Scheduling Abstractions for Local Search

Pascal Van Hentenryck and Laurent Michel

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-04-01
January 2004

Scheduling Abstractions for Local Search

Pascal Van Hentenryck[1] and Laurent Michel[2]

[1] Brown University, Box 1910, Providence, RI 02912
[2] University of Connecticut, Storrs, CT 06269-3155

Abstract. Comet is an object-oriented language supporting a constraint-based architecture for local search. This paper presents a collection of abstractions, inspired by constraint-based schedulers, to simplify scheduling algorithms by local search in Comet. The main innovation is the computational model underlying the abstractions. Its core is a precedence graph which incrementally maintains a candidate schedule at every computation step. Organized around this precedence graph are differentiable objects, e.g., resources and objective functions, which support queries to define and evaluate local moves. The abstractions enable Comet programs to feature declarative components strikingly similar to those of constraint-based schedulers and search components expressed with high-level modeling objects and control structures. Their benefits and performance are illustrated on two applications: minimizing total weighted tardiness in a job-shop and cumulative scheduling.

1 Introduction

Historically, most research on modeling and programming tools for combinatorial optimization has focused on systematic search, which is at the core of branch & bound and constraint satisfaction algorithm. It is only recently that more attention has been devoted to programming tools for local search and its variations (e.g., [2, 20, 15, 5, 8, 19]). Since constraint programming and local search exhibit orthogonal strengths for many classes of applications, it is important to design and implement high-level programming tools for both paradigms.

Comet [9, 18] is a novel, object-oriented, programming language specifically designed to simplify the implementation of local search algorithms. Comet supports a constraint-based architecture for local search organized around two main components: a declarative component which models the application in terms of constraints and functions, and a search component which specifies the search heuristic and meta-heuristic. Constraints, which are a natural vehicle to express combinatorial optimization problems, are *differentiable objects* in Comet: They maintain a number of properties incrementally and they provide algorithms to evaluate the effect of various operations on these properties. The search component then uses these functionalities to guide the local search using multidimensional, possibly randomized, selectors and other high-level control structures [18]. The architecture enables local search algorithms to be high-level, compositional, and modular. It is possible to add new constraints and to modify or remove existing ones, without having to worry about the global effect of these changes. COMET also separates the modeling and search components, allowing programmers to experiment with different search heuristics and meta-heuristics without affecting the problem modeling. COMET has been applied to many applications and can be implemented to be competitive with tailored algorithms, primarily because of its fast incremental algorithms [9].

This paper focuses on scheduling and aims at fostering the modeling features of COMET for this important class of applications. It is motivated by the remarkable success of constraint-based schedulers (e.g., [13]) in modeling and solving scheduling problems using constraint programming. Constraint-based schedulers, CB-schedulers for short, provide high-level concepts such as activities and resources which considerably simplify constraint-programming algorithms. *The integration of such abstractions within* COMET *raises interesting challenges due to the fundamentally different nature of local search algorithms for scheduling.* Indeed, in constraint-based schedulers, the high-level modeling abstractions encapsulate global constraints such as the edge finder and provide support for search procedures dedicated to scheduling. In contrast, local search algorithms move from (possibly infeasible) schedules to their neighbors in order to reduce infeasibilities or to improve the objective function. Moreover, local search algorithms for scheduling typically do not perform moves which assign the value of some decision variables, as is the case in many other applications. Rather, they walk from schedules to schedules by adding and/or removing sets of precedence constraints.[3] This is the case in algorithms for job-shop scheduling where makespan (e.g., [6, 12]) or total weighted tardiness (e.g., [3]) is minimized, flexible job-shop scheduling where activities have alternative machines on which they can be processed (e.g., [7]), and cumulative scheduling where resources are available in multiple units (e.g., [1]) to name only a few.

This paper addresses these challenges and shows how to support traditional scheduling abstractions in a local search architecture. Its main contribution is a novel computational model for the abstractions which captures the specificities of scheduling by local search. The core of the computational model is *an incremental precedence graph*, which specifies a candidate schedule at every computation step and can be viewed as a complex incremental variable. Once the concept of precedence graph is isolated, scheduling abstractions, such as resources and tardiness functions, become *differentiable objects* which maintain various properties and how they evolve under various local moves.

The resulting computational model has a number of benefits. From a programming standpoint, local search algorithms are short and concise, and they are expressed in terms of high-level concepts which have been shown robust in the past. In fact, their declarative components closely resemble those of CB-schedulers, although their search components radically differ. From a computational standpoint, the computational model smoothly integrates with the constraint-based architecture of COMET, allows for efficient incremental algorithms, and induces a reasonable overhead. From a language standpoint, the

[3] We use *precedence constraints* in a broad sense to include distance constraints.

computational model suggests novel modeling abstractions which explicit the structure of scheduling applications even more. These novel abstractions make scheduling applications more compositional and modular, fostering the main modeling benefits of COMET and synergizing with its control abstractions.

The rest of this paper is organized as follows. Section 2 describes the computational model and Section 3 provides a high-level overview of the scheduling abstractions. The next two sections present two scheduling applications in COMET: minimizing total weighted tardiness in a job-shop and cumulative scheduling. These applications were chosen since they illustrate interesting aspects of COMET. Section 6 concludes the paper.

For space reasons, we do not include a traditional job-shop scheduling algorithm. The search component of one such algorithm [6] was described in [18] and its declarative component is essentially similar to the first application herein. Reference [18] also contains experimental results showing that COMET compares very well with the original algorithm and a `C++` implementation of that algorithm. Note also that other applications, such as flexible job-shop scheduling, essentially follow the same pattern.

2 The Computational Model

The main innovation underlying the scheduling abstractions is their computational model. The key insight is to recognize that most local search algorithms move from schedules to their neighbors by adding and/or removing precedence constraints. Some algorithms add precedence constraints to remove infeasibilities, while others walk in between feasible schedules by replacing one set of precedence constraints by another. Moreover, the schedules in these algorithms always satisfy the precedence constraints, but may violate other constraints.

As a consequence, the core of the computational model is an *incremental precedence graph* which collects the set of precedence constraints between activities and specifies a *candidate schedule* at every computation step. The candidate schedule associates with each activity its earliest starting date consistent with the precedence constraints. It is incrementally maintained during the computation under insertion and deletion of precedence constraints using incremental algorithms such as those in [10].

Once the precedence graph is introduced as the core concept, it is natural to view traditional scheduling abstractions (e.g., cumulative resources) as differentiable objects. A resource now maintains its violations with respect to the candidate schedule, i.e., the times where the demand for the resource exceeds its capacity. Similarly, COMET features differentiable objects for a variety of objective functions such as the makespan and the tardiness of an activity. These objective functions maintain their values, as well as a variety of additional data structures to evaluate the effect of a variety of local moves.

Although it is a significant departure from traditional local search in COMET, this computational model smoothly blends in the overall architecture of the language. Indeed, the precedence graph can simply be viewed as an incremental variable of a more complex type than integers or sets. Similarly, the scheduling abstractions are differentiable objects built on top of the precedence graph and its candidate schedule. Each differentiable object can encapsulate efficient incremental algorithms to maintain its properties and to implement its differentiable queries, exploiting the problem structure.

The overall computational model shares some important properties with CB-schedulers, including the distinguished role of precedence constraints in both architectures. Indeed, CB-schedulers can also be viewed as being implicitly organized around a precedence graph obtained by relaxing the resource constraints.[4] The fundamental difference, of course, lies in how the precedence graph is used. In COMET, it specifies the candidate schedule and the scheduling abstractions are differentiable objects maintaining a variety of properties and how they vary under local moves. In CB-schedulers, the precedence graph reduces the domain of variables and the scheduling abstractions encapsulate global constraints, such as the edge finder, which derive various forms of precedence constraints.

3 Overview of the Scheduling Abstractions

This section briefly reviews some of the scheduling abstractions. Its goal is not to be comprehensive, but to convey the necessary concepts to approach the algorithms described in subsequent sections. As mentioned, the abstractions were inspired by CB-schedulers but differ on two main aspects. First, although the abstractions are the same, their interfaces are radically different. Second, COMET features some novel abstractions to expose the structure of scheduling applications more explicitly. These new abstractions often simplify search components, enhance compositionality, and improve performance.

Scheduling applications in COMET are organized around the traditional concepts of schedules, activities, and resources. The snippet

```
Schedule sched(mgr);
Activity a(sched,4); Activity b(sched,5);
a.precedes(b);
sched.close();
```

introduces the most basic concepts. It declares a schedule `sched`, two activities `a` and `b` of duration 4 and 5, and a precedence constraint between `a` and `b`. This excerpt highlights the high-level similarity between the declarative components of COMET and constraint-based schedulers. *What is innovative in* COMET *is the computational model underlying these modeling objects, not the modeling concepts themselves.* In constraint-based scheduling, these instructions create domain-variables for the starting dates of the activities and the precedence constraints reduce their domains. In COMET, these instructions specify a *candidate schedule* satisfying the precedence constraints. For instance, the above snippet assigns starting dates 0 and 4 to activities `a` and `b`. The expression `a.getESD()` can be used to retrieve the starting date of activity `a` which typically vary over time as the local search moves from candidate schedules to their neighbors.

[4] Note that the precedence graph is now explicit in some CB-schedulers [4].

Schedules in COMET always contain two basic activities of zero duration: the source and the sink. The source precedes all other activities, while the sink follows every other activity. The availability of the source and the sink often simplifies the design and implementation of local search algorithms.

Jobs Many local search algorithms rely on the job structure to specify their neighborhood, which makes it natural to include jobs as a modeling object for scheduling. This abstraction is illustrated in Section 4, where critical paths are computed. A job is simply a sequence of activities linked by precedence constraints. The structure of jobs is specified in COMET through precedence constraints. For instance, the snippet

```
1.   Schedule sched(mgr);
2.   Job j(sched);
3.   Activity a(sched,4); Activity b(sched,5);
4.   a.precedes(b,j);
5.   sched.close();
```

specifies a job `j` with two activities `a` and `b`, where `a` precedes `b`. This snippet also highlights an important feature of COMET: Precedence constraints can be associated with modeling objects such as jobs and resources (see line 4). This functionality simplifies the description of local search algorithms which may retrieve subsets of precedence constraints easily. Since each activity belongs to at most one job, COMET provides methods to access the job predecessors and successors of each job. For instance, the expression `b.getJobPred()` returns the job predecessor of `b`, while `j.getFirst()` returns the first activity in job `j`.

Cumulative Resources Resources are traditionally used to model the processing requirements of activities. For instance, the instruction

```
CumulativeResource cranes(sched,5);
```

specifies a cumulative resource providing a pool of 5 cranes, while the instruction

```
a.requires(cranes,2)
```

specifies that activity `a` requires 2 cranes during its execution. Once again, COMET reuses traditional modeling concepts from CB-scheduling and the novelty is in their functionalities. Resources in COMET are not instrumental in pruning the search space: They are differentiable objects which maintain invariants and data structures to define the neighborhood. In particular, a cumulative resource maintains violations induced by the candidate schedule, where a violation is a time t where the demands of the activities executing on r at t in the candidate schedule exceeds the capacity of the resource. Cumulative resources can also be queried to return sets of tasks responsible for a given violation. As mentioned, precedence constraints can be associated with resources, e.g., `a.precedes(b,crane)`, a functionality illustrated later in the paper.

Precedence Constraints It should be clear at this point that precedence constraints are a central concept underlying the abstraction. In fact, precedence constraints are first-class citizens in COMET. For instance, the instruction

```
set{Precedence} P = cranes.getPrecedenceConstraints();
```

can be used to retrieve the set of precedence constraints associated with `crane`.

Disjunctive Resources Disjunctive resources are special cases of cumulative resources with unit capacity. Activities requiring disjunctive resources cannot overlap in time and are strictly ordered in feasible schedules. Local search algorithms for applications involving only disjunctive resources (e.g., various types of jobshop and flexible scheduling problems) typically move in the space of feasible schedules by reordering activities on a disjunctive resource. As a consequence, COMET provides a number of methods to access the (current) disjunctive sequence. For instance, method `d.getFirst()` returns the first activity in the sequence of disjunctive resource `d`, while method `a.getSucc(d)` returns the successor of activity `a` on `d`. COMET also provides a number of local moves for disjunctive resources which can all be viewed as the addition and removal of precedence constraints. For instance, the move `d.moveBackward(a)` swaps activity `a` with its predecessor on disjunctive resource `d`. This move removes three precedence constraints and adds three new ones. Note that such a move do not always result in a feasible schedule: activity `a` must be chosen carefully to avoid introducing cycles in the precedence graph.

Objective Functions One of the most innovative scheduling abstractions featured in COMET is the concept of objective functions. At the modeling level, the key idea is to specify the "global" structure of the objective function explicitly. At the computational level, objective functions are differentiable objects which incrementally maintain invariants and data structures to evaluate the impact of local moves. The ubiquitous objective function in scheduling is of course the makespan which can be specified as follows:

```
Makespan makespan(sched);
```

Once declared, an objective function can be evaluated (i.e., `makespan.eval()`) and queried to determine the impact of various local moves. For instance, the expression `makespan.evalAddPrecedenceDelta(a,b)` evaluates the makespan variation of adding the precedence `a` $\rightarrow$ `b`. Similarly, the effect on the makespan of swapping activity `a` with its predecessor on disjunctive resource `d` can be queried using `makespan.evalMoveBackwardDelta(a,d)`.

The makespan maintains a variety of interesting information besides the total duration of the schedule. In particular, it maintains the latest starting date of each activity, as well as the *critical* activities, which appears on a longest path from the source to the sink in the precedence graph. These information are generally fundamental in defining neighborhood search and heuristic algorithms for scheduling. They can also be used to estimate quickly the impact of a local move. For instance, the expression `makespan.estimateMoveBackwardDelta(a,d)` returns an approximation to the makespan variation when swapping activity `a` with its predecessor on disjunctive resource `d`.

Although the makespan is probably the most studied objective function in scheduling, there are many other criteria to evaluate the quality of a schedule. One such objective is the concept of tardiness which has attracted increasing attention in recent years. The instruction

```
Tardiness tardiness(sched,a,dd);
```

declares an objective function which maintains the tardiness of activity `a` with respect to its due date `dd`, i.e., $\max(0, e - dd)$ where e is the finishing date of activity a in the candidate schedule. Once again, a tardiness object is differentiable and can be queried to evaluate the effect of local moves on its value. For instance, the instruction `tardiness.evalMoveBackwardDelta(a,d)` determines the tardiness variation which would result from swapping activity `a` with its predecessor on disjunctive resource `d`.

The objective functions share the same differentiable interface, thus enhancing their compositionality and reusability. In particular, they combine naturally to build more complex optimization criteria. For instance, the snippet

```
1.  Tardiness tardiness[j in Job](sched,job[j].getLast(),dd[j]);
2.  ScheduleObjectiveSum totalTardiness(sched);
3.  forall(j in Job)
4.    totalTardiness.add(tardiness[j]);
```

defines an objective function `totalTardiness`, another differentiable function, which specifies the total tardiness of the candidate schedule. Line 1 defines the tardiness of every job `j`, i.e., the tardiness of the last activity of `j`. Line 2 defines the differentiable object `totalTardiness` as a sum of objective functions. Lines 3 and 4 adds the job tardiness functions to `totalTradiness` to specify the total tardiness of the schedule. Queries on the aggregate objective `totalTardiness`, e.g., `totalTardiness.evalMoveBackwardDelta(a,d)`, are computed by querying the individual tardiness functions and aggregating the results. It is easy to see how similar code could define maximum tardiness as an objective function.

Disjunctive Schedules We conclude this brief overview by introducing disjunctive schedules which simplify the implementation of various classes of applications such as job-shop, flexible-shop, and open-shop scheduling problems. In disjunctive schedules, activities require at most one disjunctive resource, although they may have the choice between several such resources). Since activities are requiring at most one resource, various methods can now omit the specification of the resource which is now identified unambigously. For instance, the method `tardiness.evalMoveBackwardDelta(a)` evaluates the makespan variation of swapping activity `a` with its predecessor on its disjunctive resource.

4 Minimizing Total Weighted Tardiness in a Job Shop

This section describes a simple, but effective, local seach algorithm for minimizing the total weighted tardiness in a job shop.

The Problem We are given a set of jobs J, each of which being a sequence of activities linked by precedence constraints. Each activity has a fixed duration and a fixed machine on which it must be processed. No two jobs scheduled on the same machine can overlap in time. In addition, each job $j \in J$ is given a due date d_j and a weight w_j. The goal is to find a schedule satisfying the precedence and disjunctive constraints and minimizing the total weighted tardiness of the schedule, i.e., the function $\sum_{j \in J} w_j \max(0, c_j - d_j)$ where c_j is the completion of job j, i.e., the completion time of its last activity. This problem has received increasing attention in recent years. See, for instance, [3, 16, 17].

```
1.  void Jobshop::state() {
2.     sched = new DisjunctiveSchedule(mgr);
3.     act = new Activity[i in ActRange](sched,duration[i]);
4.     res = new DisjunctiveResource[MachineRange](sched);
5.     job = new Job[JobRange](sched);
6.     tardiness = new Tardiness[JobRange];
7.
8.     forall(a in ActRange)
9.        act[a].requires(res[machine[a]]);
10.    forall(j in JobRange) {
11.       Activity last = sched.getSource();
12.       forall(t in TaskRange) {
13.          int a = jobAct[j,t];
14.          last.precedes(act[a]);
15.          last = act[a];
16.       }
17.       tardiness[j] = new Tardiness(sched,last,duedate[j]);
18.    }
19.    obj = new ScheduleObjectiveSum(sched);
20.    forall(j in JobRange)
21.       obj.add(weight[j] * tardiness[j]);
22.    sched.close(); mgr.close();
23. }
```

Fig. 1. Minimizing Total Weighted Tardiness: The Declarative Component.

The Declarative Component The declarative component of this application is depicted in Figure 1. For simplicity, it assumes that the input data is given by a number of ranges and arrays (e.g., `duration`) which are stored in instance variables. Lines 2-6 declare the modeling objects of the application: the schedule, the activities, the disjunctive resources, the jobs, and the tardiness array. The actual tardiness functions are created later in the method. Lines 8 and 9 specify the resource constraints. Lines 10-18 specify both the precedence constraints and the tardiness functions. Lines 11-16 declare the precedence constraints for a given job `j`, while line 17 creates the tardiness function associated with job `j`. Lines 19 defines the objective function as a summation. Lines 20-21 specify the various elements of the summation. Of particular interest is line 21 which defines the weighted tardiness of job `j` by multiplying its tardiness function by its weight. This multiplication creates a differentiable object which can be queried in the same way as the tardiness functions. It is worth highlighting two interesting features of the declarative statement. First, the declarative component of a traditional job-shop scheduling problem minimizing makespan can be easily obtained by replacing lines 6, 17, and 19-21 by the instruction

```
obj = new Makespan(sched);
```

Second, observe the high-level nature of this declarative component and its independence with respect to the search algorithms. It is indeed conceivable to define constraint-based schedulers which would recognize this declarative specification and generate appropriate domain variables, constraints, and objective function. Of course, this strong similarity completely disappears in the search component.

The Search Component Figure 2 depicts the search component of the application. It specifies a simple Metropolis algorithm which swaps activities that are critical for some tardiness function. The top-level method `localSearch` is depicted in Lines 1-13. It first creates an initial schedule (line 2) and an exponential distribution. Lines 6-9 are the core of the local search and they are executed for `maxTrials` iterations. Each such iteration selects a job `j` which is late (line 6), computes a set of critical activities responsible for this tardiness (line 7), and explores the neighborhood for these activities. Line 12 restores the best solution found during the local search.

Method `exploreNeighborhood` (lines 15-30) explores the moves that swap a critical activity with its predecessor on the machine. These moves are guaranteed to be feasible by construction of the critical path. The algorithm selects a critical activity (line 18) and evaluates the move (line 19). The move is executed if it improves the candidate schedule or if it is accepted by the exponential distribution (lines 20-21) and the best solution is updated if necessary (lines 22-25). These basic steps are iterated until a move is executed or for some iterations.

Method `selectCriticalPath` (lines 32-45) is the last method of the component. The key idea is to start from the activity `a` of the tardiness object (i.e., the last activity of its associated job) and to trace back a critical path from `a` to the source. Lines 37 to 43 are the core of the method. They first test if the job precedence constraint is tight (lines 37-38), in which case the path is traced back from the job predecessor. Otherwise, activity `a` is inserted in `C` as a critical activity (the precedence constraint $p \rightarrow a$, where p is the disjunctive predecessor, is critical) and the path is traced back from the disjunctive predecessor.

This concludes the description of the algorithm. The COMET program is concise (its core is about 70 lines of code), it is expressed in terms of high-level scheduling abstractions and control structures, and it automates many of tedious and error-prone aspects of the implementation.

Experimental Results Table 1 depicts the experimental results of the COMET algorithm (algorithm `CT`), on a Pentium IV (2.1mhz) and contrasts them briefly with the large step random algorithm of [3] (algorithm `LSRW`) which typically dominates [17]. These results are meant to show the practicability of the abstractions, not to compare the algorithms in great detail. The parameters were set as follows: `maxIterations` is set to 600,000 iterations (which roughly corresponds to the termination criteria in [3] when machines are scaled), $T = 225$ and `maxLocalIterations=5`. The initial solution is a simple insertion algorithm for minimizing the makespan [21]. Algorithm `CT` was evaluated on the standard benchmarks from [3, 16, 17], where the deadline for job j is given by $\lceil f * \sum_{i=1}^{m} p_{ij} \rceil - 1$. ([3, 16, 17] specify $\lfloor f * \sum_{i=1}^{m} p_{ij} \rfloor$ in their papers but actually

```
void Jobshop::localSearch() {
   bestSoFar = findInitialSchedule();
   distr = new ExponentialDistribution(mgr);
   nbTrials = 0;
   while (nbTrials < maxTrials) {
      select(j in JobRange : tardiness[j].eval() > 0) {
         set{Activity} Criticals = selectCriticalPath(tardiness[j]);
         exploreNeighborhood(Criticals);
      }
      nbTrials++;
   }
   solution.restore(mgr);
}

void Jobshop::exploreNeighborhood(set{Activity} Criticals) {
   int i = 0;
   do {
      select(v in Criticals) {
         int delta = obj.evalMoveBackwardDelta(v);
         if (delta < 0 || distr.accept(-delta/T)) {
            v.moveBackward();
            if (obj.eval() < bestSoFar) {
               solution = new Solution(mgr);
               bestSoFar = obj.eval();
            }
            break;
         }
      }
   } while (i++ < maxLocalIterations);
}

set{Activity} Jobshop::selectCriticalPath(Tardiness t) {
   set{Activity} C();
   Activity a = t.getActivity();
   Activity source = sched.getSource();
   do {
      Activity pj = a.getJobPred();
      if (pj.getEFD() == a.getESD())
         a = pj;
      else {
         C.insert(a);
         a = a.getDisjPred();
      }
   } while (a != source);
   return C;
}
```

Fig. 2. Minimizing Total Weighted Tardiness: The Local Search.

Bench	Opt	#O	BEST	AVG	WST	$\mu(TS)$	LSRW
abz5	1405	45	1405.00	**1409.92**	1464.00	7.76	1451
abz6	436	50	436.00	**436.00**	436.00	2.79	(5) 436
la16	1170	50	1170.00	**1170.00**	1170.00	5.41	(5) 1170
la17	900	41	900.00	956.62	1239.00	6.43	(5) 900
la18	929	48	929.00	929.28	936.00	5.78	(5) 929
la19	948	33	948.00	**950.58**	956.00	9.30	(3) 951
la20	809	33	809.00	821.58	846.00	7.65	(5) 809
la21	464	50	464.00	**464.00**	464.00	2.12	(5) 464
la22	1068	1	1068.00	1100.06	1185.00	11.30	1086
la23	837	1	837.00	**874.32**	879.00	5.99	875
la24	835	50	835.00	**835.00**	835.00	4.86	(5) 835
mt10	1368	44	1368.00	1393.58	1678.00	10.58	(5) 1368
orb1	2568	30	2568.00	**2600.42**	2867.00	13.79	(2) 2616
orb2	1412	5	1412.00	**1431.80**	1434.00	6.10	1434
orb3	2113	14	2113.00	**2171.56**	2254.00	12.08	2204
orb4	1623	20	1623.00	**1639.66**	1682.00	12.51	(1) 1674
orb5	1593	2	1593.00	1700.72	1756.00	12.55	1662
orb6	1792	47	1792.00	**1800.30**	2072.00	7.43	(4) 1802
orb7	590	0	611.00	675.48	766.00	13.26	618
orb8	2429	1	2429.00	**2503.04**	2624.00	11.91	2554
orb9	1316	35	1316.00	1343.80	1430.00	10.71	(3) 1334
orb10	1679	17	1679.00	**1769.90**	1840.00	7.96	1775

Table 1. Minimizing Total Weighted Tardiness: Experimental Results.

use the given formula.) We used the value 1.3 for f which produces the hardest problems in [3] and ran each benchmark 50 times. The table reports the optimal value (O), the number of times `CT` finds the optimum (#O), the best, average, and worst values found by `CT`, as well as the average CPU time to the best solution. The table also reproduces the result given in [3], which only reports the average of `LSRW` over 5 runs and the number of times the optimum was found.

The results are very interesting. `CT` found the optimal solutions on all but one benchmark and generally with very high frequencies. Moreover, its averages generally outperform, or compare very well to, `LSRW`. This is quite remarkable since there are occasional outliers on these runs which may not appear over 5 runs (see, e.g., la18). The average time to the best solution is always below 14 seconds. Overall, these results clearly confirm the jobshop results [18] as far as the practibility of the scheduling abstractions is concerned.

5 Cumulative Scheduling

This section considers cumulative scheduling and describes the implementation in Comet of the algorithm IFlatIRelax [11], a simple, but effective, extension of the original iterative flattening algorithm [1].

The Problem We are given a set of jobs J, each of which consisting of a sequence of activities linked by precedence constraints. Each activity has a fixed duration, a fixed machine on which it executes, and a demand for the capacity of this machine. Each machine $c \in M$ has an available capacity $cap(c)$. The problem is to minimize the earliest completion time of the project (i.e., the makespan), while satisfying all the precedence and resource constraints.

```
1.   void CumulativeShop::state() {
2.      sched = new Schedule(mgr);
3.      act = new Activity[a in Activities](sched,duration[a]);
4.      resource = new CumulativeResource[m in Machines](sched,cap[m]);
5.
6.      forall(t in precedences)
7.         act[t.o].precedes(act[t.d]);
8.      forall(a in Activities)
9.         act[a].requires(resource[machine[a]],dem[a]);
10.     makespan = new Makespan(sched);
11.     sched.close();
12.     nbv = new inc{int}[m in Machines] = resource[m].getNbViolations();
13.     violations = new inc{int}(mgr) <- sum(m in Machines) nbv[m];
14.     mgr.close();
15.     bestSoFar = sum(a in Activities) duration[a];
16.  }
```

Fig. 3. Cumulative Scheduling: The Declarative Component.

The Declarative Component Figure 3 depicts the declarative component for cumulative scheduling. The first part (lines 1-11) is essentially traditional and solver-independent. It declares the modeling objects, the precedence and resource constraints, as well as the objective function. The second part (Lines 12-13) is also declarative but it only applies to local or heuristic search. Its goal is to specify invariants which are used to guide the search. Line 12 collects the violations of each resource in an array of incremental variables, while line 13 states an invariant which maintains the total number of violations. This invariant is automatically updated whenever violations appear or disappear.

The Search Component The search component of iterative flattening is particularly interesting and is depicted in Figure 4. Starting from an infeasible schedule violating the resource constraints (i.e., the candidate schedule induced by the precedence constraints), IFlatIRelax iterates two steps: a flattening step (line 6) which adds precedence constraints to remove infeasibilities and a relaxation step (line 7) which removes some of the added precedence constraints to provide a new starting point for flattening. These two steps are executed for a number of iterations (i.e., *maxIterations*) or until no improved feasible schedule has been found for a number of iterations (i.e., *maxStable*). The algorithm returns the best feasible schedule found after the flattening step (step 1). Note that iterative flattening can be seen as a large step local search, where each step removes (relaxation) and adds (flattening) a large set of precedence constraints.

The flattening step is performed by method `flatten` in lines 12-25. Flattening aims at removing all violations of the cumulative constraints by adding precedence constraints. It selects a violated cumulative constraint c (line 15) and

```
1.  void CumulativeShop::search() {
3.    coin = new UniformDistribution(1..10);
4.    nbStable = 0; int it = 0;
5.    while (it++ < maxIterations && nbStable <= maxStable) {
6.      flatten();
7.      relax();
8.    }
9.    solution.restore();
10. }
12. void CumulativeShop::flatten() {
14.   while (violations)
15.     select(m in Machines : nbv[m] > 0) {
16.       CumulativeResource r = resource[m];
17.       selectMax(t in r.getViolations())(r.getViolationDegree(t)) {
18.         Activity c[][] = r.getMinimalConflicts(t);
19.         selectMax(k in c.rng(), i in c[k].rng(), j in c[k].rng(): i!=j)
                     (c[k][i].getLSD(makespan) - c[k][j].getEFD())
21.           c[k][j].precedes(c[k][i],r);
22.       }
23.     }
24.   updateBestSolution();
25. }
27. void CumulativeShop::relax() {
29.   forall(p in 1..maxRelations) {
30.     Precedence[] critical[m in Machines] =
            resource[m].getCriticalPrecedences(makespan);
31.     forall(m in Machines, i in critical[m].rng())
32.       if (coin.get() <= prRelaxation)
33.         critical[m][i].remove();
34.   }
35. }
```

Fig. 4. Cumulative Scheduling: The Search Component.

chooses the time t with the largest violation (line 16). To remove this violation, the flattening algorithm queries the resource to obtain a number of minimal conflicts (line 18). The minimal conflicts are given by arrays of activities, potentially of different sizes. From these critical sets, the algorithm selects two activities a_i and a_j (lines 19 and 20) and inserts a precedence constraint $a_i \rightarrow a_j$ (line 21). The two activities are chosen carefully in order to minimize the impact on the makespan. More precisely, the flattening algorithm selects the two activities a_i and a_j maximizing $lsd(a_j) - efd(a_i)$ where $lds(a_j)$ denotes the latest starting date of a_j and $efd(a_i)$ is the earliest finishing date of a_i in the candidate schedule. This heuristics choice aims at keeping as much flexibility as possible in the schedule. Lines 15-23 are iterated until all violations are removed. The candidate schedule then satisfies all constraints and method `updateBestSolution` in line 24 possibly updates the best solution and the number of stable iterations.

After the flattening step, the algorithm has a feasible schedule and its set S of precedence constraints. Instead of restarting from scratch, the key idea behind the relaxation step is to consider the precedence constraints introduced

	set A	set B	set MT	set C	set D
min	-0.01	-1.17	0.37	1.7	1.4
avg	1.07	0.47	3.41	4.2	2.4
T	48.55	103.96	127.84	221.3	645

Table 2. Experimental Result of IFlatIRelax on Cumulative Scheduling

by the flattening step and removes them with some probability. Only critical precedence constraints (i.e., constraints which correspond to critical arcs) are considered during relaxation, since only these may decrease the makespan. This relaxation is iterated for a number of iterations to avoid introducing bottlenecks early in the schedule. The relaxation step is implemented by method `relax` in lines 27-35. The algorithm collects all the critical precedence constraints introduced by flattening (line 30) and iterates over them (line 31). Line 32 flips a coin (the distribution is created in line 3) for each precedence constraint and removes the precedence constraint with some probability (lines 32-33). Line 29-33 are iterated for a small number of iterations. Once again, observe the conciseness and high-level description of algorithm IFlatIRelax, which uses high-level modeling concepts both to state the problem and to specify the search procedure.

Experimental Results Algorithm IFlatIRelax was discovered when implementing and analyzing the original iterative flattening algorithm [1] in Comet. A detailed description of its performance results is available in [11] and only a brief summary is presented here, since the main purpose is to illustrate the scheduling abstractions of Comet. Algorithm IFlatIRelax found 21 new best upper bounds to standard benchmarks (with as much as 900 activities) [1, 14] and quickly delivers solutions that are typically within 1% of the best available upper bounds. Table 2 summarizes the results. For each class of benchmarks, the table reports the best and average deviation (in percentage) from the upper bounds on the various classes of problems, as well as the computation times in seconds on Pentium 4 (2.4Ghz). The averages are computed over 100 runs, except for the larger classes (C and D). These results clearly indicate the practicability and benefits from the scheduling abstractions.

6 Conclusion

This paper presented a collection of scheduling abstractions, inspired by CB-schedulers, to simplify the implementation, and enhance the compositionality and reusability, of local search algorithms. The main innovation is the computational model underlying the abstractions. Its core is a precedence graph which incrementally maintains a candidate schedule at every computation step. Organized around this precedence graph are differentiable objects which are encapsulated in the scheduling abstractions such as resources and objective functions. These differentiable objects maintain various properties incrementally and support differentiable queries to evaluate the effect of local moves. The resulting abstractions (which include some novel concepts to make the problem structure even more explicit) and computational model nicely integrate into the COMET architecture, allows declarative components to be strikingly similar to those featured in CB-schedulers, and provides high-level concepts to specify the local search. The abstractions were illustrated with two applications, minimizing of total weighted tardiness in a jobshop and cumulative scheduling. For both applications, the COMET code is short and concise, it uses high-level scheduling concepts, and it exhibits excellent performance.

References

1. A. Cesta, A. Oddi, and S. F. Smith. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. In *AAAI/IAAI*, pages 742–747, 2000.
2. L. Di Gaspero and A. Schaerf. *Optimization Software Class Libraries*, chapter Writing Local Search Algorithms Using EasyLocal++. Kluwer, 2002.
3. W. Kreipl. A large step random wlak for minimizing total weighted tardiness in a job shop. *Journal of Scheduling*, 3:125–138, 2000.
4. P. Laborie. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. *AIJ*, 143(2):151–188, 2003.
5. F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *CP'98*, Pisa, Italy, October 1998.
6. D. M. and T. M. Applying Tabu Search to the Job-Shop Scheduling Problem. *Annals of Operations Research*, 41:231–252, 1993.
7. M. Mastrolilli and L. Gambardella. Effective neighborhood functions for the flexible job shop problem. *Journal of Scheduling*, 3(1):3–20, 2000.
8. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5:41–82, 2000.
9. L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. In *OOPSLA'00*, Seattle, WA, 2002.
10. L. Michel and P. Van Hentenryck. Maintaining longest path incrementally. In *CP'03*, Cork, Ireland, 2003.
11. L. Michel and P. Van Hentenryck. Iterative relaxations for iterative flattening in cumulative scheduling. In *ICAPS'04*, Whistler, BC, Canada, 2004.
12. E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.
13. W. Nuijten and C. Le Pape. Constraint-based job shop scheduling with ilog scheduler. *Journal of Heuristics*, 3:271–286, 1998.
14. W. P. M. Nuijten and E. H. L. Aarts. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *EJOR*, 90(2):269–284, 1996.
15. P. Shaw, B. De Backer, and V. Furnon. Improved local search for CP toolkits. *Annals of Operations Research*, 115:31–50, 2002.
16. M. Singer and M. Pinedo. A computational study of branch and bound techniques for minimizing the totalweighted tardiness in job shops. *IIE Scheduling and Logistics*, 30:109–118, 1997.
17. M. Singer and M. Pinedo. A shifting bottleneck heuristic for minimizing the totalweighted tardiness in job shops. *Naval Research Logistics*, 46(1):1–17, 1999.
18. P. Van Hentenryck and L. Michel. Control abstractions for local search. In *CP'03*, Cork, Ireland, 2003. (Best Paper Award).
19. S. Voss and D. Woodruff. *Optimization Software Class Libraries*. Kluwer, 2002.
20. J. Walser. *Integer Optimization by Local Search*. Springer Verlag, 1998.
21. F. Werner and A. Winkler. Insertion Techniques for the Heuristic Solution of the Job Shop Problem. TR, Technical Universitaet Magdebourg, 1992.