Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning
Applications and Systems Track

Answer-Set Programming for Lexicographical Makespan Optimisation in Parallel Machine Scheduling

Thomas Eiter[1] , **Tobias Geibinger**[1] , **Nysret Musliu**[1,2] , **Johannes Oetsch**[1] , **Peter Skočovský**[1,3] , **Daria Stepanova**[3]
[1] Institute for Logic and Computation, TU Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
[2] CD-Lab Artis, TU Wien,
[3]Bosch Center for AI,
Robert Bosch Campus 1, D-71272 Renningen, Germany
{eiter, oetsch}@kr.tuwien.ac.at, {tgeibing, musliu}@dbai.tuwien.ac.at
{fixed-term.peter.skocovsky, daria.stepanova}@de.bosch.com

Abstract

We deal with a challenging scheduling problem on parallel-machines with sequence-dependent setup times and release dates from a real-world application of semiconductor workshop production. There, jobs can only be processed by dedicated machines, thus few machines can determine the makespan almost regardless of how jobs are scheduled on the remaining ones. This causes problems when machines fail and jobs need to be rescheduled. Instead of optimising only the makespan, we put the individual machine spans in non-ascending order and lexicographically minimise the resulting tuples. This achieves that all machines complete as early as possible and increases the robustness of the schedule. We study the application of Answer-Set Programming (ASP) to solve this problem. While ASP eases modelling, the combination of timing constraints and the considered objective function challenges current solving technology. The former issue is addressed by using an extension of ASP by difference logic. For the latter, we devise different algorithms that use multi-shot solving. To tackle industrial-sized instances, we study different approximations and heuristics. Our experimental results show that ASP is indeed a promising KRR paradigm for this problem and is competitive with state-of-the-art CP and MIP solvers.

1 Introduction

We consider a scheduling problem on unrelated parallel machines that arises in industrial semiconductor production at Bosch. The problem is involved due to several aspects. We have to deal with sequence-dependent setup-times and job release dates on the one hand; on the other hand, setup-times, release dates, and job durations are machine dependent, and jobs can only be scheduled on dedicated machines. Our goal is to maximise the *throughput*, which can be defined as the number of jobs processed per time unit. In principle, minimising the *makespan*, i.e., the total length of the schedule, accomplishes this. However, when dealing with machines with high dedication, we often find that many jobs can be processed only by few machines, thus few machines determine the makespan almost regardless of how jobs are scheduled on the remaining ones. This is not ideal when jobs need to be rescheduled because of, e.g., machine failure, and domain experts expressed the requirement that *"all machines should complete as early as possible"* to give the scheduler freedom for rearrangements.

Instead of optimising only the makespan, we lexicographically minimise the individual machine spans. In particular, we define the *lexicographical makespan* of a schedule as the tuple of all machine spans in non-ascending order. We prefer a schedule with smaller lexicographical makespan over one with a larger one, where we use lexicographical order for comparison. A schedule with minimal lexicographical makespan has therefore also a minimal makespan, but ties are broken using machines that complete earlier.

This idea of *lexicographical optimisation* to produce schedules that show to be robust when they need to be updated has been investigated and confirmed in the context of job scheduling on identical machines in recent work (Letsios, Mistry, and Misener 2021). While these results further support our motivation to use this objective, the algorithms and tool chains developed there cannot be used directly as our problem is significantly more complex due to machine dedication, sequence-dependent setup times, and release dates.

We are specifically interested in using *Answer-Set Programming* (ASP) (Brewka, Eiter, and Truszczyński 2011; Gebser et al. 2012; Lifschitz 2019), a state-of-the-art logic-based KRR paradigm, for our scheduling problem. ASP is interesting for two reasons: first, its expressive modelling language makes it easy to concisely model the problem including the objective function. This allows one to quickly come up with a first prototype that can be evaluated by domain experts and can serve as a blueprint for more sophisticated solutions. Second and more importantly, ASP makes it relatively easy to develop solutions which can be conveniently adapted to problem variations, a feature known as *elaboration tolerance* (McCarthy 1998). This is indeed needed as it is a goal to use similar scheduling solutions for other work centers with slightly different requirements.

While ASP makes modelling easy and provides enough flexibility for future adaptations, the combination of timing constraints and the considered objective function challenges

current solving technology. The former issue is addressed by using an extension of ASP with difference logic (Janhunen et al. 2017). While this solves the issue of dealing with integer domains without blowing up the size of the grounding, it makes expressing optimisation more tricky as current technology does not support complex optimisation of integer variables. We devise different algorithms that use multi-shot solving (Gebser et al. 2019) to accomplish lexicographical optimisation for our scheduling problem.

To tackle industrial-sized instances, we study different approximations and heuristics. In particular, we consider an approximate algorithm where parts of a solution are fixed after solver calls. This allows us to find near-optimal solutions in a short time. Orthogonally to the ASP model, we formulate different *heuristic rules* using a respective ASP extension (Gebser et al. 2013). These rules do not alter the solution space but guide the solver with variable assignments and improve performance.

For an experimental evaluation of our algorithms, we use random instances of various sizes that are generated based on real-life scenarios. In addition, we provide an alternative solver-independent MiniZinc model that can be used by various state-of-the-art MIP and CP solvers. The experiments aim to explore the additional costs needed when using the lexicographical makespan for optimisation instead of the standard makespan and the trade-off between performance and solution quality. Our experimental results show that the lexicographical makespan optimisers produce schedules with small makespans and thus ensure high throughput, while at the same time accomplish our other objective of early completion times for all machines. The ASP-based algorithms scale up to instances of realistic size and demonstrate that ASP is indeed a viable KRR solving paradigm for lexicographical makespan problems.

The rest of this paper is organised as follows. After some background on ASP in the next section, we formally define our scheduling problem including the lexicographical makespan objective in Section 3. We then present exact ASP approaches for lexicographical makespan minimisation in Section 4, and discuss approximation approaches in Section 5. Experiments are discussed in Section 6. We review relevant literature in Section 7, before we conclude in Section 8.

2 Background on ASP

Answer-Set Programming (ASP) (Brewka, Eiter, and Truszczyński 2011; Gebser et al. 2012; Lifschitz 2019) provides a declarative modelling language that allows one to succinctly represent search and optimisation problems, for which solutions can be computed using dedicated ASP solvers.[1][2]

ASP is a compact relational, in essence propositional, formalism where variables in the input language are replaced by constant symbols in a preprocessing step called *grounding*. An ASP program is a set of rules of the form

$$p_1 \mid \ldots \mid p_k \texttt{ :- } q_1, \ldots, q_m, \texttt{ not } r_1, \ldots, \texttt{ not } r_n \texttt{ .}$$

where all p_i, q_j, and r_l are atoms. The head are all atoms before the implication symbol :–, and the body are all the atoms and negated atoms afterwards. The intuitive meaning of this rule is that if all atoms $q_1, \ldots, q_m$ can be derived, and there is no evidence for any of the atoms $r_1, \ldots, r_n$ (i.e., the rule fires) then at least one of $p_1, \ldots, p_k$ has to be derived. An *interpretation* I is a set of atoms. It is an *answer-set* of a program, if all its rules are satisfied in a minimal and consistent way (Gelfond and Lifschitz 1991); intuitively, I must be a $\subseteq$-minimal model of all rules that fire.

A rule with an empty body is called a *fact*, with :– usually omitted. Facts are used to express knowledge that is unconditionally true. A rule with empty head is a *constraint*. The body of a constraint cannot be satisfied by any answer set and is used to prune away unwanted solution candidates.

A common syntactic extension are choice rules of the form

$$i\,\{p_1, \ldots, p_k\}\,j \texttt{ :- } q_1, \ldots, q_m, \texttt{ not } r_1, \ldots, \texttt{ not } r_n \texttt{ .}$$

The meaning is that if the rule fires, then some subset S of $p_1, \ldots, p_k$ with $i \leq |S| \leq j$ has to be true as well.

We use the hybrid system `clingo-dl` (Janhunen et al. 2017)[3] that extends the ASP solver `clingo` by difference logic to deal with timing constraints. A difference constraint is an expression of the form $u - v \leq d$, where u and v are integer variables and d is an integer constant. In contrast to unrestricted integer constraints, systems of difference constraints are solvable in polynomial time. The latter are expressed in `clingo-dl` using *theory atoms* (Gebser et al. 2016). That job `j` starts after its release time, say 10, can be expressed as `&diff{ 0 - start(j) } <= -10`. Here, `0` and `start(j)` are integer variables, where `0` has a fixed value of 0; thus `start(j)` must be at least 10.

We will also use an extension for *heuristic-driven solving* (Gebser et al. 2013) that allows to incorporate domain heuristics into ASP. These heuristics do not change the answer sets of a program but modify internal solver heuristics to bias search. The general form of a heuristic directive is

$$\texttt{\#heuristic } A \texttt{ : } B\texttt{. [}w\texttt{@}p\texttt{,}m\texttt{]}$$

where A is an atom, B is a rule body, and w, p, m are terms. In particular, w is a weight, `@`p is an optional priority, and m is a modifier like `true` or `false`. We will provide further details when we introduce specific heuristic rules later on.

Further features of the input language, like aggregation and optimisation statements, will be explained as we go.

3 Problem Statement

We study the following scheduling problem. Given m machines and n jobs, every job needs to be processed by a single machine, and every machine can process at most one job at a time; preemption is not allowed. Some machines can only handle certain jobs, such that from the view of the latter, $cap(j)$ is the set of machines that can process job j.

We assume that a *release date* $r_{j,k}$ is specified for every job j and machine k as a non-negative integer. Release dates are machine dependent because transportation time for jobs to the machines depends on the transport system and their location. No job can start before its release date.

[1] potassco.org.

[2] www.dlvsystem.com.

[3] https://github.com/potassco/clingo-dl.

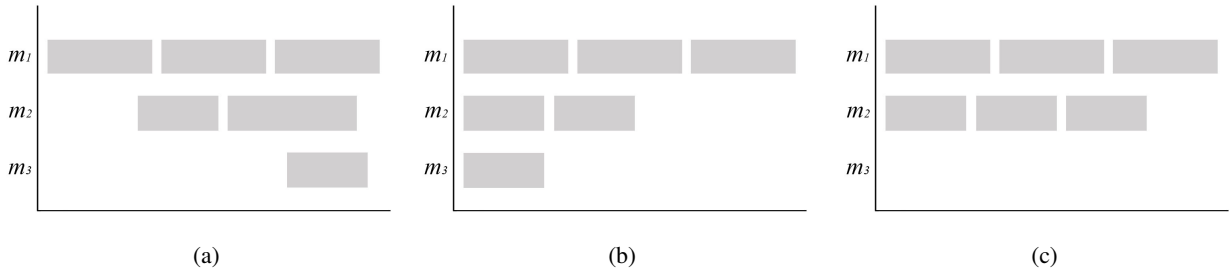

Figure 1: Different schedules involving three machines and six jobs.

A specified amount of time may be required to change from one job to the next one. Specifically, we assume that $s_{i,j,k}$ is the time needed to set up job j directly after job i on machine k. Consequently, these times are referred to as *sequence-dependent setup times*. Every job j has a positive *duration* $d_{j,k}$ that depends on the machine k it is assigned to.

A *schedule* S for a problem instance is defined by:

1. an assignment a that maps each job j to a machine $k \in cap(j)$ capable of processing it;
2. for each machine k, a total order $\preceq_k$ on the set J of jobs assigned to the machine via a. Relation $\preceq_k$ determines the sequence in which the jobs in J are processed on k.

If each job can be processed by some machine, then some schedule for the problem instance exists.

Assume that $j_1, \ldots, j_l$ is the processing sequence of the jobs assigned to machine k in a given schedule. The *processing time* p_{j_i} of a job j_i is its duration plus the setup time for its predecessor (if one exists); i.e., $p_{j_1} = d_{j_1,k}$ and $p_{j_i} = s_{j_{i-1},j_i,k} + d_{j_i,k}$, for $i > 1$. The *start time* st_{j_i} of job j_i is $r_{j_i,k}$ if $i = 1$, and $max(r_{j_i,k}, st_{j_{i-1}} + p_{j_{i-1}})$ for $i > 1$. The *completion time* c_{j_i} of job j_i is $st_{j_i} + p_{j_i}$.

The *machine span* of k, $span(k)$, is the completion time of the last job j_l on k. A common optimisation criterion is to search for a schedule with a small *makespan*, which is the largest machine span of the schedule.

Versions of this problem have been extensively studied in the literature (Allahverdi 2015). The problem presented here abstracts the actual problem statement at Bosch to its most essential elements. We left out some details due to confidentiality. Other elements, like due dates or manual labor costs have been omitted as they are not relevant for the objective function studied in this paper.

3.1 The Lexicographical Makespan Objective

Recall that we are interested in computing schedules that maximise the throughput, but high machine dedication and rescheduling due to sudden machine failure render minimal makespan as a single objective function suboptimal.

Figure 1 (a) illustrates this: assume all jobs scheduled to machine m_1 cannot be processed by any other machine. Thus, m_1 will always determine the makespan and the remaining jobs can be put almost arbitrarily on the remaining machines. This can lead to unnecessary workload on these machines. A more severe problem is when machines suddenly fail and their jobs need to be rescheduled. To cope with events like machine failure, the domain experts formulated the requirement that "all machines should complete as early as possible" with the intention to give the scheduler maximal freedom in rearranging jobs with minimal decrease in throughput.

We next define the *lexicographical makespan* for lexicographical optimisation of machine spans to obtain robust schedules (Letsios, Mistry, and Misener 2021).

Definition 1. *Given a schedule S involving m machines, the* lexicographical makespan*, or lex-makespan for short, of S is the tuple $ms(S) = (c_1, \ldots, c_m)$ of all the machine spans of S in non-ascending order.*

In this definition, c_1 is a maximal machine span and hence corresponds to the makespan.

For schedules S and S' involving m machines each, S has a smaller lex-makespan than S' if $ms(S)$ is smaller than $ms(S')$ under lexicographical order, i.e., on the least index i where $ms(S) = (c_1, \ldots, c_m)$ and $ms(S') = (c'_1, \ldots, c'_m)$ disagree, we have $c_i < c'_i$. For a set $\mathcal{S}$ of schedules, $S \in \mathcal{S}$ is then optimal if $ms(S)$ is minimal over all schedules in $\mathcal{S}$.

Consider Fig. 1 for illustration. We would prefer schedule (b) over (c) under the lex-makespan objective. For both schedules, the lex-makespan is given by the machine spans of m_1, m_2, and m_3 in that order. Both schedules have the same makespan, but schedule (b) has a smaller machine span for m_2. If machine m_1 fails and most of the jobs can only be rescheduled to machine m_2, schedule (b) would indeed be advantageous. It happens also earlier for schedule (b) that machines m_2 and m_3 complete all their jobs and are therefore free if new jobs need to be scheduled.

To describe the dynamics of a schedule, we define, for a time point t and schedule S, $M(S,t)$ as the number of machines that complete at or before t. We then obtain:

Proposition 1. *Let S and S' be two schedules for some problem instance. Then, $ms(S) < ms(S')$ iff there is a time point t such that $M(S,t) > M(S',t)$, and, for every $t' > t$, $M(S,t') \geq M(S',t')$.*

For problems involving many machines, hierarchically minimizing all the machine spans can be excessive if the overall makespan is dominated by few machines only. However, comparing lex-makespans allows for a rather natural parametrisation, namely an integer l that defines the number of components to consider in the comparison.

Definition 2. *Given schedules S and S' involving m machines each and an integer l, $1 \leq l \leq m$, we say $ms(S) = (c_1, \ldots, c_m)$ is smaller than $ms(S') = (c'_1, \ldots, c'_m)$ under* parametrised lexicographical order*, in symbols $ms(S) \leq_l ms(S')$, if under lexicographical order $(c_1, \ldots, c_l) \leq (c'_1, \ldots, c'_l)$.*

Note that for a schedule with m machines, we obtain the makespan if $l = 1$ and the full lex-makespan if $l = m$.

4 An Exact ASP Model with Difference Logic

A problem instance is described by ASP facts using some fixed predicate names. We illustrate this by an example with one machine m_1 and two jobs j_1, j_2. The machine is capable of processing all jobs and all release dates are 0. The setup time is 4 when changing from job j_1 to j_2 and 2 vice versa. Both jobs have duration 5. The according facts are

```
machine(m1). cap(m1,j1). cap(m1,j2).
job(j1). duration(j1,m1,5). release(j1,m1,0).
job(j2). duration(j2,m1,5). release(j2,m1,0).
setup(j1,j2,m1,4). setup(j2,j1,m1,2).
```

Any problem instance can be described using this format.

Next, we present the ASP encoding for computing minimal schedules. The entire program is given in Fig. 2.

Proposition 2. *For every problem instance I, the schedules of I with minimal lex-makespan are in one-to-one correspondence with the optimal answer sets of the rules in Fig. 2 augmented with the fact representation of I.*

The encoding consists of three parts: Lines 1–10 qualitatively model feasible sequences of jobs on machines, while the quantitative model for completion times is realised in Lines 12-19 with difference logic; we avoid by this doing integer arithmetic in the Boolean ASP constraints, which would blow up the size of the grounding. Finally, the optimisation is accomplished by Lines 21-23.

The first line of Fig. 2 expresses that each job is assigned to a machine capable of processing it. The notation `asg(J,M) : cap(M,J)` means that in the grounding step for each value `j` of the global variable `J` (as it occurs in the body), `asg(J,M)` is replaced by all atoms `asg(j,m)` for which `cap(j,m)` can be derived.

We further require that the jobs assigned to a machine are totally ordered. That is, for any two distinct such jobs j_1 and j_2, either $j_1 \prec j_2$ or $j_2 \prec j_1$ holds. This is achieved by the rule in Line 3. In Lines 5–6, the predicates `first/2` and `last/2`, representing the first and last job on each machine, respectively, are defined. Constraints in Lines 9-10 ensure that this selection is compatible with the order given by `before/3`. Furthermore, each job except the last (resp. first) has a unique successor (resp. predecessor); this is captured by `next/3` in Lines 7-8.

We use difference logic to express that jobs are put on the machines in the order defined by `next/3`. The rules in Lines 12–16 closely follow respective definitions from Section 3. Line 17 defines `cmax` as an upper bound of any completion time. In any answer-set, `cmax` will be the actual makespan since the solver will always instantiate integer variables with the smallest value possible. The redundant rule in Line 19 helps the solver to further prune the search space.

For optimisation, we "guess" a span for each machine in Line 21. Here `int/1` is assumed to provide a bounded range of integers. In Line 22, we enforce that machines complete not later than the guessed spans. The actual objective function is defined by the last line of Fig. 2 notably concise: any machine contributes its span c to a cost function at priority level c. The cost function accumulates contributing values and the solver minimises answer sets by lexicographically comparing cost tuples ordered by priority.

Algorithm 1: Lex-Makespan Optimisation

Input: model M involving m machines and parameter l with $1 \leq l \leq m$
Output: schedule R for M with parametrised lex-makespan $(c_1, \ldots, c_l)$

$solve(M)$ … returns a solution for M or $\emptyset$ if none is found within fixed resource limits

$bound(i \circ b), \circ \in \{<, \leq\}$ … constraints enforcing that $c_i \circ b$ for the lex-makespan $(c_1, \ldots, c_m)$

```
1  (c_1, ..., c_l) ← (0, ..., 0)
2  R ← solve(M)
3  for i ← 1 to l do
4      S ← R
5      while S ≠ ∅ do
6          R ← S
7          c_i ← ith element of the lex-makespan of S
8          S ← solve(M ∪ bound(i < c_i))
9      M ← M ∪ bound(i ≤ c_i)
10 return R with lex-makespan (c_1, ..., c_l)
```

4.1 Direct Multi-shot Optimisation

The performance bottleneck for the ASP approach from the previous section is grounding. In particular, the definition of the machine spans must be grounded over the entire relevant integer range. We can define machine spans in difference logic as bounds on completion times similar to the makespan:

```
&diff {   0  - span(M) } <= 0 :- machine(M).
&diff { c(J) - span(M) } <= 0 :- asg(J,M).
```

However, multi-objective optimisation for integer variables with priorities is unfortunately not supported in the current version (1.1.1) of `clingo-dl`. Schedules with minimal lex-makespan are still computable using multiple solver calls and incrementally adding constraints.

To this end, we present Alg. 1 for lex-makespan minimisation by using multiple solver calls.

Alg. 1 is a standard way for multi-objective minimisation by doing a (highest priority first) hierarchical descent. Due to symmetries, showing a lack of solutions is usually more costly for this problem than finding one; this makes alternative strategies with fewer expected solver calls like binary search or exponentially increasing search steps less attractive.

We use `clingo-dl` and the encoding from Fig. 2 without the optimisation statement in Lines 21–23 to implement

```
1  1 { asg(J,M)      : cap(M,J)        } 1 :- job(J).
2
3  before(J1,J2,M)   | before(J2,J1,M)     :- asg(J1,M), asg(J2,M), J1 < J2.
4
5  1 { first(J,M)    : asg(J,M)        } 1 :- asg(_,M).
6  1 { last(J,M)     : asg(J,M)        } 1 :- asg(_,M).
7  1 { next(J1,J2,M) : before(J1,J2,M) } 1 :- asg(J2,M), not first(J2,M).
8  1 { next(J2,J1,M) : before(J2,J1,M) } 1 :- asg(J2,M), not  last(J2,M).
9  :-     first(J1,M), before(J2,J1,M).
10 :-      last(J1,M), before(J1,J2,M).
11
12 &diff{    0  - c(J1)}  <= -(T+D+S) :- asg(J1,M), next(J3,J1,M),
13                                       setup(J3,J1,M,S), duration(J1,M,D), release(J1,M,T).
14 &diff{ c(J2) - c(J1) } <= -(P+S)   :- before(J2,J1,M),  next(J3,J1,M),
15                                       setup(J3,J1,M,S), duration(J1,M,P).
16 &diff{    0  - c(J1) } <= -(T+D)   :- asg(J1,M),        duration(J1,M,D), release(J1,M,T).
17 &diff{ c(J)  - cmax }  <=  0       :- job(J).
18
19 &diff{ c(J2) - c(J1) } <= -P       :- before(J2,J1,M), duration(J1,M,P).
20
21 1 { span(M,T) : int(T) } 1         :- machine(M).
22 &diff{ c(J)  - 0     } <= S        :- asg(J,M), span(M,S).
23 #minimize{ T@T,M  : span(M,T) }.
```

Figure 2: ASP encoding with difference logic for lex-makespan optimisation.

$solve(M)$ in Alg. 1. A handy feature is that `clingo-dl` supports *multi-shot solving* (Gebser et al. 2019) where parts of the solver state are kept throughout multiple runs, thereby saving computational resources. Notably $solve(M)$ in Alg. 1 does not limit us to use ASP solver. We can in principle use any exact method that is capable of producing solutions for a model M in the input language of the respective system.

The constraints for $bound(i \leq b)$ are quite easy to express in ASP: that the i-th component of the lex-makespan is smaller than or equal to b is equivalent to enforcing that at least $m - i + 1$ machines have a span of at most b. We can encode the latter by non-deterministically selecting $m - i + 1$ machines and enforcing that they complete not later than b:

```
(m-i+1) { sel(M) : machine(M) }.
&diff { span(M) - 0 } <= b :- sel(M).
```

While Alg. 1 is guaranteed to return a schedule with minimal lex-makespan when resources for $solve(M)$ are not limited, we will in practice restrict the time spent for search in $solve(M)$ by a suitable time limit.

4.2 Domain-specific Heuristics

We use two domain heuristics to improve performance by guiding search more directly to promising areas of the search space. Both heuristic directives use the modifier `true`: Whenever an atom needs to be assigned a truth value, the solver will pick the one with the highest weight among the ones with highest priority and assigns it to true at first.

Recall that job durations depend on the machines. The first heuristic expresses the idea to assign jobs to machines if their duration is low on that machine.

```
#heuristic asg(J,M) : duration(J,M,D),
  maxDuration(J,F), W=F-D. [W@2,true]
maxDuration(J,M) :- job(J),
  M = #max{ D : duration(J,_,D) }.
```

Here, `maxDuration/2` defines the longest duration of a given job over all machines. Then, the heuristic directive gives a high weight to an atom `asg(j,m)` if the duration of j is low relative to its maximal duration. The priority level of this rule is 2; this means that the solver will try to assign jobs to machines before deciding on other atoms.

The second heuristic directive affects how jobs are put on machines. We want to avoid large setup times and follow an analogous strategy as for the first heuristic:

```
#heuristic next(J,K,M) : setup(J,K,M,S),
  maxSetup(K,M,T),
  cap(J,M), cap(K,M), W=T-S. [W@1,true]
maxSetup(J,M,S) :- job(J), machine(M),
  S = #max{ T : setup(_,J,M,T) }.
```

Atom `next(j,k,m)` gets a high weight if putting job j before k results in a relatively small setup time. We also need to make sure that machine m is actually capable of processing both jobs. The order of the jobs has a lower priority than the machine assignment.

5 ASP-based Approximation

While the exact methods from the previous section have the advantage that we can run a solver until we find a guaranteed optimal solution, this works only for very small problem instances. Finding good solutions within a time limit is in practice more important than showing optimality. This is what the ASP-based approximation method we discuss next are designed to accomplish.

There is a simple way to turn the exact encoding from Fig. 2 into an approximation that scales better to larger instances. It has been introduced for `clingo-dl` optimisation in the context of train scheduling (Abels et al. 2019), and we apply it for our machine scheduling application. Recall

Algorithm 2: Lex-Makespan Approximation

Input: model M involving m machines and parameter l with $1 \leq l \leq m$
Output: schedule S for M with parametrised lex-makespan $(c_1, \ldots, c_l)$

$opt(M)$ . . . returns best solution for M found within fixed resource limits

```
(c_1, ..., c_l) ← (0, ..., 0)
S ← ∅
for i ← 1 to l do
    S ← opt(M)
    c_i ← makespan of S
    remove some machine k that completes at c_i and
      all jobs assigned to k from M
return S
```

that we use `int/1` to define a range $[0, 1, \ldots, n]$ of integers from which potential bounds for individual machine spans are taken from. We can instead consider integers from $[0, 1 \cdot g, \ldots, i \cdot g]$ where g is the granularity of the approximation, and i is chosen such that $n \leq i \cdot g$; i can be significantly smaller than n, depending on g, and thus reduce the search space and the size of the grounding. A larger g makes the approximation more coarse, a smaller g makes it closer to the exact encoding. We will compare the exact encoding and this approximation in Section 6.

5.1 Multi-shot Approximation

We next present a variant of Alg. 1 for approximation, where we assume that we have an exact optimiser $opt(\cdot)$ which is good at finding schedules with small makespans. This optimiser can then be employed to compute schedules with small lex-makespans. The specifics are presented as Alg. 2

Algorithm 2 uses $opt(\cdot)$ to recompute and improve parts of a solution by fixing the jobs on the machine with highest span after each solver call. Similar to Alg. 1, we require multiple solver calls but with a profound difference: the number of solver calls to the makespan optimiser is bounded by the number of machines, and after each solver call, the problem instance is significantly simplified and thus easier to solve.

As `clingo-dl` allows to directly minimise a single integer variable, we can implement the makespan optimiser $opt(\cdot)$ directly using our difference logic encoding and minimise `cmax`. However, we opted for using multi-shot solving again to reuse heuristic values and learned clauses from previous solver runs.

6 Experimental Evaluation

We now provide an experimental evaluation of the approaches for lex-makespan optimisation from above. Notably, any approach that produces schedules with small makespan will also produce small lex-makespans. Our primary goal is to investigate the difference in schedule quality when spending all resources for makespan optimisation versus distributing them for lex-makespan optimisation to different priorities.

	M3	M5	M10	M15	M20
m	3	5	10	15	20
n	5–50	10–50	50–200	100-200	150–200

Table 1: Machines (m) and jobs (n) per instance class.

As we cannot disclose real instances from the semi-conductor production application, we use random instances of realistic size and structure instead. In addition to experiments with ASP-based solvers, to compare with other solving paradigms, we provide a solver-independent model in MiniZinc (Nethercote et al. 2007). This model can be solved by many CP and MIP solvers. However, for a direct comparison, we also model our problem directly in `cplex` and `cpoptimizer`.

6.1 Problem Instances

We generated 500 benchmark instances of different sizes randomly. The random instance generator is designed however to reflect relevant properties of the real instances. The generator is based on previous work in the literature (Vallada and Ruiz 2011a), but also produces instances with high machine dedication and amends older benchmarks, which were designed for different objective, with random release dates. The 500 instances can be grouped into five classes, shown in Table 1, of 100 instances each. The instance generator as well as all the encoding and algorithms are online available.[4]

The machine capabilities were assigned uniformly at random for half of the instances in every class: for each job, a random number of machines were assigned as capable. For the other half, we assigned the capabilities such that 80% of the jobs can only be performed by 20% of the machines. We refer to the latter setting as high-dedication and the former as low-dedication.

For each job j and any machine k, the duration $d_{j,k}$, setup time $s_{j,i,k}$ for any other job i, and release date $r_{j,k}$ were drawn uniformly at random from $[10, 500]$, $[0, 100]$, and $[0, r_{max}]$, respectively, where

$$r_{max} = \frac{1}{m} \sum_{1 \leq j \leq n} \frac{1}{|cap(j)|} \Big(\sum_{k \in cap(j)} d_{j,k} + \sum_{1 \leq j' \leq n, k \in cap(j')} s_{j',j,k} \Big).$$

6.2 A Solver-Independent MiniZinc Model

As an alternative to ASP, we implemented a solver-independent model for schedule optimisation in the well-known high-level modelling language MiniZinc for constraint satisfaction and optimization problems. MiniZinc models, after being compiled into FlatZinc, can be used by a wide range of solvers. As this paper focuses on ASP solving, we provide only one direct MiniZinc model that serves as a baseline. Our model of the problem statement from Section 3 and the objective function are as follows.

For each job $i \in \{1, \ldots, n\}$, we use the following decision variables: $a_i \in \{1, \ldots, m\}$ for its assigned machine, $p_i \in \{0, \ldots, n\}$ representing its predecessor or 0 if it has none, and $c_i \in \{0, \ldots, h\}$ denoting its completion time where h is

[4] https://owncloud.tuwien.ac.at/index.php/s/UmD0lIh2B7B4A9g.

the scheduling horizon. For each machine $j \in \{1, \ldots, m\}$, we use $s_j \in \{1, \ldots, h\}$ denoting its span, and its level in the ordering of spans $l_j \in \{1, \ldots, m\}$.

The constraints enforcing a valid solution can be formulated in the following way.

$$\texttt{alldifferent\_except\_0}(p_{1 \leq i \leq n}) \quad (1)$$

$$\textstyle\sum_{i=1}^{n}(p_i = 0) \leq \texttt{nvalue}(a_{1 \leq i \leq m}) \quad (2)$$

$$a_i \in cap(i) \qquad 1 \leq i \leq n \quad (3)$$

$$p_i \neq 0 \rightarrow a_i = a_{p_i} \qquad 1 \leq i \leq n \quad (4)$$

$$p_i \neq i \qquad 1 \leq i \leq n \quad (5)$$

$$c_i \geq (max(c_j,\ r_{i,a_i}) + s_{j,i,a_i} + d_{i,a_i}) \cdot (p_i = j) \qquad 1 \leq i, j \leq n,\ i \neq j \quad (6)$$

$$c_i \geq r_{i,a_i} + d_{i,a_i} \qquad 1 \leq i \leq n \quad (7)$$

The global constraint (1) ensures that no two jobs have the same predecessor, while (2) enforces that the number of first jobs is less or equal to the number of assigned machines. The latter is determined through a global `nvalue` constraint returning the number of different values in $a_{1 \leq i \leq m}$. Constraint (3) ensures that every job is assigned a capable machine, and constraint (4) ensures that a job's predecessor is on the same machine. Constraint (5) expresses that no job is its own predecessor, while (6) ensures that each job starts after its predecessor and the corresponding setup time. Finally, (7) enforces that every first job starts after its release date.

Defining an objective function for lex-makespan minimisation is more intricate. For this, we add some constraints:

$$s_k = max_{1 \leq i \leq n}(c_i \cdot (a_i = k)) \qquad 1 \leq k \leq m \quad (8)$$

$$\texttt{alldifferent}(l_{1 \leq i \leq m}) \quad (9)$$

$$l_i > l_j \rightarrow s_i \geq s_j \qquad 1 \leq i, j \leq m \quad (10)$$

Here (8) defines the span for each machine to be the latest completion time of any job scheduled on it, while (9-10) ensure that each machine is assigned a different level and the levels order the machines with respect to their spans.

The objective function for minimizing the lex-makespan can then be expressed as

$$min\ \textstyle\sum_{i=1}^{m} h^{l_i - 1} \cdot s_i.$$

Intuitively, the levels represent the priorities for optimization. By assigning the span of machine i the weight $h^{l_i - 1}$, it is more important than all spans of machines on lower levels.

Note that restricting the model to constraints (1–7) and using the objective $min\ max_{1 \leq i \leq n}(c_i)$ expresses the scheduling problem with the standard makespan objective.

6.3 Systems

We use `clingo-dl` version (1.1.1) for $solve(\cdot)$ and $opt(\cdot)$ in Algs. 1 and 2, respectively. For both algorithms, the time limit for optimising any level of the lex-makespan can be set to a geometric sequence with ratio 0.5. Thus half of the total time limit is spent on optimising the highest priority level, a quarter on the next level etc.

We use a FlatZinc linearisation of the MiniZinc model to compare with four MIP/CP solvers (`cplex` 12.10[5], `cpoptimizer` 20.1[6], `gecode` 6.3.0[7] and `or-tools` 7.8[8]) against the hybrid ASP approach. All solvers could be used for makespan minimization. Regarding the lex-makespan, `gecode` could not produce any solutions due to numerical issues and both `cplex` and `cpoptimizer` wrongly reported optimality for some solutions. This is a technical issue that is probably due to the translation of MiniZinc model to FlatZinc or too high values for the lex-makespan objective function. Due to this, we also encoded the problem and both objectives in the native modelling languages of `cplex` and `cpoptimizer`.[9] Those direct approaches had no problems with wrongly reported optimal solutions and their results are included below. In difference, `or-tools` had no issues with the MiniZinc model and was thus run using this model.

6.4 Experimental Results and Discussion

All experiments were conducted on a cluster with 13 nodes, where each node has two Intel Xeon CPUs E5-2650 v4 (max. 2.90GHz, 12 physical cores, no hyperthreading), and 256GB RAM. For each run, we set a memory limit of 20GB and all solvers only used one solving thread. The application at the production site requires schedule computation within 300 seconds, which we adopted as time limit. We also did experiments with run times of up to 15 minutes, the outcome is however very similar and results are left out.

Figure 3 gives an overview of the performance of solvers on the entire set of instances, where we compare the approaches for makespan optimisation and for lex-makespan. For each solver, we report the number of instances where some solution was found (*feasible*), a minimal solution among all approaches was found (*best*), and a globally minimal solution was found (*optimal*).

The makespan comparisons provide an important baseline as every approach that aims at improving lex-makespan necessarily involves makespan minimisation, and any good lex-makespan optimisers also needs to produce small makespans. The `clingo-dl` approach finds solutions for most of the instances and shows very good performance for the number of best and optimal solutions compared to `cplex`, `cpoptimizer`, `or-tools` and `gecode`. At least when using our MiniZinc model, `or-tools` and `gecode` have difficulties to find solutions for a large proportion of the instances. The performance of `cplex` and `cpoptimizer` is better, but it is still behind `clingo-dl`. In should be noted that we did not investigate further improvements for those solvers and `cpoptimizer` finds more best solutions than `clingo-dl` with the longer 15 minute timeout. However, the results show that ASP is indeed a promising approach for this problem.

For the lex-makespan comparisons, we use `or-tools`, `cplex`, `cpoptimizer` as well as `clingo-dl`, the `clingo-dl` approximation described in Section 5 with

[5] https://www.ibm.com/analytics/cplex-optimizer.

[6] https://www.ibm.com/analytics/cplex-optimizer.

[7] https://www.gecode.org.

[8] https://developers.google.com/optimization.

[9] The encodings are available online at https://owncloud.tuwien.ac.at/index.php/s/UmD0lIh2B7B4A9g.

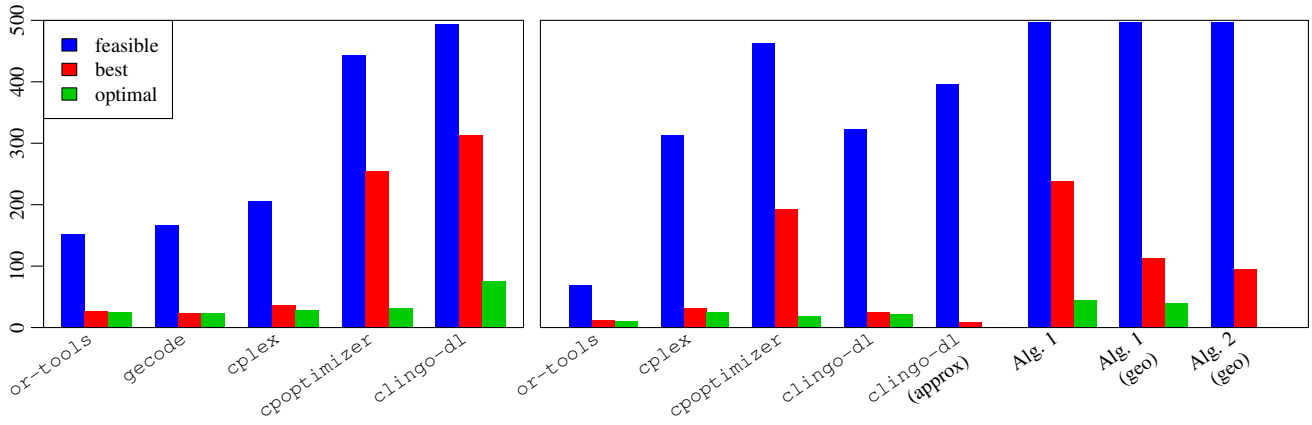

Figure 3: Different systems on all instances for makespan (left) and lex-makespan (right).

granularity $g = 10$, and Algs. 1 and 2. While `or-tools`, `cplex` and `clingo-dl` struggle now to find feasible solutions, our multi-shot approaches shine in comparison with `cpoptimizer` trailing closely behind. The `clingo-dl` approximation does find more feasible solutions than normal `clingo-dl`. However, it finds the least number of best solution when compared to the others. While the approximation does indeed improve performance of `clingo-dl` on bigger instances, it performs worse on small instances and the results for the bigger instances are dwarfed by the other approaches. If Alg. 1 is used without geometric timeouts, it reports optimal solutions for quite a number of instances and finds the most best solutions. This, when comparing with Algs. 1 and 2 with geometric timeouts, is not surprising as more time, for many instances all the time, is spent on minimising the first component of the lex-makespan.

Figure 3 does not tell us much about the quality differences of the schedules produced by the different algorithms. As our informal objective is that machines complete as early as possible, we show in Fig. 4 graphs for the ratio of machines that completed as a function of schedule time, i.e., $f(t) = M(S,t)/m$. This allows us to compare different approaches on the same instance classes, where the x-axis is schedule time in seconds and the y-axis is the average of $M(S,t)/m$ over the instances. The curves for different algorithms can be interpreted as follows: the earlier a curve reaches 1 (all machines completed), the smaller is the average makespan of the instances. The shape of the curve reveals details about the quality of the schedule prior to this point. For our informal objective, a steep incline of this curve is desired—the earlier it gets ahead and stays ahead, the better.

We only consider Algs. 1 and 2 with geometric timeouts against plain makespan optimisation with `clingo-dl` in Fig. 4. We show instance classes of increasing size from left to right and compare instances of type "low dedication" in the upper row and "high dedication" in the lower row. All approaches produce small makespans and thus schedules with a high throughput. This is worth emphasising since only half of the time limit is used here for makespan optimisation by Algs. 1 and 2. However, when comparing the shape of the curves, the lex-makespan optmisers show their strengths for meeting our informal objective; the difference to makespan is subtle for Alg. 1 but more pronounced for Alg. 2. While Alg. 2 finds fewer minimal solutions than Alg. 1 according to Fig. 3, it tends to get ahead the earliest in terms of completed machines when considering the execution of the schedules, especially for high dedication instances. We can quantify this by looking at the average ratio of machines finished at each point in time. On average, Alg 1. improves this metric by 6.3% whereas Alg. 2 shows an improvement of 9.53%.

7 Related Work

Many variants of Parallel Machine Scheduling Problem, e.g., (Allahverdi et al. 2008; Allahverdi 2015), have been studied extensively in the literature. Previous publications have considered eligibility of machines, e.g., (Afzalirad and Rezaeian 2016; Perez-Gonzalez et al. 2019; Bektur and Saraç 2019), machine dependent processing time, e.g., (Vallada and Ruiz 2011b; Avalos-Rosales, Alvarez, and Ángel-Bello 2013; Allahverdi 2015), and sequence dependent setup times, e.g., (Vallada and Ruiz 2011b; Perez-Gonzalez et al. 2019; Fanjul-Peyro, Ruiz, and Perea 2019; Gedik et al. 2018).

The idea to use lexicographical makespan optimisation to obtain robust schedules for identical parallel machines comes from Letsios, Mistry, and Misener (2021) but has not been used, to the best of our knowledge, when setup times are present. The general idea of optimising not only the element that causes the highest costs but also the second one and so on to obtain robustness, fairness, or balancedness is studied under the notion of min-max optimisation for a various combinatorial problems (Burkard and Rendl 1991; Ogryczak and Śliwiński 2006).

There are several related objective functions that can be used to achieve similar effects as minimising the lex-makespan. Load balancing can be used to obtain balanced resource utilisation by equalising the workload on machines (Rajakumar, Arunachalam, and Selladurai 2004; Yildirim et al. 2007; Sabuncu and Simsek 2020). One measure for this is to minimise the relative percentage of imbalance in workload (Rajakumar, Arunachalam, and Selladurai 2004) which is determined based on the difference between

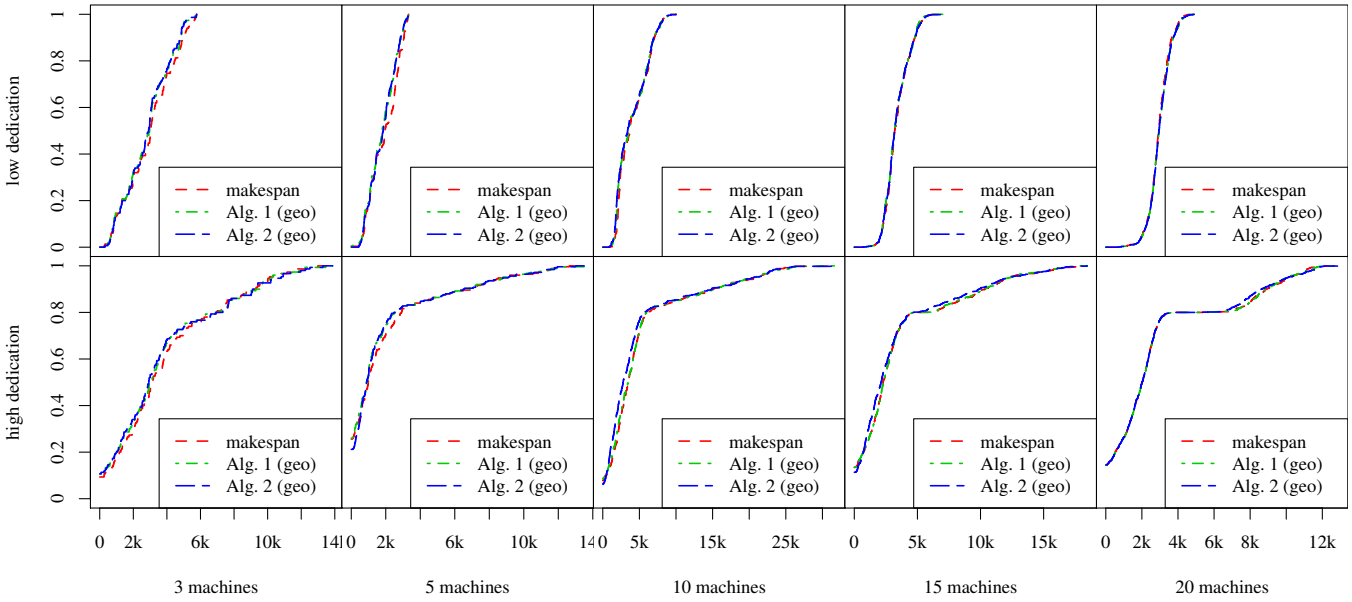

Figure 4: Machine completion rate over time for makespan and lex-makespan.

a machine span and the makespan. Other approaches try to minimise the difference between the largest and the smallest machine span (Ouazene et al. 2014). Note that the machines m_2 and m_3 in Fig. 1 (a) are balanced under this notion, but this does not ensure that the machines finish as early as they could. Sabuncu and Simsek (2020) provide a novel formulation of a machine scheduling problem in ASP. Their approach bears some similarity to ours, but their objective is to balance the workload of the given machines. In general, load balancing can lead to schedules where, e.g., longer than necessary setup times are used to artificially prolong machine spans for reducing imbalances. Another idea is to minimise workload instead of balancing it. While this achieves short processing times, it does not ensure that all machines finish as early as possible either. Similar to the workload, minimising the sum of machine spans does not prevent that jobs are scheduled in an unbalanced way; Figs. 1 (b) and (c) serve as an example of two schedules with the same total machine span. Minimising a non-linear sum of machine spans like their squares comes close to our informal objective but is different from the lex-makespan as it does not guarantee a minimal makespan (Walter 2017). Another way to obtain compact schedules is it to minimise the total completion times (Weng, Lu, and Ren 2001). This however can pull short jobs to the front of the schedule, which can adversely interfere with avoiding large setup times.

Extending ASP with difference logic is just one way to blend integer constraints and ASP and there are several other approaches (Lierler 2014; Gebser, Ostrowski, and Schaub 2009). We evaluated ASP with full integer constraints with `clingcon`, but performance on our problem was poor. The fast propagation enabled by the lower computational complexity of difference logic seems a big advantage here. The `clingo-dl` system has indeed been used for the related problem of job-shop scheduling and makespan optimisation (Janhunen et al. 2017; El-Kholany and Gebser 2020). Train scheduling for the Swiss Federal Railways is another application of `clingo-dl` that involves routing, scheduling, and complex optimisation (Abels et al. 2019). However, we are solving a different problem with a more complex objective function and also compare to other solving paradigms.

8 Conclusion

We studied the application of hybrid ASP with difference logic to solve a challenging parallel machine scheduling problem with setup-times in industrial semi-conductor production at Bosch. As objective function, we used the lex-makespan which generalises the canonical makespan to a tuple of machine spans and aims at accomplishing short completion times for all machines. Semi-conductor production involves not only one but several connected work centers that solve similar problems. Having a flexible ASP solution for one that can easily be adapted to others is highly desirable. To make ASP perform up to par, we appropriated advanced techniques like difference constraints, multi-shot solving, domain heuristics, and approximations for our application.

For the experimental evaluation, we considered random instances of realistic size and structure. We further implemented a solver-independent MiniZinc model as well as direct encodings that we used for comparisons with MIP and CP solvers. The results show that the objective of short completion times for machines is well achieved. Performance is improved by using approximations without significant deterioration of the schedules produced. It is encouraging that the ASP approaches turn out to be competitive with commercial MIP and CP solvers which are, at least to some extend, engineered for industrial scheduling problems.

We plan to study meta-heuristics for lex-makespan optimisation in combination with ASP and methods to combine the lex-makespan with other common objective functions.

Acknowledgments

We would like to thank Michel Janus, Andrej Gisbrecht, and Sebastian Bayer for very helpful discussions on the scheduling problems at Bosch, as well as Roland Kaminski, Max Ostrowski, Torsten Schaub, and Philipp Wanko for their help, valuable suggestions, and feedback on constraint ASP. Johannes Oetsch was supported by funding from the Bosch Center for Artificial Intelligence.

References

Abels, D.; Jordi, J.; Ostrowski, M.; Schaub, T.; Toletti, A.; and Wanko, P. 2019. Train Scheduling with Hybrid ASP. In *Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019)*, volume 11481 of *Lecture Notes in Computer Science*, 3–17. Springer.

Afzalirad, M., and Rezaeian, J. 2016. Resource-constrained unrelated parallel machine scheduling problem with sequence dependent setup times, precedence constraints and machine eligibility restrictions. *Computers and Industrial Engineering* 98:40–52.

Allahverdi, A.; Ng, C. T.; Cheng, T. C. E.; and Kovalyov, M. Y. 2008. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research* 187(3):985–1032.

Allahverdi, A. 2015. The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research* 246(2):345–378.

Avalos-Rosales, O.; Alvarez, A. M.; and Ángel-Bello, F. 2013. A Reformulation for the Problem of Scheduling Unrelated Parallel Machines with Sequence and Machine Dependent Setup Times. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 278–282.

Bektur, G., and Saraç, T. 2019. A mathematical model and heuristic algorithms for an unrelated parallel machine scheduling problem with sequence-dependent setup times, machine eligibility restrictions and a common server. *Comput. Oper. Res.* 103:46–63.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer Set Programming at a Glance. *Communications of the ACM* 54(12):92–103.

Burkard, R. E., and Rendl, F. 1991. Lexicographic bottleneck problems. *Operations Research Letters* 10(5):303–308.

El-Kholany, M., and Gebser, M. 2020. Job Shop Scheduling with Multi-shot ASP. In *Proceedings of the 4th Workshop on Trends and Applications of Answer Set Programming (TAASP 2020)*.

Fanjul-Peyro, L.; Ruiz, R.; and Perea, F. 2019. Reformulations and an exact algorithm for unrelated parallel machine scheduling problems with setup times. *Computers & Operations Research* 101:173–182.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. Answer Set Solving in Practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 6(3):1–238.

Gebser, M.; Kaufmann, B.; Romero, J.; Otero, R.; Schaub, T.; and Wanko, P. 2013. Domain-specific heuristics in answer set programming. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013)*, 350–356. AAAI Press.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory Solving Made Easy with Clingo 5. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*, volume 52 of *OASIcs*, 2:1–2:15. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

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.

Gebser, M.; Ostrowski, M.; and Schaub, T. 2009. Constraint Answer Set Solving. In *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *Lecture Notes in Computer Science*, 235–249. Springer.

Gedik, R.; Kalathia, D.; Egilmez, G.; and Kirac, E. 2018. A constraint programming approach for solving unrelated parallel machine scheduling problem. *Computers & Industrial Engineering* 121:139–149.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New generation computing* 9(3-4):365–385.

Janhunen, T.; Kaminski, R.; Ostrowski, M.; Schellhorn, S.; Wanko, P.; and Schaub, T. 2017. clingo goes Linear Constraints over Reals and Integers. *Theory and Practice of Logic Programming* 17(5-6):872–888.

Letsios, D.; Mistry, M.; and Misener, R. 2021. Exact Lexicographic Scheduling and Approximate Rescheduling. *European Journal of Operational Research* 290(2):469–478.

Lierler, Y. 2014. Relating Constraint Answer Set Programming Languages and Algorithms. *Artificial Intelligence* 207:1–22.

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

McCarthy, J. 1998. Elaboration Tolerance. In *Common Sense*, volume 98.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, 529–543. Springer.

Ogryczak, W., and Śliwiński, T. 2006. On Direct Methods for Lexicographic Min-Max Optimization. In *Proceedings of the 6th International Conference on Computational Science and Its Applications (ICCSA 2006)*, volume 3982 of *Lecture Notes in Computer Science*, 802–811. Springer.

Ouazene, Y.; Yalaoui, F.; Chehade, H.; and Yalaoui, A. 2014. Workload balancing in identical parallel machine scheduling using a mathematical programming method. *International Journal of Computational Intelligence Systems* 7(sup1):58–67.

Perez-Gonzalez, P.; Fernandez-Viagas, V.; García, M. Z.; and Framiñan, J. M. 2019. Constructive heuristics for the

unrelated parallel machines scheduling problem with machine eligibility and setup times. *Computers and Industrial Engineering* 131:131–145.

Rajakumar, S.; Arunachalam, V.; and Selladurai, V. 2004. Workflow balancing strategies in parallel machine scheduling. *The International Journal of Advanced Manufacturing Technology* 23(5-6):366–374.

Sabuncu, O., and Simsek, M. C. 2020. Solving assembly line workload smoothing problem via answer set programming. In *Proceedings of the 13th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2020)*, volume 2678 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Vallada, E., and Ruiz, R. 2011a. A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. *European Journal of Operational Research* 211(3):612–622.

Vallada, E., and Ruiz, R. 2011b. A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. *European Journal of Operational Research* 211(3):612–622.

Walter, R. 2017. A note on minimizing the sum of squares of machine completion times on two identical parallel machines. *Central European Journal of Operations Research* 25(1):139–144.

Weng, M. X.; Lu, J.; and Ren, H. 2001. Unrelated parallel machine scheduling with setup consideration and a total weighted completion time objective. *International Journal of Production Economics* 70(3):215–226.

Yildirim, M. B.; Duman, E.; Krishnan, K. K.; and Senniappan, K. 2007. Parallel machine scheduling with load balancing and sequence dependent setups. *International Journal of Operations Research* 4(1):42–49.