New Hybrid Optimization Algorithms for Machine Scheduling Problems

Yunpeng Pan and Leyuan Shi

***Abstract*— Dynamic programming, branch-and-bound, and constraint programming are the standard solution principles for finding optimal solutions to machine scheduling problems. We propose a new hybrid optimization framework that integrates all three methodologies. The hybrid framework leads to powerful solution procedures. We demonstrate our approach through the optimal solution of the single-machine total weighted completion time scheduling problem subject to release dates, which is known to be strongly $\mathcal{NP}$-hard. Extensive computational experiments indicate that new hybrid algorithms use orders of magnitude less storage than dynamic programming, and yet can still reap the full benefit of the dynamic programming property inherent to the problem. We are able to solve to optimality all 1,900 instances with up to 200 jobs. This more than doubles the size of problems that can be solved optimally by the previous best algorithm running on the latest computing hardware.**

***Note to Practitioners*— In practice, production scheduling is a crucial task at the majority of modern manufacturing firms. It has direct impact on the efficiency of the firms' day-to-day operations. The paper addresses the issue of optimization algorithm design for scheduling problems. With a set of algorithmic building block at our disposal, we could form solution procedures that follow standard frameworks, or we could instead obtain more efficient hybrid algorithms that bring about the best of all worlds by reaping the synergy of different frameworks. Our computational results demonstrate unequivocally the power of the latter approach.**

***Index Terms*— dynamic programming; branch-and-bound; constraint programming; hybrid algorithms**

I. Introduction

Dynamic programming and branch-and-bound are two fundamental principles for developing algorithms that find optimal solutions to combinatorial optimization problems. The main idea of dynamic programming is to exploit the optimality in the substructures of an optimal solution. The optimal solution value is *recursively* defined with respect to optimal solution values of subproblems, and subsequently computed in a *bottom-up* fashion. The efficiency of dynamic programming stems from the *overlapping* of subproblems.

Branch-and-bound, on the other hand, solves a problem by dividing the solution space into a number of smaller subregions. This process, called *branching*, is in turn applied to subregions as well, and therefore, results in an expanding tree in which a node corresponds to a subset of solutions. The growth occurs at the frontier or *active nodes* of the tree; one active node is selected for branching at a time according the *search strategy*. Meanwhile, the *upper bound* registers the best solution value attained thus far, and for each node, a *lower bound* on the best objective value attainable at the node is computed using some relaxation scheme(s). A node is *fathomed* if the lower bound is not strictly less than the upper bound (called *cutoff*). For a particular problem or problem class, *dominance conditions* may be used to quickly fathom a node without computing the lower bound. Branch-and-bound terminates with an optimal solution when all actives nodes are fathomed.

Due to the difference in their problem-solving mechanisms, dynamic programming and branch-and-bound are often applied in isolation. Depending on what specific problem is addressed, one methodology may be more appropriate than the other. However, there are many important problems for which both methodologies show strength. This has inspired researchers to consider hybrid strategies. For instance, Morin and Marsten [20] use branch-and-bound cutoff to reduce the number of subproblems in dynamic programming for the traveling salesman problem (TSP) and the nonlinear knapsack problem. Other examples can be found in the literature on shortest path problems with side constraints (e.g., [1]), which appear as subproblems in vehicle routing problems [31]. Various "labeling" algorithms proposed for the shortest path problems are all based on the dynamic programming principle (see, also, [10], [12]).

Likewise, ideas from dynamic programming can be incorporated into branch-and-bound as well. For instance, in a survey on branch-and-bound methods, Lawler and Wood [18] also use the TSP to illustrate how branch-and-bound nodes, which correspond to distinct sequences of the same subset of cities, "collapse" into one due to the fact that they are (either optimal or suboptimal) solutions of the same subproblem from the perspective of dynamic programming.

In recent years, there has also been a surge of applications that utilize hybrid branch-and-bound/constraint programming strategies. Constraint programming [5] is a relatively new methodology for solving combinatorial optimization problems. It can solve an optimization problem by solving a series of decision (or constraint satisfaction) problems resulting from a dichotomy to locate the optimal objective value. More precisely, each decision problem is solved through an enumerative search that resembles the tree search of branch-and-bound, except that it involves no bounding and only seeks a feasible solution. However, the overall optimum-seeking process can be prohibitively computation-intensive. A hybrid approach is a better alternative, where individual techniques of constraint programming are embedded in a branch-and-bound framework. The most remarkable of these techniques are *deduction* and *constraint propagation*, which reduce domains of decision variables. These techniques often result in tighter relaxation at a branch-and-bound node and a greater chance of identifying high-quality feasible solutions. In addition, deduction and

Dr. Yunpeng Pan is now with CombineNet, Inc., Fifteen 27th Street, Pittsburgh, PA 15222 USA (email: ypan@combinenet.com). Prof. Leyuan Shi is with the ISyE department, University of Wisconsin-Madison, Madison, WI 53706 USA, and also with the Center for Intelligent and Networked Systems, Tsinghua University, Beijing, P. R. China (e-mail: leyuan@engr.wisc.edu).

constraint propagation, when applied iteratively in conjunction with dichotomy, can provide strong lower bounds.

In this paper, we develop a hybrid optimization framework for machine scheduling problems. It aims to integrate all three methodologies, i.e., dynamic programming, branch-and-bound, and constraint programming. Existing hybrid strategies, while having obtained promising results experimentally, have a serious shortcoming—i.e., their *ad hoc* nature. The lack of a systematic framework prevents us from reaping the full potential of the synergy. Unlike previous efforts, we not only establish the computational benefits of the framework through empirical study, but also present theoretical results that facilitate a deeper understanding about the contributing factors behind the computational improvements attained. Thus, we are able to develop much more powerful solution strategies than those previously proposed.

We illustrate our method using the single-machine total weighted completion time scheduling problem subject to release dates (denoted by $1|r_j|\sum w_jC_j$ in the literature [15]). This $\mathscr{NP}$-hard problem has attracted substantial interest from both the scheduling and the mathematical programming communities; e.g., [17], [4], [11], [28], [32], [33]. Hybrid algorithms are constructed and tested extensively. We are able to optimally solve a set of 1,900 problem instances with 20–200 jobs.

The remainder of the paper is organized as follows. §II gives a dynamic programming recursive formulation for the scheduling problem. §III develops new hybrid algorithms. §IV establishes some properties of the algorithms. §V reports on our computational experience with the proposed algorithms. Finally, some concluding remarks are made in §VI.

II. Problem Formulation

By adapting the notation from [20], we can describe the combinatorial optimization problem that we consider, with an additive objective function, using the following dynamic programming recursion:

$$f(x',t') = \min_{x,t,d}\{w(x,t,d) + f(x,t) | (x',t'') = \Gamma(x,t,d), t'' \le t'\}, \quad \text{for } (x',t') \in \Omega \setminus \{(x_0,t_0)\} \tag{1}$$

with the boundary condition $f(x_0,t_0) = z_0$. In this recursion, (x,t), (x',t'), and (x',t'') are state variables that designate different subproblems in dynamic programming; (x_0,t_0) is the initial state; Ω is the nonempty state space with $\Omega \subseteq \Theta \times \mathbb{R}$, where Θ is a finite set and $\mathbb{R}$ is the set of real numbers. d ($d \in D$) is a decision in the finite set D of decisions. $\Gamma : \Omega \times D \mapsto \Omega$ is the mapping that defines the state $\Gamma(x,t,d)$ resulting from applying decision d at state (x,t); $w(x,t,d)$ is the incremental cost incurred by the state transition, where $w : \Omega \times D \mapsto \mathbb{R}$ is the incremental cost function. $f(x,t)$ is the minimum cost at state (x,t) with the initial cost z_0 being given at state (x_0,t_0). Let $\bar{\Omega}$ be the set of final states. Then, the optimal value of the problem is defined as $f^* = \min\{f(x,t)|(x,t) \in \bar{\Omega}\}$, and $\Omega^* = \{(x,t) \in \bar{\Omega} | f(x,t) = f^*\}$ is the set of optimal final states. The objective is to identify a sequence of decisions, called a *policy*, such that, starting at the initial state, the successive application of these decisions leads to an optimal final state.

We assume that, for any fixed $x \in \Theta$ and any fixed $d \in D$ (with $(x,t') \in \Omega, \Gamma(x,t',d) \in \Omega$ for some t'), the incremental cost $w(x,t,d)$ is nondecreasing in t. Under this assumption, it can be readily shown that, with x being fixed, $f(x,t)$ is a nonincreasing step function of t with right-hand continuity for all $(x,t) \in \Omega$. Thus, it is adequate to only consider states (x,t) of which t is a "jump" point; for such t, we refer to $\langle t, f(x,t)\rangle$ as a *label*. For a particular $x \in \Theta$, there are generally more than one label. Note that the problem studied by [20] is a special case of (1) in which there are exactly one label; alternatively, we can view their problem as one whose states do not have a t-component. Without loss of generality, we also assume that each decision $d \in D$ is applied *at most* once (it might not be applied at all). For problems in which multiple applications of the same decision do occur, we simply expand decision set D by adding duplicates of the decision but with distinct designations. Lastly, it is assumed that $z_0 = 0$, since this constant does not affect the solutions.

The dynamic programming formulation (1) captures the essence of a variety of machine scheduling problems with *regular*, additive objectives. For instance, consider the single-machine total weighted completion time scheduling problem $1|r_j|\sum w_jC_j$. An instance of the problem consists of n 3-tuples $\{\langle r_j, p_j, w_j\rangle | j \in \mathscr{J}\}$, where $\mathscr{J} = \{1,\ldots,n\}$ is a set of jobs to be scheduled on a machine, and r_j, p_j, and w_j are the release date, the nonnegative processing time, and the associated nonnegative weight, respectively. Job $j \in \mathscr{J}$ is allowed to start at time r_j; once started, the job will occupy the machine exclusively for the duration of p_j without interruption. The completion time of job j is denoted by C_j. The objective is to find a schedule that minimizes the objective function $\sum_{j\in\mathscr{J}} w_jC_j$.

For this scheduling problem, we may define x as a subset of $\mathscr{J}$ to be processed first (regardless of the internal ordering), and t as such that all jobs in x are completed no later than t. Also, $(x_0,t_0) = (\emptyset, 0)$ and $z_0 = 0$. Decision d corresponds to an unscheduled job to be scheduled next. A policy corresponds to a job processing sequence. The transition mapping is $\Gamma(x,t,d) = (x \cup d, \max\{t, r_d\} + p_d)$, and the incremental cost incurred by the transition is $w(x,t,d) = w_d \cdot (\max\{t, r_d\} + p_d)$. Each final state $(x,t) \in \bar{\Omega}$ is of the form $(\mathscr{J}, t)$. Because $|\Theta| = 2^n$, there are no less than 2^n labels; generally, there are a lot more labels than 2^n. In practice, the maximal problem size that dynamic programming procedures normally can handle is somewhere between 20 and 30 jobs, depending on the amount of storage available.

Table I lists some additional notation assumed throughout the paper. Branch-and-bound notions, including branching, dominance conditions, lower bounds, and upper bounds, are readily extensible to dynamic programming. This is reflected in the notation $\Gamma_D(x,t)$, $\Gamma'_D(x,t)$, $l(x,t)$, and $u(x,t)$, respectively. Dynamic programming builds an optimal solution in a bottom-up fashion, which may be viewed as a branching process in the state space, whereas branch-and-bound typically works in the policy space. The notion of a subpolicy π generalizes that of a policy, and is defined with respect to

TABLE I
NOMENCLATURE

$s\|_t$	the t-component of state $s \in \Omega$; hence, $(x,t)|_t \equiv t$.
$\pi = d_1 \dots d_k$	a sequence of k decisions and is said to be a (valid) subpolicy with respect to state (x,t) if $(x_1,t_1) = \Gamma(x,t,d_1) \in \Omega, (x_2,t_2) = \Gamma(x_1,t_1,d_2) \in \Omega, \dots, (x_k,t_k) = \Gamma(x_{k-1},t_{k-1},d_k) \in \Omega$; by default, $(x,t) = (x_0,t_0)$.
$d \in \pi$	decision d participates in subpolicy π.
Π	the policy space comprising all subpolicies with respect to the initial state (x_0,y_0).
Π_f	the set of feasible subpolicies with respect to the initial state (x_0,y_0); $\Pi_f \subseteq \Pi$.
$\Gamma(x,t,\pi)$	the state reached when subpolicy π is applied at state (x,t), i.e., (x_k,t_k) above.
$\pi_1 \oplus \pi_2$	the concatenation of two subpolicies.
$w(x,t,\pi)$	the sum of incremental costs incurred when subpolicy π is applied at state (x,t).
$\Gamma_D(x,t)$	$= \{(x',t') = \Gamma(x,t,d) \in \Omega \| d \in D\}$, the set of states reachable from state (x,t) in one transition.
$\Gamma'_D(x,t)$	the remaining set of states in $\Gamma_D(x,t)$ after all dominance conditions are applied.
$\mathscr{U}$	the global upper bound on f^*.
$\mathscr{L}$	the best known lower bound on f^*.
$l(x,t)$	a lower bound on the costs of all subpolicies leading to a final state when applied to state (x,t).
$u(x,t)$	the incremental cost $w(x,t,\pi)$ incurred when a subpolicy π, found by a heuristic, is applied at state (x,t) to reach a final state.

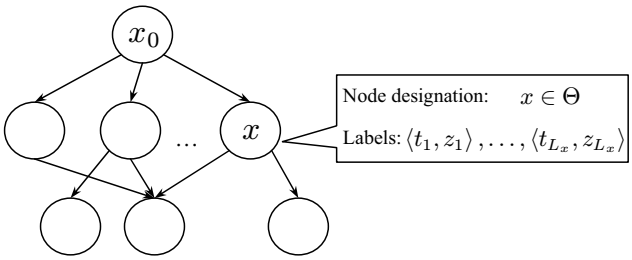


Fig. 1. Node Representation in the DP-BB Search Graph

some state $(x,t) \in \Omega$. The concatenation of two subpolicies is also straightforward. Subpolicy π is said to be *feasible* if, for some subpolicy π', $\Gamma(x,t,\pi \oplus \pi') \in \bar{\Omega}$ (i.e., the concatenation forms a policy that leads to a final state).

III. HYBRID ALGORITHMS

In this section, the individual algorithmic building blocks are developed and then integrated into various hybrid algorithms for finding optimal solutions to the problem.

III-A Hybrid Dynamic Programming/Branch-and-Bound

We present a hybrid dynamic programming/branch-and-bound (DP-BB) algorithm for solving Problem (1). In essence, DP-BB solves a problem by way of a graph search as depicted in Figure 1 (the graph is not a tree since a node can have more than one parent). The algorithmic elements are explained in what follows.

Node Representation. Associated with each node is an $x \in \Theta$ and its labels $\langle t_m^x, z_m^x \rangle, m = 1, \dots, L_x$, (where L_x is the number of labels; the superscripts are omitted whenever the dependency on x is unambiguous), each of which satisfies

$$z_m + l(x,t_m) < \mathscr{U}. \tag{2}$$

Let the node be designated by x. Without loss of generality, assume that the labels are ordered such that

$$t_1 < \dots < t_{L_x} \text{ and } z_1 > \dots > z_{L_x} \tag{3}$$

Search Strategy and Branching Scheme. The bottom-up nature of dynamic programming has in effect restricted the search strategy to breadth-first. For a given node x, the branching scheme is defined by $\Gamma'_D(x,t)$, which, by definition, takes into account various dominance conditions. To build the search graph $\mathscr{G}$, we use a first-in-first-out queue, Q, which is an ordered set, and also a mechanism, such as a hash function defined on Θ, that allows us to quickly access any node x and its associated list of labels. The hybrid DP-BB algorithm is outlined below.

Algorithm DP-BB($\mathscr{U}$)

Step 0. Initialization. Set $\mathscr{U} := \max\{\mathscr{U}, u(x_0,t_0)\}$. If $l(x_0,t_0) \geq \mathscr{U}$, *STOP*. Set $\mathscr{G} := \emptyset$, $Q := \{x_0\}$. Node x_0 has a single label $\langle t_0, z_0 \rangle$.

Step 1. If $Q = \emptyset$, *STOP*; otherwise, let x be the oldest node in Q and set $Q := Q \setminus \{x\}$. Let $\langle t_m, z_m \rangle, m = 1, \dots, L_x$ be the labels. Set $m := 0$.

Step 2. Set $m := m+1$. If $m > L_x$, go to 1; otherwise, set $S := \Gamma'_D(x,t_m)$.

Step 3. Branching. If $S = \emptyset$, go to 2; otherwise, let (x',t') be a state in S, and set $S := S \setminus \{(x',t')\}$. Let $\langle t', z' \rangle$ denote a candidate label, where $z' = z_m + w(x,t_m,d)$ and decision d satisfies $(x',t') = \Gamma(t_m,z_m,d)$.

Step 4. DP dominance. If node x' already exists in $\mathscr{G}$ and has a label $\langle t'', z'' \rangle$ such that $t'' \leq t', z'' \leq z'$, then label $\langle t', z' \rangle$ is fathomed; go to 3.

Step 5. Upper bounding. Set $\mathscr{U} := \min\{\mathscr{U}, z' + u(x',t')\}$.

Step 6. Lower bounding and cutoff. If $z' + l(x',t') \geq \mathscr{U}$, the label is fathomed; go to 3.

Step 7. Updating Q and $\mathscr{G}$. If node $x' \notin Q$, set $Q := Q \cup \{x'\}$. If node $x' \notin \mathscr{G}$, and add node x' along with label $\langle t', z' \rangle$ to $\mathscr{G}$; otherwise, suppose that the current list of labels associated with node x' is $\left\langle t_i^{x'}, z_i^{x'} \right\rangle, i = 1, \dots, L_{x'}$. Eliminate from the list, any label $\left\langle t_i^{x'}, z_i^{x'} \right\rangle$ with $t' \leq t_i^{x'}$ and $z' \leq z_i^{x'}$. Insert the label $\langle t', z' \rangle$ into the list such that the

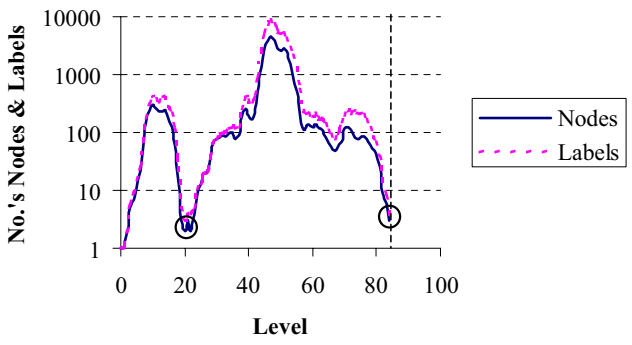


Fig. 2. Progression of the Search Tree for a 100-Job Instance

ordering in (3) is retained. Go to 3.

Figure 2 shows, for an instance of the $1|r_j|\sum w_jC_j$ problem with 100 jobs, how the numbers of nodes and labels change as the breadth-first search of DP-BB progresses from one level to the next (note that the vertical axis is log-scaled). The data is plotted up to level 84, after which the search terminates with an optimal solution. Compared to DP, this hybrid algorithm has significantly fewer states and labels, as a result of its branch-and-bound features, i.e., dominance conditions and cutoffs. We also note relatively small label-to-node ratios throughout the levels with an average of 1.65 and a maximum of 2.84.

The DP-BB algorithm takes full advantage of DP dominance, which contributes to reducing the size of the search graph. This very idea was first suggested by [20], who consider the TSP and a nonlinear knapsack problem, and later, adapted by [3] and [2] to solve two single-machine sequencing problems. However, this type of algorithm has one serious handicap: Breadth-first search has to be used. This diminishes the likelihood of obtaining a good upper bound early on during the search when the exploration is close to the root, and thus makes cutoff less effective. As a result, previous computational results obtained by this type of algorithm cannot compete with those by pure branch-and-bound that employs better search strategies. To overcome this difficulty, we propose two remedies.

III-B Branch-and-Bound with Partial Dynamic Programming Dominance

One solution to the upper bound handicap is to abandon breadth-first search altogether, and instead, to allow such search strategies as best-first and depth-first. These search strategies are more amenable to the goal of establishing tight upper bounds. Next, we describe how to incorporate DP dominance into branch-and-bound.

Recall that in a typical branch-and-bound algorithm for this type of problem, a node of the search tree corresponds to a subpolicy π. Let $(x,t) = \Gamma(x_0,t_0,\pi)$. If we can identify another subpolicy π' that leads to state (x,t') with (i) $t' \le t$ and $w(x_0,t_0,\pi') < w(x_0,t_0,\pi)$, or (ii) $t' < t$ and $w(x_0,t_0,\pi') = w(x_0,t_0,\pi)$, then the node that corresponds to subpolicy π is fathomed, which is a consequence of the dynamic programming property.

Such an idea first appeared in [18] in the context of solving the TSP, and since then, has been extensively discussed and exploited for machine scheduling by a number of authors including [13], [17], [4], and [25]. However, the dynamic programming property has been utilized only to a very small extent thus far. For machine scheduling, π' is normally sought by swapping the two most recently fixed jobs; see, e.g., [17], [4]. To expand the search for π', [25] propose to check the node against all previously visited nodes involving the same subset of jobs. Since they use depth-first search, all the nodes that need to be compared with the present one may not have yet been visited; as a result, their approach does not work very well in practice. Note that when used in conjunction with breadth-first search, their approach is equivalent to that of [20].

To identify such a π' that makes (i) hold, a recursion idea proposed by [21] for a static scheduling problem can be generalized and applied to our problem. Note that it suffices to solve a related subproblem in which the cost function is $f(x',t')$, as recursively defined in (1), and the set of final states is $\bar{\Omega} = \{(x',t') \in \Omega | x' = x, t' \le t\}$. By construction, (i) holds if and only if the optimal value f^* of the subproblem is less than $w(x_0,t_0,\pi)$. To this end, we can recursively apply the same branch-and-bound algorithm being developed for the original problem to the subproblem. For efficiency considerations, the level of recursion is limited to one. In actual implementation, the subproblem may be of the exact same form as the original one, or may be slightly different due to the extra constraint $t' \le t$.

It appears to be difficult to utilize (ii), since it requires solving a bicriterion subproblem. An even more daunting task would be to devise a method that eliminates one node between π and π' when both of the inequalities hold in equality. In this sense, the method in this subsection only partially exploits the dynamic programming property. Nevertheless, the method is still very effective as indicated by our computational experience.

III-C A Two-level Procedure Using Dichotomy and DP-BB

A second remedy for the weakness in the upper bound is inspired by the fact that we can make a good guess at the optimal value f^* and set $\mathscr{U}$ to the guessed value. One systematic way of making such guesses is dichotomy, which iteratively halves a given interval $[a,b]$ with $a \le f^* < b$. In the following dichotomy procedure, DP-BB is invoked repeatedly. We impose a space limit on DP-BB: A run of DP-BB will be aborted if the number of nonfathomed labels at any time exceeds M.

Algorithm Di-DP-BB(M)

Step 0. Initially, set a and b to the best known lower and upper bounds, respectively.

Step 1. If $a = b$, then set $\hat{f} := a$ and *STOP*. (In this case, the algorithm has failed to solve the instance to optimality, and $\hat{f}$ is the best lower bound attained.) Otherwise, set $y := \lceil (a+b)/2 \rceil$, $\mathscr{U} := y$, and run DP-BB($\mathscr{U}$). One of the following three outcomes will arise.

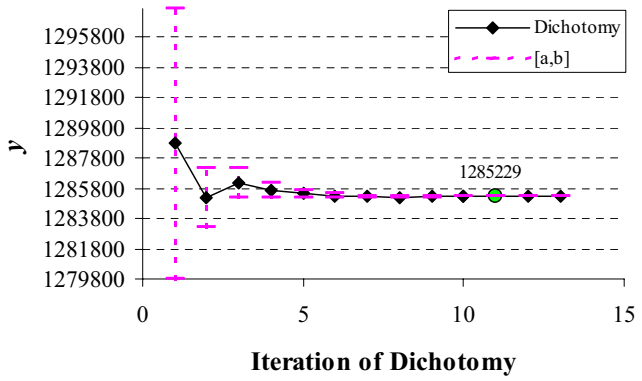


Fig. 3. Values of y during Dichotomy

Step 2. If DP-BB prematurely quits because of exceeding the M-label limit, it returns a possibly updated upper bound $\mathscr{U}$ and the best lower bound $\mathscr{L}$ established during the run. If $\mathscr{U} < y$, then set

$$a := \mathscr{L},\ b := \mathscr{U}; \tag{4}$$

otherwise, set $b := y - 1$ and a remains the same. Go to 1.

Step 3. If the algorithm terminates normally with $\mathscr{U} = y$, then set $a := y$ and go to 1.

Step 4. If the algorithm terminates normally with an improved upper bound $\mathscr{U}$ (i.e., $\mathscr{U} < x$), then the instance has been solved to optimality and $\mathscr{U}$ is the optimal value; hence, *STOP*.

The termination condition in Step 1 will be triggered if the space limit M is set too low to meet the requirements for running DP-BB with $\mathscr{U} = f^* + 1$. In this situation, we end up with a lower bound $\hat{f} \leq f^*$. To increase $\hat{f}$ or even attain optimality (via the termination condition in Step 4), the value of M has to be increased.

Figure 3 depicts the trajectory of Di-DP-BB when applied to the instance associated with Figure 2. Benefiting from the update (4) in Step 2, interval $[a, b]$ was often reduced by more than a half per iteration. For illustrative purposes, we intentionally used a small M ($M = 1,200$) to ensure that the procedure would exit through the termination condition in Step 1. Incidentally, we obtained a lower bound $\hat{f} = 1285229$, whereas the optimal value $f^* = 1285321$ can be attained if a slightly larger M is used.

Although constraint programming also uses dichotomy to find an optimal solution, there are fundamental differences between the two dichotomies. First of all, they have completely different motivations. The purpose of our dichotomy is to give the hybrid algorithm (DP-BB) a headstart with a tight upper bound. By contrast, the dichotomy of constraint programming is to convert an optimization problem into a series of decision problems, which ask for "yes" or "no" answers. Furthermore, we note that DP-BB tries to identify an optimal solution, whereas constraint programming employs a search algorithm that attempts to identify a feasible solution. Consequently, our dichotomy does not have to wait until the termination condition $a = b$ in Step 1 is met; it can also terminate when the condition in Step 3 is satisfied. As indicated by our computational experiments, the latter termination condition is often invoked.

As is true for any dichotomic process, there is a strong element of trial-and-error in the above procedure. This inevitably results in some waste, which perhaps is unjustifiable for easy instances. As will be shown in §V, most instances can be effectively dealt with by the branch-and-bound algorithm with partial dynamic programming dominance (§III-B). We would rather reserve Di-DP-BB as a last resort for cracking tough instances.

A further improvement of Di-DP-BB can be made by salvaging the work done during an iteration that later quits prematurely because of exceeding the space limit. Note that during the ensuing iteration, the search graph does not need to be grown from scratch; it resumes from the most recent level of nodes that have been completely expanded.

III-D Constraint Discovery and Propagation

The extensive use of constraint programming techniques in branch-and-bound algorithms can be traced back to the work by Carlier and Pinson [6]–[8] for the job shop scheduling problem with the objective to minimize the maximum job completion time. Some recent results include [27] and [30]. Unfortunately, the objective functions of problems that can be described by the recursion (1) are additive, and these techniques generally cannot be used for additive objectives. Therefore, we devote this subsection to developing new techniques for additive objectives.

III-D.1 General approach: We develop a general approach to constraint discovery and propagation for combinatorial optimization problems formulated as in (1). While our approach conforms to constraint programming principles manifested in many applications, it characterizes the fundamental principles at an abstract level and offers a new methodological perspective. Moreover, the constraint discovery techniques that we introduce apply to the entire class of problems, and therefore, can be used as recipes when specific problems are considered.

Constraints, which must be satisfied by all feasible solutions, are expressed *implicitly* in our problem formulation. Specifically, constraints are reflected in the state space Ω, the transition mapping Γ, and the definitions of a (valid) subpolicy and a feasible subpolicy. Validity means that subpolicy π, when applied at the initial state, leads to some state; i.e., $\Gamma(x_0, t_0, \pi) \in \Omega$. Given validity, feasibility means that there exists a subpolicy π' that further takes $\Gamma(x_0, t_0, \pi)$ to a final state. The goal of constraint discovery or *deduction* is to discover additional restrictions on Ω and Γ that reduces the set of potential branches $\Gamma_D(x, t)$ at a state (x, t), or results in a better relaxation and thus a tighter lower bound $l(x, t)$. In doing so, we need to ensure that at least one optimal solution to the original problem is retained. Apparently, dominance conditions fit this broad characterization of deduction techniques, since they result in $\Gamma'_D(x, t)$, which is a subset of $\Gamma_D(x, t)$.

Based on different motivations, we categorize deduction techniques as those driven by solution feasibility and those

driven by the objective function. Deduction techniques can also be categorized by the forms of constraints derived. Because there are no restrictions as to what form a derived constraint has to take, it would be impossible for us to enumerate all the possibilities; neither would it be necessary since the underlying ideas are the same. Therefore, we focus on the deduction of constraints in a particular form described next.

Consider a feasible policy π, which, by definition, satisfies $\pi \in \Pi_f$, $\Gamma(x_0, t_0, \pi) \in \bar{\Omega}$. Let Π_c denote the set consisting of all such π. For a decision $d \in D$, if $d \in \pi$, we can write $\pi = \pi_1 \oplus d \oplus \pi_2$, where $d \notin \pi_1, d \notin \pi_2$ (recall that d occurs at most once in any π); define $T(\pi, d) = \Gamma(x_0, t_0, \pi_1 d)|_t$. Thus, we can define, for each decision d, the minimal interval $[\alpha_d^*, \beta_d^*]$ such that for any $\pi \in \Pi_f$ with $d \in \pi$, we have $T(\pi, d) \in [\alpha_d^*, \beta_d^*]$. Such an interval is called a *feasibility-induced* interval. If additionally we are given a fixed value $\mathscr{U}$ meaning that only a solution with an objective value less than $\mathscr{U}$ is sought, then we may define an *objective-induced* interval as the minimal interval such that for any $\pi \in \Pi_f$ with $d \in \pi, w(x_0, t_0, \pi) < \mathscr{U}$, we have $T(\pi, d) \in [\bar{\alpha}_d^*, \bar{\beta}_d^*]$. Apparently, $[\bar{\alpha}_d^*, \bar{\beta}_d^*] \subseteq [\alpha_d^*, \beta_d^*]$. Neither interval is computable directly. However, we can use deduction techniques to obtain an interval $[\alpha_d, \beta_d]$ such that $[\bar{\alpha}_d^*, \bar{\beta}_d^*] \subseteq [\alpha_d, \beta_d]$ or $[\alpha_d^*, \beta_d^*] \subseteq [\alpha_d, \beta_d]$. We aim to derive intervals $[\alpha_d, \beta_d]$, $d \in D$ with as small lengths as possible. These intervals are used to reduce the size of $\Gamma'_D(x, t)$ to obtain a more restricted set $\Gamma''_D(x, t)$: For any $d \in D$, if $\Gamma(x, t, d)|_t \notin [\alpha_d, \beta_d]$, then state $\Gamma(x, t, d)$ is excluded from set $\Gamma''_D(x, t)$. Moreover, with the addition of these intervals, the relaxation usually becomes stronger, resulting in tighter lower bound $l(x, t)$. Our techniques are explained next using the $1|r_j|\sum w_j C_j$ problem as an example.

III-D.2 An illustration using $1|r_j|\sum w_j C_j$*:* The $1|r_j|\sum w_j C_j$ problem used in this illustration does not model job deadlines. However, $r_{\max} + \sum p_j$ can be assumed as an implied common deadline for the jobs [33], since it can be easily shown that there is no incentive to complete jobs after this time.

Feasibility-induced adjustments. For the total weighted completion time problem with both release dates and deadlines, finding a feasible schedule becomes a nontrivial issue. Relevant to this feasibility issue, a number of authors have studied a related problem, $1|r_j|\max L_j$, where $L_j = C_j - d_j$ is called the job *lateness* and d_j denotes a non-binding due date (e.g., Carlier Pinson [7], [8], [19]). In our particular situation, we are only interested in schedules with $L_j \leq 0$ for all j; because $C_j \leq d_j$, the due date d_j is in effect deadlines. Hence, many exact and inexact adjustment (or shaving) techniques can be applied to the release dates and deadlines. It is worth noting that computation-wise, these techniques are highly efficient.

Release dates. For a job $j_0 \in \mathscr{J}$, let us consider all those schedules in which j_0 starts exactly at time r_{j_0}. Given $\mathscr{U}$, if we can show that none of these schedules can improve $\mathscr{U}$, then r_{j_0} can be increased by one. Repeating this process yields an interval $[r_{j_0}, r'_{j_0})$ such that no schedule with j_0 starts in this interval can improve $\mathscr{U}$. If the interval is not empty, then we can set $r_{j_0} := r'_{j_0}$. We present two methods for computing r'_{j_0}.

The first method, which we refer to as the *increment* method, tries to increase the release date by one each time until the attempt fails. Consider a modified instance $\mathscr{P}_1$ with the weight w_{j_0} changed to a sufficiently large number w'_{j_0} such that job j_0 must start at r_{j_0} for any schedule to be optimal. With our notation, $z(\mathscr{P}_1)$ and $l(\mathscr{P}_1)$ denote the optimal value and a lower bound, respectively. Then, $\alpha_1 := l(\mathscr{P}_1) - (w'_{j_0} - w_{j_0})(r_{j_0} + p_{j_0})$ is a lower bound on the best objective value of the original instance $\mathscr{P}$, attainable with j_0 starting at r_{j_0}. Therefore, if $\alpha_1 \geq \mathscr{U}$, we can increase the release date by one.

It is easy to verify that we can set $w'_{j_0} := \max\{w_{\max}, r_{\max} + \sum p_j + p_{\max} - r_{j_0}\}$. Also, note that the increment method only entails the calculation of a lower bound for the modified instance, which is of the same form as the original one. The lower bound of [4] therefore can be used for this purpose. However, the main handicap of this method is that the computational cost depends on the time unit used. To alleviate this problem, we propose a second method based on dichotomy.

In the second method, which is based on dichotomy, we bound r'_{j_0} by an interval $[a, b]$. We initially set $a := r_{j_0}$ and $b := r_{\max} + \sum_{j \neq j_0} p_j$, and then reduce the interval iteratively through dichotomy until $a = b$. More precisely, during an iteration, we impose on a schedule the requirement that job j_0 must be completed by a deadline $\bar{d}_{j_0} := \lfloor (a+b)/2 \rfloor + p_{j_0}$. If we can determine with certainty that no schedule satisfying this requirement can improve $\mathscr{U}$, then j_0 must not start in the interval $[a, (a+b)/2]$. Hence, we set $a := \lfloor (a+b)/2 \rfloor + 1$. On the other hand, if such a determination cannot be made, then we set $b := \lfloor (a+b)/2 \rfloor$.

It is not difficult to see that the dichotomy method requires a means of evaluating a lower bound for a modified instance in which j_0 is constrained by a deadline. To the best of our knowledge, there are no lower bounding schemes customized for this particular type of problem. However, a lower bounding scheme has been developed by [22] for the more general problem $1|r_j, \bar{d}_j|\sum w_j C_j$, in which the jobs are constrained by both release dates and deadlines.

The insensitivity of the dichotomy method to the time unit can be used to our advantage. Therefore, we propose a two-phased release date adjustment. In Phase I, the dichotomy method is applied for each job $j \in \mathscr{J}$ so as to quickly attain the bulk of release date increase, and in Phase II, the new release dates from Phase I are further refined with the increment method. As our preliminary experiments show, Phase II is justified because it indeed results in substantial improvement. It is worth noting that Phase II is applicable only when the processing times and release times are integers.

Deadlines. Tighter deadlines may be induced by a given upper bound $\mathscr{U}$. Two methods for computing induced deadlines are proposed in the following.

In the first method, we relax the machine capacity constraint on job $j_0 \in \mathscr{J}$. This decomposes the original instance $\mathscr{P}$ into two sub-instances $\mathscr{P}_1$ and $\mathscr{P}_2$ with job sets $\mathscr{J} \setminus \{j_0\}$ and $\{j_0\}$, respectively. Since we aim at an objective value of at most $\mathscr{U} - 1$ and $l(\mathscr{P}_1) \leq z(\mathscr{P}_1) \leq \sum_{j \neq j_0} w_j C_j$, job j_0 must complete by $(\mathscr{U} - 1 - l(\mathscr{P}_1))/w_{j_0}$ in order for a schedule to

improve $\mathscr{U}$. Hence, we can impose a deadline $\bar{d}_{j_0}$ on job j_0 with

$$\bar{d}_{j_0} := \min\left\{r_{\max} + \sum p_j, \frac{\mathscr{U} - 1 - l(\mathscr{P}_1)}{w_{j_0}}\right\}.$$

The second method further reduces $\bar{d}_{j_0}$ through a dichotomy scheme, which is similar to the one used for refining release dates. In fact, the dichotomy in this case is even simpler because it turns out that we only need to compute a lower bound for a modified instance of the same form as the original one.

IV. Analysis of the DP-BB Algorithm

IV-A Completeness of Dynamic Programming Dominance

Compared to dynamic programming, the hybrid algorithm (DP-BB) has significantly fewer states and labels, because it employs branch-and-bound techniques. Furthermore, under a mild condition, we can show that the labels generated by DP-BB are among those generated by dynamic programming. More precisely, no labels in the final search graph upon the termination of DP-BB are dominated by any label generated in dynamic programming.

Definition 1 (Lower bound monotonicity): A lower bound $l : \Omega \mapsto \mathbb{R}$ is said to satisfy the *monotonicity* condition if

$$l(x,t) \leq w(x,t,d) + l(x',t') \tag{5}$$

for any (x,t), $(x',t') = \Gamma(x,t,d)$, and any $d \in D$; and

$$l(x,t) \leq l(x,t') \tag{6}$$

for any (x,t) and any $t' \geq t$.

When interpreted in the context of machine scheduling problems, the monotonicity condition essentially requires that the action of fixing a job in the first position of the sequence or imposing an additional earliest start time constraint on the jobs should not result in a decrease in subsequent lower bound values. Although not satisfied by all, this is an intuitive and mild requirement on many lower bounding procedures, and does not stipulate how tight a lower bound has to be. For instance, all linear programming relaxation-based lower bounds satisfy the monotonicity condition, since it amounts to fixing variables or adding constraints. Also, the condition is satisfied by two lower bounds suggested by [4], referred to as *simple split* and *general split*, for $1|r_j|\sum w_j C_j$ (the proof can be carried out using the same line of reasoning as used in the first half of the proof of Theorem 4.1 in [4]).

Theorem 1 (Completeness of DP Dominance): If the lower bound satisfies the monotonicity condition, then no label in the final search graph of DP-BB is dominated by a label generated by dynamic programming.

Proof: Assume for the sake of contradiction that in the final search graph of DP-BB, there exists a label $\langle t, z\rangle$ at a node $x \in \Theta$ that is dominated by a label $\langle t', z'\rangle$ in dynamic programming. In addition, suppose that

$$\langle t_0, z_0\rangle, \langle t_1, z_1\rangle, \ldots, \langle t_k, z_k\rangle = \langle t', z'\rangle \tag{7}$$

denotes the series of labels that start in label $\langle t_0, z_0\rangle$ (of the initial state) and eventually lead to the label $\langle t', z'\rangle$ in dynamic programming, and that

$$(x_0, t_0), (x_1, t_1), \ldots, (x_k, t_k) = (x, t')$$

are the states visited during the transitions.

It is clear that $\langle t', z'\rangle$ was not added to the search graph of DP-BB; otherwise, $\langle t, z\rangle$ would have been dominated, and as a result, would not have belonged to the final tree. This means that the series of labels in (7) was not realized during the run of DP-BB. Hence, it must be the case that one of the labels in (7) (say, $\langle t^i, z^i\rangle$) was fathomed due to cutoff; i.e.,

$$z_i + l(x_i, t_i) \geq \mathscr{U}. \tag{8}$$

Next, we analyze two cases.

Case I ($i = k$): Since $\langle t', z'\rangle$ dominates $\langle t, z\rangle$, we have $t' \leq t$ and $z' \leq z$. By the monotonicity assumption (specifically, (6)), it follows that $l(x,t) \geq l(x,t')$. Hence, $z + l(x,t) \geq \mathscr{U}$, which contracts the fact that label $\langle t, z\rangle$ was not fathomed.

Case II ($i < k$): By the monotonicity assumption (specifically, (5)), we have $z_{i+1} + l(x_{i+1}, t_{i+1}) \geq z_i + l(x_i, t_i) \geq \mathscr{U}$. Repeating this argument eventually leads to (8) holding for $i = k$, which allows us to draw the same contradiction as in Case I. This completes the proof. ■

Theorem 1 states that under the monotonicity condition, the hybrid algorithm (DP-BB) only explores nondominated labels in dynamic programming; in this sense, the exploitation of the dynamic programming property is complete. This is a fairly strong statement, considering that DP-BB generates only a tiny fraction ($\ll 1\%$) of the nondominated dynamic programming labels. On the other hand, even in situations where the lower bound does not satisfy the monotonicity condition, the dynamic programming property can still be computationally useful in reducing the number of nodes and labels.

IV-B The Minimal Search Tree

Another important property of DP-BB is that under some condition regarding the initial value of $\mathscr{U}$, the final search graph generated by this algorithm is never larger than that generated by *any* other branch-and-bound algorithm. To present this result, we begin with some clarification about the size of the search graph or search tree.

The search tree grows during the course of a branch-and-bound algorithm. Let us focus on the final search tree after the algorithm terminates. For a branch-and-bound algorithm, the *size* of the search tree is defined as the number of nodes (which correspond to subpolicies) in the search tree. For the hybrid algorithm DP-BB, the basic construct commensurate to a subpolicy is a label. We therefore define the *size* of the DP-BB search graph as the number of labels.

A branch-and-bound tree generally is influenced by the search strategy used. However, the following proposition indicates that this is not always the case. In fact, the proposition claims that it takes the same amount of "work" (i.e., the number of nodes visited) to disprove the existence of a better objective value than the initial one, regardless of what search

strategy is used. Meanwhile, it is worth noting that the choice of search strategy *does* affect the running space requirements of the algorithm. Evidently, the proof follows from the fact that the upper bound is never updated during the course of branch-and-bound.

Proposition 1: If the upper bound $\mathscr{U}$ is initially set such that $\mathscr{U} \leq f^*$, then the final search tree is the same for all search strategies.

The next result establishes the minimality of the search graph generated by DP-BB.

Theorem 2: Assume that the lower bound $l(x,t)$ satisfies the monotonicity condition. Additionally, assume that the same lower bound and dominance conditions are used by DP-BB and any other branch-and-bound algorithm. If the upper bound $\mathscr{U}$ is initially set such that $\mathscr{U} \leq f^*$, then DP-BB generates the minimal search graph among all branch-and-bound algorithms.

Proof: Since $\mathscr{U} \leq f^*$ initially, the search strategy used by an algorithm is of no consequence. The minimality of the DP-BB tree follows from the completeness of DP-BB in exploiting dynamic programming dominance. ■

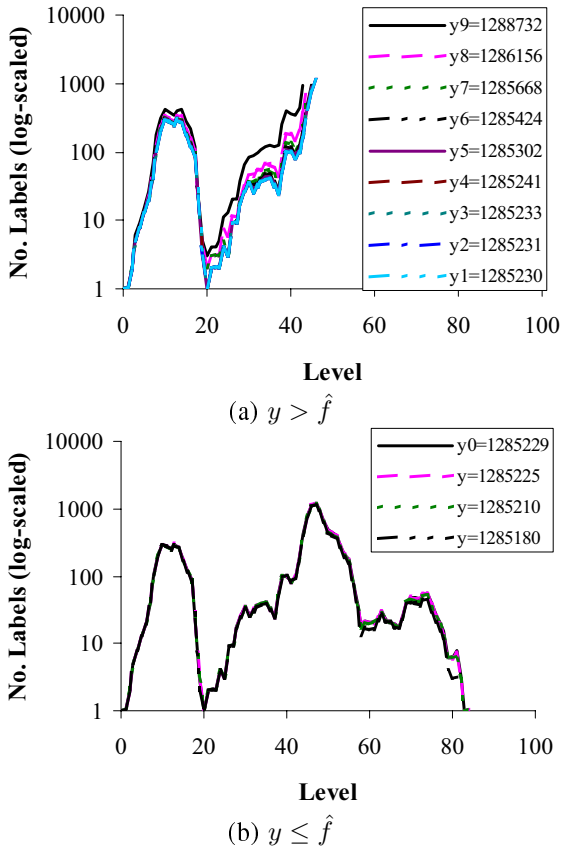
IV-C Estimation of Label Counts

We assume that the breadth-first search strategy is used. We also make the following simplifying assumption that will ensure a certain shape of the tree: For any state $(x,t) \in \Omega$, all the subpolicies $\pi \in \Pi$ with $(x,t) = \Gamma(x_0,t_0,\pi)$ consist of the equal number of decisions. This assumption implies that the search graph is *leveled*. Indeed, the assumption is satisfied by many problems including various machine scheduling problems. Under this assumption, the ith level of the minimal search graph is made up of nodes that take i transitions to reach from the initial state (x_0,t_0). We denote by N_i, $i = 0,\ldots,|D|$, the label counts at all levels of the minimal search graph (note that the maximal level is no more than $|D|$). We further expand the previous notation. For each value y, let $N_i(y)$, $i = 0,\ldots,|D|$ denote the label counts at all levels of the search graph by running DP-BB with initial upper bound $\mathscr{U} := y$. By the construction of DP-BB, we have $N_i(y) = 0$ $(i = 0,\ldots,n)$ for all $y \leq l(\mathscr{P})$.

Recall from §III-C that if M (the maximum number of labels allowed during DP-BB) is chosen too small, the dichotomy procedure will return the best lower bound $\hat{f}$, rather than f^*. We now take a closer look at the example depicted in Figure 3 with $\hat{z} = y_0 = 1285229$. Figures 4(a)(b) show the label counts for each y value visited during dichotomy; much overlapping is seen in (b).

As the first step, we try to estimate the labels counts for those y values in Figure 4(a), had their respective runs of DP-BB not been aborted due to the limitation of M. In particular, we focus on estimating the peak labels counts, which seem most likely to occur at level 47 (see Figure 5). We extrapolate $N_{47}(y_i), i = 1,\ldots,9$ using a ratio heuristic, which is based on the observation that the curves in Figures 4(a)(b) all are very similar in their shape. Take $N_{47}(y_9)$ as an example. We obtain $\hat{N}_{47}(y_9)$, an estimate of $N_{47}(y_9)$ by solving the following equation:

$$\frac{N_{43}(y_9)}{N_{43}(y_0)} = \frac{\hat{N}_{47}(y_9)}{N_{47}(y_0)},$$



(a) $y > \hat{f}$

(b) $y \leq \hat{f}$

Fig. 4. Label Counts $N_i(y)$

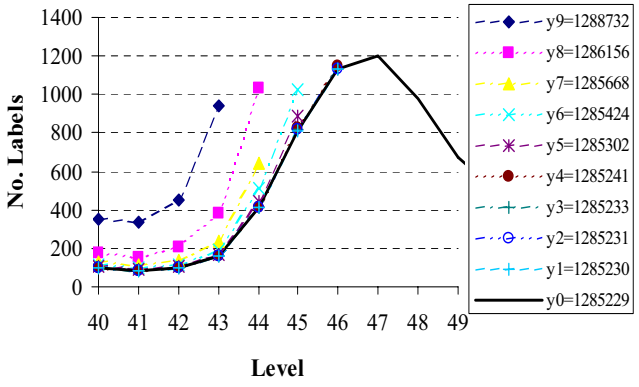


Fig. 5. Peak Label Counts $N_{47}(y), y \geq \hat{f}$

where 43 is the last *completed* level before DP-BB terminated. Hence,

$$\hat{N}_{47}(y_9) = \frac{N_{43}(y_9)N_{47}(y_0)}{N_{43}(y_0)} = \frac{941 \times 1200}{160} \approx 7058.$$

All the estimates are reported in Table II. For those y values that are close to y_0, their label estimates are fairly accurate.

Now, let us discuss how to choose M. Ideally, we wish to set $M := \|\mathscr{P}\|_s + 1$, because it is the smallest value that still allow f^* to be found by dichotomy. Unfortunately, neither f^* nor $\|\mathscr{P}\|_s$ is known in advance. Therefore, we propose the following *iterative 50/50 method* for adjusting M.

The gist of this method is to initialize M to a reasonably small value M_0, and then to gradually increase M toward

$\|\mathscr{P}\|_s + 1$ if M is too small. Specifically, M_0 was chosen to be 10,000 in our experiments (in our illustrative example, $M_0 = 1,200$). If it turns out for an instance that $M \geq \|\mathscr{P}\|_s + 1$, the dichotomy procedure will optimally solve the instance in a short time, due to small M_0. A more interesting scenario is when $M < \|\mathscr{P}\|_s + 1$, as occurred to the 100-job instance. In this case, we raise M just enough to ensure that the smaller 50% of the prematurely terminated dichotomy points will come through in the next round of dichotomy. For our example, we increase M to 1,308 so that y_5 would have a chance to come through; however, we will never know for certainty because 1,308 is just an estimate. The adjustment is repeated if the new M proves to be too small still.

V. Computational Results

In this section, we report on the results obtained by applying our hybrid algorithms to $1|r_j|\sum w_jC_j$. The previous best algorithm developed for solving the single-machine problem is due to [4]. We implemented this algorithm and used it as a basis for comparison.

V-A Implementation of Algorithms

Here we only outline the main components of the baseline algorithm, and refer the interested reader to the original paper by [4] for more details. First of all, the algorithm adopts a forward positional branching scheme. Under this scheme, a node at level i of the search tree corresponds to a partial sequence with jobs in the first i positions fixed; branching is realized by enumerating all possible assignments of the unsequenced jobs to the $(i+1)$th position.

The lower bound is based on a job-splitting idea first introduced by [24]. More precisely, two lower bounding procedures are suggested, referred to as simple split (SS) and general split (GS). SS runs in $O(n \log n)$ time and is a faster procedure than GS. On the other hand, GS, with $O(n^2)$ complexity, yields stronger lower bounds than SS.

To obtain an upper bound, the weighted-shortest-processing-time-first (WSPT) list schedule is computed at the root node before a depth-first search begins. In addition, the search tree is pruned by applying a number of dominance conditions.

Belouadah et al. [4] have experimented with six different configurations for their branch-and-bound algorithm. The best-performing variant employs the GS lower bound and a new dominance condition. In addition, the forward positional branching is enhanced with a *release date adjustment* (RDA) technique. We adopt this particular variant as the baseline.

The new hybrid algorithms, namely, DP-BB, BB-PDP, and Di-DP-BB, employ the same lower bounding procedure and dominance conditions as the baseline. In those experiments involving DP-BB, where the maximum number of labels does not exceed 200,000, we store, in addition to the basic data for a label, also the adjusted release dates obtained by the RDA technique.

In the BB-PDP algorithm, the partial DP dominance requires the solution of subproblems of the form $1|r_j, \bar{d}_j|\sum w_jC_j$, where $\bar{d}_j$ denotes a deadline for job j. This type of problem has been studied in [14], [22]. We solve the subproblems using the branch-and-bound algorithm developed in [22]. Two time-saving mechanisms are devised. One of them is to limit the size of the subproblem by only resequencing the η most recently fixed jobs, where the parameter η is called the *retrospect depth* and has been chosen to be 10 through experimentation. Also, the upper bound for the subproblem is initially set to be equal to the portion of the objective value associated with the fixed partial sequence, and branch-and-bound terminates immediately after this upper bound gets updated for the first time.

All the algorithms mentioned above were coded in C++, and were run on a Pentium IV 2.8GHz personal computer.

TABLE III
Test Instances Used in Experiments

Parameter	Values
n	20–200 in increments of 10
R	0.2, 0.4, 0.6, 0.8, 1.0, 1.25, 1.5, 1.75, 2.0, 3.0
$r_j \sim U[0, 50.5nR]^*, p_j \sim U[1, 100], w_j \sim U[1, 10]$	

* $U[a, b]$—the integer uniform distribution on interval [a,b].

V-B Experimental Design

We randomly generated 1,900 test instances with 20–200 jobs. The generation scheme that we followed is one suggested by [17] and also used by [4]. The scheme has many of the desirable characteristics described by [16], who give a number of guidelines for computational testing with randomly generated experimental data. Table III specifies the parameters and their values used by the scheme, as well as how release dates, processing times, and weights were set. A key parameter here is R, which is known as the *relative range of release dates*. For each combination of problem size n and parameter R, 10 instances were generated randomly; 1,900 instances were thus created. All these test instances, together with their optimal objective values found through this study, have been made available online at [23] to facilitate independent verification and to serve as benchmarks for the evaluation of exact and heuristic methods in the future.

V-C Results

The goal of our first experiment is to investigate the effectiveness of time window reduction via induced release dates and deadlines. For each of the instances, the reduced time windows $[r'_j, \bar{d}'_j]$ ($j \in \mathscr{J}$) were calculated and compared with the default time windows $[r_j, r_{\max} + P]$. The upper bound value, which is required by this technique, is set to the objective value of the WSPT list schedule. From Table IV, it is clear that the time window for a job can be reduced by 68–76% on average. Moreover, the required computational cost is small. Therefore, the time window reduction can be used as an effective preprocessing technique before the main algorithm is applied. For instance, in time-indexed mathematical programming formulations of scheduling problems (e.g., [33]), such preprocessing results in mixed integer programs with significantly fewer 0-1 variables.

In the second experiment, we demonstrate how partial DP dominance would improve the efficiency of branch-and-bound

TABLE II

ESTIMATION OF PEAK LABEL COUNTS $N_{47}(y)$

y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
Estimated $N_{47}(y)$	1201	1203	1206	1217	1308	1511	1885	3032	7058
Actual $N_{47}(y)$	1201	1203	1206	1217	1311	1550	2033	3612	8758

TABLE IV

TIME WINDOW REDUCTION VIA INDUCED RELEASE DATES AND DEADLINES

n	T_{avg}	$T_{\max}$	*Default*	*Reduced*	%	n	T_{avg}	$T_{\max}$	*Default*	*Reduced*	%
–	–	–	–	–	–	110	0.83	4.18	9006	2709	30
20	0.01	0.03	1562	386	25	120	1.10	7.12	9795	2966	30
30	0.02	0.08	2411	656	27	130	1.21	4.96	10618	3336	31
40	0.05	0.24	3233	790	24	140	1.58	6.52	11427	3451	30
50	0.09	0.35	4098	1138	28	150	1.87	8.97	12223	3832	31
60	0.16	0.84	4873	1371	28	160	2.34	10.10	13066	4172	32
70	0.22	0.77	5684	1620	29	170	2.72	13.13	13946	4393	32
80	0.35	1.18	6490	1880	29	180	3.41	18.51	14680	4560	31
90	0.52	2.08	7392	2165	29	190	3.99	23.38	15547	4584	29
100	0.65	3.07	8138	2558	31	200	4.01	21.02	16342	5125	31

T_{avg}, T_{max} = average and maximum CPU times in seconds; *Default* = default window size of a job; *Reduced* = window size after reduction; % = *Reduced*/*Default* $\times$ 100

algorithms. The baseline algorithm [4] is denoted by BB. We incorporate the partial DP dominance into the baseline to obtain BB-PDP. In this experiment, the problem size is limited to not exceeding 100 jobs. The CPU time limit per instance was set to two minutes, which is the same as in the experiments conducted by [4]; taking into account the fact that our computer is substantially faster than that used by those authors, the time limit here is in fact much higher in computer-independent terms. The results shown in Table V indicate that partial DP dominance can drastically improve the efficiency of a branch-and-bound algorithm in terms of number of unsolved instances and CPU time. The improvement can be attributed to fewer nodes in the search tree.

We also ran BB-PDP with the CPU limit set to 10 hours. BB-PDP were able to solve all but 72 of the 1,900 instances with 20–200 jobs. These remaining unsolved instances (with number of jobs ranging from 130–200 jobs) are indeed very challenging, and further increasing the CPU limit has little effect. Hence, more powerful methods are needed to tackle them.

The DP-BB algorithm is able to take advantage of complete DP dominance. When embedding it in a dichotomy procedure, we obtain the algorithm Di-DP-BB. In order to give a tight initial interval $[a, b]$ for dichotomy, we apply a modified version of BB-PDP which uses the best-first (or *backjump*) rule for node selection. The backjump rule is known to close the gap between upper and lower bounds quickly, nevertheless, at the expense of large storage requirements. We set the storage limit to 200,000 nodes, and the best-first BB-PDP algorithm always terminated within one hour, returning either the optimal solution value or the best upper and lower bounds. The gap between the upper and lower bounds is greater than zero for 212 instances, which consist of 90 jobs or more.

These 212 unsolved instances were used in a third experiment in which Di-DP-BB is applied with the maximum number of labels being 200,000. Within 10 hours, Di-DP-BB managed to solve 182 of these instances, whereas BB-PDP was able to solve 150 instances. Di-DP-BB also has superior performance in terms of CPU time. Note that in Table VI, the average CPU time T_{avg} for Di-DP-BB includes the time required for obtaining initial $[a, b]$.

For the remaining 30 unsolved instances, the number of labels simultaneously stored in memory at a certain point in time exceeds 200,000. Next, we re-ran Di-DP-BB with gradually increased space limits in succession of 400,000; 1,000,000; 1,500,000; 4,000,000; and 10,000,000; when the latter two limits were in use, release dates were not stored at any node for the conservation of memory space. In the end, optimal solutions were found for all the instances; the most difficulty instance took a little less than four days to solve to optimality.

VI. CONCLUSION

We have proposed new hybrid optimization algorithms for solving machine scheduling problems. The algorithms combine the three standard optimization methodologies (i.e., dynamic programming, branch-and-bound, and constraint programming) in a clever way so as to achieve greater efficiency. The extensive computational results yield strong evidence as to the value of these algorithms.

Since the initial completion of this work in 2003, there have been some new developments that are related to our topics here.

Cornuéjols et al. [9] suggest a method for estimating tree size that is different than our method discussed in §IV-C. Our method seems to give more accurate predictions of tree size, although more rigorous comparison is in order before we can be more conclusive.

Also dealing with the weighted completion time scheduling problem subject to release dates, Savelsbergh et al. [26] conduct an experimental study of linear-programming-based approximation algorithms recently suggested by other authors. With the exception of a set of instances with 30 jobs, they do not know the exact optimal values of the test instances used.

TABLE V
COMPARISON BETWEEN BB AND BB-PDP

n	NU		T_{avg}		ANN	
	BB	BB-PDP	BB	BB-PDP	BB	BB-PDP
20	0	0	0.00	0.00	33	23
30	0	0	0.01	0.01	210	71
40	0	0	0.05	0.02	1511	146
50	0	0	0.46	0.07	13650	366
60	1	0	2.58	0.20	29360	683
70	9	0	17.65	0.48	182561	1545
80	22	0	31.30	1.29	335824	3830
90	32	2	46.82	5.92	519635	15471
100	41	2	57.18	9.55	490675	22521

NU = number of unsolved instances; T_{avg} = average CPU time in seconds; ANN = average number of nodes in the search tree

TABLE VI
COMPARISON BETWEEN DI-DP-BB AND BB-PDP ON 212 HARD INSTANCES

n	NI	NU		T_{avg}		T_{max}	
		BB-PDP	Di-DP-BB	BB-PDP	Di-DP-BB	BB-PDP	Di-DP-BB
90	1	0	0	301	180	301	180
100	1	0	0	808	327	808	327
110	4	0	0	1342	160	2386	251
120	7	0	0	5680	379	15190	1001
130	12	1	0	6366	221	36000	399
140	12	0	0	4917	428	17107	1078
150	24	4	2	11274	1263	36000	10679
160	26	8	2	17755	2320	36000	22930
170	28	10	2	19044	2010	36000	12416
180	28	14	7	22734	4366	36000	28845
190	32	10	6	21999	3095	36000	19070
200	37	25	11	27274	6916	36000	36000

NI = number of instances; NU = number of unsolved instances; T_{avg}, T_{max} = average and maximum CPU times in seconds

Our published problem instances along with their optimal values should prove useful in future empirical evaluations of approximation algorithms for scheduling problems.

We should point out that the branch-and-bound algorithm that employs breadth-first search, suggested by T'kindt et al. [29], exploit the same dynamic programming dominance idea as our DP-BB algorithm. Complementing our computational results, their experimental study concerning three other scheduling problems have also shown the efficacy of the method.

Finally, we remark that our method is generic and can be readily adapted for other scheduling problems with regular objective functions. Take the total (weighted) tardiness objective as an example. Although a different lower bound scheme than the one used in this paper is needed, the organization of the algorithm essentially stays the same. With some additional effort, our method can be further extended to deal with problems with *nonregular* objective functions, as demonstrated in [34] on the earliness-tardiness scheduling problem.

ACKNOWLEDGMENTS

This research was supported in part by NSF grants DMI-0100220, DMI-0217924, and DMI-0431227, by AFOSR grant FA9550-04-1-0179, and by John Deere Horicon Works.

REFERENCES

[1] Y. P. Aneja, V. Aggarwal, and K. P. K. Nair, "Shortest chain subject to side constraints," *Networks*, vol. 13, pp. 295–302, 1983.

[2] J. F. Bard, K. Venkatraman, and T. A. Feo, "Single machine scheduling with flow time and earliness penalties," *Journal of Global Optimization*, vol. 3, no. 3, pp. 289–309, 1993.

[3] J. W. Barnes and L. K. Vanston, "Scheduling jobs with linear delay penalties and sequence dependent setup costs," *Oper. Res.*, vol. 29, no. 1, pp. 146–160, 1981.

[4] H. Belouadah, M. E. Posner, and C. N. Potts, "Scheduling with release dates on a single machine to minimize total weighted completion time," *Discrete Appl. Math.*, vol. 36, no. 3, pp. 213–231, 1992.

[5] A. Bockmayr and T. Kasper, "Branch and infer: A unifying framework for integer and finite domain constraint programming," *INFORMS J. Comput.*, vol. 10, no. 3, pp. 287–300, 1998.

[6] J. Carlier and E. Pinson, "An algorithm for solving the job-shop problem," *Management Sci.*, vol. 35, no. 2, pp. 164–176, 1989.

[7] ——, "A practical use of jackson's preemptive scheduling for solving the job shop problem," *Ann. Oper. Res.*, vol. 26, pp. 269–287, 1990.

[8] ——, "Adjustments of heads and tails for the job-shop problem," *Euro. J. Oper. Res.*, vol. 78, pp. 146–161, 1994.

[9] G. Cornuéjols, M. Karamanov, and Y. Li, "Early estimates of the size of branch-and-bound trees," *INFORMS J. Comput.*, vol. 18, no. 1, pp. 86–96, 2006.

[10] M. Desrochers and F. Soumis, "A generalized permanent labelling algorithm for the shortest path problem with time windows," *INFOR*, vol. 26, no. 3, pp. 191–212, 1988.

[11] M. E. Dyer and L. A. Wolsey, "Formulating the single machine sequencing problem with release dates as a mixed integer program," *Discrete Appl. Math.*, vol. 26, pp. 255–270, 1990.

[12] D. Feillet, P. Dejax, M. Gendreau, and C. Gueguen, "An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems," *Networks*, vol. 44, no. 3, pp. 216–229, 2004.

[13] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job Shop*. Chichester, West Sussex, England: Horwood, 1982.

[14] S. Gélinas and F. Soumis, "A dynamic programming algorithm for single machine scheduling with ready times," *Ann. Oper. Res.*, vol. 69, pp. 135–156, 1997.

[15] R. Graham, E. Lawler, J. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: A survey," *Ann. Discrete Math.*, vol. 5, pp. 287–326, 1979.

[16] N. G. Hall and M. E. Posner, "Generating experimental data for computational testing with machine scheduling applications," *Oper. Res.*, vol. 49, no. 6, pp. 854–865, 2001.

[17] A. M. A. Hariri and C. N. Potts, "Algorithm for single machine sequencing with release dates to minimize total weighted completion time," *Discrete Appl. Math.*, vol. 5, pp. 99–109, 1983.

[18] E. L. Lawler and D. E. Wood, "Branch-and-bound methods: A survey," *Oper. Res.*, vol. 14, no. 4, pp. 699–719, 1966.

[19] P. D. Martin and D. B. Shmoys, "A new approach to computing optimal schedules for the job-shop scheduling problem," *Lecture Notes in Computer Science*, vol. 1084, pp. 389–403, 1996.

[20] T. L. Morin and R. E. Marsten, "Branch-and-bound strategies for dynamic programming," *Oper. Res.*, vol. 24, no. 4, pp. 611–627, 1976.

[21] Y. Pan, "An improved branch and bound algorithm for single machine scheduling with deadlines to minimize total weighted completion time," *Oper. Res. Lett.*, vol. 31, no. 6, pp. 492–496, 2003.

[22] Y. Pan and L. Shi, "Dual constrained single machine sequencing to minimize total weighted completion time," *IEEE Trans. Automat. Sci. & Eng.*, vol. 2, no. 4, pp. 344–357, 2005.

[23] ——, "Test instances for the dynamic single-machine sequencing problem to minimize total weighted completion time," 2004, available online: http://www.cs.wisc.edu/~yunpeng/test/sm/dwct/instances.htm.

[24] M. Posner, "Minimizing weighted completion times with deadlines," *Oper. Res.*, vol. 33, no. 3, pp. 562–574, 1985.

[25] C. N. Potts and L. N. van Wassenhove, "A branch and bound algorithm for the total weighted tardiness problem," *Oper. Res.*, vol. 33, no. 2, pp. 363–377, 1985.

[26] M. W. P. Savelsbergh, R. N. Uma, and J. Wein, "An experimental study of LP-based approximation algorithms for scheduling problems," *INFORMS J. Comput.*, vol. 17, no. 1, pp. 123–136, 2005.

[27] F. Sourd and W. Nuijten, "Multiple-machine lower bounds for shop-scheduling problems," *INFORMS J. Comput.*, vol. 12, no. 4, pp. 341–352, 2000.

[28] J. P. Sousa and L. A. Wolsey, "A time indexed formulation of non-preemptive single machine scheduling problems," *Math. Program.*, vol. 54, pp. 353–367, 1992.

[29] V. T'kindt, F. Della Croce, and C. Esswein, "Revisiting branch and bound search strategies for machine scheduling problems," *J. Sched.*, vol. 7, no. 6, pp. 429 – 440, 2004.

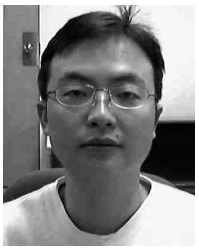
[30] P. Torres and P. Lopez, "On Not-First/Not-Last conditions in disjunctive scheduling," *Euro. J. Oper. Res.*, vol. 127, pp. 332–343, 2000.

[31] P. Toth and D. Vigo., *The Vehicle Routing Problem*, ser. SIAM Monographs on Discrete Mathematics and Applications. SIAM Publishing, 2002.

[32] J. M. van den Akker, C. A. J. Hurkens, and M. W. P. Savelsbergh, "Time-indexed formulations for machine scheduling problems: Column generation," *INFORMS J. Comput.*, vol. 12, no. 2, pp. 111–124, 2000.

[33] J. M. van den Akker, C. P. M. van Hoesel, and M. W. P. Savelsbergh, "A polyhedral approach to single-machine scheduling problems," *Math. Program.*, vol. 85, pp. 541–572, 1999.

[34] H. Yau, Y. Pan, and L. Shi, "New solution approaches to the general single machine earliness tardiness scheduling problem," *IEEE Trans. Automat. Sci. & Eng.*, 2006, conditionally accepted. Available online: http://www.optimization-online.org/DB_HTML/2006/03/1342.html.



Yunpeng Pan is an Algorithms and Formulations Architect with CombinetNet, Inc., based in Pittsburgh, PA. He received a Ph.D. in Industrial Engineering (2003) and an M.S. in Computer Sciences (2001) from University of Wisconsin-Madison, an M.S. in Operations Research from University of Delaware (1998), and a B.S. in Computational Mathematics from Nanjing University, China (1995). His research interests are concerned with developing industrial strength techniques and methods for solving difficult Mixed-Integer Programming problems that arise in E-Commerce, Combinatorial Auctions, Procurement (Reverse) Auctions, and Healthcare Informatics. His work appears in Mathematical Programming, Operations Research Letters, IEEE Trans. on Automation Science and Engineering, European Journal of Operational Research, and Journal of Systems Science and Systems Engineering. Dr. Pan is a member of IEEE, INFORMS, and the Mathematical Programming Society.

Leyuan Shi is a Professor with Department of Industrial and Systems Engineering at University of Wisconsin-Madison. She received her Ph.D. in Applied Mathematics from Harvard University in 1992, her M.S. in Engineering from Harvard University in 1990, her M.S. in Applied Mathematics from Tsinghua University in 1985, and her B.S. in Mathematics from Nanjing Normal University in 1982. Dr. Shi has been involved in undergraduate and graduate teaching, as well as research and professional service. Dr. Shi's research is devoted to the theory and applications of large-scale optimization algorithms, discrete event simulation and modeling and analysis of discrete dynamic systems. She has published many papers in these areas. Her work has appeared in Discrete Event Dynamic Systems, Operations Research, Management Science, IEEE Trans., and, IIE Trans. She is currently a member of the editorial board for Journal of Manufacturing & Service Operations Management, and is an Associate Editor of Journal of Discrete Event Dynamic Systems. Dr. Shi is a member of INFORMS and a senior member of IEEE.