Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems *

Amedeo Cesta
IP-CNR
National Research Council
Viale Marx 15
I-00137 Rome, Italy
cesta@ip.rm.cnr.it

Angelo Oddi
IP-CNR
National Research Council
Viale Marx 15
I-00137 Rome, Italy
oddi@ip.rm.cnr.it

Stephen F. Smith
The Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
sfs@cs.cmu.edu

Abstract

One challenge for research in constraint-based scheduling has been to produce scalable solution procedures under fairly general representational assumptions. Quite often, the computational burden of techniques for reasoning about more complex types of temporal and resource capacity constraints places fairly restrictive limits on the size of problems that can be effectively addressed. In this paper, we focus on developing a scalable heuristic procedure to an extended, multi-capacity resource version of the job shop scheduling problem (MCJSSP). Our starting point is a previously developed procedure for generating feasible solutions to more complex, multi-capacity scheduling problems with maximum time lags. Adapting this procedure to exploit the simpler temporal structure of MCJSSP, we are able to produce a quite efficient solution generator. However, the procedure only indirectly attends to MCJSSP's objective criterion and produces sub-optimal solutions. To provide a scalable, optimizing procedure, we propose a simple, local-search procedure called *iterative flattening*, which utilizes the core solution generator to perform an extended iterative improvement search. Despite its simplicity, experimental analysis shows the iterative improvement search to be quite effective. On a set of reference problems ranging in size from 100 to 900 activities, the iterative flattening procedure efficiently and consistently produces solutions within 10% of computed upper bounds. Overall, the concept of iterative flattening is quite general and provides an interesting new basis for designing more sophisticated local search procedures.

Introduction

Constraint-based search techniques have gained increasing attention in recent years as a basis for scheduling procedures that are capable of accommodating a wide range of constraints. In its most basic form, a constraint based scheduling model operates over some sort of constraint-network encoding of the problem at hand, and problem solving proceeds through the interleaving of three basic actions:

constraint propagation: a deduction step where consequences of the current set of scheduling decisions are inferred and inconsistent search states are pruned;

search decision commitment: a choice step where search control heuristics are applied to assign some new value to some decision variable, and move the scheduling search forward;

search decision retraction: a choice step where one or more previously made decisions are retracted, allowing problem solving to back out of and continue from an inconsistent or undesirable state.

Recent research has promoted techniques for each of these basic actions as a means for addressing scheduling problems with increasingly more complex temporal and resource capacity constraints. Work in constraint deduction (Baptiste & Le Pape 1995; Baptiste, Le Pape, & Nuijten 1997; Nuijten 1994) has achieved strong performance through specification of propagation rules that exploit the special structure and character of particular classes of resource constraints. Work in search control heuristics (Smith & Cheng 1993; Cesta, Oddi, & Smith 1998; 1999; Beck & Fox 1999), alternatively, has demonstrated the effectiveness of different forms of constraint analysis in directing the scheduling process toward good solutions. Progress with various local and iterative sampling search frameworks (Zweben *et al.* 1994; Harvey & Ginsberg 1995; Bresina 1996; Crawford 1996; Oddi & Smith 1997; Cesta, Oddi, & Smith 1999), while not directly emphasizing treatment of more complex constraints, has provided several more effective alternatives to simple, backtracking-search retraction schemes.

One challenge in extending constraint-based scheduling models to accommodate more complex constraints is scalability. Quite often, techniques for reasoning about more complex types of temporal and resource capacity constraints come at a computational cost that places fairly restrictive limits on the size of problems that can be effectively addressed. Edge finding techniques, for example, provide a very powerful basis for resource constraint propagation, but are quite computationally heavy and are often impractical in large-scale settings (Baptiste & Le Pape 1995). Likewise, the texture-based heuristics defined in (Beck & Fox 1999) for scheduling with alternative resources are effective on small problems. But they rely on an explicit representa-

* Cesta and Oddi's work has been supported by ASI (Italian Space Agency) under contract ASI-ARS-99-96 and by Italian National Research Council. Smith's work has been sponsored in part by the US Department of Defense Advanced Research Projects Agency under contract F30602-97-20227, and by the CMU Robotics Institute.

tion of the entire space of choices, which quickly becomes intractable as larger problems are considered.

In this paper, we focus on developing a scalable heuristic procedure to an extended, multi-capacity resource version of the job shop scheduling problem (MCJSSP) first proposed in (Nuijten 1994). Following from recent successes with non-systematic, local search techniques, we de-emphasize the use of sophisticated constraint propagation and analysis techniques in the generation of any given schedule. Instead, our design approach is to define a simple (and efficient) greedy procedure for generating feasible solutions, and then specify a decision retraction scheme that will allow its repeated use within a larger iterative improvement search framework.

Our starting point is the work of (Cesta, Oddi, & Smith 1998), which developed a procedure for generating feasible solutions to a more complex class of multi-capacity scheduling problem with maximum time lags. By customizing this procedure to exploit the simpler temporal structure of MCJSSP, we are able to produce a quite efficient (though less general) solution generator. To address MCJSSP's objective criteria of minimizing overall schedule makespan and define a scalable, optimizing procedure, we introduce a novel local-search framework called *iterative flattening*. The core solution generation procedure, which is designed to post precedence constraints between activities to reduce (or flatten) resource contention peaks, is first applied to produce an initial feasible solution. New contention peaks are then iteratively created and flattened by (1) retracting some subset of previously posted constraints along the current solution's critical path and (2) restarting the solution generator from this partially developed solution. Any time a lower makespan solution is produced during this iterative process, it becomes the new current solution.

The remainder of the paper is organized as follows. First, we define MCJSSP and a set of reference problems for experimental evaluation. We then describe and evaluate our core greedy search procedure for generating feasible solutions to MCJSSP instances. Next, we introduce our iterative flattening framework for extended local search using the core solution generator previously defined. Experimental results are given that demonstrate the efficacy of the approach across a range of problems of increasing scale. Finally, we briefly discuss opportunities for extending and enhancing the basic iterative flattening search concept.

MCJSSP and its Benchmarks

The Scheduling Problem. The Multi-Capacity Job-Shop Scheduling Problem (MCJSSP) involves synchronizing the use of a set of resources $R = \{r_1 \ldots r_m\}$ to perform a set of jobs $J = \{j_1 \ldots j_n\}$ over time. The processing of a job j_i requires the execution of a sequence of m activities $\{a_{i_1} \ldots a_{i_m}\}$, each a_{ij} has a constant processing time p_{ij} and requires the use of a single unit of resource $r_{a_{ij}}$ for its entire duration. Each resource r_j is required only once in a job and can process at most c_j activities at the same time ($c_j \geq 1$). A *feasible solution* to a MCJSSP is any temporally consistent assignment to the activities' start times which does not violate resource capacity constraints. An *optimal solution* is a feasible solution with minimal overall duration or makespan. Generally speaking, MCJSSP has the same structure as JSSP but involves multi-capacitated resources instead of unit-capacity resources.

Benchmarks. In (Nuijten & Aarts 1996) a method for creating challenging problems is proposed that starts from the JSSP benchmarks of (Lawrence 1984). The idea is to take each original JSSP problem, double (or triple) the capacity of each resource, and then duplicate (or triplicate) the activities of each job. Since all activities continue to require 1 unit of resource capacity, the result is a similarly structured MCJSSP.

We have used this procedure to obtain 80 MCJSSPs from Lawrence's 40 original problems: 40 problems with resources of capacity 2 and 40 with resources of capacity 3. The generated problems range in size from 100 to 900 activities. To organize the presentation of results we subdivide the 80 problems in 4 sets of 20 each:

Set A: LA1-10 x2 x3 (Lawrence's problems numbered 1 to 10, duplicated and triplicated). Using the notation #jobs $\times$ #resources (resource capacity), this set consists of 5 problems each of sizes 20x5(2), 30x5(3), 30x5(2), 45x5(3).

Set B: LA11-20 x2 x3. 5 problems each of sizes 40x5(2), 60x5(3), 20x10(2), 30x10(3).

SetC: LA21-30 x2 x3. 5 problems each of sizes 30x10(2), 45x10(3), 40x10(2), 60x10(3).

Set D: LA31-40 x2 x3. 5 problems each of sizes 60x10(2), 90x10(3), 30x15(2), 45x15(3).

Why have we chosen this benchmarks? From one side because in relatively few instances they cover a wide range of problem sizes; from the other because they also provide a direct basis for comparative evaluation. In fact, as noted in (Nuijten & Aarts 1996), one consequence of the problem generation method is that the optimal makespan for the original JSSP is also a tight upper bound for the corresponding MCJSSP. Hence, distance from these upper-bound solutions can provide one useful measure of solution quality.

The problems in set A and the smallest, 20x10 problems in set B have also been solved by (Nuijten & Aarts 1996), providing a further basis for calibrating and evaluating performance on the smaller problems. Nuijten's approach relies on a quite sophisticated set of resource propagation rules including edge-finding, and is well known for its strong performance on this problem subset. The goal of our current investigation is not to perform a competitive comparison with this work per se, but to instead develop a MCJSSP procedure that avoids the computational cost of sophisticated constraint analysis techniques and effectively scales to larger-scale problems such as those in Sets C and D.[1]

[1]For example, Nuijten's approach incorporates a family of propagation rules of different computational complexity. The most effective rule of his set is cubic in the number of activities on a resource, which becomes increasingly problematic as the number of activities increases. The decision step of the algorithm we present

A Greedy Algorithm for MCJSSP

As indicated earlier, our approach to designing a scalable heuristic procedure for solving MCJSSPs follows an iterative improvement schema. Our procedure is composed of two basic steps: (a) a greedy search algorithm is first applied to produce an initial feasible solution; (b) a local-search algorithm is then used to iteratively improve the quality of the current solution until a termination condition is met. In this section we first consider the core procedure for generating feasible solutions. In the next, we turn attention to the extended local search process.

Our greedy search algorithm is inspired by prior work on the Earliest Start Time Algorithm (ESTA), proposed originally in (Cesta, Oddi, & Smith 1998) and further refined in (Cesta, Oddi, & Smith 1999). ESTA was designed to address more general, multi-capacity scheduling problems with generalized precedence relations between activities (i.e., corresponding to metric separation constraints with minimum and maximum time lags). We briefly summarize the basic ideas that have been taken from this work, and then describe the modifications made to better address the MCJSSP domain.

Previous Profile-Based Work. ESTA is a variant of a class of profile-based scheduling procedures, characterized by a two-phase, solution generation process:

Construct an infinite capacity solution:
A constraint based representation of the current problem is formulated as an STP (Dechter, Meiri, & Pearl 1991) temporal constraint network.[2] In this initial representation temporal constraints are modeled and satisfied (via constraint propagation) but resource constraints are ignored, yielding a time feasible solution that assumes infinite resource capacity.

Level resource demand by posting precedence:
Resource constraints are super-imposed by projecting "resource demand profiles" over time. Detected resource conflicts are then resolved by iteratively posting simple precedence constraints between pairs of competing activities.

To perform the process of constraint posting ESTA follows a four step cycle:

(a) An ESS (for Earliest Start Solution) consistent with currently imposed temporal constraints is computed. This can be done quickly since the earliest start values of all nodes in any STP network are known to constitute a temporally feasible solution.

(b) Given the ESS, a *contention peak* is recognized on resource r_k at time t if condition $req_k(ESS, t) > c_k$ holds (with req_k being the sum of requirements of resource r_k at time t). Intuitively, a contention peak on resource r_k identifies a set of activities that simultaneously require r_k with a combined capacity requirement $> c_k$.

(c) For each peak, Minimal Critical Sets (MCSs) are computed. A *Minimal Critical Set* (MCS) specifies *a set of activities that simultaneously require a resource r_k with a combined capacity requirement $> c_k$, such that the combined requirement of any subset is $\leq c_k$*. The important advantage of isolating MCSs is that a single precedence relation between any pair of activities in the MCS eliminates the conflict. Since complete enumeration of MCSs is a combinatorial problem, a *sampling strategy* of fixed computational complexity (e.g., linear, quadratic in the number of activities) is proposed in (Cesta, Oddi, & Smith 1999) to collect some subset of MCSs in each peak.

(d) A single MCS is selected and resolved by posting a precedence constraint between two of the constituent activities. MCS selection (variable ordering) is performed according to the K estimator proposed in (Laborie & Ghallab 1995). The specific constraint to be posted (value ordering) is determined so as to preserve maximal temporal slack (in a style similar to that proposed in (Smith & Cheng 1993)).

Previous research has shown ESTA to be effective in overcoming efficiency problems that have plagued other profile-based scheduling procedures, while tending to better minimize the number of precedence constraints posted. As such, it seems a good starting point for building a heuristic approach to MCJSSP.

Adapting ESTA to MCJSSP. Though the basic ESTA procedure just described is directly applicable to MCJSSP, the simpler temporal structure of this problem domain suggests two adaptations in the interest of obtaining a computationally lighter procedure for initial solution generation.

A first change concerns the choice of how to sample MCSs. Previous point (c) performs MCS sampling on each peak detected in the current solution. Simplifying, we not only pay attention to non-redundancy in peak computation but also restrict MCS sampling to only the "maximal peaks", i.e., those peaks that contain the maximum number of activities. This choice is motivated by the observation that in the absence of maximal time lags the criticality of a given resource conflict is more clearly a function of its size.

A second change involves the propagation algorithms used to maintain consistency of the problem's temporal information. The solution of scheduling problems with generalized precedence relations require computation of the transitive closure of the STP representation of the current solution (Dechter, Meiri, & Pearl 1991). Computation of this information is fundamental, for example, for early detection and pruning of temporally infeasible search states. It is well known that such information is computable via all pairs shortest path algorithms, which are quadratic in space and cubic in time with regard to the number of temporal variables in the network. Unfortunately, as problem size increases, this computation tends to increasingly dominate overall solution time.

below is also cubic in the worst case. But within the local search framework we propose, this algorithm is executed from scratch only once, as opposed to repeatedly in Nuijten's case.

[2] In a STP (Simple Temporal Problem) network: temporal variables (nodes or time-points) represent beginning and end of activities and beginning and end of temporal horizon; distance constraints (edges) represent duration of activities and separation constraints including simple precedences.

ESTA$_M$(Problem)
1. TCSP ← CreateCSP(Problem)
2. **loop**
3. Propagate(TCSP)
4. ConflictSet ← ComputeResourceConflicts(TCSP)
5. **if** Empty(ConflictSet)
6. **then return**(ExtractSolution(TCSP))
7. **else**
8. **if** Unsolvable(ConflictSet)
9. **then return**(EmptySolution)
10. **else**
11. Conflict ← SelectConflict(ConflictSet)
12. PrecedenceConstraint ← SelectPrecedence(Conflict)
13. PostCostraint(TCSP, PrecedenceConstraint)
14. **end-loop**
15. **end**

Figure 1: Basic ESTA$_M$ Search Procedure

A second observation concerning MCJSSP helps in this case: in any MCJSSP all separation constraints between activities are simple precedence, and the only metric temporal constraints present are activity durations. This being the case, it is possible to detect infeasible orderings between pairs of activities using simpler, single source shortest path algorithms. In fact, temporal consistency can be maintained by a single source shortest path algorithm whose complexity instead depends on the number of edges (or temporal constraints) in the network. Because the STP network representing a scheduling problem is typically sparse (few edges, number of edges of the same order as number of nodes) this leads to a real advantage in terms of lightening the algorithm on problems of significant size.

One side-effect of this shift to a more efficient (but less informative) propagation algorithm is a simplification of the the K estimator computation utilized by the basic search algorithm, In particular, K is instead computed in terms of simple calculations of temporal slack (i.e., given a pair of activities a_i a_j, $slack(a_i, a_j)$ is defined as the difference between a_j's latest start time the and a_i's earliest finish time).

We refer to the procedure resulting from incorporation of the above changes as ESTA$_M$, where the suffix M indicates the modified version. A schematic view of ESTA$_M$ is given in Figure 1. In this formulation we have hidden details of the previous points (a)-(c) in Step 4, where `ComputeResourceConflicts` performs both peak detection on the ESS, and MCS sampling on the maximal peak.

Experimental Evaluation

In Table 1 we show the performance of ESTA$_M$ on the four benchmark problem sets. The algorithm is implemented in Allegro Common Lisp and the reported results are obtained on a SUN UltraSparc 30 (266MHz). For each algorithm and for each set we report $\Delta_{UB}\%$ (upper row), the average relative deviation from the known upper bound, and CPU_{sec} (lower row), the average CPU time in seconds. The column "All" gives the average values over all 80 problems. We also include the performance results obtained in (Nuijten & Aarts 1996) (the rows labelled CCA), which unfortunately are available only for set A and a few instances of set B.[3] But we note again that the upper-bound reference also provides a very good comparison value.

Observing the Table, two comments are appropriate: (a) it is not surprising that, compared with an algorithm like CCA aimed at finding a solution with an optimal makespan, a single run of ESTA$_M$ is generally not able to find an optimal solution. ESTA$_M$ does not contain any attempt at optimizing, but is designed to simply search for a feasible solution. The fact that it attempts to minimize the number of precedence constraints posted will at best contribute only indirectly to minimizing makespan; (b) Generally, ESTA$_M$ finds a solution quite efficiently, with resulting makespan that varies from 15 to 25% of the upper bound. In fact, the 758 seconds required on average for set D are due mostly to the 5 largest, 900 activity, problems which on average took over 2450 seconds to solve. These problems are characterized by huge peaks that require large numbers of precedence to be leveled. This is the only subset in which the time spent is really relevant.

Table 1: ESTA$_M$ vs. CCA

Algorithm	Set A	Set B	Set C	Set D	All
ESTA$_M$	17.91	16.04	25.27	24.83	21
	45	125	190	758	279
CCA	1.57	*	–	–	–
	369	*	–	–	–

Improving ESTA$_M$ by Local Search

Given an efficient procedure for generating feasible solutions to MCJSSP, the important remaining design issue concerns how to exploit it to perform an extended optimizing search. In both (Nuijten & Aarts 1996; Cesta, Oddi, & Smith 1999), an iterative sampling framework is used to provide a basis for optimization. However, in the interest of scalability we choose instead to emphasize a local search approach. The intuition is simply that computation of neighborhood solutions is likely to be more cost effective than solution regeneration. But this requires an effective approach to generating neighborhood solutions.

To describe our approach, we first introduce a few definitions. Assume that a given solution Sol to a MCJSSP produced by ESTA$_M$ is represented as a directed graph $G_S(A, E)$. A is the set of activities specified in MCJSSP, plus a fictitious a_{source} activity temporally constrained to occur before all others and a fictitious a_{sink} activity temporally constrained to occur after all others. E is the set of precedence constraints imposed between activities in A. E can be partitioned in two subsets, $E = E_{prob} \cup E_{post}$,

[3] Only the smallest 5 20x10 problems were solved (not enough for comparison on the whole set). The performance amounts to an average Δ of -0.84% and an average CPU of 591 seconds (i.e., very good solutions but with substantially increasing CPU times).

where E_{prob} is the set of precedence constraints originating in the problem definition, and E_{post} is the set of precedence constraints posted by ESTA$_M$ to resolve various resource conflicts. A *path* in $G_S(A, E)$ is a sequence of activities $a_1 \ldots a_k$, such that, $(a_i, a_{i+1}) \in E$ with $i = 1 \ldots (k-1)$. The length of a path is the sum of the activities' processing times and a *critical path* is a path from a_{source} to a_{sink} which determines the solution's makespan.

As is well recognized in the scheduling literature, information about *critical paths* can provide a strong heuristic basis for makespan minimization. In the case of solutions generated by ESTA$_M$, any improvement in makespan will necessarily require retraction of some subset of precedence constraints situated on the *critical path*, since these constraints collectively determine the solution's current makespan. Following this observation, we propose a simple, two-step method for generating *moves* in the neighborhood of a given solution:

Shrinking Step: We first randomly retract a subset of precedence constraints $pc_i \in E_{post}$ which fall on the solution's *critical path*. In this way, the Earliest Start Solution is compressed and new peaks appear in the resource profiles.

Flattening Step: We then re-apply the ESTA$_M$ algorithm to level (or flatten) the newly introduced resource conflicts by posting new precedence constraints.

Because of the character of this local search cycle, we call the algorithm *iterative flattening* (i-FLAT).

From a performance standpoint, note that ESTA$_M$ is not applied from scratch (as is done on the initial problem formulation), but in an incremental way to a partially generated solution. In fact, the removal of a subset of precedence constraints generally creates both fewer and smaller peaks than the set of peaks contained in an initial infinite capacity solution. Additional efficiency gains are obtained through use of temporal reasoning techniques that support incremental retraction of precedence constraints $pc_i \in E_{post}$ (Cesta & Oddi 1996).

Figure 2 shows the *iterative flattening* algorithm in more detail. It takes as input three elements: (1) a starting solution Sol; (2) an integer number P$_{rem} \in 1..100$ designating the percentage of precedence constraints $pc_i \in E_{post}$ on the critical path to be removed at each execution of the basic move; and (3) a positive integer $MaxFail$ which specifies the maximum number of moves without an improvement in makespan that the algorithm will tolerate before terminating.

After initialization (Steps 1-2), within the `While` loop at Step 3, a solution is repeatedly modified by the application of the following subprocedures: `SamplingCriticalPath` is first applied to randomly select a set of previously posted precedence constraints on the solution critical path; `RemovePrecedence` then removes this set of constraints through application of incremental temporal reasoning algorithms; ESTA$_M$ is next invoked to find a new earliest start time solution. In the case that a better makespan solution is found (at Step 7), the new solution is stored in S$_{best}$ and the counter is reset to 0 (Steps 9-10). Otherwise, if no improvement is found in $MaxFail$ moves, the algorithm terminates

i-FLAT(Sol,P$_{rem}$,MaxFail)
1. S$_{best}$ $\leftarrow Sol$
2. counter $\leftarrow$ 0
3. **while** (counter $\leq$ MaxFail) **do begin**
4. PrecedenceToRemove
   $\leftarrow$ SamplingCriticalPath(Sol,P$_{rem}$)
5. RemovePrecedence(Sol,PrecedenceToRemove)
6. Sol$\leftarrow$ESTA$_M$(Sol)
7. **if** Mk(Sol) $<$ Mk(S$_{best}$)
8. **then begin**
9. S$_{best}$ $\leftarrow$ Sol
10. counter $\leftarrow$ 0
11. **end**
12. **else** counter $\leftarrow$ counter + 1
13. **end-while**
14. **return**(S$_{best}$)
15. **end**

Figure 2: The *Iterative Flattening* Algorithm

Table 2: i-FLAT Performance

Algorithm	Set A	Set B	Set C	Set D	All
ESTA$_M$	17.91	16.04	25.27	24.83	21
	45	125	190	758	279
i-FLAT(E+1)	8.99	8.29	14.61	12.22	11
	60	161	286	1199	426
i-FLAT(E+5)	7.76	7.10	13.03	11.92	9.95
	124	329	657	1875	746

and returns the best solution found.

Experimental Results

Table 2 shows the performance of i-FLAT on the benchmark problem set. For this set of experiments, P$_{rem}$ was set to 10% and $MaxFail$ to 300. The version of ESTA$_M$ used within i-FLAT is the same as the basic version. We report results for basic ESTA$_M$, for i-FLAT starting from the solution produced by ESTA$_M$ (row labeled i-FLAT(E+1)), and for an extended execution of i-FLAT (row i-FLAT(E+5)) that will be explained below. CPU times reported for the i-FLAT rows include the initial execution of ESTA$_M$, giving the total time taken by the algorithm to produce its best solution.

Two observations are immediate: (a) the time spent by ESTA$_M$ in generating an initial solution dominates the time spent improving it by the iterative i-FLAT process across all problem sets (e.g., 45 seconds versus 15 seconds in the case of Set A); (b) the improvement of ESTA$_M$'s initial solution by i-FLAT is rather significant (lowering an average deviation of 21% to 11%).

It should be noted that i-FLAT currently performs a relatively undirected search (basically a random walk) and there appear several opportunities for further empowering the local search strategy (see concluding discussion below). Here, we consider an alternative approach to improving experimental performance, by simply restarting i-FLAT multiple times from the same ESTA$_M$ solution. This approach is

justified by the random step `SamplingCriticalPath`, which guarantees different execution on different restarts. Table 2 shows the results obtained with 5 random restarts of i-FLAT (labeled i-FLAT(E+5)). In this case, performance is improved on all four problem sets and the overall average deviation is lowered below 10%.

Finally, a comment concerning comparison with Nuijten's work. The performance of the two i-FLAT configurations on Set A should be seen as quite positive in relation to the results obtained by the CCA algorithm. If we consider the relatively simple steps that i-FLAT executes in contrast to the rather sophisticated resource propagation rule that is coupled with random restarting in the CCA approach, the difference in deviation of about 6% is quite respectable. If we consider further that i-FLAT's deviation from upper-bound solutions does not worsen substantially with increasing problem scale, and that CCA scalability (particularly with use of the most effective propagation rules) is an open issue, then the results obtained with i-FLAT assume even greater significance.

Conclusions and Future Work

This paper has presented a new iterative improvement technique, called *iterative flattening* for solving large-scale multi-capacity scheduling problems.

Several aspects of this algorithm are worth underscoring. First, the algorithm provides a novel, local search model that integrates naturally with typical constraint-guided schedule generation methods and heuristics. Second, the algorithm is quite general and is applicable not only to the MCJSSP, but also to problems such as resource constrained project scheduling where activities require multiple resources and/or resource capacity of varying amounts. Finally, the algorithm has be shown to scale effectively to large-scale problem instances of the MCJSSP, creating a reference point (being within 10% of computed upper bounds) for other approaches on a rich set of benchmark problems.

The basic concept of iterative flattening presented in this paper is very general, and many possibilities of performance enhancement through incorporation of more sophisticated and better informed local-search procedures appear possible. Among the directions we are considering for future work are the following: examination of different precedence constraint retraction strategies; enrichment of the basic random search with standard concepts such as neighborhood analysis or a taboo-list; and, more generally, the insertion of iterative flattening within a *meta* local search strategy (e.g., (Nowicki & Smutnicki 1996) for the classical JSSP).

References

Baptiste, P., and Le Pape, C. 1995. A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling. In *Proceedings of the* 14^{th} *Int. Joint Conference on Artificial Intelligence.*

Baptiste, P.; Le Pape, C.; and Nuijten, W. 1997. Satifiability Tests and Time-Bound Adjustments for Cumulative Scheduling Problems. Technical report, University of Compiégnie. to appear in *Annals of Operations Research*.

Beck, J., and Fox, M. 1999. Scheduling Alternative Activities. In *Proceedings* 16^{th} *National Conference on AI (AAAI-99)*.

Bresina, J. 1996. Heuristic-biased Stochastic Sampling. In *Proceedings* 13^{th} *National Conference on AI (AAAI-96)*.

Cesta, A., and Oddi, A. 1996. Gaining Efficiency and Flexibility in the Simple Temporal Problem. In *Proceedings of the Third International Workshop on Temporal Representation and Reasoning (TIME-96)*.

Cesta, A.; Oddi, A.; and Smith, S. 1998. Profile Based Algorithms to Solve Multiple Capacitated Metric Scheduling Problems. In *Proceedings of the* 4^{th} *Int. Conf. on Artificial Intelligence Planning Systems (AIPS-98)*.

Cesta, A.; Oddi, A.; and Smith, S. 1999. An Iterative Sampling Procedure for Resource Constrained Project Scheduling with Time Windows. In *Proceedings of the* 16^{th} *Int. Joint Conference on Artificial Intelligence (IJCAI-99)*.

Crawford, J. 1996. An Approach to Resource Constrained Project Scheduling. In *Proceedings of the 1996 Artificial Intelligence and Manufacturing Research Planning Workshop*.

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

Harvey, W., and Ginsberg, M. 1995. Limited Discrepancy Search. In *Proceedings of the* 14^{th} *Int. Joint Conference on Artificial Intelligence (IJCAI-95)*.

Laborie, P., and Ghallab, M. 1995. Planning with Sharable Resource Constraints. In *Proceedings of the* 14^{th} *Int. Joint Conference on Artificial Intelligence (IJCAI-95)*.

Lawrence, S. 1984. Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement). Technical report, Graduate School of Industrial Administration, Carnegie Mellon University.

Nowicki, E., and Smutnicki, C. 1996. A Fast Taboo Search Algorithm for the Job Shop Problem. *Management Science* 42:797–813.

Nuijten, W., and Aarts, E. 1996. A Computational Study of Constraint Satisfaction for Multiple Capacitated Job Shop Scheduling. *European Journal of Operational Research* 90(2):269–284.

Nuijten, W. 1994. *Time and Resource Constrained Scheduling - A Constraint Satisfaction Approach.* Ph.D. Dissertation, Eindhoven University of Technology, The Netherlands.

Oddi, A., and Smith, S. 1997. Stochastic Procedures for Generating Feasible Schedules. In *Proceedings* 14^{th} *National Conference on AI (AAAI-97)*.

Smith, S., and Cheng, C. 1993. Slack-Based Heuristics for Constraint Satisfaction Scheduling. In *Proceedings* 11^{th} *National Conference on AI (AAAI-93)*.

Zweben, M.; Duan, B.; Davis, E.; and Deale, M. 1994. Scheduling and Rescheduling with Iterative Repair. In Zweben, M., and Fox, S. M., eds., *Intelligent Scheduling*. Morgan Kaufmann.