Maximizing Availability: A Commitment Heuristic for Oversubscribed Scheduling Problems

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 reconsider a "task-swapping" procedure for improving schedules in the face of resource oversubscription. Prior work has demonstrated that use of a retraction heuristic to determine which tasks to rearrange in an existing schedule allows for addition of new tasks which would otherwise fail to be scheduled. The existing task swap procedure employs a variable ordering heuristic for task insertion that is the same as the retraction heuristic, but scored in the reverse. That is, the least constrained tasks are retracted, and of these the most constrained are committed first. Value selection for commitment is defaulted to a task's earliest feasible start time. We have found that by applying a value selection heuristic, *max-availability*, to the choice of *where* to assign the retracted tasks, both solution quality and runtime performance of task swap can be improved greatly. Max-availability considers resource contention for an unassigned task and places it where availability is predicted to be maximal over the range of that task. This heuristic is applicable not only in a repair context, but can also promote resource levelling in the context of constructive task allocation. Finally, we show that use of max-availability in task swapping promotes schedule stability when compared to the prior greedy task insertion policy.

Introduction

In (Kramer & Smith 2003; 2004a), a task swapping algorithm was introduced as a mechanism for improving schedules in oversubscribed problem domains. The essence of the **TaskSwap** procedure is very simple: given a schedule in place and one or more tasks that are unable to be added to that schedule, a heuristic is employed to de-assign enough tasks so that the previously unschedulable task(s) can be scheduled. A retraction heuristic is used to select tasks based on a measure of their potential to be rescheduled elsewhere. If any of the retracted tasks fails to be rescheduled, the algorithm is applied recursively until it succeeds, or eventually fails with all competing tasks sampled. We have shown the **TaskSwap** procedure to be very effective in the context of large-scale mission scheduling problems.

Our previous analysis of task swapping has centered on improving the decision of which task(s) to retract, and little attention has been given to the mechanics of reinserting tasks back into the schedule. We have assumed a simple reinsertion strategy: place a retracted task at its earliest available start time. However, when many tasks are retracted this simple strategy can be quite myopic and can lead to insertion decisions that eliminate possibilities for inserting other retracted tasks.

In this paper we consider the potential for improving task swapping performance by making the task insertion decision more informed. In the spirit of prior scheduling work in the area of contention-based focus of attention, we use simple estimates of the overall resource demand of the set of retracted tasks, combined with current resource commitments, to detect intervals of best potential availability and make insertion decisions that tend to promote better task distribution over resources.

We demonstrate experimentally on the original problem set that our emphasis on an informed and efficient retraction heuristic was somewhat misplaced, and that crafting of a value commitment heuristic – *max-availability* – provides much better swap results with even random task retraction. In other words, when re-assigning retracted tasks, it is important not just to replace them anywhere, but to place them where they will best "fit together." The result of this intelligent commitment is that there are far fewer rescheduling failures for retracted tasks, thus less need for the task swap procedure to recurse, leading to better end results in a much less time. An additional benefit, is that by eliminating a good deal of recursion, we reduce the number of tasks that are swapped, and thus promote schedule stability.

We study several implementations of the *max-availability* heuristic itself, and our work shows that it is not only applicable to schedule augmentation and schedule repair, but also to schedule generation where resource contention is an issue.

Before examining max-availability as applied to task swapping, we begin by briefly summarizing the AMC (US Air Force Air Mobility Command) scheduling domain (Becker & Smith 2000) and the previously developed **TaskSwap** procedure.

Problem Context

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 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. We have developed an application, the AMC (Barrel) Allocator(Becker & Smith 2000), to address this problem. The approach adopted within the AMC Allocator 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. The task swapping procedure of (Kramer & Smith 2004a) 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

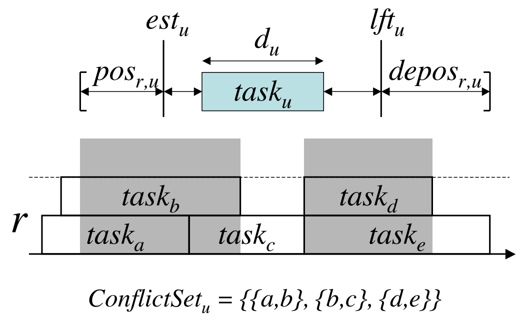

Figure 1: An unassignable task u and the conflict set generated for all intervals on resource, r.

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 conflict 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 **TaskSwap**, 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 reassign each

```
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.  ScheduleTask($task$)
6.  ScheduleInPriorityOrder($Retracted$, most-constrained-first)
7.  loop for ($i \in Retracted \wedge status_i = unassigned$) do
8.      $Protected \leftarrow$ TaskSwap($i, Protected$)
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 TaskSwap Search Procedure

retracted task, most constrained first (line 6). Tasks are assigned at their first feasible start time, and for those retracted tasks that remain unassignable, **TaskSwap** is recursively applied (lines 7-10). As a given task is inserted by **TaskSwap**, it is marked as protected, which prevents subsequent retraction by any later calls to **TaskSwap**.

```
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 3: InsertUnassignableTasks procedure

In Figure 3, top-level **InsertUnassignableTasks** procedure is shown. Once **TaskSwap** has been applied to all unassignable tasks, one last attempt is made to schedule any remaining tasks. This step attempts to capitalize on any opportunities that have emerged as a side-effect of **TaskSwap's** schedule re-arrangement.

Retraction Heuristics

The driver of the above repair process is the retraction heuristic **ChooseTaskToRetract**. In (Kramer & Smith 2003), three candidate retraction heuristics are defined and analyzed, 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[1]

$$Flex_i = \sum_{r \in Res_i} \frac{\textit{alloc-dur}_{r,i}}{ReqInt_{r,i}}$$

  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:

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

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

- **Min-Contention** - A more informed, contention based measure is one that considers 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 Experimental Results

The original experiments of (Kramer & Smith 2003) (carried out on a suite of 100 problems) demonstrated the efficacy of the **TaskSwap** procedure in the target domain. In this study, max-flexibility was shown to be the strongest performer; its application enabled **TaskSwap** to schedule, on average, 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.

[1]This formulation varies from the one presented in our earlier work (Kramer & Smith 2003; 2004a) where we compute the denominator of $Flex_i$ as $lft_i - est_i$, which did not take into account the duration of positioning and depositioning legs (setup duration). Given that the positioning durations vary by resource, but are constant per resource, a heuristic that takes them into account should be more informed. Our latest experiments have borne this out.

These results were expanded upon in later work (Kramer & Smith 2004a). We generated a comparable suite of 100 problems, using the same real-world problem data as a seed: five data sets of twenty problems each were generated, with the resource (wing) capacities randomly reduced from 0 to 10%, 0 to 20%, 0 to 30%, 0 to 40%, and 0 to 50% in successive data sets. Several search pruning techniques were tested, which demonstrated that the **TaskSwap** algorithm could be sped up significantly without sacrificing solution quality. With these pruning techniques in place, the min-contention retraction heuristic outperforms max-flexibility in solution quality, however the latter retained a significant runtime advantage.

Several iterated stochastic techniques were applied to the best solutions found in the 100-problem set, and it was shown that further modest gains in solution quality were achievable given a reasonable amount of time.

In (Kramer & Smith 2004b) we suggest problem structuring techniques that should allow the swap procedure to be applied successfully to several other classes of real-world scheduling problems.

Max-availability, an Insertion Heuristic for Task Commitment

As we have discussed, our prior work focused on heuristics for task retraction. A policy of "most constrained first" is a natural choice for task (variable) selection. Little attention was paid, though, to the issue of value selection for task commitment, the thought being that the extra computation would not be worth the effort. Tasks were assigned to their earliest feasible start time.

This scheduling policy is not arbitrary in that it has the benefit of reducing search and tends to produce schedules that are front-loaded, thus providing more opportunities for delay and rescheduling during execution. As we will see, though, front-loaded schedules have their downside in oversubscribed environments, as resource profiles are less level and less tasks can be accommodated in the schedule.

By employing a heuristic for value selection (task insertion), we were able to significantly improve our original results both in quality and performance. Before describing the max-availability insertion heuristic, we provide a motivating example, which will be referenced throughout the rest of the paper.

The Max-availability Heuristic: A Motivating Example

Consider the situation depicted in Figure 4, which represents the state of one resource R from $time_0$ through $time_{10}$ before three retracted tasks have been assigned. (I.e., just before line 6 in the TaskSwap procedure in Figure 2.) $Task_1$ has a duration of 5, and must schedule in the interval [0,8], $task_2$ has a duration of 4, and must schedule in the interval [3,10], and $task_3$ has a duration of 2 and must schedule in [2,7]. Each task requires one unit of capacity.

Resource R has a total capacity of 7 over the interval [0, 10][2], and as indicated in the figure, varying amounts of capacity are currently consumed over this interval. Four units over [0,3], five units over [3,5], six in the interval [5,6], and so on.

[2]In general in the AMC domain, total capacity for a resource may vary over time, but we have chosen not to represent that in this simple example.

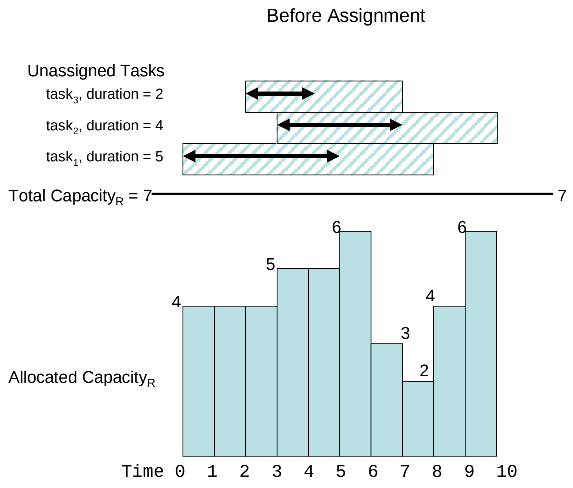

Figure 4: Retracted tasks before assignment

Assuming that we are inserting tasks in most constrained first order, we would schedule $task_1$ first, $task_2$ second, and finally $task_3$. As mentioned earlier, each task would be scheduled at its earliest feasible start time.

In our current example, after $task_1$ and $task_2$ are scheduled at their earliest start times, it is clear that $task_3$ can no longer be scheduled, as can be seen in figure 5, due to the capacity plateau of 7 in the interval [3,6]. (Technically speaking, $task_3$ may have other opportunities for assignment on alternate resources, but they could also be at capacity over its feasible window.) Is it possible that a simple capacity look ahead procedure could have avoided this dead end?

Taking into Account Actual Capacity

What we propose as capacity look ahead is a simple scan across a task's feasible window before assigning it. This scan collects capacity availabilities for each subinterval for each start time, ruling out intervals that include subintervals with zero capacity availability. We refer to these collections of capacity availabilities as *capacity-subinterval profiles*, and represent them as lists of n values, where n equals the duration of $task_i$.

As can be seen in Figure 6, $task_1$ has 3 units of capacity available over the interval [0,3] and 2 units available over [3,5]. Thus, starting at t_0, its capacity-subinterval profile can be represented as (3,3,3,2,2). Starting at t_1 its capacity-subinterval profile is (3,3,2,2,1). Starting at t_2 and t_3, respectively, the profiles are (3,2,2,1,4) and (2,2,1,4,5).

Given these profiles, where should $task_1$ be assigned? It is intuitive that we would want to assign a task where it has maximum resource availability, with the hope that this

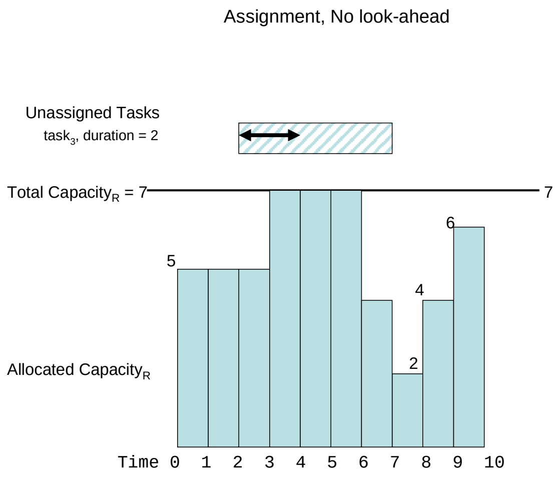

Figure 5: Lack of look-ahead has forced a dead end

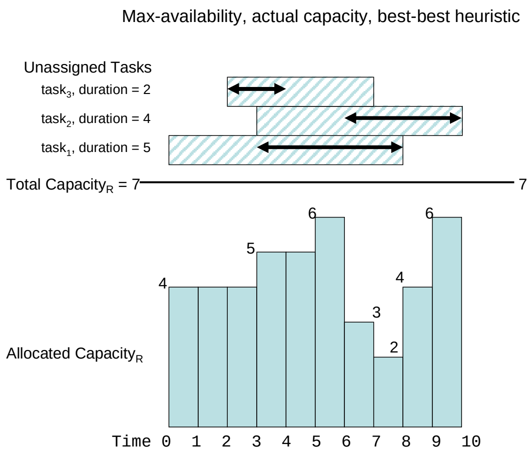

Figure 6: Taking into account actual capacity

will leave sufficient capacity for other tasks to be scheduled. We call the heuristic to choose a capacity-subinterval profile the max-availability heuristic. There are several possibilities for a max-availability metric. One is to select the capacity-subinterval profile with the maximum sum of availabilities; another is to select the capacity-subinterval profile that contains the maximum availability value.

We first examine the latter metric, referring to it as the *best-best* max-availability heuristic. In the case of ties between capacity-subinterval profiles, we choose the one with the maximum average availability. Figure 6 depicts the tasks after applying this max-availability heuristic to each in turn.[3] After $task_1$ is assigned at the end of its window to capture the availability peak [7 8], $task_2$ *must* assign at the end of its window, since that is the only interval not broken up by the capacity peak (zero availability) at subinterval [5,6]. At this point $task_3$ has two feasible start times, and picks the first based on the max-availability constraint. By applying this simple heuristic all three tasks were scheduled where only two were scheduled originally.

[3]Note that Figure 6 depicts capacity before assigning any of the tasks, but shows placement within their windows *after* assignment.

Applying max-availability to TaskSwap

To validate that the max-availability heuristic could aid the **TaskSwap** procedure, we incorporated it into the Ozone Scheduling Engine (Smith, Lassila, & Becker 1996) for selective use when executing the method ScheduleInPriorityOrder. **TaskSwap** itself was altered in no other way. We then re-ran the 100-problem set of (Kramer & Smith 2004a). The results were striking. Not only were 44% more tasks assigned over all problems, but run-time was cut by 40%.

Applying max-availability to the Initial Schedule

The surprising performance of **TaskSwap** incorporating max-availability led us to investigate its applicability to the task of generating an initial schedule. It is possible that this look-ahead procedure would not scale up to hundreds of missions and that max-availability might be thrown off early in schedule generation. The results were very encouraging. When used to generate initial schedules in our 100-problem set, a constant additional time factor was incurred compared to the original runs. It took the same amount of time to build schedules for the “harder” problems as it did for the “easier” problems. Furthermore, except for some of the problems in the 10% overallocation set, fewer unassignable tasks remained on average when generating schedules with max-availability heuristic.

Given these encouraging results, we re-ran the previous experiment, incorporating the max-availability/best-best heuristic in both the initial schedule generation and task swapping phases. The end results showed a 10% incremental improvement in quality (tasks assigned), however at the expense of an incremental 24% increase in run-time.

Taking into Account Potential Capacity

By design, the max-availability heuristic uses the existing capacity data structures in the Ozone scheduler, but in doing so, there is some task capacity that is unaccounted for: the “potential” capacity of those tasks that are yet to be assigned. It is possible that accounting for potential, or predicted, capacity could give the max-availability heuristic more guidance with little additional computational cost.

Accordingly, we extend the capacity representation to account for potential capacity by recording potential capacity in each time interval. Potential capacity is determined by aggregating the capacity usage of all unassigned tasks whose feasible windows intersect that interval. The potential capacity is tracked separately from actual allocated capacity.[4]

[4]This representation differs from that used in the ORR (Sadeh & Fox 1996) and SumHeight (Beck *et al.* 1997) resource contention measurements, where more capacity weight is given to resource intervals over which a task is more likely – or must – be scheduled. It is not yet clear whether incorporating this sophistication in our representation would be worth the extra computation.

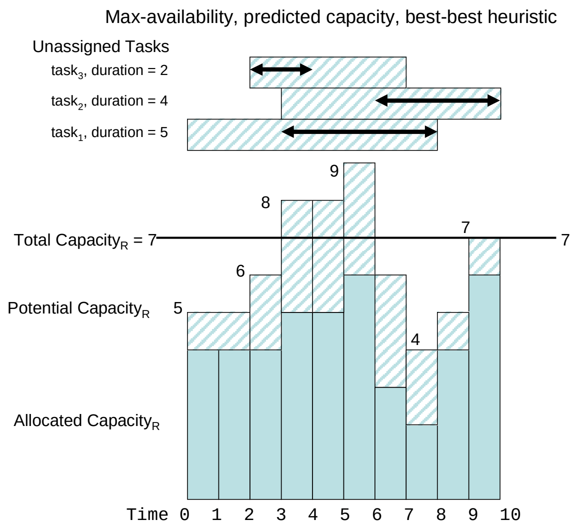

Figure 7: Taking into account potential capacity

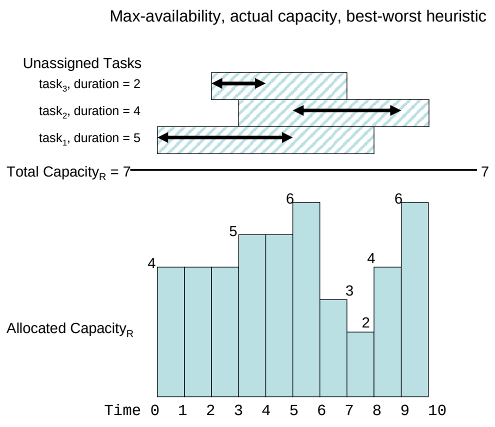

Figure 8: Using a best-worst case metric

Potential capacity usage is depicted in Figure 7, extending our original example.

Max-availability is defined as before, but now the availability value for a $subinterval_i$ in a capacity-subinterval profile is computed as $availability_i = TotalCapacity_R - (PotentialCapacity_i + AllocatedCapacity_i)$. As can be seen from the diagram, it is now quite possible to have availability values that are negative. For instance, the first capacity-subinterval profile for $task_1$ is (2,2,1,-1,-1). Searching for the best capacity-subinterval profile proceeds as before, and as before we only rule out intervals with zero actual capacity. As before (see Figure 7), we select the last feasible start time for $task_1$, and thus $task_2$ and $task_3$ are allocated the same as well.

As each task is assigned, actual capacity values it contributes to a resource are incremented, and *all* of the values it contributes to potential capacity are decremented by an amount equal to the capacity usage of the task. As more tasks are assigned and more capacity values converted from potential to actual, we would expect the max-availability heuristic to become more and more informed, until the last task to be assigned is just considering actual capacity.

While this particular example has not benefited from taking into account potential capacity, it is certainly possible that other task configurations could. To get a feel for this, we re-ran the 100-problem set. The results were somewhat encouraging. When potential capacity was considered in the max-availability heuristic in the both the initial and swap phases of scheduling, solution quality improved by 3% over using the basic max-availability heuristic (based on actual capacity alone), and runtime decreased by 25%.

Using a Best-Worst Case Metric

We have formulated max-availability as a search to find *the* interval that contains a subinterval of maximum available capacity, what we have called a best-best heuristic. In essence, this is a search for the “best valley” in terms of a capacity profile. Looking at the current example, though, what is the best valley could better be described as a narrow ravine – a deep depression with peaks on either side.

This leads to the notion that in searching for available capacity, it might also pay to avoid adjoining peaks. Formalizing this intuition produces a metric for max-availability that we call *best-worst*, defined as: when selecting a capacity-subinterval profile for over which to assign a given task, choose that one that has the greatest minimum value for availability. Again, we will use average availability to break ties.

Referring to figure 8, recall that $task_1$ can be assigned in intervals represented by the successive profiles (3,3,3,2,2), (3,3,2,2,1), (3,2,2,1,4) and (2,2,1,4,5). Of these, the first contains the greatest minimal value, 2, as all the other profiles encompass a value of 1. A max-availability best-worst heuristic suggests assigning $task_1$ at its earliest start time. Looking at the diagram, we see that we have found a valley – or actually, a plateau – in the [0,3] interval.

After $task_1$ is assigned, $task_2$ can be assigned in intervals represented by the capacity-interval profiles (1,1,1,4), (1,1,4,5), (1,4,5,3), and (4,5,3,1). All of these have the same greatest minimal value: 1. Breaking ties with an average, still leaves the last two choices tied, so we assume that the first of these – (1,4,5,3) – would be selected and $task_2$ would be assigned a starting time of t_5.

At this point $task_3$ has two available intervals in which it could be scheduled, but would be assigned at its earliest start time given a better average best-worst max-availability.

Testing the best-worst max-availability heuristic on the problem set produced the following results. When employed in both the initial and swap phases of scheduling 10% more tasks were assigned than the previous best result with a 16% improvement in runtime performance.

Combining Predicted Availability and Best-Worst Case Metric

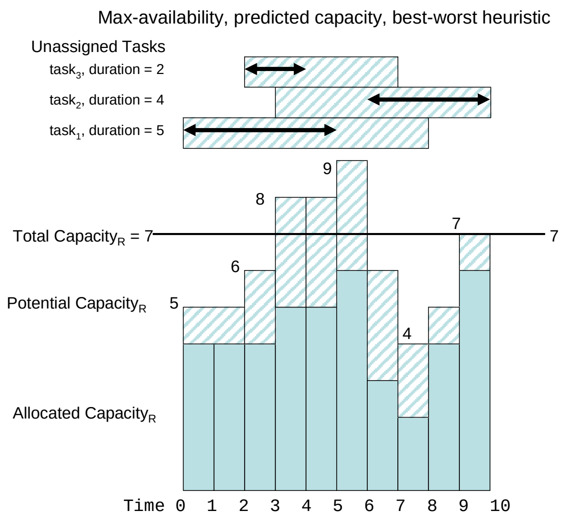

Figure 9: Combining predicted availability and best-worst case metric

Now, we return to the issue of predicted availability, and examine the best-worst metric in that light. Looking at Figure 9, we see that $task_1$ will be assigned again to its earliest start time. Now, however, $task_2$ will be assigned at its latest start time in an attempt to avoid the potential peak in the [5,6] interval, which still juts up above the total capacity line by one unit even after $task_1's$ potential capacity is converted to actual.

$Task_3$ will still prefer it's earliest start time, and be scheduled there, but it now has three total opportunities for assignment. In terms of resource levelling then, it is not hard to see that this is the optimal assignment policy for this particular resource/task configuration.

To see if the promise of this example bears out we again test the 100-problem suite, now with a best-worst max-availability constraint and potential capacity activated. First we run in just the swap phase, and then in the initial scheduling pass and swap phase. The results were very gratifying, particularly in the latter run. 16% more tasks were scheduled than when using actual capacity, with a 77% decrease in runtime.

By this point cumulative improvement over the results reported in (Kramer & Smith 2004a) amounted to an increase of 109% in average number of tasks assigned (7.99 vs. 16.8) and an order of magnitude decrease in average runtime from 76.2 seconds to 8.26 seconds.

Leximin Ordering

Use of a best-worst case, or "maximin," metric for the max-availability heuristic aims at making sure that one good capacity value doesn't drown out some very bad ones. Recall that in the case of best-worst ties between intervals, an average of capacity-subinterval values is taken to break the tie. A possibly better approach to evaluation fairness is to replace the maximin metric with a leximin ordering(Moulin 1988). This method ensures Pareto-optimality [5] by taking into account each value in the capacity profile, information that is lost in an average.

To understand how leximin ordering works, suppose that there were seven units of actual capacity allocated in the [0,1] interval in the example in Figure 9 (i.e. it is booked to capacity in this interval). Then the first capacity-subinterval profile would not have been generated for $task_1$ (since only *possibly* feasible ones are). The remaining three profiles then have the same best worst value of -2. Sorting the profiles numerically, we get (-2,-1,-1,1,2), (-2,-1,-1,1,1), and (-2,-1,-1,1,3). Comparing each individual value – i.e., a lexical comparison – we see that they are identical save for the last value, and since the last profile has the greatest value of 3, it would be first in a lexical ordering. Note that this profile would also be selected by an average calculation. However, suppose we were comparing the profiles (1, 3, 4) and (1, 7, 2). Given the tied maximin value of 1, the latter profile is on average greater than the former and would be selected. A lexical ordering, though, would indicate that 134 is greater than 127, so a leximin ordering would select the (1, 3, 4) profile.

We substituted a leximin metric for maximin (best-worse) in the max-availability heuristic, and reevaluated the experimental test set. Averaging all results, leximin outperformed maximin by 14% in run time and 8% in solution quality (tasks assigned).

Task Swapping and Schedule Stability

One issue that we have not touched on is that of schedule stability. In the AMC scheduling domain there are three aspects related to stability with respect to task swapping that should be mentioned. These are germane in that they apply to many, though not all, real-world scheduling domains. First is task priority. We take as a given that the process of swapping will not cause higher priority tasks to drop out of the schedule. Furthermore, task swapping will not even attempt to bump lower priority tasks to insert higher priority tasks into a schedule. In other words, we see bumping as an orthogonal issue, and take task swapping as purely additive.

Secondly, we require that any rescheduling of a task in the course of task swapping schedule that task within its feasible window. Schedule delay is not permitted. These requirements are enforced by the **TaskSwap** procedure.

Finally, it is *preferred* that the amount of change in some sense be minimized, or at least mitigated when inserting additional tasks into an existing schedule. Certainly we would prefer not to shuffle 100 tasks in a schedule to insert one additional task. From an operational scheduling perspective it would probably make sense to delay the additional task and keep the rest of the schedule constant. On the other hand, if minimal schedule change is desired, a task swap procedure that inserts *no* additional tasks would be optimal.

[5] A Pareto-optimal solution is one in which it is impossible to make any individual better off without making any other individual worse off.

Prob. Set	1	2	3	4	5	Q/C Ratio
ICAPS-04 Time	16.80	21.85	26.25	30.85	37.75	30%
ICAPS-05 Time	0.75	4.05	8.30	20.15	30.80	59%
04 Resource	7.85	14.00	19.75	22.90	29.90	42%
05 Resource	0.75	4.05	8.30	20.15	30.80	59%
04 AggTime	16.29	34.82	56.93	65.25	74.67	16%
05 AggTime	1.20	5.19	8.46	20.91	31.70	56%

Table 1: Solution Stability

Given a desire to maximize the number of tasks inserted into a schedule, while at the same time minimizing the extent of change, we recorded change over all experimental runs that we have reported on, beginning with our prior ICAPS (Kramer & Smith 2004a) results. These results are summarized in table 1. We tracked three change metrics: number of tasks that shifted in time (rows 1 and 2 in table 1 compare ICAPS-04 results with ICAPS-05), number of tasks that changed resource (rows 3 and 4), and aggregate amount of time shifted in days (rows 5 and 6). As time change and resource change are not comparable values, we did not attempt to come up with one aggregate number for change. To compute the values in the final column of the table, the Q/C (quality to change) ratio, we divided the number of tasks that were added to the schedule (that is beginning tasks minus end unassignables) for all problem sets, divided by the change metric. Therefore higher values are taken to be better as they reflect more tasks added per amount of change.

What we conclude from this table is that not only did the max-availability heuristic benefit task swapping in terms of tasks added and runtime, it also mitigated the amount of schedule change in doing so.

Related Work

There is a reasonable body of work on contention-based heuristics for scheduling. Some approaches, classified as "texture-based" (Sadeh & Fox 1996; Beck *et al.* 1997), use a probabilistic measure of resource contention to select the best resource to assign, and the best start time to assign a task. These methods are used in the context of unit-capacity resources, and it is not clear if they would scale to a large, multi-capacity domain. Other techniques, for example (Nuitjen & Aarts 1996), do consider the problem of multi-capacity resources. Their powerful "edge-finding" propagation techniques can provide even stronger guidance to task placement, but are quite computationally intensive. Another approach, that of (Giuliano 1997), is perhaps the closest philosophically to ours. Decisions as to where to insert a task in the schedule are based on an informed estimate of overall resource contention along that task's timeline, and don't rely on expensive task-to-task contention analysis.

For some multi-capacity, multi-resource oversubscribed scheduling problems, (Barbulescu, Howe, & Whitley 2004) claim that time spent in analyzing individual task placement may not be worth the effort, and that large scale change ("leaping before you look") at each choice point may be the best policy in avoiding large plateaus in the search space. Our experience in the AMC scheduling domain has proven otherwise, as augmentation of the **TaskSwap** procedure with the max-availability task insertion heuristic has both provided for quicker and better solutions, but has also mitigated the extent of schedule change. In one sense, though, our results do not contradict the work of Barbulescu and colleagues, as each iteration of **TaskSwap** procedure is designed to shuffle multiple tasks – possibly many – to add one additional task to an existing schedule.

Conclusions

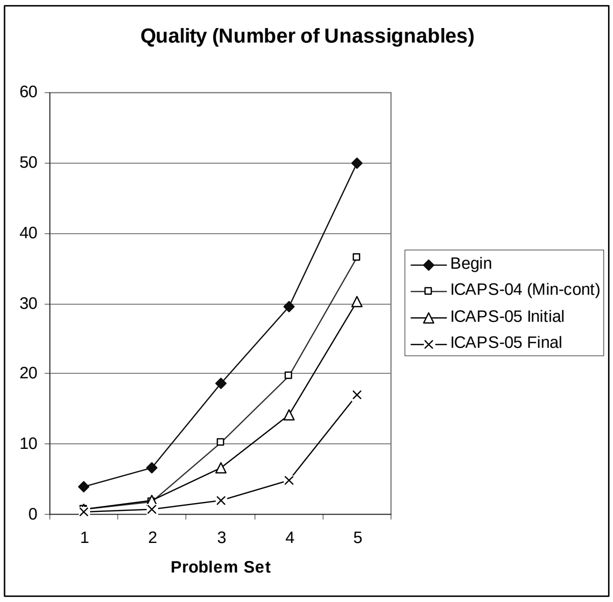

Figure 10: Solution Quality

In this paper we have described the application of a commitment heuristic, max-availability, to guiding a task-swapping procedure in inserting additional tasks in an over-subscribed, multi-capacity problem. A synergistic result of this work is the applicability of this heuristic to constructive schedule generation. The end results of our experiments are summed up powerfully in the two graphs in figures 10 and 11.

In terms of solution quality – i.e., minimizing the number of tasks left unassigned – results from our current work completely dominate the work reported in (Kramer & Smith 2004a). ICAPS-05 "Initial" refers to the simple look ahead of actual capacity using the best-best max-availability heuristic during the swap phase. ICAPS-05 "Final" tracks a combination of using the leximin-ordered max-availability heuristic with predicted capacity in both the initial and swap phases of scheduling.

The computational performance gains in figure 11 are similarly impressive, reflecting an order of magnitude speed-up on average from ICAPS-04 to ICAPS-05 Final.

Much of our prior work on task swapping concentrated on studying which retraction heuristic produced gains in quality or performance over various problem types and with various

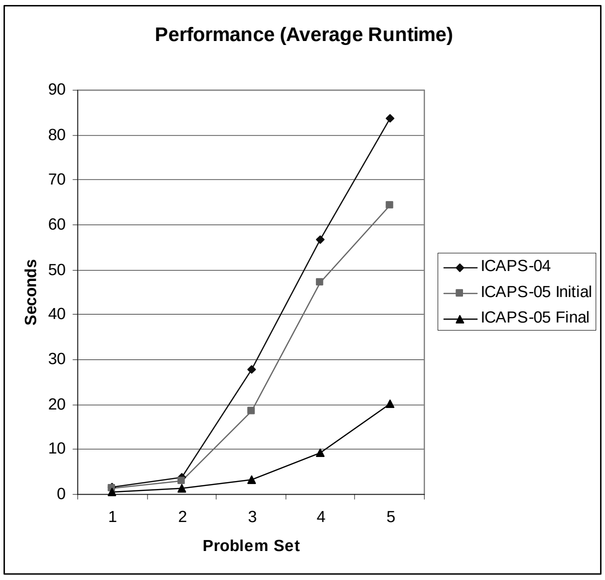

Figure 11: Solution Performance

pruning techniques. We have elided the details of individual retraction heuristic performance in this current work – although they were tested in each succeeding scenario – for the simple reason that as the commitment techniques of max-availability grew more and more sophisticated, the importance of the retraction heuristic employed became less significant, until in the final analysis a random retraction policy was usually just as good as min-contention, min-conflicts, or max-flexibility.

This may be a product of the AMC scheduling domain itself, and other problem domains may benefit from the retraction heuristic. This points to one area of future work that we began to explore in (Kramer & Smith 2004b), and intend to continue to extend: investigation of the application of task swapping in other oversubscribed scheduling environments.

Application of the max-availability heuristic has raised a number of questions that we have yet to answer. Would a more sophisticated capacity prediction, for instance SumHeight, lead to better results? Is a best-worst policy always better than best-best, or is it better to trade off between them as the scheduling process progresses? Does one dominate the other in terms of various problem classes?

Prior work in the area of contention-based heuristics has been somewhat deficient in terms of problems with multi-capacity resources. We feel that the research that we have begun is a small step in the direction to remedy that.

Acknowledgements

The work reported in this paper was sponsored in part by 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, by the NSF under contract 9900298, and by the CMU Robotics Institute. The authors would also like to thank the anonymous reviewer that pointed us in the direction of exploring a leximin ordering operator as a possible improvement to maximin.

References

Barbulescu, L.; Howe, A. E.; and Whitley, L. 2004. Leap before you look: An effective strategy in an oversubscribed scheduling problem. In *Proc. 19th National Conference on Artificial Intelligence (AAAI-04)*.

Beck, J. C.; Davenport, A. J.; Sitarski, E. M.; and Fox, M. S. 1997. Texture-based heuristics for scheduling revisited. In *Proc. 14th National Conference on Artificial Intelligence (AAAI-97)*.

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.

Giuliano, M. 1997. Achieving stable observing schedules in an unstable world. In *Proc. Seventh Annual Conference on Astronomical Data Analysis Software and Systems (ADASS '97)*.

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

Kramer, L. A., and Smith, S. F. 2004a. Task swapping for schedule improvement, a broader analysis. In *Proc. 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*.

Kramer, L. A., and Smith, S. F. 2004b. Task swapping: Making space in schedules in space. In *Proc. Fourth International Workshop on Planning and Scheduling for Space (IWPSS-04)*. Darmstadt Germany: European Space Agency.

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.

Moulin, H. 1988. *Axioms of Cooperative Decision Making*. Cambridge University Press.

Nuitjen, W. P. M., and Aarts, E. H. L. 1996. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research* 90:269–284.

Sadeh, N., and Fox, M. 1996. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence* 86(1):1–41.

Smith, S.; Lassila, O.; and Becker, M. 1996. Configurable, mixed-initiative systems for planning and scheduling. In Tate, A., ed., *Advanced Planning Technology*. Menlo Park: AAAI Press.

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.