Intelligent Backtracking Techniques for Job Shop Scheduling

Yalin Xiong **Norman Sadeh** **Katia Sycara**

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

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 these types of problems consists in using depth-first backtrack search. Our earlier work focused on developing efficient consistency enforcing techniques and efficient variable/value ordering heuristics to improve the efficiency of this 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: *Dynamic Consistency Enforcement* dynamically enforces higher levels of consistency in selected critical subproblems, *Learning From Failure* dynamically modifies the order in which variables are instantiated based on earlier conflicts, and *Heuristic Backjumping* gives up searching areas of the search space that appear too difficult. These schemes are shown to (1) further reduce the average complexity of the search procedure, (2) enable our system to efficiently solve problems that could not be solved otherwise due to excessive computational cost, and (3) be more effective at solving job shop scheduling problems than other look-back schemes advocated in the literature.

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 (i.e. earliest possible start time/latest possible finish time windows). This problem is a well-known NP-complete Constraint Satisfaction Problem (CSP) [11]. Examples of such problems 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.

A generic approach to solving CSPs relies on depth-first backtrack search [24, 13, 2]. Using this paradigm, scheduling problems are solved through the iterative selection of a variable (i.e. an operation) and the tentative assignment of a value (i.e. a reservation) to that variable. If in the process of constructing a solution, a partial solution is reached that cannot be completed without violating some of the problem constraints, one or several earlier assignments have 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 have been proposed in the literature to reduce its average-case complexity [7]:

- *Consistency Enforcing Schemes*:[15] prune the search space from alternatives that cannot participate in a global solution . There is generally a tradeoff between the amount of consistency enforced in each search state[1] and the savings achieved in search time.

- *Look-ahead Schemes*: variable/value ordering heuristics[2, 14, 16, 7, 9, 20] help judiciously decide which variable to instantiate next and which value to assign to that variable. By first instantiating difficult variables, the system increases its chances of completing the current partial solution without backtracking [14, 9, 20]. Good value ordering heuristics reduce backtracking by selecting values that are expected to participate in a large number of solutions [7, 20].
- *Look-back Schemes*:[22, 8, 12, 5] While it is possible to design consistency enforcing schemes and look-ahead schemes that are, on the average, very good at efficiently reducing backtracking, it is generally impossible to 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 .

Our earlier work focused on developing efficient consistency enforcing techniques and efficient look-ahead techniques for job shop scheduling CSPs [17, 18, 9, 23, 21, 20, 19]. 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 the schemes described in this paper are more effective at solving job shop scheduling problems than other look-back schemes advocated in the literature.

The simplest deadend recovery strategy consists in going back to the most recently instantiated variable with at least one alternative value left, and assigning a different value to that variable. This strategy is known as *chronological backtracking*. Often the source of the current deadend is not the most recent assignment but one performed earlier. By changing assignments that are irrelevant to the current conflict, 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 [12]. 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 [22, 8]. Dependency-directed backtracking is a technique incorporating both backjumping and constraint recording [22]. Although dependency-directed backtracking can greatly reduce the number of search states that need to be explored, this scheme is often impractical due to its exponential worst-case complexity (both in time and space). For this reason, simpler techniques have been developed that approximate dependency-directed backtracking. *Graph-based backjumping* reduces the amount of book-keeping required by fullblown backjumping by assuming that any two variables directly connected by a constraint may have been assigned conflicting values [5]. *N-th order deep and shallow learning* only record conflicts involving N or fewer variables. [4].

Graph-based backjumping works best on CSPs with sparse constraint graphs [5]. 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 [14] mechanisms or stronger consistency enforcing mechanisms such as those entailed by job shop scheduling problems [20]. Experiments reported at the end of this paper also suggest that N-th 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 only criterion to decide whether or not to 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 schemes which have yielded very 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 From Failure* (LFF): an adaptive scheme that suggests new variable orderings based on earlier conflicts,
3. *Heuristic Backjumping* (HB) 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 N-th 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 culprit [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

[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.

schemes developed for this study. Experimental results are presented in section 7. Section 8 summarizes the contributions of this paper.

2 Job Shop Constraint Satisfaction Problem and Search Procedure

The job shop scheduling problem requires scheduling a set of jobs $J = \{j_1, ..., j_n\}$ on a set of physical resources $RES = \{R_1, ..., R_m\}$. Each job j_l consists of a set of operations $O^l = \{O^l_1, ..., O^l_{n_l}\}$ 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, $\Omega^l_{ij} = \{r^l_{ij1}, ..., r^l_{ijq^l_{ij}}\}$, with $r^l_{ijk} \in RES (1 \leq k \leq q^l_{ij})$ (e.g. several possible milling machines).

More formally, the problem can be defined as follows:

VARIABLES:

A vector of variables is associated with each operation, $O^l_i (1 \leq l \leq n, 1 \leq i \leq n_l)$, which includes:

1. the *operation start time*, st^l_i, and
2. each *resource requirement*, $R^l_{ij} (1 \leq j \leq p^l_i)$ for which the operation has several alternatives.

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: $st^l_i + du^l_i \leq st^l_j$ (i.e. O^l_i BEFORE O^l_j);
2. *Capacity constraints* that restrict the use of each resource to only one operation at a time translate into disjunctive constraints of the form: $(\forall p \forall q R^k_{ip} \neq R^l_{jq}) \vee st^k_i + du^k_i \leq st^l_j \vee st^l_j + du^l_j \leq st^k_i$. These constraints simply express that, unless they use different resources, two operations O^k_i and O^l_j cannot overlap [2].

Additionally, there are unary constraints restricting 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. The model can actually 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. Finally, each resource requirement R^l_{ij} has to be selected from a set of resource alternatives, $\Omega^l_{ij} \subseteq RES$.

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.

3 The Search Procedure

A depth-first backtrack search procedure is assumed, 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. Search proceeds according to the following steps:

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.

The default consistency enforcing scheme and variable/value ordering heuristics used in the procedure are the ones described in [20]:

Consistency Enforcement: The consistency enforcing procedure is a hybrid procedure that differentiates between precedence constraints and capacity constraints. It guarantees that backtracking only occurs

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

as the result of capacity constraint violations. Essentially, consistency with respect to precedence constraints is enforced by updating in each search state a pair of earliest/latest possible start times for each unscheduled operation. Consistency enforcement with respect to capacity constraints tends to be significantly more expensive due to the disjunctive nature of these constraints. For capacity constraints, a forward checking type of consistency checking is generally carried out by the system. Whenever a resource is allocated to an operation over some time interval, the forward checking procedure checks the set of remaining possible start times of other operations requiring that resource, and removes those start times that would conflict with the new assignment. The system further checks for consistency with respect to a set of redundant capacity constraints, which can be quickly enforced in each search state. This includes checking that no two unscheduled operations totally rely on the same resource over overlapping time intervals [3].

Variable/Value Ordering Heuristics: The default variable/value ordering heuristics used by the search procedure are the *Operation Resource Reliance* (ORR) variable ordering heuristic and *Filtered Survivable Schedules* value ordering heuristic described in [20]. The ORR variable ordering heuristic aims at reducing backtracking by first scheduling difficult operations, namely operations whose resource requirements are expected to conflict with the resource requirements of other operations. The FSS value ordering heuristic is a least constraining value ordering heuristic. It attempts to further reduce backtracking by assigning reservations that are expected to be compatible with a large number of schedules.

These default consistency enforcing schemes and variable/value ordering 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 [20, 19]. These are efficient schemes that 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, because the job shop CSP is an NP-complete problem, these 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. It will be seen that, when the default consistency enforcing scheme and/or variable ordering scheme are not sufficient to stay clear of deadends, look-back mechanisms can be devised that will modify these schemes so as to avoid repeating past mistakes (i.e.so as to avoid reaching similar deadend states).

[3] See [20] for further details.

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. For this reason, when it reaches a deadend, the system will generally start thrashing if it keeps on using the same default mechanisms[4]. Theoretically, thrashing could be eliminated by enforcing full consistency in each search state. Clearly such an approach is prohibitively expensive. Instead, if one could heuristically identify small subproblems that are likely to be at the source of the conflict and just check for consistency among the variables in these subproblems, thrashing could often be eliminated at a lower computational cost. This is the approach described in this section. A backtracking scheme called *Dynamic Consistency Enforcement* (DCE) is presented that dynamically identifies small critical resource subproblems expected to be at the source of the current deadend. Experimental results reported in Section 7 suggest that, by *selectively* checking for consistency with respect to capacity constraints among the operations in these small subproblems, this scheme is often able to quickly recover from deadends.

When a deadend is detected, DCE checks for consistency with respect to capacity constraints in critical resource subproblems, in order to approximate the full extent of the current deadend and decide how far to backtrack. The critical subproblems used by DCE consist of groups of operations participating in the current conflict along with groups of critical operations identified at an earlier stage. Below, we refer to the group(s) of operations participating in the current conflict, as the *Partial Conflicting Set* of operations (PCS): these are the operations identified by the default consistency enforcing mechanism as having no possible reservations left in the current search state. The objective of the backtracking scheme is to identify the most recent assignment(s), which, if undone, will produce a consistent search state, i.e. a search state in which operations in PCS have reservations that do not seem to conflict with earlier assignments. To this end, DCE checks for consistency with respect to capacity constraints between operations in PCS and critical operations in a so-called *Dangerous Group* (DG) of operations identified earlier. At each level (while backtracking), the set consisting of the union of the PCS, the DG and the set of undone operations up to that level is referred to as the *Deadend Operation Set* (DOS). While backrtacking, DCE performs full consistency checking with re-

[4] Experiments reported in [20, 19] consistency displayed a dual behavior: the vast majority of the scheduling problems were either solved *without* backtracking whatsoever, or required an *exponential amount of chronological backtracking*.

spect to capacity constraints among operations in the DOS. Generally, because the DOS may contain operations requiring different resources, the backtracking scheme checks for consistency with respect to capacity constraints in several resource subproblems [5]. During backtracking the PCS and the DG remain the same and the DOS varies as more undone operations are unioned. At the end of a backtracking episode, DOS has maximum size, call it DOS_{max}. Assuming that the procedure was able to backtrack to a consistent search state [6], DOS_{max} contains all the operations at the origin of the deadend (and often more). DOS_{max} is then saved for later use in a data structure referred to as the *Former Dangerous Groups* (FDG). Details regarding the management of this data structure are provided in subsection 4.1. If a related backtracking episode is later encountered by the system, DOS_{max} can then be retrieved and serve as the DG for this new episode [7]. If a subsequent backtracking episode is unrelated to any of the previous ones, then the DG for this episode is empty.

The behavior of the DCE procedure is illustrated in Figure 1. Each node represents a search state, labeled by the operation that was last scheduled to reach that state, the resource allocated to that operation, and the operation's start time. In this example, search is assumed to have reached a deadend at depth D_5. Operations in the PCS are those operations whose domains of possible start times were identified as empty at depth D_5 due to capacity constraint violations. The resources associated with operations in the PCS are called the *critical resources.* Although a PCS can in general contain operations associated with more than one critical resources, it is often the case that the operations in PCS require the same resource (i.e., the deadend happened as a result of capacity constraint violations on a single resource). Upon encountering a deadend at D_5, DCE backtracks to D_4 and performs full consistency checking with respect to capacity constraints on the set of operations $DOS_4 = PCS \cup DG \cup O_m$ If there are still capacity constraint violations at D_4, operation O_l is undone, and full consistency checking is performed on the new DOS, namely $DOS_3 = PCS \cup DG \cup O_l, O_m$ The procedure is repeated until a consistent DOS is found ($DOS_{max} = DOS_1$ in this example). At this point, the DOS_{max} is saved to be used in the DG for the next related backtracking episode.

[5] Because full consistency checking is expensive, if this set is too large, two approaches can be taken to limit computational cost: (1) full consistency checking can be performed only for a subset of the DOS, or (2) k-consistency [10] can be performed, where k is some predetermined number.

[6] Clearly, there is no guarantee that the search state in which DCE stops backtracking is a consistent search state. Experimental results suggest however that this is often the case.

[7] Two backtracking episodes are defined to be related if they are due to capacity constraint violations on *the same resource and over close time intervals.* Otherwise, they are unrelated.

4.1 Management of Dangerous Groups

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 the 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 the FDG are built from the *Deadend Operation Sets* (DOS) obtained at the end of previous backtracking episodes (DOS_{max}). Indeed, whenever a backtracking episode is completed, DOS_{max} is expected to contain all the conflicting operations at the origin of this episode. Generally, DOS_{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 the FDG. If there is no related group in FDG, the new group is separately added to the data structure.

As operations are scheduled, they are removed from the FDG.

4.2 Additional "Watch Dog" Consistency Checks

Because groups of operations in the FDG are likely deadend candidates, our system further performs simple "watch dog" consistency checks on these dynamic groups of operations. More specifically, for each group G of operations in FDG, the system performs a rough consistency check to see if the resource can still accommodate all the operations in the group. This is done using redundant constraints of the form: $Max(lst_i^l + du_i^l, O_i^l \in G) - Min(est_i^l, O_i^l \in G) \geq \sum_{O_i^l \in G} du_i^l$ 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 [8]. Clearly, some inconsistencies can still escape these rough checks.

[8] Notice that, when a "watch dog" check fails, PCS is empty.

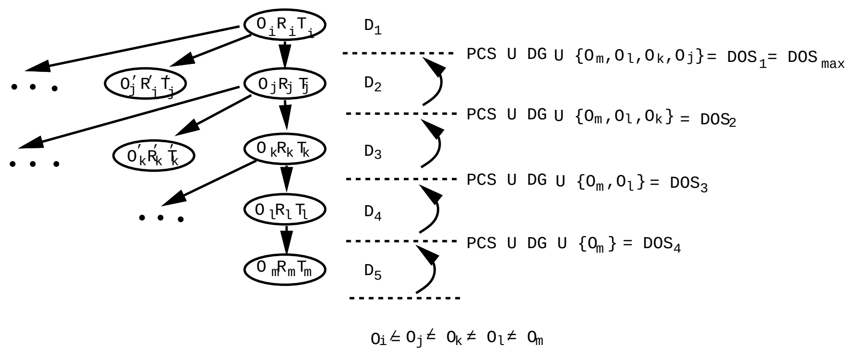

Figure 1: The DCE Backtracking Scheme.

5 Learning From Failures (LFF)

Encounter of a deadend is also often 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 than the operations selected by the default variable ordering heuristic. It is therefore a good idea to first schedule the operations participating in the conflict that was just resolved. *Learning From Failure* (LFF) is an adaptive procedure that overrides the default variable ordering in the presence of conflicts.

After recovering from a deadend (i.e. after backtracking all the way to an apparently consistent search state), LFF uses the Partial Conflicting Set (PCS) of the deadend to reorganize the ordering in which operations will be rescheduled and make sure that operations in the PCS are scheduled first. This is done using a quasi-stack on which operations in the PCS are pushed in descending order of domain size (operations with more available start times go first)[9]. This orders operations in terms of their criticality (most critical operation on top) so as to ensure that, as S is popped, the most critical operations will be scheduled first. As long as S is non-empty, operations from S are popped and successively scheduled, thus overriding the default variable ordering.

6 A Backjumping Heuristic

Traditional
backtrack search procedures only undo decisions that have been proven to be wrong/inconsistent. Proving that an assignment is inconsistent with others can be very expensive, especially when dealing with large conflicts. Graph-based backjumping and N-th order shallow/deep learning attempt to reduce the complexity of fullblown 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. None of these techniques can be expected to perform well when dealing with large complex [10] conflicts, either because they are too expensive to run or because they deliberately overlook large conflicts. Large complex conflicts can force the search procedure to thrash, even when using procedures such as DCE. In these situations, it may be worth undoing decisions that are not provably wrong but simply appear overly restrictive. Clearly, the resulting search procedure is no longer complete and may fail to find solutions to feasible problems.

Texture measures such as the ones described in [9] 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 [11]. Assignments leading to much tighter search states would be prime candidates to be undone. The *Backjumping Heuristic* (BH) used in this study is simpler and, yet, often seems to get the job done. Whenever the system starts thrashing, this heuristic backjumps all the way to the first search state and simply tries the next best value (i.e. reservation) for the critical operation in that state (i.e. the first operation selected by the variable ordering heuristic). BH considers that the search procedure is thrashing when more than θ assignments had to be undone since the procedure began or since the last time the system was thrashing,

[9] If a candidate operation is already on S, i.e. it is encountered for a second time, it is pushed again as though it had a smaller domain.

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

[11] A search state whose partial solution is compatible with a large number of global solutions is a loosely constrained search state, whereas one compatible with a small number of global solutions is tightly constrained.

where θ is a parameter of the search procedure.

7 Experimental Results

Two sets of 40 scheduling problems each were generated that differed in the number of major bottlenecks (one and two major bottlenecks respectively). Each problem had 50 operations and 5 resources (i.e., 10 jobs). All jobs were released at the same time and had to be completed by the same due date. In each problem, the common due date was set so that all operations had to be scheduled within a rather tight estimate of the problem makespan (see [20] for details). These are the conditions in which the default variable/value ordering and consistency enforcing schemes work least effectively (see study reported in [20]). Among these 80 problems, we only report performance on problems in which the default schemes were not sufficient to guarantee backtrack-free search [12]. This leaves 16 scheduling problems with one bottleneck, and 15 with two bottlenecks.

We successively report the results of two studies. The first study compares the performance of three complete backtrack schemes: chronological backtracking, 2d-order deep learning, and the procedure combining the DCE and LFF backtrack schemes described in Section 4 and 5. The second study compares a complete search procedure using the DCE and LFF schemes with an incomplete search procedure combining DCE and LFF with the Backjumping Heuristic (BH) described in Section 6.

7.1 Comparison of Complete Search Procedures

The two intelligent backtracking techniques, DCE and LFF are complementary and were used in combination, denoted by DCE & LFF, for experimentation to assess performance[13]. Each of the problems in the experiment set was run using chronological backtracking, 2d-order deep learning [6] and the DCE & LFF procedures advocated in Section 4 and 5. 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 times reported below were obtained on a DECstation 5000 running Knowledge Craft on top of Allegro Common Lisp [15].

Results for the one-bottleneck problems are reported in Table 1. Chronological backtracking solved only 4 problems out of 16. 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 naive backtracking was originally ascertained. On the other hand, DCE & LFF 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 & LFF is able to discover larger more useful conflicts than 2d-order deep learning, while requiring only a fraction of the time required by deep learning. Another observation is that DCE & LFF expanded fewer search states than chronological backtracking for the problems that chronological backtracking solved. However, each of the DCE & LFF 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 2. Similar results are observed here again: deep learning shows no improvement over chronological backtracking and seems significantly slower. The difference between chronological backtracking and DCE&LFF is not as impressive as in the first set of experiments. This is probably because both bottlenecks may have capacity conflicts at the same time. DCE & LFF may then have problems determining which one to consider first. As can be seen from Table 2, chronological backtracking solved 7 out of 15 problems, whereas DCE & LFF solved 8 out of 15. On the problems solved by both chronological backtracking and DCE & LFF, DCE & LFF turned out to be slightly faster overall.

7.2 Complete vs. Incomplete Search Procedures

Table 3 and 4 compare the performance of the complete search procedure using DCE & LFF against that of an incomplete search procedure using DCE & LFF in combination with the Backjumping Heuristic (BH) described in Section 6. While DCE & LFF was able to solve only 10 out of 16 one-bottleneck problems and 8 out 15 two-bottleneck problems, DCE & LFF combined with BH solved 14 one-bottleneck problems and 13 two-bottleneck problems. The only one-bottleneck

[12] Clearly, performance in the absence of backtracking is uninteresting, since our backtracking schemes would never be invoked, i.e. CPU time remains unchanged.

[13] Besides the experiments reported below, additional experiments were performed to assess the benefits of using DCE and LFF 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.

[15] Comparison between C programs and Knowledge Craft programs suggests that the code would run 10 to 20 times faster in C.

Table 1: Results of One-bottleneck Experiments.

Exp. No.	Chronological Backtracking			DCE & LFF			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

S: Solved ; F: Failure; S*: Proved infeasible
Time Limit: 1800 sec (Except Deep Learning)
Node Limit: 500

Table 2: Results of Two-bottleneck Experiments

Exp. No.	Chronological Backtracking			DCE & LFF			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

S: Solved ; F: Failure; S*: Proved infeasible
Time Limit : 1800 sec. (36000 sec. for Deep Learning)
Node Limit : 500

problems that were not solved by DCE & LFF & BH are the two infeasible problems identified by the complete search procedure DCE & LFF. This is hardly a surprise. While the addition of BH to DCE & LFF 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 identified). Additional experiments combining BH with a simple chronological backtracking scheme also indicate that both DCE & LFF and BH contribute to the good performance of DCE & LFF & BH. Results on two-bottleneck problems (See Table 4) suggest that the impact of the backjumping heuristic 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 may simply be too difficult for any exact backtracking scheme to work. Instead, because it can undo assignments that are not provably wrong but simply appear overly restrictive, BH seems more effective at dealing with these more complex conflicts.

8 Concluding Remarks

We have presented three intelligent backtracking schemes for the job shop scheduling CSP:

1. *Dynamic Consistency Enforcement* (DCE), a dependency-directed scheme, that dynamically focuses its effort on small critical subproblems,
2. *Learning From Failure* (LFF), which modifies the order in which variables are instantiated based on earlier conflicts, and
3. a *Backjumping Heuristic* which, when thrashing occurs, can undo assignments that are not provably inconsistent but appear overly restrictive.

Table 3: Results of One-bottleneck Experiments.

Exp. No.	DCE & LFF			DCE & LFF & BH		
	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

S: Solved ; F: Failure; S*: Proved infeasible
Time Limit: 1800 sec. Node Limit: 500

Table 4: Results of Two-bottleneck Experiments

Exp. No.	DCE & LFF			DCE & LFF & BH		
	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

S: Solved ; F: Failure; S*: Proved infeasible
Time Limit: 1800 sec. Node Limit: 500

The significance of this research is twofold:

1. Job shop scheduling problems with non-relaxable time windows have multiple applications, including both manufacturing and space-related applications. We have shown that our schemes combined with powerful techniques that we had previously developed (1) further reduce the average complexity of backtrack search, and (2) enable our system to efficiently solve problems that could not be solved otherwise due to excessive computational cost. 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.

2. This research also points to the deficiencies of dependency-directed backtracking schemes advocated earlier in the literature. In particular, comparison with N-th order deep learning indicates that this technique failed (in our set of experiments) to improve performance when applied to job shop scheduling problems. This is because N-th order deep learning uses constraint size as the only criterion to decide whether or not to record earlier failures. When deep learning limits itself to small-size conflicts, it fails to record some important constraints; when it considers conflicts of larger size, its computational complexity becomes prohibitive. Traditional backtracking schemes never undo assignments unless they can prove that they are 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 backjumping heuristics that undo decisions simply because they *appear* overly restrictive. When using such heuristics, search completeness can no longer be guaranteed.

Acknowledgements

This research was supported, in part, by the Defense Advanced Research Projects Agency under contract F30602-91-C-0016 and, in part, by the Robotics Institute at Carnegie Mellon University.

References

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

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

[3] Peter Burke and Patrick Prosser. A distributed asynchronous system for predictive and reactive scheduling. Technical Report AISL-42, Department of Computer Science, University of Strathclyde, 26 Richmond Street, Glasgow, GI IXH, United Kingdom, October 1989.

[4] Rina Dechter. Learning while searching in constraint satisfaction problems. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 178–183, 1986.

[5] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1989.

[6] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, 1989.

[7] Rina Dechter and Judea Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1988.

[8] John Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[9] Mark S. Fox, Norman Sadeh, and Can Baykan. Constrained heuristic search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 309–315, 1989.

[10] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[11] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Co., 1979.

[12] John Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, 1979.

[13] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *Journal of the Association for Computing Machinery*, 12(4):516–524, 1965.

[14] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[15] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1985.

[16] Jr. Paul W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[17] N. Sadeh and M.S. Fox. Preference propagation in temporal/capacity constraint graphs. Technical Report CMU-CS-88-193, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, 1988. Also appears as Robotics Institute technical report CMU-RI-TR-89-2.

[18] N. Sadeh and M.S. Fox. Focus of attention in an activity-based scheduler. In *Proceedings of the NASA Conference on Space Telerobotics*, January 1989.

[19] 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 15213, 1992.

[20] Norman Sadeh. *Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1991.

[21] Norman Sadeh and Mark S. Fox. Variable and value ordering heuristics for activity-based job-shop scheduling. In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management, Hilton Head Island, S.C.*, pages 134–144, 1990.

[22] R. Stallman and G. Sussman. Forward reasoning and dependency-directed backtracking in a sysem for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[23] K. Sycara, S. Roth, N. Sadeh, and M. Fox. Distributed constrained heuristic search. *IEEE Transactions on System, Man and Cybernetics*, 21(6), 1991.

[24] R.J. Walker. *An Enumerative Technique for a Class of Combinatorial Problems*, volume 10, chapter 7, pages 91–94. American Mathematical Society, Rhode Island, 1960.