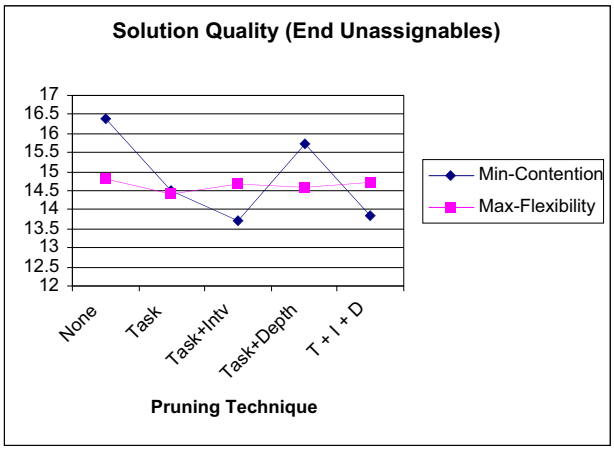


Figure 7: Comparative solution quality

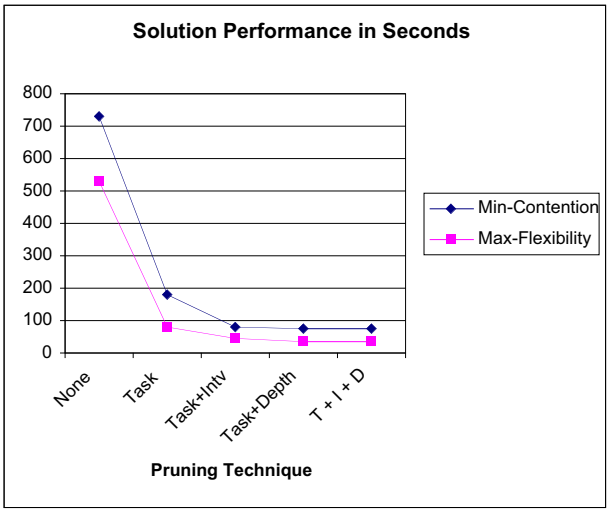


Figure 8: Comparative computational cost

In this section, we focus alternatively on mechanisms for expanding the task-swapping repair search that is performed, for use in obtaining better solutions in circumstances where extended computation is possible. Specification of a complete search procedure is out of the question due to the intractable nature of the scheduling domains we are considering. Even a limited discrepancy search (Harvey & Ginsberg 1995), is problematic for the application studied in this paper, given the large size of the search space and the high branching factor. At the same time, the current **MissionSwap** procedure relies heavily on the quality of its retraction heursitic and the ability to broaden this search in some way could be beneficial.

One approach to extending the search follows from the fact that the overall **InsertUnassignableTasks** procedure cycles just once through the set of unassignable tasks. Yet, any time that **MissionSwap** is successful in inserting a new

unassignable into the schedule, a new problem state is defined for inserting other tasks. From the standpoint of optimizing solution quality, this "try to insert only once" assumption is restrictive.

Below we explore to ways of relaxing this assumption:

1. by iteratively invoking **InsertUnassignableTasks** until quiessence, i.e., until a complete pass over the set of unassignable tasks is made in which no new tasks are scheduled. This extended search configuration is referred to as *Iterative-Deterministic* below.
2. by iteratively invoking **InsertUnassignableTasks** for some number of iterations using a non-deterministic variant of the retraction heuristic. A couple of alternative configurations are defined in the next subsection

Biased-Stochastic Task Retraction

The performance of **MissionSwap** depends heavily on the decisions of its retraction heuristic, yet no heuristic is infallable. Even very good retraction heuristics such as min-contention and max-flexibility sometimes make the wrong decisions and hence fail to reach the best solution on a given problem. And sometimes, albeit rarely, even a policy of random retraction will outperform. Starting from the assumption that one has a good search heuristic, a number of techniques have demonstrated an ability to boost search through randomization (Bresina 1996; Oddi & Smith 1997; Cicirello & Smith 2002). The basic idea is to broaden search to encompass a stochastically defined "neighborhood" of the heuristic's deterministic trajectory through the search space. Typically this is accomplished by re-running the search multiple times within an iterative sampling (or random restarting) search framework.

In the context of task swapping, we adopt a slightly different methodology. Rather than restarting the search from scratch each iteration, the search continually proceeds (as in the case of *iterative-deterministic*) from the best solution found so far. For instance, if our base heuristic is able to reduce the initial number of unassignables from ten to five, we start the stochastic search process in the state with five unassignables. We give the stochastic procedure *n* iterations to improve the solution, and if it improves the solution in less than *n*, say from five to four, the stochastic search continues from the state of four unassignables. This proceeds until either *n* iterations of stochastic search have been completed or the problem has been "solved' (zero unassignables achieved).

We consider two approaches to stochastically-biasing the choices made by a given retraction heuristic:

1. *acceptance band* - This approach, based on work of (Oddi & Smith 1997), chooses which task to retract randomly from the set of all choices (tasks) whose heuristic value falls within a certain percentage x of the highest rated choice (i.e., the task that would have been retracted if the retraction heuristic had been applied deterministically). For example, suppose the base retraction heuristic is max-flexibility and the highest ranked task has a flexibility value of 0.12. (Recall from the earlier definition of flexibility that the smaller the rating, the more flexible the task.) Now suppose the next three highest ranked tasks have values of 0.13, 0.15, and 0.18 respectively. In the range (0, 1) these tasks do not differ much from the highest ranked task, and the heuristic would have to be almost perfect to differentiate between them. By specifying an appropriate acceptance band these tasks are considered equally attractive, and the decision of which to retract is made randomly. For the experiments reported below, we settled on an acceptance band value of 10% after some amount of preliminary experimentation. Note that as values for this parameter increase toward 100% the acceptance band method tends toward the same behavior as random choice.
2. *value-biased stochastic sampling (VBSS)* - This approach is based on the WHISTLING algorithm introduced in (Cicirello & Smith 2002). Like the acceptance band method, VBSS biases its choice based on heuristic value. In the case of VBSS, however, the probability of selecting a particular choice (task) is tied directly to the relative difference of its heuristic value and those of competing choices. Thus if one task is rated five times more flexible than a second task, then this task will have a five times greater probability of being retracted. One potential advantage of this scheme over the acceptance band approach is that no tasks are ever excluded from being selected in a given context; rather some are simply not very likely to be chosen. VBSS can be tuned through a bias parameter which affects the probability that the highest ranked task will be selected. At one extreme it is tuned so that VBSS approaches deterministic selection,and at the other random selection.

To provide a baseline for comparison we also define and evaluate a third, random retraction procedure.

Performance Results

Trials with stochastic sampling algorithms were conducted on the same 100 problem set as the previous experiments. In this case, though, the starting point was the end point of those experiments. The intent of these trials was aimed at determining how much the previous results could be improved upon, and what level of effort would be necessary to achieve this improvement.

The base retraction heuristic we chose was max-flexibility, with task, interval, and depth pruning all activated. We chose max-flexibility because it is the most efficient; our expectation is that we would obtain a comparable level of improvement in solution quality with any informed retraction heuristic.

Starting from the best initial deterministic state, we first ran *Iterative Deterministic* to quiessence on each of the 100 problems to see if it could improve on itself from the new state. These results were then compared to runs using acceptance band with a 10% band, vbss, and random choice. In these cases, the methods were given ten iterations, with early termination in the case where an optimal solution (zero unassignables) was found.

We arrived at ten iterations of stochastic search by doing a number of runs up to 100 iterations, and noting that no

	Unassignables		Runtime		Best Iteration	
	Avg.	Std.Dev.	Avg.	Std.Dev.	Avg.	Std.Dev.
Begin	21.83	n/a				
Random	12.71	0.14	961.41	31.85	4.11	0.28
VBSS	12.10	0.12	769.96	14.71	3.81	0.08
A-Band	12.68	0.07	689.46	12.95	2.97	0.17
It.Det.	14.28	n/a	107.76	2.89	0.35	0.00

Table 2: Extended Search Results

further progress was usually achieved by running more than ten iterations. This gives us an indication that we are probably converging to optimal solutions, but it is always possible that some further progress might be made. For each stochastic procedure, the results reported for each problem instance are the average of five trials.

To give a gross sense of the total time spent on this set of experiments, we ran 100 problems five times for each of four techniques, consuming over 350 cpu hours. (The experiments summarized in Table 1 took over 105 cpu hours.)

The results (See Table 2) show the following:

- At least in the test domain, using *Iterative Deterministic* to improve on itself is probably not the best choice. Although some improvement was shown on a number of problems, this improvement was at the expense of approximately 108 seconds on average. The final value of 14.28 average unassignables was still greater than what min-contention was able to achieve – 13.85 average unassignables in 75.19 seconds (Table 1) – without application of neighborhood search techniques.
- VBSS proved to be the best overall technique in terms of solution quality when applied for up to ten iterations, achieving a final average unassignable value of 12.10. The average time it took was 770 seconds, or over 12 minutes per problem. However, on average it achieved its best result by 3.8 iterations. This does not suggest running the method for four iterations rather than ten, as many good results would not be reached. What it does say is that improvement can be made incrementally, and if time is limited, the algorithm can be terminated prematurely without sacrificing some improvement.

Conclusions

In this paper we have revisted a recently developed task swapping procedure for improving oversubscribed schedules. The original work demonstrated promise for the method, MissionSwap, and touted an informed and fast retraction heuristic, *max-flexibility*. We have conducted a more in-depth analysis of the task swapping algorithm, in order to answer two questions:

1. Is it possible to improve the speed of the original algorithm without sacrificing solution quality?
2. Is it possible to improve the solution quality of the original algorithm given a reasonable amount of additional search?

The answer to both questions is “yes”. Through judicious pruning of the search space it was possible to achieve an order of magnitude speedup in solution time without sacrificing solution quality. Through application of iterated stochastic task swapping it was possible on average to assign 45% of the initial unassignable tasks, as opposed to at best a 37% reduction (using min-contention) without stochastic search.

Experiments in the design space of the MissionSwap algorithm have shown that max-flexibility as a retraction heuristic is not as clear a winner in solution quality as we had originally thought, however it remains by far the least expensive computationally.

We feel that the task swapping procedure is broadly applicable to other scheduling domains, and intend to test its applicability through further experimentation.

Acknowledgements

The work reported in this paper was sponsored in part by the Department of Defense Advanced Research Projects Agency (DARPA) and the US Air Force Research Laboratory under contracts F30602-00-2-0503 and F30602-02-2-0149, by the USAF Air Mobility Command under subcontract 10382000 to Northrop-Grumman Corporation, and by the CMU Robotics Institute.

References

Becker, M., and Smith, S. 2000. Mixed-initiative resource management: The amc barrel allocator. In *Proc. 5th Int. Conf. on AI Planning and Scheduling*, 32–41.

Bresina, J. 1996. Heuristic-baised stochastic sampling. In *Proceedings 13th national Conference on AI*, 271–278. Portland OR: AAAI Press.

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.

Harvey, W., and Ginsberg, M. 1995. Limited discrepancy search. In *Proceedings 14th International Joint Conference on Artificial Intelligence*.

Kramer, L., and Smith, S. 2003. Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems. In *Proceedings 18th International Joint Conference on Artificial Intelligence*.

Minton, S.; Johnston, M.; Philips, A.; and Laird, P. 1992. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 58(1):161–205.

Oddi, A., and Smith, S. 1997. Stochastic procedures for generating feasible schedules. In *Proceedings 14th National Conference on Artificial Intelligence*.

Russell, S., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Zweben, M.; Daun, B.; Davis, E.; and Deale, M. 1994. Scheduling and rescheduling with iterative repair. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. Morgan Kaufmann Publishers.