ELSEVIER

Artificial Intelligence 76 (1995) 455–480

Artificial Intelligence

Backtracking techniques for the job shop scheduling constraint satisfaction problem*

Norman Sadeh*, Katia Sycara, Yalin Xiong

5000 Forbes Avenue, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA

Received June 1993; revised May 1994

Abstract

This paper studies a version of the job shop scheduling problem in which some operations have to be scheduled within non-relaxable time windows (i.e. earliest/latest possible start time windows). This problem is a well-known NP-complete Constraint Satisfaction Problem (CSP). A popular method for solving this type of problems involves using depth-first backtrack search. In our earlier work, we focused on the development of consistency enforcing techniques and variable/value ordering heuristics that improve the efficiency of this search procedure. In this paper, we combine these techniques with new look-back schemes that help the search procedure recover from so-called deadend search states (i.e. partial solutions that cannot be completed without violating some constraints). More specifically, we successively describe three "intelligent" backtracking schemes: (1) *Dynamic Consistency Enforcement* dynamically identifies critical subproblems and determines how far to backtrack by selectively enforcing higher levels of consistency among variables participating in these critical subproblems, (2) *Learning Ordering From Failure* dynamically modifies the order in which variables are instantiated based on earlier conflicts, and (3) *Incomplete Backjumping Heuristic* abandons areas of the search space that appear to require excessive computational efforts. These schemes are shown to (1) further reduce the average complexity of the backtrack search procedure, (2) enable our system to efficiently solve problems that could not be solved otherwise due to excessive computation cost, and (3) be more effective at solving job shop scheduling problems than other look-back schemes advocated in the literature.

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

* Corresponding author.

SSDI 0004-3702(95)00078-6

1. Introduction

This paper is concerned with the design of recovery schemes for incremental scheduling approaches that sometimes require undoing earlier scheduling decisions in order to complete the construction of a feasible schedule.

Job shop scheduling deals with the allocation of resources over time to perform a collection of tasks. The job shop scheduling model studied in this paper further allows for operations that have to be scheduled within non-relaxable time windows (e.g. earliest possible start time/latest possible finish time windows). This problem is a well-known NP-complete Constraint Satisfaction Problem (CSP) [10]. Instances of this problem 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 revising the schedule of other operations, etc.

One approach to solving CSPs is to use depth-first backtrack search [2, 12, 26]. Using this approach, scheduling problems can be solved through the iterative selection of an operation to be scheduled next (i.e. variable selection) and the tentative assignment of a reservation (i.e. value) to that operation. If in the process of constructing a schedule, a partial solution is reached that cannot be completed without violating some of the problem constraints, one or several earlier assignments need to be undone. This process of undoing earlier assignments is referred to as *backtracking*. It deteriorates the efficiency of the search procedure and increases the time required to come up with a solution. While the worst-case complexity of backtrack search is exponential, several techniques to reduce its average-case complexity have been proposed in the literature [6]:

- *Consistency enforcing schemes:* These techniques prune the search space from alternatives that cannot participate in a global solution [16]. There is generally a tradeoff between the amount of consistency enforced in each search state[1] and the savings achieved in search time.
- *Variable/value ordering heuristics:* These heuristics help judiciously decide which variable to instantiate next and which value to assign to that variable [2, 6, 8, 13, 17, 18]. By first instantiating difficult variables, the system increases its chances of completing the current partial solution without backtracking [8, 13, 18]. Good value ordering heuristics reduce backtracking by selecting values that are expected to participate in a large number of solutions [6, 18].
- *Look-back schemes* [4, 7, 11, 24]*:* While it is possible to design consistency enforcing schemes and variable/value ordering heuristics that are, on average, very effective at reducing backtracking, it is generally impossible to

[1] A search state is associated with each partial solution. Each search state defines a new CSP whose variables are the variables that have not yet been instantiated and whose constraints are the initial problem constraints along with constraints reflecting current assignments.

efficiently guarantee backtrack-free search. Look-back schemes are designed to help the system recover from deadend states and, if possible, learn from past mistakes.

In our earlier work, we focused on the development of efficient consistency enforcing techniques and variable/value ordering heuristics for job shop scheduling CSPs [8, 18, 20–23, 25]. In this paper, we combine these techniques with new look-back schemes. These schemes are shown to further reduce the average complexity of the search procedure. They also enable our system to efficiently solve problems that could not be efficiently solved otherwise. Finally, experimental results indicate that these techniques are more effective at solving job shop scheduling problems than other look-back schemes advocated in the literature.

The simplest deadend recovery strategy goes back to the most recently instantiated variable with at least one alternative value left, and assigns one of the remaining values to the variable. This strategy is known as *chronological backtracking*. Often the source of the current deadend is not the most recent assignment but an earlier one. Because it typically modifies assignments that have no impact on the conflict at hand, chronological backtracking often returns to similar deadend states. When this happens, search is said to be *thrashing*. Thrashing can be reduced using backjumping schemes that attempt to backtrack all the way to one of the variables at the source of the conflict [11]. Search efficiency can be further improved by learning from past mistakes. For instance, a system can record earlier conflicts in the form of new constraints that will prevent it from repeating earlier mistakes [7, 24]. Dependency-directed backtracking is a technique incorporating both backjumping and constraint recording [24]. Although dependency-directed backtracking can greatly reduce the number of search states that need to be explored, this scheme is often impractical due to the exponential worst-case complexity of its constraint recording component (both in time and space). Simpler techniques have also been developed that approximate dependency-directed backtracking. *Graph-based backjumping* reduces the amount of book-keeping required by full-blown backjumping by assuming that any two variables directly connected by a constraint may have been assigned conflicting values [4].[2] *Nth-order deep and shallow learning* reduce the constraint recording complexity of dependency-directed backtracking by only recording conflicts involving N or fewer variables [4].

Graph-based backjumping works best on CSPs with sparse constraint graphs [4]. Instead, job shop scheduling problems have highly interconnected constraint graphs. Furthermore graph-based backjumping does not increase search efficiency when used in combination with forward checking [13] mechanisms or stronger consistency enforcing mechanisms such as those entailed by job shop scheduling problems [18]. Our experiments suggest that Nth-order deep and shallow learning techniques often fail to improve search efficiency when applied to job shop scheduling problems. This is because these techniques use constraint size as the

[2] Two variables are said to be "*connected*" by a constraint if they both participate in that constraint.

only criterion to decide whether or not record earlier failures. When they limit themselves to small-size conflicts, they fail to record some important constraints. When they do not, their complexities become prohibitive.

Instead, this paper presents three look-back techniques that have yielded good results on job shop scheduling problems:

(1) *Dynamic Consistency Enforcement* (DCE): a selective dependency-directed scheme that dynamically focuses its effort on critical resource subproblems;
(2) *Learning Ordering From Failure* (LOFF): an adaptive scheme that suggests new variable orderings based on earlier conflicts;
(3) *Incomplete Backjumping Heuristic* (IBH): a scheme that gives up searching areas of the search space that require too much work.

Related work in scheduling includes that of Prosser and Burke who use Nth-order shallow learning to solve one-machine scheduling problems [3], and that of Badie et al. whose system implements a variation of deep learning in which a minimum set is heuristically selected as the source of the conflict [1].

The remainder of this paper is organized as follows. Section 2 provides a more formal definition of the job shop CSP. Section 3 describes the backtrack search procedure considered in this study. Sections 4, 5 and 6 successively describe each of the three backtracking schemes developed in this study. Experimental results are presented in Section 7. Section 8 summarizes the contributions of this paper.

2. The job shop constraint satisfaction problem

The job shop scheduling problem requires scheduling a set of jobs $J = \{j_1, \ldots, j_n\}$ on a set of resources $RES = \{R_1, \ldots, R_m\}$. Each job j_l consists of a set of operations $O^l = \{O^l_1, \ldots, O^l_{q_l}\}$ to be scheduled according to a process routing that specifies a partial ordering among these operations (e.g. O^l_i BEFORE O^l_j).

In the job shop CSP studied in this paper, each job j_l has a release date rd_l and a due date dd_l between which all its operations have to be performed. Each operation O^l_i has a fixed duration du^l_i and a variable start time st^l_i. 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. one or several shifts in factory scheduling). In order to be successfully executed, each operation O^l_i requires p^l_i different resources (e.g. a milling machine and a machinist) R^l_{ij} $(1 \leq j \leq p^l_i)$, for each of which there may be a pool of physical resources from which to choose, $\Omega^l_{ij} \subseteq RES$ (e.g. one or several milling machines).

More formally, the problem can be defined as follows:

Variables

A vector of variables is associated with each operation, O_i^l $(1 \leq l \leq n,\ 1 \leq i \leq q_l)$, which consists of:

(1) the *start time*, st_i^l, of the operation, and
(2) its *resource requirements*, R_{ij}^l $(1 \leq j \leq p_i^l)$.

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: $st_i^l + du_i^l \leq st_j^l$ (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: $(\forall p\ \forall q\ R_{ip}^k \neq R_{jq}^l) \vee st_i^k + du_i^k \leq st_j^l \vee st_j^l + du_j^l \leq st_i^k$. These constraints simply express that, unless they use different resources, two operations O_i^k and O_j^l cannot overlap.[3]

Additionally, our model can accommodate unary constraints that restrict the set of possible values of individual variables. These constraints include non-relaxable due dates and release dates, between which all operations in a job need to be performed. More generally, the model can accommodate any type of unary constraint that further restricts the set of possible start times of an operation.

Time is assumed discrete, i.e. operation start times and end times can only take integer values and each resource requirement R_{ij}^l has to be selected from a set of resource alternatives, $\Omega_{ij}^l \subseteq RES$.

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. 1 depicts a simple job shop scheduling problem with four jobs $J = \{j_1, j_2, j_3, j_4\}$ and four 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. It is further assumed that all jobs are released at time 0 and have to be completed by time 20. Please note that none of these simplifying assumptions is required by the techniques to be discussed: jobs can have different release and due dates, operations can have several resource requirements, and several alternatives for each of these requirements. Note also that the problem we have just defined is infeasible. None of the operations on resource R_2 can start before time 3 and the sum of durations of these operations is 18. Hence, it is impossible to complete

[3] These constraints have to be generalized when dealing with resources of capacity larger than one.

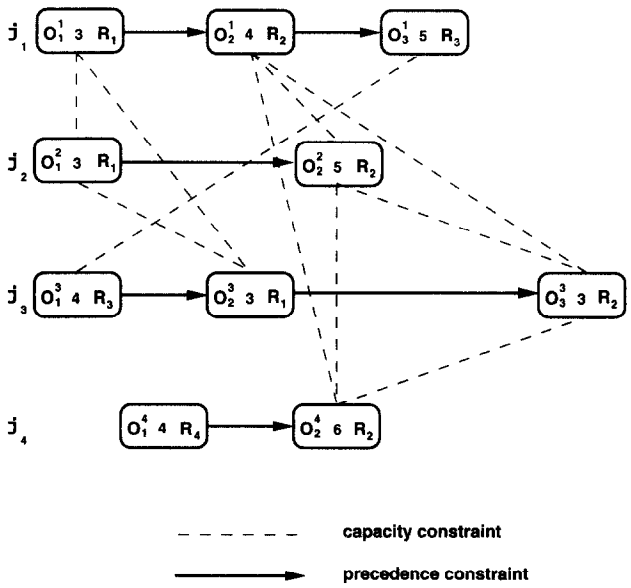

Fig. 1. A simple problem with four jobs. Each node is labeled by the operation that it represents, its duration and the resource it requires.

these operations before time 21. As we will see, this observation can easily be operationalized in the form of a simple consistency checking rule. However, as the number of operations to schedule grows, the exponential complexity of applying this simple rule to all possible subsets of operations on a given resource quickly becomes prohibitive, hence the need to be more selective in applying such checks. Additionally, passing such a check is no guarantee that a problem is feasible, hence the need to also rely on more complex mechanisms, as described below.

3. The search procedure

A depth-first backtrack search procedure is considered, in which search is interleaved with the application of consistency enforcing mechanisms and variable/value ordering heuristics that attempt to steer clear of deadend states, as described in Fig. 2.

Specifically, search starts in a state where all operations still have to be scheduled. The BASIC-DEPTH-FIRST procedure proceeds by incrementally scheduling operations one by one. Each time an operation is scheduled, a new search state is created in which a consistency enforcing procedure (or constraint propagation procedure) is first applied to update the set of possible reservations of unscheduled operations. Next, an operation is selected to be scheduled and a reservation is selected for that operation. The procedure goes on, recursively

```
Procedure BASIC-DEPTH-FIRST ()
 If UNSCHED-OP = ∅ Then solution found, STOP.
 Let CONFLICT = FALSE
 Call CONSTRAINT-PROPAGATION
 If CONFLICT = FALSE
    Then Begin
            Let OP = OPER-SELECTION (UNSCHED-OP)
            Let REMAINING-RESERV(OP) = list of possible reservations for OP
                                       ordered according to the value ordering
                                       heuristic (best reservations first)
            Let RESERV = Pop first reservation in REMAINING-RESERV(OP)
            Push (OP,RESERV) onto SCHEDULE
            Remove OP from UNSCHED-OP
            Call BASIC-DEPTH-FIRST ()
          End
    Else Call SIMPLE-BACKTRACK ()
 End If
End Procedure

Procedure SIMPLE-BACKTRACK ()
 If SCHEDULE = ∅ Then Let NO-SOLUTION = TRUE and STOP.
 Remove the last (operation, reservation) pair pushed onto SCHEDULE
 and Let OP be the operation in that pair
 Insert OP in  UNSCHED-OP
 If  REMAINING-RESERV(OP) ≠ ∅
     Then Begin
               RESERV = Pop first reservation in REMAINING-RESERV(OP)
               Push (OP,RESERV) onto SCHEDULE
               Remove OP from UNSCHED-OP
               Call BASIC-DEPTH-FIRST ()
     End
     Else Call SIMPLE-BACKTRACK ()
 End If
End Procedure

Begin Program
 Let SCHEDULE = ∅
 Let UNSCHED-OP = {O_1^1, ... O_{q_1}^1, O_1^2, ... O_{q_2}^2 .., O_1^n, ... O_{q_n}^n}
 Let NO-SOLUTION = FALSE
 Call BASIC-DEPTH-FIRST()
 If NO-SOLUTION = FALSE
    Then PRINT-SOLUTION
    Else PRINT "Infeasible Problem"
 End If
End Program
```

Fig. 2. Basic depth-first backtrack search procedure.

calling itself, until either all operations are successfully scheduled or an inconsistency (or conflict) is detected. In the latter case, the procedure needs to undo earlier decisions or backtrack. The backtracking mechanism in Fig. 2, SIMPLE-BACKTRACK, is a chronological backtracking procedure 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. If the procedure returns to the initial search state (i.e. the state with an empty schedule), the problem is infeasible.

The default consistency enforcing mechanisms and variable/value ordering heuristics used in our study are the ones described in [18]. These mechanisms, which have been favorably compared against a number of other heuristics [18, 23], are briefly described below.

Consistency enforcing procedure

The consistency enforcing procedure we use combines three consistency mechanisms:

(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 [14], earliest start time constraints are propagated downstream within the job whereas latest start time constraints are propagated upstream (Fig. 3). The complexity of this simple propagation mechanism is linear in the number of precedence constraints. In the absence of capacity constraints, the procedure can be shown to guarantee decomposability [5], i.e. it is sufficient to guarantee backtrack-free search [18].

(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 [13] 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 [15] (see Fig. 4).

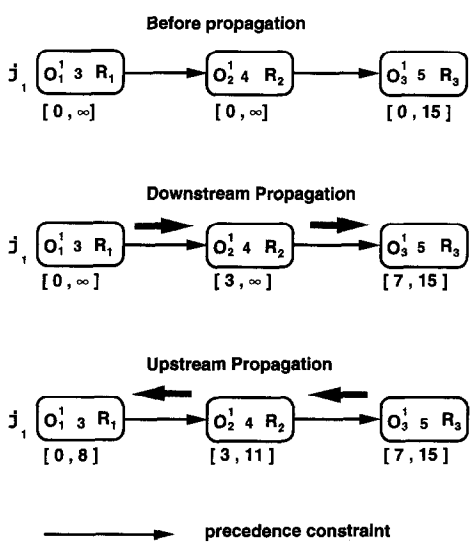

Fig. 3. Consistency with respect to precedence constraints.

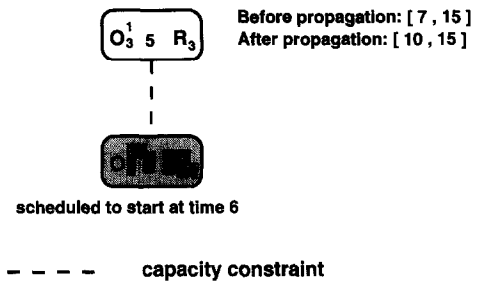

Fig. 4. Forward consistency checks with respect to capacity constraints.

(3) *Additional consistency checks with respect to capacity constraints:* Additionally, our default consistency enforcing mechanism checks that no two unscheduled operations require overlapping resource/time intervals. An example of such a situation is illustrated in Fig. 5, where two operations requiring the same resource, O_i^k and O_j^l, rely on the availability of overlapping time intervals, namely the intervals between their respective latest start times and earliest finish times ($[lst_i^k, eft_i^k]$ and $[lst_j^l, eft_j^l]$). This additional consistency mechanism has been shown to often increase search efficiency, while only resulting in minor computational overheads [18].

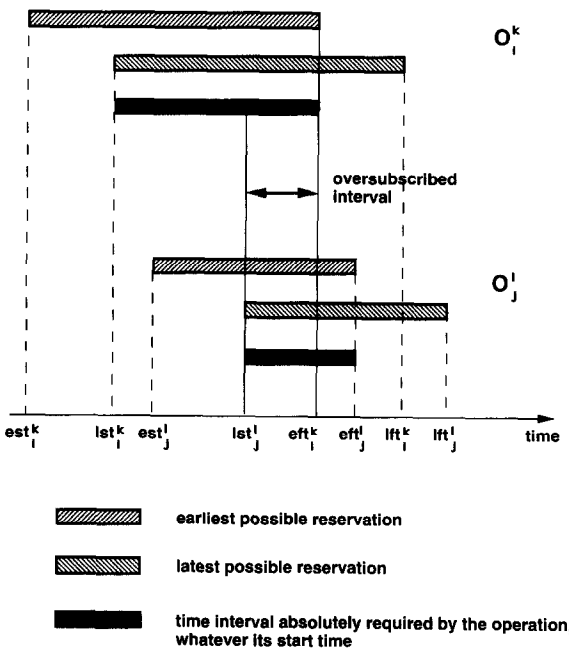

Fig. 5. Detecting situations where two unscheduled operations requiring the same resource are in conflict.

Variable/value ordering heuristics

The default variable/value ordering heuristics used by our search procedure are the *Operation Resource Reliance* (ORR) variable ordering heuristic and *Filtered Survivable Schedules* (FSS) value ordering heuristic described in [18]. The ORR variable ordering heuristic aims at reducing backtracking by first scheduling difficult operations, namely operations whose resource requirements are expected to conflict with those of other operations. The FSS value ordering heuristic is a least constraining value ordering heuristic. It attempts to further reduce backtracking by selecting reservations that are expected to be compatible with a large number of schedules.

These default heuristics have been reported to outperform several other schemes described in the literature, both generic CSP heuristics and specialized heuristics designed for similar scheduling problems [18, 23]. They seem to provide a good compromise between the efforts spent enforcing consistency, ordering variables, or ranking assignments for a variable and the actual savings obtained in search time. Nevertheless, the job shop CSP is NP-complete and, hence, these efficient procedures are not sufficient to guarantee backtrack-free search.

The remainder of this paper describes new backtracking schemes that help the system recover from deadend states. We show that, when the default consistency enforcing mechanisms and/or variable ordering heuristics are not sufficient to steer clear of deadends, look-back mechanisms can be devised that modify these schemes so as to avoid repeating past mistakes (i.e. so as to avoid reaching similar deadend states).

4. Dynamic Consistency Enforcement (DCE)

Backtracking is generally an indication that the default consistency enforcing scheme and/or variable/value ordering heuristics used by the search procedure are insufficient to deal with the subproblems at hand. Consequently, if search keeps on relying on the same default mechanisms after reaching a deadend state, it is likely to start thrashing. Experiments reported in [18, 23], in which search always used the same set of consistency enforcing procedures and variable/value ordering heuristics, clearly illustrated this phenomenon. Search in these experiments exhibited a dual behavior. The vast majority of the problems fell in either of two categories: a category of problems that were solved with no backtracking whatsoever (by far the largest category) and a category of problems that caused the search procedure to thrash.

Theoretically, thrashing could be eliminated by enforcing full consistency in each search state. Clearly, such an approach is impractical as it would amount to performing a complete search. Instead, our approach involves (1) heuristically identifying one or a few small subproblems that are likely to be at the source of the conflict, (2) determining how far to backtrack by enforcing full consistency among the variables in these small subproblems, and (3) recording conflict information for possible reuse in future backtracking episodes. This approach is

operationalized in the context of a backtracking scheme called *Dynamic Consistency Enforcement* (DCE). Given a deadend state and a history of earlier backtracking episodes within the same search space (i.e. while working on the same problem), this technique dynamically identifies small critical resource subproblems expected to be at the source of the current deadend. DCE then backtracks, undoing assignments in a chronological order, until a search state is reached, within which consistency has been fully restored in each critical resource subproblem (i.e. consistency with respect to capacity constraints in these subproblems). Experimental results reported in Section 7 suggest that often, by *selectively* checking for consistency in small resource subproblems, DCE can quickly recover from deadends. The remainder of this section further describes the mechanics of this heuristic.

4.1. Identifying critical resource subproblems

The critical resource subproblems used by DCE consist of groups of operations participating in the current conflict along with groups of critical operations identified during earlier backtracking episodes involving the same resources. Below, we refer to the group of (unscheduled) operations identified by the default consistency enforcing mechanism as having no possible reservations left as the *Partial Conflicting Set* of operations (*PCS*) (see Fig. 6). In order to restore consistency, the search procedure needs to at least go back to a search state in which each *PCS* operation has one or more possible reservations.[4] DCE attempts to identify such additional operations by maintaining a group of critical resource subproblems identified during earlier backtracking episodes. Below, we refer to this data structure as the *Former Dangerous Groups* of operations (*FDG*). Details on how this data structure is created and maintained are provided in Section 4.3.

For each capacity constraint violation among operations in *PCS*, DCE checks the *FDG* data structure and retrieves all related resource subproblems. A resource subproblem in *FDG* is considered related to a capacity constraint violation in *PCS* if, in an earlier backtracking episode, operations in that resource subproblem were involved in a capacity constraint violation on the same resource and over a "close" time interval. A system parameter is used to determine if two resource conflicts are "close". In the experiments reported at the end of this paper, two conflicts were considered close if the distance separating them was not greater than twice the average operation duration. Related critical subproblems identified by inspecting the *FDG* data structure are then merged with corresponding operations in *PCS* to form a new set of one or more critical resource subproblems, which we refer to as the *Dangerous Group* of operations (*DG*) for the conflict at hand. Like *FDG*, *DG* is organized in subgroups of resource subproblems consisting of operations contending for the same resource over close or overlapping time intervals. While backtracking, operations that are un-

[4] Clearly, this is not guaranteed to be sufficient, as other operations may also contribute to the conflict.

```
Procedure DEPTH-FIRST-WITH-DCE ()
 If UNSCHED-OP = ∅ Then solution found, STOP.
 Let PCS = ∅
 Call CONSTRAINT-PROPAGATION, which places all operations with
      no remaining reservations in PCS
 If PCS = ∅
    Then Begin
           Let OP = OPER-SELECTION (UNSCHED-OP)
           Let REMAINING-RESERV(OP) = list of possible reservations for OP
                                      ordered according to the value ordering
                                      heuristic (best reservations first)
           Let RESERV = Pop first reservation in REMAINING-RESERV(OP)
           Push (OP,RESERV) onto SCHEDULE
           Remove OP from UNSCHED-OP
           Call DEPTH-FIRST-WITH-DCE ()
         End
    Else Begin
           Let DG be the set of consolidated resource subproblems obtained
               by merging resource subproblems in PCS with related resource
               subproblems in FDG
           Call DCE-BACKTRACK (DG)
         End
 End If
End Procedure

Procedure DCE-BACKTRACK (DG)
 If SCHEDULE = ∅ Then Let NO-SOLUTION = TRUE and STOP.
 Remove the last (operation, reservation) pair pushed onto SCHEDULE
 and Let OP be the operation in that pair
 Insert OP in  UNSCHED-OP and in DG
 Let CONFLICT = FALSE
 For each resource subproblem in DG, prune the set of remaining reservations
     of each operation in that subproblem by enforcing full consistency
     with respect to capacity constraints. In the process, if an operation
     is found to have no possible reservations left then stop enforcing
     consistency and Let CONFLICT = TRUE
 If  CONFLICT = FALSE
     Then Begin
               RESERV = Pop first reservation in REMAINING-RESERV(OP)
               Push (OP,RESERV) onto SCHEDULE
               Remove OP from UNSCHED-OP
               UPDATE-FDG(DG)
               Call DEPTH-FIRST-WITH-DCE ()
     End
     Else Call DCE-BACKTRACK (DG)
 End If
End Procedure

Begin Program
 Let SCHEDULE = ∅     Let FDG = ∅      Let NO-SOLUTION = FALSE
 Let UNSCHED-OP = { O_1^1, ... O_{q_1}^1, O_1^2, ... O_{q_2}^2 .., O_1^n, ... O_{q_n}^n }
 Call DEPTH-FIRST-WITH-DCE()
 If NO-SOLUTION = FALSE
    Then PRINT-SOLUTION
    Else PRINT "Infeasible Problem"
 End If
End Program
```

Fig. 6. DCE procedure.

scheduled are inserted in *DG*, either by being added to existing resource subproblems or by creating new resource subproblems.

4.2. Backtracking while selectively enforcing consistency

Once the initial *DG* has been identified, DCE backtracks, undoing assignments in a chronological order, until it reaches a search state in which consistency is restored within each of the resource subproblems defined by operations in *DG* (see Fig. 6). This is done by enforcing full consistency with respect to capacity constraints in each of the resource subproblems in *DG*. As long as conflicts are detected, the procedure continues to backtrack and unscheduled operations are inserted into existing or new resource subproblems in *DG*. While restoring consistency within each of these resource subproblems is a necessary condition to backtrack to a consistent search state, it is not always a sufficient one. In other words, the effectiveness of DCE critically depends on its ability to heuristically focus on the right resource subproblems.[5]

Because full consistency checking can be expensive on large subproblems, if a resource subproblem in *DG* becomes too large, *k*-consistency is enforced instead of full consistency, where *k* is a parameter of the system [9]. In the experiments reported at the end of this paper, *k* was set to 4. At the end of the backtracking episode, *DG* has maximum size, call it DG_{max}. Assuming that the procedure was able to backtrack to a consistent search state, DG_{max} is expected to contain all the operations at the origin of the deadend[6] and often more. DG_{max} is then saved for later use in the *FDG* data structure. Additional details regarding the management of this data structure are provided in the next subsection. If a related backtracking episode is later encountered by the system, DG_{max} can be retrieved and combined with the *PCS* of this new episode.

4.3. Storing information about past backtracking episodes

The purpose of the *Former Dangerous Groups* of operations (*FDG*) maintained by the system is to help determine more efficiently and more precisely the scope of each deadend by focusing on critical resource subproblems. Each group of operations in *FDG* consists of operations that are in high contention for the allocation of a same resource. Accordingly, whenever, a conflict is detected that involves some of the operations in one group, the backtracking procedure checks for consistency among *all* operations in that group.

The groups of operations in *FDG* are build from the *Dangerous Groups* (*DG*s) obtained at the end of previous backtracking episodes (DG_{max}). Indeed, whenever a backtracking episode is completed, DG_{max} is expected to contain all the

[5] Note that DCE is not expected to be very effective at recovering from complex conflicts involving interactions between multiple resource subproblems. A heuristic which is often more effective for these complex conflicts is described in Section 6.

[6] Clearly, while this is not guaranteed, experimental results suggest that this is often the case.

conflicting operations at the origin of this episode. Generally, DG_{max} may involve one or several resource subproblems (i.e. groups of operations requiring the same resource). Each one of these subproblems is merged with *related* subproblems currently stored in *FDG*. If there is no related group in *FDG*, the new group is separately added to the data structure. Finally, as operations are scheduled, they are removed from *FDG*.

4.4. An example

Fig. 7 illustrates the behavior of DCE on the small scheduling problem introduced in Fig. 1. After scheduling operations O_2^4 and O_2^2 on resource R_2, the procedure detects that operation O_3^3 has no possible reservations left. Given that the *FDG* data structure is initially empty (no prior backtracking episode), we have $PCS = DG = \{O_3^3\}$. The procedure unschedules the most recently scheduled operation, namely O_2^2, and inserts it in *DG* together with operation O_3^3, as both of these operations require the same resource. At this point, DCE enforces full consistency with respect to capacity constraints between these two operations[7] and finds that, after consistency checking, the operations still admit some possible reservations. This marks the end of the first backtracking episode. The procedure saves the current *DG* in *FDG*, for possible reuse, then schedules operation O_2^2 at its next best available start time,[8] namely start time 6. In the process, O_2^2 is removed from *FDG*. Another conflict is detected in this new search state, which marks the beginning of a second backtracking episode. This time the consistency enforcing procedure finds that operation O_1^2 has no possible reservations left (i.e. $PCS = \{O_2^1\}$). Using *FDG*, the system adds operation O_3^3 to the group of dangerous operations, $DG = \{O_2^1, O_3^3\}$. Accordingly, this time, when it unschedules operation O_2^2, DCE enforces full consistency[9] with respect to capacity constraints in $DG = \{O_2^1, O_2^2, O_3^3\}$. When it finds that the current search state is still inconsistent, DCE proceeds and unschedules operation O_2^4, thereby returning to the root search state with $DG = \{O_2^1, O_2^2, O_3^3, O_2^4\}$. In this search state, full consistency with respect to capacity constraints between operations in *DG* indicates that the problem is infeasible. In total, the system only generates three search states to find that the problem is infeasible. In contrast, a total of 50 search states is required for the same small problem, when relying on the SIMPLE-BACKTRACK procedure outlined in Fig. 2. The example also shows how the use of the Formerly Dangerous Groups (*FDG*) of operations helps the system identify critical resource subproblems. If it was not for this mechanism, the procedure would not detect an inconsistency when it comes back to depth 1 in the second backtracking episode, as it would only check for consistency between O_2^1 and O_2^2.

---

[7] This is equivalent to 2-consistency or arc-consistency, given that there are only two operations [9].

[8] Actually, start time 6 is not the start time picked by our reservation ordering heuristic. The system was manually forced to pick this value to make the example more interesting.

[9] This time the system enforces 3-consistency, given that there are three operations in *DG*.

```
>> Depth: 0, Number of states visited: 0, FDG=∅
   O_2^4 is scheduled between 14 and 20 on R_2

>> Depth: 1, Number of states visited: 1, FDG=∅
   O_2^2 is scheduled between 9 and 14 on R_2

>> Depth: 2, Number of states visited: 2, FDG=∅
   Conflict detected: O_3^3 has no possible reservations left:
   PCS=DG={[O_3^3]}   [Beginning of first backtracking episode]
   O_2^2 is unscheduled

>> Depth: 1, Number of states visited: 2, FDG=∅
   DG={[O_3^3, O_2^2]}
   Full consistency checking with respect to capacity constraints in DG:
        Remaining possible start times:
           O_2^2: {3,4,5,6}
           O_3^3: {8,9,10,11}
   FDG={[O_2^2, O_3^3]} [End of first backtracking episode]
   O_2^2 is scheduled between 6 and 11 on R_2

>> Depth: 2, Number of states visited: 3, FDG={[O_3^3]}
   Conflict detected: O_2^1 has no possible reservations left:
   PCS={[O_2^1]}, DG={[O_2^1, O_3^3]}   [Beginning of second backtracking episode]
   O_2^2 is unscheduled

>> Depth: 1, Number of states visited: 3, FDG={[O_3^3]}
   DG={[O_2^1, O_2^2, O_3^3]}
   Full consistency checking with respect to capacity constraints in DG:
   Conflict detected
   O_2^4 is unscheduled

>> Depth: 0, Number of states visited: 3, FDG={[O_3^3]}
      DG={[O_2^1, O_2^2, O_3^3, O_2^4]}
   Full consistency checking with respect to capacity constraints in DG:
   Conflict detected
   Infeasible Problem   [End of second backtracking episode]
```

Fig. 7. An edited trace illustrating the DCE procedure.

More generally, experimental results presented in Section 7 show that DCE often results in important increases in search efficiency and important reductions in computation time.

4.5. *Additional 'watch dog' consistency checks*

Because groups of operations in *FDG* are likely deadend candidates, our system further performs simple "watch dog" checks on these dynamic groups of operations.

More specifically, for each group G of operations in FDG, the system performs a rough check to see if the resource can still accommodate all the operations in the group. This is done using redundant constraints of the form:

$$\text{Max}(lst_i^l + du_i^l, O_i^l \in G) - \text{Min}(est_i^l, O_i^l \in G) \geq \sum_{O_i^l \in G} du_i^l ,$$

where est_i^l and lst_i^l are respectively the earliest and latest possible start times of O_i^l in the current search state.

Whenever such a constraint is violated, an inconsistency has been detected. Though very simple and inexpensive, these checks enable to catch inconsistencies involving large groups of operations that would not be immediately detected by the default consistency mechanisms. Clearly, some inconsistencies can still escape these rough checks.

While backtracking, the same "watch dog" checks can be used prior to enforcing full consistency with respect to capacity constraints in the critical resource subproblems in DG. This can significantly reduce computation time. For instance, in the second backtracking episode in Fig. 7, these simple checks are sufficient to detect inconsistencies at depth 1 and 0. For example, at depth 1, where $DG = \{[O_2^1, O_2^2, O_3^3]\}$,

$$\text{Max}(lst_i^l + du_i^l, O_i^l \in DG) - \text{Min}(est_i^l, O_i^l \in DG) = 14 - 3 = 11 ,$$

while

$$\sum_{O_i^l \in DG} du_i^l = 12 .$$

5. Learning Ordering From Failures (LOFF)

Often, reaching a deadend state is also an indication that the default variable ordering was not adequate for dealing with the subproblem at hand. Typically, the operations participating in the deadend turn out to be more difficult to schedule than the ones selected by the default variable ordering heuristic. In other words, it is often a good idea to first schedule the operations participating in the conflict that was just resolved. *Learning Ordering From Failure* (LOFF) is an adaptive procedure that overrides the default variable ordering in the presence of conflicts.

After recovering from a deadend, namely after backtracking all the way to an apparently consistent search state, LOFF uses the Partial Conflicting Set (PCS) of the deadend to reorganize the order in which operations will be rescheduled and make sure that operations in the PCS are scheduled first. This is done using a quasi-stack, QS, on which operations in PCS are pushed in descending order of domain size, i.e. PCS operations with a large number of remaining reservations are pushed first on the quasi-stack. When the quasi-stack is empty, the procedure uses its default variable ordering heuristic, as described in Section 3. However,

when QS contains some operations, the procedure first schedules these operations, starting with the ones on top of the quasi-stack, namely those QS operations with the smallest number of remaining reservations.

If a candidate operation is already in QS, i.e. if it is encountered for a second time, it is pushed again on QS as if it had a smaller domain. This orders operations based on the recency of the conflict in which they were last involved and based on their number of remaining reservations.

6. An Incomplete Backjumping Heuristic

Traditional backtrack search procedures only undo decisions that have been proven to be inconsistent. Proving that an assignment is inconsistent with others can be very expensive, especially when dealing with large conflicts. Graph-based backjumping and Nth-order shallow/deep learning attempt to reduce the complexity of full-blown dependency-directed backtracking by either simplifying the process of identifying inconsistent decisions (e.g. based on the topology of the constraint graph) or restricting the size of the conflicts that can be detected. The Dynamic Consistency Enforcement (DCE) procedure described in Section 6 also aims at reducing the complexity of identifying the source of a conflict by dynamically focusing its effort on small critical subproblems. Because these techniques focus on smaller conflicts, they all have problems dealing with more complex conflicts involving a large number of variables.[10] It might in fact turn out that the only effective way to deal with more complex conflicts is by using heuristics that undo decisions not because they have been proven inconsistent but simply because they appear overly restrictive. This is the approach taken in the heuristic described in this section. Clearly, the resulting search procedure is no longer complete and may fail to find solutions to feasible problems, hence the name of *Incomplete Backjumping Heuristic* (IBH).

Texture measures such as the ones described in [8] could be used to estimate the tightness of different search states, for instance, by estimating the number of global solutions compatible with each search state. Clearly, a search state whose partial solution is compatible with a large number of global solutions is loosely constrained, whereas one compatible with a small number of global solutions is tightly constrained. Assignments leading to much tighter search states would be prime candidates to be undone when a complex conflict is suspected. The *Incomplete Backjumping Heuristic* (IBH) used in this study is simpler and, yet, often seems to be sufficient. Whenever the system starts thrashing, this heuristic backjumps all the way to the first search state and simply tries and next best value (i.e. reservation) for the critical operation in that state (i.e. the first operation selected be the variable ordering heuristic). IBH considers that the search

[10] Clearly, there are some conflicts involving large numbers of variables that are easy to catch, as illustrated by the watch dog checks described in Section 4.

procedure is thrashing, and hence that it is facing a complex conflict, when more than θ assignments had to be undone since the last time the system was thrashing or since the procedure began, if no thrashing occurred earlier; θ is a parameter of the procedure.

7. Empirical evaluation

This section reports the results of empirical studies conducted to assess the performance of the look-back schemes presented in this paper. The first study reports performance on a suite of 60 benchmark problems introduced in [18]. This is followed by a more detailed study comparing the performance of the first two look-back schemes introduced in this paper (DCE & LOFF) against that of second-order deep learning [4] and chronological backtracking. Finally, we compare the performance of the complete search procedure relying on DCE & LOFF with that of an incomplete procedure combining all three of the look-back schemes presented in this paper (DCE & LOFF & IBH).

7.1. Performance evaluation on a first suite of problems

A first set of experiments was run on a testsuite of 60 job shop scheduling problems first introduced in [18]. In the experiments reported in [18], the default variable and value ordering heuristics used in our study (i.e. the ORR and FSS heuristics described in Section 3) were shown to outperform a variety of other variable/value ordering combinations, though they still failed to solve 8 out of the 60 problems. In contrast, the results presented below indicate that the combination of our three look-back techniques (DCE & LOFF & IBH) can efficiently solve all 60 problems in the testsuite.

Specifically, the testsuite consists of six groups of ten problems each. Each problem requires scheduling ten jobs on five resources and involves a total of 50 operations (five operations per job). Each job has a linear process routing specifying a sequence in which it has to visit each one of the five resources. This sequence varies from one job to another, except for a predetermined number of bottleneck resources (one or two in these experiments) which are always visited after the same number of steps. The six groups of problems were obtained by varying two parameters:

(1) the number of a priori bottlenecks ($BTNK$): one ($BTNK = 1$) or two ($BTNK = 2$), and
(2) the spread (SP) of the release and due dates between which each job has to be scheduled: wide ($SP = \mathrm{W}$), narrow ($SP = \mathrm{N}$), or null ($SP = 0$).

The SP parameter and the operation durations have been adjusted so that bottleneck utilization remains close to 100% over most of the span of each problem. In these problems, each operation had slightly over 100 possible start

times (i.e. values) after application of the consistency enforcing techniques in the initial search state. Additional details on these problems can be found in [18].[11]

Table 1 compares the performance of the following two procedures:

(1) the basic depth-first procedure described in Fig. 2, namely a procedure relying on chronological backtracking and on the default consistency enforcing techniques and variable/value ordering heuristics described in Section 3 (this is also the procedure reported to perform best in [18]);

Table 1
Comparison of chronological backtracking and DCE & LOFF & IBH on six sets of ten job shop problems

		Chronological	DCE & LOFF & IBH
$SP = W$	Search efficiency	0.96	0.96
$BTNK = 1$	# experiments solved (out of 10)	10	10
	CPU seconds	88.5	90.5
$SP = W$	Search efficiency	0.99	0.99
$BTNK = 2$	# experiments solved (out of 10)	10	10
	CPU seconds	93	95
$SP = N$	Search efficiency	0.78	0.91
$BTNK = 1$	# experiments solved (out of 10)	8	10
	CPU seconds	331.5	106
$SP = N$	Search efficiency	0.87	0.93
$BTNK = 2$	# experiments solved (out of 10)	9	10
	CPU seconds	184	119.5
$SP = 0$	Search efficiency	0.73	0.88
$BTNK = 1$	# experiments solved (out of 10)	7	10
	CPU seconds	475	134.5
$SP = 0$	Search efficiency	0.82	0.84
$BTNK = 2$	# experiments solved (out of 10)	8	10
	CPU seconds	300.5	226.5
Overall performance	Search efficiency	0.86	0.92
	# experiments solved (out of 60)	52	60
	CPU seconds	245.5	128.7

[11] The problems are also accessible via anonymous ftp to cimds3.cimds.ri.cmu.edu, where they can be found in /usr/sadeh/public/csp_test_suite. A README file details the content of the various files in the directory.

(2) the same procedure enhanced with the DCE, LOFF and IBH look-back schemes presented in this paper.

For each of the 60 problems, search was stopped if it required more than 500 search states. Performance in each problem category is reported along three dimensions:

(1) *Search efficiency:* the average 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.
(2) *Number of experiments* solved in less than 500 search states.
(3) *CPU seconds:* this is the average CPU time required to solve a problem. When a solution could not be found, this time was approximated as the CPU time taken to explore 500 search states (this approximation was only used for chronological backtracking, since DCE & LOFF & IBH solved all problems). All CPU times were obtained on a DECstation 5000 running Knowledge Craft on top of Allegro Common Lisp. Experimentation with a variation of the system written in C indicates that the search procedure would run about 30 times faster if reimplemented in this language [19].

The results indicate that DCE & LOFF & IBH consistently outperformed the chronological backtracking scheme in terms of CPU time, search efficiency and number of problems solved. On the easier problems ($SP = \mathrm{W}$), both techniques solved all 20 problems in approximately the same amount of time. On the more difficult problems ($SP = \mathrm{N}$ and $SP = 0$), DCE & LOFF & IBH clearly dominated chronological backtracking. In particular, on problems with $SP = 0$ and $BTNK = 1$, DCE & LOFF & IBH solved 40% more problems than the chronological backtracking scheme and, on average, proved to be 3.5 times faster. Overall, while chronological backtracking failed to solve 8 problems out of 60, DCE & LOFF & IBH efficiently solved all 60 problems, and, on average, was almost twice as fast as the procedure with chronological backtracking. Had we not stopped the chronological backtracking procedure after 500 search states, the speedup achieved by DCE & LOFF & IBH would be even more significant. In fact, based on a couple of problems for which the chronological procedure was allowed to expand a larger number of search states, it appears that problems that are not solved in 500 states often require thousands more to be solved (with chronological backtracking).

7.2. Further evaluation

To further evaluate our look-back schemes, we picked the most difficult problem category in the testsuite, namely the category for which the default consistency enforcing procedure and variable/value ordering heuristics are least effective ($SP = 0$) and generated an additional 80 scheduling problems, 40 with $BTNK = 1$ and 40 with $BTNK = 2$. The $SP = 0$ problem category was also the most difficult one for all the other combinations of variable and value ordering heuristics tested in the study reported in [18]. It corresponds to problems in which

all jobs are released at a common date and need to be completed by a common due date. Among the resulting 80 problems, we only report performance on those problems for which the default schemes were not sufficient to guarantee backtrack-free search.[12] This leaves 16 scheduling problems with one bottleneck ($SP = 0$ and $BTNK = 1$), and 15 with two bottlenecks ($SP = 0$ and $BTNK = 2$).

Below, we successively report the results of two studies. The first one compares the performance of three complete backtracking schemes: chronological backtracking, second-order deep learning, and the procedure combining the DCE and LOFF backtracking heuristics.[13] The second study compares the complete search procedure using DCE and LOFF with the incomplete search procedure combining DCE, LOFF and IBH.

The results of the first study comparing chronological backtracking, second-order deep learning [4] and the DCE & LOFF procedures advocated in Sections 4 and 5 are summarized in Tables 2 and 3. The results reported here were obtained using a search limit of 500 nodes and a time limit of 1800 seconds (except for deep learning, for which the time limit was increased to 36,000 seconds[14]). All CPU

Table 2
Results of one-bottleneck experiments. (S: solved; F: failure; S*: proved infeasible; time limit: 1800 sec (except deep learning); node limit: 500)

Exp. No.	Chronological			DCE & LOFF			Deep learning		
	No. of nodes	CPU (sec)	Result	No. of nodes	CPU (sec)	Result	No. of Nodes	CPU (sec)	Result
1	500	1427	F	122	1232	S*	500	5756	F
2	500	1587	F	500	1272	F	500	5834	F
3	74	148	S	63	117	S	25	36000	F
4	69	152	S	52	120	S	69	391	S
5	500	1407	F	65	134	S	500	11762	F
6	500	1469	F	500	1486	F	500	8789	F
7	500	1555	F	59	130	S	500	9681	F
8	500	1705	F	41	145	S*	500	9560	F
9	53	108	S	53	102	S	53	122	S
10	500	1529	F	500	1536	F	500	9114	F
11	500	1460	F	85	1800	F	500	14611	F
12	500	1694	F	500	1131	F	500	21283	F
13	51	109	S	51	81	S	51	88	S
14	500	1762	F	63	138	S	500	18934	F
15	500	1798	F	69	142	S	500	9600	F
16	500	1584	F	500	1183	F	65	36000	F

[12] Clearly, performance on problems that do not require backtracking is of no interest, since our backtracking schemes never get invoked, and hence CPU time remains unchanged.

[13] Besides the experiments reported below, additional experiments were performed to assess the benefits of using DCE and LOFF separately. These experiments show that both techniques contribute to the improvements reported in this section.

[14] This was motivated by the fact that our implementation of deep learning may not be optimal.

times reported below were obtained on a DECstation 5000 running Knowledge Craft on top of Allegro Common Lisp. As already indicated above, comparison between C and Knowledge Craft implementations of similar variable and value ordering heuristics indicates that the code would run about 30 times faster in C [19].

On the one-bottleneck problems, chronological backtracking solved only 4 problems out of 16 (see Table 2). Interestingly enough, deep learning showed no improvement over chronological backtracking either in the number of problems solved or in CPU time. As a matter of fact, deep learning was even too slow to find solutions to some of the problems solved by chronological backtracking. This is attributed to the fact that the constraints in job shop scheduling are more tightly interacting than those in the zebra problem, where the improvement of deep learning over chronological backtracking was originally ascertained [4]. On the other hand, DCE & LOFF solved 10 problems out of 16 (2 out of these 10 problems were successfully proven infeasible). As expected, by focusing on a small number of critical subproblems, DCE & LOFF is able to discover larger more useful conflicts than second-order deep learning, while requiring only a fraction of the time. Another observation is that DCE & LOFF expanded fewer search states than chronological backtracking for the problems that chronological backtracking solved. However, each of the DCE & LOFF expansions took slightly more CPU time, due to the higher level of consistency enforcement.

Results for the set of two-bottleneck problems are reported in Table 3. Similar results are observed here again: deep learning shows no improvement over chronological backtracking and seems significantly slower. The difference be-

Table 3
Results of two-bottleneck experiments. (S: solved; F: failure; S*: proved infeasible; time limit: 1800 sec (36,000 sec for deep learning); node limit: 500)

Exp. No.	Chronological			DCE & LOFF			Deep learning		
	No. of nodes	CPU (sec)	Result	No. of nodes	CPU (sec)	Result	No. of Nodes	CPU (sec)	Result
1	500	1139	F	113	1800	F	18	36000	F
2	500	1444	F	425	1800	F	115	36000	F
3	84	175	S	109	202	S	84	811	S
4	56	123	S	56	112	S	56	213	S
5	51	101	S	51	113	S	13	36000	F
6	500	1531	F	321	1800	F	328	36000	F
7	500	1775	F	500	1357	F	500	2793	F
8	52	102	S	52	115	S	33	36000	F
9	500	1634	F	247	974	S	500	1519	F
10	500	1676	F	91	1800	F	26	36000	F
11	66	163	S	59	104	S	66	2240	S
12	56	139	S	58	104	S	58	281	S
13	54	129	S	52	91	S	54	28900	S
14	500	1676	F	346	1800	F	500	9031	F
15	500	1522	F	324	1800	F	296	36000	F

tween chronological backtracking and DCE & LOFF is not as impressive as in the first set of experiments. As can be seen in Table 3, chronological backtracking solved 7 out of 15 problems, whereas DCE & LOFF solved 8. On the problems solved by both chronological backtracking and DCE & LOFF, DCE & LOFF turned out to be slightly faster overall. These less impressive results suggest that the presence of multiple bottlenecks often introduces more complex conflicts. Results presented in the following subsection suggest that in this case incomplete backtracking procedures such as the one entailed by the IBH heuristic are often much more effective.

7.3. *Complete versus incomplete search procedures*

Tables 4 and 5 compare the performance of the complete search procedure based on DCE & LOFF against that of an incomplete search procedure using DCE & LOFF in combination with the IBH heuristic described in Section 6. While DCE & LOFF could only solve 10 out of 16 one-bottleneck problems and 8 out 15 two-bottleneck problems, DCE & LOFF combined with IBH solved 14 one-bottleneck problems and 13 two-bottleneck problems. The only one-bottleneck problems that were not solved by DCE & LOFF & IBH are the two problems identified as infeasible by the complete procedure with DCE & LOFF (see Table 2). This is hardly a surprise. While the addition of IBH to DCE & LOFF enables the search procedure to solve a larger number of problems, it also makes the procedure incomplete (i.e. infeasible problems can no longer be

Table 4
Results of one-bottleneck experiments (S: solved; F: failure; S*; proved infeasible; time limit: 1800 sec; node limit: 500)

Exp. No.	DCE & LOFF			DCE & LOFF & IBH		
	No. of nodes	CPU (sec)	Result	No. of nodes	CPU (sec)	Result
1	122	1232	S*	350	1800	F
2	500	1272	F	203	1124	S
3	63	117	S	63	123	S
4	52	120	S	52	116	S
5	65	134	S	65	144	S
6	500	1486	F	127	424	S
7	59	130	S	59	125	S
8	41	145	S*	457	1800	F
9	53	108	S	53	100	S
10	500	1536	F	67	170	S
11	85	1800	F	74	170	S
12	500	1131	F	164	616	S
13	51	81	S	51	92	S
14	63	138	S	63	149	S
15	69	142	S	69	158	S
16	500	1183	F	156	524	S

Table 5
Results of two-bottleneck experiments (S: solved; F: failure; S*; proved infeasible; time limit: 1800 sec; node limit: 500)

Exp. No.	DCE & LOFF			DCE & LOFF & IBH		
	No. of nodes	CPU (sec)	Result	No. of nodes	CPU (sec)	Result
1	113	1800	F	151	456	S
2	425	1800	F	371	1780	S
3	109	202	S	95	210	S
4	56	112	S	56	108	S
5	51	113	S	51	97	S
6	321	1800	F	420	1800	F
7	500	1357	F	159	534	S
8	52	115	S	52	96	S
9	247	974	S	423	1705	S
10	91	1800	F	440	1800	F
11	59	104	S	59	113	S
12	58	104	S	58	112	S
13	52	91	S	52	102	S
14	346	1800	F	239	512	S
15	324	1800	F	73	195	S

identified). Additional experiments combining IBH with a simple chronological backtracking scheme produced results that were not as good as those obtained by DCE & LOFF & IBH, indicating that both IBH and DCE & LOFF contribute to the performance improvement observed in Tables 4 and 5.

Results on two-bottleneck problems (see Table 5) also suggest that the impact of IBH is particularly effective on these problems. This is attributed to the fact that two-bottleneck problems give rise to more complex conflicts. Identifying the assignments participating in these more complex conflicts might simply be too difficult for any exact backtracking scheme. Instead, because it can undo assignments that are not provably wrong but simply appear overly restrictive, IBH seems more effective at solving these problems.

8. Concluding remarks

We have presented three look-back techniques for the job shop scheduling CSP:

(1) *Dynamic Consistency Enforcement* (DCE), a heuristic that dynamically focuses on restoring consistency within small critical subproblems,
(2) *Learning Ordering From Failure* (LOFF), a technique that modifies the order in which variables are instantiated based on earlier conflicts, and
(3) *Incomplete Backjumping Heuristic* (IBH) which, when thrashing occurs, can undo assignments that are not provably inconsistent but appear overly restrictive.

The significance of this research is twofold:

(1) Job shop scheduling problems with non-relaxable time windows have multiple applications (e.g. manufacturing, space, transportation, health care, etc.). We have shown that our look-back heuristics combined with powerful techniques that we had previously developed (i) further reduce the average complexity of backtrack search, and (ii) enable this search procedure to efficiently solve problems that could not be solved otherwise due to excessive computational requirements. While the results reported in this study were obtained on problems that require finding a feasible schedule, the backtracking schemes presented in this paper can also be used on optimization versions of the scheduling problem, such as the Just-In-Time job shop scheduling problems described in [19].

(2) This research also points to shortcomings of dependency-directed backtracking schemes advocated earlier in the literature. In particular, comparison with second-order deep learning indicates that this technique failed to improve performance on our set of job shop scheduling problems. More generally, *N*th-order deep and shallow learning techniques often appear inadequate when applied to job shop scheduling problems because they rely solely on constraint size to decide whether or not to record earlier failures. When these techniques limit themselves to small-size conflicts, they often fail to record some important constraints; when they consider larger conflicts, their computational complexity becomes prohibitive. A more general weakness of traditional backtracking schemes has to do with the fact that they never undo assignments unless they can be proven to be at the source of the conflict. When dealing with large complex conflicts, proving that a particular assignment should be undone can be very expensive. Instead, our experiments suggest that, when thrashing cannot easily be avoided, it is often a better idea to use incomplete backjumping heuristics that undo decisions simply because they *appear* overly restrictive.

References

[1] C. Badie, G. Bel, E. Bensana and G. Verfaillie, Operations research and artificial intelligence cooperation to solve scheduling problems, in: *Proceedings First International Conference on Expert Planning Systems* (1990).

[2] J.R. Bitner and E.M. Reingold, Backtrack programming techniques, *Commun. ACM* **18** (11) (1975) 651–655.

[3] P. Burke and P. Prosser, A distributed asynchronous system for predictive and reactive scheduling, Technical Report AISL-42, Department of Computer Science, University of Strathclyde, Glasgow, Scotland (1989).

[4] R. Dechter, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, *Artif. Intell.* **41** (1989) 273–312.

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

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

[7] J. Doyle, A truth maintenance system, *Artif. Intell.* **12** (3) (1979) 231–272.
[8] M.S. Fox, N. Sadeh and C. Baykan, Constrained heuristic search, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 309–315.
[9] E.C. Freuder, A sufficient condition for backtrack-free search, *J. ACM* **29** (1) (1982) 24–32.
[10] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, San Francisco, CA, 1979).
[11] J. Gaschnig, Performance measurement and analysis of certain search algorithms, Technical Report CMU-CS-79-124, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA (1979).
[12] S.W. Golomb and L.D. Baumert, Backtrack programming, *J. ACM* **12** (4) (1965) 516–524.
[13] R.M. Haralick and G. L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artif. Intell.* **14** (3) (1980) 263–313.
[14] L.A. Johnson and D.C. Montgomery, *Operations Research in Production Planning, Scheduling, and Inventory Control* (Wiley, New York, 1974).
[15] C. Le Pape and S.F. Smith, Management of temporal constraints for factory scheduling, Technical Report, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (1987); in: *Proceedings Working Conference on Temporal Aspects in Information Systems*, Paris, France (1987).
[16] A.K. Mackworth and E.C. Freuder, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artif. Intell.* **25** (1) (1985) 65–74.
[17] P.W. Purdom Jr, Search rearrangement backtracking and polynomial average time, *Artif. Intell.* **21** (1983) 117–133.
[18] N. Sadeh, Look-ahead techniques for micro-opportunistic job shop scheduling. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1991).
[19] N. Sadeh, Micro-opportunistic scheduling: the MICRO-BOSS factory scheduler, in: M.S. Fox and M. Zweben, eds., *Intelligent Scheduling* (Morgan Kaufmann, San Mateo, CA, 1994) Chapter 4.
[20] N. Sadeh and M.S. Fox, Preference propagation in temporal/capacity constraint graphs, Technical Report CMU-CS-88-193, Comuter Science Department, Carnegie Mellon University, Pittsburgh, PA (1988); also: Robotics Institute Technical Report CMU-RI-TR-89-2.
[21] N. Sadeh and M.S. Fox, Focus of attention in an activity-based scheduler, in: *Proceedings NASA Conference on Space Telerobotics* (1989).
[22] 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.
[23] N. Sadeh and M.S. Fox, Variable and value ordering heuristics for hard constraint satisfaction problems: an application to job shop scheduling, Technical Report CMU-RI-TR-91-23, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (1992).
[24] R. Stallman and G. Sussman, Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artif. Intell.* **9** (1977) 135–196.
[25] K. Sycara, S. Roth, N. Sadeh and M.S. Fox, Distributed constrained heuristic search, *IEEE Trans. Syst. Man Cybern.* **21** (6) (1991).
[26] 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* (AMS, Providence, RI, 1960) 91–94, Chapter 7.