Problem Decomposition and Multi-shot ASP Solving for Job-shop Scheduling

MOHAMMED M. S. EL-KHOLANY
University of Klagenfurt, Austria and Cairo University, Egypt
(*e-mail:* Mohammed.El-Kholany@aau.at)

MARTIN GEBSER
University of Klagenfurt, Austria and Graz University of Technology, Austria
(*e-mail:* Martin.Gebser@aau.at)

KONSTANTIN SCHEKOTIHIN
University of Klagenfurt, Austria
(*e-mail:* Konstantin.Schekotihin@aau.at)

Abstract

Scheduling methods are important for effective production and logistics management, where tasks need to be allocated and performed with limited resources. In particular, the Job-shop Scheduling Problem (JSP) is a well known and challenging combinatorial optimization problem in which tasks sharing a machine are to be arranged in a sequence such that encompassing jobs can be completed as early as possible. Given that already moderately sized JSP instances can be highly combinatorial, and neither optimal schedules nor the runtime to termination of complete optimization methods is known, efficient approaches to approximate good-quality schedules are of interest. In this paper, we propose problem decomposition into time windows whose operations can be successively scheduled and optimized by means of multi-shot Answer Set Programming (ASP) solving. From a computational perspective, decomposition aims to split highly complex scheduling tasks into better manageable subproblems with a balanced number of operations so that good-quality or even optimal partial solutions can be reliably found in a small fraction of runtime. Regarding the feasibility and quality of solutions, problem decomposition must respect the precedence of operations within their jobs and partial schedules optimized by time windows should yield better global solutions than obtainable in similar runtime on the entire instance. We devise and investigate a variety of decomposition strategies in terms of the number and size of time windows as well as heuristics for choosing their operations. Moreover, we incorporate time window overlapping and compression techniques into the iterative scheduling process to counteract window-wise optimization limitations restricted to partial schedules. Our experiments on JSP benchmark sets of several sizes show that successive optimization by multi-shot ASP solving leads to substantially better schedules within the runtime limit than global optimization on the full problem, where the gap increases with the number of operations to schedule. While the obtained solution quality still remains behind a state-of-the-art Constraint Programming system, our multi-shot solving approach comes closer the larger the instance size, demonstrating good scalability by problem decomposition.

KEYWORDS: job-shop scheduling problem, answer set programming, problem decomposition

1 Introduction

Effective scheduling methods are essential for complex manufacturing and transportation systems, where allocating and performing diverse tasks within resource capacity limits is one of the most critical challenges for production management (Uzsoy and Wang 2000). The Job-shop Scheduling Problem (JSP) (Baker 1974; Taillard 1993) constitutes a well-known mathematical abstraction of industrial production scheduling in which sequences of operations need to be processed by machines such that a given objective, like the makespan for completing all jobs or their tardiness w.r.t. deadlines, is minimized. Finding optimal JSP solutions, determined by a sequence of operations for each machine, is an NP-hard combinatorial problem (Garey *et al.* 1976; Lenstra *et al.* 1977; Liu *et al.* 2008). Therefore, optimal schedules and termination guarantees can be extremely challenging or even unreachable for complete optimization methods already for moderately sized instances. For example, it took about 20 years to develop a search procedure able to find a (provably) optimal solution for an instance called FT10 with 10 jobs (Adams *et al.* 1988; Zhang and Wu 2010), each consisting of a sequence of 10 operations to be processed by 10 machines.

In real-world production scheduling, the number of operations to process can easily go into tens of thousands (Da Col and Teppan 2019; Kopp *et al.* 2020; Kovács *et al.* 2021), which exceeds exact optimization capacities even of state-of-the-art solvers for Answer Set Programming (ASP), Mixed Integer Programming (MIP), or Constraint Programming (CP) (Daneshamooz *et al.* 2021; Francescutto *et al.* 2021; Shi *et al.* 2021). Hence, more efficient approaches to approximate good-quality schedules instead of striving for optimal solutions have attracted broad research interest. On the one hand, respective methods include greedy and local search techniques such as dispatching rules (Blackstone *et al.* 1982), shifting bottleneck (Adams *et al.* 1988) and genetic algorithms (Pezzella *et al.* 2008). On the other hand, problem decomposition strategies based on a rolling horizon (Singer 2001; Liu *et al.* 2008) or bottleneck operations (Zhang and Wu 2010; Zhai *et al.* 2014) have been proposed to partition large-scale instances into better manageable sub-problems, where no single strategy strictly dominates in minimizing the tardiness (Ovacik and Uzsoy 2012).

While decision versions of scheduling problems can be successfully modeled and solved by an extension of ASP with Difference Logic (DL) constraints (Gebser *et al.* 2016), implemented by *clingo*[DL] on top of the (multi-shot) ASP system *clingo* (Gebser *et al.* 2019), the optimization capacities of *clingo*[DL] come to their limits on moderately sized yet highly combinatorial JSP instances (El-Kholany and Gebser 2020), for some of which optimal solutions are so far unknown (Shylo and Shams 2018). Successful applications in areas beyond JSP include industrial printing (Balduccini 2011), team-building (Ricca *et al.* 2012), shift design (Abseher *et al.* 2016), course timetabling (Banbara *et al.* 2019), lab resource allocation (Francescutto *et al.* 2021), and medical treatment planning (Dodaro *et al.* 2021), pointing out the general attractiveness of ASP for modeling and solving scheduling problems.

In this paper, we significantly extend our preliminary study (El-Kholany and Gebser 2020) on problem decomposition into time windows and successive schedule optimization through multi-shot ASP modulo DL solving with *clingo*[DL]. The goal of the decomposition is to split highly complex scheduling tasks into balanced portions for which partial

schedules of good quality can be reliably found within tight runtime limits. Then, the partial schedules are merged into a global solution of significantly better quality than obtainable in similar runtime with single-shot optimization on the entire problem. In this process, problem decomposition must satisfy two criteria: *(i)* the precedence of operations within their encompassing jobs to guarantee the feasibility of partial schedules and *(ii)* the ordering by time windows of operations sharing a machine should come close to the sequences of operations in (unknown) optimal schedules. We address computational efficiency as well as solution quality by devising and investigating decomposition strategies regarding the size of time windows and heuristics to choose their operations.

The contributions of our work going beyond the study in El-Kholany and Gebser (2020) are

- In addition to problem decomposition based on the earliest starting times of operations, we consider the most total work remaining criterion and refinements of both strategies by bottleneck machines. We encode static as well as dynamic decomposition variants, the latter taking partial schedules for operations of previous time windows into account, by stratified ASP programs (without DL constraints).
- Since a decomposition into time windows may be incompatible with the optimal sequences of operations sharing a machine, we incorporate overlapping time windows into the iterative scheduling process to offer a chance for revising "decomposition mistakes." Moreover, the makespan objective, which we apply to optimize (partial) schedules, tolerates unnecessary idle times of machines as long as they do not yield a greater scheduling horizon. Therefore, we devise a stratified ASP encoding to postprocess and compress partial schedules by reassigning operations to earlier idle slots available on their machines.
- We experimentally evaluate decomposition strategies varying the number of operations per time window, as well as multi-shot ASP, solving augmented with overlapping and compression techniques on JSP benchmark sets of several sizes. In particular, our experiments demonstrate that successive optimization by multi-shot ASP solving leads to substantially better schedules within tight runtime limits than global optimization on the full problem, where the gap increases with the number of operations to schedule. While the state-of-the-art CP system OR-tools (Perron and Furnon 2019) is still ahead regarding the solution quality, our multi-shot solving approach comes the closer the larger the instance size gets.

The paper is organized as follows: Section 2 briefly introduces ASP along with the relevant extensions of multi-shot solving and DL constraints. In Section 3, we present our successive optimization approach and detail the ASP programs encoding problem decomposition or iterative scheduling by time windows, respectively. Section 4 provides experimental results on JSP benchmark sets, assessing different decomposition strategies as well as the impact of overlapping and compression techniques. Conclusions and future work are discussed in Section 5.

2 Preliminaries

ASP (Lifschitz 2019) is a knowledge representation and reasoning paradigm geared for the effective modeling and solving of combinatorial (optimization) problems. A (first-

order) ASP program consists of *rules* of the form `h :- b₁,...,bₙ.`, in which the head `h` is an atom `p(t₁,...,tₘ)` or a choice `{p(t₁,...,tₘ)}` and each body literal `bᵢ` is an atom `p(t₁,...,tₘ)`, possibly preceded by the default negation connective `not` and/or followed by a condition `: c₁,...,cₗ`, a built-in comparison $t_1 \circ t_2$ with $\circ \in \{<,=\}$, or an aggregate t_0 = `#count{t₁,...,tₘ : c₁,...,cₗ}`. Each t_j denotes a term, that is, a constant, variable, tuple, or arithmetic expression, and each element c_k of a condition is an atom that may be preceded by `not` or a built-in comparison. Roughly speaking, an ASP program is a shorthand for its ground instantiation, obtainable by substituting variables with all of the available constants and evaluating arithmetic expressions, and the semantics is given by *answer sets*, that is, sets of (true) ground atoms such that all rules of the ground instantiation are satisfied and allow for deriving each of the ground atoms in the head of some rule whose body is satisfied. The syntax of the considered ASP programs is a fragment of the modeling languages described in Calimeri *et al.* (2020); Gebser *et al.* (2015), the ground instantiation process is detailed in Kaminski and Schaub (2021), and the answer set semantics is further elaborated in Gebser *et al.* (2015); Lifschitz (2019).

Multi-shot ASP solving (Gebser *et al.* 2019) allows for iterative reasoning processes by controlling and interleaving the grounding and search phases of a stateful ASP system. For referring to a collection of rules to instantiate, the input language of *clingo* supports `#program name(c).` directives, where `name` denotes a *subprogram* comprising the rules below such a directive and the parameter `c` is a placeholder for some value, for example, the current time step in case of a planning problem, supplied upon instantiating the subprogram. Moreover, `#external h : b₁,...,bₙ.` statements are formed similar to rules yet declare an atom `h` as *external* when the body is satisfied: such an external atom can be freely set to true or false by means of the Python interface of *clingo*, so that rules including it in the body can be selectively (de)activated to direct the search.

ASP modulo DL integrates DL constraints (Cotton and Maler 2006), that is, expressions written as `&diff{t₁ - t₂} <= t₃`, in the head of rules. With the exception of the constant `0`, which denotes the number zero, the terms t_1 and t_2 represent DL variables that can be assigned any integer value. However, the difference $t_1 - t_2$ must not exceed the integer constant t_3 if the body of a rule with the DL constraint in the head is satisfied. That is, the DL constraints asserted by rules whose body is satisfied restrict the feasible values for DL variables, and the *clingo*[DL] system extends *clingo* by checking the consistency of DL constraints imposed by an answer set. If these DL constraints are satisfiable, a canonical assignment of smallest feasible integer values to DL variables can be determined in polynomial time and is output together with the answer set.

3 Multi-shot JSP solving

This section describes our successive optimization approach to JSP solving by means of multi-shot ASP with *clingo*[DL]. We start with specifying the fact format for JSP instances, then detail problem decomposition based on earliest starting times of operations, present our ASP encoding with DL constraints for optimizing the makespan of partial

```
operation(1,1,1,3).   operation(1,2,2,3).   operation(1,3,3,1).
operation(2,1,2,4).   operation(2,2,1,6).   operation(2,3,3,2).
operation(3,1,3,9).   operation(3,2,1,3).   operation(3,3,2,8).
```

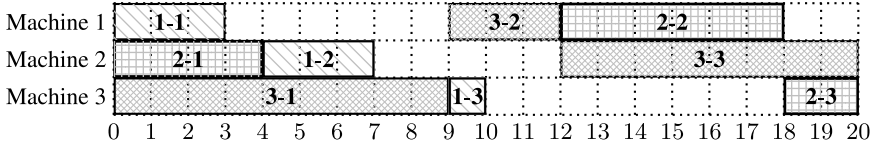Listing 1. *Example JSP instance*

Fig. 1. Optimal schedule for example JSP instance

schedules, and finally outline the iterative scheduling process along with the incorporation of time window overlapping and compression techniques.

3.1 Problem instance

Each job in a JSP instance is a sequence of operations with associated machines and processing times. Corresponding facts for an example instance with three jobs and three machines are displayed in Listing 1.

An atom of the form `operation(`j`,`s`,`m`,`p`)` denotes that the step s of job j needs to be processed by machine m for p time units. For example, the second operation of job 3 has a processing time of 3 time units on machine 1, as specified by the fact `operation(3,2,1,3)`. The operation cannot be performed before the first operation of job 3 is completed, and its execution must not intersect with the first operation of job 1 or the second operation of job 2, which need to be processed by machine 1 as well. That is, a schedule for the example instance must determine a sequence in which to process the three mentioned operations on machine 1, and likewise for operations sharing machine 2 or 3, respectively.

Figure 1 depicts a schedule with the optimal makespan, that is, the latest completion time of any job/operation, for the JSP instance from Listing 1. The **J-S** pairs in horizontal bars indicated for the machines 1, 2, and 3 identify operations by their job **J** and step number **S**. For each machine, observe that the bars for operations it processes do not intersect, so that the operations are performed in sequential order. Moreover, operations belonging to the same job are scheduled one after another. For example, the second operation of job 3 is only started after the completion of the predecessor operation at time 9, regardless of the availability of its machine 1 from time 3 on. As the precedence of operations within their jobs must be respected and the sum of processing times for operations of job 3 matches the makespan 20, it is impossible to reduce the scheduling horizon any further, which in turn means that the schedule shown in Figure 1 is optimal.

3.2 Problem decomposition

Since JSP instances are highly combinatorial and the ground representation size can also become problematic for large real-world scheduling problems, achieving scalability of

```
1 #const n = 2.
2 width((N + n - 1) / n) :- N = #count{J,S : operation(J,S,M,P)}.

4 est(J,1,P,0)       :- operation(J,1,M,P).
5 est(J,S,P,P' + T) :- operation(J,S,M,P), est(J,S - 1,P',T).

7 index(J,S,N) :- est(J,S,P,T),
8       N = #count{J',S' : est(J',S',P',T'), (T',P',J') < (T,P,J)}.

10 window(J,S,(N + W) / W) :- index(J,S,N), width(W).
```

Listing 2. *J-EST decomposition encoding*

complete optimization methods necessitates problem decomposition. In order to enable a successive extension of good-quality partial schedules to a global solution, we consider strategies for partitioning the operations of JSP instances into balanced time windows, each comprising an equal number of operations such that their precedence within jobs is respected. In the following, we first detail problem decomposition based on the earliest starting times of operations, and then outline further strategies that can be encoded by stratified ASP programs as well.

Our encoding for *Job-based Earliest Starting Time (J-EST)* decomposition in Listing 2 takes a JSP instance specified by facts over `operation/4` as input. In addition, a constant `n`, set to the default value 2 in line 1, determines the number of time windows into which the given operations shall be split. As we aim at time windows of (roughly) similar size, the target number of operations per time window is in line 2 calculated by $\lceil N/n \rceil$, where `N` is the total number of operations. For example, we obtain `width(5)` for partitioning the nine operations of the JSP instance in Listing 1 into two time windows.

The rules in lines 4 and 5 encode the J-EST calculation per operation of a job, given by the sum of processing times for predecessor operations belonging to the same job. This yields, for example, `est(3,1,9,0)`, `est(3,2,3,9)`, and `est(3,3,8,12)` for the three operations of job 3 in our example instance, where the third argument of an atom over `est/4` provides the processing time and the fourth the earliest starting time of an operation. Note that the obtained earliest starting times match the first feasible time points for scheduling operations and do thus constitute an optimistic estimation of when to process the operations.

With the earliest starting times of operations at hand, the rule in lines 7-8 determines a *total order* of operations in terms of consecutive indexes ranging from 0. That is, each operation is mapped to the number of operations with (i) a smaller earliest starting time, (ii) the same earliest starting time and shorter processing time, or (iii) a smaller job identifier as a tie-breaker in case of similar earliest starting and processing times. For the example JSP instance in Listing 1, we obtain the indexes 0 to 2 for the first operations of the three jobs, the indexes 3 and 4 for the second operation of job 1 or 2, respectively, in view of their earliest starting times 3 and 4, and indexes from 5 to 8 for the remaining operations. A relevant condition that is guaranteed by such a total order is that indexes increase according to the precedence of operations within their jobs, given that the earliest starting times grow along the sequence of operations in a job.

The last rule in line 10 inspects the total order of operations to partition them into time windows of the size `W` determined by `width(W)`, where only the last time window

```
window(1,1,1).   window(2,1,1).   window(3,1,1).
window(1,2,1).   window(2,2,1).   window(3,2,2).
window(1,3,2).   window(2,3,2).   window(3,3,2).
```

Listing 3. *Example time windows*

may possibly include fewer operations in case the split is uneven. As the ASP program encoding problem decomposition is stratified, its ground instantiation can be simplified to (derived) facts, as shown in Listing 3 for our example instance. Time window numbers from 1 to n = 2 are given by the third argument of atoms over `window/3`, so that the second operation of job 3 and the third operation of each job form the time window 2, while time window 1 consists of the five other operations.

In addition to J-EST decomposition, we have devised a similar ASP program for *Job-based Most Total Work Remaining (J-MTWR)* decomposition, where the total order of operations is decreasing by the sum of processing times for an operation and its successors in a job. For example, we obtain the J-MTWR values 7, 12, and 20 for the first operation of job 1, 2, or 3, respectively, for the JSP instance in Listing 1, matching the time for executing all three operations of each job. Hence, the first operation of job 3 is considered as the most important and is associated with the index 0. The other operations follow the total order taken for partitioning into time windows. As J-MTWR values are decreasing along the sequence of operations in a job, the obtained time windows also respect the precedence of operations.

Beyond partitioning operations in a purely Job-based fashion, we have encoded *Machine-based* decompositions *M-EST* and *M-MTRW* in which an operation from a bottleneck machine with the greatest sum of processing times for yet unordered operations is considered next. For our example instance, the sum of processing times for operations is 12, 15, or again 12, respectively, for the machines 1, 2, and 3. Hence, the operation to be inserted into the total order first is picked from machine 2, where the smallest M-EST or greatest M-MTWR value is used for choosing a particular job/operation. Unlike Job-based decompositions, this may lead to the choice of an operation such that predecessor operations processed by other machines are yet unordered. In this case, they are inserted into the total order directly before the operation from a bottleneck machine, and all machine loads are reduced by processing times before determining the next machine to consider.

Moreover, we have devised *dynamic versions* of the Job- and Machine-based decomposition strategies in which partial schedules are taken into account, and already scheduled operations from previous time windows may change the EST and MTWR values used to order the yet unscheduled operations. However, the principle of applying a stratified ASP program to determine operations for the time window to schedule next is similar for all decomposition strategies, and we experimentally evaluate them in Section 4.

3.3 Problem encoding

Given a JSP instance as in Listing 1 along with facts like those in Listing 3 providing a decomposition into time windows, the idea of successive schedule optimization is to consider time windows one after the other and gradually extend a partial schedule that

fixes the operations of previous time windows. In this process, we adopt the makespan as optimization objective for scheduling the operations of each time window, thus applying the rule of thumb that small scheduling horizons for partial schedules are likely to lead towards a global solution with short makespan. While we use DL variables to compactly represent the starting times of operations to schedule, we assume that a partial schedule for the operations of previous time windows is reified in terms of additional input facts of the form `start((j,s),t,w).`, where t is the starting time scheduled for the step s of job j at the previous time window indicated by w.

The `step(w)` subprogram until line 27 in Listing 4 constitutes the central part of our multi-shot ASP modulo DL encoding, whose parameter `w` stands for consecutive integers from 1 identifying time windows to schedule. Auxiliary atoms of the form `use(m,w',w)`, supplied by the rules in lines 3 and 4, indicate the latest time window $1 \leq w' \leq w$ including some operation that needs to be processed by machine m. The next rule in lines 6-10 identifies pairs (j_1,s_1) and (j_2,s_2) of operations sharing the same machine m, where (j_2,s_2) belongs to the time window w and (j_1,s_1) is either (i) contained in the latest time window $1 \leq w' < w$ indicated by `use(m,w',w-1)` or (ii) also part of the time window w, in which case $j_1 < j_2$ establishes an asymmetric representation for the pair of operations in derived atoms `share((j_1,s_1),(j_2,s_2),p_1,p_2,x,w)`. If the flag $x = 1$ signals that (j_1,s_1) belongs to a previous time window w', the rule in line 12 derives the atom `order((j_1,s_1),(j_2,s_2),p_1,w)` to express that (j_1,s_1) needs to be completed before performing (j_2,s_2), that is, the execution order must comply with the decomposition into time windows. The rule in lines 13–14 yields a similar atom when (j_1,s_1) is the predecessor operation $s_1 = s_2 - 1$ of (j_2,s_2) in the same job $j_1 = j_2$. In contrast to the cases in which (j_1,s_1) must be processed before (j_2,s_2), the choice rule in line 16 allows for performing two operations sharing a machine in the lexicographic order of their jobs if the operations belong to the same time window. In case the atom representing execution in lexicographic order is not chosen, the rule in lines 17 and 18 derives an atom expressing the inverse, given that the operations must not intersect and some sequence has to be determined. Note that applications of the rules in lines 16–18 for choosing an execution order require that the operations (j_1,s_1) and (j_2,s_2) are part of the same time window w and belong to distinct jobs, so that (j_1,s_1) and (j_2,s_2) are out of scope of the rules imposing a fixed order in lines 12–14.

While the rules up to line 18 yield atoms of the form `order(o_1,o_2,p_1,w)`, expressing the hard requirement or choice to perform an operation o_1 with processing time p_1 before the operation o_2 belonging to time window w, the remaining rules assert corresponding DL constraints. To begin with, the starting times of operations o from the previous time window $w-1$ (if any) are in the lines 20 and 21 fixed by restricting them from above or below, respectively, to the value t in an (optimized) partial schedule for time window $w-1$, as supplied by reified facts `start(o,t,w-1)`. In line 23, the lower bound 0 is asserted for the starting time of the first operation of some job, included in the time window w for which a partial schedule is to be determined next. In addition, constraints reflecting the order of performing operations are imposed in line 24, which concerns operations sharing a machine as well as successor operations within jobs. Since such constraints trace the sequence of operations in a job, they establish the earliest starting time, considered for problem decomposition in Section 3.2, as lower bound for scheduling an operation, and the execution order on the machine processing the operation can increase its starting

```
1  #program step(w).

3  use(M,w,w) :- operation(J,S,M,P), window(J,S,w).
4  use(M,W,w) :- use(M,W,w-1), not window(J,S,w) : operation(J,S,M,P).

6  share((J1,S1),(J2,S2),P1,P2,X,w) :- operation(J1,S1,M,P1),
7                                        operation(J2,S2,M,P2),
8                                        window(J1,S1,W),
9                                        window(J2,S2,w), (W,J1) < (w,J2),
10                                       use(M,W,w - X), X = 0..1.

12 order(O1,O2,P1,w)         :- share(O1,O2,P1,P2,1,w).
13 order((J,S1),(J,S2),P,w) :- operation(J,S1,M,P), window(J,S2,w),
14                               S1 = S2 - 1.

16 {order(O1,O2,P1,w)} :- share(O1,O2,P1,P2,0,w).
17  order(O2,O1,P2,w)  :- share(O1,O2,P1,P2,0,w),
18                        not order(O1,O2,P1,w).

20 &diff{O - 0} <=  T :- start(O,T,w - 1).
21 &diff{O - 0} <= -T :- start(O,T,w - 1).

23 &diff{0 - (J,1)} <=  0 :- window(J,1,w).
24 &diff{O1 - O2}   <= -P :- order(O1,O2,P,w).

26 &diff{(J,S) - makespan} <= -P :- operation(J,S,M,P), window(J,S,w),
27                                    not window(J,S + 1,w).

29 #program optimize(m).
30 #external horizon(m).

32 &diff{makespan - 0} <= m :- horizon(m).
```

Listing 4. *Multi-shot ASP modulo DL encoding*

time further. The last rule of the `step(w)` subprogram in lines 26 and 27 asserts that
the value for the DL variable `makespan` cannot be less than the completion time of any
operation of the time window w. That is, the least feasible `makespan` value provides the
scheduling horizon of a partial schedule for operations of time windows up to w.

The task of optimizing the horizon of a (partial) schedule means choosing the exe-
cution order of operations of the latest time window sharing a machine such that the
value for the `makespan` variable is minimized. In single-shot ASP modulo DL solving
with *clingo*[DL], the minimization can be accomplished via the command-line option
`--minimize-variable=makespan`. This option, however, is implemented by means of a fixed
control loop that cannot be combined with (other) multi-shot solving methods. For the
successive optimization by time windows, where the scheduling horizon gradually in-
creases, we thus require a dedicated treatment of DL constraints limiting the value for
`makespan`. To this end, the `optimize(m)` subprogram below line 29 declares an external
atom `horizon(m)` for controlling whether a DL constraint asserted in line 32 is active and
limits the `makespan` value to an integer supplied for the parameter `m`.

Starting with $H := \emptyset$, for $w := 1$ to number n of time windows:

1. For each $h \in H$, set `horizon(`h`)` to false
2. If $w = 1$, then $P := \{$`base`$\}$;
   else $P := \{$`start((`j`,`s`),`t`,`$w-1$`) .` $|$ $\{$`window(`j`,`s`,`$w-1$`)`, `(`j`,`s`)` $= t\} \subseteq X\}$
3. Ground $P \cup \{$`step(`w`)`$\}$
4. While there is some answer set X and the time limit per time window is not reached:
   a. $h := t - 1$, where `(makespan` $= t) \in X$
   b. If $h \notin H$, then $H := H \cup \{h\}$ and ground $\{$`optimize(`h`)`$\}$
   c. Set `horizon(`h`)` to true

Fig. 2. Control loop for successive schedule optimization by multi-shot ASP modulo DL solving.

3.4 Iterative scheduling

The main steps of our control loop for successive schedule optimization by multi-shot ASP modulo DL solving, implemented by means of the Python interface of *clingo*[DL], are displayed in Figure 2 and further detailed in the following. When launching the optimization process for a new time window, any instance of the `horizon(m)` atom introduced before is set to false in step 1 for making sure that some (partial) schedule X is feasible. No such atoms have been introduced yet for the first time window $w = 1$, where the static (default) subprogram called `base`, supplying a JSP instance along with its decomposition in terms of facts over `window/3`, and the `step(1)` subprogram for operations of the first time window are instantiated in steps 2 and 3. Once some schedule X with a horizon $h + 1$ is found in step 4a, the step 4b consists of instantiating the subprogram `optimize(`h`)` on demand, that is, in case h has not been passed as a value for `m` before, and the corresponding `horizon(`h`)` atom is set to true in step 4c for activating the DL constraint reducing the admitted scheduling horizon to h. The step 4 of successively reducing the horizon h in order to find better partial schedules stops when the imposed `makespan` value turns out to be infeasible, meaning that an optimal partial schedule has been found. As already mentioned above, the introduced instances of the external `horizon(m)` atom are then set to false in step 1, and the successive optimization proceeds by in steps 2 and 3 instantiating the `step(`$w + 1$`)` subprogram for the next time window $w + 1$ (if any) and also supplying the determined starting times of operations from time window w by reified facts. The described control loop for successively extending good-quality partial schedules to a global solution can be configured with a time limit, included as secondary stopping condition in step 4, to restrict the optimization efforts per time window and thus make sure that the iterative scheduling progresses.

For illustrating some phenomena going along with problem decomposition and iterative scheduling, let us inspect the schedule in Figure 3 that can be obtained with the decomposition into windows given in Listing 3. The separation between the two time windows is indicated by bold double lines marking the completion of the latest operation of the first time window on each of the three machines. Notably, the partial schedule for the first time window as well as its extension to the second time window are optimal in terms of makespan. However, the obtained global solution has a makespan of 21 rather than just 20 as for the optimal schedule in Figure 1. The reason is that the second operation
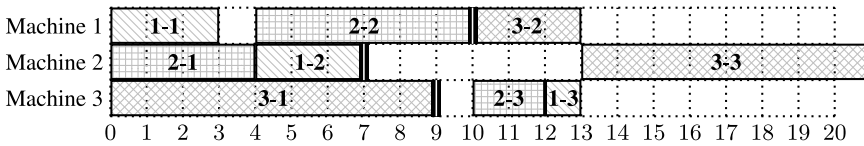
Fig. 3. Decomposed schedule for example JSP instance.

of job 3 would need to be scheduled before the second operation of job 2 on machine 1, while the decomposition into time windows dictates the inverse order and necessitates the later completion of job 3. Given the resulting buffer for scheduling operations with comparably short processing times on machine 3, the third operation of job 1 can be performed after the third operation of job 2 without deteriorating the makespan, yet introducing an unnecessary idle time of machine 3 from 9 to 10 that could be avoided by choosing the inverse execution order. Even though this may seem negligible for the example instance at hand, idle slots can potentially propagate when a partial schedule gets extended to later time windows.

In order to counteract limitations of window-wise successive optimization due to "decomposition mistakes" and unnecessary idle times that do not directly affect the makespan to some extent, we have devised two additional techniques that can be incorporated into the iterative scheduling process. The first extension is time window *overlapping*, where a configurable percentage of the operations per time window can still be rescheduled when proceeding to the next time window. To this end, an (optimized) partial schedule is postprocessed and the configured number of operations to overlap are picked in decreasing order of starting times. For example, if one operation from the first time window is supposed overlap for the decomposed schedule in Figure 3, the second operation of job 2 whose starting time 4 is latest may be chosen, which in turn enables its processing after the second operation of job 3 by rescheduling together with operations of the second time window. The addition of overlapping operations to the encoding in Listing 4 requires handling them similar to operations of the time window to schedule, that is, enabling the choice of an execution order by the rules in lines 16-18 rather than fixing the order by the rule in line 12. As the second extension, we can make use of a stratified ASP program to postprocess an (optimized) partial schedule by inspecting operations of the latest time window in the order of starting times whether idle slots on their machines allow for an earlier execution. The starting times obtained by the *compression* are then taken instead of those calculated in the actual schedule optimization, and, for example, the starting time 12 of the third operation of job 1 would be turned into 9 to fill the idle slot available on machine 3 when compressing the decomposed schedule displayed in Figure 3.

4 Experiments

In order to evaluate our multi-shot ASP modulo DL approach to JSP solving, we ran experiments on JSP benchmark sets by Taillard (1993) of several sizes, considering ten instances each with 50 × 15, 50 × 20, or 100 × 20 jobs and machines. These are the largest instances provided by common JSP benchmark sets, so that complete optimization methods do not terminate within tight runtime limits and the amount of opera-

Table 1. *Experiments varying the number of time windows on JSP benchmark sets of three sizes*

Size	Average	1	2	3	4	5	6	10
50 × 15	Makespan	3542.1	3149.7	**3083.8**	3110.7	3213.6	3225.3	3524.9
	Time	1000.0	1000.0	1000.0	934.4	615.5	243.8	18.5
	Interrupts	1.0	2.0	3.0	3.6	2.7	1.0	0.1
50 × 20	Makespan	3879.4	3479.4	3290.5	**3260.3**	3355.5	3414.8	3793.1
	Time	1000.0	1000.0	1000.0	944.9	583.4	321.5	33.3
	Interrupts	1.0	2.0	3.0	3.5	2.5	1.6	0.1
100 × 20	Makespan	46786.1	26210.4	7201.5	6288.7	6106.1	**6031.9**	6234.0
	Time	1000.0	1000.0	1000.0	1000.0	1000.0	1000.0	861.8
	Interrupts	1.0	2.0	3.0	4.0	5.0	6.0	7.9

tions makes problem decomposition worthwhile. The instances are generated such that each job includes one operation per machine, for example, 15 operations in case of the 50 × 15 instances, where the sequence in which the operations of a job are allocated to machines varies. In our experiments, we assess decomposition strategies presented in Section 3.2 with different number and size of time windows as well as the impact of time window overlapping and compression techniques described in Section 3.4. For the comparability of results between runs with a different number of time windows and respective optimization subproblems addressed by multi-shot solving, we divide the total runtime limit of 1000 s for *clingo*[DL] (version 1.3.0) by the number of time windows to evenly spend optimization efforts on subproblems. Our experiments have been run on an Intel® Core™ i7-8650U CPU Dell Latitude 5590 machine under Windows 10.[1]

Our first experiments, shown in Table 1, concern instances with 50 × 15, 50 × 20, and 100 × 20 jobs and machines with 750, 1000, or 2000 operations, respectively. These operations were split into time windows with the M-EST decomposition strategy as a baseline because it leads to more robust solution quality than Job-based problem decompositions. For each of the three problem sizes, we report the average makespan over ten instances, where smaller values indicate better schedules, the average runtime of *clingo*[DL], and the average number of interrupted optimization runs on subproblems. Each run was executed with a total timeout of 1000 s, where each of n subproblems got at most $1000/n$ s computation time.

For every set of instances, we gradually increase the number of time windows from 1 to 6 and additionally include results for 10 time windows to outline the trend of degrading solution quality when the partition into time windows becomes too fragmented for merging (optimized) partial schedules into proficient global solutions. In fact, the shortest makespans, highlighted in boldface, were obtained with problem decomposition into 3, 4, or 6 time windows, respectively, for the instances of size 50 × 15, 50 × 20, and 100 × 20. Compared to the results for 1 time window, which represent global optimization on the full problem, we observe substantial makespan improvements, amounting to almost one

---

[1] The benchmarks and our implementation are available at: https://github.com/prosysscience/Job-Shop-Scheduling/tree/master/ICLP%20Paper/Implementation

Table 2. *Experiments comparing Job- and Machine-based problem decomposition strategies*

Size Strategy	50 × 15			50 × 20			100 × 20		
	Makespan	Time	Ints	Makespan	Time	Ints	Makespan	Time	Ints
J-EST	3524.8	1000.0	2.0	3540.8	1000.0	2.0	6964.4	1000.0	4.0
J-MTWR	3456.2	1000.0	2.0	3561.0	1000.0	2.0	7149.6	1000.0	4.0
M-EST	3083.8	1000.0	3.0	**3260.3**	944.9	3.5	6031.9	1000.0	6.0
M-MTWR	3060.3	1000.0	3.0	3287.9	986.1	3.9	**5948.6**	1000.0	7.0
M-EST^d	3098.4	1000.0	3.0	3309.5	1000.0	4.0	6115.1	1000.0	7.0
M-MTWR^d	**3041.6**	1000.0	3.0	3287.0	980.4	3.9	6167.3	1000.0	5.0

order of magnitude for the largest instances with 100 × 20 jobs and machines. These advantages are not surprising, considering that the JSP instances are highly combinatorial (Shylo and Shams 2018) and each global optimization run times out with a more or less optimized solution, where a good-quality schedule is the more challenging to find, the greater the number of operations gets. Except for about 12% completed optimization runs in multi-shot solving with 4 time windows on the 50 × 20 instances, the best schedules were obtained with optimization runs timing out on subproblems. Nevertheless, they resulted in good-quality partial schedules that can be merged into better global solutions than achievable with more time windows, which become smaller and easier to solve but also cut off good-quality schedules.

In Table 2, we provide a comparison of the Job- and Machine-based EST and MTWR approaches with dynamic versions of M-EST^d and M-MTWR^d that determine M-EST and M-MTWR values w.r.t. partial schedules from previous time windows. For each decomposition strategy and JSP instance size, we present results for the number of time windows, which amounts to the next greater integer of the average number of interrupted optimization runs in columns headed by Ints, leading to the shortest average makespan. The considerably increased average makespans in the first two rows clearly indicate that time windows determined with Job-based decomposition strategies are less adequate than those investigating bottleneck machines in the first place, and then picking their operations based on the smallest EST or greatest MTWR value as a secondary criterion. While Machine-based decomposition turns out to be advantageous, there is no clear gap between the static M-EST and M-MTWR variants as well as their dynamic versions M-EST^d and M-MTWR^d, and the same indifference applies to Job-based decomposition strategies whose dynamic counterparts are thus omitted. That is, the heuristic to schedule operations from highly loaded machines as soon as possible works well for decomposing the considered JSP instances, yet further solution quality differences due to minor deviations in the decomposition strategy or optimization performance of *clingo*[DL] are negligible.

Focusing again on the static M-EST decomposition strategy, Table 3 assesses different percentages of overlapping operations from previous time windows as well as the impact of compressing (optimized) partial schedules in a postprocessing phase. The baseline

Table 3. *Experiments comparing time window overlapping and compression techniques*

	50 × 15			50 × 20			100 × 20		
Size Overlap	Makespan	Time	Ints	Makespan	Time	Ints	Makespan	Time	Ints
0%	3083.8	1000.0	3.0	3260.3	944.9	3.5	6031.9	1000.0	6.0
10%	3050.0	1000.0	3.0	**3194.9**	1000.0	4.0	5979.9	1000.0	6.0
20%	**3046.0**	1000.0	3.0	3200.6	1000.0	4.0	**5954.3**	1000.0	6.0
30%	3051.2	1000.0	3.0	3451.8	1000.0	4.0	6933.3	1000.0	6.0
0%	3010.3	907.5	3.4	3174.5	839.6	3.0	5671.9	1000.0	7.0
10%	2960.6	989.4	3.9	3154.6	956.4	3.9	**5637.5**	1000.0	7.0
20%	2946.9	1000.0	4.0	**3120.3**	1000.0	4.0	5640.7	1000.0	7.0
30%	**2932.9**	1000.0	4.0	3169.0	1000.0	4.0	5722.7	1000.0	7.0

approach as above is included in the first row with 0% of the operations per time window overlap and without the use of compression. The next three rows exclude compression as well but introduce 10%, 20%, or 30% of overlapping operations that can be rescheduled together with operations of the next time window. In view of the considered number of time windows, picked according to the shortest average makespan achieved with 0% overlap, between 18 and 35 operations constitute an overlap of 10%, so that 30% range up to more than 100 operations. Such non-negligible amounts can make the optimization of subproblems for time windows harder, while the moderate overlaps of 10% or 20% manage to consistently improve the solution quality. Even more significant advantages are obtained by applying compression in the last four rows, where the average makespan gets already shorter with 0% overlap. Further improvements of roughly 2% in solution quality are achieved with 20% overlap, which yield consistently better schedules than the smaller overlap of 10% and avoid deteriorations as observed with 30% overlap on the largest instances with 100 × 20 jobs and machines. In summary, a Machine-based problem decomposition strategy along with postprocessing to compress partial schedules as well as a modest overlap of about 20% of the time window size turn out to be most successful for the JSP benchmark sets under consideration.

For putting our results, obtained with single-shot ASP modulo DL solving (1 time window in Table 1) and the best-performing multi-shot solving approach indicated in the lower half of Table 3, in relation, we also ran the state-of-the-art CP system OR-tools (version 9.2) with a timeout of 1000 s, using the JSP encoding provided by Tassel *et al.* (2021). The respective average makespans and distances to the known optima reported by Taillard (1993) are shown in Table 4. Notably, the optimal schedules from the literature have been determined by greedy or local search techniques, which can in general not compute (provably) optimal solutions, yet the optimality of their schedules is guaranteed by theoretical lower bounds that were not supplied as background knowledge to *clingo*[DL] and OR-tools. As a consequence, all runs of the complete OR-tools and *clingo*[DL] solvers are interrupted at the runtime limit and do thus not yield optimality proofs. However, we observe that the schedules of OR-tools are of high quality

Table 4. *Comparison of single- and multi-shot ASP modulo DL solving approaches to OR-tools*

Size	Optima	OR-tools		Multi-shot		Single-shot	
		Makespan	Distance	Makespan	Distance	Makespan	Distance
50 × 15	2773.8	2778.8	5.0	2932.9	159.1	3542.1	768.3
50 × 20	2843.9	2914.3	70.4	3120.3	276.4	3879.4	1035.5
100 × 20	5365.7	5559.3	193.6	5637.5	271.8	46786.1	41420.4

and come close to the optima for the instances of size 50 × 15. The larger the instances get, the closer is our decomposition and multi-shot solving approach, while it remains behind OR-tools. On the one hand, this can be due to "decomposition mistakes" of our method, deteriorating the quality of the resulting global solutions. On the other hand, the CP encoding taken for OR-tools features dedicated interval variables and global constraints, which are tailored to scheduling problems and give it an advantage over the use of DL constraints. Nevertheless, the improvements due to problem decomposition and multi-shot solving are apparent relative to the single-shot solving approach, which is significantly behind for each of the three problem sizes.

5 Conclusions

Our work develops multi-shot ASP modulo DL methods for JSP solving, based on problem decomposition into balanced time windows that respect the operation precedence within jobs and successive makespan optimization while extending partial schedules to a global solution. We demonstrate that splitting highly complex JSP instances into balanced portions, for which partial schedules of good quality can be reliably found within tight runtime limits, leads to significantly better global solutions than obtainable with single-shot ASP modulo DL optimization in similar runtime. Related works taking advantage of single- (Abels *et al.* 2021) and multi-shot solving (Francescutto *et al.* 2021) by *clingo*[DL] to tackle real-world scheduling problems evaluate threshold values on DL variables and perform optimization by means of common `#minimize` statements of *clingo* rather than minimizing the value of a DL variable like `makespan` in Listing 4. Applying a similar approach can be advantageous in our future work as it may potentially improve computational efficiency as well as solution quality further and is even a necessity to switch from the makespan as optimization objective to tardiness w.r.t. deadlines, which are crucial in many application areas. While Machine-based decomposition turned out to be helpful for finding better schedules, the strategies presented in Section 3.2 can be regarded as naive heuristics based on straightforward instance properties, and incorporating machine-learning approaches (El-Kholany *et al.* 2022; Tassel *et al.* 2021) for more adept partitioning is another promising direction for future work. Provided the adjustment of problem decomposition strategies, our multi-shot solving methods can also be applied to the Flow-shop and Open-shop Scheduling Problems (Taillard 1993). Moreover, we aim to generalize our methods further, for example, by considering flexible machine

allocations and varying processing times, and scale them up to large real-world scheduling problems with an order of magnitude more operations than in synthetic benchmarks.

Acknowledgments

References

ABELS, D., JORDI, J., OSTROWSKI, M., SCHAUB, T., TOLETTI, A. AND WANKO, P. 2021. Train scheduling with hybrid ASP. *Theory and Practice of Logic Programming*, *21*, 3, 317–347.

ABSEHER, M., GEBSER, M., MUSLIU, N., SCHAUB, T. AND WOLTRAN, S. 2016. Shift design with answer set programming. *Fundamenta Informaticae*, *147*, 1, 1–25.

ADAMS, J., BALAS, E. AND ZAWACK, D. 1988. The shifting bottleneck procedure for job shop scheduling. *Management Science*, *34*, 3, 391–401.

BAKER, K. 1974. *Introduction to Sequencing and Scheduling.* John Wiley & Sons.

BALDUCCINI, M. 2011. Industrial-size scheduling with ASP+CP. In *LPNMR* 2011, volume 6645. *LNCS*. Springer, 284–296.

BANBARA, M., INOUE, K., KAUFMANN, B., OKIMOTO, T., SCHAUB, T., SOH, T., TAMURA, N. AND WANKO, P. 2019. teaspoon: Solving the curriculum-based course timetabling problems with answer set programming. *Annals of Operations Research*, *275*, 1, 3–37.

BLACKSTONE, J., PHILLIPS, D. AND HOGG, G. 1982. A state-of-the-art survey of dispatching rules for manufacturing job shop operations. *International Journal of Production Research*, *20*, 1, 27–45.

CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., KAMINSKI, R., KRENNWALLNER, T., LEONE, N., MARATEA, M., RICCA, F. AND SCHAUB, T. 2020. ASP-Core-2 input language format. *Theory and Practice of Logic Programming*, *20*, 2, 294–309.

COTTON, S. AND MALER, O. 2006. Fast and flexible difference constraint propagation for DPLL(T). In *SAT* 2006, volume 4121. *LNCS*. Springer, 170–183.

DA COL, G. AND TEPPAN, E. 2019. Industrial size job shop scheduling tackled by present day CP solvers. In *CP* 2019, volume 11802. *LNCS*. Springer, 144–160.

DANESHAMOOZ, F., FATTAHI, P. AND HOSSEINI, S. 2021. Mathematical modeling and two efficient branch and bound algorithms for job shop scheduling problem followed by an assembly stage. *Kybernetes*, *50*, 12, 3222–3245.

DODARO, C., GALATÀ, G., GRIONI, A., MARATEA, M., MOCHI, M. AND PORRO, I. 2021. An ASP-based solution to the chemotherapy treatment scheduling problem. *Theory and Practice of Logic Programming*, *21*, 6, 835–851.

EL-KHOLANY, M. AND GEBSER, M. Job shop scheduling with multi-shot ASP. In *TAASP* 2020. URL: http://www.kr.tuwien.ac.at/events/taasp20/accepted.html

EL-KHOLANY, M., SCHEKOTIHIN, K. AND GEBSER, M. 2022. Decomposition-based job-shop scheduling with constrained clustering. In *PADL* 2022, volume 13165. *LNCS*. Springer, 165–180.

FRANCESCUTTO, G., SCHEKOTIHIN, K. AND EL-KHOLANY, M. Solving a multi-resource partial-ordering flexible variant of the job-shop scheduling problem with hybrid ASP. In *JELIA* 2021, volume 12678. *LNCS*. Springer, 313–328.

GAREY, M., JOHNSON, D. AND SETHI, R. 1976. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, *1*, 2, 117–129.

GEBSER, M., HARRISON, A., KAMINSKI, R., LIFSCHITZ, V. AND SCHAUB, T. 2015. Abstract Gringo. *Theory and Practice of Logic Programming*, *15*, 4-5, 449–463.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T. AND WANKO, P. 2016. Theory solving made easy with clingo 5. In *ICLP (Technical Communications)* 2016, volume 52 of *OASIcs*. Schloss Dagstuhl, 2:1–2:15.

GEBSER, M., KAMINSKI, R., KAUFMANN, B. AND SCHAUB, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, *19*, 1, 27–82.

KAMINSKI, R. AND SCHAUB, T. 2021. On the foundations of grounding in answer set programming. *CoRR*, *abs/2108.04769*.

KOPP, D., HASSOUN, M., KALIR, A. AND MÖNCH, L. 2020. SMT2020—a semiconductor manufacturing testbed. *IEEE Transactions on Semiconductor Manufacturing*, *33*, 4, 522–531.

KOVÁCS, B., TASSEL, P., KOHLENBREIN, W., SCHROTT-KOSTWEIN, P. AND GEBSER, M. Utilizing constraint optimization for industrial machine workload balancing. In *CP* 2021, volume 210. *LIPIcs*. Schloss Dagstuhl, 36:1–36:17.

LENSTRA, J., KAN, R. AND BRUCKER, P. 1977. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, *1*, 343–362.

LIFSCHITZ, V. 2019. *Answer Set Programming*. Springer.

LIU, M., HAO, J. AND WU, C. 2008. A prediction based iterative decomposition algorithm for scheduling large-scale job shops. *Mathematical and Computer Modelling*, *47*, 3–4, 411–421.

OVACIK, I. AND UZSOY, R. 2012. *Decomposition Methods for Complex Factory Scheduling Problems*. Springer.

PERRON, L. AND FURNON, V. 2019. OR-tools. URL: https://developers.google.com/optimization

PEZZELLA, F., MORGANTI, G. AND CIASCHETTI, G. 2008. A genetic algorithm for the flexible job-shop scheduling problem. *Computers & Operations Research*, *35*, 10, 3202–3212.

RICCA, F., GRASSO, G., ALVIANO, M., MANNA, M., LIO, V., IIRITANO, S. AND LEONE, N. 2012. Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming*, *12*, 3, 361–381.

SHI, G., YANG, Z., XU, Y. AND QUAN, Y. 2021. Solving the integrated process planning and scheduling problem using an enhanced constraint programming-based approach. *International Journal of Production Research*, *Latest Articles*.

SHYLO, O. AND SHAMS, H. 2018. Boosting binary optimization via binary classification: A case study of job shop scheduling. *CoRR*, *abs/1808.10813*.

SINGER, M. 2001. Decomposition methods for large job shops. *Computers & Operations Research*, *28*, 3, 193–207.

TAILLARD, E. 1993. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, *64*, 2, 278–285.

TASSEL, P., GEBSER, M. AND SCHEKOTIHIN, K. 2021. A reinforcement learning environment for job-shop scheduling. In *PRL* 2021. URL: https://prl-theworkshop.github.io/prl2021/

UZSOY, R. AND WANG, C. 2000. Performance of decomposition procedures for job shop scheduling problems with bottleneck machines. *International Journal of Production Research*, *38*, 6, 1271–1286.

ZHAI, Y., LIU, C., CHU, W., GUO, R. AND LIU, C. 2014. A decomposition heuristics based on multi-bottleneck machines for large-scale job shop scheduling problems. *Journal of Industrial Engineering and Management*, *7*, 5, 1397–1414.

ZHANG, R. AND WU, C. 2010. A hybrid approach to large-scale job shop scheduling. *Applied intelligence*, *32*, 1, 47–59.