ELSEVIER

Artificial Intelligence 86 (1996) 1-41

Artificial Intelligence

Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem *

Norman Sadeh [a,*], Mark S. Fox [b,1]

[a] *School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3891, USA*

[b] *Department of Industrial Engineering, University of Toronto, 4 Taddle Creek Road, Toronto, Ont., Canada M5S 1A4*

Received April 1992; revised August 1995

Abstract

Practical constraint satisfaction problems (CSPs) such as design of integrated circuits or scheduling generally entail large search spaces with hundreds or even thousands of variables, each with hundreds or thousands of possible values. Often, only a very tiny fraction of all these possible assignments participates in a satisfactory solution. This article discusses techniques that aim at reducing the effective size of the search space to be explored in order to find a satisfactory solution by judiciously selecting the order in which variables are instantiated and the sequence in which possible values are tried for each variable. In the CSP literature, these techniques are commonly referred to as *variable* and *value ordering* heuristics. Our investigation is conducted in the job shop scheduling domain. We show that, in contrast with problems studied earlier in the CSP literature, generic variable and value heuristics do not perform well in this domain. This is attributed to the difficulty of these heuristics to properly account for the *tightness* of constraints and/or the *connectivity of the constraint graphs* induced by job shop scheduling CSPs.

A new probabilistic framework is introduced that better captures these key aspects of the job shop scheduling search space. Empirical results show that variable and value ordering heuristics derived within this probabilistic framework often yield significant improvements in search efficiency and significant reductions in the search time required to obtain a satisfactory solution.

The research reported in this article was the first one, along with the work of Keng and Yun (1989), to use the CSP problem solving paradigm to solve job shop scheduling problems. The suite of benchmark problems it introduced has been used since then by a number of other

* This research was supported, in part, by the Advanced Research Projects Agency under contract #F30602-88-C-0001 and #F30602-91-F-0016, and in part by grants from McDonnell Aircraft Company and Digital Equipment Corporation.

* Corresponding author. E-mail: sadeh@cs.cmu.edu.

[1] E-mail: msf@ie.utoronto.ca.

SSDI 0004-3702(95)00098-4

researchers to evaluate alternative techniques for the job shop scheduling CSP. The article briefly reviews some of these more recent efforts and shows that our variable and value ordering heuristics remain quite competitive.

1. Introduction

Practical constraint satisfaction problems (CSPs) such as design problems (e.g. [31, 49]) or scheduling problems (e.g. [10,39,52]) generally entail large search spaces with hundreds or even thousands of variables, each with several hundred or thousand possible values. Often, only a very tiny fraction of all these possible assignments participates in a satisfactory solution, potentially requiring prohibitive amounts of search before one such solution can be found. This article discusses techniques that aim at reducing the effective size of the search space to be explored by judiciously selecting the order in which variables are instantiated and the sequence in which possible values are tried for each variable. In the CSP literature, these techniques are commonly referred to as *variable* and *value ordering* heuristics. Our investigation is conducted in the job shop scheduling domain [2,12,26].

Specifically, we study a class of job shop scheduling problems in which operations have to be performed within non-relaxable time windows [11,39,42,44]. We refer to this class of problems as the job shop constraint satisfaction problem or job shop CSP. Examples of job shop CSPs include factory scheduling problems, in which some operations have to be performed within one or several shifts, spacecraft mission scheduling problems, in which time windows are determined by astronomical events over which we have no control, factory rescheduling problems, in which a small set of operations need to be rescheduled without disturbing the schedule of other operations, etc. When solving a job shop CSP, the objective is to find as quickly as possible a feasible schedule, namely a schedule where each operation is performed within one of its legal time windows and no resource is oversubscribed. The techniques presented in this paper have also been adapted to solve just-in-time job shop scheduling problems, where the objective is to reduce the sum of tardiness and inventory costs of a set of jobs to be processed subject to relaxable due dates [39,40].

The job shop CSP can easily be shown to be NP-complete [14]. Accordingly, the worst-case complexity of any procedure to solve this problem is expected to be exponential. At the time we started this study, CSP techniques that interleave search with consistency enforcing mechanisms and variable/value ordering heuristics had been reported to yield important increases in search efficiency when applied to a number of different CSPs [6,9,11,13,17,23,29,37,51]. One of the objectives of our study was to determine if similar results could be obtained on large-scale tightly connected problems such as those found in the job shop scheduling domain.

In this article, we first review generic variable and value ordering heuristics that have been reported to perform well on other classes of CSPs. We explain why these heuristics are unlikely to perform as well on large-scale tightly connected CSPs like job shop scheduling. In particular, we show that these heuristics fail to adequately account for

the *tightness of constraints* and/or for the interactions induced by the high *connectivity* of the constraint graphs characteristic of job shop CSPs.[2] The second part of this paper introduces a probabilistic framework, within which new variable and value ordering heuristics are defined that attempt to better account for these interactions. Empirical results indicate that our new heuristics outperform both generic CSP heuristics as well as more specialized heuristics recently developed for resource- and time-constrained CSPs [20]. Our study suggests that a key to defining these more powerful heuristics lies in the ability of the probabilistic framework to provide estimates of the *reliance* of a variable on the availability of one of its remaining values (e.g., in job shop scheduling, the reliance of an operation on the availability of a reservation) and measures of *contention* between variables for the allocation of incompatible values (e.g., in job shop scheduling, measures of *resource contention* between unscheduled operations).

While our work shows that the CSP problem solving paradigm does scale up to complex large-scale domains such as the job shop CSP, it also suggests that benchmark problems considered in earlier CSP studies are not representative of this and probably other classes of complex CSPs. We hope that this research will prompt others in the field to revisit earlier studies and look for more challenging problems on which to evaluate their techniques.

The balance of this paper is organized as follows. Section 2 provides a formal definition of the job shop scheduling CSP. Section 3 details the backtrack search procedure used in our study. Generic variable and value ordering heuristics are reviewed in Sections 4 and 5 respectively. Section 6 describes new variable and value ordering heuristics based on a probabilistic model of the search space. The complexity of these heuristics is discussed in Section 7. Empirical results comparing our new heuristics with other heuristics discussed in this paper are presented in Section 8. The work reported in this article was the first one, along with that of Keng and Yun [20], to use the CSP problem solving paradigm to solve job shop scheduling problems. The suite of benchmark problems it introduced has since then been used by a number of other researchers to evaluate alternative techniques for the job shop scheduling CSP. Section 8 briefly reviews some of these more recent efforts and shows that our variable and value ordering heuristics remain quite competitive. Section 9 provides a summary of the paper and further discusses the implications of this study.

Earlier variations of the techniques presented in this paper are discussed in [11,38, 39,41–44].

2. The job shop constraint satisfaction problem

The job shop CSP requires scheduling a set of jobs $J = \{j_1, \ldots, j_n\}$ on a set of physical resources $RES = \{R_1, \ldots, R_m\}$. Each job j_l consists of a set of operations $O^l = \{O^l_1, \ldots, O^l_{n_l}\}$ to be scheduled according to a process routing that specifies a

[2] Constraint graphs are graphical representations of binary CSPs (i.e. CSPs with binary constraints) in which each variable is represented by a node, and binary constraints are represented by arcs between two nodes. Other graphical representations also exist for non-binary CSPs.

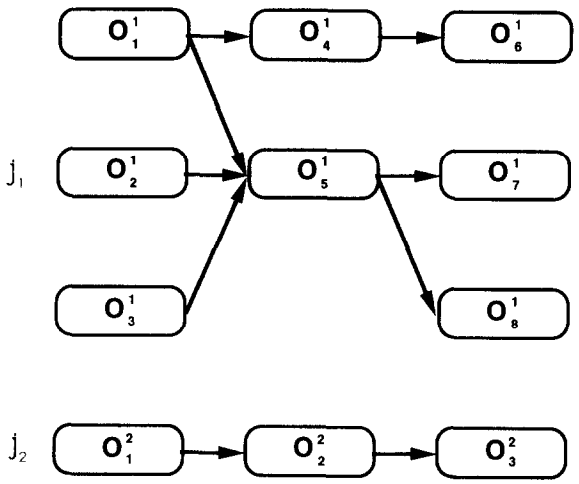

Fig. 1. Examples of tree-like process routings.

partial ordering among these operations (e.g. O_i^l BEFORE O_j^l). This study assumes job shop CSPs with *tree-like* process routings. A tree-like process routing is one whose graph of precedence constraints forms a tree (see Fig. 1). This is by far the most common situation, especially in factory scheduling. Extensions of the techniques presented in this paper to more general types of process routings will be briefly discussed as well.

In the job shop CSP studied in this paper, each job j_l has a release date rd_l and a due date (or deadline) dd_l between which all its operations have to be performed. Each operation O_i^l has a fixed duration du_i^l and a start time st_i^l whose value has to be selected. The domain of possible start times of each operation is initially constrained by the release and due dates of the job to which the operation belongs. If necessary, the model allows for additional unary constraints that further restrict the set of admissible start times of each operation, thereby defining one or several time windows within which an operation has to be carried out (e.g. a specific shift in factory scheduling). In order to be successfully executed, each operation O_i^l requires p_i^l different resources (e.g. a milling machine, a set of fixtures and a machinist) R_{ij}^l $(1 \leqslant j \leqslant p_i^l)$, for each of which there may be a pool of physical resources from which to choose,

$$\Omega_{ij}^l = \{r_{ij1}^l, \ldots, r_{ijq_{ij}^l}^l\},$$

with $r_{ijk}^l \in RES$ $(1 \leqslant k \leqslant q_{ij}^l)$ (e.g. several possible milling machines).

More formally, the problem can be defined as follows:

Variables

A vector of variables (or aggregate variable) is associated with each operation, O_i^l $(1 \leqslant l \leqslant n,\ 1 \leqslant i \leqslant n_l)$, which consists of

(1) the *operation start time*, st_i^l, and

(2) its *resource requirements*, R_{ij}^l $(1 \leqslant j \leqslant p_i^l)$.

In our search procedure, each operation is considered a single (aggregate) variable whose start time and resource requirements are simultaneously instantiated. A tuple of instantiations associated with an operation, namely a start time and a set of specific resource assignments, is referred to as a *reservation* for that operation.

Constraints

The non-unary constraints of the problem are of two types:

(1) *Precedence constraints* defined by the process routings translate into linear inequalities of the type: $st_i^l + du_i^l \leqslant st_j^l$ (i.e. O_i^l BEFORE O_j^l).

(2) *Capacity constraints* that restrict the use of each resource to only one operation at a time translate into disjunctive constraints of the form: $(\forall p \forall q\ R_{ip}^k \neq R_{jq}^l) \vee st_i^k + du_i^k \leqslant st_j^l \vee st_j^l + du_j^l \leqslant st_i^k$. These constraints simply express that, unless they use different resources, two operations O_i^k and O_j^l cannot overlap.

Additionally, there are unary constraints restricting the set of possible values of individual variables. These constraints include non-relaxable due dates (or deadlines) and release dates, between which all operations in a job have to be scheduled. The model actually allows for any type of unary constraint that further restricts the set of possible start times of an operation. As a result, the domain of possible start times of an operation will generally consist of one or several non-contiguous time windows within which the operation has to start. Time is assumed discrete, i.e. operation start times and end times can only take integer values. Finally, each resource requirement R_{ij}^l has to be selected from a set of resource alternatives $\Omega_{ij}^l \subset RES$.

Objective

In the job shop CSP studied in this paper, the objective is to come up with a feasible solution as fast as possible. Notice that this objective is different from simply minimizing the number of search states visited. It also accounts for the time spent by the system deciding which search state to explore next.

Example

Fig. 2 depicts a simple job shop scheduling problem with four jobs $J = \{j_1, j_2, j_3, j_4\}$ and four physical resources $RES = \{R_1, R_2, R_3, R_4\}$. In this example, each operation has a single resource requirement with a single possible value. Operation start times are the only variables. For the sake of simplicity, it is assumed that all operations have the same duration, namely three time units, that all jobs are released at time 0 and have to be completed by time 15 (the minimum makespan of this problem).[3] None of these simplifying assumptions are required by the techniques that will be discussed: jobs usually have different release and due dates, operations can have different durations, several resource requirements, and several alternatives for each of these requirements. However simple, this example will often prove sufficient to contrast the merits of a number of heuristics discussed in this paper. When appropriate, we will consider slight

[3] The makespan of a schedule is the length of the time interval that spans from the earliest operation start time to the latest operation end time [2].

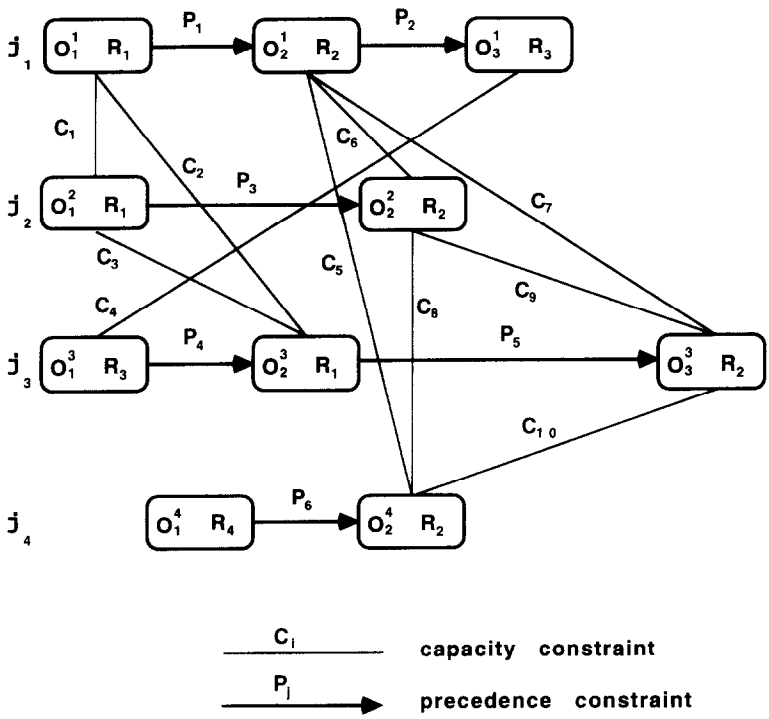

Fig. 2. A simple job shop problem with four jobs. Each node is labeled by the operation that it represents and the resource required by this operation.

variations of this base problem to discuss issues that would not be immediately visible otherwise.

Notice that, in this problem, resource R_2 is the only one to be required by four operations (one from each job). Since all operations in the example have the same duration, resource R_2 is expected to be a small bottleneck.[4]

3. The search procedure

A general paradigm for solving CSPs relies on the use of depth-first backtrack search [3, 16, 35, 50]. Variables or groups of variables (i.e. subproblems) are successively instantiated. Each time a new variable (or group of variables) is instantiated, a new search state is created that corresponds to a new, more complete, partial solution. This process goes on until either a complete solution is obtained or until a deadend state is reached. A deadend state is one whose partial solution cannot be completed without violating one or several problem constraints. When in a deadend state, the procedure

[4] Informally, a bottleneck is a resource or group of resources whose utilization is expected to be close to or larger than its available capacity.

has to undo one or several assignments and try alternative ones, if there are any left (otherwise the problem is infeasible). This process of undoing earlier assignments is known as *backtracking*. It results in lower search efficiency, and, hence, is undesirable.

In the *worst case*, exponential amounts of backtracking may be necessary to come up with a feasible solution (schedule). In practice, the *average complexity* of the procedure can be improved by interleaving search with the application of consistency enforcing mechanisms and variable/value ordering heuristics:

- *Consistency enforcing (checking) techniques*: These techniques prune the search space by eliminating local inconsistencies that cannot participate in a global solution [23]. This is done by inferring new constraints and adding them to the current problem formulation. If, during this process, the domain of a variable becomes empty, a deadend situation has been identified.
- *Variable/value ordering heuristics*: These heuristics are concerned with the order in which variables are instantiated and values assigned to each variable. As discussed in the remainder of this paper, these heuristics can have a great impact on search efficiency.

In this study, we consider a depth-first search procedure that starts in a state where all operations still have to be scheduled and proceeds by scheduling operations one by one (Fig. 3). Each time an operation is scheduled, a new search state is created in which a consistency enforcing procedure updates the set of possible reservations of unscheduled operations to account for the latest assignment. Next, the procedure determines which operation to schedule (variable ordering) and which reservation to assign to that operation (value ordering). The procedure goes on, recursively calling itself, until either all operations have been successfully scheduled or until an inconsistency (or deadend) is detected. In the latter case, it needs to undo one or several earlier decisions (i.e. backtrack). If there are no decisions left to undo (i.e. the procedure is back in the initial search state), the problem is infeasible and the procedure terminates.

The results reported in this study were obtained using a simple chronological backtracking mechanism that systematically goes back to the most recently scheduled operation and tries alternative reservations for that operation. If no alternative reservation is left, the procedure goes back to the next most recently scheduled operation and so on.

1. If all operations have been scheduled then stop, else go on to 2.
2. Apply the *consistency enforcing* procedure.
3. If a deadend is detected then *backtrack* (i.e. select an alternative if there is one left and go back to 1, else stop and report that the problem is infeasible), else go on to step 4.
4. Select the next operation to be scheduled (*variable ordering* heuristic).
5. Select a promising reservation for that operation (*value ordering* heuristic).
6. Create a *new search state* by adding the new reservation assignment to the current partial schedule. Go back to 1.

Fig. 3. Depth-first backtrack search procedure.

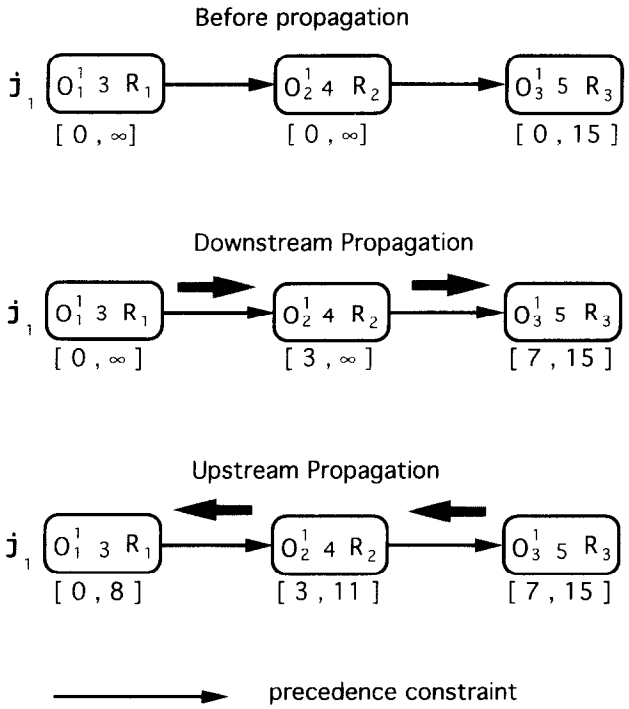

Fig. 4. Consistency with respect to precedence constraints.

Consistency enforcing in our procedure combines three types of computations:

(1) *Consistency with respect to precedence constraints*: Consistency with respect to precedence constraints is maintained using a longest path procedure that incrementally updates, in each search state, a pair of earliest/latest possible start times for each unscheduled operation. Essentially, as in PERT/CPM [18], earliest start time constraints are propagated downstream within the job whereas latest start time constraints are propagated upstream (Fig. 4). The complexity of this simple propagation mechanism is linear in the number of precedence constraints. In the absence of capacity constraints (e.g. problems in which no two operations require the same resource), the procedure can be shown to guarantee decomposability [5,8], i.e., if applied in each search state, it is sufficient to guarantee backtrack-free search [39].

(2) *Forward consistency checks with respect to capacity constraints*: Enforcing consistency with respect to capacity constraints is more difficult due to the disjunctive nature of these constraints. Whenever a resource is allocated to an operation over some time interval, a "forward checking" mechanism [17,29] checks the set of remaining possible reservations of other operations requiring that same resource, and removes those reservations that would conflict with the new assignment, as first proposed in [21] (see Fig. 5).

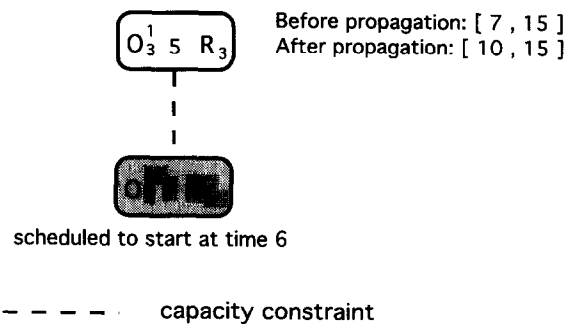

Fig. 5. Forward consistency checks with respect to capacity constraints.

(3) *Additional consistency checks with respect to capacity constraints*: Additionally, our consistency enforcing mechanism checks that no two unscheduled operations require overlapping resource/time intervals. An example of such a situation is illustrated in Fig. 6, where two operations requiring the same resource, O_i^k and O_j^l, rely on the availability of overlapping time intervals. Whichever start

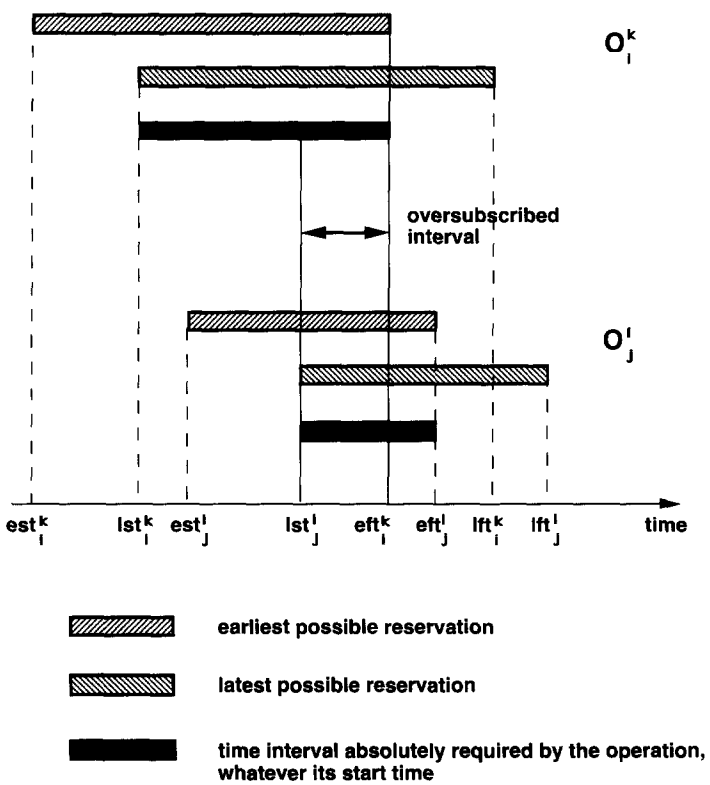

Fig. 6. Detecting situations where two unscheduled operations requiring the same resource are in conflict.

time is selected for operation O_i^k (within its earliest/latest start time window), this operation will always require its resource over the time interval that spans between its latest start time and its earliest finish time ($[lst_i^k, eft_i^k]$). Similarly, operation O_j^l will always require that same resource between its latest start time and its earliest finish time (interval $[lst_j^l, eft_j^l]$). Interval $[lst_i^k, eft_i^k]$ and $[lst_j^l, eft_j^l]$ overlap. This represents a capacity constraint conflict. This additional consistency mechanism, which enforces a higher level of arc-consistency than forward checking [17, 24, 29], has been shown to often increase search efficiency, while only resulting in minor computational overheads [39].

Because it is only possible to efficiently enforce partial consistency with respect to capacity constraints, backtracking will sometimes occur. In other words, the scheduling procedure will sometimes reach a search state, in which several unscheduled operations competing for a resource appear to each have some possible reservations left, while the total capacity available on the resource is actually insufficient to simultaneously accommodate all these operations. Notice, however, that because consistency enforcement with respect to precedence constraints is sufficient to guarantee decomposability (with respect to precedence constraints), backtracking can only occur as the result of capacity constraint violations.

Because it is impossible to efficiently guarantee backtrack-free search for job shop CSPs, variable and value ordering heuristics are generally critical in determining the actual complexity of the search procedure. The next two sections examine popular variable and value ordering heuristics developed for generic CSPs as well as more specialized heuristics and identify key weaknesses of these heuristics when applied to job shop scheduling problems.

4. A look at some popular variable ordering heuristics

A powerful way of reducing the average complexity of backtrack search is to judiciously select the order in which variables are instantiated. The intuition is that, by instantiating difficult variables first, backtrack search will generally avoid building partial solutions that it will not be able to complete later on. This reduces the chances (i.e. the frequency) of backtracking. Instantiating difficult variables first can also help reduce the amount of backtracking when the system is in a deadend state that is not immediately detected by its consistency checking mechanism. Indeed, by instantiating difficult variables, the system moves to more constrained deadend states that are easier to detect. This reduces the time the system wastes attempting to complete partial solutions that cannot be completed.

One can distinguish between two broad types of variable ordering heuristics:

(1) *Fixed variable ordering heuristics*: A unique variable ordering is determined prior to starting the search and used in each branch of the search tree.

(2) *Dynamic variable ordering heuristics*: The ordering is dynamically revised in each search state in order to account for earlier assignments. Different branches in the search tree generally entail different variable orderings.

Clearly, fixed variable orderings require less computation since they are determined once and for all. On the other hand, dynamic variable ordering heuristics are potentially more powerful because they can identify difficult variables within specific search states rather than for the overall search tree. While a number of early CSP studies performed on simple problems such as N-queens or on moderate-size CSPs seemed to suggest that dynamic variable ordering heuristics might be too expensive, Purdom showed that there are more difficult CSPs, for which dynamic variable ordering heuristics can be expected to achieve exponential savings in the average amount of search required to come up with a solution [37]. For these more difficult classes of problems, the CSP literature generally recommends using a simple heuristic known as *dynamic search rearrangement* (DSR) [3,6,7,15,37]. In each search state, DSR looks for the variable with the smallest number of remaining values, and selects this variable to be instantiated next. DSR has often been used as a benchmark to determine whether it is worthwhile using a dynamic variable ordering heuristic for a given class of problems. Not only do the experiments presented at the end of this paper clearly show that job shop scheduling belongs to the class of more difficult problems for which a dynamic variable ordering is justified, they also clearly indicate that DSR is too weak a heuristic for the job shop CSP.

The scheduling problem introduced in Fig. 2 helps understand the shortcomings of this variable ordering heuristic. Fig. 7 depicts this problem after application of the consistency enforcing procedure described in Section 3.

According to DSR, six operations appear to be equally good candidates to be scheduled first, namely O_1^1, O_2^1, O_3^1, O_1^3, O_2^3, and O_3^3, as they each have seven possible start times (values) left, while the other four operations have ten possible start times (values). It is easy to see however that, among these six "critical" operations, some are in fact more difficult to schedule than others. Consider operations O_2^1 and O_3^3. Both require resource R_2, which is required by a total of four operations. Additionally, three out of the four operations requiring resource R_2 are the last operations in their jobs. In other words, most of these operations appear to be in contention for resource R_2 at about the same time. This high contention for resource R_2 strongly suggests that O_2^1 and O_3^3 are more difficult to schedule than the other four operations with seven possible start times. For instance, an operation like O_1^3, which has also seven possible start times, competes only with one other operation for resource R_3, namely operation O_3^1. Additionally, O_1^3 is the first operation in job j_1, while O_3^1 is the last operation in its job (job j_3). In other words, these two operations are not in high contention for their resource (resource R_3), and hence are expected to be easier to schedule than operations O_2^1 and O_3^3. Unfortunately, DSR cannot account for these observations. It simply counts the number of remaining values of each variable, but *fails to estimate the likelihood that these values remain available later on*. Clearly start times of operations competing for highly contended resources are more likely to become unavailable than those of other operations.

In this example, the bottleneck resource R_2 also corresponds to the largest clique of capacity constraints. Therefore, a variable ordering heuristic that identifies difficult variables (i.e. nodes in the constraint graph) as those with many incident constraints might actually perform better than DSR. Several such variable ordering heuristics have been proposed in the literature. These heuristics are generally fixed variable ordering

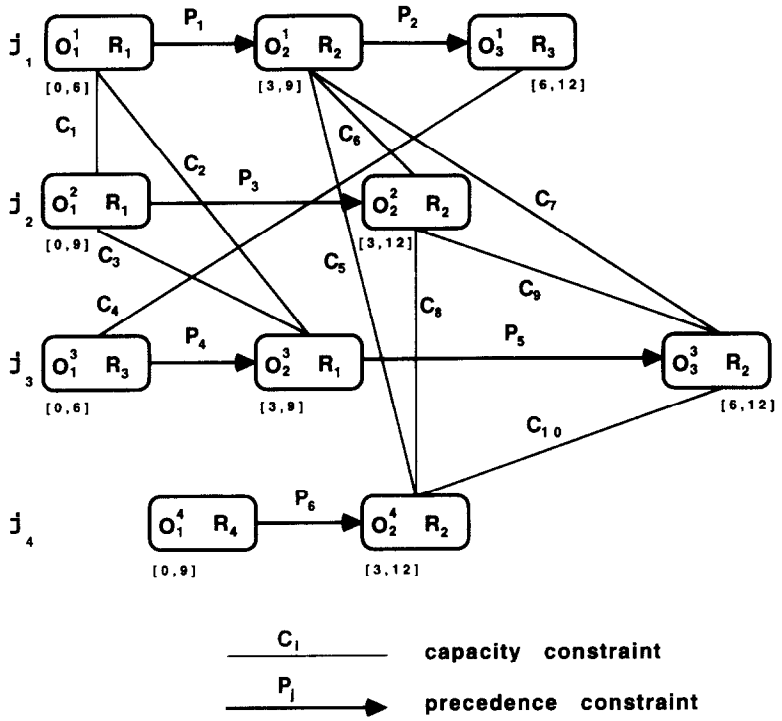

Fig. 7. The same job shop CSP after consistency labeling. Start time labels are represented as intervals. For instance, [0,6] represents all start times between time 0 and time 6, as allowed by the time granularity, namely $\{0,1,2,3,4,5,6\}$.

heuristics, unless new constraints are added to the problem as it is solved. One such heuristic is the *minimum width* (MW) heuristic [6, 13]. MW *orders the variables from last to first by selecting, at each stage, a node in the constraint graph which has a minimal degree*[5] *in the graph remaining after deleting all nodes that have already been selected* [6]. A variation of this heuristic known as the *minimum degree* (MD) heuristic simply ranks variables according to their degree in the initial constraint graph [6]. In the example depicted in Fig. 2, MD would select O_2^1 to be scheduled first. There are also MW orderings starting with this operation. In general, scheduling problems are not that simple, and fixed variable ordering heuristics like MD or MW do not provide very good advice either. This is best illustrated by slightly modifying the scheduling problem depicted in Fig. 2.

Suppose, for instance, that we change the problem and introduce a fifth resource, say R_5. Suppose also that we allow any of the operations requiring R_1 or R_3 in the original problem to use R_5 as an alternative resource. We now have:

- $\Omega_{11}^1 = \Omega_{11}^2 = \Omega_{21}^3 = \{R_1, R_5\}$,
- $\Omega_{31}^1 = \Omega_{11}^3 = \{R_3, R_5\}$.

[5] The degree of a node is the number of constraints incident to that node.

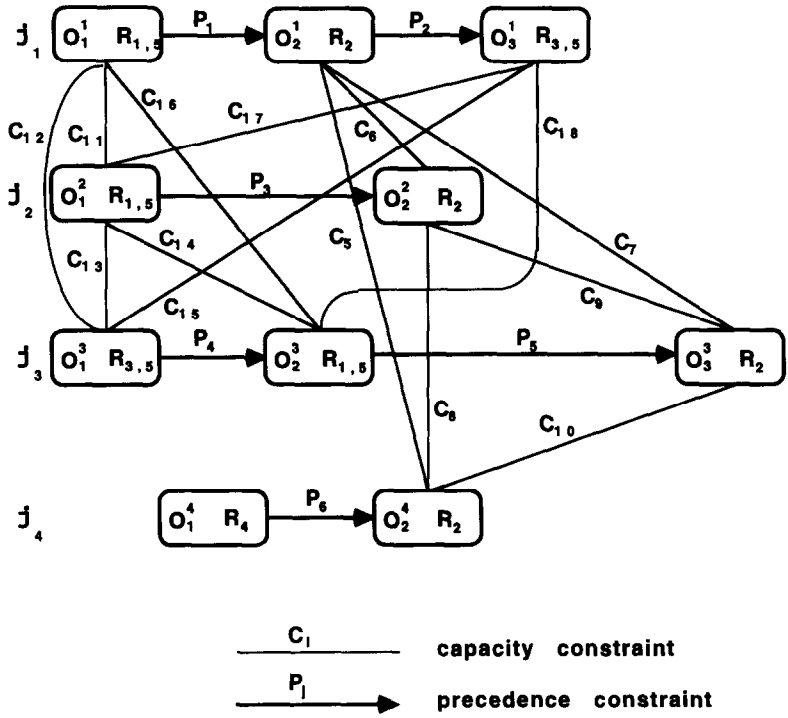

Fig. 8. A new resource R_5 is added to the problem. $R_{1,5}$ stands for R_1 or R_5. $R_{3,5}$ stands for R_3 or R_5.

The two cliques of capacity constraints corresponding to R_1 and R_3 are now subsumed by a larger clique of capacity constraints involving five operations: O_1^1, O_3^1, O_1^2, O_1^3, and O_2^3 (Fig. 8). Notice that capacity constraints between operations belonging to the same job are subsumed by precedence constraints in that job. This is the case for the capacity constraint between O_1^3 and O_2^3, which would require that either O_1^3 precede O_2^3 *or* O_2^3 precede O_1^3, if both operations use R_5. This constraint is subsumed by the precedence constraint between the two operations, which requires that O_1^3 always precede O_2^3. Due to the additional capacity constraints resulting from the introduction of R_5, there are now MW orderings and MD orderings starting with some of these five operations. In fact the addition of R_5 has significantly loosened the capacity constraints participating in the new clique, and the operations connected by these constraints are even easier to schedule than before. Failure of MW and MD to identify that these operations are actually easy to schedule is due to the inability of these heuristics to account for *constraint tightness*, namely the difficulty of satisfying a specific constraint [11,30]. Another example of a variable ordering heuristic that does not account for constraint tightness is the *max cardinality* (MC) search order which arbitrarily selects the first variable to be instantiated, and then at each stage picks the variable connected to the largest number of already instantiated variables [6,24]. This heuristic can also be viewed as a fixed variation of DSR.

Another weakness of generic variable ordering heuristics described in the CSP literature comes from the fact that they treat all problem constraints uniformly. In many practical CSPs, different types of constraints entail different levels of consistency checking. This in turn impacts the effectiveness of different variable ordering heuristics. For instance, in the job shop CSP, consistency enforcing techniques can efficiently ensure that backtracking only occurs as a result of capacity constraint violations, as explained in Section 3. Consequently, the criticality of an operation should solely be a function of how difficult it is to find that operation a reservation that does not violate any capacity constraints. This can be exploited to design more effective variable ordering heuristics.

A specialized variable ordering heuristic that takes advantage of this observation is the one developed by Keng and Yun [20], though its authors apparently failed to relate the strength of their heuristic to this observation. Keng and Yun suggested a generalization of DSR in which each operation reservation (i.e. each value) is assigned a survivability measure reflecting its chance of satisfying the capacity constraints (i.e. its chance of surviving contention with other operations for the allocation of its resource). The operation to be scheduled next is the one with the smallest global survivability, as determined by the sum of the survivabilities of each of its (remaining) possible reservations. Experiments presented at the end of this paper, show that this heuristic performs better than all the generic heuristics described above. They also show that this heuristic is quite expensive, as it requires inspecting all the remaining reservations (i.e. values) of all unscheduled operations.[6] In scheduling problems with several hundred operations or more, each with several hundred possible start times and several possible resources, this heuristic may not be cost effective. More efficient heuristics can be obtained by focusing on one or a small number of cliques of tight capacity constraints, and selecting the operation most likely to violate a constraint in these cliques. A heuristic based on this idea is described in Section 6, which runs faster than Keng and Yun's heuristic while achieving an even higher search efficiency.

5. A look at some popular value ordering heuristics

Another powerful way of reducing the average complexity of backtrack search relies on judiciously selecting the order in which possible values are tried for each variable. A good value ordering heuristic is one that assigns *least constraining values*. A least constraining value is one that is expected to participate in many solutions to the overall problem or, better, one expected to participate in a large number of solutions compatible with the current search state. By first trying least constraining values, the system will generally maximize the number of values left to variables that still need to be instantiated, and hence it it will avoid building partial solutions that cannot be completed.

[6] Notice also that this heuristic may still identify operations with just a few remaining possible reservations as critical while in fact the reservations of these operations may not be in contention with those of any other operations. This could be the case if operation O_1^4 in the example in Fig. 2 had only a small number of possible start times. In fact, consistency enforcing is sufficient to ensure that backtracking will never be caused by this operation, since there is no capacity constraint incident to it.

Attempting to exactly compute the number of global solutions in which a given assignment (or value) participates would be futile as it would require finding all solutions to the problem. Instead, Dechter and Pearl have developed an advised backtracking (*ABT*) value ordering heuristic that relies on tree-like relaxations of the problem to estimate the goodness of a value. A tree-like relaxation of a CSP is one whose constraint graph is a tree that spans some or all the nodes (i.e. variables) of the original CSP. It turns out that, within such relaxations, the number of solutions in which a value participates can be efficiently computed in $O(nk^2)$ steps, where n is the number of variables in the CSP, and k the maximum number of possible values of a variable. The idea is that, if one can find a tree-like relaxation that is close enough to the original CSP, a good value for the relaxation should also be a good value for the original CSP. One way to obtain tight tree-like relaxations is to associate with each (binary) constraint C in the original constraint graph a weight $w(C)$ equal to the satisfiability of that constraint (i.e. the number of value pairs that satisfy the constraint). A tight tree-like relaxation can then be obtained by looking for a minimum spanning tree (MST) in the resulting network.

Even for a fixed variable ordering, this heuristic generally requires the computation of a fixed MST for each of the n levels in the search tree. This amounts to n MST computations, each of which typically requires $O(n^2)$ elementary computations [48], hence a total of $O(n^3)$ elementary computations. Empirical results presented in Section 8 indicate that a fixed variable ordering is generally not enough to efficiently solve job shop scheduling problems. Under these conditions, it might even be necessary to identify new tree-like relaxations in each search state. This in turn would require updating the weights of each constraint in each search state. The resulting computations can become quite expensive for large CSPs. More generally, while ABT has been reported to perform particularly well on some classes of CSPs, it does not seem to lend itself very well to tightly connected CSPs such as job shop scheduling, whether using minimum spanning tree relaxations or not. Indeed, when dealing with tightly connected CSPs such as job shop CSPs, *it is unlikely that one can find a tight tree-like relaxation*, namely a tree-like relaxation that will provide sufficiently good advice to guide search. [7] This is illustrated below with an example.

Consider constraint P_1 in the scheduling problem depicted in Fig. 7. P_1 is a precedence constraint between operation O_1^1 and operation O_2^1. The set of start time pairs (st_1^1, st_2^1) that satisfy constraint P_1 is:

$$\{(0,3),(0,4),\ldots,(0,9),(1,4),(1,5),\ldots,(1,9),\ldots,(6,9)\}.$$

In order to identify a tight tree-like relaxation, P_1 is assigned a weight, $w(P_1)$, equal to the cardinality of that set, namely $w(P_1) = 7+6+5+4+3+2+1 = 28$. Similar computations can be performed to compute the weights of other constraints. These weights are as follows:

[7] The experiments reported in [9] seemed to indicate the opposite. In these experiments, it appeared that often the advice provided by ABT was too expensive and too accurate. Instead, advice provided by looser relaxations ended up being more cost effective. However, these results were obtained on rather small problems with a relatively high density of solutions.

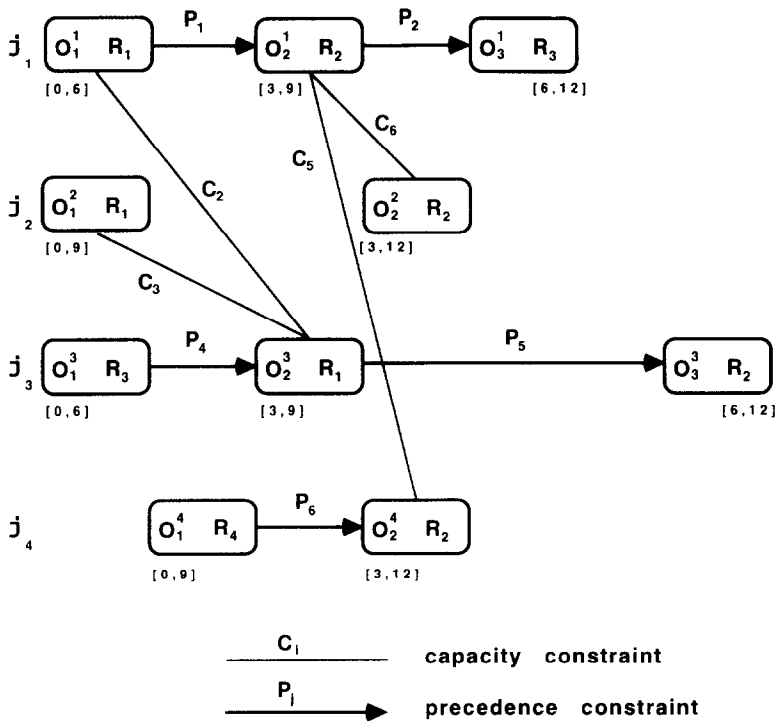

Fig. 9. An MST relaxation of the scheduling problem.

- $w(P_1) = w(P_2) = w(P_4) = w(P_5) = 28$,
- $w(P_3) = w(P_6) = 55$,
- $w(C_1) = 38$, $w(C_2) = 29$, $w(C_3) = 38$,
- $w(C_4) = 43$,
- $w(C_5) = w(C_6) = 38$, $w(C_7) = 29$, $w(C_8) = 56$, $w(C_9) = w(C_{10}) = 38$.

Fig. 9 shows an MST relaxation of the scheduling problem obtained using these weights. It appears that the MST relaxation includes 10 out of the 16 constraints present in the original CSP. The loss of information initially contained in the cliques of capacity constraints is even more dramatic. Only 2 out of the 6 constraints in the clique corresponding to R_2 have been preserved. This is not an accident. *In general a resource required by M operations will result in a clique of* $\binom{M}{2}$ *capacity constraints. At most* $M-1$ *of these capacity constraints can be preserved in any tree-like relaxation of the problem.* Under these conditions, we should not be surprised if the advice provided by ABT for job shop CSPs is not very effective. Suppose for instance that the system selects O_2^1 to be instantiated first.[8] Using the MST relaxation represented in Fig. 9, ABT would recommend assigning start time 4 to this operation. A careful examination of the scheduling problem reveals however that there is no feasible schedule with O_2^1

[8] It should now be clear that this is a good choice, since this operation has only seven possible start times and requires resource R_2, the main bottleneck of the problem.

starting at 4. If O_2^1 were to start at time 4, the other three operations requiring resource R_2 would all have to be scheduled between time 7 and time 15. Given that each of these operations has a duration of three time units, this is clearly impossible.

Keng and Yun have developed a specialized value ordering heuristic that can deal more effectively with cliques of capacity constraints [20]. This heuristic first estimates overall contention for each resource as a function of time. Based on these estimates, operation reservations are ranked according to how well they are expected to prevent contention with the resource requirements of other operations. Empirical results in Section 8 show that Keng and Yun's value ordering heuristic generally outperforms ABT. However their heuristic does not attempt to leave enough room to other operations within the same job so that they too can be assigned least constraining reservations. In other words, Keng and Yun's heuristic only accounts for capacity constraints incident to the current operation, but fails to account for capacity constraints incident to other operations within the same job.

The next section describes a probabilistic model of the search space that better accounts for the high connectivity of constraint graphs typically found in job shop scheduling, and for the constraint interactions induced by these graphs. New variable and value ordering heuristics are defined within this framework that attempt to remedy the shortcomings identified above.

6. New variable and value ordering heuristics

6.1. General considerations

Good variable and value ordering heuristics are heuristics that minimize the time required for search to complete (i.e. either with a solution, if one exists, or with the answer that the problem is overconstrained). If the problem is infeasible, search time is independent of the value ordering heuristic (except for the time spent applying the heuristic); once a variable has been selected, the system has to try each one of its remaining values to conclude that the current partial solution cannot be completed. In general variable and value ordering heuristics affect the number of search states that are explored, the average amount of time spent enforcing consistency in each search state, and the amount of time spent applying these heuristics. Variable and value ordering heuristics can also affect each other's performance. The complexity of these interactions precludes the design of heuristics that directly minimize the expected search time. Instead, our approach aims at developing heuristics that *efficiently reduce the expected number of search states that need to be explored.* Assuming that the time spent enforcing consistency is mainly a function of the number of operations that have already been scheduled (i.e. the depth in the search tree) rather than a function of the specific operations that have been scheduled, this approach is in effect expected to yield heuristics that reduce search time as well.

We postulate that *a critical variable is one that is expected to cause backtracking*, namely one whose remaining possible values are expected to conflict with the remaining possible values of other variables. Under a set of simplifying independence assumptions,

Haralick and Elliott have shown that a variable ordering heuristic based on such a criticality measure will minimize the expected length of branches in the search tree, and hence the total number of search states that need to be visited to come up with a solution [17].[9] We also postulate that a *good value is one that is expected to participate in many solutions* compatible with the current search state.

In the next subsection, we introduce a probabilistic model of the search space, which we will use to compute estimates of variable criticality and value goodness.

6.2. A probabilistic model of the search space

A critical variable is one expected to be involved in a conflict. To approximate variable criticality, we introduce a probabilistic framework that accounts for (1) the chance that a given value will be assigned to a variable (or the *reliance* of a variable on a particular value) and (2) the chances that values assigned to different variables conflict with each other (measures of *value contention*), taking into account only those conflicts that cannot be prevented by the consistency enforcing procedure. Given that the only conflicts that cannot be prevented by our consistency enforcing procedure are capacity constraint violations, a critical operation is one whose resource requirements are likely to conflict with the resource requirements of other unscheduled operations.

We consider a probabilistic model in which each remaining reservation/value ρ of an unscheduled operation O_i^l is assigned a *subjective probability* $\sigma_i^l(\rho)$ to be allocated to that operation. Because, a priori, there is no reason to believe that one reservation is more likely to be selected than another, each operation reservation is assigned an equal probability to be selected. Clearly, in any given schedule, an operation will be assigned only one reservation, hence:

$$\sigma_i^l(\rho) = \frac{1}{NBR_i^l},$$

where NBR_i^l is the number of remaining reservations of O_i^l in the current search state. This distribution mirrors our intuition that an operation with many possible reservations does not heavily *rely* on any one of its remaining reservations, and hence the probability of anyone of these reservations to be selected is rather low. On the other hand, operations with few remaining reservations rely more heavily on each of their remaining reservations.

Using these subjective reservation distributions, we can estimate the *reliance* of an operation O_i^l on the availability of a resource $R_k \in RES$ at time τ as the probability that the reservation allocated to this operation will require that resource at that time. We refer to this probability as the *individual demand* of operation O_i^l for resource R_k

[9] See [17, pp. 307–312]. At the end of their proof, the authors make the unnecessary assumption that each variable value is equally likely to become unavailable. Under this assumption, the variable with what they call the *smallest success probability* (or equivalently the variable most likely to create backtracking) is the one with the smallest number of remaining values. The authors exploit this result to motivate their dynamic search rearrangement heuristic. When this last assumption is omitted, Haralick and Elliott's proof shows that (under several other simplifying assumptions made earlier in their proof) choosing the variable most likely to create backtracking will minimize the expected length of each branch in the search tree.

at time τ and denote it $D_i^l(R_k,\tau)$. $D_i^l(R_k,\tau)$ can simply be computed by adding the probabilities $\sigma_i^l(\rho)$ of all remaining reservations ρ of operation O_i^l that require resource R_k at time τ.

By adding the individual demands of all unscheduled operations requiring resource R_k, an *aggregate demand profile*, $D_{R_k}^{aggr}(\tau)$, is obtained that measures contention between unscheduled operations for resource R_k as a function of time.

Equivalently, if we were to assume a stochastic mechanism that completes the current partial schedule (solution) by randomly assigning a reservation (value) to each unscheduled operation (variable) O_i^l based on its σ_i^l distribution, $D_i^l(R_k,\tau)$ would be the probability that the stochastic mechanism assigns operation O_i^l a reservation that requires R_k at time τ and $D_{R_k}^{aggr}(\tau)$ would be the expected number of reservations for R_k at time τ (or the expected number of operations requiring that resource at that time).

Similar demand profiles are built by Keng and Yun's heuristics [20]. Our variable and value ordering heuristics differ from those of Keng and Yun in the way they exploit these demand profiles.[10] Earlier, Muscettola and Smith also proposed techniques to build probabilistic demand profiles, based on a *predefined variable ordering* [28].

The following illustrates the construction of these profiles for the example introduced in Fig. 2.

Consider operation O_2^1 in the initial search state depicted in Fig. 7. After enforcing consistency, this operation has seven possible reservations (i.e. start times $st_2^1 = 3,4,\ldots,9$), each with a probability $\sigma_2^1(st_2^1) = 1/7$ to be selected. Similarly, O_2^2 has ten possible start times and hence $\sigma_2^2(st_2^2) = 1/10$, $st_2^2 = 3,4,\ldots,12$.

The individual demand of operation O_2^1 for resource R_2 at time t can be obtained by adding the probabilities of all its possible reservations starting between t and $t - du_i^l$:

$$D_2^1(R_2,t) = \sum_{t-du_2^1<\tau\leqslant t} \sigma_2^1(\tau).$$

For instance, $D_2^1(R_2,t) = 1/7$ for $3 \leqslant t < 4$ and $D_2^1(R_2,t) = 2/7$ for $4 \leqslant t < 5$. Similar computations can be performed for the other time intervals over which O_2^1 may require resource R_2. Fig. 10 shows the individual demands of all four operations requiring resource R_2, as well as the aggregate demand for that resource obtained by adding the individual demands of these four operations over time. As expected, the two operations with only seven possible start times (namely O_2^1 and O_3^3) have more compact individual demands than the two operations with ten possible start times (namely O_2^2 and O_4^2). Notice also, that, because of the normalization of the $\sigma_i^l(\rho)$ distributions, the total individual demand of an operation with only one possible resource (like all the operations in this example) is always equal to the duration of that operation. This total demand is simply spread differently over time, depending on the number of start times still available to the operation.

Fig. 11 displays aggregate demands for all four resources in the example. As anticipated, resource R_2 is the most contended for.

[10] The work presented here was performed concurrently with Keng and Yun's [11,41,42]. Notice also that Keng and Yun's interpretation of their demand profiles is not a probabilistic one.

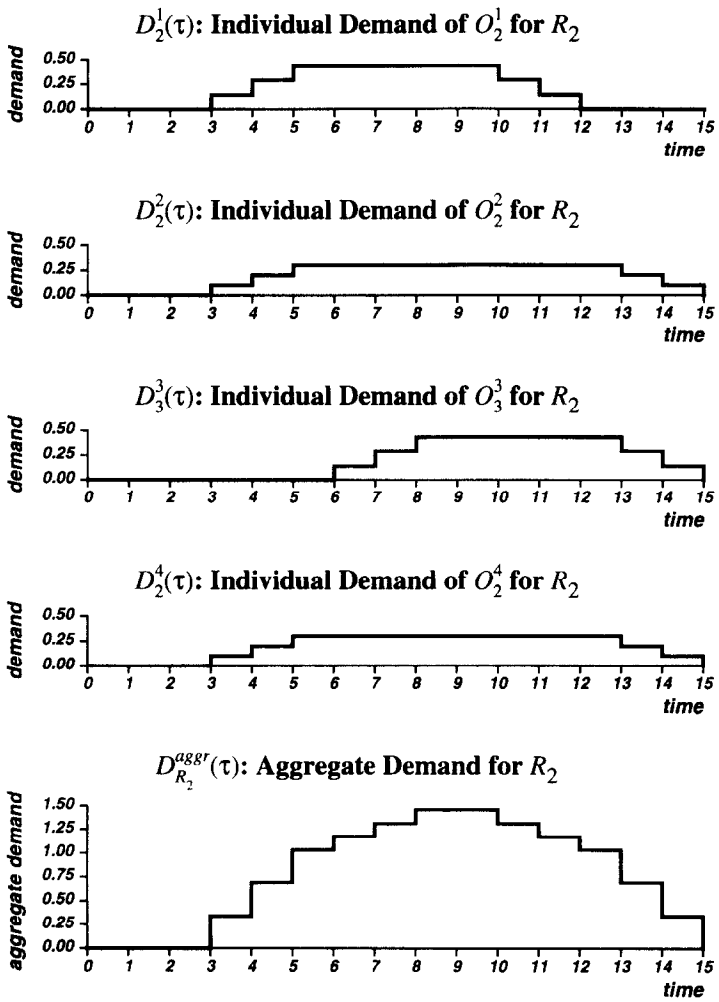

Fig. 10. Building R_2's aggregate demand profile in the initial search state.

In general, building aggregate demand profiles requires looking at each remaining reservation of each unscheduled operation. Hence, in each search state, the complexity of this procedure is $O(Nk)$, where N is the number of unscheduled operations and k the number of remaining reservations of an unscheduled operation. In practice, the sets of remaining reservations of many operations do not change from one search state to another. Accordingly, the computation of demand profiles could potentially be made more efficient by dynamically updating individual demands of operations when their sets of possible reservations shrink (i.e. subtracting their old individual demands from the aggregate demand profiles and adding the new ones). Empirical results presented in Section 8 were obtained using a procedure that did not take advantage of this observation.

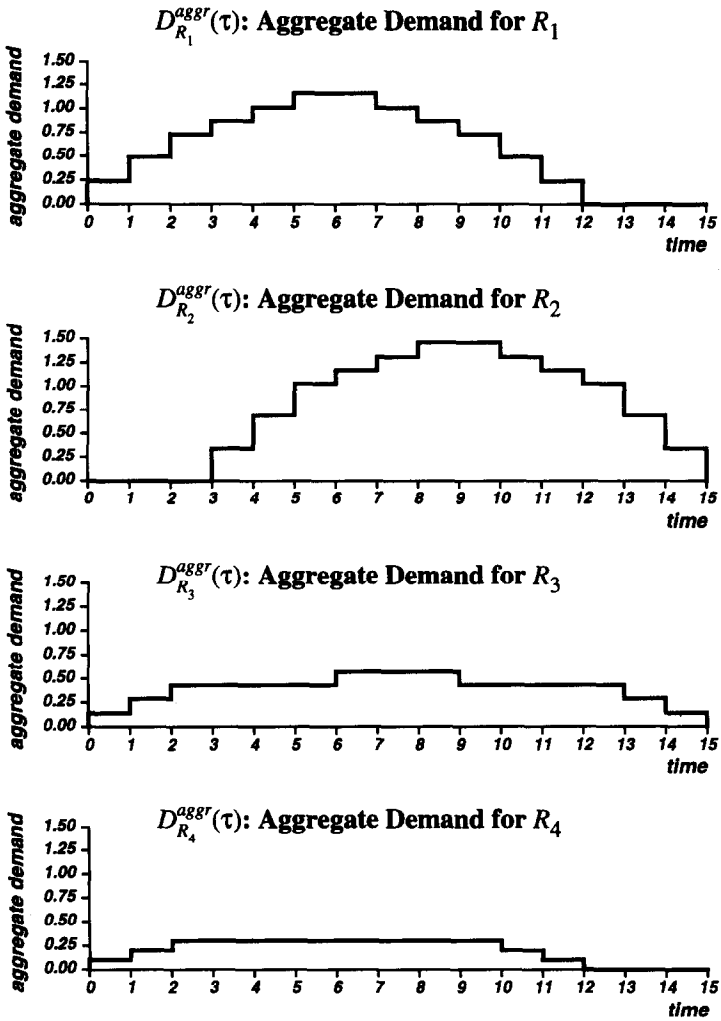

Fig. 11. Aggregate demands in the initial search state for each of the four resources.

6.3. *A variable ordering heuristic based on measures of resource contention*

The aggregate demand for a resource over a time interval is a measure of contention between unscheduled operations for that resource/time interval. The resource/time interval with the highest demand/contention can be expected to be the one where capacity constraints are most likely to be violated (specifically, the capacity constraints that connect operations contributing to the demand for this resource/time interval). Accordingly, *the operation with the highest contribution to the demand for the most contended-for resource/time interval can be considered the most critical one.* This is the operation/variable most likely to be involved in a conflict. It is also the operation that relies most on the availability of the highly contended-for resource/time interval.

Several variations of this variable ordering heuristic have been implemented. The simplest and often most effective one inspects each resource's aggregate demand profile using time intervals of duration equal to the average duration of the operations requiring that resource. The heuristic then picks the resource/time interval with the highest demand and the operation with the largest contribution to this resource/time interval. This is the variable ordering heuristic used in the empirical study reported in Section 8. We refer to it as *ORR* (for "operation resource reliance" heuristic).

Fig. 11 displays the demand profiles of R_1, R_2, R_3, and R_4, the four resources in our example. The highest demand peak is the one on resource R_2 between time 8 and 11. This resource/time interval corresponds precisely to the clique of tight capacity constraints identified earlier. Fig. 12 indicates that the operation with the largest contribution to the demand for this peak is O_3^3. This is no coincidence: O_3^3 competes for the most contended resource and belongs to the group of six operations that have only seven possible start times left after consistency checking. Notice that, in this example, there are actually two intervals in the demand profile of R_2 that qualify as most contended for: $[7, 10[$ and $[8, 11[$. Had our heuristic chosen $[7, 10[$ instead of $[8, 11[$, it would have selected O_2^1 as the operation to be scheduled next. In fact, O_3^3 and O_2^2 appear equally critical in this example.

The ORR heuristic requires looking successively at each resource, and each time interval on that resource's calendar, in order to identify the most contended interval. If there are m resources and if the scheduling horizon is H, this requires $O(Hm)$ elementary computations.

6.4. *A reservation ordering heuristic that attempts to minimize contention*

In Section 4, we showed that the computational overhead associated with ABT's selection of minimum spanning tree relaxations could become prohibitive on large job shop CSPs and is often unlikely to help due to the difficulty of tree-like relaxations to properly account for cliques of capacity constraints. We now describe a value ordering heuristic that attempts to minimize resource contention while relying on predetermined tree-like relaxations. The predetermined tree-like relaxations are comprised of some or all unscheduled operations in the job to which the current critical operation (i.e. the operation to be scheduled next) belongs, along with all precedence constraints between these operations. However, rather than simply counting the number of solutions to these relaxations, *our value ordering heuristic also accounts for the probability that a solution to the relaxation satisfies the cliques of capacity constraints*. The probability that a solution (to the relaxation) satisfies the cliques of capacity constraints (or "survives" resource contention) is estimated using the same demand profiles that are constructed for the ORR variable ordering heuristic.

For job shop CSPs with tree-like process routings, the tree-like relaxation adopted by our value ordering heuristic is comprised of all the unscheduled operations connected by precedence constraints to the current critical operation, along with these precedence constraints. Each candidate reservation (for the critical operation) is ranked according to the number of solutions to the relaxation with which it is compatible that are expected to satisfy capacity constraints (or "survive" resource contention). The reservation compat-

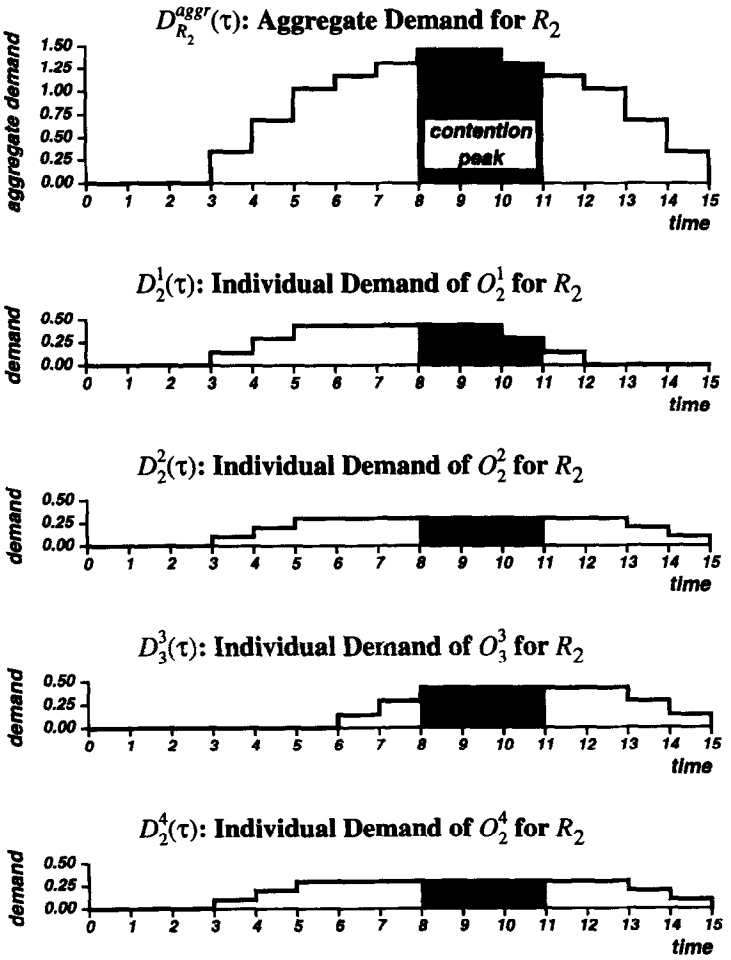

Fig. 12. ORR heuristic: the most critical operation is the one that relies most on the most contended resource/time interval.

ible with the largest number of such schedules is the one selected by our value ordering heuristic. Below, we detail the approximations used in our value ordering heuristic to compute the probability that a reservation and a solution to the relaxation "survive" resource contention.

6.4.1. Estimating the probability that a reservation survives contention

Let O_i^l be an unscheduled operation and $\rho = \langle st_i^l = t, R_{i1}^l = r_{i1k_1}^l, R_{i2}^l = r_{i2k_2}^l, \ldots \rangle$ one of its remaining reservations. We refer to the probability that assigning reservation ρ to operation O_i^l will not conflict with the resource requirements of other operations as the *survivability* of reservation ρ (for O_i^l). It will be denoted $surv_i^l(\rho)$. The survivability of reservation ρ (for O_i^l) is approximated by the product of the probability that each

one of the resources required by that reservation will be available between t and $t + du_i^l$ (independence assumption):

$$surv_i^l(\rho) = \prod_{r_{ijk}^l \in \{r_{i1k_1}^l, r_{i2k_2}^l, \ldots\}} avail_i^l(r_{ijk}^l, t, t + du_i^l), \tag{1}$$

where $avail_i^l(r_{ijk}^l, t, t + du_i^l)$ stands for the probability that resource r_{ijk}^l will not be required by any other operation between t and $t + du_i^l$ (or the probability that assigning this resource to O_i^l will not create backtracking).

Let $r_{ijk}^l = R_p \in RES$. The probability $avail_i^l(r_{ijk}^l, t, t + du_i^l)$ that resource $r_{ijk}^l = R_p$ will not be required by any other operation between t and $t + du_i^l$ can be approximated using the aggregate demand profile of resource R_p introduced in Section 6.2. Our value ordering heuristic also requires keeping track of $n_p(\tau)$, namely the number of (unscheduled) operations competing for R_p at time τ, which is also the number of operations contributing to the aggregate demand for R_p at time τ, denoted $D_{R_p}^{aggr}(\tau)$.

At any time $t \leqslant \tau < t + du_i^l$, there are by definition $n_p(\tau) - 1$ unscheduled operations competing with operation O_i^l for resource R_p. The total demand of these other unscheduled operations for R_p at time τ is $D_{R_p}^{aggr}(\tau) - D_i^l(R_p, \tau)$. Assuming that each of these $n_p(\tau) - 1$ other operations equally contributes to this demand, the probability that none of these operations requires R_p at time τ is given by:

$$\left(1 - \frac{D_{R_p}^{aggr}(\tau) - D_i^l(R_p, \tau)}{n_p(\tau) - 1}\right)^{n_p(\tau)-1}. \tag{2}$$

It is tempting to approximate $avail_i^l(r_{ijk}^l, t, t + du_i^l)$, i.e. the probability that $r_{ijk}^l = R_p$ will be available to O_i^l between t and $t + du_i^l$, as the product of the probabilities that R_p will be available to O_i^l on each one of the du_i^l time intervals between t and $t + du_i^l$. In general, this approximation is too pessimistic. It assumes that the operations competing with O_i^l have a duration equal to 1, i.e. that any of these operations could require R_p over time interval $[\tau, \tau + 1[$ without requiring it over time interval $[\tau + 1, \tau + 2[$ or over time interval $[\tau - 1, \tau[$. Instead, because operations competing for R_p generally require several contiguous time intervals, a better approximation can be obtained by subdividing the calendar of that resource into buckets of duration $AVG(du)$, where $AVG(du)$ is the average duration of the operations competing for $r_{ijk}^l = R_p$. $avail_i^l(r_{ijk}^l, t, t + du_i^l)$ is then approximated as the probability that O_i^l will be able to secure the $(du_i^l)/(AVG(du))$ time buckets that it requires to fit on the resource's calendar. Using Eq. (2), this can be approximated as:

$$avail_i^l(R_p, t, t + du_i^l) = \left(1 - \frac{AVG(D_{R_p}^{aggr}(\tau) - D_i^l(R_p, \tau))}{AVG(n_p(\tau) - 1)}\right)^{AVG(n_p(\tau)-1)\times(du_i^l)(AVG(du))^{-1}}, \tag{3}$$

where $AVG(D_{R_p}^{aggr}(\tau) - D_i^l(R_p, \tau))$ and $AVG(n_p(\tau) - 1)$ are respectively the average of $D_{R_p}^{aggr}(\tau) - D_i^l(R_p, \tau)$ and the average of $n_p(\tau) - 1$ over time interval $[t, t + du_i^l[$.

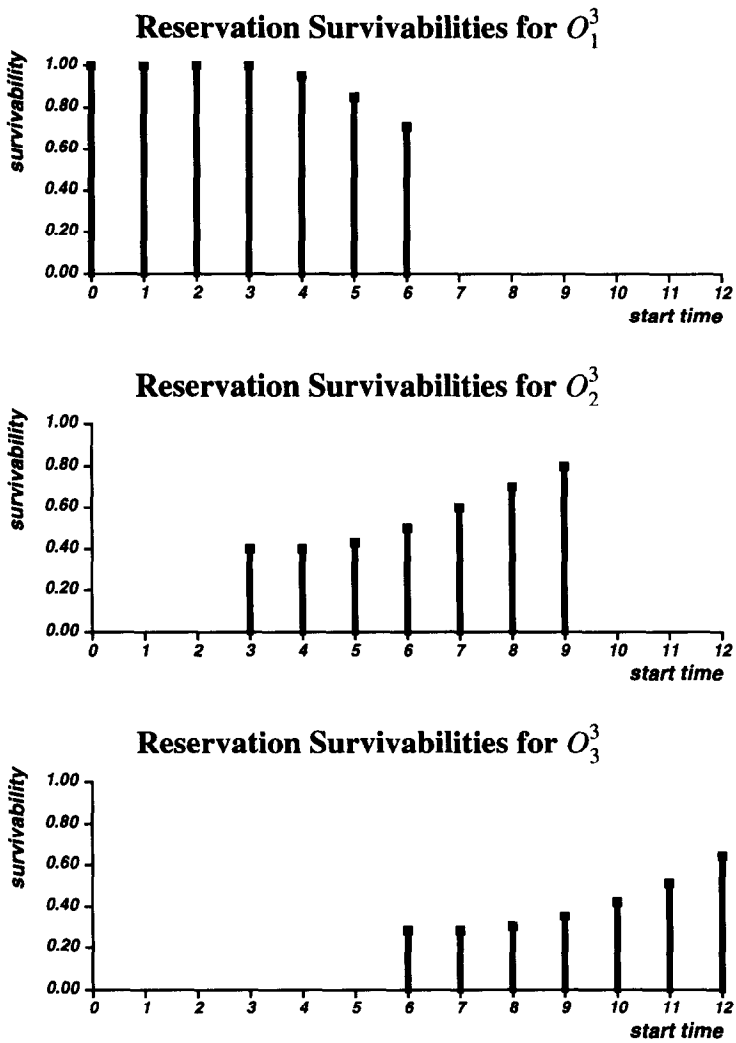

Fig. 13. Survivability measures for the reservations of operations in job j_3, the job to which O_3^3 belongs, the current critical operation.

Fig. 13 depicts reservation survivabilities for the three operations in job j_3, the job to which O_3^3 belongs (the operation selected to be scheduled in the initial search state). The shape of these survivability curves is easily interpreted by looking at Figs. 7 and 11. Consider operation O_1^3. Fig. 7 indicates that O_1^3 only competes with one other operation for resource R_3, namely operation O_3^1. Because operation O_1^3 has a duration $du_1^3 = 3$ and because the earliest possible start time of operation O_3^1 is $st_3^1 = 6$, operation O_1^3 will never conflict with operation O_3^1 if it is scheduled at $st_1^3 = 0$, 1, 2, or 3. This is why the survivability of each of these start times is equal to 1. For start times $st_1^3 = 4, 5$ and 6 the probability of conflicting with a reservation assigned to O_3^1 increases, as indicated in

Fig. 11 by the higher aggregate demand for resource R_3 between time 6 and 9 (the only times where a conflict between the two operations is possible). Since the probability of such a conflict remains fairly low (i.e. the conflicts involve only two operations and only a small fraction of the reservations of these two operations conflict with each other), the survivabilities of start times $st_1^3 = 4, 5$ and 6 remain fairly close to 1 (though smaller than 1). Operations O_2^3 and O_3^3 compete with more operations than O_1^3. Accordingly, their reservation survivabilities are smaller. The shape of the survivability curves of these two operations can be interpreted using similar, though slightly more complex arguments.

6.4.2. Estimating the probability that a job schedule survives contention

A good reservation ρ for a critical operation O_i^l is not just one that is likely to (locally) survive contention for the resources it requires. It should also leave enough room to other unscheduled operations in the same job (job j_l) so that they too have reservations that are likely to survive resource contention. Accordingly, our value ordering heuristic ranks each remaining reservation (of the critical operation) by estimating the number of job schedules compatible with this reservation that are likely to survive resource contention (in short, the *expected number of survivable schedules*). When some operations in the job have already been scheduled, rather than looking at the entire job, it is sufficient to look at the relaxation comprised of all unscheduled operations that can be reached from the current (critical) operation via precedence constraints without visiting a scheduled operation.

The following details the way in which our value ordering heuristic approximates the number of job schedules compatible with a given reservation (for the critical operation) that are expected to survive resource contention. The reader who is not interested in these details can safely jump to Section 7 or 8.

In order to proceed, a few notations need to be introduced:

- O_i^l: the current critical operation (i.e. the operation selected to be scheduled next).
- ρ: one of O_i^l's remaining reservations.
- $RELAX_i^l \subseteq O^l$: the set of operations that make up the relaxation used by our value ordering heuristic. This set consists of O_i^l and the unscheduled operations that can be reached from O_i^l via precedence constraints without visiting a scheduled operation.
- $good_i^l(\rho)$: the goodness of assigning ρ to O_i^l, expressed as the expected number of survivable solutions to the relaxation.
- $comp_i^l(\rho)$: the set of solutions to the relaxation that are compatible with the assignment of ρ to O_i^l.
- $sol \in comp_i^l(\rho)$: a solution to the relaxation that is compatible with the assignment of ρ to O_i^l.
- $\rho(O_k^l \mid sol)$: the reservation assigned to an operation $O_k^l \in RELAX_i^l$ in solution *sol*.

Assuming that the probability that a solution *sol* survives resource contention can be approximated by the product of the probabilities that each reservation $\rho(O_k^l \mid sol)$ in *sol* survives contention, the goodness of assigning ρ to O_i^l is:

$$good_i^l(\rho) = \sum_{sol \in comp_i^l(\rho)} \prod_{O_k^l \in RELAX_i^l} surv_k^l(\rho(O_k^l \mid sol)). \tag{4}$$

This independence assumption is equivalent to omitting the interactions induced by precedence constraints in other jobs. Empirical results reported in Section 8 suggest that this independence assumption is generally acceptable. Thanks to this assumption, the only reservation survivabilities that need to be computed in each search state are those of operations in $RELAX_i^l \subseteq O^l$.

Expression (4) can be rewritten to separate the survivability of reservation ρ from that of other operations in $RELAX_i^l$:

$$good_i^l(\rho) = surv_i^l(\rho) \times \sum_{sol \in comp_i^l(\rho)} \prod_{O_k^l \in RELAX_i^l \setminus \{O_i^l\}} surv_k^l(\rho(O_k^l \mid sol)). \tag{5}$$

This can be further rewritten as:

$$good_i^l(\rho) = surv_i^l(\rho) \times compsurv_i^l(\rho), \tag{6}$$

where $compsurv_i^l(\rho)$ is the number of solutions compatible with the assignment of ρ to O_i^l that are expected to survive contention:

$$compsurv_i^l(\rho) = \sum_{sol \in comp_i^l(\rho)} \prod_{O_k^l \in RELAX_i^l \setminus \{O_i^l\}} surv_k^l(\rho(O_k^l \mid sol)).$$

Note that, in fact, $compsurv_i^l(\rho)$ is only a function of the start time st_i^l allocated to O_i^l in reservation ρ an can therefore be written as $compsurv_i^l(t)$.

In tree-like process routings, it is possible to evaluate $compsurv_i^l(t)$ for all the possible start times t of O_i^l in $O(\nu_l k)$ steps, where $\nu_l \leqslant n_l$ is the number of operations in relaxation $RELAX_i^l$, and k the maximum number of possible reservations of an operation. This is done using a dynamic programming procedure described in Appendix A. This technique is an adaptation of a procedure described in [9]. The complexity of Dechter's procedure is $O(\nu_l k^2)$ for general tree-like CSPs. Here we have further reduced this complexity to $O(\nu_l k)$ by taking advantage of the linearity of precedence constraints. If the model was to allow for other temporal constraints such as those described in [1], the complexity of the algorithm would be $O(\nu_l k^2)$. For non-tree-like process plans, it should be possible to remove a small number of precedence constraints (e.g. precedence constraints that are not on a critical path) to transform the process routing into a tree-like one, and use the resulting relaxation to compute goodness measures.

In the example discussed earlier, the critical operation is O_3^3. Since no operation has been scheduled yet, the relaxation used by the heuristic consists of all three operations in job j_3. Fig. 14 displays the goodness measures computed using (6). Start time $st_3^3 = 6$ for instance is only compatible with one solution to the relaxation, namely a solution in which $st_2^3 = 3$ and $st_1^3 = 0$. Therefore, the goodness of this start time is given by: $good(st_3^3 = 6) = surv_1^3(st_1^3 = 0) \times surv_2^3(st_2^3 = 3) \times surv_3^3(st_3^3 = 6)$. On the other hand, start time $st_3^3 = 7$ is compatible with three solutions to the relaxation, one with $st_2^3 = 3$ and $st_1^3 = 0$, one with $st_2^3 = 4$ and $st_1^3 = 0$, and one with $st_2^3 = 4$ and $st_1^3 = 1$. The survivability of this start time was obtained by adding the survivabilities of each of these three solutions.

Start time $st_3^3 = 12$ is the one compatible with the largest number of survivable solutions to the relaxation. Hence this is the start time selected by the value ordering

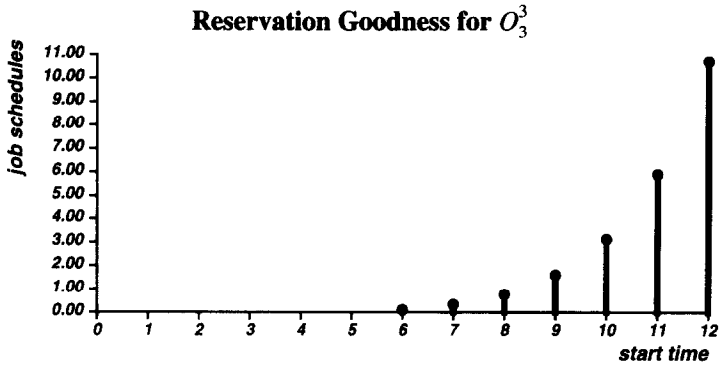

Fig. 14. Value goodness for O_3^3 expressed as the number of compatible job schedules expected to survive resource contention.

heuristic. By assigning this start time to O_3^3, and iteratively using the variable and value ordering heuristics that were just described, the search procedure easily completes the schedule without backtracking. This problem is relatively easy, and is also solved without backtracking by Keng and Yun's heuristic.

No heuristic is perfect. Although our value ordering heuristic recommends the right start time, a careful analysis reveals for instance that its second best choice, namely $st_3^3 = 11$, is actually infeasible. Notice however that, in the absence of backtracking, the scheduler does not need to try the second best value recommended by the heuristic: it is enough for the first value to work.

In Appendix B, we describe a filtering mechanism used in our value ordering heuristic to further refine the ranking of reservations. We refer to the resulting heuristic as the *FSS* value ordering heuristic—FSS stands for "filtered survivable schedules".

7. Overall complexity

In each search state, the worst-case complexity of the look-ahead analysis is $O(\max(Nk, Hm))$, where N is the number of unscheduled operations in the current search state, k the maximum number of reservations left to an operation in that state, H the scheduling horizon, and m the number of resources in the system. In general $O(Nk)$ appears to be the dominant factor. In the absence of backtracking (i.e. the number of search states generated by the system is equal to the number of operations to be scheduled), the overall complexity of the approach is $O(NOP^2k)$, where NOP denotes the total number of operations to be scheduled. Experimentation with problems of different sizes suggests that, in the absence of backtracking, this is the true complexity of the approach. Clearly, when backtracking occurs, the overall complexity of the procedure can be much higher, though empirical results presented in Section 8 show that this is not often the case.

8. Empirical evaluation

This section reports the results of an experimental study comparing the ORR variable ordering heuristic and FSS value ordering heuristic against the DSR (dynamic search rearrangement) variable heuristic (DSR) [3,17,37], the ABT (advised backtracking) value ordering [9], and the combination of variable and value ordering heuristics developed by Keng and Yun [20].

8.1. Design of the test data

A set of 60 scheduling problems was randomly generated, each with five resources and ten jobs of five operations each (i.e. a total of 50 operations per problem). Each job had a linear process routing specifying a sequence in which the job had to visit each one of the five resources. This sequence was randomly generated for each job, except for bottleneck resources, which were each visited after a fixed number of operations (in order to further increase resource contention).

Two parameters were adjusted to cover different scheduling conditions: a *range parameter*, *RG*, controlled the distribution of job due dates and release dates, and a *bottleneck parameter*, *BK*, controlled the number of major bottleneck resources. Six groups of ten problems were randomly generated, by considering three different values of the range parameter and two different bottleneck configurations. The value of a third parameter, which will be referred to as the *slack parameter*, *S*, had to be adjusted as a function of the first two in order to keep demand for bottleneck resource(s) close to 100% over the major part of each problem. If this parameter had been fixed while the other parameters were allowed to change, a large proportion of the problems would have been either trivial or infeasible.

The three parameters were set as follows:

- *Range parameter* (*RG*): this parameter controlled the release date and due date distributions in each problem. Due dates were randomly drawn from a uniform distribution $(1+S)\,M\,U(1-RG,1)$, where $U(a,b)$ represents a uniform probability distribution between a and b, M is an estimate of the minimum makespan of the problem, and S is the slack parameter, which is defined below as a function of BK and RG. The minimum makespan of the problem was estimated as $M = (n-1)\overline{du}_{R_{btnk}} + \sum_{R=R_1}^{R_m} \overline{du}_R$, where n is the number of jobs, m the number of resources, R_{btnk} the main bottleneck resource (or one of them if there are several) and $\overline{du}_{R_i}$ denotes the average duration of the operations requiring resource R_i. This estimate was first suggested in [34]. Similarly, release dates were randomly drawn from a uniform distribution of the form: $(1+S)\,M\,U(0,RG)$. Three values of the range parameter were used to generate problems: $RG = 0.0$, 0.1, and 0.2. Due to the moderate size of the scheduling problems considered here, larger values of RG quickly tend to produce less resource contention. This is also in part due to the fact that, to keep from generating infeasible problems, we increase the value of the slack parameter, S, as RG becomes larger, as detailed below.
- *Bottleneck parameter* (*BK*): in half of the problems, there was only one major bottleneck ($BK = 1$), while in the other half there were two major bottlenecks ($BK = 2$).

- *Slack parameter* (S): for problems with two bottlenecks or jobs with different release dates and due dates, the time span of each problem was inflated to $(1+S)M$ so that most problems remained feasible. The slack parameter was empirically set to $S = 0.1 \times (BK - 1) + RG$. While ensuring that most problems remained feasible, this provided for close to 100% utilization of bottleneck resources over the major part of each problem.

Finally, operation durations were randomly drawn from two different distributions, depending on whether an operation required a bottleneck resource or not. Bottleneck operations had durations randomly drawn from a uniform distribution $U(8, 16)$ whereas non-bottleneck operations had their durations randomly drawn from a uniform distribution $U(3, 11)$. On average, operations in these problems had slightly over 100 possible start times (i.e. values) left after application of the consistency enforcing procedure in the initial search state.

8.2. Comparison with other heuristics

Five combinations of variable and value ordering heuristics were compared:

- *DSR& ABT*: the dynamic search rearrangement heuristic [3] combined with the advised backtracking value ordering heuristic [9]. The version of ABT used in these experiments was one based on the same predetermined tree-like relaxation as FSS, namely it used the process routing to which the current operation belonged. This version of ABT was carefully implemented to run in $O(\nu_l k)$ steps in each search state (where ν_l is the number of operations in the tree-like relaxation and k the maximum number of remaining start times of an operation after consistency checking). This was done using a procedure similar to the one implemented in FSS. Notice that an implementation of ABT using MST relaxations would have been too slow to be competitive. It would have required computing constraint satisfiabilities and identifying an MST relaxation in each search state. Additionally, the time required to count the number of solutions to a general MST relaxation would have been $O(\nu_l k^2)$.
- *DSR& FSS*: the DSR heuristic combined with the filtered survivable schedules (FSS) value ordering heuristic (with $\Phi = 2.5$).
- *ORR& ABT*: the operation resource reliance (ORR) variable ordering heuristic together with the ABT value ordering heuristic.
- *ORR& FSS*: the ORR and FSS heuristics (with $\Phi = 2.5$) advocated in this paper.
- *SMU*: the variable and value ordering heuristics developed by Keng and Yun at the Southern Methodist University [20].

All combinations of variable and value ordering heuristics were run in a modular testbed in which all common functions were shared (e.g. consistency enforcing module, backtracking module, etc.), and unnecessary functions were bypassed whenever possible (e.g. bypassing the construction of demand profiles in DSR&ABT). All functions were implemented with equal care.

On each problem, search was stopped if it required more than 500 search states. The performance of each combination of variable and value ordering heuristics was compared along three dimensions:

(1) *Search efficiency*: the ratio of the number of operations to be scheduled over the total number of search states that were explored. In the absence of backtracking, only one search state is generated for each operation, and hence search efficiency is equal to 1. While a high search efficiency is not necessarily synonymous with a fast procedure, evaluation of heuristics with respect to search efficiency is important. It tells us if a heuristic is doing a good job at focusing search on critical variables and promising values for these variables. In particular, we want to make sure that the probabilistic framework introduced in Section 6 is doing a good job at capturing key constraint interactions that are not well accounted for in generic CSP heuristics. This metric can tell us if this is indeed the case.

(2) *Number of experiments solved* in less than 500 search states each. When a combination of variable and value ordering heuristics cannot solve a given experiment in less than 500 search states, it typically needs thousands of search states to reach a solution. At that point, it does not make sense to let the procedure continue, as it will not return a solution within any reasonable amount of time.

(3) *Average CPU time* (in seconds): this is the average CPU time required to *successfully* schedule a problem. When a solution cannot be found in less than 500 search states, this time is approximated as the CPU time required to explore 500 search states. All CPU times were obtained on a DECstation 5000/200 running Knowledge Craft on top of Allegro Common Lisp. Experimentation with a more recent version of our system written in C++ indicates that the procedure runs about 30 times faster in this language (on the same platform).

The results are summarized in Table 1. They indicate that DSR is generally not sufficient to solve realistic job shop scheduling problems. Combined with ABT, this heuristic was only able to solve 29 problems out of 60 in less than 500 search states each. Even when combined with the FSS value ordering heuristic, DSR only achieved a search efficiency of 58%, and failed to solve 27 problems out of 60 in less than 500 search states. These results not only suggest that job shop scheduling requires a dynamic variable ordering heuristic.[11] They also indicate that the variable ordering heuristics proposed so far in the CSP literature are often too shallow for problems such as job shop scheduling. After replacing DSR with ORR in combination with ABT, search efficiency went up by 16% and 11 additional problems were solved in less than 500 search states each. The SMU heuristic achieved a higher efficiency of 72% and solved 43 problems out of 60 in less than 500 states. Even this heuristic had trouble solving many problems. In fact, it could hardly solve more problems than ORR&ABT. ORR&FSS, the variable and value ordering heuristics advocated in this paper, yielded an impressive 86% search efficiency, and solved 52 problems out of 60 in less than 500 search states. Among the 52 experiments that it was able to solve, ORR&FSS never generated more than 78 search states and never took over 150 CPU seconds to solve a problem. This heuristic combination also achieved important speedups over all the other heuristics.

[11] In [39], we also reported experiments comparing variations of our ORR heuristic that differed in the number of critical operations scheduled at once, namely less dynamic variations of ORR where two or more critical operations are selected at once. These experiments show that the performance of the variable ordering heuristic quickly degrades as it becomes less dynamic.

Table 1
Comparison of five heuristics over six sets of ten job shop problems; standard deviations appear between parentheses

		Performance of five heuristics				
		DSR&ABT	DSR&FSS	ORR&ABT	SMU	ORR&FSS
	Search efficiency	0.72	0.82	0.96	1.00	0.96
		(0.42)	(0.38)	(0.06)	(0.00)	(0.07)
RG = 0.2	Nb. exp. solved	8	8	10	10	10
BK = 1	CPU seconds	524	380	78.5	188	88.5
		(695.5)	(515)	(10.5)	(14)	(13)
	Search efficiency	0.49	0.73	0.54	0.79	0.99
		(0.40)	(0.43)	(0.39)	(0.38)	(0.02)
RG = 0.2	Nb. exp. solved	7	7	6	8	10
BK = 2	CPU seconds	886.5	456.5	566.5	384.5	93
		(819)	(489)	(591.5)	(379.5)	(7.5)
	Search efficiency	0.60	0.82	0.79	0.64	0.78
		(0.44)	(0.38)	(0.36)	(0.46)	(0.36)
RG = 0.1	Nb. exp. solved	7	8	9	6	8
BK = 1	CPU seconds	473.5	266	290	464.5	331.5
		(486.5)	(249)	(416)	(390.5)	(503.5)
	Search efficiency	0.22	0.46	0.31	0.71	0.87
		(0.27)	(0.46)	(0.37)	(0.42)	(0.29)
RG = 0.1	Nb. exp. solved	2	4	4	7	9
BK = 2	CPU seconds	925	483	918	355	184
		(460)	(324)	(575)	(301.5)	(281)
	Search efficiency	0.28	0.32	0.53	0.46	0.73
		(0.38)	(0.38)	(0.44)	(0.46)	(0.43)
RG = 0.0	Nb. exp. solved	2	3	6	4	7
BK = 1	CPU seconds	857	659	832	626	475
		(411)	(379)	(817)	(399.5)	(640.5)
	Search efficiency	0.31	0.37	0.46	0.75	0.82
		(0.33)	(0.43)	(0.40)	(0.41)	(0.38)
RG = 0.0	Nb. exp. solved	3	3	5	8	8
BK = 2	CPU seconds	679.5	615	907	383.5	300.5
		(514)	(420)	(830)	(415)	(444)
	Search efficiency	0.44	0.58	0.60	0.72	0.86
		(0.41)	(0.45)	(0.41)	(0.41)	(0.31)
Overall	Nb. exp. solved	29	33	40	43	52
performance	CPU seconds	724.5	476.5	598.5	400	245.5
		(585.5)	(411.5)	(665.5)	(356.5)	(403.5)

8.3. Recent developments and additional results

The benchmark problems used in this study have been made available to the research community at large through an anonymous ftp account at CMU and have been widely disseminated, providing for the first time a common set of problems in this area. A high point in the history of the benchmark was reached at the AAAI Spring Symposium held

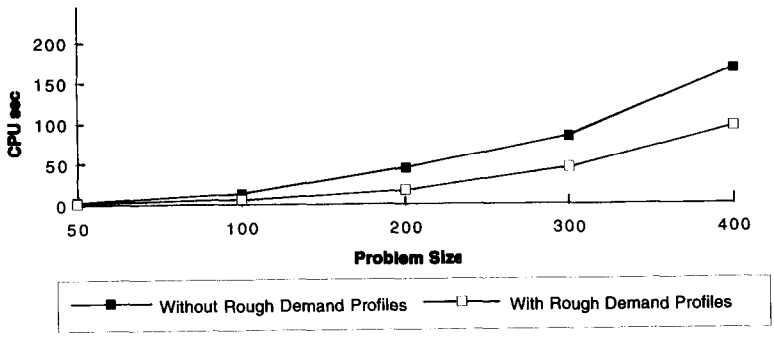

Fig. 15. Scale-up experiments: versions with and without rough demand profiles. Problem sizes are the number of operations to be scheduled. CPU times are on a DECstation 5000/200 running C++.

at Stanford in March 1992, when three groups of researchers simultaneously announced they were able to efficiently solve all 60 problems, using three different approaches:

(1) A bottleneck partitioning approach developed by Muscettola within the context of his HSTS system, in which precedence constraints are imposed between subsets of operations contending for the same resource [27]. Resource contention is approximated via Monte Carlo simulation. If the procedure fails to find a feasible solution, it restarts from scratch, relying on the stochasticity of its Monte Carlo simulation to produce a different solution.

(2) A trial-and-error approach developed by Johnston et al. within the context of the SPIKE system. This approach relies on a collection of initialization heuristics [19]. The schedule produced by the initialization heuristics often violates one or more constraints. When this is the case, it is passed on to a "min-conflict" heuristic that attempts to get rid of conflicts within the solution (or "repair" the solution) [25]. If "min-conflict" fails to produce a feasible schedule within a prespecified number of repair cycles, a new schedule is generated by the initialization heuristics and, if necessary, passed on to "min-conflict" for repair. The specific initialization techniques used in SPIKE have never been published. [27] has reported that "min-conflict" by itself can only solve about 24 out of our 60 benchmark problems. This suggests that the ability of this approach to solve all 60 problems should mainly be attributed to its initialization heuristics.

(3) A procedure we developed that combines our ORR&FSS heuristics with intelligent backtracking mechanisms described in [45,46].

More recently, we reimplemented our heuristics in C++. Most of the problems can be solved in 3 to 4 CPU seconds on a DECstation 5000/200. We were also able to further speed up the computation of demand profiles, using simpler "rough" demand profiles to identify areas of high contention over which the more detailed demand profiles described in Section 6 are then constructed. The rough demand profiles are obtained by evenly spreading the demand of each unscheduled operation between its earliest start time and its latest finish time. Rough demand profiles can easily be updated from one search state to the next and can significantly reduce computation time by focusing the construction of more detailed demand profiles over those areas of highest contention. Using these

rough demand profiles, the CPU time required to solve our 60 benchmark problems fell between 1.5 and 2.5 CPU seconds on a DECstation5000/200, showing that these heuristics remain quite competitive in comparison with more recent techniques proposed for the job shop CSP [22, 33, 47]. The speedups obtained using rough demand profiles also become more significant on larger problems, as illustrated in Fig. 15. Problems with up to several thousand operations have been solved by the procedure with consistently low backtracking.

9. Summary and concluding remarks

In this paper, we studied a variation of the job shop scheduling problem in which operations have to be performed within one or several non-relaxable time windows. We refer to this problem as the job shop CSP. Examples of this formulation of the job shop scheduling problem can be found in the factory scheduling domain when some operations have to be performed within one or several shifts. Other examples include spacecraft mission scheduling problems where activities need to be scheduled within time windows imposed by various astronomical events (e.g. [27]). This formulation is also representative of a number of rescheduling situations where one needs to revise a schedule subject to hard constraints imposed by other operations whose schedule we cannot or would rather not modify. More generally, the job shop CSP formulation can be used to model any scheduling problem with hard deadlines. The job shop CSP cannot be solved with traditional scheduling techniques such as priority dispatch rules or similar one-pass scheduling techniques [12, 26, 39].[12] Traditional mixed integer programming techniques have so far been overwhelmed by the combinatorial number of binary variables required to account for the limited resource capacities in this type of problems [32]. Our work, which, along with that of Keng and Yun [20], was the first one to apply the CSP problem solving paradigm to this class of problems, demonstrates that this paradigm provides a promising alternative to traditional scheduling approaches. Our approach relies on a depth-first backtrack search procedure that combines consistency enforcing mechanisms with a probabilistic look-ahead analysis to decide which operation to schedule next (variable ordering) and which reservation to assign to each operation (reservation ordering).

In the first part of the paper, we reviewed a number of popular variable and value ordering heuristics proposed in the CSP literature, both generic heuristics that had been reported to perform particularly well on other CSPs as well as Keng and Yun's scheduling heuristics [20]. We showed that these heuristics often fail to adequately account for the *tightness of constraints* and/or for the interactions induced by the high *connectivity* of the constraint graphs characteristic of job shop CSPs. In the second part of this article, we introduced a new probabilistic model of the search space that allows us to estimate the reliance of a variable (i.e. an operation) on the availability of a value (i.e. a reservation), and the degree of contention among uninstantiated variables for the assignment of conflicting values (i.e. contention among unscheduled operations for the

[12] See [4] for experiments applying priority dispatch rules to our set of 60 benchmark problems.

allocation of a resource over some time interval). Based on this probabilistic model, new variable and value ordering heuristics were defined:

(1) The "operation resource reliance" (ORR) variable ordering heuristic selects the operation that relies most on the most contended resource/time interval, and
(2) the "filtered survivable schedules" (FSS) value ordering heuristic assigns to that operation the reservation expected to be compatible with the largest number of survivable job schedules, i.e. job schedules that are expected to survive resource contention.

Experimental results show that this pair of heuristics can *efficiently* solve a number of job shop CSPs that could not be efficiently solved by prior CSP heuristics (both generic CSP heuristics and specialized heuristics developed by Keng and Yun). The results indicate that the ORR and FSS heuristics not only yield significant increases in search efficiency but also achieve important reductions in search time.

The estimates of resource contention used in the ORR and FSS heuristics are based on several independence assumptions. More sophisticated versions of these heuristics have also been implemented, which attempt to better account for different dependencies, some using more complex analytical models [41,42,44] others relying on Monte Carlo simulations [43]. The improvements in search efficiency generally achieved by these more sophisticated versions do not seem to justify their heavier computational requirements.

While our ORR and FSS heuristics were developed for the job shop CSP, the probabilistic measures of reliance and contention that were described can be used in any resource allocation problem, and, in fact, any CSP with disequality constraints (i.e. constraints preventing two variables from being assigned the same value), since these problems can be formulated as resource allocation problems. For instance, the N-queens problem often used to evaluate CSP techniques can be formulated as a resource allocation problem in which each queen/row is a task and each column is a resource. [13]

In fact, the lessons learned from this work go beyond job shop scheduling and CSPs with disequality constraints. Fundamental weaknesses of generic variable and value ordering heuristics often praised in the CSP literature have been identified. Variable ordering heuristics like DSR count the number of values left to each variable but do not account for the chances that these values remain available as a solution is constructed. Variable ordering heuristics like MW or MC count the number of constraints incident to a variable but do not account for the tightness of these constraints. Value ordering heuristics like ABT assume that the CSP admits a tight tree-like relaxation. The probabilistic model of the search space introduced in this paper aims at providing a framework in which more sophisticated approximations of variable criticality and value goodness can be defined. For instance, within this framework, our ORR and FSS heuristics can base their decisions on measures of resource contention that account for entire cliques of capacity constraints rather than tree-like relaxations of these cliques.

[13] Constraints representing the ability of queens to attack each other along diagonals can be represented as constraints further restricting admissible resource assignments [20].

Finally, while our work shows that the CSP problem solving paradigm does scale up to complex large-scale domains such as the job shop scheduling CSP, it also suggests that benchmark problems considered in earlier CSP studies are not representative of this and probably other classes of complex CSPs. We hope that this research will prompt others in the field to revisit earlier studies and look for more challenging problems on which to evaluate their techniques.

Appendix A. Counting the number of survivable schedules

This appendix describes a dynamic programming procedure that efficiently counts the number of survivable job schedules (or more generally the number of survivable solutions to the relaxation defined in Section 6.4 for the FSS value ordering heuristic) that are compatible with the assignment of a reservation ρ to the current critical operation O_i^l. This number was referred to as $compsurv_i^l(t)$, where t is the start time allocated to O_i^l in reservation ρ. The procedure presented here is a variation of a similar method developed by Dechter and Pearl for the ABT value ordering heuristic [9] (see also [36]). While a direct generalization of Dechter and Pearl's procedure would have an $O(\nu_l k^2)$ complexity (where ν_l is the number of operations in the relaxation used by the FSS value ordering heuristic, and k the maximum number of possible reservations of an operation in that relaxation), the procedure described here takes advantage of the linearity of precedence constraints to reduce this complexity to $O(\nu_l k)$.

Fig. A.1 represents a prototypical tree-like process routing, which has been reorganized with the current critical operation as the root of the tree. The arrows represent precedence constraints between operations in the process routing. The children of the critical operation O_i^l in the tree are those operations that are directly connected to O_i^l by a precedence constraint, the grandchildren the operations directly connected to these operations by precedence constraints, etc.

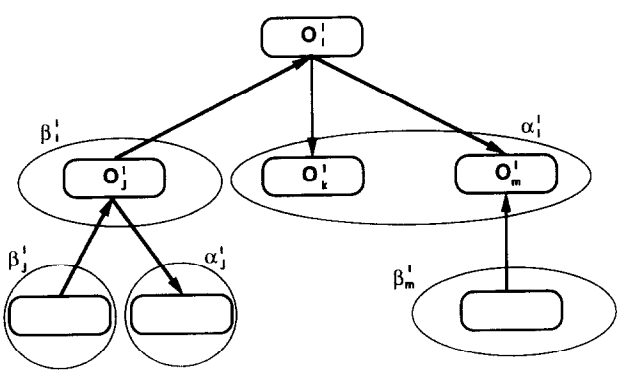

Fig. A.1. A tree-like process routing, organized with the current critical operation as its root. Arrows represent precedence constraints.

All the computations presented in this appendix refer to a single search state, in which consistency checking has already been performed. The notations are those used in Section 6.4. A few extra notations need to be defined:

- $surv_p^l(t) = \sum_{\rho \in G} surv_p^l(\rho)$, where G is the set of remaining reservations of O_p^l with $st_p^l = t$.
- α_p^l: the direct children of O_p^l that are after O_p^l in the process routing.
- β_p^l: the direct children of O_p^l that are before O_p^l in the process routing.
- Δ: the time granularity of the problem. In Section 6.4, it was assumed that $\Delta = 1$ (i.e. that all start times and end times have to be integers). For the sake of clarity, the formulas presented in this appendix account explicitly for Δ.

In tree-like process routings, each operation O_p^l is the unique link between otherwise disjoint sets of operations, that each correspond to one of its children. Each of these sets contains exactly one child of operation O_p^l and defines a subproblem that only interacts with the other subproblems via operation O_p^l. Accordingly:

- For each $O_j^l \in \beta_p^l$, we define $BEF_{p,j}^l(t)$ as the number of survivable solutions to the subproblem defined by operation O_j^l and its descendants that are compatible with the assignment of $st_p^l = t$ to O_p^l.
- For each $O_k^l \in \alpha_p^l$, we define $AFT_{i,k}^l(t)$ as the number of survivable solutions to the subproblem defined by operation O_k^l and its descendants that are compatible with the assignment of $st_p^l = t$ to O_p^l.

Given that operation O_i^l is the only link between the subproblems defined by each one of its children, we have:

$$compsurv_i^l(t) = \prod_{j \in \beta_i^l} BEF_{i,j}^l(t) \times \prod_{k \in \alpha_i^l} AFT_{i,k}^l(t).$$

Notice that this formula also relies on an independence assumption made in Section 6.4: the probability that a solution to the relaxation survives contention is assumed to be given by the product of the probabilities that each one of the reservation assignments in that solution survives contention.

$BEF_{i,j}^l(t)$ is obtained by adding all the subproblem solutions compatible with the precedence constraint $st_j^l + du_j^l \leqslant t$:

$$BEF_{i,j}^l(t) = \sum_{\tau \leqslant t - du_j^l} \left[surv_j^l(\tau) \times \prod_{p \in \beta_j^l} BEF_{j,p}^l(\tau) \times \prod_{q \in \alpha_j^l} AFT_{j,q}^l(\tau) \right].$$

Similarly for $AFT_{i,k}^l(t)$, we have:

$$AFT_{i,k}^l(t) = \sum_{\tau \geqslant t + du_i^l} \left[surv_k^l(\tau) \times \prod_{s \in \beta_k^l} BEF_{k,s}^l(\tau) \times \prod_{u \in \alpha_k^l} AFT_{k,u}^l(\tau) \right].$$

We can speed up the computation of this recurrence using partial sums:

$$BEF^l_{i,j}(t) = BEF^l_{i,j}(t - \Delta) + \left[surv^l_j(t - du^l_j) \times \prod_{p \in \beta^l_j} BEF^l_{j,p}(t - du^l_j) \times \prod_{q \in \alpha^l_j} AFT^l_{j,q}(t - du^l_j) \right],$$

$$AFT^l_{i,k}(t) = AFT^l_{i,k}(t + \Delta) + \left[surv^l_k(t + du^l_i) \times \prod_{s \in \beta^l_k} BEF^l_{k,s}(t + du^l_i) \times \prod_{u \in \alpha^l_k} AFT^l_{k,u}(t + du^l_i) \right].$$

The recurrence is initialized with:

$$BEF^l_{j,p}(est^l_j - \Delta) = 0,$$
$$AFT^l_{k,s}(lst^l_k + \Delta) = 0$$

and uses the convention:

$$\prod_{\emptyset} = 1.$$

In order to compute $compsurv^l_i(t)$ for all remaining start times of the critical operation O^l_i, the system starts by computing all $BEF^l_{j,p}(t)$ or all $AFT^l_{j,p}(t)$ at the leaf operations in the tree depicted in Fig. A.1. The procedure then moves up in the tree by combining at each level the $BEF^l_{j,p}(t)$ and $AFT^l_{j,p}(t)$ computed at the previous level. At each operation O^l_p in the tree, the procedure computes at most λ $BEF^l_{j,p}(t)$ expressions if O^l_p is before O^l_j, its parent operation, or λ $AFT^l_{j,p}(t)$ expressions, if O^l_p is after O^l_j (where λ is the maximum number of possible start times of an operation). Each such computation involves two multiplications and one addition. Hence, if ν_l is the number of operations in the relaxation used by the FSS value ordering heuristic, computing all $compsurv^l_i(t)$ can be done in $O(\nu_l \lambda)$ elementary computations. Computing $surv^l_p(t) = \sum_{\rho \in G} surv^l_p(\rho)$ for all the possible start times of all the operations in the relaxation requires however $O(\nu_l k)$ steps where k is the maximum number of reservations left to an operation.[14] Hence the overall complexity of the procedure is also $O(\nu_l k)$.

Appendix B. Value ordering filter

The following describes a filtering mechanism used to refine the ranking of reservations in our FSS value ordering heuristic.

[14] The real complexity is actually $O(\nu_l k du)$, where du is the duration of the longest operation in the relaxation. This duration is assumed to be bounded by a constant.

For some reservations ρ, $compsurv_i^l(\rho)$ can become very large and have too much influence in (6) compared to $surv_i^l(\rho)$. Consider the following two reservations ρ_1 and ρ_2:

- ρ_1: $compsurv_i^l(\rho_1) = 1000$ and $surv_i^l(\rho_1) = 0.5$.
- ρ_2: $compsurv_i^l(\rho_2) = 200$ and $surv_i^l(\rho_2) = 1.0$.

Ideally, a good value ordering heuristic should recognize that reservation ρ_2 is better than reservation ρ_1, despite the fact that, according to Eq. (6) $good_i^l(\rho_1) = 500$ is larger than $good_i^l(\rho_2) = 200$. Indeed, in this example, it does not really matter whether $compsurv_i^l(\rho)$ equals 200 or 1000: in either case there will certainly be enough compatible schedules. Instead, the factor that really matters here is the survivability of the reservation itself (i.e. locally). In the experiments reported at the end of this paper, this problem was handled by filtering the number of survivable solutions compatible with a reservation ρ, $compsurv_i^l(\rho)$. Instead of relying on Eq. (6), our value ordering heuristic measures reservation goodness according to the following revised formula:

$$good_i^l(\rho) = surv_i^l(\rho) \times MIN(\Phi^{\nu_l - 1}, compsurv_i^l(\rho)), \tag{B.1}$$

where *MIN* denotes the minimum function and Φ is a parameter of the system that is empirically adjusted. By using a filter of the form $\Phi^{\nu_l - 1}$, the heuristic attempts to ensure that, on the average, each one of the $\nu_l - 1$ other operations in the relaxation has Φ survivable reservations.[15]

References

[1] J.F. Allen, Maintaining knowledge about temporal intervals, *Commun. ACM* **26** (1983) 832–843.

[2] K.R. Baker, *Introduction to Sequencing and Scheduling* (Wiley, New York, 1974).

[3] J.R. Bitner and E.M. Reingold, Backtrack programming techniques, *Commun. ACM* **18** (1975) 651–655.

[4] C.-C. Cheng, Scheduling by precedence constraints posting, Ph.D. Thesis, Graduate School of Industrial Administration and the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (1995).

[5] E. Davis, Constraint propagation with interval labels, *Artif. Intell.* **32** (1987) 281–331.

[6] R. Dechter and I. Meiri, Experimental evaluation of preprocessing techniques in constraint satisfaction problems, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 271–277.

[7] R. Dechter and I. Meiri, Experimental evaluation of preprocessing algorithms for constraint satisfaction problems, *Artif. Intell.* **68** (1994) 211–241.

[8] R. Dechter, I. Meiri and J. Pearl, Temporal constraint networks, in: *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ont. (1989).

[9] R. Dechter and J. Pearl, Network-based heuristics for constraint satisfaction problems, *Artif. Intell.* **34** (1988) 1–38.

[10] M.S. Fox, *Constraint-Directed Search: A Case Study of Job-Shop Scheduling* (Morgan Kaufmann, Los Altos, CA, 1987).

[11] M.S. Fox, N. Sadeh and C. Baykan, Constrained heuristic search, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 309–315.

[15] A more sophisticated way of filtering $compsurv_i^l(\rho)$ would involve filtering the number of compatible reservations of each operation in the relaxation. This would ensure that each one of the operations in the relaxation has enough compatible reservations. In general, because the critical operation is also the one in the relaxation whose reservations are the least survivable, a single filter for all the other operations in the relaxation seems sufficient.

[12] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop* (Wiley, New York, 1982).
[13] E.C. Freuder, A sufficient condition for backtrack-free search, *J. ACM* **29** (1982) 24–32.
[14] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, San Francisco, CA, 1979).
[15] M.L. Ginsberg, M. Frank, M.P. Halpin and M.C. Torrance, Search lessons learned from crossword puzzle, in: *Proceedings of the Eighth National Conference on Artificial Intelligence* (1990) 210–215.
[16] S.W. Golomb and L.D. Baumert, Backtrack programming, *J. ACM* **12** (1965) 516–524.
[17] R.M. Haralick and G.L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artif. Intell.* **14** (1980) 263–313.
[18] L.A. Johnson and D.C. Montgomery, *Operations Research in Production Planning, Scheduling, and Inventory Control* (Wiley, New York, 1974).
[19] M.D. Johnston and S. Minton, Analyzing a heuristic strategy for constraint satisfaction and scheduling, in: M. Fox and M. Zweben, eds., *Intelligent Scheduling* (Morgan Kaufmann, Los Altos, CA, 1994) 257–289, Chapter 9.
[20] N. Keng and D.Y.Y. Yun, A planning/scheduling methodology for the constrained resource problem, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 998–1003.
[21] C. Le Pape and S.F. Smith, Management of temporal constraints for factory scheduling, Tech. Rept., The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (1987); also in: *Proceedings Working Conference on Temporal Aspects in Information Systems*, Paris (North-Holland, Amsterdam, 1987).
[22] J. Liu and K. Sycara, Collective problem solving through coordination in a society of reactive agents, Tech. Rept. CMU-RI-TR-94-23, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (1994).
[23] A.K. Mackworth and E.C. Freuder, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artif. Intell.* **25** (1985) 65–74.
[24] J.J. McGregor, Relational consistency algorithms and their applications in finding subgraph and graph isomorphisms, *Inform. Sci.* **19** (1979) 229–250.
[25] S. Minton, M.D. Johnston, A.B. Philips and P. Laird, Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems, *Artif. Intell.* **58** (1992) 161–205.
[26] T.E. Morton, and D.W. Pentico, *Heuristic Scheduling Systems*, Wiley Series in Engineering and Technology Management (Wiley, New York, 1993).
[27] N. Muscettola, HSTS: integrating planning and scheduling, in: M. Fox and M. Zweben, eds., *Intelligent Scheduling* (Morgan Kaufmann, Los Altos, CA, 1994) 169–212, Chapter 6.
[28] N. Muscettola and S. Smith, A probabilistic framework for resource-constrained multi-agent planning, in: *Proceedings AAAI-87*, Seattle, WA (1987) 1063–1066.
[29] B. Nadel, Tree search and arc consistency in constraint satisfaction algorithms, in: L. Kanal and V. Kumar, eds., *Search in Articial Intelligence* (Springer, Berlin, 1988).
[30] B.A. Nadel, Theory-based search-order selection for constraint satisfaction problems, Tech. Rept. DCS-TR-183, Department of Computer Science, Laboratory for Computer Research, Rutgers University, New Brunswick, NJ (1986).
[31] D. Navinchandra, *Exploration and Innovation in Design* (Springer, Berlin, 1990).
[32] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization* (Wiley, New York, 1988).
[33] W.P.M. Nuijten, Time and resource constrained scheduling, Ph.D. Thesis, Technische Universiteit Eindhoven, Eindhoven (1994).
[34] P.S. Ow, Focused scheduling in proportionate flowshops, *Manage. Sci.* **31** (1985) 852–869.
[35] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (Addison-Wesley, Reading, MA, 1984).
[36] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* (Morgan Kaufmann, Los Altos, CA, 1988).
[37] P.W. Purdom Jr, Search rearrangement backtracking and polynomial average time, *Artif. Intell.* **21** (1983) 117–133.
[38] N. Sadeh, Look-ahead techniques for activity-based job-shop scheduling, Thesis Proposal (1989).
[39] N. Sadeh, Look-ahead techniques for micro-opportunistic job shop scheduling, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1991).

[40] N. Sadeh, Micro-opportunistic scheduling: the MICRO-BOSS factory scheduler, in: M.S. Fox and M. Zweben, eds., *Intelligent Scheduling* (Morgan Kaufmann, Los Altos, CA, 1994) 99–135, Chapter 4.

[41] N. Sadeh and M.S. Fox, Preference propagation in temporal/capacity constraint graphs, Tech. Rept. CMU-CS-88-193, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA (1988); also: Robotics Institute Tech. Rept. CMU-RI-TR-89-2.

[42] N. Sadeh and M.S. Fox, Focus of attention in an activity-based scheduler, in: *Proceedings NASA Conference on Space Telerobotics* (1989).

[43] N. Sadeh and M.S. Fox, CORTES: an exploration into micro-opportunistic job-shop scheduling, in: *Proceedings IJCAI-89*, Detroit, MI (1989).

[44] N. Sadeh, and M.S. Fox, Variable and value ordering heuristics for activity-based job-shop scheduling, in: *Proceedings Fourth International Conference on Expert Systems in Production and Operations Management*, Hilton Head Island, SC (1990) 134–144.

[45] N. Sadeh, K. Sycara and Y. Xiong, Backtracking techniques for hard scheduling problems, Tech. Rept. CMU-RI-TR-92-06, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (1992); also: *Artif. Intell.* **76** (1995) 455–480 (improved version).

[46] N. Sadeh, K. Sycara and Y. Xiong, Backtracking techniques for the job shop scheduling constraint satisfaction problem, *Artif. Intell.* **76** (1995) 455–480; also: CMU Tech. Rept. CMU-RI-TR-94-31; an earlier version of this paper also appeared as CMU Tech. Rept. CMU-RI-TR-92-06.

[47] S.F. Smith and C. Cheng, Slack-based heuristics for constraint satisfaction scheduling, in: *Proceedings AAAI-93*, Washington, DC (1993).

[48] R.E. Tarjan, Minimum spanning trees, in: *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics **44** (SIAM, Philadelphia, PA, 1983) Chapter 6.

[49] P. Van Hentenryck, H. Simonis and M. Dincbas, Constraint satisfaction using constraint logic programming, *Artif. Intell.* **58** (1992) 113–159.

[50] R.J. Walker, An enumerative technique for a class of combinatorial problems, in: R. Bellman and M. Hall, eds., *Combinatorial Analysis, Proceedings Symposium on Applied Mathematics* (American Mathematical Society, Providence, RI, 1960) 91–94, Chapter 7.

[51] R. Zabih and D. McAllester, A rearrangement search strategy for determining propositional satisfiability, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 155–160.

[52] M. Zweben, B. Daun, E. Davis and M. Deale, Scheduling and rescheduling with iterative repair, in: M.S. Fox and M. Zweben, eds., *Intelligent Scheduling* (Morgan Kaufmann, Los Altos, CA, 1994) 241–255, Chapter 8.