Communication Papers of the 19th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 69–76
DOI: 10.15439/2024F9975
ISSN 2300-5963 ACSIS, Vol. 41

Job Shop Scheduling with Integer Programming, Shifting Bottleneck, and Decision Diagrams: A Computational Study

Brannon B. King
0000-0002-9269-6206
Computer Science Dept.
Virginia Tech in Blacksburg, VA
Email: brannonking@vt.edu

Robert Hildebrand
0000-0002-2730-0084
Industrial Systems and Engineering Dept.
Virginia Tech in Blacksburg, VA
Email: rhil@vt.edu

***Abstract*—We study heuristic algorithms for job shop scheduling problems. We compare classical approaches, such as the shifting bottleneck heuristic with novel strategies using decision diagrams. Balas' local refinement is used to improve feasible solutions. Heuristic approaches are combined with Mixed Integer Programming and Constraint Programming approaches. We discuss our results via computational experiments.**

I. Introduction

THE job shop scheduling problem (JSP) has long been a challenging area in operations research, historically tackled through disjunctive integer programming formulations that often yield poor linear programming relaxations. Constraint programming (CP) has emerged as a more effective approach for these problems, outperforming traditional mixed-integer programming (MIP) models. Heuristics, including scheduling and dispatching rules, have been extensively studied and applied to provide feasible solutions that can be further refined through local search algorithms. This paper explores the integration of problem-specific heuristics into modern solvers, with a focus on compound heuristic approaches utilizing Decision Diagrams (DDs) and Balas's local search methods.

Scheduling problems are known for having poor linear programming relaxations. To quote [1]: "In spite of a great deal of effort, the disjunctive integer programming formulation of the job-shop problem appears to be of little assistance in solving instances of even moderate size; furthermore, its natural linear programming relaxation has been shown to give very poor lower bounds for the problem." Solving the problem for a subset of machines/jobs seems to be the go-to mechanism for finding a better lower bound, and the solvers rely mostly on branching to improve the bounds.

On the other hand, primal heuristics for the problem abound [2]–[4]. They are often referred to as scheduling or dispatching "rules". And, as these provide feasible solutions, it's recommended to follow the heuristic with a local minimization algorithm. Also, approximation algorithms exist [5]. Balas [6] gave an inspiring local search algorithm many years ago based upon the critical path in a disjunctive graph representation of a scheduling problem. This heuristic is easily conjoined to modern MIP solvers.

Recent studies comparing constraint programming (CP) to mixed-integer programming (MIP) models show that CP clearly outperforms MIP in the realm of scheduling problems. See [7], [8], and [9]. The formulas in those papers are fairly standard and build on long-existing formulations [10]. We find, though, that tools/mechanisms for mixing heuristics with CP tools are lacking.

Recently, *Decision Diagrams* have shown to be a useful approach to some types of optimization problems. These perform a decomposition of the problem based on a sequential decision-making process. Bergman et al's book [11] on Decision Diagrams (DDs) is the starting point for our work. We assume the reader's familiarity with said book's content. We also assume familiarity with scheduling problems and their triplet notation (as in [12]). At first sight, DDs appear to be nothing more than a formal method for enumerating all possible solutions to a problem, with the detection of duplicated intermediates. However, their power is found in two separate mechanisms: 1) the restricted form that uses fixed memory to generate multiple feasible solutions, and in 2) how they can provide a relaxed form of the problem. This is done by intelligently merging nodes as the tree of solutions grows too wide. Solutions that go through one or more of these relaxed nodes provide a useful dual bound.

A. Contributions

We develop heuristic approaches based on Balas's work and on Decision Diagrams. Our novel Decision Diagram models for the JSP encourage minimizing stored symmetry, and thus reducing computational effort.

We evaluate the effectiveness of Restricted Decision Diagrams compared to traditional heuristics for JSPs. Lastly, we investigate the impact of warm-starting modern solvers with a heuristic solution. We then discuss the conclusions of

B. King and R. Hildebrand were partially funded by ONR Grant N00014-20-1-2156 and AFOSR grant FA9550-21-1-0107. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research or the Air Force Office of Scientific Research.

our computational experiments. We use GUROBI [13] and CPLEX [14] for MIP solvers and also CPLEX's constraint programming solver when testing CP versions of the JSP.

II. Formulation Background

A. Job shop scheduling

The scheduling problem denoted as $Jm||C_{max}$ refers to a specific class of job shop scheduling problems (JSPs) characterized by the goal of minimizing the makespan across multiple machines. Formulas for it are common in existing literature, e.g [10]. Formal description follows:

Let there be a set of n jobs $\{J_1, J_2, \ldots, J_n\}$ and a set of m machines $\{M_1, M_2, \ldots, M_m\}$. Each job J_i consists of a sequence of operations $\{O_{i1}, O_{i2}, \ldots, O_{im}\}$, where each operation O_{ij} must be processed on a specific machine $M_{\pi_i(j)}$ for some permutation map $\pi_i \colon [m] \to [m]$ and for a predetermined duration p_{ij} without interruption. Each machine can process only one operation at a time, and each operation can be processed on exactly one machine as per its predefined sequence in a job. The objective is to find a schedule, i.e., an allocation of operations to time slots on each machine, that minimizes the makespan C_{max}, which is the time when the last job completes processing.

1) Mathematical Optimization Formulation: We will focus on the disjunctive MIP formulation, which generally solves the quickest using MIP solvers [10]. The idea is to model the problem using binary variables to represent the sequencing decisions between operations on the same machine. Here's a step-by-step formulation:

Variables:

- S_{ij}: Nonnegative start time of operation O_{ij}.
- C_{ij}: Completion time of operation O_{ij} and easily collapsed into $S_{ij} + p_{ij}$.
- $C_{\max}$: Maximum completion time (makespan).
- x_{ijkl}: Binary variable indicating operation O_{ij} follows O_{kl}, only existing if both are on the same machine.

Constraints:

- Precedence Constraints: Ensure the correct order of operations within each job.
- Disjunctive Constraints (Eqs. 1c, 1d): Ensure that no two operations on the same machine overlap by enforcing that one must precede the other.

Mixed Integer Programming Model:

$$\min C_{\max} \quad \text{subject to} \tag{1a}$$
$$S_{ij} \geq C_{i(j-1)} \quad \forall i \in [n], \forall j \in J_i, j > 1 \tag{1b}$$
$$S_{ij} \geq C_{kl} - \overline{M}(1 - x_{ijkl}) \quad \forall ij, kl : \pi_i(j) = \pi_k(l) \tag{1c}$$
$$S_{kl} \geq C_{ij} - \overline{M} x_{ijkl} \quad \forall ij, kl : \pi_i(j) = \pi_k(l) \tag{1d}$$
$$S_{ij} \geq 0 \quad \forall i, j \tag{1e}$$
$$C_{ij} = S_{ij} + p_{ij} \quad \forall i, j \tag{1f}$$
$$C_{\max} \geq C_{ij} \quad \forall i, j : O_{ij} \text{ is final task of job } i \tag{1g}$$
$$x_{ijkl} \in \{0, 1\} \quad \forall i, j, k, l : \pi_i(j) = \pi_k(l) \tag{1h}$$

where $\overline{M}$ is a big-M multiplier, typically set to the one plus the sum of the delays: $1 + \sum_{ij} p_{ij}$. Those big-M constraints may also be formulated using the solver's indicator constraint functionality.

2) Constraint programming formulation: See the full explanation in [9].

Variables:

- I_{ij}: Interval variable containing the start and end of operation O_{ij} with width as the duration$= p_{ij}$.

CP Formula:

$$\min \max_{i \in [n]}(\text{end\_of}((I_{im}))) \quad \text{subject to:} \tag{2a}$$
$$\text{no\_overlap}(\{I_{ij} \mid i \in [n]\}) \quad \forall j \in [m] \tag{2b}$$
$$\text{end\_before\_start}(I_{ij}, I_{i(j+1)}) \quad \forall i \in [n], j \in [m-1] \tag{2c}$$
$$I_{ij} = \text{IntervalVar}(p_{ij}) \quad \forall i \in [n], j \in [m] \tag{2d}$$

B. Existing Work regarding scheduling via DD

Bergman et al. [11] discuss how a DD can solve scheduling problems in general and give this example: the single-machine makespan minimization with sequence-dependent delays: $1|p_{ij}|C_{max}$. They don't use a binary expansion tree; instead, they represent the problem as a permutation of possible orderings, known as a multivalue decision diagram (MDD), as shown in Table I.

TABLE I: Bergman's simple state operators

State	S_j holds the j jobs already done
Transition	$S_j \cup x$ where $x \in [n]$ and $x \notin S_j$
Cost	the delay from S_j to S_{j+1} via x

They do not give merge and split definitions for their scheduling example. They do give merge operations for other problems, namely, maximum independent set, maximum cut, and maximum 2-SAT. They also discuss how some merge operations might be possible for a scheduling problem if we separate the jobs already done into two groups: those that are surely done no matter what path arrives at a given state, and those that are done in at least one path arriving at a given state. This latter group is the "maybes". In [11]'s terms, the two groups are "All" and "Some". The maybes doesn't exist in a full expansion because, in that context, we don't have any nodes that represent more than one unique solution.

Hooker, in [15], expands on the above ideas for a more complicated scheduling problem shown in Table II, minimizing tardiness given release dates and due dates $1|r_j, d_j|\sum T_j$.

TABLE II: Hooker's tardiness operations

State	a tuple $S_j = (V, U, f)$: V holds up to j jobs surely done, U holds jobs done on some route, f is the running completion time
Transition	$(V \cup x, U \cap x, \max(r_x, f) + p_x)$ where $x \in [n]$ and $x \notin V$
Cost	the delay from S_j via x, the value of p_x
Merge	$(V \cap V', U \cup U', \min(f, f'))$

His paper demonstrates that these operations are sufficient to ensure that the relaxed tree contains a path that represents a dual (in this case, lower) bound. He also gives a mechanism for proving any merge operation to be sufficient. He expands his effort with a later paper, [16], where he includes merge operations for sequencing with time windows, time-dependent processing times, sequence-dependent processing times, and state-dependent processing times. They all follow a pattern very similar to the one given above.

Moreover, in [16], Hooker suggests optimizing the Lagrangian dual where he incorporates an additional penalty for sequences that repeat operations – a common infeasibility in a relaxed DD representation of a schedule. It was inspired by [17]. The paper also includes a table of timed runs on the CPW and Biskup-Feldman public datasets containing tardiness problems. (We were unable to make this succeed in our context. Perhaps it was due to some failure of our model to meet the necessary assumptions, or it required an extremely high number of iterations for convergence, or coding error.)

Building on Hooker's work, [18] provides two recent papers tackling multi-machine scheduling. In [18], they focus on a tardiness problem with substantial state; $(V, U, f, t, f^u, t^u, g)$. f, t, f^u, t^u are all vectors, where f refers to running completion times per machine, t refers to running release times, and the superscript u implies the same for the maybes (the items included in some ancestral lines but not all). They show that their merge operation is suitable using Hooker's rules. It's given here:

$$(V \cap V',\ U \cup U',\ \min(f, f'),\ \min(t, t'),\ \max(f^u, f^{u\prime}),\ \max(t^u, t^{u\prime}),\ \min(g, g'))$$

In their later work, [19], they build rules for uniform scheduling over total tardiness, or $Um|r_j, st_{jj'}, d_j|\sum T_j$. They track the current machine as part of the state. This leads to a notable limitation; their merge operation is only allowed to merge nodes where the current machine matches. They generally follow the patterns given above for tardiness problems.

C. Decision Diagrams

See [20] for a recent review of decision diagrams (DDs) for discrete optimization. We follow that survey for some details.

III. DD Operators for the JSP

In this section we present several different models that store and transition state in DDs. All solve the JSP, but not at the same efficiency. They are not the only possible models. For merging state, additional information must be stored, and that is covered in a separate section.

If we assign each operation a unique identifier, we have all feasible solutions as the permutations of those identifiers. Of course, most permutations are infeasible in that some job's operations may be out of order. Moreover, if we have any permutation, either complete or partial, we can trivially compute its completion times, feasibility, and, for partial sets, a lower bound on the completion max. See Algorithm 1.

Algorithm 1 Cost from partial solution (CFP)

Require: X is a list of to-be-done operations.
Require: $\mathbf{f}^M$ is machine completion times, **0** by default.
Require: O operations already done with times $\mathbf{f}^O$.
  for $x \in X$ **do**
2: $s \leftarrow \mathbf{f}^M_{mach(x)}$
  if $\exists\ pre(x)$ **then**
4: **if** $pre(x) \notin O$ **then**
  return Error: Missing Prerequisite
6: $s \leftarrow \text{MAX}(s, \mathbf{f}^O_{pre(x)})$
  $\mathbf{f}^O_x \leftarrow s + \text{DELAY}(x)$
8: $\mathbf{f}^M_{mach(x)} \leftarrow s + \text{DELAY}(x)$
  return $\text{MAX}(\mathbf{f}^M)$

Given a partial ordering and the next operation to be concatenated to it, we can trivially compute the change in completion times brought about by the additional operation. We can also update the running C_{max} if it is changed by this additional operation.

A. The basic permutation model

Since we're just storing a partial ordering, all we need is a list. We make use of some helpers such as $pre(x)$, which returns the operation required right before x if there is one. $mach(x)$ returns the required machine for operation x. $delay(x)$ is the delay for operation x, commonly referred to as p_{ij}. $trailer(x)$ contains the sum of the operation times that must follow operation x, where this could be either the amount of work left on the job of x or the machine required for x (or the maximum of both of those). Given a list of operations with labels in $[n]$, we get Table III.

TABLE III: JSP-for-MDD Model 0

State	$S := V$, an ordered list of ops. done so far
Cost	$c(S, x) := \text{CFP}([V, x])$
Transition	$\phi(S, x) := [V, x] : x \in [n] \backslash V,\ pre(x) \in V$

If we make our model slightly more advanced, we can cache the completion times for each operation (in $\mathbf{f}^O$) and each machine (in $\mathbf{f}^M$), we get Table IV.

TABLE IV: JSP-for-MDD Model 1

State	a tuple $S := (V, \mathbf{f}^O, \mathbf{f}^M)$
Cost	$c(S, x) := \max(\mathbf{f}^M_{mach(x)}, \mathbf{f}^O_{pre(x)}) + delay(x)$
Transition	$\phi(S, x) := (V \cup \{x\},$ $\mathbf{f}^O_x \leftarrow c(S, x), \mathbf{f}^M_{mach(x)} \leftarrow c(S, x))$

It's implied that $(V, \mathbf{f}^O, \mathbf{f}^M)$ are independent for every state – copied from the parent state and then modified/extended. When storing the states in each layer, it's useful to store duplicate states only once. When comparing these states, all fields of the tuple must be compared. Notice that only states within a given layer will have equal cardinality for V (unless states are merged in some way that violates that, as discussed in the merge section below).

B. A model to detect more symmetry

When growing a tree of possible solutions, such as is done by decision diagrams, one may arrive at equivalent solutions through differing paths. In this context such solutions are symmetrical. With a goal of keeping the tree of possible solutions as small as possible, DDs benefit from any reduction in state space. We want to avoid symmetry in the storage of our accumulated state.

The above model is not bad, but notice (Table V) that many of the stored completion times can never eclipse any machine's finish time. Sometimes those may happen in some other order and still produce the same state; we redesign the state to capture that symmetry.

TABLE V: JSP-for-MDD Model 2

State	a tuple $S := (V, V_L, \mathbf{f}^O, \mathbf{f}^M)$;
Cost	$c(S,x) := \max(\mathbf{f}^M_{mach(x)}, \mathbf{f}^O_{pre(x)}) + delay(x)$
Transition	$\phi(S,x) := (V \cup \{x\} \backslash pre(x), V_L \cup pre(x),$ $\mathbf{f}^O_x \leftarrow c, \mathbf{f}^M_{mach(x)} \leftarrow c)$

V_L refers to those operations that are long-done, those that will never need to be used as an immediate prerequisite. *For hash and comparison we ignore the completion times of operations in V_L.* If $pre(x) \notin \mathbf{f}^O$ you can return zero, although tracking some completion time for items in V_L can be handy for generating the final schedule at the end.

In Table VI, we show the number of nodes for full expansion averaged over 10 random problems to demonstrate that the number of nodes expanded is reduced by capturing more symmetry:

TABLE VI: Node expansion demonstration

	Model 1	**Model 2**
Valid nodes, 4x5 JSP	1245k	793k
Duplicates	1017k	776k
Valid nodes, 3x10 JSP	726k	528k
Duplicates	638k	504k

C. Modeling disjunctives directly

While the MILP formulation described above relies on real variables for start and completion times, the values for these variables can be fully determined from a fixed set of binary variables, denoted as x. This is common in scheduling problems. Consequently, this can be directly modeled as a binary decision diagram (BDD), as opposed to the multivalue or MDDs mentioned earlier. For each disjunction, a binary variable x_i determines whether the path is forward or reverse. In this context, forward implies that $i < j$ in an adjacency matrix representation of the disjunctive graph of the problem, as explained in [6]. Following the approach in that paper, it is assumed that the process begins with a feasible set of disjunctives, all oriented forward.

This approach encompasses a significantly larger number of variables and consequently necessitates many more layers compared to MDDs. For example, the well-known JSP problem *abz5* involves 100 operations but 900 disjunctives. (The number of dijunctives matches the number of binary variables shown in the Gurobi log, but it can also be easily computed from the number of bidirectional arcs in 10 cliques of 10 nodes each: $10n(n-1)$.) Grouping these bits into bytes could reduce the number of layers by a factor of eight while simultaneously increasing the width of each layer by the same factor. Additionally, this method might potentially reduce the 256 possible expansions by eliminating common 3- or 4-cycle problems from the possible values, although this approach has not yet been explored.

Given a disjunctive model, including invalid models with cycles, we can always find the longest path through that model in polynomial time or less. That's done trivially with a flow LP or an adjacency matrix selection LP. The challenge is to reduce the number of LP calls needed, as one for each state (or every other state as shown below) gets expensive. This approach was generally unhelpful although it did work correctly; it's included here in Table VII for contrast.

TABLE VII: Basic BDD-with-LP for layer j

State	V holds the reversed disjunctives' index
Transition	$V' = [V, j]$ if $x = 1$ else $V' = V$
Cost	LP(V)

With this approach, each node is unique; there is no duplication and no reduction in possible values as you progress through the layers. You cannot cull states that have cycles in their graph as later reversals may eliminate those cycles. As with all DD approaches, it would be worth it to cull via some known primal bound, heuristically determined.

D. Merging state

Both Model 1 and Model 2 support merge operations. Following [15] and [18], we add additional fields to the state: V_s and $\mathbf{f}^s$, where s stands for "some", meaning that some ancestral path covered the items in this set. For merging Model 1 state, $S' \leftarrow S \oplus \overline{S}$:

$$\mathbf{f}^{M\prime}_i \leftarrow \min(\mathbf{f}^M_i, \overline{\mathbf{f}^M_i}) \quad \forall i \in M \tag{3a}$$

$$V' \leftarrow V \cap \overline{V} \tag{3b}$$

$$\mathbf{f}^{O\prime}_x \leftarrow \max(\mathbf{f}^O_x, \overline{\mathbf{f}^O_x}) \quad \forall x \in V' \tag{3c}$$

$$V'_s \leftarrow (V \cup \overline{V}) - V' \tag{3d}$$

$$\mathbf{f}^{s\prime}_x \leftarrow \max(\mathbf{f}^O_x, \overline{\mathbf{f}^O_x}, \mathbf{f}^s_x, \overline{\mathbf{f}^s_x}) \quad \forall x \in V'_s \tag{3e}$$

Hooker [15] gives two criteria for a valid merge: any possible control values leaving S' must be a valid relaxation of the same control leaving S, and both S and $\overline{S}$ must be relaxed/interchangable. The latter criteria holds from the lack of order-dependent operations in the above formulation. The former criteria holds in that we always take the minimum score for our $C_{max} = \max(\mathbf{f}^{M\prime}_i)$, and we always use the worst-case prerequisite completion time when pulling items from $\mathbf{f}^O$. Model 2 is merged similarly with V_L receiving equivalent treatment to V.

We used the merge operation to validate that Bergman's branch-and-bound algorithm [11] worked for JSP. It did not scale well, though. We were unable to make the relaxed DD give a better lower bound than the linear polytope of the

disjunctive formulation, so we do not include this in our computational results below. As part of that, we pruned nodes that exceeded the current primal bound, which is known as LocB pruning in this context [21]. See the other thoughts on the algorithm in the appendix.

IV. A simple local search refinement

In Balas' original paper [6] about solving JSP via disjunctive graph iteration, he included this special proposition:

Proposition of Balas 1969:

> Let C_h be a critical path in G_h [which is a DAG]. Any graph G_k obtained from G_h by complementing one arc $(i,j) \in C_h$ is circuit-free.

Balas' proof: We know that (i,j) must be the longest path from i to j as it is part of C_h. However, it is also the shortest path from i to j or we would have chosen the longer path. Because i to j is the only path, we can reverse it without creating a circuit.

That leads us to this refinement algorithm as a local search method (Alg. 2):

Algorithm 2 Balas Local Refinement 1 (LNS1)

Require: $g = (V, \mathcal{A})$ is a DAG of the fixed conjunctive arcs.
Require: $W_e \geq 0$ for $e \in \mathcal{A}$ as the weight of each arc.
Require: s_{parent} is the s value from the caller.
$p \leftarrow$ LONGEST_PATH(g)
$s \leftarrow \sum_{(u,v)\in p} W_{uv}$
if $s > s_{parent}$ **then**
  return s // remove this to search more space
for $e \in p$ **do**
  if not $e.fixed$ **then**
    $g \leftarrow$ REMOVE_ARC(g, e)
    $g \leftarrow$ ADD_FIXED_ARC$(g, e.v, e.u, W_{vu})$
    $t \leftarrow$ RECURSE(g, W, s)
    if $t < s$ **then**
      $s = t$ // can also track swaps here
    $g \leftarrow$ REMOVE_ARC$(g, e.v, e.u)$
    $g \leftarrow$ ADD_FIXED_ARC(g, e)
return s // return swaps also if desired

V. Computational Experiments

A. Use as a Heuristic

In this section we show how a restricted Model 2 compares to other common heuristics including MOR, MWR, and the shifting bottleneck [22]. We include multiple widths for the restricted DD, but this parameter does not substantially improve the bound it computes; see [15].

Observation: The restricted DD always produces at least one feasible solution. Given any partial solution that is feasible, the remaining operations can always be added in a feasible order. Note that this does not hold if you filter the nodes in the DD with anything additional to the maximum width filter. For example, if you filter nodes whose C_{max} exceeds some threshold (in addition to reducing row width), you may filter all possible paths toward the conclusion of the DD.

In Table VIII, we run each heuristic over twenty random 10×10 JSP instances, making use of Gurobi [13], CPLEX [14], and Job Shop Library [23]. $\mathcal{W}$ refers to the maximum width of the DD. Overage is how far the final bound was above the optimum. We use default solver settings with the exception of Gurobi's AggFill=10 and GomoryPasses=1, which was recommended by their tuner for these problems. The random instances are integer and similar to those published with [22]).

TABLE VIII: Avg. heuristic overages for 20 random

Heuristic	Time	Overage	After LNS1
Gurobi, MIPGap=0.25	0.27s	11.1%	9.18%
Shifting Bottleneck	4.0s	15.8%	15.3%
Restricted DD, $\mathcal{W}$=200	0.25s	18.5%	14.6%
Restricted DD, $\mathcal{W}$=400	0.50s	16.8%	12.5%
Most Work Remaining (MWR)		26.9%	16.1%
Most Ops. Remaining (MOR)		20.1%	14.1%
Shortest Proc. Time (SPT)		90.0%	40.1%

TABLE IX: Avg. heuristic overages for 18 from JSPLIB

Heuristic	Time	Overage	After LNS1
Gurobi, MIPGap=0.25	1.6s	7.77%	5.85%
Gurobi, MIPGap=0.40	0.12s	25.8%	12.5%
Shifting Bottleneck		22.8%	21.9%
Restricted DD, $\mathcal{W}$=200		14.4%	11.9%
Restricted DD, $\mathcal{W}$=400		11.6%	9.70%
Most Work Remaining (MWR)		31.4%	19.8%
Most Ops. Remaining (MOR)		29.9%	23.6%
Shortest Proc. Time (SPT)		80.8%	39.7%

Interestingly, (Table IX) the DD approach worked better on the real-world problems found in JSPLIB [24] – it's eighteen 10×10 problems. Similarly, the small 0.25 MIPGAP for Gurobi was much more difficult to achieve on the JSPLIB problems.

The comparison is a little bit unfair, in that the Restricted DDs generate many feasible solutions for the LNS1 whereas the top four only produce a single solution to be refined. However, it shows that the local refinement eliminates the need for the shifting bottleneck heuristic.

We recognize that there are more sophisticated versions of the shifting bottleneck algorithm, e.g. [25]. There are also a variety of other local search mechanisms designed for JSP that are far more sophisticated and far-reaching, typically built on taboo search, e.g. [26], [27]. We did not consider simulated annealing nor any evolutionary algorithm as part of this research either, though papers on those approaches for JSP abound.

Regarding Gurobi NoRel: We ran Gurobi's NoRel heuristic for 4 seconds on the same 20 problems. It failed to find any solutions on 80% of the problems, but on the other four, it found solutions within 3% of optimal. Note that Gurobi can solve a 10x10 disjunctive program in 2 to 8 seconds on our test machine, so running a 4-second heuristic for it would not make sense generally. The NoRel runtime has to be specified as an input.

Regarding runtimes: Note that the Shifting Bottleneck (SB) without readjustment of machines in M_0 plus the LNS at the end still achieves 18%. This takes about two seconds to run whereas the other takes 4 seconds per 10x10. SB can be modified to solve subproblems in parallel, which was not a part of our implementation. We rely on Gurobi to solve the $1|r_j|L_{max}$ subproblems in the SB. This takes up 90% of the runtime for it. Generally, though, when using SB one would use Carlier's approximation [28] for the L_{max} instead of solving it via a MIP solver (and probably still run them in parallel). See other ideas here: [5], [29]. With our DD written in Go, the 10x10 on a max-width of 400 takes half a second to run and half that time for the 200 width (using no parallelism). Most of that time is in comparison to previous nodes on the same row. Better hashing would improve that time. The LNS1 adds some additional time to that as it runs on each resulting node. This is not included in the time measurement listed above. The dispatching rule approximations obviously use a trivial amount of time.

Most of the items arriving at the bottom row of a DD tend to be similar, which comes from the sort-and-truncate approach. It needs some other selection criteria toward the top of the tree so that it keeps more diversity early on, which should give it better chances of enabling a good/unique neighborhood. The DD is quite sensitive to changes in the running C_{max} computation. For example, you can use the C_{max}-so-far or you can add to the trailer for the remaining items to be done on each machine or you can add the work remaining on the job. Those selections all change results quite drastically. The run recorded above does not add a trailer, as going without seemed slightly better on average.

B. The value of a starting point

Here we demonstrate the value of using a heuristic to select a starting solution when solving the JSP to its optimum. In Table X we run each solver over the same 20 instances used above but give it a single starting point. The starting points are derived using MOR followed by the refinement of the LNS1 algorithm described above.

TABLE X: Solve time with single warmstart

Solver on 10x10	**Time/problem**	**With warmstart**
Gurobi, big-M	2.1s	1.9s
Gurobi, indicator	4.0s	3.6s
CPLEX MP, big-M	3.4s	3.8s
CPLEX MP, indicator	130s	110s
CPLEX CP	1.17s	1.16s
Solver on 12x12	**Time/problem**	**With warmstart**
Gurobi, big-M	36s	43s
CPLEX MP, big-M	66s	60s
CPLEX CP	5.3s	5.5s

We conclude from this table that you should be using a constraint solver for exact solutions on this, and that the big-M path is more optimized than the indicator constraint feature, and that warm-starting it is unnecessary. Runs were made with Gurobi 11.0.2, CPLEX 22.1.1 on a i7-8750H processor.

Gurobi supports a heuristic parameter for controlling the percentage of time spent in heuristics. The default is 5%. We explored other values from 0% through 50% but could find no other value to improve the average time. Increasing the parameter by 5% generally added 5% to the runtime.

C. The value of LNS via callback

We demonstrate (Table XI) the value of calling LNS1 in the MIP node callback (CB). We take the ordering given by the start time variables (S) and run the local search on that. We can actually just submit the solution given by the ordering instead of running a local optimizer on it – herein called "Nearest".

TABLE XI: LNS1 in MIP-node callback

Solver	**Default**	**LNS1**	**Wins**	**Nearest**	**Wins**
Gurobi	2.1s	2.6s	5.6	2.1s	6.7

Note that we subtract the time of the callback itself, in that it is assumed that we can come up with more efficient implementations of it or curtail the calls to it as it becomes unlikely to assist. This optimization of how often the heuristic is called is a separate issue. Writing our LNS algorithm in a heuristic form that is fast enough to justify its use is nontrivial.

CPLEX, as documented, supports heuristic callbacks. However, in attempting to use them with version 22.1.1 on Linux, accessed via the Python docplex wrapper, we were unable to determine whether or not the solver was utilizing the provided heuristic solutions. No errors were given, but the incumbent scores were not updating as expected, so we did not include the numbers for it. We also have interesting numbers for FICO Xpress, but their license prohibits publishing.

D. Using our state model for A*

Unfortunately, Model 2 alone does not seem to be sufficient to allow solving a 10x10 via A*. We note that there are other efforts to make A* utilize relaxation features of the DD approach such as [30]. The nearness of the running C_{max} to reality is of critical importance in A*-search. It is possible to improve the trailer estimate by solving 2 of ten machines: see [31]. This is fairly quick, especially as the problems progress and most tasks have release times available. However, empirically, it's not enough accuracy to make the A* approach feasible.

VI. Conclusion

Conclusions from our experimentation:

1) Relaxed decision diagrams are useful as a simple JSP heuristic. They are not difficult to write/use and run fast. They produce better results than other common (simple) heuristics on real-world problems.
2) Passing a start point to the solver is not useful at 14% away from optima. Perhaps it would be worth it if you were using some more sophisticated heuristic that could

generate starting points within just a few percentage points from optima.
3) Running Balas' critical path refining, the LNS1, does generally improve a given feasible solution. It is fast to run and generally worth it.
4) The big-M handling in Gurobi and CPLEX is superior to their indicator constraint handling at present. It may be that our $\overline{M}$ value was small enough to tip that balance.
5) For problems where feasible solutions are rare, it may be helpful to find a nearby feasible solution in the callback if it can be done quickly. This computation is very fast for the JSP. It did help significantly on some of the test problems. Especially consider it if you use FICO Xpress.

Ideas for future work:

1) The selection of keeper nodes in the restricted DD is of critical importance. Using a basic rule like keeping the smallest 200 is a general failure – most of the nodes with the optimum are weeded out early on. That's the curse of these scheduling problems – the conflicts on the attractive solutions don't show up until late in the game. If we had some kind of machine learning approach that could identify bad nodes early on, we would have a higher chance of retaining the good nodes (or vice-versa). Node selection ideas from modern solvers may also apply [32].
2) Another idea for retaining nodes is to try to keep a diverse set using some kind of diversity measure that would increase the likelihood of keeping the optimum path.
3) Relaxed decision diagrams produce many infeasible solutions. Order them and you can expand these infeasible paths until you arrive at the first and best feasible solution, a best first approach similar to A*. We attempted this. However, there are so many infeasible solutions to weed through that this is generally not a viable approach for problems at scale. If we had some equivalent to cuts-for-LP, perhaps we could cut out subtrees in a way that allowed us to arrive at the best solution much faster.

APPENDIX

A. Notes on existing algorithms for exact solutions via DD

[11] presents two general algorithms for finding an exact solution to any program representable by a DD. The first mechanism is what they term "compiling DDs by separation", condensed form of the algorithm given in [33].

Algorithm summary: Begin with a relaxed decision diagram (DD) and identify the optimal path through it. If this path violates any constraints, separate the relaxed nodes on that path into two or more replacements. Adjust the inputs feeding into the violating node so that some go to each replacement. Similarly, replicate the outputs of the violating node to each replacement. Continue this process until the optimal path is feasible.

The process requires a split or "separation" operation, which essentially undoes the merge operation, though the necessary bookkeeping for this may be expensive. If no split operation is available, a possible solution is to backtrack to the parents of the merged nodes and regenerate their children. Additionally, we assume that the arcs store and maintain the variable value (also known as the control) and the cost of traversing them. This assumption differs from our previous experiments, where we kept only the fringe nodes with the running cost in the state. Furthermore, decision diagram (DD) creation generally employs node reduction (combining identical nodes), and this reduction must be maintained after adding additional nodes to the graph. If all identical nodes are on the same layer, the check is reasonable; otherwise, it becomes too expensive.

The second algorithm represents a branch-and-bound (BnB) approach, where you branch on a cutset of exact nodes, making a new subtree pair, both relaxed and restricted, for the decedents of each node in the cutset. Cutset refers to a set of exact nodes such that any path through the tree goes through one and only one of these nodes (before hitting any relaxed nodes).

Algorithm summary: While there are nodes in the queue, remove node u. Update the primal bound, which is the cost to node u plus a heuristic from it to the end, potentially determined by a restricted tree. Construct a relaxed decision diagram (DD) with u as its root. If the best relaxation is worse than the primal bound, exclude the entire u subtree. Otherwise, add the exact cutset of u to your queue and repeat the process.

Some general notes on this algorithm:

1) The processing of these subtrees is parallelizable (as noted in the reference).
2) It does not require a split operation, although it does require a working merge operation for building the relaxed trees.
3) It doesn't require a restricted tree if you have some other heuristic mechanism that completes partial solutions, as that may also provide a reasonable primal bound, especially if it's refined by a fast local search as the final step of the heuristic.
4) It doesn't make any use of the dual bound for subtree exclusion. This is its fundamental weakness.
5) Empirically, it's highly unlikely that to be able to exclude the whole relaxed tree based on its best node being worse than the current overall primal bound. Hence, you can simply return the cutset as soon as it is discovered. This eliminates the need for a merge operation.

The algorithm makes use of two things from the relaxed DD: its best path cost and its exact cutset. [11] gives three algorithms for selecting the cutset: take the first layer, take the last layer before any nodes are merged, or take the "frontier", meaning all the exact nodes that have at least one relaxed child. From that, we make these observations:

1) If we merge many nodes into one, that node has a high likelihood of being very relaxed. Thus we will keep it, as it has a good score, which will in turn lead to the best path through the relaxed tree being a poor estimate

of reality. Hence, again we will keep that tree's cutset, as our primal bound won't be able to exclude it.

2) If we take some layer before we merge any nodes, our cutset will be very shallow. Shallow nodes have lower likelihood of being excluded by constraints, assuming most constraints incorporate more than the first few variables. Moreover, it is utilizing less of our DD.
3) If we choose merge a lot in hopes of not over-relaxing any one path, we will force our cutset to be more shallow, thus getting less advantage from our DD expansion.

REFERENCES

[1] P. Martin and D. B. Shmoys, "A new approach to computing optimal schedules for the job-shop scheduling problem," in *Integer Programming and Combinatorial Optimization*, W. H. Cunningham, S. T. McCormick, and M. Queyranne, Eds. Springer Berlin Heidelberg, 1996, vol. 1084, pp. 389–403. ISBN 978-3-540-61310-7 978-3-540-68453-4. [Online]. Available: http://link.springer.com/10.1007/3-540-61310-2_29

[2] Y. P. Gupta, M. C. Gupta, and C. R. Bector, "A review of scheduling rules in flexible manufacturing systems," *International Journal of Computer Integrated Manufacturing*, vol. 2, no. 6, pp. 356–377, Nov. 1989. doi: 10.1080/09511928908944424. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/09511928908944424

[3] S. S. Panwalkar and W. Iskander, "A Survey of Scheduling Rules," *Operations Research*, vol. 25, no. 1, pp. 45–61, Feb. 1977. doi: 10.1287/opre.25.1.45

[4] A. Allahverdi, "The third comprehensive survey on scheduling problems with setup times/costs," *European Journal of Operational Research*, vol. 246, no. 2, pp. 345–378, Oct. 2015. doi: 10.1016/j.ejor.2015.04.004. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0377221715002763

[5] N. Grigoreva, "Worst-case analysis of an approximation algorithm for single machine scheduling problem," in *Proceedings of the 16th Conference on Computer Science and Intelligence Systems*, ser. FedCSIS 2021. IEEE, Sep. 2021. doi: 10.15439/2021f66. ISSN 2300-5963

[6] E. Balas, "Machine Sequencing Via Disjunctive Graphs: An Implicit Enumeration Algorithm," *Operations Research*, vol. 17, no. 6, pp. 941–957, Dec. 1969. doi: 10.1287/opre.17.6.941

[7] W.-Y. Ku and J. C. Beck, "Mixed integer programming models for job shop scheduling: A computational analysis," *Computers & Operations Research*, vol. 73, pp. 165–173, 2016. doi: 10.1016/j.cor.2016.04.006

[8] G. Da Col and E. C. Teppan, "Industrial-size job shop scheduling with constraint programming," *Operations Research Perspectives*, vol. 9, p. 100249, 2022. doi: 10.1016/j.orp.2022.100249

[9] B. Naderi, R. Ruiz, and V. Roshanaei, "Mixed-integer programming vs. constraint programming for shop scheduling problems: New results and outlook," *INFORMS Journal on Computing*, vol. 35, no. 4, pp. 817–843, 2023. doi: 10.1287/ijoc.2023.1287

[10] C.-H. Pan, "A study of integer programming formulations for scheduling problems," *International Journal of Systems Science*, vol. 28, no. 1, pp. 33–41, Jan. 1997. doi: 10.1080/00207729708929360. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/00207729708929360

[11] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker, *Decision Diagrams for Optimization*, ser. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer International Publishing, 2016. ISBN 978-3-319-42847-5 978-3-319-42849-9. [Online]. Available: http://link.springer.com/10.1007/978-3-319-42849-9

[12] Michael L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*. Springer International Publishing, 2022. ISBN 978-3-031-05920-9 978-3-031-05921-6. [Online]. Available: https://link.springer.com/10.1007/978-3-031-05921-6

[13] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual*, 2023. [Online]. Available: https://www.gurobi.com

[14] IBM, *IBM ILOG CPLEX Optimization Studio*, 2023. [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio

[15] J. N. Hooker, "Job Sequencing Bounds from Decision Diagrams," in *Principles and Practice of Constraint Programming*, J. C. Beck, Ed. Springer International Publishing, 2017, vol. 10416, pp. 565–578. ISBN 978-3-319-66157-5 978-3-319-66158-2. [Online]. Available: http://link.springer.com/10.1007/978-3-319-66158-2_36

[16] ——, "Improved Job Sequencing Bounds from Decision Diagrams," in *Principles and Practice of Constraint Programming*, T. Schiex and S. De Givry, Eds. Springer International Publishing, 2019, vol. 11802, pp. 268–283. ISBN 978-3-030-30047-0 978-3-030-30048-7. [Online]. Available: http://link.springer.com/10.1007/978-3-030-30048-7_16

[17] D. Bergman, A. A. Cire, and W.-J. Van Hoeve, "Lagrangian bounds from decision diagrams," *Constraints*, vol. 20, no. 3, pp. 346–361, Jul. 2015. doi: 10.1007/s10601-015-9193-y. [Online]. Available: http://link.springer.com/10.1007/s10601-015-9193-y

[18] P. Van Den Bogaerdt and M. De Weerdt, "Multi-machine scheduling lower bounds using decision diagrams," *Operations Research Letters*, vol. 46, no. 6, pp. 616–621, Nov. 2018. doi: 10.1016/j.orl.2018.11.003. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016763771830227X

[19] ——, "Lower Bounds for Uniform Machine Scheduling Using Decision Diagrams," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, L.-M. Rousseau and K. Stergiou, Eds. Springer International Publishing, 2019, vol. 11494, pp. 565–580. ISBN 978-3-030-19211-2 978-3-030-19212-9. [Online]. Available: http://link.springer.com/10.1007/978-3-030-19212-9_38

[20] M. P. Castro, A. A. Cire, and J. C. Beck, "Decision Diagrams for Discrete Optimization: A Survey of Recent Advances," *INFORMS Journal on Computing*, vol. 34, no. 4, pp. 2271–2295, Jul. 2022. doi: 10.1287/ijoc.2022.1170

[21] X. Gillard, V. Coppé, P. Schaus, and A. A. Cire, "Improving the filtering of branch-and-bound mdd solver," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, P. J. Stuckey, Ed. Cham: Springer International Publishing, 2021. doi: 10.1007/978-3-030-78230-6_15. ISBN 978-3-030-78230-6 pp. 231–247.

[22] J. Adams, E. Balas, and D. Zawack, "The Shifting Bottleneck Procedure for Job Shop Scheduling," *Management Science*, vol. 34, no. 3, pp. 391–401, Mar. 1988. doi: 10.1287/mnsc.34.3.391

[23] Pablo Ariño, *Job Shop Library*. [Online]. Available: https://github.com/Pabloo22/job_shop_lib

[24] T. Yasumasa, *JSPLIB-Benchmark instances for the job-shop scheduling problem*. [Online]. Available: https://github.com/tamy0612/JSPLIB

[25] E. Balas, N. Simonetti, and A. Vazacopoulos, "Job shop scheduling with setup times, deadlines and precedence constraints," *Journal of Scheduling*, vol. 11, no. 4, pp. 253–262, Aug. 2008. doi: 10.1007/s10951-008-0067-7. [Online]. Available: http://link.springer.com/10.1007/s10951-008-0067-7

[26] É. D. Taillard, "Parallel Taboo Search Techniques for the Job Shop Scheduling Problem," *ORSA Journal on Computing*, vol. 6, no. 2, pp. 108–117, May 1994. doi: 10.1287/ijoc.6.2.108

[27] E. Nowicki and C. Smutnicki, "An Advanced Tabu Search Algorithm for the Job Shop Problem," *Journal of Scheduling*, vol. 8, no. 2, pp. 145–159, Apr. 2005. doi: 10.1007/s10951-005-6364-5. [Online]. Available: http://link.springer.com/10.1007/s10951-005-6364-5

[28] J. Carlier, "The one-machine sequencing problem," *European Journal of Operational Research*, vol. 11, no. 1, pp. 42–47, 1982.

[29] M. Sinai and T. Tamir, "Minimizing tardiness in a scheduling environment with jobs' hierarchy," in *Proceedings of the 16th Conference on Computer Science and Intelligence Systems*, ser. FedCSIS 2021. IEEE, Sep. 2021. doi: 10.15439/2021f36. ISSN 2300-5963

[30] M. Horn, J. Maschler, G. R. Raidl, and E. Rönnberg, "A*-based construction of decision diagrams for a prize-collecting scheduling problem," *Computers & Operations Research*, vol. 126, p. 105125, 2021. doi: 10.1016/j.cor.2020.105125

[31] B. Jurisch, "Lower bounds for the job-shop scheduling problem on multi-purpose machines," *Discrete Applied Mathematics*, vol. 58, no. 2, pp. 145–156, 1995. doi: 10.1016/0166-218X(93)E0124-H

[32] T. Achterberg and T. Berthold, "Hybrid branching," *Lecture Notes in Computer Science*, p. 309–311, 2009. doi: 10.1007/978-3-642-01929-6_23

[33] T. Hadzic, J. N. Hooker, B. O'Sullivan, and P. Tiedemann, "Approximate Compilation of Constraints into Multivalued Decision Diagrams," in *Principles and Practice of Constraint Programming*, P. J. Stuckey, Ed. Springer Berlin Heidelberg, 2008, vol. 5202, pp. 448–462. ISBN 978-3-540-85957-4 978-3-540-85958-1. [Online]. Available: http://link.springer.com/10.1007/978-3-540-85958-1_30