# CP or DP? Why Not Both: A Case Study in the Partial Shop Scheduling Problem.

**Emma Legrand**
ICTEAM, UCLouvain, Belgium

**Roger Kameugne**
ICTEAM, UCLouvain, Belgium

**Pierre Schaus**
ICTEAM, UCLouvain, Belgium

## Abstract

Dynamic Programming (DP) and Constraint Programming (CP) are well-established paradigms for solving combinatorial optimization problems. Usually, these two approaches are used separately. This paper aims to show that the two can be combined effectively and elegantly, providing an alternative way to address the problem, with DP serving as the primary search framework and CP used as a subroutine to leverage global constraint propagation. This paper presents such an approach for the Partial Shop Scheduling Problem (PSSP), for which a pure DP method has previously been proposed, and efficient CP filtering algorithms are available. The PSSP is a general scheduling problem where each job consists of a set of operations with arbitrary precedence constraints. The approach is flexible enough to accommodate anytime DP strategies, such as anytime column search, whereas the original DP algorithm operated in a strictly layer-wise manner. While not competitive with state-of-the-art pure CP solvers for this specific problem, our primary contribution is demonstrating the viability of this hybrid integration.

**2012 ACM Subject Classification** Mathematics of computing → Combinatorial optimization

**Keywords and phrases** Partial Shop Scheduling Problem, dynamic programming, decision diagram, constraint programming

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

## 1 Introduction

This work considers solving the *Partial Shop Scheduling Problem* (PSSP) using dynamic programming (DP), constraint programming (CP), and a hybrid DP-CP approach. The PSSP is a general scheduling problem where each job consists of a set of operations with arbitrary precedence constraints. A *Partial Shop Scheduling Problem* (PSSP) defined by a finite set of operations $O = \{o_1, \ldots, o_N\}$ to be executed on a finite set of machines $M = \{M_1, \ldots, M_m\}$ with $|M| = m$. This set of operations is partitioned into $n = N/m$ subsets, with each machine assigned to exactly one operation in each partition. Each operation $o \in O$ has a specified processing time $p_o > 0$, requires a specific machine $m(o) \in M$ for its exclusive use without preemption, and is associated with a specific partition $j(o)$. Furthermore, the operations are subject to a set of precedence constraints represented by a directed acyclic graph (DAG) $G = (O, E)$. An edge $(o, o') \in E$ indicates that operation $o$ must be completed before operation $o'$ can start, denoted $o \lessdot o'$. The set of successors of an operation $o$ is denoted by $succs(o)$ and its predecessors by $preds(o)$. A solution to this problem is an assignment of start times $\psi_o$ to each operation $o \in O$ such that all precedence constraints ($\psi_{o'} \geq \psi_o + p_o$ for all $(o, o') \in E$) and machine and partition disjunctive constraints (no two operations on the same machine/partition overlap in time) are satisfied. The objective is to minimize the makespan $C_{\max} = \max_{o \in O}(\psi_o + p_o)$.

The PSSP generalizes classical scheduling problems such as the job-shop scheduling


problem (JSP) and the open-shop scheduling problem (OSP). In these problems, a set of operations must be scheduled on distinct machines, subject to given precedence constraints. The common objective is to minimize the makespan. These scheduling problems are typically strongly NP-hard [6]. Despite extensive research into exact methods for these problems, DP approaches remain relatively rare.

For the JSP, Gromicho et al. [7] introduced a DP formulation that views a solution as a sequence of operations, applying specific techniques to maintain optimality guarantees. By leveraging dominance properties to prune the search space, this algorithm successfully solves moderate-sized instances. Hoorn et al. [19] later refined this approach, demonstrating its ability to find optimal solutions for particularly hard instances on established benchmarks [20]. Building on this DP formulation, Ozolins [15] proposed an improved bounded variant, combining DP with branch-and-bound, experimented on medium-sized instances.

Pure CP approach combines the well-known NoOverlap global constraint [21] with specific search strategies [1, 12]. To achieve maximal domain filtering at each node of the search tree, this global constraint integrates several propagation techniques: *overload checking*, *edge-finding*, *detectable precedences*, and *not-first/not-last* rules.

In [8], the authors review the integration of CP and operations research (OR) to solve combinatorial optimization problems. A closely related work by Marijnissen et al. [14] developed in parallel and independently of this one also proposes a generic framework for integrating CP propagation into Domain-Independent Dynamic Programming (DIDP) [10].

In this paper, we demonstrate how a CP model can be elegantly integrated as a subroutine within a DP approach to leverage the powerful global constraint propagation implemented in existing CP solvers. To achieve this, the current DP state, the upper bound on the makespan, and the current decision are imposed as constraints within the CP model. Computing the fixpoint of this propagation yields tighter execution intervals for the tasks and a stronger lower bound on the makespan, ultimately reducing the number of transitions in the DP approach. Furthermore, CP can be used as a proxy model to tighten the lower bound via a dichotomic search based on feasibility checks at each state.

This hybrid DP-CP framework naturally models the PSSP. Empirical evaluations on standard benchmarks confirm the viability of this hybridization, demonstrating that our approach effectively exploits CP global constraints and is competitive with standard pure CP approaches in minimizing the number of explored nodes. Our contributions are summarized as follows:

- To improve anytime performance, we replace the layer-based DP search proposed in [7] with an *Anytime Column Search* (ACS) [18].
- We propose a hybridization of DP and CP models where the power of the global constraint NoOverlap plays a central role.
- The new framework natively handles the PSSP.
- First preliminary results are conducted on well-known suites of benchmarks.

The rest of the paper is organized as follows. Section 2 describes the state-of-the-art DP model while Section 3 presents our adaptation of the anytime column search for the layer-based DP. Section 4 details the proposed DP-CP hybrid framework for the PSSP. Section 5 provides a comprehensive empirical evaluation of the proposed approach. Finally, Section 6 concludes the paper.

## 2 Dynamic Programming Approach

A DP formulation for the JSP was originally proposed by Gromicho et al. [7], subsequently refined by Hoorn et al. [19], and reused by Ozolins [15]. In this approach, operations are scheduled by fixing their start times. An operation becomes eligible for scheduling only when all of its predecessors in the precedence graph have already been scheduled. This operation is then scheduled at its earliest possible start time, ensuring no temporal overlap on its required machine. A node in the search space represents a DP state, defined by the earliest possible start times of the remaining unscheduled operations. Because different sequences of scheduling decisions can lead to the exact same state, the search space forms a graph rather than a simple tree. The transition cost between two nodes corresponds to the incremental increase in the makespan of the partial solution. Hoorn et al. [19] explore this state graph layer by layer, where the shortest path through the graph yields the schedule with the optimal makespan. Formally the DP state is a triplet $S = \langle \psi, \Lambda, \mathit{done} \rangle$, where:

- $\psi$ tracks the earliest possible start time for every operation.
- $\Lambda$ records the machine ID of the last scheduled operation.
- $\mathit{done}$ represents the set of operations that have already been scheduled.

The state $S$ defines a partial sequence, $T_S$, with a current makespan denoted by $C_{\max}(S)$. Based on the eligibility rules, the set of currently available operations is mathematically specified as: $\varepsilon(S) = \{o \in O \setminus \mathit{done}(S) \mid \mathit{preds}(o) \subseteq \mathit{done}(S)\}$

To mitigate the exponential size of the search space, the authors introduced dominance rules. These rules allow the algorithm to safely discard dominated transitions and nodes while mathematically guaranteeing that at least one path to the optimal solution is preserved.

### 2.1 Transition Dominance

Given a complete solution and a start-time assignment, this solution can be obtained by successively selecting an operation from $\varepsilon(S)$ and scheduling it as early as possible without violating machine constraints, continuing until all tasks are scheduled. To break symmetries, a *transition dominance* rule is applied, ensuring that any specific solution is generated by only a single sequence of tasks. Formally, we define $\eta(S)$ as the subset of available operations that are legally permitted to expand $T_S$. An operation only qualifies for $\eta(S)$ if it strictly extends the makespan or resolves ties using $\Lambda$:

▶ **Definition 1.** *An operation $o \in \varepsilon(S)$ belongs to $\eta(S)$ if it satisfies one of the following conditions:*

- $\psi_o + p_o > C_{\max}(S)$
- $\psi_o + p_o = C_{\max}(S) \wedge m(o) > \Lambda(S)$

The *transition function*, denoted *Transition*, is defined by the operations in the set $\eta(S)$. Given a state $S$ and an operation $o \in \eta(S)$, the transition function returns a new state $S' = \langle \psi', \Lambda', \mathit{done}' \rangle$. First, the set of scheduled operations is updated: $\mathit{done}' = \mathit{done}(S) \cup \{o\}$. Second, the earliest starting times $\psi$ are updated to $\psi'$ using the *UpdateEst* function. This function updates the earliest start time of all unscheduled operations on the same machine as $o$ and for each operation $o'$ whose $\psi'_{o'}$ has been increased, it recursively updates its successors in the precedence graph $G$. Third, the ID of the last machine is updated: $\Lambda' = m(o)$. The transition function thus returns the new state $S' = \mathit{Transition}(S, o) = (\psi', \Lambda', \mathit{done}')$. From a state $S$ and an operation $o \in \eta(S)$, the *transition cost function* denoted $h$, associated with the transition is the makespan of the new state minus the makespan of the current state,

i.e., $h(S, o) = C_{\max}(S') - C_{\max}(S)$. The *domain function* of a state $S$, denoted $d(S)$, is the set of all operations that can directly extend the schedule. The size of this set represents the branching factor of our search tree. In the DP formulation of the JSP in [7, 19], it is set $d(S) = \eta(S)$ for all states $S$.

## 2.2 State Dominance

Another dominance rule used in [7] is based on the notion of the earliest completion time of the operation $o$ following the state $S$ denoted $\alpha_S(o)$ and specified by:

$$\alpha_S(o) = \begin{cases} \psi_o + p_o & \text{if } o \in \eta(S) \\ C_{\max}(S) + p_o & \text{otherwise} \end{cases} \tag{1}$$

▶ **Definition 2.** *[7] A state $S_1$ dominates a state $S_2$, denoted $S_1 \preceq S_2$ if*

$$\begin{cases} done(S_1) = done(S_2) \\ \alpha_{S_1}(o) \leq \alpha_{S_2}(o) \quad \forall o \in \varepsilon(S_1) = \varepsilon(S_2) \end{cases} \tag{2}$$

Another state dominance is proposed in [19]. A state $S$ is dominated when there is at least one machine with an unscheduled operation belonging to $\eta(S)$ and all operations of this machine in $\eta(S)$ cannot start before $C_{\max}(S)$. Such states can be discarded, since the available operation must be scheduled at its earliest start time. Formally a state $S$ is dominated if there is a machine $M_j$ and an operation $o \notin \eta(S)$ with $m(o) = M_j$ such that

$$\alpha_S(o') = C_{\max}(S) + p_{o'} \quad \forall o' \in \varepsilon(S) \quad \wedge \quad m(o') = M_j. \tag{3}$$

# 3 Anytime Column Search (ACS)

The DP model starts from the root state $\hat{r} = \langle (0, \ldots, 0), -1, \emptyset \rangle$. The solution space is then explored layer-wise from the root, as in decision diagrams [2], to apply and detect all state dominance rules. A first drawback of this approach is that the optimal solution (i.e., the shortest path in the transition graph) is discovered only at the very end, which leads to poor anytime behavior. A second drawback is that a branch-and-bound search would enable more effective state filtering by pruning nodes that are provably unable to improve the incumbent solution. To address these issues, we propose exploring the search space using the Anytime Column Search (ACS) [18], as depicted in Algorithm 1, rather than a layer-wise exploration.

In ACS, to limit the size of the search space, only a fixed number of nodes, denoted $W$, are expanded at each layer, similarly to a beam search. In [20], the lower bound of a state is computed with the Jackson Preemptive Schedule (JPS) [9]. The JPS works as follows. On a machine, at least one available operation with the minimum latest completion time is scheduled. This operation is processed either up to its completion or until a more urgent operation becomes available. The function $\mathit{LowerBoundJPS}(S)$ denotes the lower bound on the current state $S$ using JPS and taking the maximum JPS makespan across all machines. The priority used to select the nodes to expand in ACS is defined by $f(S) = \mathit{LowerBoundJPS}(S)$.

Algorithm 1 receives as input an instance $I$ of JSP or OSP and the parameter $W$, and provides the optimal solution within the time limit or an approximate solution otherwise. An array of empty queues of states (one queue per decision layer) sorted in increasing value of $f$ is initialized at Line 1. The first queue links to the decisions on the root state, is filled in the loop of Line 5 with the transition from the root and their domain $d(\hat{r})$. In the loop of

**Algorithm 1** **AnytimeColumnSearch**$(I, W)$

**Input:** An instance $I$ with $N$ operations and a maximal width $W$
**Output:** The optimal sequence or an approximation solution

```
1  for l = 1 to N do
2      X[l] ← ∅ ;                    // empty queue of states sorted according to f
3  ub ← horizon ;
4  r̂ ← ⟨(0, ..., 0), −1, ∅⟩;
5  foreach o ∈ d(r̂) do
6      X[1] ← X[1] ∪ {t(r̂, o)} ;
7  while X ≠ ∅ do
8      for l = 1 to N − 1 do
9          candidates ← ∅;
10         while |candidates| < W ∧ X[l] ≠ ∅ do
11             S ← dequeueFirst(X[l]);
12             if isNotDominated(S) ∧ LowerBoundJPS(S) < ub then
13                 candidates ← candidates ∪ {S} ;
14         for S ∈ candidates do
15             foreach o ∈ d(S) do
16                 S′ ← Transition(S, o) ;
17                 X[l + 1] ← X[l + 1] ∪ {S′};
18     while X[N] ≠ ∅ do
19         S ← dequeueFirst(X[N]) ;
20         if C_max(S) < ub then
21             ub ← C_max(S) ;
```

Line 7, the best $W$ non-dominated states of each layer are made in the loop of Line 8. The function *isNotDominated* at Line 12 checks if the state is not dominated by any other state of the same layer, i.e., $\forall S' \in X[l] \mid S' \not\preceq S$, as defined in Definition 2 or if it is not dominated according to the Equation (3). The loop at Line 15 fills the queue associated with the next layer with the transition states of the selected states of the previous decision layer and their domains. During the transition, the lower bound for the state will be computed. The loop of Line 18 checks if there exists an improving solution in the queue of the last decision.

## 4 Hybrid DP-CP Model for the PSSP

In the PhD thesis of A. van Hoorn [20], the idea of using a filtering algorithm called the *parallel head–tail adjustment* [3] is briefly mentioned. In the constraint programming community, this algorithm is more commonly referred to as a type of *edge-finding algorithm* [21]. Such algorithms allow the detection of precedence constraints. Van Hoorn informally explains that these precedences could be used to prune transitions further by allowing only expansions for which all predecessors are already in the partial solution [20].

Our contribution with respect to this idea is twofold:

- Use of full CP propagation. Instead of relying on the parallel head–tail adjustment, we invoke a CP solver implementing all the filtering algorithms from [21] for the NoOver-

**Algorithm 2** **InitializeCSP**$(S, ub)$

**Input:** A state $S$, an upper bound $ub$.
**Output:** A CSP $P = \langle \mathcal{I}, \mathcal{C} \rangle$ where $\mathcal{I}$ maps operations to interval variables and $\mathcal{C}$ is a set of constraints.

$\mathcal{I} \leftarrow \mathit{GenerateIntervals}(S, ub)$ ;
$\mathcal{C} \leftarrow \emptyset$ ;
**for** $(o', o'') \in \delta(S)$ **do**
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathit{StartBeforeEnd}(\mathcal{I}(o'), \mathcal{I}(o''))\}$;
**for** $j = 1$ **to** $m$ **do**
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathit{NoOverlap}(\{\mathcal{I}(o') \mid o' \in O \wedge m(o') = j\})\}$;
**for** $i = 1$ **to** $n$ **do**
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathit{NoOverlap}(\{\mathcal{I}(o') \mid o' \in O \wedge j(o') = i\})\}$;
**return** $\langle \mathcal{I}, \mathcal{C} \rangle$ ;

LAP constraint on each machine, including *overload checking*, *edge-finding*, *detectable precedences*, and *not-first/not-last* rules. Petr Vilím proved [21] that after reaching a fixpoint, all detectable precedences can be retrieved from the propagated time windows by verifying whether the earliest completion time of one activity exceeds the latest start time of another (denoted $\lessdot$ in the following). By computing tighter time windows through CP propagation and identifying additional precedence relations, the explored search space can therefore be further reduced.

- Formalization of the approach. We fully formalize this idea and provide a precise transition function based on calls to the CP solver, thereby turning the previously informal suggestion into a well-defined algorithmic framework applicable to any scheduling problem with general precedences.

## 4.1 Extended State and Update of the Domain Function

The state definition from [7] is extended to explicitly represent the set of precedences (both original from the problem and those dynamically discovered through CP time-window filtering). We denote by $S = \langle \psi, \Lambda, \mathit{done}, \delta \rangle$ the extended state definition, where $\delta(S)$ denotes the set of precedence constraints currently known. Given an operation $o \in O$, we denote by $\delta(S, o)$ the set of predecessors of $o$ in the precedence relations of $\delta(S)$, i.e., $\delta(S, o) = \{\, o' \in O \mid (o', o) \in \delta(S) \,\}$. At the root state, the set $\delta$ is initialized to $E$, the initial precedence graph of the problem.

The domain function $d(S)$ is adapted to take $\delta(S)$ into account. Only operations whose predecessors are all already scheduled in $\delta(S)$ can be appended, i.e., $\{o \in \eta(S) \mid \delta(S, o) \subseteq \mathit{done}(S)\}$. This restriction reduces the size of the domain function and, consequently, the branching factor and the size of the search space explored by the DP approach.

## 4.2 Transition with Constraint programming (CP)

The transition function relying on CP is given in Algorithm 3 and replaces the transition function in ACS at line 16. Given an extended state $S$, an operation $o \in \eta(S)$, and an upper bound on the makespan, it returns a new state $S'$ in which the set of precedences includes both the precedence induced by the decision to append operation $o$ to the sequence and the detectable precedences identified after CP filtering.

Internally, this function uses a CP model for our scheduling problem, as given in Algorithm 2, further restricted by the current state. This function returns a constraint satisfaction problem (CSP) as a pair $P = \langle \mathcal{I}, \mathcal{C} \rangle$ where $\mathcal{I}$ is a mapping from operations to their corresponding CP interval variables [11] and $\mathcal{C}$ is the set of CP constraints. CP interval variable ranges are initialized by the function *GenerateIntervals* as follows.

- For operations in $done(S)$, the start time is fixed to their starting time in the state.
- For operations in $\eta(S)$, the minimum start time is set to their earliest starting time in $S$.
- For operations in $\varepsilon(S) \setminus \eta(S)$, the minimum start time is set to $C_{\max}(S)$.
- For operations in $done(S)$, the end time is fixed to their starting time plus their duration.
- For operation in $O \setminus (done(S))$, the maximum end time is fixed to the $ub$.

The precedence constraints $\delta(S)$ and disjunction constraints for any machine and/or job are collected into a set $\mathcal{C}$ (Lines 5, 7).

**Algorithm 3** **TransitionCP**$(S, o, ub)$

**Input:** A state $S$, an operation $o$ and an upper bound $ub$.
**Output:** The new state $S'$.

```
1  ψ' ← UpdateEst(ψ, o), Λ' = m(o), done' = done(S) ∪ {o}, δ' ← δ(S);
2  S' ← ⟨ψ', Λ', done', δ'⟩;
3  P = ⟨I, C⟩ ← InitializeCSP(S', ub) ;
4  if Fixpoint(P) = Failure then
5      return None.
6  δ' ← δ(S);
7  foreach o' ∈ O do
8      foreach o'' ∈ O \ {o'} do
9          if I(o'') ⋘ I(o') then
10             δ' ← δ' ∪ {(o'', o')};
11 return S' = ⟨ψ', Λ', done', δ'⟩;
```

In Algorithm 3, after the initialization of the CSP at Line 3, the fixed point is computed at Line 4. The loop of Line 7 analyzes the interval at the fixed point to retrieve the detectable precedences to be added in $\delta'$. The value of $\psi'$ is updated with the function *UpdateEst* described in Section 2.1.

Notice that we do not update $\psi$ with the earliest starting time of the interval variables in $\mathcal{I}$ because we want to preserve the dominance rules defined in Section 2.2. If we were to update it, the computation of $\eta$ would become invalid.

## 4.3 Lower Bound With Constraint Programming

The lower bound is used in state-space exploration to reduce the search space, thereby speeding up and tightening the problem bounds. We adopt a lower bound similar to that in [4], which uses a dichotomic approach with filtering algorithms, denoted *LowerBoundCP*. This function replaces the function *LowerBoundJPS* at line 12 in the 1. It receives as input a state $S$ and an upper bound $ub$ and computes a lower bound from $S$ to be used in the anytime column search. A first lower bound is computed with the Jackson preemptive scheduling procedure. Then a dichotomic search is performed between this lower bound and the upper bound given as a parameter. The lower bound plays a central structural role in anytime column search (ACS) since it drives convergence, pruning, and the anytime behavior.

| Instances | State-of-the-art [7] | DP-JPS | DP-CP |
|---|---|---|---|
| ft06 | 30 409 | 195 | **176** |
| la01 | 63 170 946 | 1 689 | **239** |
| la02 | 80 862 876 | 28 826 | **241** |
| la03 | 50 910 284 | 11 742 | **242** |
| la04 | 68 208 819 | 24 010 | **234** |
| la05 | 40 229 147 | 686 | **239** |

**Table 1** Nodes explored: DP-CP vs DP-JPS vs state-of-the-art [7]

## 5 Experimental Results

In this section, we report preliminary results. All instances are available [1]. For the JSP, we use the benchmark suites of Fisher and Thompson [5] and Lawrence [13]. The implementation of the DP part of the approach uses an open-source Java version [17] and the CP part uses MaxiCP [16]. The computational experiments reported in this section were run on an Intel(R) Xeon(R) Platinum 8160 CPU with 314GB of RAM.

### Is an anytime column search necessary for the DP model ?

We compare the number of explored nodes of DP-JPS with the results reported from [7]. Despite the simplicity of the lower bound (*LowerBoundJPS*) coupled with ACS, the results reported in Table 1 clearly show that the number of explored nodes is substantially reduced by the anytime column search used to find and prove the optimality. This is expected because the state-of-the-art approach uses a breadth-first search, and the JPS-based lower bound enables ACS to control the search and guarantee the quality of the current solution.

### Does hybridization DP-CP help to improve the individual approaches?

The DP model, denoted DP-JPS, is compared to our DP-CP model. The DP-JSP model explores the search space with ACS but without CP in transition, and for the lower bounds computation (*LowerBoundJPS* is used). The models differ in the transition and lower-bound phases, where the CP is used in DP-CP to reduce the branching factor of the DP formulation. The Table 1 shows that our DP-CP model takes fewer nodes to find and prove the optimality

## 6 Conclusion

In this paper, we propose, for the first time, a hybridization of DP and CP for partial shop scheduling problems such as JSP and OSP. The CP was used as a subroutine to leverage global constraint propagation, and the DP serves as a primary search framework. The DP formulation uses the CP during the transition phase to reduce the branching factor of the search, and the CP is also used to compute a lower bound, which is useful for the ACS used to explore the DP search space. The preliminary results show that the hybridization of DP and CP seems to be a promising research direction for solving other combinatorial optimization problems.

[1] `https://github.com/ScheduleOpt/benchmarks/`

## References


1. Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2001.
2. David Bergman, André A. Ciré, Willem-Jan van Hoeve, and John N. Hooker. *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016.
3. Wolfgang Brinkkötter and Peter Brucker. Solving open benchmark instances for the job-shop problem by parallel head–tail adjustments. *Journal of Scheduling*, 4(1):53–64, 2001.
4. Jacques Carlier and Éric Pinson. An algorithm for solving the job-shop problem. *Management science*, 35(2):164–176, 1989.
5. H Fisher and GL Thompson. Job-shop scheduling rules. *Industrial Scheduling*, page 225, 1963.
6. Michael R. Garey, David S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.*, 1(2):117–129, 1976.
7. Joaquim A. S. Gromicho, Jelke J. van Hoorn, Francisco Saldanha-da-Gama, and Gerrit T. Timmer. Solving the job-shop scheduling problem optimally by dynamic programming. *Comput. Oper. Res.*, 39(12):2968–2977, 2012.
8. John N. Hooker and Willem Jan van Hoeve. Constraint programming and operations research. *Constraints An Int. J.*, 23(2):172–195, 2018.
9. James R Jackson. Scheduling a production line to minimize maximum tardiness. 1955.
10. Ryo Kuroiwa and J Christopher Beck. Domain-independent dynamic programming: Generic state space search for combinatorial optimization. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 33, pages 236–244, 2023.
11. Philippe Laborie and Jerome Rogerie. Reasoning with conditional time-intervals. In *FLAIRS*, pages 555–560. AAAI Press, 2008.
12. Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling - 20+ years of scheduling with constraints at IBM/ILOG. *Constraints An Int. J.*, 23(2):210–250, 2018.
13. S Lawrance. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques. *GSIA, Carnegie Mellon University*, 1984.
14. Imko Marijnissen, J Christopher Beck, Emir Demirović, and Ryo Kuroiwa. Domain-independent dynamic programming with constraint propagation. *arXiv preprint arXiv:2603.16648*, 2026.
15. Ansis Ozolins. Bounded dynamic programming algorithm for the job shop problem with sequence dependent setup times. *Oper. Res.*, 20(3):1701–1728, 2020.
16. Pierre Schaus, Guillaume Derval, Augustin Delecluse, Laurent Michel, and Pascal Van Hentenryck. MaxiCP: A Not So Mini Constraint Programming Solver. `http://www.maxicp.org/`, 2026.
17. Pierre Schaus and et al. Ddolib: A unified framework for solving dynamic programming models with decision diagrams and informed search. Technical report, UCLouvain (ICTeam) and CETIC (COAL), 2026.
18. Satya Gautam Vadlamudi, Piyush Gaurav, Sandip Aine, and Partha Pratim Chakrabarti. Anytime column search. In *AI 2012: Advances in Artificial Intelligence: 25th Australasian Joint Conference, Sydney, Australia, December 4-7, 2012. Proceedings 25*. Springer.
19. Jelke J. van Hoorn, Agustín Nogueira, Ignacio Ojea, and Joaquim A. S. Gromicho. An corrigendum on the paper: Solving the job-shop scheduling problem optimally by dynamic programming. *Comput. Oper. Res.*, 78:381, 2017.
20. J.J. {van Hoorn}. Phd-thesis, Vrije Universiteit Amsterdam, 2016. VU Vrije Universiteit.
21. Petr Vilím. Global constraints in scheduling. *Univerzita Karlova, Matematicko-fyzikální fakulta*, 2007.