Multi-Machine Scheduling Lower Bounds Using Decision Diagrams[☆]

Pim van den Bogaerdt, Mathijs de Weerdt[*]

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Abstract

Relaxed decision diagrams can be used to obtain lower bounds on multi-machine scheduling problems. We consider the problem where jobs have deadlines, release times, and where a partition of jobs is given such that jobs in the same partition may have precedence relations and cannot overlap. Our formulation greatly improves upon a naive extension of the state-of-the-art for single-machine scheduling, being able to give decent lower bounds in short time. Our implementation is competitive with lower bounds of existing solvers.

Declaration of interest: When this work was carried out, the first author had a paid internship, and the second author was a visiting scientist at the Dutch Railways (NS).
Role of funding source: The NS had no involvement in the conduct of research, this article, or the decision to submit for publication.

Keywords: multi-machine scheduling, lower bounds, decision diagrams
2010 MSC: 90B35

1. Introduction

We consider the problem of finding lower bounds for multi-machine scheduling instances. We are given n *jobs* that are to be scheduled without preemp-

[☆]This work is based on the first author's Master's thesis [1].

[*]Corresponding author.
Email address: M.M.deWeerdt@tudelft.nl (Mathijs de Weerdt)

tion on $m \leq n$ *machines*. Let $\mathcal{J}$ denote the set of all jobs. Each job $J \in \mathcal{J}$ has a *processing time* $p(J) > 0$, a *release time* $r(J) \geq 0$, and a *due time* $d(J)$.

We assume the jobs are partitioned into k subsets (partitions), and that the jobs in a given subset may not overlap ("partition constraints"). Let $\mathcal{P}$ be the set of partitions. We allow each partition $i \in \mathcal{P}$ to have a *release time* $r(i)$, meaning its jobs cannot start before that time. Let $P(J)$ denote the partition of job J. We also allow *precedence constraints* between jobs of the same partition. The precedence constraint $J_i \to J_j$ specifies that J_i has to end before J_j starts. Let $\omega(J) = \{J' \mid J' \to J\}$ be the set of jobs that have to be completed before J can start. Partition and precedence constraints model a service site for trains, but we believe our model is sufficiently general to have wider applicability.

A *schedule* assigns each job a starting time and machine such that each machine processes at most one job at a time. In a given schedule S, a job J has *starting time* $s^S(J) \geq r(J)$, *completion time* $c^S(J) = s^S(J) + p(J)$, and $M^S(J)$ denotes the machine J is scheduled on. The *processing time of a machine* M is $c^S(M) = \max_{J \text{ on } M} c^S(J)$. We define an ordering $\prec_S$ on the machines, where $M_i \prec_S M_j$ if either $c^S(M_i) < c^S(M_j)$ or $c^S(M_i) = c^S(M_j) \wedge M_i < M_j$. We omit the superscript if schedule S is clear from the context.

The objective function defines the cost of a schedule. We assume objective functions are non-decreasing in the job start times. This is a reasonable assumption: we typically want jobs to complete as soon as possible. We also assume the objective function can be written as a sum $z(S) = \sum_J z_J(s^S(J))$. For example, the *total tardiness* is defined by $T(S) = \sum_J \max\{0, s^S(J) + p(J) - d(J)\}$.

Previous results. MIPs for several multi-machine scheduling problems exist (e.g. [2]), of which the relaxations provide lower bounds on the optimum. Baptiste et al. [3] focuses on lower bounds for multi-machine scheduling by giving MIPs and a reduction to a flow problem, among other approaches. For single-machine scheduling, very good lower bounds can be found using decision diagrams [4, 5]. In spite of this promising result, there is no previous work on the use of decision diagrams for finding lower bounds for multi-machine scheduling. Finding lower bounds enables one to appreciate the quality of a schedule, and save time by not searching for a better schedule if the lower bound is equal to the optimum.

Our results. We present and prove correctness of a decision diagram (DD) formulation for the multi-machine scheduling problem which greatly improves

upon a naive extension of the state of the art for single-machine scheduling [4]. We furthermore show that, when embedded in a solver, this provides lower bounds competitive with existing solvers.

2. Background on DDs

First we summarize the concept of a DD for a minimization problem P involving variables $x_1, \ldots, x_n$ with finite and equal domain (see, e.g., [6]). Many problems fall in this category, including the scheduling problem we consider.

A *decision diagram* for an instance I of problem P is a directed graph with the following properties. The nodes are partitioned into $n+1$ *layers* $L_1, \ldots, L_{n+1}$, and each node v contains a *state* $S(v)$. For a node v in layer L_i, the choices to x_i we can make in state $S(v)$ are defined by the set of *feasible choices* $F_i(S(v))$, and each choice $x_i \in F_i(S(v))$ induces an outgoing edge of v with cost $c_i(S(v), x_i)$ to a node in layer L_{i+1} with state $\varphi_i(S(v), x_i)$. Layer L_1 (resp. L_{n+1}) consists of a single *root node* r (resp. *leaf node* t).

A root-leaf path corresponds to an assignment to $x_1, \ldots, x_n$. The definitions of S, F_i, c_i, φ_i are problem-specific and also depend on the type of DD. We define two types of DDs.

An *exact DD* is a DD such that the cost of a root-leaf path equals the objective value for the corresponding assignment, and the set of root-leaf paths corresponds exactly to the set of feasible solutions. The shortest root-leaf path in an exact DD corresponds with an optimal solution of I.

A problem with exact DDs is their exponential size, and we therefore consider another type of DD: relaxed DDs. These provide *lower bounds* on the optimum. Given an integer $W \geq 1$, a *relaxed DD of width* $\leq W$ is a DD with the following additional properties. The cost of a root-leaf path is *at most as high* as the objective value for the corresponding assignment to $x_1, \ldots, x_n$; the set of root-leaf paths corresponds to a set of solutions that is a *superset* of the feasible solutions; and each layer contains at most W nodes.

A relaxed DD can be built layer-by-layer. After building each layer, multiple nodes can be *merged* into one to prevent exceeding the allowed width W. This algorithm is called *top-down compilation*. We define the merged nodes by means of a merge operator $\oplus$ that transforms two states into one merged state. We assume the operator is associative, so that we can unambiguously allow it to have multiple states as input.

A theorem by Hooker [4, Theorem 1] describes when a merge operator indeed yields a relaxed DD (called *validity*). The theorem is fundamental for proving the correctness of our DD formulation (in section 3). We present this theorem in a slightly more formal way, using the following definition of a state relaxation relation.

Definition 1. *A* state relaxation relation *is a binary operator $\preceq$ that operates on states with the following properties:*

- *(R1) $\preceq$ is reflexive and transitive.*
- *(R2) If $T \preceq S$, then $F_i(T) \supseteq F_i(S)$.*
- *(R3) If $T \preceq S$, then for $j \in F_i(S), c_i(T, j) \leq c_i(S, j)$.*
- *(R4) If $T \preceq S$, then for $j \in F_i(S)$, $\varphi_i(T, j) \preceq \varphi_i(S, j)$.*

Theorem 1 (Hooker [4])**.** *Let a state relaxation relation $\preceq$ and merge operator $\oplus$ be given. Then the merge operator is valid if $S \oplus T \preceq S, T$ for all states S, T.*

To compute the shortest root-leaf path in a DD, we keep track of the partial objective in each node. We can use this value also as a heuristic to choose what nodes to merge [4]: it is reasonable to merge nodes with a high partial objective. Such nodes are not likely to lie on the shortest path. Thus, merging these nodes loses information that likely does not make the shortest root-leaf path smaller.

3. Multi-Machine Scheduling Using DDs

Our DD formulation for multi-machine scheduling uses an efficient search space, state description, and merge operator. We explain these elements and prove their correctness.

3.1. Search Space

We may assume no machine has unnecessary idle time because the objective function is non-decreasing in the job start times. Hence, it is sufficient to describe a schedule by a machine assignment for each job, and a job ordering per machine.

We use the concept of *minimal* schedules. Roughly speaking, a schedule is minimal if the completion times of the machines are approximately equal.

It turns out each minimal schedule can be represented as a permutation of the jobs, and at least one optimal schedule is minimal. Thus, it is sufficient to consider minimal schedules if we are to compute (a lower bound on) the optimum.

The concept of using permutations to represent minimal schedules is not new [7, 8, 9, 10]. We show that the permutation representation also works in our problem with the partition and precedence constraints, and we propose a relaxed DD formulation for this representation. This representation is promising because we only need to consider $n!$ schedules. We show that a reduction in search space [10 cited by 11] also applies to our search type, and extend it to the relaxation.

To show that permutations are sufficient, we define two types of *neighbours*, and we define when a schedule is minimal.

Definition 2. *Given a schedule S. Suppose there exists a job J and a machine M such that schedule S', created as follows, is feasible. Then S' is a* type 1 neighbour *of S.*

- *If $M(J) \neq M$, swap the machine assignments for the jobs with starting time $\geq s(J)$ on M and $M(J)$.*
- *Decrease the starting time of J (now on M) by one.*

Definition 3. *Given a schedule S. Suppose there exists a job J and a machine $M < M(J)$ such that M is idle on the interval $(s(J), s(J)+1)$. Let M be the smallest such machine. Create schedule S' by swapping the machine assignments for the jobs with starting time $\geq s(J)$ on M and $M(J)$. If M exists and S' is feasible, then S' is a* type 2 neighbour *of S.*

Definition 4. *A schedule S is* minimal *if it has no defined neighbours.*

We first show considering minimal schedules is sufficient.

Lemma 1. *There is an optimal schedule which is minimal.*

Proof. Consider some schedule S. We first show that each neighbour of S has at most as high objective value as S. We then show that the following algorithm for transforming an optimal schedule to a minimal schedule terminates: apply type 1 to jobs as long as possible; in the order of starting times, apply type 2 to jobs as long as possible; repeat if the schedule is not yet

minimal. Together, these claims prove the lemma, since they show we can transform any optimal schedule into a minimal one without losing optimality.

First, we show that applying a neighbour operator does not increase the objective value. Consider a type 2 neighbour S' of S. No starting time has changed between S and S'. Consequently, S' has equal objective value as S. For a type 1 neighbour S'', the only extra step compared to type 1 is that some job J gets rescheduled to an earlier starting time. Consequently, S'' has *at most as high* objective value as S.

Next, we show the above algorithm terminates. To see that we can only apply a type 1 neighbour a finite number of times, note that each application schedules a job J to a strictly smaller (integer) starting time.

Furthermore, we can apply type 2 at most n times per iteration. After applying type 2 to a job, this job does not have a type 2 neighbour any more, since J got moved to the machine that satisfied the type 2 predicate *with the smallest index*. When we apply type 2 to another machine with the same starting time as J, this continues to hold. When we apply type 2 to a machine with a later starting time than that of J, the part of the schedule up to and including J does not change. We conclude that we can apply type 2 at most once for each job (if we apply it in order of starting times). □

Note that in the above proof, any precedence and partition constraints are dealt with through the definition of feasible schedules.

We next show that each minimal schedule can be represented as a permutation of the jobs for our problem type. The mapping from permutations to schedules is given by the Smallest Processing Time First algorithm (SPTF); see Algorithm 1. SPTF schedules each job on the machine with smallest processing time so far. The vector T represents when a job can start: T_i is the finishing time of the last job of partition i. Thus, T_i can be interpreted as a "dynamic release time": as we schedule a job of partition i, we use it as its release time, and change it to its completion time.

SPTF does not take into account precedence constraints, but if for each constraint $J_i \to J_j$, job J_i occurs before J_j in the permutation, then SPTF gives a feasible schedule.

We show that, in fact, each minimal schedule for our problem type can be represented as a permutation with non-decreasing start times, like [10 cited by 11].

Lemma 2. *For each minimal schedule, there is a permutation for which SPTF generates it and for which the start times are non-decreasing.*

Algorithm 1 Smallest Processing Time First (SPTF).

Input: Permutation π of $\mathcal{J}$.
Output: Schedule S induced by π.
$S \leftarrow$ empty schedule
$T \leftarrow (r(1), \ldots, r(k))$
for $i = 1, \ldots, n$ **do**
  $M \leftarrow$ smallest machine according to $\prec_S$
  Update S by scheduling job π_i on machine M
  $T_{P(\pi_i)} \leftarrow s(\pi_i) + p(\pi_i)$

Proof. Consider a minimal schedule S, and create permutation π (initially empty) iteratively as follows. Consider the first machine M with minimal finishing time when only considering jobs already in π (breaking ties by smallest index). Add the first job on M (that is not yet in π) to π. Repeat.

Define $S' = \text{SPTF}(\pi)$. We claim that $S' = S$. Suppose for contradiction that $S' \neq S$. Let i be as small as possible such that job J in S', at index i in π, has a different starting time, or is scheduled on a different machine (with the same starting time). Consider the partial schedule S'' consisting of the jobs scheduled by SPTF before index (iteration) i.

First consider the case that J has a different starting time in S and S'. At iteration i, SPTF adds J to S'' as soon as possible on the machine with smallest processing time (in S''). Thus, J is scheduled *later* in S than in S'. Consequently, machine $M^S(J)$ is idle on S in the interval $(s^S(J) - 1, s^S(J))$. But then S has the type 1 neighbour: contradiction.

Now consider the case that J is scheduled on machine M in S and M' in S' with $M \neq M'$, and $s^S(J) = s^{S'}(J)$. At iteration i, SPTF schedules J on the machine *with the smallest index* among the machines with smallest processing time (in S''). Consequently, machine $M' < M$ is idle on S in the interval $(s^S(J), s^S(J)+1)$. But then S has a type 2 neighbour: contradiction.

We conclude that S must be equal to S', and π is a permutation that generates S. It remains to show that π has non-decreasing start times. Suppose for contradiction that two adjacent jobs in π, say J_{π_i} and $J_{\pi_{i+1}}$, have $s^{S'}(J_{\pi_i}) > s^{S'}(J_{\pi_{i+1}})$. By the construction of π, these jobs are scheduled on different machines. When J_{π_i} was added to π, machine $M^{S'}(J_{\pi_i})$ had finishing time no larger than $M^{S'}(J_{\pi_{i+1}})$ (in S). In particular, machine $M^{S'}(J_{\pi_i})$ was idle at time $s^{S'}(J_{\pi_{i+1}})$ (in S). But this means S has a type 1 neighbour,

so that it is not minimal: contradiction. □

Combining this result with Lemma 1, we may conclude.

Theorem 2. *It is sufficient to consider schedules based on SPTF when computing (a lower bound on) the optimum. Furthermore, it is sufficient to consider permutations with non-decreasing start times.*

3.2. Exact DD Formulation

Our exact DD formulation for multi-machine scheduling represents precisely all permutations that we need to consider as per the discussion in the previous section. To ensure this, we define each node to contain several variables as state, which we next introduce gradually.

As a first step, we let the state of a node v be a pair (V, F). Such a state gives us sufficient information to build an exact DD, but without the partition and precedence constraints. Like [4], we define V as the set of jobs that appear on the path from the root to v, representing the schedule built so far. However, instead of a single number f, we now have a *vector* $F = (F_1, \ldots, F_m)$ of length m that contains the processing times *of each machine*. Since we consider identical machines, the ordering of elements in F is irrelevant. It is useful to assume that F is sorted in increasing order. For a vector x, let $\text{sort}(x)$ denote the sorted version of x (in increasing order).

If we are in a node v on layer i with state $S(v) = (V, F)$, the set of feasible choices is $F_i(S(v)) = \mathcal{J} - V$. In words, we do not consider jobs in V because such a permutation would contain that job at least twice, which is infeasible. When we choose job J as the next item in the permutation, the new state is $\varphi_i(S(v), J) = (\overline{V}, \overline{F})$ where $\overline{V} = V \cup \{J\}$ and $\overline{F}$ is a new vector that represents scheduling J at the first machine in F (like SPTF: recall that F is sorted). More precisely, $\overline{F} = \text{sort}(\,(F_1 + p(J), F_2, \ldots, F_m)\,)$. The edge cost of choosing J in this state is the objective increase that would result from scheduling J according to F, that is, at time F_1: $c_i(S(v), J) = z_J(F_1)$.

To take the partition constraints into account, we extend the state with a k-vector T, which has a similar interpretation as in Algorithm 1: it represents when a job can start because the previous jobs of the partition have completed. The root has $T_i = r(i)$ for $1 \leq i \leq k$, which are the release times of partitions. If we choose job J, resulting in completion time x, the new state $\overline{T}$ is updated with $\overline{T_{P(J)}} = x$. For $1 \leq j \leq k, j \neq P(J)$, we set $\overline{T_j} = T_j$.

In words, as we schedule job J, other jobs of its partition can only start after job J ends. We modify the transition for F: job J can only start at time $T_{P(J)}$. That is, $\overline{F} = \text{sort}(\ (\max\{F_1, T_{P(J)}\} + p(J), F_2, \ldots, F_m)\ )$. Similarly, the cost is now computed as $c_i(S(v), J) = z_J(\max\{F_1, T_{P(J)}\})$.

To incorporate precedence constraints, we extend the state of a node v with a set U (similar to [5]). The set U represents which jobs occur on *some* path from the root to v, as opposed to V, which contains jobs that occur on *every* path from the root to v. In an exact DD, U is equal to V. We make a distinction between these sets when we create a relaxation in the next subsection.

A precedence constraint may remove a job from the feasible set in v. Suppose v is a node on layer i, and consider a precedence constraint $J' \to J$. If $J' \notin U$, then we have not scheduled J' yet, and so we cannot schedule J at this point. Hence, we may remove J' from $F_i(S(v))$. In general, we remove a job J from $F_i(S(v))$ if $\omega(J) \not\subseteq U$.

The partition constraints already ensure that if we schedule a job J, it starts after any job $J' \in \omega(J)$ has completed: recall that we only allow precedence constraints between jobs of the same partition.

When we make a job choice J, we add J to the set U in the new node, just like we do for V. Each root-leaf path thus induces a permutation for which J_i occurs before J_j if $J_i \to J_j$. The set of permutations represented by the DD therefore only includes feasible solutions.

The last step of our exact DD formulation is related to only considering permutations with non-decreasing start times. We extend the state of a node v with a m-vector F^u, a k-vector T^u and a number g. The vectors F^u and T^u again represent the finishing times of the machines and partitions, respectively, and g represents the starting time of the incoming job. In our exact DD formulation, $F^u = F$ and $T^u = T$. Again, in the next section we make the distinction between these vectors and explain the meaning of the superscript u.

If we were to schedule a job J at time $\max\{F_1^u, T_i^u\} < g$, then we may ignore this job choice: by Theorem 2, it is sufficient to consider permutations with non-decreasing start times, but any permutation in the subtree for job choice J would violate this predicate. When making a job choice, we set $g = \max\{F_1, T_{P(J)}\}$, which is the time J starts.

The transition for F^u uses these vectors as input also: $\overline{F^u} = \text{sort}(\ (\max\{F_1^u, T_{P(J)}^u\} +$

Figure 1: Exact DD formulation. v is a node on layer L_i with state $S(v) = (V, U, F, T, F^u, T^u, g)$.

$$
\begin{aligned}
S(r) &= (\varnothing, \varnothing, \vec{0}, (r(1), \ldots, r(k)), \vec{0}, (r(1), \ldots, r(k)), 0) \\
F_i(S(v)) &= \mathcal{J} - (V \cup \{J \mid \omega(J) \not\subseteq U\} \\
&\qquad\qquad \cup \{J \mid \max\{F_1^u, T_{P(J)}^u\} < g\}) \\
c_i(S(v), J) &= z_J(\max\{F_1, T_{P(J)}\}) \\
\varphi_i(S(v), J) &= (V \cup \{J\}, U \cup \{J\}, \overline{F}, \overline{T}, \overline{F^u}, \overline{T^u}, \\
&\qquad \max\{F_1, T_{P(J)}\}), \text{ where} \\
\overline{F} &= \text{sort}(\, (\overline{T_{P(J)}}, F_2, \ldots, F_m)\,) \\
\overline{T_j} &= \begin{cases} \max\{F_1, T_j\} + p(J) & \text{if } j = P(J) \\ T_j & \text{otherwise} \end{cases} \\
\overline{F^u} &= \text{sort}(\, (\overline{T^u_{P(J)}}, F_2^u, \ldots, F_m^u)\,) \\
\overline{T_j^u} &= \begin{cases} \max\{F_1^u, T_j^u\} + p(J) & \text{if } j = P(J) \\ T_j^u & \text{otherwise} \end{cases}
\end{aligned}
$$

$p(J), F_2^u, \ldots, F_m^u)\,)$. Similarly, we define $\overline{T_i^u} = \max\{F_1^u, T_i^u\} + p(J)$ if $i = P(J)$ and $\overline{T_i^u} = T_i^u$ otherwise. Our exact DD formulation including all of the above is given in Figure 1.

3.3. Relaxed DD Formulation

Recall that our state is a tuple $(V, U, F, T, F^u, T^u, g)$. Our relaxed DD formulation is defined in terms of a state relaxation relation $\preceq$ and merge operator $\oplus$ that operate on a pair of states. Given two states, the relation $\preceq$ holds if each of the following relations hold pairwise in the order of the state: $\subseteq, \supseteq, \leq, \leq, \geq, \geq, \leq$. (On vectors, $\leq$ and $\geq$ denote pairwise comparisons.) The operator $\oplus$ applies the following operators in the same order: $\cap, \cup, \min, \min, \max, \max, \min$.

Here we have a difference between F and F^u, and similarly between T and T^u. The u stands for "upper bound": in a relaxed state S' of S, $F^{u\prime}$ and $T^{u\prime}$ are upper bounds compared to F^u and T^u in S. In contrast, F' and T' are lower bounds compared to F and T. There is now also a difference between V and U. To show that $\preceq$ satisfies Definition 1 of being a valid state relaxation relation, we use the following lemma.

Lemma 3. *Let $A \leq B$ be sorted vectors of length m, and let $\widetilde{A_1}, \widetilde{B_1} \in \mathbb{R}$. If $\widetilde{A_1} \leq \widetilde{B_1}$, then* $\text{sort}(\,(\widetilde{A_1}, A_2, A_3, \ldots, A_m)\,) \leq \text{sort}(\,(\widetilde{B_1}, B_2, B_3, \ldots, B_m)\,)$.

The proof is straightforward but technical; it can be found in the first author's Master's thesis [1, Section 3.3]. With this result, we show that the state relaxation relation satisfies Definition 1.

Lemma 4. *The relation $\preceq$ satisfies Definition 1.*

Proof. We show each of the properties (R1)-(R4) in order. Let S, S' be states on the same layer i such that $S' \preceq S$. (R1) follows directly from the reflexivity and transitivity of $\subseteq, \supseteq, \leq, \geq$ (pairwise and regular).

For (R2), let J be a job in $F_i(S)$. In other words, assume the following about J. First, $J \notin V$. Second, for all $J' \in \omega(J)$ we have $J' \in U$. Third, $\max\{F_1^u, T_{P(J)}^u\} \geq g$. We have to show that J is in $F_i(S')$, that is, we have to show three similar predicates.

The predicate $J \notin V'$ follows from $V' \subseteq V$. Now suppose for contradiction that some job $J' \in \omega(J)$ is not in U'. From $U' \supseteq U$, this job J' is also not in U. This contradicts the assumption that for all $J' \in \omega(J)$ we have $J' \in U$. For the inequality, note that $F^{u\prime} \geq F^u, T^{u\prime} \geq T^u, g' \leq g$, so indeed $\max\{F_1^{u\prime}, T_{P(J)}^{u\prime}\} \geq \max\{F_1^u, T_{P(J)}^u\} \geq g \geq g'$.

For (R3), let J be a feasible choice in S. We have to show $c_i(S', J) \leq c_i(S, J)$. The cost $c_i(S', J)$ is $z_J(\max\{F_1', T_{P(J)}'\})$, which is indeed at most the cost of making the choice in S, $z_J(\max\{F_1, T_{P(J)}\})$, since $F' \leq F$ and $T' \leq T$ and the z_J are non-decreasing.

For (R4), consider a job choice J that is feasible in S. Let $\overline{S'}$ and $\overline{S}$ denote the state after making this choice in S' and S, respectively. We have to show $\overline{S'} \preceq \overline{S}$. We do this by showing the predicates in the definition of $\preceq$.

$\overline{V'} \subseteq \overline{V}$ and $\overline{U'} \supseteq \overline{U}$ follow directly from $V' \subseteq V, U' \supseteq U$.

For $\overline{F'} \leq \overline{F}$, consider the definition of φ:

$$\overline{F} = \text{sort}(\,(\max\{F_1, T_{P(J)}\} + p(J), F_2, \ldots, F_m)\,)$$
$$\overline{F'} = \text{sort}(\,(\max\{F_1', T_{P(J)}'\} + p(J), F_2', \ldots, F_m')\,)$$

We have $\max\{F_1', T_{P(J)}'\} + p(J) \leq \max\{F_1, T_{P(J)}\} + p(J)$ and we may thus apply Lemma 3 to conclude $\overline{F'} \leq \overline{F}$.

For $\overline{T'} \leq \overline{T}$, note that from $F_1' \leq F_1$ and $T_{P(J)}' \leq T_{P(J)}$, we get $\overline{T_{P(J)}'} \leq \overline{T_{P(J)}}$. For $1 \leq i \leq k, i \neq P(J)$, we have equality in $\overline{T_i'} \leq \overline{T_i}$. Thus, we indeed have $\overline{T'} \leq \overline{T}$.

For $\overline{F^{w\prime}} \geq \overline{F^u}$, consider the equivalent predicate $\overline{F^u} \leq \overline{F^{w\prime}}$, which we can prove similarly to $\overline{F'} \leq \overline{F}$. The inequality $\overline{T^{w\prime}} \geq \overline{T^u}$ can be shown similarly to $\overline{T'} \leq \overline{T}$. Lastly, we have $\overline{g'} = \max\{F'_1, T'_{P(J)}\} \leq \max\{F_1, T_{P(J)}\} = \overline{g}$. □

We now consider our merge operator $\oplus$. The min (max) operator on vectors takes pairwise minima (maxima). This merge operator is associative, as required. We now prove the validity of our merge operator. Let us first consider an observation that is easy to prove.

Observation 1. *Let X, Y be sorted vectors of length m. Then* $\min(X, Y)$ *and* $\max(X, Y)$ *are sorted as well.*

This observation implies that in the merged state $S \oplus \dot{S}$, the vectors $\min(F, \dot{F})$ and $\max(F^u, \dot{F^u})$ are automatically sorted. To show that our merge operator is valid, we use Theorem 1.

Theorem 3. *The merge operator $\oplus$ is valid.*

Proof. Let $S, \dot{S}$ be states. We have to show $S \oplus \dot{S} \preceq S, \dot{S}$.

First, $V \cap \dot{V} \subseteq V, \dot{V}$ and $U \cup \dot{U} \supseteq U, \dot{U}$ are trivial. Further, $\min(F, \dot{F}) \leq F, \dot{F}$ holds as well because we take pairwise minima and do pairwise comparisons. Similarly, $\min(T, \dot{T}) \leq T, \dot{T}$ as well as $\max(F^u, \dot{F^u}) \geq F^u, \dot{F^u}$ and $\max(T^u, \dot{T^u}) \geq T^u, \dot{T^u}$. Lastly, $\min\{g, \dot{g}\} \leq g, \dot{g}$ holds. □

The nodes with the highest partial objective are merged, breaking ties by (high) *slack* of the last scheduled job. The slack is the difference between due time and completion time. We found the choice of merge heuristic is important for obtaining decent bounds, like [4].

4. Experiments

We evaluate our relaxed DD formulation using the total tardiness objective function. All experiments were conducted in the following environment: Intel Core i7-3537U (4 threads), 12GB RAM, C# (x64, .NET 4.7), Windows 10 version 1709.

Figure 2: Naive DD formulation. v is a node on layer L_i with state $S(v) = (V, U, F, T)$, and $\preceq, \oplus$ correspond to the real formulation.

$$
\begin{aligned}
&S(r) = (\varnothing, \varnothing, \vec{0}, (r(1), \ldots, r(k))) \\
&F_i(S(v)) = \{(J, M) \mid J \in \mathcal{J} - V, M \in \mathcal{M}, \omega(J) \subseteq U\} \\
&c_i(S(v), (J, M)) = z_J(\max\{F_M, T_{P(J)}\}) \\
&\varphi_i(S(v), (J, M)) = (V \cup \{J\}, U \cup \{J\}, \overline{F}, \overline{T}), \text{ where} \\
&\overline{F} = (F_1, \ldots, F_{M-1}, \overline{T_{P(J)}}, F_{M+1}, \ldots, F_m) \\
&\overline{T_j} = \begin{cases} \max\{F_M, T_j\} + p(J) & \text{if } j = P(J) \\ T_j & \text{otherwise} \end{cases}
\end{aligned}
$$

4.1. Instance Solver

To compare our DD formulation to existing solvers, we extend it to a solver that attempts to prove the optimality of a solution. We assume a feasible solution is given and try to find a lower bound equal to the solution to prove optimality.

We try to find an appropriate width W, that is, a width for which the lower bound is likely equal to the optimum. We empirically found that the bound is logarithmic in the width (see subsection 4.3). Hence, we employ linear extrapolation on $\log W$. In other words, we use linear extrapolation on w where $W = 2^w$.

We use $w = 12$ and $w = 15$ to do the initial extrapolation: we build a relaxed DD with the widths $W = 2^w$. If the lower bound is not yet equal to the solution, we use the new w and bound for the next extrapolation. We iterate this procedure until we either obtain a lower bound equal to the solution or hit our time or memory limit.

We increase w by at most 3 in each iteration. This is to mitigate extrapolation errors, which can be severe as an error in w is an exponentially large error in W. Also, if the second bound ($w = 15$) is at most 3 more than the first bound ($w = 12$), we try $w = 16$, hoping to get a better extrapolation.

4.2. Setup

We compare our DD formulation to a constraint program (CP), a mixed integer program (MIP), and a naive extension of the single-machine DD formulation by [4].

Figure 3: Constraint Program.

$$
\begin{aligned}
\text{Minimize} \quad & \sum_J \max\{0, \text{StartOf}(I_J) + p(J) - d(J)\} \\
\text{s.t.} \quad & I_{J,M} \in \text{Intervals}([r(J), \infty), p(J)) && (J \in \mathcal{J},\ M \in \mathcal{M}) \\
& I_J \in \{I_{J,M} \mid 1 \leq M \leq m\} && (J \in \mathcal{J}) \\
& \text{NoOverlap}(\{I_{J,M} \mid 1 \leq J \leq n\}) && (M \in \mathcal{M}) \\
& \text{NoOverlap}(\{I_J \mid P(J) = i\}) && (i \in \mathcal{P}) \\
& \text{EndBeforeStart}(J, J') && (J, J' \in \mathcal{J},\ J \to J')
\end{aligned}
$$

The naive formulation (Figure 2) does not use permutations, but instead uses a list of n (job, machine) pairs. Each job is scheduled on the machine it is in a pair with, and the order of jobs on each machine is given by the order of the list. We also do not sort the vector F. The permutation optimization naturally does not apply.

Our CP (Figure 3) is based on one of the samples provided in the IBM ILOG CP solver. We enabled presolving and used the "restart" method, which gives decent performance as well as intermediate lower bounds.

Our MIP is based on a time-indexed one given by [3]. It has a binary variable for each (job, time) pair, where "time" is any time step a job could be scheduled.

We use the makespan of a random schedule as horizon, and we adapted the MIP to include the partition and precedence constraints. See Figure 4, where ω^{-1} is defined as the inverse of ω, i.e., $\omega^{-1}(J) = \{J' \mid J \to J'\}$.

In our DD implementation, we used parallelization for building and merging: each thread builds and merges part of each layer.

Regarding instances, we used parameters and distributions much like those used by [4], for single-machine scheduling. The processing times $p(J)$ are integers drawn from $\{100, \ldots, 160\}$, the release times $r(i)$ are drawn from $\{0, \ldots, 50n\}$, and the due times $d(J)$ are drawn from $\{r(P(J)) + (p(J) + 4x)/m \mid x = 100, \ldots, 160\}$. The divisor m is not used in [4] and models the fact that with m machines, we can roughly finish m times as quickly. The jobs are partitioned as evenly as possible, and each job has a random partition assigned. We set $m = 2, k = 3$.

Compared to [4], we have scaled most numbers by a factor 10. The

Figure 4: Mixed Integer Program, where $g(t, J) = \max\{0, t - p(J) + 1\}$.

$$\text{Minimize} \quad \sum_{J} \sum_{t=0}^{H} x_{J,t} \max\{0, t + p(J) - d(J)\}$$

$$\begin{aligned}
\text{s.t.} \quad & x_{J,t} = 0 && (J \in \mathcal{J},\ t < r(P(J))) \\
& \sum_{t=0}^{H} x_{J,t} = 1 && (J \in \mathcal{J}) \\
& \sum_{J} \sum_{t'=g(t,J)}^{t} x_{J,t'} \leq m && (0 \leq t \leq H) \\
& \sum_{\substack{J \\ P(J)=i}} \sum_{t'=g(t,J)}^{t} x_{J,t'} \leq 1 && (i \in \mathcal{P},\ 0 \leq t \leq H) \\
& \sum_{J' \in \omega^{-1}(J)} \sum_{t'=0}^{t} x_{J',t'} \leq (t+1)|\omega^{-1}(J)|(1 - x_{J,t}) \\
& && (J \in \mathcal{J},\ 0 \leq t \leq H) \\
& x_{J,t} \in \{0, 1\} && (J \in \mathcal{J},\ 0 \leq t \leq H)
\end{aligned}$$

reason is that for such instances, the MIP is slow and therefore DDs seem more promising to improve performance.

For half of the instances, we set precedence constraints as follows. Consider a partition with p jobs $J_i, J_{i+1}, \ldots, J_{i+p-1}$. We add precedence constraints $J_i \to J_{i+1}, J_{i+2} \to J_{i+3}, \ldots$, ignoring the last job if p is odd. The reason we use these precedence constraints is that adding more may cause the search space to be reduced to a rather large extent.

4.3. Lower Bound Versus Width

Recall the logarithmic extrapolation in our instance solver. We found this relation empirically; see Figure 5. The plot shows the median and the $0\%, 10\%, \ldots, 100\%$ percentiles of the lower bound for various widths. The 3965 instances we used have $n = 12$. We normalized the lower bound by dividing by the best upper bound we found for each instance to allow comparison. (The upper bounds are found by a modification of [7], which in more than 97% of the instances was exact.) The highlighted lines are $W = 2^{12}$ and $W = 2^{15}$, which we use for the initial extrapolation in the solver.

It should be noted that the percentiles in Figure 5 are computed separately for each width W, so the lines do not necessarily correspond to specific instances. But the plot does give an idea how the bound behaves as W varies.

The percentiles are not exactly straight lines. However, for a given width, the curves are roughly straight at that point, that is, there appears to be a logarithmic relation between the width and the lower bound.

What the plot also suggests is that precedence constraints make the problem easier: the bound for a given width is typically higher, and the exact bound is generally found for a smaller width than when precedence constraints are absent.

4.4. Comparison to Naive DDs

To investigate to what extent our formulation improves upon a naive extension of the single-machine scheduling formulation by [4], we compare our DD formulation to the naive extension. We use $n = 10$ because the naive formulation uses more memory (more job choices per state).

We tested 600 instances. Figure 6 shows the decrease in gap for both our "real" DD formulation and the naive one. We see that our formulation is clearly superior. For the naive formulation, the average time for a non-zero lower bound is more than 1.7 seconds, and only 90 instances were solved

Figure 5: Relative lower bound for varying widths W. The shades of blue represent percentiles that are a multiple of 10. The widths $W = 2^{12}$ and $W = 2^{15}$ are highlighted. The number of instances are 1955 and 2010, respectively. [single column]

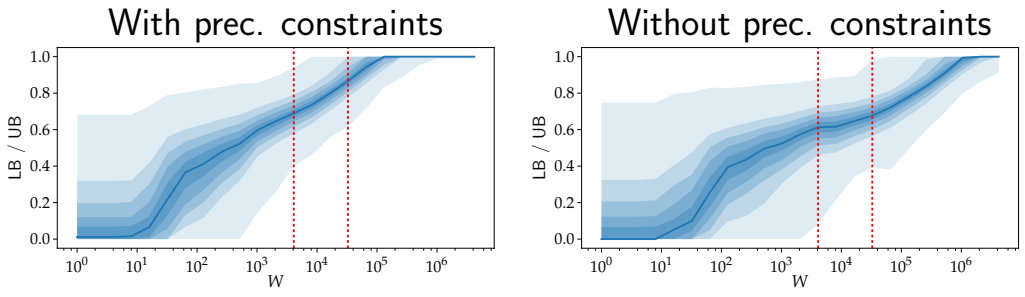


Figure 6: Gap over time for our DD formulation and the naive extension of the single-machine formulation by Hooker [4]. The number of instances are 303 and 297, respectively. [single column]

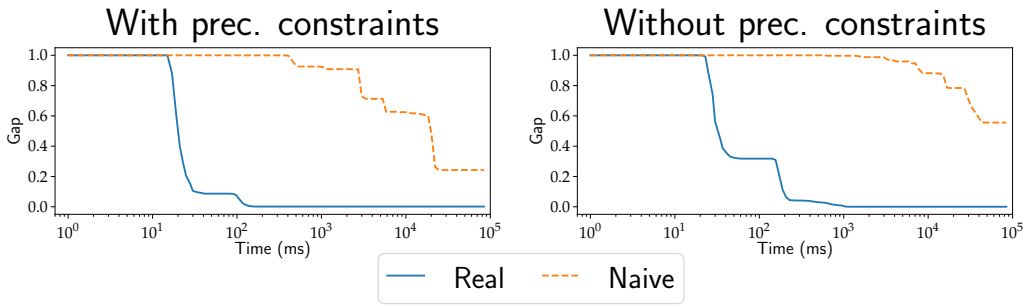
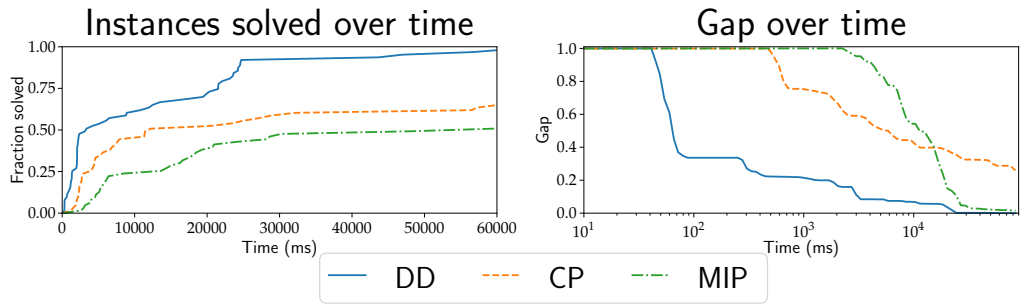


Figure 7: Comparison to CP and MIP. The number of instances is 63. [single column]



to optimality. To compare, the real formulation solved 583 instances to optimality, and the average *solve time* for these instances was 0.25 seconds.

We conclude that extending [4] to multi-machine scheduling is not trivial, and that the relaxation we introduced in this paper is essential to obtain decent performance.

4.5. Comparison to CP and MIP

Figure 7 shows the performance of our DD formulation and the CP and MIP models. Both plots show that our formulation improves upon these models: it solves instances more quickly to optimality, and on average the gap decreases faster. Note that it typically takes some time for the MIP to give a nonzero lower bound. This is due the high processing times, which causes this model to have a large number of binary variables.

5. Conclusion and Future Work

In this paper, we presented an exact and relaxed DD formulation for multi-machine scheduling. We found that our formulation improves upon a CP and MIP model, and greatly improves upon a naive extension of the state of the art for single-machine scheduling [4]. Nevertheless, DDs may have more potential than we have shown.

A downside of DDs is their memory usage. Each node in our formulation uses $O(m + n)$ memory for its state, so that the total memory usage is $O(nW(m + n))$. The factor W is unfortunate because we found empirically that one has to double W to get a constant increase in the lower bound. One method of reducing W might be accomplished by using a more sophisticated merge heuristic. Second, it might be beneficial to merge multiple sets of nodes. Currently, we simply merge many nodes into one merged node, and leave the (at most) $W - 1$ other nodes as is. This might not be the ideal choice. However, sometimes an exponential DD size is inevitable with our model (see Appendix A in [1]).

It is also interesting to tackle other multi-machine scheduling problems using DDs. For example, one extension we considered during research is allowing the processing time of a job to depend on the machine the job is scheduled on (called the *unrelated machines* setting). See Appendix B in [1].

An optimization regarding the solver is a better extrapolation method. Recall that we perform linear extrapolation on the logarithm of the width to

decide for which width the DD likely yields the optimum. Extrapolation errors are disadvantageous here because the error in the width is exponential. Although we found a roughly logarithmic relation, it is not exactly logarithmic. A more sophisticated extrapolation technique might provide better bounds.

References

[1] P. van den Bogaerdt, Multi-Machine Scheduling Lower Bounds Using Decision Diagrams, Master's thesis, Delft University of Technology, The Netherlands, 2018.

[2] N. Balakrishnan, J. J. Kanet, V. Sridharan, Early/tardy scheduling with sequence dependent setups on uniform parallel machines, Computers & Operations Research 26 (2) (1999) 127–141.

[3] P. Baptiste, A. Jouglet, D. Savourey, Lower bounds for parallel machine scheduling problems, International Journal of Operational Research 3 (6) (2008) 643–664.

[4] J. N. Hooker, Job Sequencing Bounds from Decision Diagrams, Principles and Practice of Constraint Programming (2017) 565–578.

[5] A. A. Cire, W.-J. van Hoeve, Multivalued decision diagrams for sequencing problems, Operations Research 61 (6) (2013) 1411–1428.

[6] D. Bergman, A. A. Cire, W.-J. van Hoeve, J. Hooker, Decision diagrams for optimization, Springer, 2016.

[7] R. Rodrigues, A. Pessoa, E. Uchoa, M. Poggi de Aragão, Heuristic algorithm for the parallel machine total weighted tardiness scheduling problem, Relatório de Pesquisa em Engenharia de Produçao 8 (10).

[8] B. Simons, Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines, SIAM Journal on Computing 12 (2) (1983) 294–299.

[9] J. M. J. Schutten, List scheduling revisited, Operations Research Letters 18 (4) (1996) 167–170.

[10] F. Yalaoui, C. Chu, New exact method to solve the $Pm/r_j/\sum C_j$ schedule problem, International Journal of Production Economics 100 (1) (2006) 168–179.

[11] A. Jouglet, D. Savourey, Dominance rules for the parallel machine total weighted tardiness scheduling problem with release dates, Computers & Operations Research 38 (9) (2011) 1259–1266.