An Effective Algorithm For Project Scheduling With Arbitrary Temporal Constraints

Tristan B. Smith
CIS Department
University of Oregon
Eugene, OR 97403
tsmith@cs.uoregon.edu

John M. Pyle
On Time Systems, Inc.
1850 Millrace Drive, Suite 1
Eugene, OR 97403
john@otsys.com

Abstract

The resource-constrained project scheduling problem with time windows (RCPSP/max) is an important generalization of a number of well studied scheduling problems. In this paper, we present a new heuristic algorithm that combines the benefits of squeaky wheel optimization with an effective conflict resolution mechanism, called bulldozing, to address RCPSP/max problems. On a range of benchmark problems, the algorithm is competitive with state-of-the-art systematic and non-systematic methods and scales well.

Introduction

Project scheduling requires the assignment of resources and times to activities in a plan with the goal of makespan minimization. Such problems arise in a range of fields including construction, manufacturing, software development and a number of space applications. Because of their importance, many varieties of project scheduling problems have been well studied by the OR and AI communities.

Less attention has been paid to RCPSP/max problems. These generalize many of the well-known and well-studied classes of scheduling problems (including job-shop problems and their variants) through non-unary resource constraints and, most importantly, arbitrary temporal constraints. The latter permit both minimum and maximum time lags between activities to be represented. This allows a vast number of real-world constraints to be modelled, including fixed start times, releases and deadlines, set-up times, minimum and maximum overlaps, activities that must start or finish simultaneously, and shelf life constraints.

RCPSP/max problems have been tackled by both the AI and OR communities with most results achieved by the latter (Dorndorf, Pesch, & Phan-Huy 2000; Cesta, Oddi, & Smith 2002; Neumann, Schwindt, & Zimmerman 2003). Successful exact methods incorporate constraint propagation into a variety of branch and bound algorithms. Since these algorithms have limited scalability, a number of non-systematic methods, including tabu search, simulated annealing and genetic algorithms have also been developed.

In this paper, we describe how another heuristic approach, squeaky wheel optimization (SWO), can be applied effectively to RCPSP/max problems. The main component of SWO is a priority-based greedy schedule constructor. Search is used to find effective prioritizations for that constructor.

Arbitrary temporal constraints can cause problems for a greedy constructor since it is NP-hard to find any feasible schedule; if most prioritizations do not yield feasible schedules, SWO will be ineffective. Therefore, to increase the power of the greedy constructor, we have added a new conflict resolution mechanism, called bulldozing that allows the greedy constructor to move sets of activities later in a partial schedule to maintain feasibility. Finally, similar mechanisms move sets of activities earlier in a feasible schedule to allow shorter schedules to be discovered.

Our algorithm is compared with the best reported results in the OR and AI literature for benchmark RCPSP/max problems ranging from 10 to 500 activities. Competitive for all problem sizes, it begins to outperform most algorithms as problem size increases.

The RCPSP/max Problem

An RCPSP/max problem contains a set $\{A_1, ..., A_n\}$ of activities and a set $\{R_1, ..., R_m\}$ of resources. Each activity A_i has a start time S_i and a duration d_i. Each resource has a maximum capacity c_k and execution of each activity A_i requires an amount r_{ik} of resource k for each time unit of execution. Binary precedence constraints $[T_{i,j}^{min}, T_{i,j}^{max}]$ enforce minimum ($T_{i,j}^{min}$) and maximum ($T_{i,j}^{max}$) time lags between the start times of activities A_i and A_j.

The two constraints types of an RCPSP/max problem can be summarized as follows:

- **Precedence Constraints:** $T_{i,j}^{min} \leq S_j - S_i \leq T_{i,j}^{max}$
- **Resource Constraints:** $\Sigma_{\{i|S_i \leq t < S_i + d_i\}} r_{ik} \leq c_k, \forall t, \forall k$

A schedule is an assignment of a value to the start time S_i of each activity. A schedule is *time-feasible* if it satisfies all precedence (temporal) constraints and is *resource-feasible* if it satisfies all resource constraints. A schedule that satisfies all constraints is *feasible*.

The goal of an RCPSP/max problem is to find a feasible schedule such that the project makespan, $MK = max_{1 \leq i \leq n}\{S_i + d_i\}$, is minimized. Unlike scheduling problems without both minimum and maximum time lags, even finding a feasible schedule for the RCPSP/max problem is NP-hard (Bartusch, Möhring, & Radermacher 1988).

Solving RCPSP/max Problems

Time Windows and Constraint Propagation

At the core of our algorithm is a greedy schedule constructor that selects a start times for each activity from a domain of possible values. We represent each such domain with a *time window*, $[ES_{A_i}, LS_{A_i}]$, representing all time points between the earliest and latest start times we wish to consider.

Temporally consistent time windows are efficiently maintained via well known constraint propagation techniques. In our algorithm, each window is initialized to $[0, horizon - d_i]$ (where *horizon* is the goal makespan) and then temporal constraint propagation achieves arc-consistency (Dechter, Meiri, & Pearl 1991).

Windows are updated during schedule construction. When a start time t is selected for an activity A_i, the PLACEACTIVITY(A_i,t) method will reduce windows as necessary to maintain temporal consistency. Similarly, when a start time is retracted (as we do in bulldozing), the UNPLACEACTIVITY(A_i) method will expand windows if possible. These methods allow schedule construction to always produce time feasible schedules without the need for backtracking (Dechter, Meiri, & Pearl 1991).

While the above propagation considers only precedence constraints, the propagation of resource constraints can further reduce start time domains (Muscettola 2002; Laborie 2003).While some resource constraint propagation is clearly beneficial, it comes at a computational cost and limits the scalability of algorithms.

The algorithms we describe in this paper explicitly avoid resource constraint propagation as our intent is to show that search can be effective without it and to highlight the differences between our algorithm and others that have been proposed. However, there is no inherent reason for us to avoid resource constraint propagation and it could complement our approach.

Squeaky Wheel Optimization

Squeaky Wheel Optimization (SWO) is an iterative approach to optimization that combines a greedy algorithm with a priority scheme (Joslin & Clements 1999). A number of results suggest that SWO can be applied effectively to a range of real-world problems and scales well (Joslin & Clements 1999; Chen, Fu, & Lim 2002).

Each iteration of the SWO algorithm can be divided into two stages: construction and prioritization. The construction stage takes a variable ordering and builds a solution using a greedy algorithm. In the prioritization stage, a variable is penalized with 'blame' depending on how well that variable was handled during construction. The updated priorities result in a new variable ordering for the next iteration.

The key to SWO is that elements that are handled poorly by the greedy constructor have their priority increased and are therefore handled sooner (and hopefully better) the next time ("The squeaky wheel gets the grease"). Over time, elements that are difficult to handle drift toward the top of the queue, those that are always easy to handle drift toward the bottom, and the rest settle somewhere in between.

A SWO implementation that uses the time window framework for the RCPSP/max problem, embedded in a binary search over possible horizons, is outlined here:

```
SWO($MaxIter$)
1  $counter \leftarrow 1$
2  $MK_{best} \leftarrow +\infty$
3  SETHORIZONLOWERBOUND
4  $(P_1, ..., P_n)$ = initial priorities
5  for $counter \leftarrow 1$ to $MaxIter$
6      do $feasible \leftarrow$ TRUE
7         for $i \leftarrow 1$ to $n$
8             do
9                $A_i \leftarrow$ unscheduled activity with
                     highest $P_i$
10               if SCHEDULE($A_i$) fails
11                 then $feasible \leftarrow$ FALSE
12        if $feasible$
13          then $MK_{best} \leftarrow MK_{current}$
14               DECREASEHORIZON
15        if 10 iterations without feasible solution
16          then INCREASEHORIZON
17        $(P_1, ..., P_n)$ = updated priorities
```

Line 3 and lines 14 through 16 perform a version of binary search over possible makespans, ranging from the resource-unconstrained lower bound to double the trivial upper bound.[1] Although not included in the pseudocode, our algorithm will quit if the minimum makespan is reached.

Line 4 initializes the priorities. Lines 7 to 11 make up the greedy schedule constructor and priorities are updated on line 17.

The basic SCHEDULE(A_i) method is outlined below. It looks for the earliest time-feasible and resource-feasible place to start A_i. If none is found, resource constraints are ignored and A_i is put at the earliest time-feasible start time ES_{A_i}.

```
SCHEDULE($A_i$)
1  $t \leftarrow$ earliest resource-feasible time for $A_i$
       in $[ES_{A_i}, LS_{A_i}]$
2  if t = NIL
3     then PLACEACTIVITY($A_i, ES_{A_i}$)
4          return FALSE
5     else PLACEACTIVITY($A_i, t$ )
6          return TRUE
```

Example 1 *Consider the following three activity, one resource problem:*

- *Durations* $d_1 = d_2 = d_3 = 2$
- *Resource capacity* $r_1 = 2$
- *Resource requirements* $r_{11} = r_{21} = r_{31} = 1$

[1] $ub = \Sigma_{i=1}^{n} max(d_i, max(T_{i,j}^{min}))$ (Dorndorf, Pesch, & Phan-Huy 2000). We found that doubling this enabled bulldozing to find feasible schedules more easily.

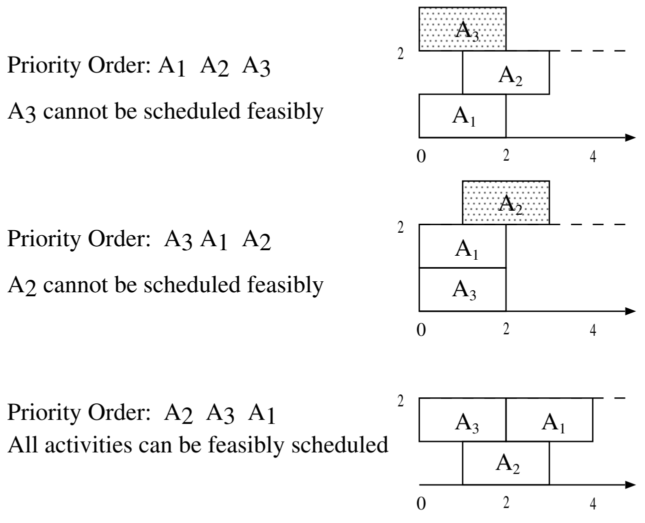
Priority Order: A1 A2 A3
A3 cannot be scheduled feasibly
A3
A2
A1
Priority Order: A3 A1 A2
A2 cannot be scheduled feasibly
A2
A1
A3
Priority Order: A2 A3 A1
All activities can be feasibly scheduled
A3
A1
A2
0
2
4

Figure 1: The third iteration of SWO finds a feasible schedule. For illustration, an activity is moved to the beginning of the priority queue when it cannot be scheduled feasibly.

- *A constraint that A_3 must start exactly 1 unit before A_2. That is, $S_3 - S_2 = -1$ (or $T^{min}_{2,3} = T^{max}_{2,3} = -1$).*

Figure 1 demonstrates SWO beginning with the priority order (A_1, A_2, A_3). Using this order, no resource-feasible time is available for A_3, its priority is increased and the priority order (A_3, A_1, A_2) is obtained. On the second iteration, A_2 has the same problem. Finally, in the third iteration, the order (A_2, A_3, A_1) leads to a feasible schedule with $MK = 4$. SWO will then reduce the horizon to 3, recompute time windows and continue.[2]

For initial priorities, we have chosen to use the LS_{A_i} values calculated by the temporal constraint propagation (where the activity with the earliest LS_{A_i} value gets the highest priority).

In our implementation, the priorities of activities that are not scheduled feasibly are increased by a constant amount. To add randomness, priorities of other activities are increased by a smaller amount with a small probability. Since binary search ensures that the horizon is less than the best makespan found, we know that on each iteration either a new best schedule will be found or at least one activity will have its priority changed.

Bulldozing

In Example 1, we see how SWO can find a feasible schedule by getting the three activities in the right priority order. For large or highly constrained problems, there can be many subproblems of this nature and it may be difficult for SWO to get all of them right at the same time.

To avoid this potential problem and strengthen the greedy construction, we add a conflict-resolution method called

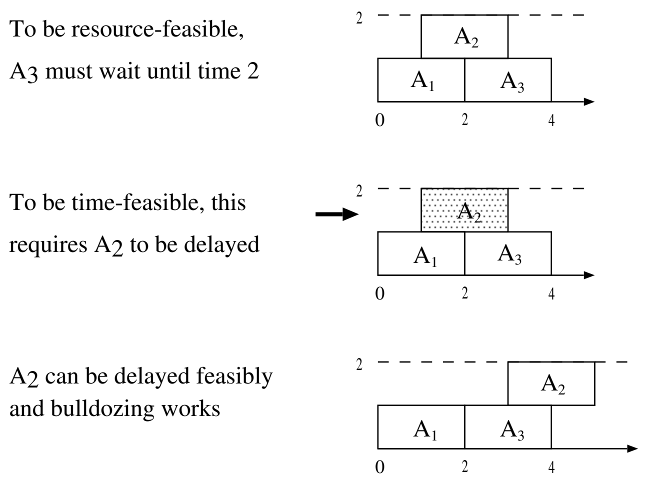
To be resource-feasible,
A3 must wait until time 2
A2
A1
A3
To be time-feasible, this
requires A2 to be delayed
A2
A1
A3
A2 can be delayed feasibly
and bulldozing works
A2
A1
A3
0
2
4

Figure 2: Bulldozing.

bulldozing to the SCHEDULE method.

Bulldozing is inspired by the complication of maximum time lags. Without maximum time lags, a greedy algorithm can always find a feasible schedule since each A_i can be postponed as long as necessary until resources are available. When there are maximum time lags, however, A_i may not be able to be postponed long enough if activities that constrain it have been placed 'too early' (consider the first iteration of Example 1 where A_3 cannot be postponed long enough since it is constrained by A_2).

If such an A_i is encountered, bulldozing will attempt to delay the activities that constrain it so that A_i can be scheduled in the postponed position.

Example 2 *Consider again the simple problem of Example 1. Figure 2 shows what happens if we add bulldozing to the constructor. Activities A_1 and A_2 are placed successfully. When A_3 is considered there is no feasible start time. Times $t \geq 2$ are resource-feasible but only time $t = 0$ is time-feasible since A_3 is constrained by the chosen start time for A_2. Therefore, we place A_3 at 2 and bulldoze A_2. Since A_2 can then be feasibly scheduled at time 3, bulldozing is successful and the resulting schedule is feasible.*

We get algorithm $SWO(B)$ by replacing SCHEDULE(A_i) with SCHEDULEWITHDOZING(A_i, X), a recursive procedure outlined below (X is the set of activities that must be delayed; in the initial call, $X = \emptyset$). Instead of simply searching the current window $[ES_{A_i}, LS_{A_i}]$ for a feasible time, SCHEDULEWITHDOZING considers the larger interval $[ES_{A_i}, LS^{orig}_{A_i}]$ (where $LS^{orig}_{A_i}$ is the latest start of A_i when no other activities are scheduled). If t is not a feasible time or is before LS_{A_i}, the algorithm proceeds as before.

However, if the earliest resource-feasible time is outside of the current window but within the original window, bulldozing is invoked. Activity A_i is placed at this potential start time and line 9 updates set X with the activities that must be moved.[3] Each A_j in this set is then unplaced (one

[2]Notice that this schedule is optimal. However, with only temporal constraint propagation, a schedule of length 3 cannot be ruled out and SWO will continue to try to find one.

[3]These will be activities that forced $LS_{A_i} < t$ in the first place.

at a time) and a recursive SCHEDULEWITHDOZING(A_j,X) call is made. If all activities can find new feasible start times, bulldozing succeeds. Otherwise, all delayed activities revert to their previous positions and A_i is placed back at ES_{A_i}.

```
SCHEDULEWITHDOZING($A_i, X$)
 1  $t \leftarrow$ earliest resource-feasible time for $A_i$
         in $[ES_{A_i}, LS_{A_i}^{orig}]$
 2  if t = NIL
 3     then PLACEACTIVITY($A_i, ES_{A_i}$)
 4          return FALSE
 5     else if $t \leq LS_{A_i}$
 6             then PLACEACTIVITY($A_i, t$)
 7                  return TRUE
 8             else PLACEACTIVITY($A_i, t$)
 9                  $X \leftarrow X \cup \{A_j$ forced to move by $A_i\}$
10  while $X \neq \emptyset$
11      do $A_k \leftarrow$ randomly selected element of $X$
12         UNPLACEACTIVITY($A_k$)
13         if SCHEDULEWITHDOZING($A_k, X$) fails
14            then UNDOALLBULLDOZING
15                 UNPLACEACTIVITY($A_i$)
16                 PLACEACTIVITY($A_i, ES_{A_i}$)
17                 return FALSE
```

The recursive nature of bulldozing means that more activities can be moved than the original set forced by A_i. In fact, we have observed that A_i itself is often bulldozed further in attempts to settle on start times where all activities are resource-feasible. The reason seems to be that a highly constrained sub-problem may need to be moved out past other activities to fit. This is the motivation for selecting activities to bulldoze randomly on line 11; if a subset of activities proves difficult to schedule, attempts to reschedule them will be done in different orders.

This suggests a nice feature of bulldozing. If a problem has subsets of activities that are highly constrained, the subsets will be pushed out past other activities until they can be feasibly scheduled. It is interesting to note that in other work (Neumann, Schwindt, & Zimmerman 2003), RCPSP/max problems are explicitly divided into such subproblems that are solved separately and then combined into an overall solution. Bulldozing similarly isolates subproblems from the rest of the problem but does so only when subproblems prove difficult.

If there is a difficult subproblem, SWO without bulldozing may also work because the activities involved are likely to move up in the priority queue and be scheduled before others. However, if there are multiple such subproblems, they will likely be jumbled together in the early part of the priority queue and SWO may have trouble fitting them all together. Bulldozing appears to overcome this problem.

Bulldozing can be considered a form of intelligent backtracking (Ginsberg 1993) since we unvalue and revalue the variables whose values contribute to an infeasible state. It also has the flavor of iterative repair algorithms that build schedules and then eliminate conflicts through local search. However, a key difference with local repair is that bulldozing is done with partial rather than complete schedules.

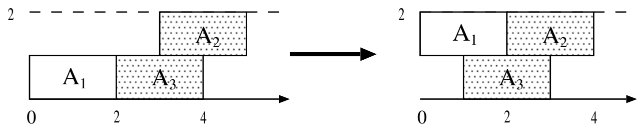

Figure 3: With refilling, A_2 and A_3 are bulldozed left and the makespan is reduced from 5 units to 4.

Refilling

Consider our simple example once more. In Example 2, we see how bulldozing can allow feasible schedules to be constructed with priority queues that result in infeasible schedules without bulldozing. However, notice that the resulting schedule is not optimal. As shown in Figure 3, the two activities involved in bulldozing (A_2 and A_3) can be shifted left to fill in the space vacated by the bulldoze.

Algorithm $SWO(BR)$ results from adding two such 'refilling' pieces to the SCHEDULEWITHDOZING procedure:

1. **Left Bulldozing** After a successful bulldoze, an attempt is made to bulldoze the same set of activities back to earlier start times. This often works because of the space vacated by the bulldozed activities. This step is also bulldozing because the subset of activities considered may be highly constrained among themselves (hence the bulldoze in the first place). Therefore, it may not be possible to left-shift any of them individually but they may move as a group.
2. **Left Shifting** After a successful bulldoze, an attempt is made to left-shift activities that can take advantage of resources vacated by bulldozed activities. We simply consider each activity A_i for which $S_i > ES_{A_i}$.

These refilling mechanisms are able to reduce the makespan of an already feasible schedule or partial schedule. When the horizon is small, they may also indirectly help the construction of more feasible schedules since they leave more room for activities that have yet to be scheduled.

Example 3 *In the simple problem of Example 1, there are 6 possible priority orders. For SWO, 2 of them are both feasible and optimal. For $SWO(B)$, all 6 become feasible and 4 are optimal. Finally, for $SWO(BR)$, 6 are feasible and 5 are optimal.*

It is straightforward to show that each version of our algorithm is guaranteed to terminate in polynomial time and therefore cannot be guaranteed to find a feasible solution (since the problem is NP-hard).

Experimental Results

We have tested our algorithms on five sets of benchmark problems (set A is divided into 3 subsets). All benchmarks were generated with ProGen/Max (Schwindt 1995) using a number of parameters to vary problem characteristics. Table 1 lists the benchmarks with the number of instances and how many are feasible as well as the number of activities and resources of each instance.

Our algorithm has been implemented in C++ and was run on a 1700 MHz Pentium 4. It uses well under 15 MB of memory on the largest problems.

Table 1: Benchmark names and parameters

Set	$N_{instances}$	N_{feas}	N_{act}	N_{res}
J10	270	187	10	5
J20	270	184	20	5
J30	270	185	30	5
B	1080	1059	100	5
C	120	119	500	5

Table 2: Results of schedule construction with all priority queues for feasible *J10* problems

	$\%_{feas}$			$\%_{opt}$		
Constructor	worst	avg	best	worst	avg	best
SWO	0	4	54	0	2	53
$SWO(B)$	18	94	100	0.0002	7	100
$SWO(BR)$	15	94	100	0.02	13	100

Quality of the Greedy Constructor

For a SWO algorithm to be effective, it is crucial that the greedy constructor be capable of finding feasible and optimal solutions. We want to be confident that there are at least some (preferably many) priority orders from which optimal schedules will be produced. The *J10* problems are small enough that all $10! = 3628800$ possible priority queue permutations can be tried for each one.

In table 2, we see what percentage of those queues lead to feasible and optimal schedules using the constructors of our three versions of SWO. For both $\%_{opt}$ and $\%_{feas}$, we give the worst and best cases among all 187 feasible problems in *J10* as well as the average.

The benefits of strengthening the constructor are clear; bulldozing yields an enormous jump in the number of feasible schedules produced while refilling yields another jump in the number of optimal schedules produced. While the SWO constructor fails to find any solution to 3 feasible problems, the other two find feasible solutions with at least 15% of the prioritizations for every feasible problem.[4]

Another consideration, of course, is time. On average, the $SWO(B)$ constructor took 3.2 times longer than that of SWO and the $SWO(BR)$ constructor took twice again as long. Experiments suggest that these computational costs are worth paying.

Benchmark Results

We compare $SWO(BR)$ with the best methods reported in the literature:

- $\mathbf{B\&B_{S98}}$ (Schwindt 1998) is a branch and bound algorithm that delays activities by adding precedence constraints.
- $\mathbf{B\&B_{D98}}$ (Dorndorf, Pesch, & Phan-Huy 2000) is a branch and bound algorithm that reduces the search space with a significant amount of resource constraint propagation.
- $\mathbf{B\&B_{F99}}$ (Fest *et al.* 1999) is a branch and bound algorithm that dynamically increases the release dates of some activities.
- **ISES** (Cesta, Oddi, & Smith 2002) is a heuristic algorithm that begins with all activities scheduled as early as possible and then iteratively finds and levels "resource contention peaks", by imposing additional precedence constraints and restarting.
- **KDE** (Cicirello 2003) improves upon priority rule methods by using models of search performance to guide heuristic based iterative sampling.

[4]It is unclear to us why the $SWO(BR)$ constructor did worse than the $SWO(B)$ one in the worst case.

Table 3: Results for benchmark problems

Set	Algorithm	$\Delta_{LB}\%$	$\%_{opt}$	$\%_{feas}$	Scaled Cpu_{sec}
J10	$SWO(BR)$	0.2	94.0	100	0.31
	$B\&B_{S98}$	0.0	100	100	-
	ISES	1.3$^{(a)}$	85.9	99.5	0.08
J20	$SWO(BR)$	4.9	66.4	100	0.63
	$B\&B_{S98}$	4.3	85.3	100	-
	ISES	5.4	64	100	0.53
J30	$SWO(BR)$	10.3	51.1	100	1.07
	$B\&B_{S98}$	9.6$^{(a)}$	62.3	98.9	-
	ISES	11.0	49.4	100	2.67
B	$SWO(BR)$	6.8	64.2	100	1.85
	KDE	4.6$^{(a)}$	63.8	99.9	1.97
	ISES	8.0$^{(a)}$	63.2	99.9	$^{(b)}$
	$B\&B_{S98}$	9.6	63.7	100	-
	$B\&B_{D98}$	4.6	73.1	100	3.49
	$B\&B_{F99}$	7.0	72.5	100	$^{(b)}$
C	$SWO(BR)$	0.5	79.2	100	3.27
	$B\&B_{D98}$	0.5$^{(a)}$	71.4	97.5	11.53
	$B\&B_{F99}$	5.2	58.8	100	$^{(c)}$

(a) Not directly comparable to other numbers since problems not feasibly solved are excluded.
(b) Cutoff of 11.8 seconds used but no average time reported.
(c) Cutoff of 23.6 seconds used but no average time reported.

For each of the above, results are available for a subset of the benchmark problems. The results from a number of algorithms that do less well than the above four algorithms are not reported. These include a genetic algorithm, tabu search, simulated annealing and priority rule based methods (Dorndorf, Pesch, & Phan-Huy 2000, see this paper for a summary of results).

For $SWO(BR)$, we report the average of 5 runs for each benchmark set. Each run was capped at 10 seconds (or 1000 iterations, whichever came first) in an attempt to ensure that we used no more computation time than other algorithms.

Results are summarized in table 3. Measure $\mathbf{\Delta_{LB}}\%$ gives the average relative deviation from the known lower bounds.

Measures $\%_{\mathbf{opt}}$ and $\%_{\mathbf{feas}}$ give the percentage of feasible problems for which optimal[5] and feasible solutions, respectively, were found. Finally, $\mathbf{Cpu_{sec}}$ gives the average computation time, scaled by processor speed. This last measure should only be used for rough comparisons since the algorithms were developed in different languages and run on different platforms.

Algorithm $SWO(BR)$ was able to find solutions to all 1733 feasible problems (on all 5 runs). The only other algorithm that does not miss feasible solutions is $B\&B_{F99}$, which does so at the expense of quality, especially on set C.

Not surprisingly, exact methods get the best results for small problems. $B\&B_{S98}$ is able to solve all *J10* problems optimally, does fairly well on the *J20* problems but already fails to find a number of feasible solutions to some of the problems in *J30*. $B\&B_{D98}$ is better able to handle larger problems and is by far the best algorithm on problem set B. However, when the number of activities is increased from 100 to 500 (set C), it fails to find all feasible solutions and solves fewer to optimality than $SWO(BR)$.

$SWO(BR)$ is also very competitive with the best non-systematic algorithms. It outperforms *ISES* on all metrics and on all problem sets. While *KDE* is significantly better on measurement $\mathbf{\Delta_{LB}}\%$, $SWO(BR)$ is slightly better than *KDE* on the other three dimensions.

Finally, in addition to scaling well in terms of relative solution quality, $SWO(BR)$ appears to scale effectively in terms of running time. For example, while *ISES* takes 33 times longer on *J30* than on *J10*, $SWO(BR)$ takes less than 4 times longer.[6] Similarly, $B\&B_{D98}$ takes 3.3 times longer on C than on B; the difference for $SWO(BR)$ is under 2. It is unclear how *KDE* scales since results are only reported for problems of a single size.

Concluding Remarks

We have described a new heuristic algorithm, $SWO(BR)$, for RCPSP/max problems. It uses bulldozing and refilling to improve the performance of greedy schedule construction in SWO. On a range of benchmarks $SWO(BR)$ is competitive with systematic and non-systematic state-of-the-art techniques. Able to consistently solve all feasible problems, it scales well, both in terms of solution quality and running time, relative to the best known OR and AI algorithms.

There are a number of ways $SWO(BR)$ could be improved. We have already mentioned that some resource constraint propagation should help guide SWO. Preliminary experiments using one of the consistency tests of Dorndorf et al. (Dorndorf, Pesch, & Phan-Huy 2000) give improvements for *J10* and *J20*. However, the benefits seem to be outweighed by the extra computational costs for the larger problem sizes.

It might be effective to incorporate $SWO(BR)$ into a meta-heuristic like tabu search (Chen, Fu, & Lim 2002, they find that tabu search combines well with SWO) or simulated annealing so that priority space is explored more intelligently; the current search is relatively unstructured.

Finally, $SWO(BR)$ should be tested on more difficult problems. While it didn't struggle with the above benchmarks, we do not know how it will do on problems that are larger or more difficult.

[5]Solutions that matched the lower bound or have been proven optimal by some algorithm.

[6]Of course, such comparisons must be taken with a grain of salt; running times depend partly on the cutoff times chosen.

References

Bartusch, M.; Möhring, R. H.; and Radermacher, F. J. 1988. Scheduling project networks with resource constraints and time windows. *Annals of OR* 16:201–240.

Cesta, A.; Oddi, A.; and Smith, S. 2002. A constraint-based method for project scheduling with time windows. *Journal of Heuristics* 8(1):109–136.

Chen, P.; Fu, Z.; and Lim, A. 2002. The yard allocation problem. *AAAI-2002* 3–8.

Cicirello, V. A. 2003. *Boosting Stochastic Problem Solvers Through Online Self-Analysis of Performance*. Ph.D. Dissertation, The Robotics Institute, Carnegie Mellon Univ.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Dorndorf, U.; Pesch, E.; and Phan-Huy, T. 2000. A time-oriented branch-and-bound algorithm for resource-constrained project scheduling with generalised precedence constraints. *Manag. Sci.* 46(10):1365–1384.

Fest, A.; Möhring, R. H.; Stork, F.; and Uetz, M. 1999. Resource constrained project scheduling with time windows: A branching scheme based on dynamic release dates. Technical Report 596, TU Berlin, Germany.

Ginsberg, M. L. 1993. Dynamic backtracking. *Journal of AI Research* 1:25–46.

Joslin, D. E., and Clements, D. P. 1999. Squeaky wheel optimization. *Journal of AI Research* 10:353–373.

Laborie, P. 2003. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artif. Intel.* 143(2):151–188.

Muscettola, N. 2002. Computing the envelope for stepwise constant resource allocations. In *CP 2002*, 139–154.

Neumann, K.; Schwindt, C.; and Zimmerman, J. 2003. *Project Scheduling with Time Windows and Scarce Resources*. Germany: Springer-Verlag.

Schwindt, C. 1995. ProGen/max: A new problem generator for different resource constrained project scheduling problems with minimal and maximal time lags. Technical report WIOR-449, Universitat Karlsruhe, Germany.

Schwindt, C. 1998. A branch-and-bound algorithm for the resource-constrained project duration problem subject to temporal constraints. Technical Report WIOR-544, Universitat Karlsruhe, Germany.