Dead-end driven learning *

Daniel Frost and Rina Dechter
Dept. of Information and Computer Science
University of California, Irvine, CA 92717
{dfrost,dechter}@ics.uci.edu

Abstract

The paper evaluates the effectiveness of learning for speeding up the solution of constraint satisfaction problems. It extends previous work (Dechter 1990) by introducing a new and powerful variant of learning and by presenting an extensive empirical study on much larger and more difficult problem instances. Our results show that learning can speed up backjumping when using either a fixed or dynamic variable ordering. However, the improvement with a dynamic variable ordering is not as great, and for some classes of problems learning is helpful only when a limit is placed on the size of new constraints learned.

1. Introduction

Our goal in this paper is to study the effect of *learning* in speeding up the solution of constraint problems. The function of learning in problem solving is to record in a useful way some information which is explicated during the search, so that it can be reused either later on the same problem instance, or on similar instances which arise subsequently. The approach we take involves a during-search transformation of the problem representation into one that may be searched more effectively. This is done by enriching the problem description by new constraints (sometimes called *nogoods*), which do not change the set of solutions, but make certain information explicit. The idea is to learn from dead-ends; whenever a dead-end is reached we record a constraint explicated by the dead-end.

This type of learning has been presented in dependency-directed backtracking strategies in the TMS community (Stallman & Sussman 1977), and within intelligent backtracking for Prolog(Bruynooghe & Pereira 1984). Recently, it was treated more systematically by Dechter (1990) within the constraint network framework. Different variants of learning were examined there, while taking into account the tradeoff between the overhead of learning and performance improvement. The results, although preliminary, indicated that learning could be cost-effective.

The present study extends (Dechter 1990) in several ways. First, a new variant of learning, called jumpback learning, is introduced and is shown empirically to be superior to other types of learning. Secondly, we experiment with and without restrictions on the size of the constraints learned. Thirdly, we use a highly efficient version of backjumping as a comparison reference. Finally, our experiments use larger and harder problem instances than previously studied.

2. Definitions and Preliminaries

A Constraint Network consists of a set of n variables, $X_1, \ldots, X_n$; their respective value domains, $D_1, \ldots, D_n$; and a set of constraints. A *constraint* $C_i(X_{i_1}, \ldots, X_{i_j})$ is a subset of the Cartesian product $D_{i_1} \times \ldots \times D_{i_j}$, consisting of all tuples of values for a subset $(X_{i_1}, \ldots, X_{i_j})$ of the variables which are compatible with each other. A *solution* is an assignment of values to all the variables such that all the constraints are satisfied. Sometimes the goal is to find all solutions; in this paper, however, we focus on the task of finding one solution, or proving that no solution exists. A constraint satisfaction problem (CSP) can be associated with a *constraint graph* consisting of a node for each variable and an arc connecting each pair of variables that are contained in a constraint. A *binary* CSP is one in which each of the constraints involves at most two variables.

Backjumping

Many algorithms have been proposed for solving CSPs. See (Dechter 1992; Mackworth 1992) for reviews. One algorithm that was shown always to dominate naive backtracking is *backjumping* (Gaschnig, 1979; Dechter, 1990). Like backtracking, backjumping considers each variable in some order and assigns to each successive variable a value from its domain which is consistent with the values assigned to the preceding variables. When a variable is encountered such that none of its possible values is consistent with previous assignments (a situation referred to as a *dead-end*), a backjump

*This work was partially supported by NSF grant IRI-9157636, by Air Force Office of Scientific Research grant AFOSR 900136 and by grants from Toshiba of America and Xerox.

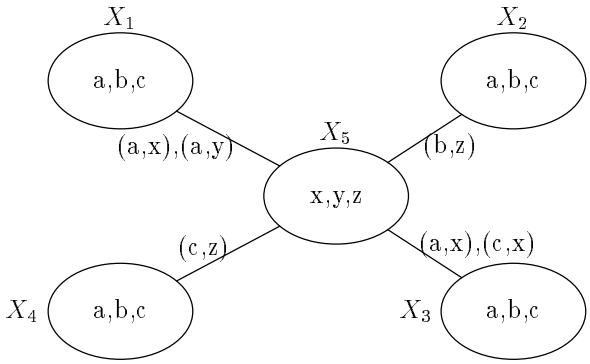

Figure 1: A small CSP. Note that the *dis*allowed pairs are shown on each arc.

takes place. The idea is to jump back over several irrelevant variables to a variable which is more directly responsible for the current conflict. The backjumping algorithm identifies a *jump-back set*, that is, a subset of the variables preceding the dead-end variable which are inconsistent with all its values, and continues search from the last variable in this set. If that variable has no untried values left, then a *pseudo dead-end* arises and further backjumping occurs.

Consider, for instance, the CSP represented by the graph in Fig. 1. Each node represents a variable that can take on a value from within the oval, and the binary constraint between connected variables is specified along the arcs by the disallowed value pairs. If the variables are ordered $(X_1, X_5, X_2, X_3, X_4)$ and a dead-end is reached at X_4, the backjumping algorithm will jump back to X_5, since X_4 is not connected to X_3 or X_2.

The version of backjumping we use here is a combination of Gaschnig's (1979) backjumping and Dechter's (1990) graph-based backjumping, as proposed by Prosser (1993). Prosser calls the algorithm *conflict-directed* backjumping. In this version, the jump-back set is created by recording, for each value v of V, the variable to be instantiated next, the first past variable (relative to the ordering) whose assigned value conflicts with $V = v$. The algorithm will be combined with both fixed and dynamic variable orderings.

Variable Ordering Heuristics

It is well known that variable ordering affects tremendously the size of the search space. In previous studies it has been shown that the *min-width* ordering is a very effective fixed ordering (Dechter & Meiri 1989), while dynamic variable ordering (Haralick & Elliott 1980; Purdom 1983; Zabih & McAllester 1988) frequently yields best performance. We incorporate both strategies in our experiments.

The *minimum width* (MW or min-width) heuristic (Freuder 1982) orders the variables from last to first by selecting, at each stage, a variable in the constraint graph that connects to the minimal number of variables that have not yet been selected. For instance, the ordering X_1, X_5, X_2, X_3, X_4 is a min-width ordering of the graph in Fig. 1.

Dynamic variable ordering (DVO) allows the order of variables to change during search. The version we use selects at each point the variable with the smallest remaining domain size, when only values that are consistent with all instantiated variables are considered. Ties are broken randomly. The variable that participates in the most constraints is selected to be first in the ordering. If any future variable has an empty domain, then it is moved to be the next in the ordering, and a dead-end will occur on that variable. Otherwise, a variable with the smallest domain size is selected (similar to unit-propagation in Boolean satisfiability problems).

3. Learning Algorithms

In a dead-end at X_i, when the current instantiation $S = (X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$ cannot be extended by any value of X_i, we say that S is a *conflict set*. An opportunity to learn new constraints is presented whenever backjumping encounters a dead-end, since had the problem included an explicit constraint prohibiting the dead-end's conflict-set, the dead-end would have been avoided. To learn at a dead-end, we record a new constraint which makes explicit an incompatibility among variable assignments that already existed, implicitly. The trade-off involved is in possibly finding out earlier in the remaining search that a given path cannot lead to a solution, versus the cost of having to process a more extensive database of constraints.

There is no point in recording S as a constraint at this stage, because this state will not recur. However, if S contains one or more subsets that are also in conflict with X_i, then recording these smaller conflict sets as constraints may prove useful in the continued exploration of the search space because future states may contain these subsets.

Different types of learning differ in the way they identify smaller conflict sets. In (Dechter 1990) learning is characterized as being either *deep* or *shallow*. Deep learning only records *minimal* conflict sets, that is, those that do not have subsets which are conflict sets. Shallow learning allows recording non-minimal conflict sets as well. Learning can also be characterized by *order*, the maximum constraint size that is recorded. In (Dechter 1990) experiments were limited to recording unary and binary constraints, since constraints involving more variables are applicable less frequently, require more space to store, and are more expensive to consult.

In this paper we experiment with four types of learning: *graph-based shallow* learning, *value-based shallow* learning, and *deep* learning, already presented in (Dechter 1990), as well as a new type, called *jump-back* learning.

In **value-based learning** all irrelevant variable-value pairs are removed from the initial conflict set S. If a variable-value pair $X_j = x_j$ doesn't conflict with any value of the dead-end variable then it is redundant and can be eliminated. For instance, if we try to solve the problem in Fig. 1 with the ordering $(X_1, X_2, X_3, X_4, X_5)$, after instantiating $X_1 = a, X_2 = b, X_3 = b, X_4 = c$, the dead-end at X_5 will cause value-based learning to record $(X_1 = a, X_2 = b, X_4 = c)$, since the pair $X_3 = b$ is compatible with all values of X_5. Since we can pre-compute in $O(n^2k)$ time a table that will tell us whether $X_i = x_j$ conflicts with any value of each other variable, the complexity of value-based learning at each dead-end is $O(n)$.

Graph-based shallow learning is a relaxed version of value-based learning, where information on conflicts is derived from the constraint graph alone. This may be particularly useful on sparse graphs. For instance, in Fig. 1 graph-based shallow learning will record $(X_1 = a, X_2 = b, X_3 = b, X_4 = c)$ as a conflict set relative to X_5, since all variables are connected to X_5. The complexity of learning at each dead-end here is $O(n)$, since each variable is connected to at most $n - 1$ other variables.

Jump-back learning uses as the conflict-set the jump-back set that is explicated by the backjumping algorithm itself. Recall that conflict-directed backjumping examines, starting from the first variable, each instantiated variable and includes it in the jump-back set if it conflicts with a value of the current variable that previously did not conflict with any variable. For instance in Fig. 1, when using the same ordering and reaching the dead-end at X_5, jump-back learning will record $(X_1 = a, X_2 = b)$ as a new constraint. These two variables are selected because the algorithm first looks at $X_1 = a$ and notes that it conflicts with $X_5 = x$ and $X_5 = y$. Proceeding to $X_2 = b$, the conflict with $X_5 = z$ is noted. At this point all values of X_5 have been ruled out, and the conflict set is complete. Since the conflict set is already assembled by the underlying backjumping algorithm, the added complexity of computing the conflict set is constant.

In **deep learning** all and only minimal conflict sets are recorded. In Fig. 1, deep learning will record two minimal conflict sets, $(X_1 = a, X_2 = b)$ and $(X_1 = a, X_4 = c)$. Although this form of learning is the most accurate, its cost is prohibitive and in the worst-case is exponential in the size of the initial conflict set (Dechter 1990).

4. Complexity of backtracking with learning

We will now show that graph-based learning yields a useful complexity bound on the algorithm performance, relative to a graph parameter known as w^*. Since graph-based learning is the most conservative learning algorithm (when no order restrictions are imposed), the bound is applicable to all the variants of learning we discuss.

	Cross-over value of C	
N	$T = 1/9$	$T = 2/9$
25	199	89
50	380	166
75	565	244
100	747	317
150	1100	468
200	1477	621
250	1842	771

Figure 2: Empirically determined values of C that generate 50% solvable CSP instances. $K = 3$ for this data.

Given a constraint graph and a fixed ordering of the nodes d, the *width* of a node is the number of arcs that connect that node to previous ones, called its *parents*. The width of the graph relative to d is the maximum width of all nodes in the graph. The induced graph is created by considering each node in the original graph in order from last to first, and adding arcs connecting each of its parents to each other parent. The induced width of an ordering, *w*(d)*, is the width of its induced graph.

Theorem 1: Let d be an ordering and let *w*(d)* be its induced width. Any backtrack algorithm using ordering d and graph-based learning has a space complexity of $O((nk)^{w^*(d)})$ and a time complexity of $O((2nk)^{w^*(d)})$.

Proof: Due to graph-based learning there is a one-to-one correspondence between dead-ends and conflict sets. It can be shown that backtracking with graph-based learning along d records conflict-sets of size *w*(d)* or less. Therefore the number of dead-ends is bounded by

$$\sum_{i=1}^{w^*(d)} \binom{n}{i} k^i = O((nk)^{w^*(d)}).$$

This gives the space complexity. Since deciding that a dead-end occurred requires testing all constraints containing the dead-end variable and at most *w*(d)* prior variables, at most $O(2^{w^*(d)})$ constraints are checked per dead-end, yielding a time bound of

$$O((2nk)^{w^*(d)}).$$

5. Methodology and Results

The experiments reported in this paper were run on random instances generated using a four parameter model: N, K, T and C. The problem instances have N variables, each having a domain of size K. The problems we experiment with always start off as binary CSPs, but can become non-binary as constraints involving more than two variables are added by learning. The parameter T (tightness) specifies a fraction

N	K	Statistic	No Learning	With this type of learning			
				Graph-based	Value-based	Jump-back	Deep
25	3	CC	16,930	30,636	29,185	10,203	117,556
		DE	156	178	181	82	67
		CPU secs	0.048	0.083	0.077	0.032	0.325
		NGs		178	181	82	153
		Size		11.6	6.8	3.5	3.4
25	6	CC	274,133	1,340,512	1,428,109	330,672	55,771,462
		DE	2,777	2,833	2,932	1,276	832
		CPU secs	0.777	2.067	2.183	0.667	78.283
		NGs		2,833	2,932	1,276	1894
		Size		11.2	10.4	5.2	4.4
50	3	CC	303,668	8,051,435	7,111,384	119,642	27,134,341
		DE	2,205	5,107	4,512	437	333
		CPU secs	1.298	11.492	9.913	0.367	44.788
		NGs		5,107	4,512	437	654
		Size		20.6	13.6	4.6	4.2

Figure 3: Detailed results of comparing backjumping with no learning to backjumping with each of four kinds of learning. $T = 1/9$ and C is set to the cross-over point. See the text for discussion.

of the K^2 value pairs in each constraint that are disallowed by the constraint. The incompatible pairs in a constraint are selected randomly from a uniform distribution, but each constraint will always have the same fraction T of such incompatible pairs. T ranges from 0 to 1, with a low value of T, such as 1/9, termed a loose or relaxed constraint. The fourth parameter, C, specifies the number of constraints out of the $N*(N-1)/2$ possible. Constraints are chosen randomly from a uniform distribution.

As in previous studies (Cheeseman, Kanefsky, & Taylor 1991; Mitchell, Selman, & Levesque 1992), we observed that the hardest instances tend to be found where about half the problems are solvable and half are not (the "cross-over" point). Most of our experiments were conducted with instances drawn from this 50% range; the necessary parameter combinations were determined experimentally (Frost & Dechter 1994) and are given in Fig. 2.

Results

We first compared the effectiveness of the four learning schemes. Fig. 3 presents a summary of experiments with sets of problems of several sizes (N) and number of values (K). 100 problems in each class were generated and solved by five algorithms: backjumping without learning, and then backjumping with each of the four types of learning. In all cases a min-width variable ordering was applied and no bound was placed on the size of the constraints recorded. For each problem instance and for each algorithm we recorded the number of consistency checks (CC), the number of (non-pseudo) dead-ends (DE), the CPU time (CPU secs), the number of new nogoods recorded (NGs), and the average size of (number of variables in) the learned constraints. A consistency check is recorded each time the algorithm checks if the values of two or more variables are consistent with respect to the constraint between them. The number of dead-ends is a measure of the size of the search space explicated. All experiments were run using a single program with as much shared code and data structures as possible. Therefore we believe CPU time is a meaningful comparative measure.

This experiment demonstrated that only the new jump-back type of learning was effective on these reasonably large size problems. Once the superiority of jump-back learning was established we stopped experimenting with other types of learning. In the following discussion and figures, all references to learning should be taken to mean jump-back learning.

To determine whether learning would be effective for CSPs with many variables, in our next set of experiments we generated instances from parameters $K = 3$, $T = 1/9$, $N = \{25, 50, 75, 100\}$, and C set to the appropriate cross-over points. We used backjumping with a min-width ordering on 200 instances in each category, both with and without learning. (No limit was placed on the order of learning.) The mean numbers of consistency checks, dead-ends, and CPU seconds are reported in Fig. 4. The results were encouraging: by all measures learning provided a substantial improvement when added to backjumping. (Experiments with $T = 2/9$ and $T = 3/9$, not reported here due to space, show a similar pattern.) Our only reservation was that from other work we knew that a dynamic variable ordering can be a significant improvement over the static min-width. Would learning be able to improve backjumping with DVO?

To find out, we ran another set of experiments, using the same instances, plus some generated with higher values of N; this time backjumping used a dynamic variable ordering. We also experimented with vari-

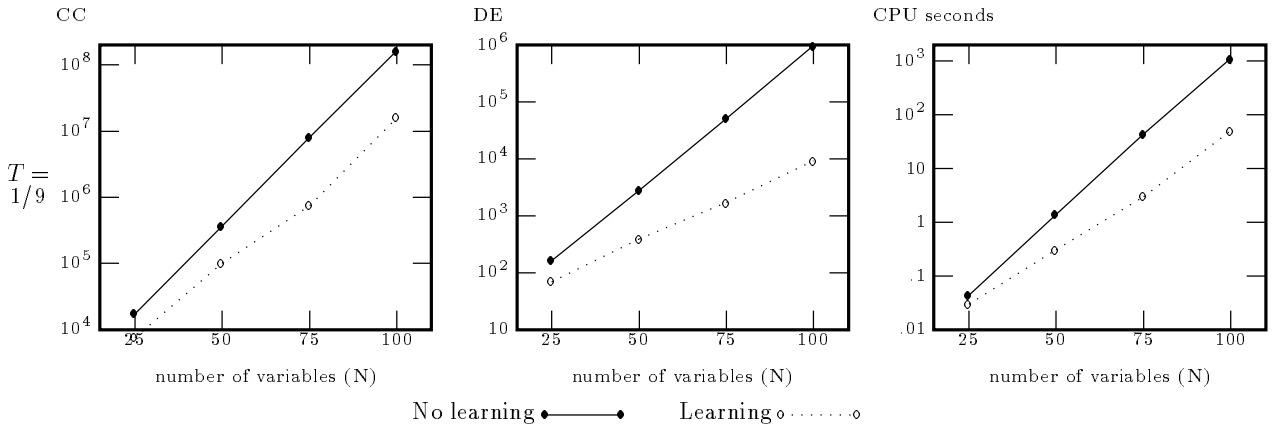

Figure 4: Comparison of BJ+MW with and without learning (of unlimited order); $K = 3$.

ous orders of learning. Recall that in i-order learning, new constraints are recorded only if they include i or fewer variables. In (Dechter 1990) experiments were conducted with first and second order learning. Here, we tried second-, third-, and fourth-order learning, as well as learning without restriction on the size of new constraints. However, only third-order and unlimited-order learning are reported, due to space constraints, in Fig. 6.

We make several observations from these data. First, learning becomes more effective as the number of variables in the problem increases. With DVO, when $N < 100$, the absence or presence of learning, of whatever order, makes very little difference. With the powerful DVO ordering, there are too few dead-ends for learning to be useful, or for the overhead of learning to cause problems. As N increases from 100 on up, learning becomes more effective. For instance, looking at data for $T = 2/9$, and comparing CPU time for No Learning with CPU time for unlimited order learning, we see improvements at $N = 100$ of 1.6 (0.815 / 0.499), at $N = 150$ of 6.9 (25.463 / 3.710), and at $N = 200$ of 7.8 (170.990 / 21.808).

A second observation is that when the individual constraints are loose ($T = 1/9$), learning is at best only slightly helpful and can sometimes deteriorate performance. The reason is that the conflict-sets with loose constraints tend to be larger, since each variable in the conflict set can invalidate (in the case of $T = 1/9$) only one value from the dead-end variable's domain.

Thirdly, we note that as the order of learning becomes higher, the size of the search space decreases (as measured by the number of dead-ends), but the amount of work at each node increases, indicated by the larger count of consistency checks. For instance, the data for $N = 200, T = 2/9$, show that in going from third-order learning to unlimited order learning, dead-ends go down slightly while consistency checks increase by a factor of five. The overall CPU time for the two versions is almost identical, because in our implementation consistency checking is implemented very efficiently. If the cost to perform each consistency check were higher in relation to the cost to expand the search to a new node, unlimited learning might require more CPU time than restricted order learning.

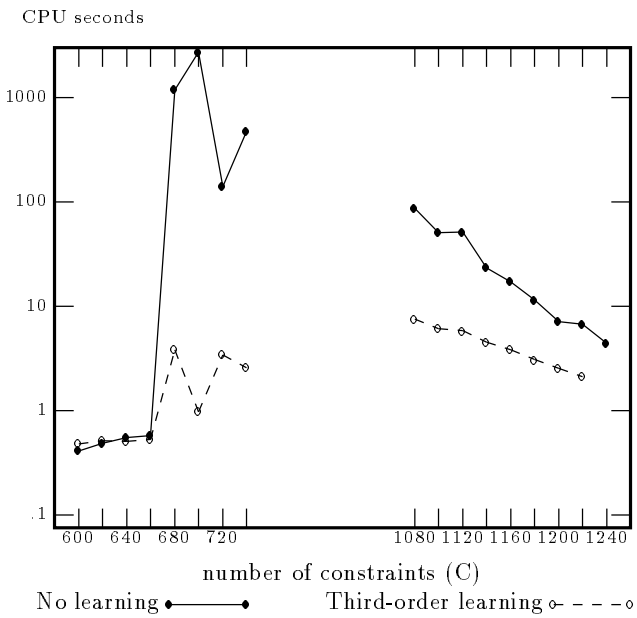

Figure 5: BJ+DVO without learning and with third-order learning, for N=300, K=3, T=2/9, and non-50% values of C. All problems with $C \leq 740$ were solvable; all with $C \geq 1080$ had no solution.

As expected, learning is more effective on problem instances that have more dead-ends and larger search spaces, where there are more opportunities for each learned constraint to be useful. Comparing the means

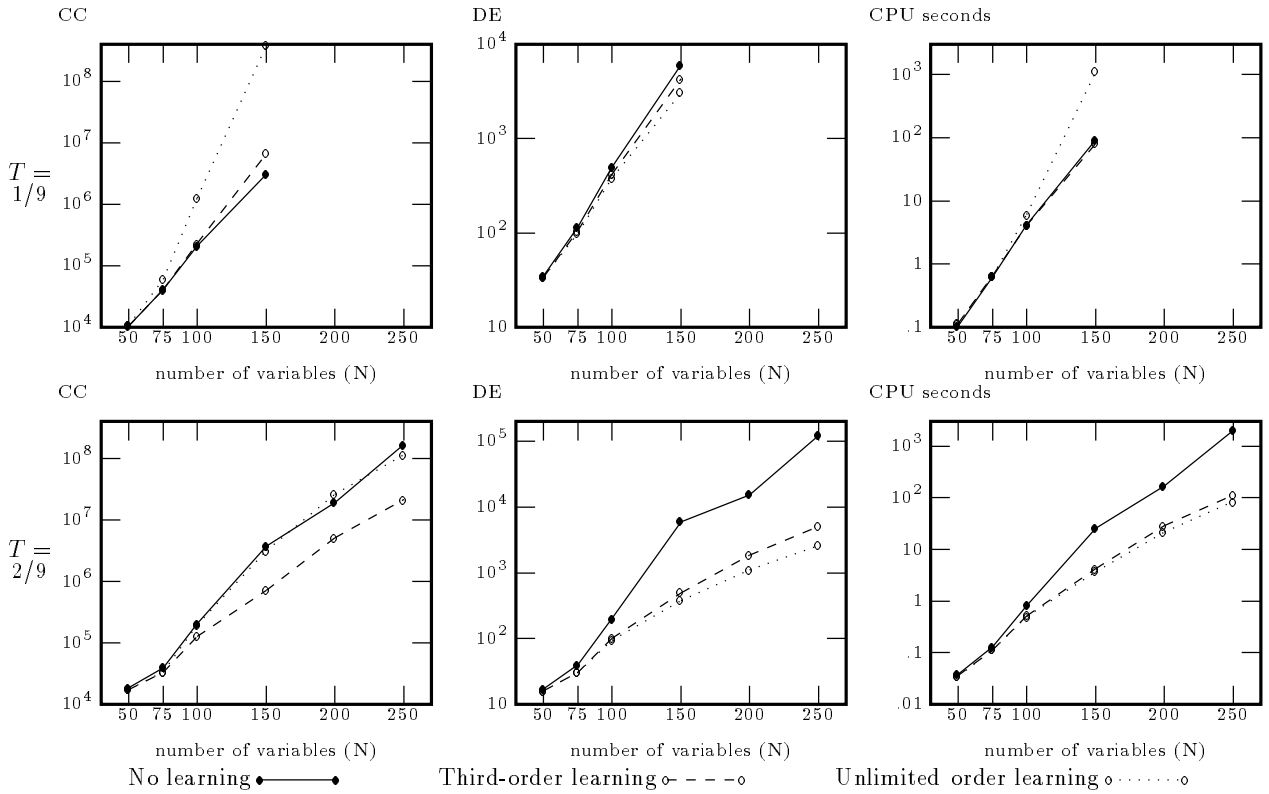

Figure 6: Comparison of BJ+DVO, without learning, and with third-order and unlimited order learning; $K = 3$.

of 200 problem instances solved both with and without learning can obscure the trend that the improvement from learning is generally much greater for the very hardest instances of the population. For instance, the data for $N = 200, T = 2/9$ show that the mean CPU time for 200 instances is 170.990 without learning and 21.808 with learning, improving by a factor of 7.8 (170.990 / 21.808). If we just consider the 20 problem instances out of the 200 which required the most CPU time to be solved without learning, the mean CPU time of those 20 instances without learning is 1107.18, and 72.48 with unlimited order learning. The improvement for the hardest problems is a factor of 15, about twice that of the entire sample.

Fig. 5 shows that with large enough N, problems do not have to be drawn from the 50% satisfiable area in order to be hard enough for learning to help. Learning was especially valuable on extremely hard solvable problems generated by slightly underconstrained values for C. For instance, at $N = 300, K = 3, T = 2/9, C = 680$, the hardest problem (out of 200 instances) took 47 CPU hours without learning, and under one CPU minute with learning. The next four hardest problems took 4% as much CPU time with learning as without.

Controlling the order of learning has a greater impact on the constraints recorded as N increases. We see this in Fig. 7 (drawn from the same set of experiments as Fig. 6), where the average constraint size increases for unlimited order learning, but not for third-order. The primary cause of this effect is that learned non-binary constraints are becoming part of conflict-sets. The first constraint learned with these parameters (particularly $K = 3$) can have at most three variables in it, one eliminating each value of the dead-end. Once a 3-variable constraint exists, it may contribute two variables to a conflict set, and thus a four variable conflict set can arise. For $N = 250$, the largest conflict set we observed had 11 elements. Recording such a constraint is unlikely to be helpful later in the search.

It is worth noting that we did not find the space requirements of learning to be overwhelming, as has been reported by some researchers. For instance, the average problem at $N = 250$ and $T = 2/9$ took about 100 CPU seconds and recorded about 2600 new constraints (with unlimited order learning). Each constraint requires fewer than 25 bytes of memory, so the total added memory is well under one megabyte. We found that computer memory is not the limiting factor; time is.

	Order 3			Unlimited order		
N	Dead-ends	Learned	Avg. Size	Dead-ends	Learned	Avg. Size
50	16	15	2.20	16	16	2.25
75	31	30	2.06	31	31	2.11
100	101	90	2.11	94	94	2.27
150	499	447	2.07	383	383	2.29
200	1,874	1,561	2.09	1,110	1,110	2.58
250	5,119	4,046	2.10	2,608	2,608	2.86

Figure 7: Figures for $T = 2/9$; learning with BJ+DVO. "Learned" is number of new constraints learned; "Avg. Size" is the average number of variables in the constraints.

5. Conclusions

We have introduced a new variant of learning, called jump-back learning, which is more powerful than previous versions. Our experiments show that it is very effective when augmented on top of an efficient version of backjumping, resulting in at least an order of magnitude reduction in CPU time for some problems.

Learning seems to be particularly effective when applied to instances that are large or hard, since it requires many dead-ends to be able to augment the initial problem in a significant way. However, on easy problems with few dead-ends, learning will add little if any cost, thus perhaps making it particularly suitable for situations in which there is a wide variation in the hardness of individual problems. In this way learning is superior to other CSP techniques which modify the initial problem, such as by enforcing a certain order of consistency, since the cost will not be incurred on very easy problems. Moreover, we have shown that the performance of any backtracking algorithm with learning, using a fixed ordering, is bounded by $\exp(w^*)$.

An important parameter when applying learning is the order, or maximum size of the constraints learned. With no restriction on the order, it is possible to learn very large constraints that will be unlikely to prune the remaining search space. We plan to study the relationship between K, the size of the domain, and the optimal order of learning. Clearly with higher values of K, the order may need to be higher, especially for loose constraints, possibly rendering learning less effective.

References

Bruynooghe, M., and Pereira, L. M. 1984. Deduction revision by intelligent backtracking. In Campbell, J. A., ed., *Implementation of Prolog*. Ellis Horwood. 194–215.

Cheeseman, P.; Kanefsky, B.; and Taylor, W. M. 1991. Where the *really* hard problems are. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 331–337.

Dechter, R., and Meiri, I. 1989. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence*, 271–277.

Dechter, R. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence* 41:273–312.

Dechter, R. 1992. Constraint networks. In *Encyclopedia of Artificial Intelligence*. John Wiley & Sons, 2nd edition.

Freuder, E. C. 1982. A sufficient condition for backtrack-free search. *JACM* 21(11):958–965.

Frost, D., and Dechter, R. 1994. Search for the best constraint satisfaction search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*.

Gaschnig, J. 1979. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University.

Haralick, R. M., and Elliott, G. L. 1980. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* 14:263–313.

Mackworth, A. K. 1992. Constraint satisfaction problems. In *Encyclopedia of Artificial Intelligence*. John Wiley & Sons, 2nd edition.

Mitchell, D.; Selman, B.; and Levesque, H. 1992. Hard and Easy Distributions of SAT Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 459–465.

Prosser, P. 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9(3):268–299.

Purdom, P. W. 1983. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence* 21:117–133.

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

Zabih, R., and McAllester, D. 1988. A Rearrangement Search Strategy for Determining Propositional Satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 155–160.