Boosting the Performance of Iterative Flattening Search

Paper ID 670

Abstract

Iterative Flattening search is a local search schema introduced for solving scheduling problems with a makespan minimization objective. It is an iterative two-step procedure, where on each cycle of the search a subset of ordering decisions on the critical path in the current solution are randomly retracted and then recomputed to produce a new solution. Since its introduction, other variations have been explored and shown to yield substantial performance improvement over the original formulation. In this spirit, we propose and experimentally evaluate further extensions to this basic local search schema. Specifically, we examine the utility (1) of operating with a more flexible solution representation, (2) of adopting a more focused decision retraction strategy and (3) of integrating iterative-flattening search with a complementary tabu search procedure. We evaluate these extensions on large benchmark instances of the Multi-Capacity Job-Shop Scheduling Problem (MCJSSP) which have been used in previous studies of iterative flattening search procedures.

Introduction

The integration of local search and heuristic procedures has produced interesting and efficient approaches to several complex scheduling problems. One such example is the *iterative flattening* (or I-FLAT) algorithm proposed in (Cesta, Oddi, & Smith 2000) for solving scheduling problems with a makespan minimization objective. I-FLAT consists of an iterative, two-step local search schema. Within each cycle of the search, a *relaxation step* is first applied to remove some search decisions from the current solution and create a partial solution. In this context, search decisions correspond to precedence constraints that must be added between pairs of activities to resolve resource conflicts, and the relaxation step limits its attention to those constraints residing on the solution's *critical path*. Starting from this new partial solution, the second *flattening step* then incrementally adds back new precedence constraints to regain a feasible solution. In the original work, I-FLAT was shown to produce high-quality solutions in reasonable time on a challenging set of large multi-capacitated job-shop scheduling problem MCJSSP benchmarks.

More recently two works (Michel & Van Hentenryck 2004; Godard, Laborie, & Nuitjen 2005) have extended the results of the original paper through refinement of the basic I-FLAT search schema. (Michel & Van Hentenryck 2004) identified an anomaly in I-FLAT search and proposed a simple extension, which dramatically improved the quality of its schedules while preserving its computational efficiency. Their key idea was to iterate the relaxation step multiple times, hence the name I-FLATRELAX used in what follows. The resulting algorithm found many new upper bounds and produced solutions within 1% of the best upper bounds on average. The work of (Godard, Laborie, & Nuitjen 2005) presented an approach which follows the same two step schema of I-FLAT but used different engines for both the flattening and the relaxation steps. This procedure was able to find additional optimal solutions and to further improve known upper-bounds for MCJSSP benchmarks.

In this same spirit, this paper proposes and evaluates further extensions to the I-FLAT family of search procedures. We focus specifically on three potential shortcomings of the basic approach: (1) the lack of temporal flexibility in the solutions that are manipulated by I-FLAT, (2) the tendency to re-explore the same solution subspaces and (3) the inability to perform a finer-grained neighborhood search in the vicinity of near-optimal solutions. To cope with these issues, we explore the use of partial order schedules (Policella *et al.* 2004) as an underlying solution representation, add a *tabu-list* to better control the relaxation step, and introduce a *tabu-seach* procedure to refine solutions found by I-FLAT.

The paper is organized as follows. We first review the iterative flattening search schema that has evolved from previous work. The central part of the paper then introduces each of the proposed extensions, and indicates how they are integrated (individually and in combination) within the basic search schema. Next we define the MCJSSP problem domain and benchmark problem sets used in our evaluation. Performance results are then given that demonstrate the leverage provided by the extended search procedure. We conclude by briefly discussing further opportunities to extend and enhance the basic iterative flattening search concept.

The Iterative Flattening Schema

Before describing the iterative flattening approach it is necessary to introduce the modeling perspective on which this schema is based. Underlying the approach, a solution Sol 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 defined between activities in A. Following a Precedence Constraint Posting (PCP) approach, the set E can be partitioned in two subsets, $E = E_{prob} \cup E_{post}$, where E_{prob} is the set of precedence constraints originating from the problem definition, and E_{post} is the set of precedence constraints posted to resolve resource conflicts. In general the directed graph $G_S(A, E)$ represents a set of temporal solutions. The set E_{post} is added in order to guarantee that at least one of those temporal solutions is also resource feasible.

Given this description of the solution model, lets turn attention to the Iterative Flattening search procedure itself (Cesta, Oddi, & Smith 2000). This heuristic procedure iterates two main steps:

Relaxation step: first, it relaxes a feasible schedule into a possibly resource infeasible, but precedence feasible, schedule by removing some search decisions represented as precedence constraints between pair of activities;

Flattening step: second, it posts a sufficient set of new precedence constraints to re-establish a feasible schedule.

These two steps are executed for some number of iterations or until no better feasible schedule has been found for some number of iterations.

Figure 1 shows the *iterative flattening* algorithm in detail. I-FLAT takes as input four elements: (1) a starting solution Sol; (2) a value $P_{rem} \in [0, 1]$ designating the percentage of precedence constraints $pc_i \in E_{post}$ on the critical path to be removed; (3) a positive integer $MaxFail$ which specifies the maximum number of non-makespan-improving moves that the algorithm will tolerate before terminating; and (4) a positive integer $MaxRelaxations$ which specifies the maximum number of relax iterations to be performed in the relaxation step. Note that it is the value of this last parameter that distinquishes between the (Cesta, Oddi, & Smith 2000) and the (Michel & Van Hentenryck 2004) approach (see below). Returning to the algorithm description in Figure 1, after initialization (Steps 1-2), within the while loop (Steps 3-10) a solution is repeatedly modified by the application of the RELAX and FLATTEN procedures. In the case that a better makespan solution is found (at Step 6), the new solution is stored in S_{best} and the counter is reset to 0. Otherwise, if no improvement is found in $MaxFail$ moves, the algorithm terminates and returns the best solution found. The rest of this section describes the relaxation and the flattening steps in more detail.

Relaxation. The relaxation step is based on the concept of *critical path*. As known in the scheduling literature, information about *critical paths* can provide a strong heuristic basis for makespan minimization. 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.

Any improvement in makespan will necessarily require change to some subset of precedence constraints situated on the *critical path*, since these constraints collectively determine the solution's current makespan. Following this observation, the relaxation step introduced in (Cesta, Oddi, & Smith 2000) is designed to retract some number of posted precedence constraints in the solution's critical path.

I-FLAT($Sol, P_{rem}, MaxFail, MaxRelaxations$)
begin
1. $S_{best} \leftarrow Sol$
2. $counter \leftarrow 0$
3. **while** ($counter \leq MaxFail$) **do**
4. RELAX($Sol, P_{rem}, MaxRelaxations$)
5. $Sol \leftarrow$FLATTEN(Sol)
6. **if** Mk(Sol) < Mk(S_{best}) **then**
7. $S_{best} \leftarrow Sol$
8. counter $\leftarrow$ 0
9. **else**
10. counter $\leftarrow$ counter + 1
11. **return** S_{best}

end

Figure 1: The *Iterative Flattening* Algorithm

Figure 2 shows the RELAX procedure. Steps 2-4 consider the set of posted precedence constraints ($pc_i \in E_{post}$) which belong to the current critical path. A subset of these constraints is randomly selected and then removed from the current solution. Step 1 represents the crucial difference between the (Cesta, Oddi, & Smith 2000) and the (Michel & Van Hentenryck 2004) approaches. In the former approach the steps 2-4 are repeated only once (i.e. $MaxRelaxations = 1$) whereas in (Michel & Van Hentenryck 2004) these steps are iterated several times (from 2 to 6). In practice the new critical path of Sol is computed at each iteration. Notice that this path can be completely different from the previous one. This allows the relaxation step to also take into account those paths that have a criticality very close to the most one.

RELAX($Sol, P_{rem}, MaxRelaxations$)
begin
1. **for** 1 **to** $MaxRelaxations$
2. **forall** $(a_i, a_j) \in$ CriticalPath(Sol)$\cap E_{post}$
3. **if** random(0,1)< P_{rem}
4. $Sol \leftarrow Sol \backslash (a_i, a_j)$

end

Figure 2: The RELAX procedure

Flattening. The flattening step used in (Cesta, Oddi, & Smith 2000) is inspired by prior work on the Earliest Start Time Algorithm (ESTA) from (Cesta, Oddi, & Smith 1998). 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). This algorithm is a variant of a class of PCP scheduling procedures, characterized by a two-phase, solution generation process:

Constructing an infinite capacity solution: the current problem is formulated as an STP (Dechter, Meiri, & Pearl 1991) temporal constraint network.[1] In this initial problem representation, temporal constraints are modeled

[1] 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.

```
FLATTEN(Problem)
begin
1.  TCSP ← CreateCSP(Problem)
2.  loop
3.      Propagate(TCSP)
4.      ConflictSet ← ComputeResourceConflicts(TCSP)
5.      if ConflictSet= ∅ then
6.          return ESS(TCSP)
7.      else
8.          if Unsolvable(ConflictSet) then
9.              return ∅
10.         else
11.             Conflict ← SelectConflict(ConflictSet)
12.             pc_i ← SelectPrecedence(Conflict)
13.             TCSP ← TCSP∪{pc_i}
end
```

Figure 3: The FLATTEN implementation

and satisfied (via constraint propagation) but resource constraints are ignored, yielding a time feasible solution that assumes infinite resource capacity.

Leveling 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.

The constraint posting process of ESTA is based on the Earliest Start Solution (ESS) consistent with currently imposed temporal constraints (computed in step 3). It then proceeds to compute the set of resource conflicts (step 4). If this set is empty the ESS is also resource feasible and a solution is found; otherwise this set of conflicts is considered. If a conflict exists that can be solved, a new precedence constraint is posted to do so (step 11-13); otherwise the process fails and returns the empty set (step 9). For further details on the functions ComputeResourceConflict(), SelectConflict(), and SelectPrecedence() the reader should refer to the original references.

Extensions

The concept of Iterative Flattening is quite general and provides an interesting new basis for designing more sophisticated and effective local search procedures for scheduling optimization. The procedure proposed in (Michel & Van Hentenryck 2004) is a nice example of an I-FLAT extension which obtains substantial improvements over its original version. This section describes a further analysis of the I-FLAT schema based on the three possible *drawbacks* identified in the introduction.

A first potential shortcoming is the lack of temporal flexibility in the initial solution provided to seed I-FLAT. Previous work has used ESTA as an initial solution generator and, as indicated earlier, ESTA only guarantees that the earliest start time solution (ESS) is resource feasible. In fact, the effectiveness of the I-FLAT procedure relies on finding new orderings of activities such that an increasingly more compact solution is found on each cycle. The greater the flexibility, the higher the probability that new start times for relaxed activities can be found on a given *relax-and-flatten* cycle that reduce the overall makespan.

A second possible drawback stems from the simple manner in which precedence constraints are selected for retraction, which can lead to repeated selection of the same constraints and repeated solving of the same (or very similar) partial solution.

A third possible drawback is the lack of an ability to conduct a *fine-grained* search when a near-optimal solution is generated by I-FLAT. In fact, due to its random behavior, the procedure is unlikely to be able to explore *close* neighbors of a near-optimal solution, in order to further improve it.

In the following subsections we propose extensions to address each of these potential limitations.

Introducing Partial Order Schedules

The first extension of the search schema, aimed at increasing the temporal flexibility of solutions generated during the flattening step, is to substitute the use of partial order schedules (POS) for the flexible solutions produced by ESTA. Both types of the solutions are based on a graph representation. The difference is that while flexible solutions guarantee that at least one of the temporal solutions they represent is also resource feasible, a POS guarantees that *all* delineated temporal solutions are also resource feasible. The use of a POS in general increases the possibilities for rearranging relaxed activities. Notice that in (Godard, Laborie, & Nuitjen 2005) POSs are instead used to overcome the natural lack of flexibility of fixed time solutions.

The common thread underlying a POS is the characteristic that activities which require the same resource units are linked via precedence constraints into precedence chains. Given this structure, each constraint becomes more than just a simple precedence. It also represents a producer-consumer relation, allowing each activity to know the precise set of predecessors that will supply the units of resource it requires for execution. In this way, the resulting network of chains can be interpreted as a flow of resource units through the schedule; each time an activity terminates its execution, it passes its resource unit(s) on to its successors. It is clear that this representation is flexible if and only if there is temporal slack that allows chained activities to move "back and forth".

Methods for producing partial order schedules from a fixed time solution have been introduced in (Cesta, Oddi, & Smith 1998; Policella *et al.* 2004). Given a earliest start solution, a transformation method, named *chaining*, is defined that proceeds to create sets of chains of activities. This operation is accomplished by deleting all previously posted leveling constraints and using the resource profiles of the earliest start solution to post a new set of constraints. The first step is to consider a resource r_k with capacity c_k as a set R_k of c_k single capacity sub-resources. In this light the second step is to ensure that each activity is allocated to the same subset of R_k. This step maintains the same subset of sub-resources for each activity over time. The third step then is to build a chain for each sub resource in R_k. Thus our first extension is to introduce this chaining procedure into the initial solution generation procedure.

Using a Tabu List

The I-FLAT search procedure can be viewed as a kind of *random walk* in the space of feasible solutions which is driven

by a simple and effective heuristic criterion: only precedence constraints in the solution's critical path are selected for random removal, as they represent the most probable candidate decisions which can be changed for improving the solution's makespan. An unfortunate side-effect though is that the same constraints can be selected repeatedly.

As a second extension, we propose the use of a *tabu-list* mechanism to dampen the probability of quickly turning back to previously explored solutions. The list has a length l, such that, at each flattening cycle, a subset of posted precedence constraints are inserted in the tabu list. We use two additional search parameters to modulate the use of the tabu-list: the number of precedence constraints inserted in the tabu list at each flattening cycle δl and the tabu *tenure* t, that is, the number of cycles that a precedence constraint remains tabu and cannot be removed from the current solution.

A Tabu Search Procedure for Fine-Grained Exploration

To complement I-FLAT's randomized search behavior, our third extension involves the introduction of a tabu search algorithm, to enable fine-grained exploration in the neighborhood of near-optimal solutions discovered by I-FLAT. The tabu search procedure we have developed can be seen as an extension of the algorithm first proposed in (Nowicki & Smutnicki 1996) for Job Shop Scheduling Problems (JSSP). As is the case for the core I-FLAT algorithm, our tabu search procedure is designed to operate on the directed graph $G(A, E)$ representation for a scheduling solution, with analogous definitions of *path*, *length* of a path and *critical path*. More specifically, the search is conducted on partial order schedules, enabling the use of this earlier described extension to be coupled.

As is well-known, tabu search algorithms use the notion of *move* - a function which transforms one solution into another. For any solution S, a subset of moves m applied to S is defined. This subset of moves induces a subset of solutions called the *neighborhood* of S. Tabu search starts from an initial solution S_0, and at each step i the neighborhood of a given solution is searched for the neighbor S_i that has the best value of a fixed objective function (in our case, the objective function is the solution's makespan). The move leading to the best neighbor is performed, and a new neighborhood is calculated and searched to find a new best neighbor S_{i+1}. In order to prevent cycling in the sequence of explored solutions, it is forbidden at step i to execute one of the previous $MaxSt$ executed moves. Generally this is realized by a queue with bounded length to $MaxSt$ called the *tabu-list*, which contains the last $MaxSt$ (forbidden) moves.

Our basic *tabu search* algorithm interleaves two types of moves in searching for a minimum makespan solution: *vertical moves* and *horizontal moves*. A *vertical move* on a resource r_i is defined as the movement of an activity a_k on a *critical path* from one $chain_i$ to another $chain_j$. We use a heuristic criterion to dermine where in $chain_j$ to insert a_k. For each pair of consecutive activities (a_i, a_{i+1}) in $chain_j$, we compute a penalty function which estimates the increase in the solution's makespan if a_k is inserted between a_i and a_{i+1}, and the insertion point with the minimum penalty is chosen. Alternatively, a *horizontal move* on a resource r_i is defined as the swap of the execution order of a pair of consecutive activities (a_i, a_j) belonging to the same chain and *critical path*. Our implementation of horizontal moves directly exploits the results introduced in (Nowicki & Smutnicki 1996) concerning neighborhood definition in the case of unit capacity problems. The basic algorithm takes as input the initial type of move from which to start (*init-move*), the *tabu-list-length* and two integer parameters: max_{intrlv} and max_{iter}. These parameters respectively designate the maximum number of interleaving steps between moves of each type without makespan improvements and the maximum number of move steps of a given move type without further makespan improvement. In practice, the algorithm alternates the use of the two types of moves and each time it changes type, restarts from the best solution found with the previous type of move.

Interleaving Iterative Flattening and Tabu Search

In the next section we evaluate several extensions of the I-FLAT algorithm obtained by composing the three modifications proposed above. In addition, we consider two approaches to integrating I-FLAT with our tabu-search procedure. A first approach, called *serial* integration, simply executes the I-FLAT and tabu-search algorithms in sequence, submiting the best solution found in the first step to a tabu-search session for further improvement. A second approach, called *loop* integration, iteratively applies the following two steps a until a termination condition is met:

1. the I-FLAT procedure takes as input a solution S (the initial solution or the output of the tabu search): with probability p it returns the last solution found in the search process; with probability $(1 - p)$ the best solution found.
2. the output of the previous step becomes the input of the tabu search procedure: with probability p it returns the last solution found in the search process; with probability $(1 - p)$ the best solution found.

The value of probability p works as *noise* mechanism, to avoid the circumstance where a near optimal solution *circulates* in the loop without any improvement. In fact, it is possible that a solution S cannot be improved either by the I-FLAT algorithm or by the tabu search algorithm. In this case, we propose to restart one of the two component procedures from the last solution found by the previous step in the loop.

The MCJSSP Scheduling Problem and Test Sets

As in previous research we use the Multi-Capacity Job-Shop Scheduling Problem (MCJSSP) as a basis for evaluating the performance of our extended search procedures. This problem involves synchronizing the use of a set of resources $R = \{r_1 \dots r_m\}$ to perform a set of jobs $J = \{j_1 \dots j_n\}$ over time. The processing of a job j_i requires the execution of a sequence of m activities $\{a_{i_1} \dots a_{i_m}\}$, 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. For our analysis, we use the benchmarks introduced in (Nuijten & Aarts 1996). They consist of four sets of problems which are derived from the Lawrence job-shop scheduling problems (Lawrence 1984) by increasing the number of activities and the capacity of the resources. In particular we distinguish:

Set A: LA1-10 x2 x3 (Lawrence's problems numbered 1 to 10, with resource capacity 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).

Set C: 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).

We observe that the proposed benchmark set still represents a challenging and an effective benchmark for comparing algorithms. In fact, (a) in relatively few instances they cover a wide range of problem sizes; (b) 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 (Lawrence upper bounds). Hence, even if for many instances there are known better solutions, distance from these upper-bound solutions can provide a useful measure of solution quality.

Experimental Results

The experimental analysis has been performed in two phases. A first, explorative, phase evaluates the effect of the set of the *component* modifications described in the previous sections. In this phase we work only with the Set C benchmark. This set is a representative sub-set of instances ranging from 300 to 600 activities. A second phase then compares the best performing configuration of extensions from the first phase with current best MCJSSP benchmark results in a more CPU intensive test. All algorithms are implemented in Allegro Common Lisp., and all experiments were run on a P4 processor 1.8 GHz under Windows XP.

Comparing I-FLAT extensions. We have defined a progression of composed strategies with respect to the previous best I-FLATRELAX from (Michel & Van Hentenryck 2004). In particular they are defined as follows:

1. I-FLATRELAX-pos: this variant simply augments I-FLATRELAX with a POS representation of the input solution.
2. I-FLATRELAX-pos-tl: in this variant, the previous algorithm is integrated with the tabu list mechanism;
3. I-FLATRELAX-pos-ts-serial: in this configuration, the best solution found by I-FLATRELAX-pos is serialized with the tabu search algorithm;

Algorithm	$\Delta LWU_{\%}$	$\Delta iFlat_{\%}$
I-FLATRELAX	4.01	0.0
I-FLATRELAX-pos	3.61	-0.37
I-FLATRELAX-pos-tl	2.93	-1.03
I-FLATRELAX-pos-ts-serial	2.99	-0.97
I-FLATRELAX-pos-tl-ts-loop	2.80	-1.16

Table 1: Comparative performance on Set C

4. I-FLATRELAX-pos-tl-ts-loop: this variant interleaves I-FLATRELAX-pos-tl with tabu search in loop integration mode.

In running the phase 1 experiments, the following settings were used for I-FLATRELAX: $P_{rem} = 0.2$, $MaxFail = 400$, $MaxRelaxations = 6$. When a tabu list was used, we adopted the following parameters: length $l = 16$, $\delta l = 16$ and tenure value $t = 1$. The Tabu Search parameters were set as follows: *tabu-list*'s length $MaxSt = 9$, *init-move*= *'vertical'*, $max_{intrlv} = 1$ and $max_{iter} = 50$. Finally, the noise value for the strategy I-FLATRELAX-pos-tl-ts-loop was set to $p = 0.2$.

We also imposed a timeout of 1000 seconds for each instance of the Set C and for each composite strategy. It is worth noting how we have implemented the previous four strategies in order to met the imposed CPU bounds. In the case of the strategy I-FLATRELAX-pos-tl-ts-loop, the procedure executes the loop until the time bound is reached. For the other three strategies, we adopt the same restarting schema used in previous works (Cesta, Oddi, & Smith 2000; Michel & Van Hentenryck 2004). In the case a first run finishes before the imposed time limit, the random procedure restarts from the initial solution until the time bound is reached. At the end, the best solution found is returned.

Table 1 compares the basic I-FLATRELAX with its extensions on the Set C. The column *Algorithm* represents the algorithmic variant, the column $\Delta LWU_{\%}$ represents the percentage deviation from the Lawrence upper bound (Lawrence 1984), and finally, the column $\Delta iFlat_{\%}$, represents the percentage deviation from the performance of the original I-FLATRELAX algorithm. The results show that all extensions improve on the base I-FLATRELAX algorithm. The simple use of a partial order schedule as an input solution improves the previous approach described in (Michel & Van Hentenryck 2004) from 4.01% to 3.61%. Note also that the relatively simple introduction of the tabu list also produces a rather interesting further improvement to 2.93%. The best performance (2.80%) is obtained with the I-FLATRELAX-pos-tl-ts-loop. Consequently this variant was selected for the intensive experiments of the second phase.

Intensive evaluation. In the second phase experiments, we compareed three procedures: I-FLATRELAX-pos, I-FLATRELAX-pos-tl-ts-loop (for short labelled I-FLATRELAX-X) and the best results obtained in (Godard, Laborie, & Nuitjen 2005) whose algorithm is called STRand.

In this phase, the following settings have been adopted for I-FLATRELAX: $P_{rem} = 0.2$, $MaxFail = 1000$, $MaxRelaxations = 6$. For the tabu list and the tabu serch

Algorithm	Set A	Set B	Set C	Set D	All
I-FLATRELAX-pos	-0.03	-1.12	1.52	0.51	0.29
I-FLATRELAX-X	-0.05	-1.40	1.03	0.26	0.02
STRand	-0.21	-1.71	-0.33	-0.16	-0.57

Table 2: $\Delta LWU_{\%}$ values on the complete benchmark

we preserved the same parameters as before. We note that the selection of the values of these parameters was not random, but rather the result of a set of preliminary exploratory runs. Finally, for the intensive evaluation we set a timeout of 8000 seconds for each of the 80 instances of the MCJSSP benchmark.

The results confirm the earlier opbservation that the composed strategy I-FLATRELAX-X improves over the performance of I-FLATRELAX-pos. Additionally, the introduction of the tabu list and the tabu search in the iterative flattening schema further boosts the performance of the approach described in (Michel & Van Hentenryck 2004). In fact, although in (Michel & Van Hentenryck 2004) the authors use a more intensive search, the average $\Delta LWU_{\%}$ is only about 1%, which is significantly higher than the 0.02% obtained with our extensions.

The I-FLATRELAX-X configuration does not achieve the level of performance of the STRand procedure (Godard, Laborie, & Nuitjen 2005), which currently maintains the best performance on the MCJSSP benchmarks. However in this case it is worth again noting that while this approach shares the same search schema of I-FLAT it uses different components to implement the flattening and the relaxation steps. Additional analysis is needed to assess the real difference between I-FLATRELAX-X and STRand given that the results of the latter are from the original paper and take advantage of an implementation built on top of the ILOG suite. It may also be the case that the STRand approach can benefit from one or more of the same introductions that we have introduced into I-FLATRELAX-X In fact, this empirical experimentation demonstrates the effectiveness of both the tabu list mechanism and the use of a companion tabu search strategy in conjunction with the iterative flattening search schema.

Conclusions and Future Work

In this paper we have explored a set of extensions to iterative flattening search, a local search procedure for solving large-scale scheduling problems with a makespan minimization opjective criterion (Cesta, Oddi, & Smith 2000; Michel & Van Hentenryck 2004). These extensions were motivated by three potential limitations in the I-FLAT algorithm: (1) the lack of flexibility in the initial seed solution, (2) the potential for repeatedly searching the same solution subspace due to the nature of the relaxation (i.e., constraint retraction) step, and (3) The inabilitty of I-FLAT procedure to explore the close neighborhood of a near-optimal solution for purposes of further improvement. To address these issues the I-FLAT search schema was enhanced in several ways, including use of a partial order schedule as the initial seed solution, the addition of a *tabu-list* to better control the relaxation step, and integration with a complementary *tabu-seach* procedure to refine solutions found by means of I-FLAT.

The proposed extensions were found to significantly improve the performance of the reference strategies on benchmark multi-capacitated job-shop scheduling problems, and these results give first experimental evidence of the effectiveness of coupling tabu search concepts and procdures with iterative flattening search. Further study will be necessary to clearly understand the effectiveness of the algorithms proposed, especially with regard to the best results available in the current literature and given by STRand (Godard, Laborie, & Nuitjen 2005). However, we believe that the proposed extensions are quite general and can be also usefully used within the STRand algorithm.

There are several directions for future research. One particular interest is investigation of a more sophisticated tabu list mechanism, which biases the tenure value according to the estimated quality of a given constraint. Another general focus will be exploration of alternative approaches to integrating iterative flattening and tabu search. In this regard, we believe a Back Jumping Tracking schema (Nowicki & Smutnicki 1996), where search is restarted from promising solutions accumulated during the search, holds particular promise.

References

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

Cesta, A.; Oddi, A.; and Smith, S. F. 2000. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. In *AAAI/IAAI, Seventeenth National Conference on Artificial Intelligence*, 742–747.

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

Godard, D.; Laborie, P.; and Nuitjen, W. 2005. Randomized Large Neighborhood Search for Cumulative Scheduling. In *Proceedings of the International Conference on Automated Planning & Scheduling (ICAPS 2005)*.

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.

Michel, L., and Van Hentenryck, P. 2004. Iterative relaxations for iterative flattening in cumulative scheduling. In *ICAPS*, 200–208.

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.

Policella, N.; Smith, S. F.; Cesta, A.; and Oddi, A. 2004. Generating robust schedules through temporal flexibility. In *ICAPS*, 209–218.