INFORMS JOURNAL ON COMPUTING
Vol. 00, No. 0, Xxxxx 0000, pp. 000–000
ISSN 0899-1499 | EISSN 1526-5528 | 00 | 0000 | 0001

INFORMS
DOI 10.1287/xxxx.0000.0000
© 0000 INFORMS

Authors are encouraged to submit new papers to INFORMS journals by means of a style file template, which includes the journal title. However, use of a template does not certify that the paper has been accepted for publication in the named journal. INFORMS journal templates are for the exclusive purpose of submitting to an INFORMS journal and should not be used to distribute the papers in print or online or to submit the papers to another publication.

Solving Variants of the Job Shop Scheduling Problem through Conflict-Directed Search

Diarmuid Grimes
Cork Constraint Computation Centre, University College Cork, Ireland. d.grimes@4c.ucc.ie

Emmanuel Hebrard
LAAS; CNRS, Université de Toulouse, France. hebrard@laas.fr

We introduce a simple technique for disjunctive machine scheduling problems and show that this method can match or even outperform state of the art algorithms on a number of problem types. Our approach combines a number of generic search techniques such as restarts, adaptive heuristics and solution guided branching on a simple model based on a decomposition of disjunctive constraints and on the reification of these disjuncts.

This paper describes the method and its application to variants of the job shop scheduling problem (*JSP*). We show that our method can easily be adapted to handle additional side constraints and different objective functions, often outperforming the state of the art and closing a number of open problems.* Moreover, we perform in-depth analysis of the various factors that makes this approach efficient. We show that, while most of the factors give moderate benefits, the variable and value ordering components are key.

Key words: scheduling, combinatorial optimization

1. Introduction

Scheduling problems have proven to be fertile research ground for constraint programming and other combinatorial optimization techniques. Numerous such problems occur in industry, and whilst relatively simple in their formulation - they typically involve only *Sequencing* and *Resource* constraints - they remain extremely challenging to solve.

The most efficient methods for solving disjunctive scheduling problems like *Open shop* and *Job shop* scheduling problems are usually dedicated local search algorithms, such as

* We provide extended results of those already published in Grimes et al. (2009), Grimes and Hebrard (2010, 2011)

tabu search (Nowicki and Smutnicki (1996, 2005)) for job shop scheduling and particle swarm optimization (Sha and Hsu (2008)) for open shop scheduling. However, constraint programming often remains the solution of choice. It is relatively competitive (Beck (2007), Watson and Beck (2008), Malapert et al. (2008)) with the added benefit that optimality can be proven.

The best constraint programming (*CP*) models to date are those based on strong inference methods, such as *Edge-Finding* (Carlier and Pinson (1989), Nuijten (1994)), and specific search strategies, such as *Texture* (Fox et al. (1989)). Indeed, the conventional wisdom is that many of these problems are too difficult to solve without such dedicated techniques. After such a long period as an active research topic (over half a century since the seminal work of Johnson (1954)) it is natural to expect that methods specifically engineered for each class of problems would dominate approaches with a broader spectrum.

We show that this is not always the case. Our empirical study reveals that the complex inference methods and search strategies currently used in state-of-the-art constraint models can often, surprisingly, be advantageously replaced by a conflict-directed search strategy using the *weighted degree* heuristic (Boussemart et al. (2004)) on a simple model where unary resources are decomposed into a clique of pairwise disjunctions, hence without dedicated filtering algorithms.

Moreover, since this method does not use specific propagation methods, it can be applied without modification to a wide range of problems, i.e., all those that can be represented by a *disjunctive graph*. Therefore, with only minor adjustments, it can efficiently handle variants of the job shop scheduling problem involving side constraints such as maximal time-lag or sequence dependent setup times constraints. Similarly, changing the objective function, be it makespan minimization or minimizing penalties associated with early/late completion of jobs, does not seem to hinder its efficiency.

This paper integrate results from a series of publications. However, novel contributions of this paper include: a full comparison with a standard CP scheduling model implemented in IBM ILOG CP Optimizer (Laborie (2009)), a state-of-the-art commercial solver; updated results for JSP/OSP (the algorithm in our earlier publication did not contain solution guided value ordering); analysis of the impact of the different components of our algorithm; and new analysis of the behavior of constraint weighting on the different problem types.

In Section 2 we discuss the *disjunctive graph* representation of the unary resource scheduling problem. We introduce our basic model in Section 3, which is suitable to all problems that can be represented using a disjunctive graph, and we describe the different components of our search algorithm. Then we provide compelling empirical proofs of the benefits of our method compared with IBM ILOG CP Optimizer, and with state-of-the-art exact and approximate methods for some sample variants of the job shop scheduling problem in Section 4, extending results previously published in Grimes et al. (2009), Grimes and Hebrard (2010, 2011).

Finally, Section 5 provides a new analysis of the behavior of our algorithm to develop understanding of why it is effective on a number of problem types of this nature. In particular we provide a detailed evaluation of the different components in our algorithm, identifying those that are key to the performance, and we analyse the constraint weights produced on sample instances of the different problem types.

2. Disjunctive Scheduling

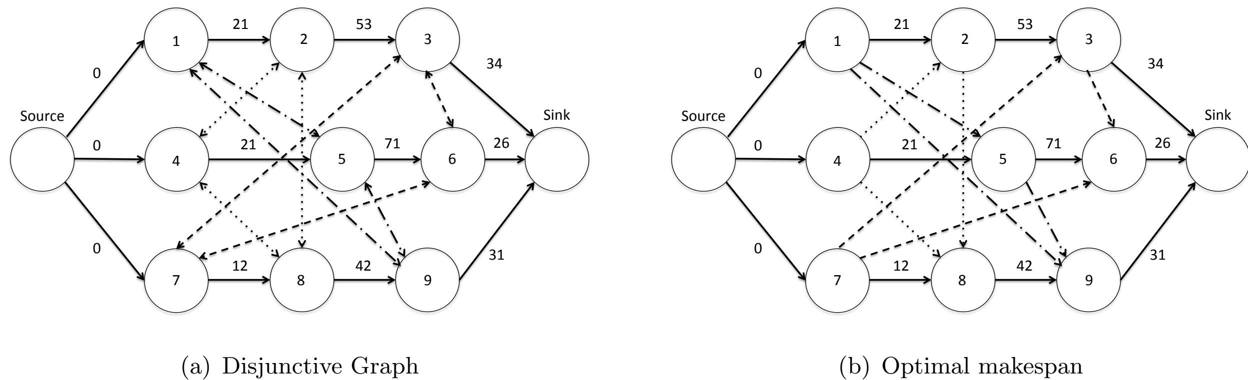
Disjunctive scheduling problems can be generally defined as the problem of scheduling n jobs $\mathcal{J} = \{J_1, ..., J_n\}$, on a set of m unary resources $R = \{R_1, ..., R_m\}$. A job J_i consists of a set of tasks $\mathcal{T} = \{t_1, ..., t_k\}$, where each task has an associated processing time and an associated resource on which it must be processed. A resource (also referred to as a machine) is exclusive and can process only one task at any time point.

These problems are commonly represented with a disjunctive graph (Roy and Sussman (1964)) $G = (N, D, A)$, where N is a set of nodes standing for tasks, D is a set of directed arcs standing for precedences, and A is a set of bidirectional (dashed) arcs standing for disjunctive constraints. The length of the arc is the duration of the task on the arc's tail.

A solution to the scheduling problem is an acyclic graph where all bidirectional arcs are replaced with directed arcs. The makespan, C_{max}, of the scheduling problem is the length of the *critical path* from the source node to the sink node, i.e. the longest path from the source to the sink. To find the optimal solution, one therefore needs to direct the bidirectional arcs such that the critical path is minimized.

We introduce a sample job shop scheduling problem to illustrate the disjunctive graph. We define a task as a triple $\langle i, p_i, m\rangle$ where i is an index for the task, p_i its processing time and m the machine it must be processed on. The sample problem involves three machines,

and the following three jobs: $[\langle 1,21,2\rangle, \langle 2,53,1\rangle, \langle 3,34,3\rangle]$, $[\langle 4,21,1\rangle, \langle 5,71,2\rangle, \langle 6,26,3\rangle]$ and $[\langle 7,12,3\rangle, \langle 8,42,1\rangle, \langle 9,31,2\rangle]$. Figure 1 presents the disjunctive graph representation of this problem, along with a sample graph of an optimal solution. The critical path of the optimal solution is: "Source" $\rightarrow$ 4 $\rightarrow$ 2 $\rightarrow$ 8 $\rightarrow$ 9 $\rightarrow$ "Sink", which gives optimal C_{max} of 147 (=0+21+53+42+31).



(a) Disjunctive Graph (b) Optimal makespan

Figure 1 Disjunctive Graph and optimal solution for sample 3x3 JSP.

2.1. Extensions

Using disjunctive graphs, one can easily express *job shop* and *open shop* problems. In both cases, some tasks share a common unary resource, which is represented as a clique of bidirectional arcs (in A). In the former problem, jobs are predefined sequences of tasks, represented as a path of directed arcs (in D). In the latter problem, jobs need to be sequenced, but any order is possible, so one can use a clique of bidirectional arcs, as is done for resources in both cases.

We consider the following additional constraint types which occur in many real-world scenarios. Firstly, *maximal time lag* constraints impose an upper bound on the time allowed between a task finishing and the next task of the same job starting. These can be expressed in the disjunctive graph using arcs of negative length. This type of constraint arises in many situations, for example in the chemical (Rajendran (1994)), steel (Wismer (1972)), and pharmaceutical industries (Raaymakers and Hoogeveen (2000)) to name but a few.

Secondly, there may be a setup time between two consecutive tasks for each machine, which is dependent on the order with which the two tasks are processed. This is referred to as the *sequence dependent setup times* constraint. Problems of this kind can be found in the semiconductor industry (Ovacik and Uzsoy (1997)).

Moreover, observe that temporal CSPs (Dechter et al. (1991)) can be relatively straightforwardly represented by disjunctive graphs. Therefore, the proposed approach is likely to be suited to this class of problems as well.

2.2. Constraint Programming notations

A constraint network is defined by a triplet $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ maps each variable x to finite sets of values $\mathcal{D}(x)$ and $\mathcal{C}$ is a set of *constraints*. A constraint C defines a relation, i.e. a set of allowed tuples of values, over a sequence of variables. More formally, a constraint C over $[x_1, \dots, x_n]$ is a subset of the cartesian product of the domains ($C \subseteq \prod_{i=1}^{n} \mathcal{D}(x_i)$). Given a tuple τ we denote $\tau[i]$ the value of the ith element of τ.

Moreover, we denote $\min(x)$ (resp. $\max(x)$) the minimum (resp. maximum) value currently in the domain $\mathcal{D}(x)$ of x. The constraint C is bounds consistent (BC) *iff*, for each variable x_i in its scope, there exist two tuples $\tau_1, \tau_2 \in C$ such that $\tau_1[i] = \min(x_i)$, $\tau_2[i] = \max(x_i)$ and for all $j \in [1, \dots, n]$, $\min(x_j) \leq \tau_1[j] \leq \max(x_j)$ and $\min(x_j) \leq \tau_2[j] \leq \max(x_j)$.

2.3. Traditional Constraint Programming approach (“Heavy Model”)

We first provide a brief overview of some of the main inference techniques and search heuristics developed for the non-preemptive machine scheduling problem. Modeling this type of problem as a constraint optimization problem (COP) is quite straightforward: the variables are the tasks, their domains are the possible starting times of the task, and the constraints specify relations between the tasks, e.g. no two tasks sharing a resource can overlap. The objective function is a criteria for deciding what the best schedule should be, e.g. minimize C_{max}.

2.3.1. Unary Resource Constraint Propagation Algorithms Let t_i represent task i. The *time-table* propagation rule (Le Pape (1994)) identifies time periods for which a resource *must* be used by a task. It computes the required resource usage for each time point k, maintaining a set of binary variables $X(t_i, k)$ which take the value 1 *iff* t_i must use the resource at time point k. Since resources are unary, the possible start time of a task t_j sharing a resource with t_i may be updated when conflicting with a time point where t_i must use the resource.

One of the most popular filtering techniques for the unary resource constraint is known as *Edge-Finding* (Carlier and Pinson (1989), Nuijten (1994)). Let T denote a set of tasks

sharing a unary resource, Ω denote a subset of T, and let $t_i \ll \Omega$ (respectively $t_i \gg \Omega$) denote that t_i must start before (*resp.* after) the set of tasks Ω, for $t_i \notin \Omega$. Edge-Finding involves detecting that a task must be scheduled first or last in the set of tasks $(\Omega \cup t_i)$. In other words, it infers precedences between t_i and each task in Ω. These precedences can then be used to prune the possible start times of the involved tasks.

The complement to the above is the “Not First, Not Last” filtering technique which detects that a task $t_i \notin \Omega$ cannot be scheduled first or last in the set of tasks $\Omega \cup t_i$, in which case the domain of t_i is updated accordingly (Baptiste and Le Pape (1996)).

The *Shaving* filtering technique (Carlier and Pinson (1994), Martin and Shmoys (1996)) updates the task time windows by assessing the earliest and latest start times. For each unassigned task t_i, at each node, it temporarily assigns t_i a starting time (either est_i or lst_i) and propagates the assignment using the filtering algorithms (Edge-Finding, etc.). If this results in a failure, it updates the relevant domain bound. The process iterates until a fixed point is reached. This can be viewed as a form of Singleton Bounds Consistency.

There are many variations of the constraints above , as well as further filtering techniques such as the *balance* constraint (Laborie (2003)). The reader is pointed to Baptiste et al. (2001), Laborie (2003) for further details on inference techniques for constraint-based scheduling, and the dissertation of Vilím (2007) for details and improvements to unary resource filtering techniques in particular.

2.3.2. Variable and Value Ordering Heuristics It is common to branch on *precedences* between two tasks sharing a resource. An unresolved pair of tasks t_i, t_j is selected and the constraint $st_i + p_i \leq st_j$ is posted on the left branch whilst $st_j + p_j \leq st_i$ is posted on the right branch.

Most heuristics are based on the identification of *critical tasks*. The branching scheme *Profile* introduced in Beck et al. (1997) (which is an extension of the *ORR/FSS* heuristic (Sadeh and Fox (1996))) involves selecting a pair of critical tasks sharing the same unary resource and ordering them by posting a precedence constraint on the left branch. The criticality is based on two texture measurements, *contention* and *reliance* (Sadeh (1991)). Contention is the extent to which tasks compete for the same resource over the same time interval, reliance is the extent to which a task must use the resource for given time points.

The heuristic determines the most constrained resources and tasks. For each task on each resource, the probability of the task requiring the resource is calculated for each

time period. This probabilistic profile is referred to as the individual demand curve. The contention is based on the aggregated demand over all tasks on the resource. At each node, the resource and the time point with the maximum contention are identified by the heuristic, then a pair of tasks that rely most on this resource at this time point are selected (provided the two tasks are not already connected by a path of temporal constraints).

Once the pair of tasks has been chosen, the order of the precedence has to be decided. For that purpose, a number of randomized value ordering heuristics have also been proposed by Beck et al. (1997), such as the centroid heuristic. The centroid of a task on a resource is based on the individual demand curve for the task on the resource, and is computed for the two critical tasks. The centroid of a task is the point that divides its probabilistic profile equally. If the centroids are at the same position, a random ordering is chosen. Finally, we note that a similar contention based approach has been proposed by Laborie (2005), based on the detection and resolution of *minimal critical sets (MCS).*

3. *Light Weighted* Approach (LW)

In this section, we describe the basic CP model and the search strategies that we propose. We consider as input a disjunctive graph and an objective function and define a constraint model. Unlike the standard CP approach, the model does not take further advantage of the problem structure, such as cliques of disjunctions corresponding to common resources. The search strategy relies to a large extent on methods developed for generic constraint programming, such as dichotomy, restarts, solution guided value ordering, and weighted degree variable ordering. However some of these methods have been slightly adapted to properties of disjunctive scheduling problems. Moreover we shall see that, in some cases, small adaptations for a specific variant of disjunctive problem may be useful.

Let $G = (N, D, A)$ be a disjunctive graph. We denote $\overrightarrow{l(e)}$ the integer labelling an arc $e \in D$, similarly, we denote $\langle \overrightarrow{l(e)}, \overleftarrow{l(e)} \rangle$ the pair of integers labelling an arc $e \in A$. Finally let K be some integer standing for an upper bound on the total makespan C_{max}.

- For each task $t_i \in N$, we introduce an integer variable $st_i \in [0, \ldots, K]$ standing for the start time of this task.
- For each directed arc $(t_i, t_j) \in D$ we introduce a precedence constraint $st_i + \overrightarrow{l((t_i, t_j))} \leq st_j$.
- For each arc $(t_i, t_j) \in A$ we introduce a Boolean variable b_{ij} which represents the relative ordering between t_i and t_j. A value of 0 for b_{ij} means that task t_i precedes task t_j,

whilst a value of 1 stands for the opposite ordering. The variables st_i, st_j and b_{ij} are linked by the following constraint: $b_{ij} = \begin{cases} 0 \Leftrightarrow st_i + \overrightarrow{l((t_i,t_j))} \leq st_j \\ 1 \Leftrightarrow st_j + \overleftarrow{l((t_i,t_j))} \leq st_i \end{cases}$

3.1. Solving Method

We first describe the methods and strategies used by the solver to either find a solution of this model or prove that none exists, given a constraint model as shown above,. In particular we detail the type of constraint propagation, variable and value heuristics as well as some additional features of the strategy.

Then, in Section 3.2, we show how this is used in the context of our general method.

3.1.1. Constraint Propagation We maintain bounds consistency on the constraint network. All constraints have a fixed arity and can be made BC in constant time by applying simple rules. In particular, achieving BC on a precedence $st_i + p_i \leq st_j$ can be done by setting the value of $\min(st_j)$ to $\max(\min(st_j), \min(st_i) + p_i)$ and conversely $\max(st_i)$ to $\min(\max(st_i), \max(st_j) - p_i)$. Given a disjunctive constraint on st_i, st_j and b_{ij}, there are two cases. First, if the domain of b_{ij} is a singleton, the constraint is now a precedence, hence the procedure explained above applies. Otherwise, we detect if the domain of b_{ij} *should* be reduced. This is the case if $\min(st_i) + p_i > \max(st_j)$, or if $\min(st_j) + p_j > \max(st_i)$, in which case the value of b_{ij} must be 0 or 1, respectively. Otherwise, it can be shown that every value in the domains of st_i, st_j and b_{ij} are bounds consistent.

Applying BC on this constraint network enforces a very basic level of consistency. It is denoted *arc-B-consistency* in Baptiste and Le Pape (1995), and shown to be less efficient than more complex propagation methods such as Edge-Finding, albeit using different search strategies than the one proposed in this paper. Given a problem defined by a disjunctive graph (N, D, A), achieving BC on the COP is sufficient to ensure that, for any task t_i, $\min(st_i)$ is the length of the longest directed path between the source and t_i in the graph (N, D). Similarly, $\max(st_i)$ is equal to the length of the maximum allowed makespan minus the length of the longest directed path between t_i and the sink in the graph (N, D).

For a job shop problem with n jobs and m machines, this model involves $nm(n-1)/2$ Boolean variables (and as many ternary disjunctive constraints). Constraint propagation works as follows: when the domain of a Boolean variable b_{ij}, or of an integer variable st_i changes, for instance because of a decision or because establishing BC on a constraint

tightened its bounds, every constraint involving this variable is *triggered* and put on a stack. At each step we pick a constraint in this stack and achieve BC on it in constant time, possibly triggering further constraints. In the worst case, each constraint may shave the domains of the integer variable by 1. Therefore, this process is guaranteed to reach a fixed point in time $O(C_{max} * nm(n-1)/2)$. However, it rarely reaches this bound in practice.

3.1.2. Variable Ordering In our model, we simply branch on the Boolean disjunct variables, which precisely simulates the strategy of branching on precedences discussed earlier, and thus significantly reduces the search space. We use the domain/weighted-degree heuristic (Boussemart et al. (2004)) which chooses the variable with minimum ratio of domain size to sum of weight of neighboring constraints, initialized to its degree. A constraint's weight is incremented by one each time the constraint becomes unsatisfied during constraint propagation.

However, at the start of the search, this heuristic is completely uninformed since every Boolean variable has the same domain size and the same degree (i.e. 1). We therefore used the domain size of the two tasks t_i, t_j associated to every disjunct b_{ij} to alleviate this issue. The domain size of task t_i (denoted $dom(t_i)$) is the number of possible starting times of t_i, i.e. $dom(t_i) = (lst_i - est_i + 1)$ (we assume that domains are discrete).

With regard to the weighted component of the heuristic, there are a number of ways to incorporate failure information. We focused on the following two methods. In the first, we use the weight $(w(i,j))$ on the Boolean variable alone, i.e. the number of times search failed while propagating the constraint between st_i, st_j and b_{ij}. The heuristic then chooses the variable minimizing the sum of the tasks' domain size divided by the weighted degree:

$$\frac{dom(t_i) + dom(t_j)}{w(i,j)} \tag{1}$$

Our second method uses the weighted degree associated with the task variables instead of the Boolean variable. Let $\Gamma(t_i)$ denote the set of tasks sharing a resource with t_i. We call $w(t_i) = \sum_{t_j \in \Gamma(t_i)} w(i,j)$ the sum of the weights of every ternary disjunctive constraint involving t_i. Now we can define an alternative variable ordering as follows:

$$\frac{dom(t_i) + dom(t_j)}{w(t_i) + w(t_j)} \tag{2}$$

We refer to the two heuristics above as $Tdom/Bwt$ and $Tdom/Twt$ respectively, where $Tdom$ is the sum of the domain sizes of the tasks associated with the Boolean variable, and Bwt (Twt) is the weighted degree of the Boolean (associated tasks *resp.*). Ties were broken randomly.

The relative light weight of our model allows the search engine to explore many more nodes, thus quickly accruing information in the form of constraint weights. It is important to stress that the behaviour of the weighted degree heuristic is dependent on the modeling choices. Indeed two different, yet logically equivalent, sets of constraints may distribute the weights differently.

3.1.3. Value Ordering Our value ordering is based on the solution guided method (SGMPCS) proposed for JSPs (Beck (2007)). This approach uses previous solutions as guidance for the current search, intensifying search around a previous solution in a similar manner to the tabu search algorithm of Nowicki and Smutnicki (2005). In SGMPCS, a set of elite solutions is initially generated. Then, at the start of each search attempt, a solution is randomly chosen from the set and is used as a value ordering heuristic (i.e. where possible each selected variable is assigned the value it took in the chosen solution). When an improving solution is found, it replaces the solution in the elite set that was used for guidance.

The logic behind this approach is its combination of intensification (through solution guidance) and diversification (through maintaining a set of diverse solutions). Note that this is a generic technique that can be applied to both optimization and satisfaction problems (Heckman (2007)). In the latter case, partial solutions are stored and used to guide subsequent search.

Interestingly, Beck found that the intensification aspect was more important than diversification for solving JSPs. Indeed, for the instances studied, there was little difference in performance between an elite set of size 1 and larger elite sets (although too large a set did result in a deterioration in performance). We therefore use an elite set of 1 for our approach, i.e. once an initial solution has been found this solution is used, and updated if improved, throughout search.

Furthermore, up until the first solution is found, we use a value ordering working on the principle of best *promise* (Geelen (1992)). The value zero for b_{ij} is visited first *iff* the domain reduction directly induced by the corresponding precedence ($st_i + p_i \leq st_j$) is

less than that of the opposite precedence ($st_j + p_j \leq st_i$). We use a static value ordering heuristic for breaking ties, based on the tasks' relative position in their jobs. For example, if t_j is the fourth task in its job and t_i is the sixth task in its job, then the value zero for b_{ij} sets the precedence $st_j + p_j \leq st_i$.

3.1.4. Additional Features The following additional features are used during dichotomic search and during branch and bound search. We use a geometric restarting strategy (Walsh (1999)) which is a sequence of search cutoffs of the form $s, sr, sr^2, sr^3, \ldots$ where s is the base and r is the multiplicative factor. In our approach we defined the cutoff in terms of failures, where the base was 256 failures and the multiplicative factor was 1.3.

We also incorporate the nogood recording from restarts strategy of Lecoutre et al. (2007), where nogoods are generated from the final search state when the cutoff has been reached. In other words, a nogood is stored for each right branch on the path from the root to the search node at which the cutoff was reached.

In order to make the results reproduceable, we implemented the cutoff for each dichotomic step in terms of propagations since the vast majority of constraints are the same (ternary disjuncts). The observed propagation speed for a subset of problems was approximately 20 000 000 calls to a constraint propagator (achieving BC on a disjunctive or precedence constraint) per second, the cutoff was then calculated by multiplying this by 30 (as this would generally result in a time cutoff of 30 seconds).

3.2. General Method

The general method ("*LW*") used on all problem types is given in Algorithm 1. The procedure `DisjunctiveGraph` returns a disjunctive graph as well as a set of constraints to link a distinguished variable representing the objective to the rest of the network. For instance, when the objective is the makespan, the objective variable is simply constrained to be larger than or equal to the completion time of every task.

For the *earliness/tardiness* cost objective we use a set of Boolean variables constrained to be *true* if and only if the last task of each job is early, and another such set standing for lateness. Moreover, we have a weighted sum constraint to link the objective variable to a sum of terms, each involving such a Boolean variable, the start time of the last task of a job, and the constant penalty associated with earliness or lateness.

Computing the disjunctive graph is straightforward for all considered variants of the JSP, except for the JSP variant involving *no-wait* constraints (NW-JSP) where there is no

time lag allowed between a task finishing and the next task of the job starting . Indeed, in this case, we have shown that it is possible to reduce this graph by merging the nodes corresponding to the tasks of a single job (Grimes and Hebrard (2011)), and it was this model that was used for the NW-JSP instances.

Algorithm 1: *LW*(`Problem`: P, `Var Order`: varh, `Val Order`: valh, `Cutoff`: t1, t2, t3))

```
start ← time.now; lwP ← Model(DisjunctiveGraph(P))
(bestsol, lb, ub) ← Initialize(lwP)
dicho_lb ← lb
while dicho_lb < ub − 1 do
    obj ← (dicho_lb + ub)/2
    result ← Solve(lwP,varh,valh,bestsol,obj,t2)
    if result.sat then ub ← result.obj; bestsol ← result.solution
    else if result.unsat then dicho_lb ← lb ← obj
    else dicho_lb ← obj
if (lb = ub − 1) then return (true, ub, bestsol )
else return BnbSolve(lwP,varh,valh,bestsol,ub,lb,t3 + start − time.now)
```

Algorithm 2: *Initialize*(`Problem`: `lwP`, `Cutoff`: t1)

```
ub ← Initub(lwP)
if (P.type = tljsp) then ub ← Jtlgreedy(lwP)
result ← Solve(lwP,varh,valh,bestsol,ub,t1)
if result.sat then return (result.solution, Initlb(lwP), result.obj)
else return (∅, Initlb(lwP), ub)
```

3.2.1. Initialization The function *Initialize* (outlined in Algorithm 2) initializes the lower and upper bounds. For makespan minimization, the lower bound was set to the maximum job/machine duration, while the initial upper bound was obtained with a randomized insertion method called 1000 times for most problem types. However, this simplistic method

is not guaranteed to produce a feasible solution for problems involving time lag constraints, namely the NW-JSP and the variant of the JSP involving general time-lag constraints (TL-JSP) . The upper bound for these two problem types was set to the trivial upper bound of the sum of the durations of all tasks, i.e. start each job only after the preceding job has finished. The lower bound for minimization of earliness/tardiness penalties was simply set to 0, while the upper bound was computed as the sum of maximum earliness costs for each job, plus the cost of each job finishing late based on a makepsan computed in the same way as for makespan minimization.

For the upper bound we added two further steps based on observations of search behavior. Firstly, the initial upper bound on job shop problems with time-lags is extremely poor. Indeed, time lag constraints make it difficult to find a feasible solution even if no restriction is put on the makespan. Moreover, we observed that for a few large instances, when it is not given an initial upper bound, our method may take a long time to converge for the general time-lag variant of the job shop. To remove this issue we implemented a procedure to initialize the upper bound that greedily explores the set of jobs $\mathcal{J}$ (Grimes and Hebrard (2011)), this procedure is referred to as *Jtlgreedy* in Algorithm 2 and is only performed for instances of type TL-JSP.

The second observation was that the search method generally works considerably better when a solution, even one of relatively poor quality, is used to guide the value selection. To this end we perform an extremely short run with our standard search method to find a solution with objective value less than or equal to the initial upper bound. The time limit for this ($t1$ in Algorithm 2) was set to approximately 3 seconds.

The function *Solve* in Algorithm 1 always refers to the constraint method described in Sections 3.1.1, 3.1.2, 3.1.3 and 3.1.4. It takes as input a problem, the variable heuristic, value heuristic, best solution found thus far, the maximum objective value allowed, and a time limit ($t2$ for dichotomic search). The value heuristic is only used if no solution has been found thus far (i.e. $bestsol = \emptyset$).

Observe that, except for the computation of initial bounds for TL-JSP and ET-JSP, the constraint network given to the function *Solve* is completely determined by the disjunctive graph and the objective function. In particular, it is blind to particular structures such as cliques of disjunctive constraints corresponding to resources or even to the variant of JSP that is being solved.

3.2.2. Dichotomic Search There are two phases to our search strategy: a dichotomic search phase, followed by branch-and-bound search if optimality hasn't been proven.

In the dichotomic search phase, we repeatedly solve the decision problem with an additional constraint bounding the objective value by $\frac{ub+dicho_lb}{2}$, thus converting the instance to a satisfaction problem (i.e. one merely searches for any solution, not the best solution). The solve method has three stopping conditions: a solution is found (update the upper bound ub with the objective value of the solution and store the new solution as $bestsol$), it is proven that no solution exists for this objective value (update the lower bound, lb, and the dichotomic search lower bound $dicho_lb$), or the time limit was reached (update only the dichotomic search $dicho_lb$ as there may be a solution for this upper bound on the objective value).

3.2.3. Branch & Bound The function *BnbSolve* performs branch-and-bound search if optimality has not been proven during the dichotomic search phase. This takes additional inputs of the current upper bound and proven lower bound, and runs for the remainder of the time (where $t3$ is an overall time limit). However, the key point is that the search strategy employed in *BnbSolve* is identical to that in *Solve*, i.e. the same variable/value heuristics, restarting strategy, etc. are used throughout.

4. Comparison with the State of the Art

We evaluated the model described in Section 3 on a number of disjunctive scheduling problem types. The aim of the evaluation was twofold. Firstly, in this section we show that the approach outperforms a state-of-the-art traditional CP method on most problem types, is extremely close to the best dedicated techniques on several benchmarks, and is often much more efficient than one would expect from a relatively basic method. Secondly, in Section 5, we provide insight into the behavior of the algorithm by analyzing the different factors at work.

We implemented our approach in Mistral (Hebrard (2008)). All reported experiments were performed on an Intel Xeon 2.66GHz machine with 12GB of RAM on Fedora 9, unless otherwise stated. Each algorithm run on a problem had an overall time limit of 3600s. There were 10 runs per instance.

In order to provide a direct comparison for our method on all problem types, we implemented a standard CP model in IBM ILOG CP Optimizer v12.5, a state-of-the-art commercial CP solver. We adapted a model for the JSP from the CP Optimizer distribution

to all other variants. It uses the dedicated constraints for unary resource, setup times, precedence and "equalities" in the case of no-wait instances (respectively, `IloNoOverlap`, `IloTransitionDistance`, `IloEndBeforeStart`, `IloStartAtEnd`). The inference level for the `IloNoOverlap` constraint is set to `IloCP::Extended`. CP Optimizer features a sophisticated generic default search strategy. This heuristic uses information from decision "Impact" (Refalo (2004)), randomized restarts and Large Neighborhood Search (LNS Shaw (1998)), however, the exact details are not published. We used a single-thread version and we randomized each run by using a different random seed. Moreover, we also considered a variant where we allowed CP Optimizer to also branch on the construct `IloIntervalSequenceVar`, representing permutations of tasks in a unary resource, still using the black box strategy.

Due to the number of benchmarks our primary method of comparison is the average percentage relative deviation ($APRD$) per problem set. The PRD is given by

$$PRD = \frac{C_{Alg} - C_{Ref}}{C_{Ref}} * 100$$

where C_{Alg} is the objective value found by Alg, and C_{Ref} is the best objective value found for the instance over all comparison methods.

The same hardware, number of randomized runs, and time limit were used for CP Optimizer as for LW. For each problem type, we chose the best of the two CP Optimizer strategies, based on lowest APRD of the best solution found over the instances tested. However, it should be noted that on most problem types the results were quite similar for both strategies. Similarly, we tested both variable heuristics $Tdom/Bwt$ and $Tdom/Twt$ for LW and present the results for the best per problem type.

Detailed results of our approach on each problem variant are available at `http://4c.ucc.ie/~dgrimes/JSSP/jssp.htm`.

4.1. Problem Types

We first provide a brief description of the problem types tested (unless otherwise stated, the objective function is makespan minimization):

- Open Shop Scheduling (OSP): Tasks of the same job or running on the same machine cannot overlap.
- Job Shop Scheduling (JSP): Identical to the OSP with the exception that the ordering of tasks in a job is predefined.

- Job Shop Scheduling with Sequence Dependent Setup Times (*SDS-JSP*): A JSP with the additional constraint that there is a setup time for each task on a machine which is dependent on which preceding task was performed on the machine.
- Job Shop Scheduling with Time Lags (*TL-JSP*): A JSP with the additional constraint that there is a maximum time lag allowed between a task of a job finishing and the next task of the job starting.
- No-Wait Job Shop Scheduling (*NW-JSP*): A special case of the JTL where the maximum time lag is always 0, i.e. a task of a job must start immediately after the preceding task in the job has finished.
- Job Shop Scheduling with Earliness/Tardiness Costs (*ET-JSP*): A JSP with the objective function of earliness/tardiness cost minimization, where each job has an associated due date and cost for each time unit the job finishes early/late.

4.1.1. Benchmarks There are three sets of **OSP** instances which are widely studied in the literature, 60 instances, denoted `tai-∗`, of Taillard (1993); 52 instances, denoted `j-∗`, of Brucker et al. (1997); and 80 instances, denoted `gp-∗`, of Guéret and Prins (1999). All instances involve "square" problems, i.e. where the number of jobs and machines are equal. The instances range in size (in $n \times m$ format) from 3×3 to 20×20.

There are a large number of **JSP** benchmarks, stretching back to the 3 `ft-∗` instances proposed by Fisher and Thompson (1963). The other benchmarks we consider here are: 40 instances of Lawrence (1984) (denoted `la-∗`), 5 instances proposed by Adams et al. (1988) (denoted `abs-∗`), 10 instances proposed by Applegate and Cook (1991) (denoted `orb-∗`), 4 instances proposed by Yamada and Nakano (1992) (denoted `yn-∗`), 20 instances proposed by Storer et al. (1992) (denoted `swv-∗`), and finally 70 instances of Taillard (1993) (denoted `tai-∗`). Instances range in size from 6×6 to 50×20.

The **SDS-JSP** benchmarks we consider are those proposed by Brucker and Thiele (1996). These instances were generated by adding setup times to a subset of the job shop instances from Lawrence (1984). The problems in the set `t2-ps`∗ correspond to the job shop instances `la`01-15, problems in the set `t2-pss∗` are variations of `t2-ps`∗ where an alternative setup-time distribution was used. Instances range in size from 10×5 to 20×5.

The **TL-JSP** instances that have been studied in the literature, as presented in Caumond et al. (2008), were generated by adding time lag constraints to well known JSP benchmarks. In particular we tested on forty Lawrence (1984) instances, two instances of Fisher and

Thompson (1963), and four flow shop instances of Carlier (1978). In all these instances, the time lags are defined, for each job, by a factor ($\beta \in \{0, 0.25, 0.5, 1, 2, 3, 10\}$) of the average duration of tasks in the job. The `la` instances range in size from 10×5 to 30×10, while the four flow shop instances, denoted `car-*`, range in size from 10x6 to 8x9.

The **NW-JSP** instances are 82 of the JSP instances described above (all instances except Taillard), with the no-wait time lag constraint added, plus the four Carlier flow shop instances.

There are two sets of **ET-JSP** instances which have been proposed. The first benchmark consists of 9 sets of problems, each containing 10 instances. These were generated by Beck and Refalo (2003) using the random JSP generator of Watson et al. (1999). Three sets of JSPs of size 10×10, 15×10 and 20×10 were generated, each set containing ten instances. For each JSP instance, three ET-JSP instances were generated with different due dates and costs. The costs were uniformly drawn from the interval [1, 20]. The due dates were calculated based on the Taillard (1993) lower bound (*tlb*) of the base JSP instance, and were uniformly drawn from the interval:

$$[0.75 \text{ x } tlb \text{ x } lf,\ 1.25 \text{ x } tlb \text{ x } lf]$$

where *lf* is the looseness factor and takes one of the three values {1.0, 1.3, 1.5}. Thus for each problem size, there are 30 instances (10 for each looseness factor). Jobs do not have release dates in these instances.

The second benchmark is taken from the genetic algorithms literature (Morton and Pentico (1993)). There are 12 instances, split into sets `jb` and `ljb`, with problem size ranging from 10×3 to 50×8. Jobs do have release dates in these instances, furthermore earliness and tardiness costs of a job are equal.

4.2. Comparison with CP Optimizer

In Table 1, we compare *LW* and CP Optimizer (denoted *CPO* in tables and figures) on all instances of each problem type in terms of APRD and ability to prove optimality. The APRD was calculated for both the best objective value from the 10 randomized runs on an instance (*best*), and the average objective value over these runs (*avg*) for *LW* and CP-Optimizer (C_{ref} was the best found between *LW* and CP Optimizer). For optimality proofs, results are given in terms of number of instances for which proofs of optimality were obtained on at least one run (*atleast1*), and on all ten runs (*every*) for the instance.

In terms of APRD, the results obtained with CP Optimizer are extremely good in many cases. Indeed it found several new best results (NW-JSP instances `swv16` through `swv20`, ET-JSP instances `ljb2`, `ljb9` and `ljb10`, as well as numerous TL-JSP instances). The results suggest that CP Optimizer scales better. It is more efficient on data sets involving larger instances, such as `swv` or the larger `tai` instances for JSP, whereas *LW* tends to be best on smaller instances. This is very significant on ET-JSP data sets `jb` (maximum size 15x5) and `ljb` (maximum size 50x8). Although *LW* was relatively good on the former set, and in particular proving optimality in 85% of the runs, it was far from CP Optimizer in terms of APRD on the latter set of six instances (26.2 versus 2.1 for APRD of best objective found per instance). Part of this observation might be explained by the different time complexity of reaching a fix point on a resource constraint during propagation. Dedicated filtering algorithms can run in $O(n \log n)$ Vilím (2007) whereas the decomposition uses a quadratic number of constraints and the worst case time complexity also linearly grows with the makepan. However, in practice, the decomposition is inherently incremental and the worst case analysis is very far from the typical behavior. We suspect that the better scaling may actually be better explained by the use of Large Neighborhood Search by CP Optimizer. Overall, however, JSP was the only problem type where *LW* had worse APRD than CP Optimizer.

Table 1 Direct APRD and Proven Optimality Comparison: LW vs CP Optimizer

Problem	#inst.	APRD				Opt			
		LW		*CPO*		*LW*		*CPO*	
		Best	Avg	Best	Avg	atleast1	every	atleast1	every
OSP	192	**0**	**0**	0.00	0.01	**192**	**192**	127	127
JSP	152	1.14	1.81	**0.08**	0.54	**67**	63	66	58
ET-JSP	102	**1.59**	4.88	2.52	11.5	**75**	73	34	33
SDS-JSP	25	**0**	0.45	0.27	0.61	**13**	12	1	1
TL-JSP	287	**0.33**	1.21	0.70	1.40	**198**	172	80	69
NW-JSP	86	**0.59**	2.01	1.18	2.22	**41**	38	1	1

The results of *LW* are even more impressive in terms of instances where optimality was proven, solving 586 of the 844 instances on at least one run. In comparison, CP Optimizer only proved optimality on 309 instances. This may be partly due to strategies, such as LNS, used by CP Optimizer. Such hypothesis is difficult to corroborate because of the black-box nature of CP Optimizer.. However, similar behavior has already been observed with older versions (such as Ilog Scheduler), or when comparing comparable models in Choco in the case of OSPs (Grimes et al. (2009)). Notice that we do not give an initial lower

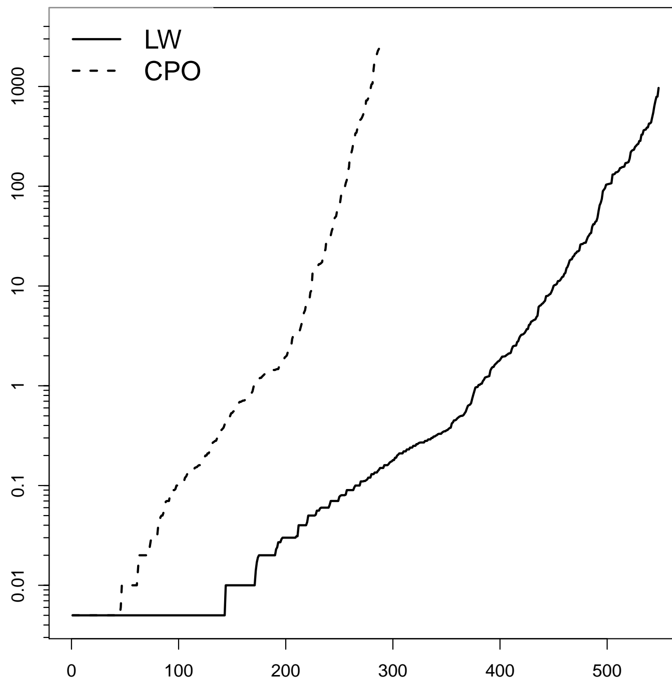


Figure 2 **Average runtimes on instances where optimality was proven on all ten runs.**

bound to the CP Optimizer model. However, this cannot explain the observed gap. Indeed, straightforward constraint propagation on precedence and unary resource constraints is sufficient to obtain the same bounds as those given to the LW approach.

We plot the average runtime for instances where optimality was proven on every run in Figure 2. Of the 550 such instances for LW, 494 were solved to optimality in less than one minute on average (note that the y-axis is in log scale), and 380 of these required less than one second on average.

CP Optimizer only solved 172 instances to optimality in under one second and 251 in under one minute, of the 287 instances where it proved optimality on every run. Furthermore, 157 of these 287 instances required no proof as the trivial lower bound was the optimal value, compared to 132 of the 584 instances requiring no proof for LW. Overall, LW closed 3 open instances of SDS-JSP, 187 open instances of TL-JSP, 3 open instances of ET-JSP (and found a number of improving solutions).

One influencing factor in the ability to prove optimality is the restarting strategy. The default method in CP Optimizer is a geometric strategy with a base, s, of 100 failures, and factor, r, of 1.05. This would mean a much more gradual increase in the cutoff compared to that used by LW. In order to test the impact of the restarting strategy, we reran CP

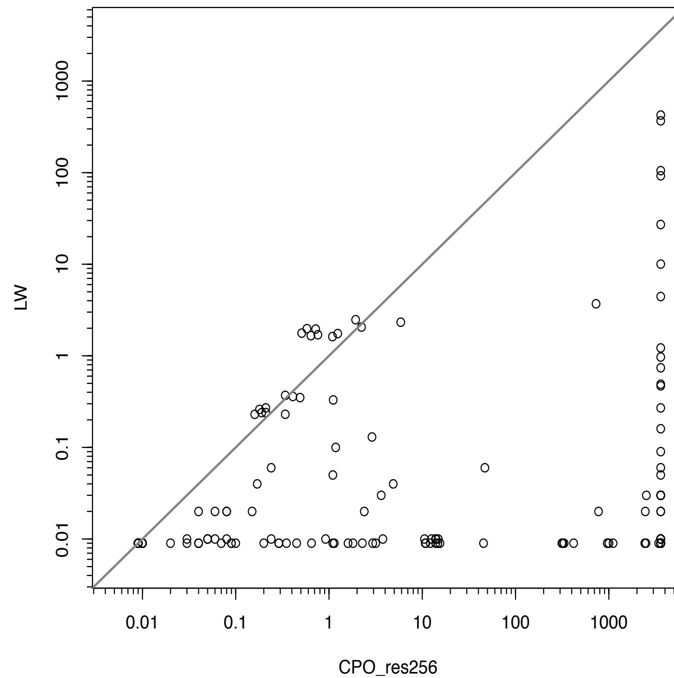


Figure 3 Average runtime for 192 OSP instances using same restarting strategy.

Optimizer with the same restart settings as those used in *LW*, on a subset of the problem types (192 OSP instances and 25 SDS-JSP instances).

The results showed only a minor improvement for the OSP instances, where optimality was proven for three further instances, and no further instances were solved to optimality in the SDS-JSP instances. Figure 3 is a scatter plot of average runtime per instance for *LW* versus CP Optimizer with the same restarting strategy (*CPO_res256*), and illustrates that most of the instances where the latter failed to prove optimality in one hour, were solved in less than one second by *LW*.

Although the inference used by *LW* is of course much weaker, we believe that the success in proving optimality can be explained by the weighted-degree heuristic. The heuristic focuses search on the critical part of the constraint network, identified through the repeated involvement in failures, which is key for generating proofs. Moreover, whereas CP Optimizer must essentially go through the same computations whenever it restarts or when the upper bound is improved, a significant part of the work is cached in the *LW* approach. In particular, the weights learned by the heuristics are still relevant and, as we show later, are used to focus search on roughly the same area (even though this evolves during search).

Overall, if the *LW* approach is more efficient on the tested benchmarks, it does not seem to scale as well as CP Optimizer. However, if (as we believe) this is for a large part due to

the use of LNS steps by CP Optimizer, it suggests that a method based on LNS, but using LW when solving a subproblem may be promising. Indeed, LW is extremely efficient on small instances and the size of the subproblems does not necessarily grows linearly with the size of the problem.

4.3. State-of-the-art Systematic and Metaheuristic Comparison Methods

Each variant of JSP has been the focus of significant research work. For the subsequent comparison, we used the best known values reported in the literature for the following complete and incomplete methods:

- *OSP*: The best systematic method is a dedicated CP approach of Malapert et al. (2008), implemented in the solver Choco. This method contains a number of dedicated filtering techniques for the open shop problem, and was the first method able to solve all 192 instances to optimality in less than an hour. The best metaheuristic approach for this problem type is a particle swarm optimization proposed by Sha and Hsu (2008).
- *JSP*: Metaheuristic approaches have proven to be the most successful on JSPs, in particular the well-known tabu search method *i-TSAB* of Nowicki and Smutnicki (2005). The current best metaheuristic method is a tabu search with path-relinking recently proposed by Peng et al. (2014), which improved the best-known solutions on 49 of 205 well-known instances studied. The SGMPCS approach of Beck (2007), discussed earlier in Section 3.1.3, has shown best performance of systematic methods proposed to date. It combines the standard strong filtering techniques with solution-guided value ordering.
- *SDS-JSP*: The systematic method we compare with is a CP approach of Artigues and Feillet (2008) which computes solutions to the traveling salesman problem (TSP) with time windows induced by the setup times on a machine in order to get better bounds. The metaheuristic comparison is a hybrid genetic algorithm (GA) with local search introduced by González et al. (2008).
- *TL-JSP*: The general TL-JSP has only recently received attention in the literature (most methods have been proposed for the special case, NW-JSP, where all time lags are zero). Caumond et al. (2008) introduced a memetic algorithm, which is a combination of population based search and local search with an emphasis on using problem specific knowledge. Artigues et al. (2011) proposed a CP branch and bound method using dichotomic search and the branching scheme of Torres and Lopez (2000).

• *NW-JSP*: van den Broek (2009) proposed a Mixed Integer Programming (MIP) formulation of the problem incorporating the *alternative graph* representation (Mascis and Pacciarelli (2002)), immediate selection rules, and a new job insertion heuristic to initialize the upper bound. A large number of metaheuristic approaches have been proposed in recent years for the NW-JSP, most decompose the problem into sequencing and timetabling components. The current best method is a local search technique (*OJILS2*) proposed by Bürgy and Gröflin (2012) incorporating a job insertion heuristic based on their approach to the optimal job insertion problem.

• *ET-JSP*: Danna and Perron (2003) proposed complete ("unstructured large neighborhood search" *uLNS*) and incomplete ("structured large neighborhood search method" *sLNS*) methods, incorporating MIP-based *large neighborhood search* (Shaw (1998)).

We report a summary of the APRD results comparing *LW* and CP Optimizer to the best dedicated complete and metaheuristic methods on all problem types in Table 2. The reference used to compute the PRD is the best objective value obtained by either *LW*, CP Optimizer, or the comparison dedicated approaches. Note that we only present results for a subset of instances as the dedicated methods didn't provide results for all instances for some problem types. Moreover, except for *LW*, CP Optimizer, and the best complete methods on OSPs and JSPs, the following results where not obtained on the same machines nor necessarily with the same time limit.

Table 2 APRD comparison with the State of the Art

Problem	#inst.	*LW*		*CPO*		SOA Systematic		SOA Heuristic	
		Best	Avg	Best	Avg	Best	Ref	Best	Ref
OSP	175	**0**	**0**	**0**	0.01	**0**	Malapert et al. (2008)	0.04	Sha and Hsu (2008)
JSP	115	1.49	2.18	0.48	1.01	0.81	Beck (2007)	**0.01**	Peng et al. (2014)
ET-JSP	42	**3.85**	16.14	6.27	34.15	274.4	Danna and Perron (2003)	100.46	Danna and Perron (2003)
SDS-JSP	15	**0.04**	0.56	0.19	0.52	2.06	Artigues and Feillet (2008)	0.14	González et al. (2008)
TL-JSP	27	**0**	0.04	0.06	0.08	6.00	Artigues et al. (2011)	15.60	Caumond et al. (2008)
NW-JSP	52	2.16	4.47	2.74	4.41	-	van den Broek (2009)	**0.18**	Bürgy and Gröflin (2012)

The results show that both CP Optimizer and *LW* are competitive with the state of the art on most problem types. Indeed they generally outperformed the best systematic methods, and were only worse than the best metaheuristic methods on problems of type JSP and NW-JSP. We do not present results on NW-JSP for the systematic method of van den Broek as there was not sufficient overlap in results given. However, we note that this method proved optimality on 13 more instances than *LW*, albeit with the proofs of optimality for these instances taking over 21 hours on average.

5. Analysis of *LW* method

5.1. Component Analysis

In this section we assess the relative importance of the different components of our algorithm to its overall performance. The motivation for this is twofold. Firstly, the results provide further insight into the behavior of the *LW* method, which could be of use to other researchers considering a similar approach. Secondly, the results provide insight into the problems themselves, which can also clarify why certain components of our approach are more suited to one type of problem than another.

For each problem type we compared the default method with the following variations on the default setting:

- Variable Ordering: $Tdom$, Bwt, and Twt alone; and the defaults $Tdom/Bwt$ or $Tdom/Twt$.
- Value Ordering: Promise and lexical (for the job shop problem and its variants, this is the static heuristic as described in Section 3.1.3). The default used solution guided value ordering.
- Without nogood recording from restarts (*noNgd*). The default used nogood recording from restarts.
- Dichotomic Search: None versus 3 or 300 second cutoffs (*noDs/Ds3s/Ds300s*). The default used a 30 second cutoff for dichotomic search.
- Restarting: None versus Luby (Luby et al. (1993)) restarting strategy (*noRestart/Luby*). For the latter, the scale factor was 4096 failures. The default used the geometric restarting strategy.

For each variant of an algorithmic component, all other components used the default settings (for example *Luby* combined the Luby restarting strategy with the default variable heuristic for the problem type, solution guided value ordering, nogood recording from restarts, 30 second cutoff for each dichotomic step, and random tie breaking).

5.1.1. Experimental Setup and Benchmarks There were ten runs per instance with an overall time limit of twenty minutes per run on an instance. A subset of ten benchmarks were randomly selected for each problem type. However, the selection process was biased so as to contain a number of difficult instances (based on the results of the previous sections), and to have problems of varying size.

The algorithms were compared in terms of APRD of average objective values, where C_{ref} was the best objective value found on the instance over the different variants; and in terms of proven optimality. (More detailed analysis can be found in Grimes (2012).) For ET-JSPs the sample contained five of the instances generated by Beck and Refalo (2003) and five of the instances from Morton and Pentico (1993). The APRD for the latter was calculated relative to the normalized objective as described in the previous section, and the overall results for ET-JSPs are averaged over the two sets.

The results for APRD are given in Figure 4, while Figure 5 shows the performance in terms of number of runs where optimality was *not* proven. For clarity, we have restricted the APRD to be within 15%. The ET-JSP results for *noRestart*, Twt and, in particular, $Tdom$ had much larger APRDs, with the latter failing to improve on the initial (extremely poor) upper bound for some of the Beck-Refalo instances.

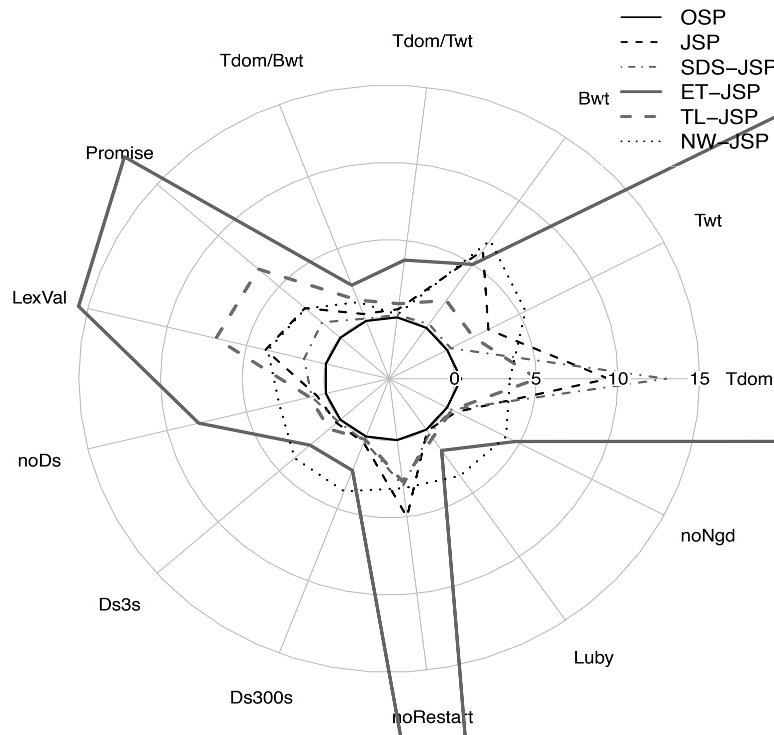


Figure 4 Average Percent Relative Deviation.

A peak in the curve for a given non-default value of a component indicates a particularly suboptimal choice. For example, the curve for JSP shows that the four following parameters are the most important, in that order: using the information about constraint weights;

using the information about task domain size; using restarts; and guiding search with the previous best solution.

The rankings are relatively consistent in terms of the key algorithmic factors across the different problem types. The weighted component of the variable heuristic was generally most important, as evidenced by the poor performance of $Tdom$, which was worst overall on four of the six problem types and the worst of the variable heuristic variations on one other. Although all variations found extremely good, if not optimal, solutions on the OSPs, Figure 5 reveals that the variable ordering heuristic ($Tdom/Bwt$) was key as the other variations of the variable heuristic failed to prove optimality on all instances.

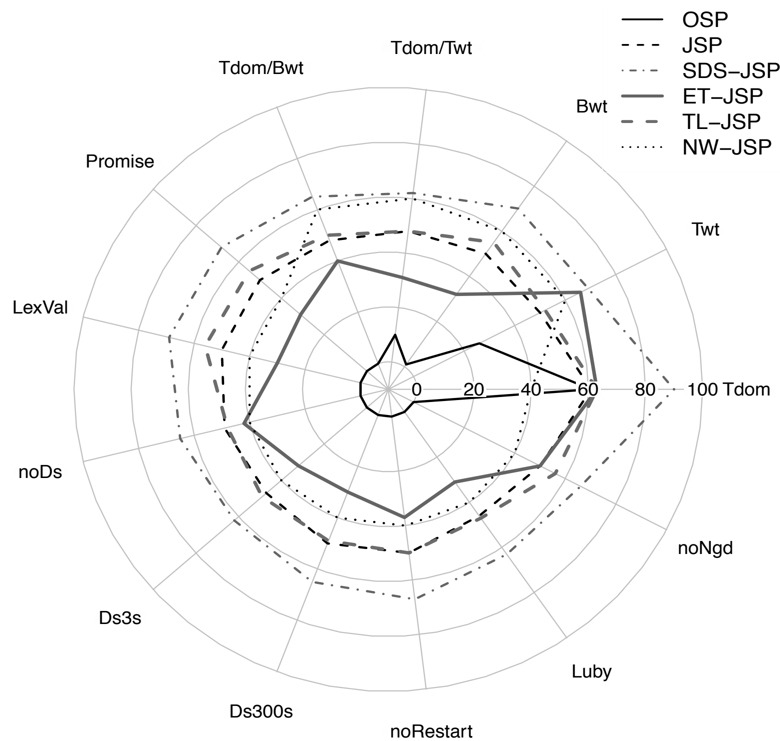


Figure 5 **Number of runs where optimality was not proven.**

Interestingly, for problems involving time lag constraints, the ranking of the components was similar for the general case but quite different for the no-wait case. The most obvious difference is that the weighted degree component of the variable heuristic was much less important than the (tasks) domain size for NW-JSP. Similarly, solution guided value ordering and restarting appear to have much less influence on this problem type compared to the others. Only minor gains were found in general with the other factors, such as dichotomic search and nogood recording.

Finally, we investigated the importance of the initial lower and upper bounds computed. Figure 6 presents averages per problem type, together with the average of the best objectives value found by LW. Note that we do not include the results for ET-JSPs as the lower bound was always 0, while the average upper bound was nearly two orders of magnitude greater than the average best objective value found by LW.

We found that the initial lower and upper bounds were not particularly strong on average, indeed the upper bounds used for TL-JSPs, NW-JSPs and ET-JSPs were quite poor. This only impacted the performance of LW (without using the greedy heuristic for improving the upper bound) on the problem type TL-JSP, where it failed to improve on the initial solution for some of the largest instances (Grimes (2012)).

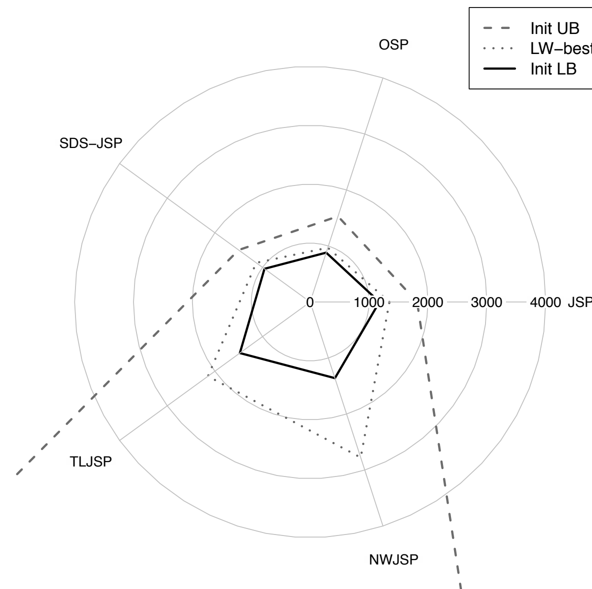


Figure 6 Radial plot of average initial lower and upper bounds, and average best LW objective for different problem types.

Overall these results show that the variable heuristic is the key factor in our algorithm, although the relative importance of the two components of the heuristic varies depending on the problem type. Furthermore, it is clear that our algorithm could be improved further by fine tuning the various parameters for the different problem types. However, the goal of this work was to illustrate that a relatively simple generic method can be successfully applied to scheduling problems without the need for parameter tuning for each specific problem type.

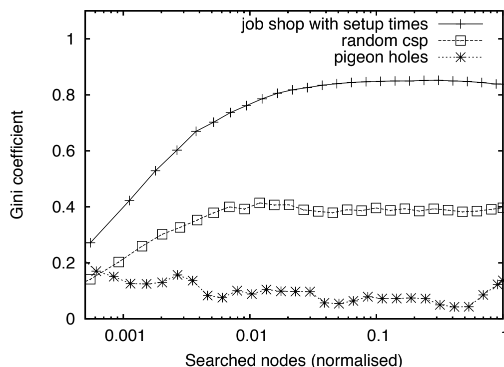


Figure 7 Weight distribution bias: Gini coefficient over the (normalized) number of searched nodes.

5.2. Weight analysis

We generated weight profiles on a sample of problems for each problem type. The experimental setup involved running branch and bound search with a 30 second cutoff. Weights were stored after either the cutoff was reached or the problem was solved. The purpose of these experiments is to provide insight into the behaviour of search on the different problem sets and to ascertain whether there was a correlation between search performance and the level of discrimination amongst the variable weights.

We used the *Gini* coefficient (Gini (1912)) to characterize the weight distribution, which is a metric of inequality, used for instance to analyse distribution of wealth in social science. The Gini coefficient is based on the *Lorenz* curve, mapping the cumulated proportion of income y of a fraction x of the poorest population. When the distribution is perfectly fair, the Lorenz curve is $y = x$. The Gini coefficient is the ratio of the area lying between the Lorenz curve and $x = y$, over the total area below $x = y$.

In simplistic terms, a low Gini coefficient for the weight distribution means the weights are spread across many variables while a high coefficient would mean that the weights are concentrated on a small subset of variables. In the latter case, this means that there were a small number of variables which were repeatedly the cause of failure, i.e. the weights identify the bottleneck search variables in the problem.

Let us first consider search trees for unsatisfiable instances. In an ideal situation, search converges immediately toward a given set of variables from which a short proof of unsatisfiability can be extracted. In this case, the Gini coefficient of the weight distribution typically increases rapidly and monotonically. In Figure 7 we plot the Gini coefficient of the proofs for an SDS-JSP instance; a random binary CSP instance with 100 variables,

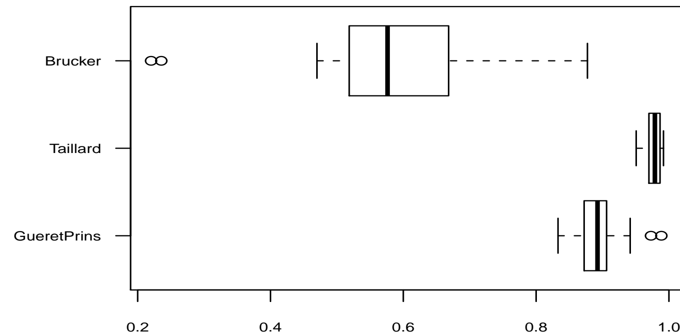


Figure 8 **Boxplot representation of Gini coefficient of weight increment distribution for largest OSPs of the three sets.**

a domain size of 15, 250 binary constraints of tightness 0.53 uniformly distributed; and a pigeonholes instance, which is an academic problem where all variables and values are symmetrically equivalent.

After each geometric restart, the Gini coefficient is computed and plotted against the current number of explored nodes. Note that the Gini coefficient is low at the start, as the weighted degree of all variables is simply the number of constraints involving the variable.

We observe first that, as expected, the weight distribution is evenly spread amongst all variables throughout search on the pigeonholes problem (due to the symmetry amongst variables). The Gini coefficient for the random CSP illustrates that slightly greater discrimination is offered by the weights, albeit the weight is still spread amongst a large proportion of the variables. The behavior of the Gini coefficient is quite different for the SDS-JSP instance, where it quickly increases before stabilizing at a high value. This indicates that the weight is accruing on a small subset of variables.

5.2.1. Weight profiles for OSPs We next analyzed the weight profiles generated on OSPs with the heuristic $Tdom/Bwt$, comparing the weight *increment* received by each variable using the Gini coefficient. The problems tested were the 20 largest instances from both Taillard ($tai15$-$*$, $tai20$-$*$) and Gueret-Prins sets ($gp09$-$*$, $gp10$-$*$), and the 26 largest Brucker instances ($j6per$-$*$, $j7per$-$*$, $j8per$-$*$). Boxplots of the results, in terms of the Gini coefficient of the weight increments, are shown in Figure 8.

We see that the Taillard and Gueret-Prins instances always yielded a Gini coefficient of greater than 0.83. The Brucker instances, on the other hand, had Gini coefficients ranging from 0.22 to 0.88, with a median value of 0.58. This means that there was a relatively small set of variables which were identified as the primary sources of conflict for the Taillard and

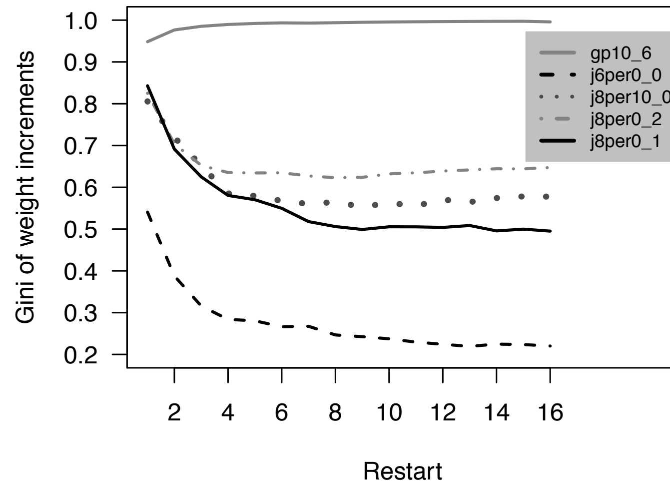


Figure 9 **Evolution of weight increment distribution across restarts for sample OSPs (gp∗ is Gueret-Prins instance, j∗per∗ is Brucker instance).**

Gueret-Prins problems, whereas there were fewer clearly defined bottleneck variables in most of the Brucker instances. This matches the results with LW, where all Taillard and Gueret-Prins instances were solved in under 5 seconds, compared to 100s of seconds for proving optimality on some Brucker instances.

However, there is a caveat that should be noted. Since weight is distributed by finite increments, when the number of failures is low with respect to the number of variables, it cannot be fairly distributed and we obtain an artificially high Gini coefficient. As the number of failures increase, the Gini coefficient of the weighted degrees and the weight increments converge.

Figure 9 illustrates the evolution of the distribution of weight increments across restarts for a sample of OSPs. These instances were chosen as there were sufficient failures to be of interest. The results again show that the heuristic found it more difficult to identify bottleneck variables in the Brucker instances than in the Gueret-Prins instance. The important point is that there was a sharp decline in the Gini coefficient on all Brucker instances over the first three restarts (approximately 1000 failures). This refutes the hypothesis that the results in Figure 8 were due to a greater number of failures encountered on the Brucker instances.

Focusing on the instances gp10_6 and j8per0_2, we first note that search effort was similar on both instances (∼50K/60K failures). Analysis of the weight profiles on these two instances further corroborates the findings of Figure 9. Here only 33% of variables

received any weight increment during search on gp10_6 , compared to 85% of the variables in j8per0_2.

5.2.2. Weight profiles for the JSP and its variants The results on the OSPs suggest that the weight discrimination is the reason for the good performance of $Tdom/Bwt$. We tested this hypothesis further on variants of the JSP, generating weight profiles with both $Tdom/Bwt$ and $Tdom/Twt$. The benchmarks tested were a sample of ten Lawrence instances la06-15, as these have SDS-JSP, TL-JSP, and NW-JSP variants, i.e. the same base JSP is used for each variant. Of particular interest is the weight discrimination on the NW-JSPs, as $Tdom/Bwt$ performed relatively poorly on these problems.

Figure 10 shows the boxplots of the Gini coefficients for the ten instances of the different variants. We include two sets of SDS-JSPs (denoted ps-$*$ and pss-$*$ in our earlier experiments, here *sds1* and *sds2* respectively), and two sets of TL-JSPs (for time lag $\beta = 0$, 1, denoted $jtl0$, $jtl1$ respectively in the figure). Weight profiles for the base JSPs are not included as these were solved in under 0.1 seconds in most cases, encountering few failures.

The clearest pattern observable in the figure illustrates the difference between the distribution of weights for the two heuristics. The Gini coefficients of the Bwts were consistently much higher than those of the Twts, while there was greater variation in the Gini coefficients of the Twts across instances for each variant except on the NW-JSP instances which were consistently low.

This is to be expected as the Twt of a Boolean variable is the sum of the weights on the two tasks sharing the disjunct. A task variable with high weight will contribute this high weight to the Twt calculation of *all* Booleans sharing a disjunct with this task. Therefore the weight will be spread over more Booleans than with Bwt.

However, the results show little correlation between search performance and the Gini coefficients in these experiments as $Tdom/Twt$ here performed much better (both in terms of proving optimality and objective found) than $Tdom/Bwt$ on both sets of SDS-JSP instances and on the NW-JSP instances (although the opposite was the case on the two TL-JSP sets). Therefore a high Gini coefficient does not seem to be sufficient to guarantee good performance.

6. Conclusion

In this paper we introduced a new approach for solving disjunctive scheduling problems, combining relatively simple inference with a number of generic CP techniques such as

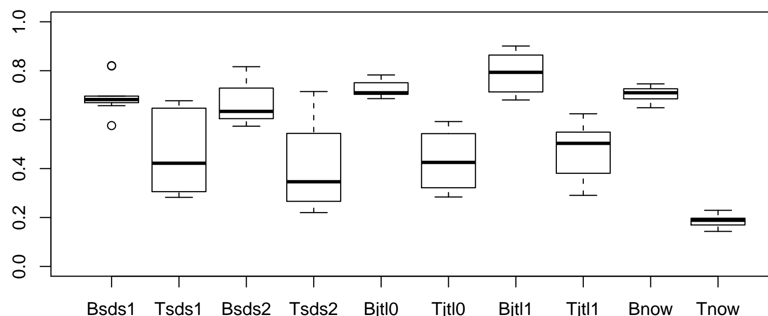


Figure 10 **Gini coefficients of Twt increment (T$*$) and Bwt increment (B$*$) for different variants of sample Lawrence instances.**

restarting, weighted degree variable ordering, and solution guided value ordering. We took a minimalistic approach to modeling the problem, simply decomposing the unary resource constraints into primitive disjunctive constraints. In comparison, most CP techniques model the unary resource constraint as a global constraint, devising specialized filtering algorithms for the constraint.

We demonstrated the benefits of our approach on a variety of problems, and in so doing we have refuted the conventional wisdom that problem-specific heuristics and, in particular, problem-specific inference methods are necessary to achieve good performance on problems of this nature. The advantages of using a weighted degree based heuristic for these problems, and indeed in general, is twofold: it can identify *critical variables* without (a) costly calculations at each node and (b) costly inference methods.

We have further shown that our basic method can be easily adapted to handle a number of side constraints (setup times and maximum time constraints) and the objective of minimizing the earliness/tardiness costs. This is important as these side constraints and the alternative objective have proven troublesome for traditional CP methods due to their impact on the dedicated inference methods. Since our method is more search intensive than inference intensive, it suffers less from these issues.

However, our method cannot be considered to be completely generic as domain specific knowledge was incorporated to good effect in certain cases. Firstly, the variable heuristic was improved by adding information regarding the domain sizes of the tasks, and in some cases weight information of the tasks. Secondly, our method for solving the NW-JSP was improved by using a model specific to the problem based on the simple observation that each task in a job is functionally dependent on the other tasks of the job. Thirdly, the addition of a dedicated metaheuristic for finding good initial upper bound also improved

performance on the TL-JSPs. These results show that there is still room for improvement through the combination of generic techniques with problem-specific information.

Acknowledgments

The authors would like to thank the following: Arnaud Malapert for the work on open shop problems, Hadrien Cambazard for helpful feedback, and Chris Beck for providing both source code and feedback.

References

Adams, J., E. Balas, D. Zawack. 1988. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science* **34** 391–401.

Applegate, D., W. J. Cook. 1991. A Computational Study of the Job-Shop Scheduling Problem. *INFORMS Journal on Computing* **3** 149–156.

Artigues, C., D. Feillet. 2008. A branch and bound method for the job-shop problem with sequence-dependent setup times. *Annals of Operations Research* **159** 135–159.

Artigues, C., M-J. Huguet, P. Lopez. 2011. Generalized Disjunctive Constraint Propagation for Solving the Job Shop Problem with Time Lags. *Engineering Applications of Artificial Intelligence* **24** 220 – 231.

Baptiste, P., C. Le Pape. 1995. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. *Proceedings of the 14th International Joint Conference on Artificial Intelligence - IJCAI'95*. 600–606.

Baptiste, P., C. Le Pape. 1996. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. *Proceedings of the 15th Workshop of the U.K. Planning Special Interest Group*.

Baptiste, P., C. Le Pape, W Nuijten. 2001. *Constraint-Based Scheduling: Applying Constraint Programming Techniques to Scheduling Problems*. Kluwer Academic Publishers.

Beck, J. C. 2007. Solution-Guided Multi-Point Constructive Search for Job Shop Scheduling. *Journal of Artificial Intelligence Research* **29** 49–77.

Beck, J. C., A. J. Davenport, E. M. Sitarski, M. S. Fox. 1997. Texture-Based Heuristics for Scheduling Revisited. *Proceedings of the 14th National Conference on Artificial Intelligence - AAAI'97*. 241–248.

Beck, J. C., P. Refalo. 2003. A Hybrid Approach to Scheduling with Earliness and Tardiness Costs. *Annals of Operations Research* **118** 49–71.

Boussemart, F., F. Hemery, C. Lecoutre, L. Sais. 2004. Boosting Systematic Search by Weighting Constraints. *Proceedings of the 16th European Conference on Artificial Intelligence - ECAI'04*. 146–150.

Brucker, P., J. Hurink, B. Jurisch, B. Wöstmann. 1997. A branch & bound algorithm for the open-shop problem. *Discrete Applied Mathematics* **76** 43–59.

Brucker, P., O. Thiele. 1996. A branch and bound method for the general- shop problem with sequence-dependent setup times. *Operation Research Spektrum* **18** 145–161.

Bürgy, R., H. Gröflin. 2012. Optimal job insertion in the no-wait job shop. *Journal of Combinatorial Optimization* 1–27.

Carlier, J. 1978. Ordonnancements à contraintes disjonctives. *R.A.I.R.O Recherche opérationelle / Operations Research* **12** 333–350.

Carlier, J., E. Pinson. 1989. An Algorithm for Solving the Job-shop Problem. *Management Science* **35** 164–176.

Carlier, J., E. Pinson. 1994. Adjustment of Heads and Tails for the Job-Shop Problem. *European Journal of Operations Research* **78** 146–191.

Caumond, A., P. Lacomme, N. Tchernev. 2008. A memetic algorithm for the job-shop with time-lags. *Computers & Operations Research* **35** 2331–2356.

Danna, E., L. Perron. 2003. Structured vs. Unstructured Large Neighborhood Search: A Case Study on Job-Shop Scheduling Problems with Earliness and Tardiness Costs. *Proceedings of Principles and Practice of Constraint Programming - CP'03. LNCS No. 2833*. 817–821.

Dechter, R., I. Meiri, J. Pearl. 1991. Temporal Constraint Networks. *Artif. Intell.* **49** 61–95.

Fisher, H., G. L. Thompson. 1963. Probabilistic Learning Combinations of Local Job-shop Scheduling Rules. *Industrial Scheduling* 225–251.

Fox, M. S., N. M. Sadeh, C. A. Baykan. 1989. Constrained Heuristic Search. *Proceedings of the 11th International Joint Conference on Artificial Intelligence - IJCAI'89*. 309–315.

Geelen, P. A. 1992. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. *Proceedings of the 10th European Conference on Artificial Intelligence - ECAI'92*. 31–35.

Gini, C. 1912. Variabilita e mutabilita contributo allo studio della distribuzioni. *Studie Economico-Guiridici della R. Universita di Cagliari* .

González, M. A., C. R. Vela, R. Varela. 2008. A New Hybrid Genetic Algorithm for the Job Shop Scheduling Problem with Setup Times. J. Rintanen, B. Nebel, J. C. Beck, E. A. Hansen, eds., *ICAPS*. AAAI, 116–123.

Grimes, D. 2012. Identifying Sources of Global Contention in Constraint Satisfaction Search. Ph.D. thesis, National University of Ireland, Cork. URL `http://cora.ucc.ie/handle/10468/646`.

Grimes, D., E. Hebrard. 2010. Job shop scheduling with setup times and maximal time-lags: A simple constraint programming approach. A. Lodi, M. Milano, P. Toth, eds., *CPAIOR*, *Lecture Notes in Computer Science*, vol. 6140. Springer, 147–161.

Grimes, D., E. Hebrard. 2011. Models and Strategies for Variants of the Job Shop Scheduling Problem. *Proceedings of Principles and Practice of Constraint Programming - CP'11. LNCS No. 6876*. Springer, 356–372.

Grimes, D., E. Hebrard, A. Malapert. 2009. Closing the open shop: Contradicting conventional wisdom. *Proceedings of Principles and Practice of Constraint Programming - CP'09. LNCS No. 5732*. 400–408.

Guéret, C., C. Prins. 1999. A new lower bound for the Open-Shop problem. *Annals of Operations Research* **92** 165–183.

Hebrard, E. 2008. Mistral, a Constraint Satisfaction Library. *The Third International CSP Solver Competition*. 31–40.

Heckman, I. 2007. Empirical analysis of solution guided multi-point constructive search. Master's thesis, University of Toronto.

Johnson, S. M. 1954. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly* **1** 61–68.

Laborie, P. 2003. Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and new Results. *Artificial Intelligence* **143** 151–188.

Laborie, P. 2005. Complete MCS-based search: Application to Resource Constrained Project Scheduling. *Proceedings of the 19th International Joint Conference on Artificial Intelligence - IJCAI'05*. 181–186.

Laborie, P. 2009. IBM ILOG CP optimizer for detailed scheduling illustrated on three problems. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 148–162.

Lawrence, S. 1984. Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques (Supplement). *Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania.* .

Le Pape, C. 1994. Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems. *Intelligent Systems Engineering* **3** 55–66.

Lecoutre, C., L. Sais, S. Tabary, V. Vidal. 2007. Nogood recording from restarts. *Proceedings of the 20th International Joint Conference on Artificial Intelligence - IJCAI'07*. 131–136.

Luby, M., A. Sinclair, D. Zuckerman. 1993. Optimal Speedup of Las Vegas Algorithms. *Israel Symposium on Theory of Computing Systems*. 128–133.

Malapert, A., H. Cambazard, C. Guéret, N. Jussien, A. Langevin, L-M. Rousseau. 2008. An Optimal Constraint Programming Approach to the Open-Shop problem. *submitted to INFORMS, Journal of Computing* .

Martin, P., D. B. Shmoys. 1996. A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem. *5th International Conference on Integer Programming and Combinatorial Optimization-IPCO'96. LNCS No. 1084*. 389–403.

Mascis, A., D. Pacciarelli. 2002. Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research* **143** 498–517.

Morton, T. E., D. W. Pentico. 1993. *Heuristic Scheduling Systems*. John Wiley and Sons.

Nowicki, E., C. Smutnicki. 1996. A Fast Taboo Search Algorithm for the Job Shop Problem. *Manage. Sci.* **42** 797–813.

Nowicki, E., C. Smutnicki. 2005. An Advanced Tabu Search Algorithm for the Job Shop Problem. *Journal of Scheduling* **8** 145–159.

Nuijten, W. 1994. Time and Resource Constraint Scheduling: A Constraint Satisfaction Approach. Ph.D. thesis, Eindhoven University of Technology.

Ovacik, I. M., R. Uzsoy. 1997. Decomposition methods for scheduling semiconductor testing facilities. *International Journal of Flexible Manufacturing Systems* **8** 357–387.

Peng, B., Z. Lu, T.C.E. Cheng. 2014. A Tabu Search/Path Relinking Algorithm to Solve the Job Shop Scheduling Problem. *arXiv preprint arXiv:1402.5613* .

Raaymakers, W. H. M., J. A. Hoogeveen. 2000. Scheduling multipurpose batch process industries with no-wait restrictions by simulated annealing. *European Journal of Operational Research* **126** 131 – 151.

Rajendran, C. 1994. A No-Wait Flowshop Scheduling Heuristic to Minimize Makespan. *The Journal of the Operational Research Society* **45** 472–478.

Refalo, P. 2004. Impact-Based Search Strategies for Constraint Programming. *Proceedings of Principles and Practice of Constraint Programming - CP'04. LNCS No. 3258*. 557–571.

Roy, B., M. A. Sussman. 1964. Les problèmes d'ordonnancement avec contraintes disjonctive. Tech. Rep. Note DS No.9 bis, SEMA, Paris.

Sadeh, N. M. 1991. Lookahead techniques for micro-opportunistic job-shop scheduling. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Sadeh, N. M., M. S. Fox. 1996. Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem. *Artificial Intelligence* **86** 1–41.

Sha, D. Y., C-Y. Hsu. 2008. A new Particle Swarm Optimization for the Open Shop Scheduling Problem. *Computers & Operations Research* **35** 3243–3261.

Shaw, P. 1998. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. *Proceedings of Principles and Practice of Constraint Programming - CP'98. LNCS No. 1520*. 417–431.

Storer, R. H., S. D. Wu, R. Vaccari. 1992. New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling. *Management Science* **38** 1495–1509.

Taillard, E. 1993. Benchmarks for basic scheduling problems. *European Journal of Operations Research* **64** 278–285.

Torres, P., P. Lopez. 2000. On Not-First/Not-Last conditions in disjunctive scheduling. *European Journal of Operational Research* **127** 332–343.

van den Broek, J. J. J. 2009. MIP-based Approaches for Complex Planning Problems. Ph.D. thesis, Eindhoven University of Technology, Netherlands. URL `http://www.bsik-bricks.nl/documents/2009_PhD_van_den_Broek.pdf`.

Vilím, P. 2007. Global Constraints in Scheduling. Ph.D. thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Logic, KTIML MFF, Universita Karlova, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic. URL `http://vilim.eu/petr/disertace.pdf`.

Walsh, T. 1999. Search in a Small World. *Proceedings of the 16th International Joint Conference on Artificial Intelligence - IJCAI'99*. 1172–1177.

Watson, J-P., L. Barbulescu, A. E. Howe, L. D. Whitley. 1999. Algorithm Performance and Problem Structure for Flow-shop Scheduling. *Proceedings of the 16th National Conference on Artificial Intelligence - AAAI'99*. 688–695.

Watson, J-P., J. C. Beck. 2008. A Hybrid Constraint Programming / Local Search Approach to the Job-Shop Scheduling Problem. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems-CPAIOR'08. LNCS No. 5015*. 263–277.

Wismer, D. A. 1972. Solution of the Flowshop-Scheduling Problem with No Intermediate Queues. *Operations Research* **20** 689–697.

Yamada, T., R. Nakano. 1992. A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems. *Parallel Problem Solving from Nature 2 (PPSN-II)*. 283–292.