A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraint Satisfaction Problem

Roberto J. Bayardo Jr. and Daniel P. Miranker

Department of Computer Sciences and Applied Research Laboratories
The University of Texas at Austin (C0500)
Austin, Texas 78712
[bayardo, miranker]@cs.utexas.edu

Abstract

Learning during backtrack search is a space-intensive process that records information (such as additional constraints) in order to avoid redundant work. In this paper, we analyze the effects of polynomial-space-bounded learning on runtime complexity of backtrack search. One space-bounded learning scheme records only those constraints with limited size, and another records arbitrarily large constraints but deletes those that become irrelevant to the portion of the search space being explored. We find that relevance-bounded learning allows better runtime bounds than size-bounded learning on structurally restricted constraint satisfaction problems. Even when restricted to linear space, our relevance-bounded learning algorithm has runtime complexity near that of unrestricted (exponential space-consuming) learning schemes.

Introduction

Tractable subclasses of the finite constraint satisfaction problem can be created by restricting constraint graph structure. The algorithms that exploit this structure most effectively have time and space complexities exponential in the induced width of the constraint graph (Dechter 1992). One such algorithm (Frost & Dechter 1994) applies unrestricted learning during backtrack search. Because the exponential space requirement of unrestricted learning is often impractical, we investigate the effects of space-bounded learning on runtime complexity. There are two distinct methods for limiting the space consumption of learning. One bounds the size of the constraints recorded (Dechter 1990; Frost & Dechter 1994), and the other records constraints of arbitrary size, but deletes those that become irrelevant to the current portion of the search space being explored (Ginsberg 1993; Ginsberg & McAllester 1994).

We define graph parameters that reflect the exponential complexity of backtrack search enhanced with each of these two learning schemes. With respect to bounding runtime, we find that relevance-bounded learning is a better investment of space than size-bounded learning. Further, our linear space relevance-bounded learning algorithm has runtime complexity ($O(\exp(l_1))$ near that of unrestricted learning schemes ($O(\exp(w^*))$). An analysis of expected values for l_1 and w^* on randomly generated graphs reveals that l_1 is usually within a small constant of w^*, and that dedicating more space to learning further reduces the difference in runtime bounds.

Our results are similar to several other structure-exploiting techniques. What distinguishes this work from the cycle-cutset method (Dechter 1990) and graph-splitting techniques (Freuder & Quinn 1985; Bayardo & Miranker 1995) is we do not require positioning variables with few constraints between them first in the ordering in order to effectively bound runtime. We demonstrate that our techniques provide good bounds across a wide variety of graph arrangement policies. Unlike schemes that exploit nonseparable components of the constraint graph (Freuder 1985; Dechter & Pearl 1987), our algorithms are backtrack-driven and require no exponential time preprocessing phases. They thereby preserve good performance on easy instances and allow application of additional proven backtrack enhancement schemes including lookahead (Nadel 1988), conflict-directed backjumping (Prosser 1993), and restricted forms of dynamic variable ordering (Bayardo & Miranker 1995).

Definitions and Problem Statement

A *constraint satisfaction problem* (CSP) consists of a set of *variables*, a finite *value domain*, and a set of *constraints*. The idea is to assign values to variables such that no constraint is violated. More formally, an *assignment* is a mapping of values to some subset of the variables. A *constraint* is a set of assignments (called *nogoods*) that map values to the same set of variables. An assignment is said to *violate* a constraint if any nogood from the constraint is contained within the assignment. A *partial solution* is an assignment which violates no constraint. A *solution* to the CSP is a partial solution mentioning every variable.

The *constraint graph* of a CSP has a vertex for each variable and the property that variables mentioned in a constraint are completely connected. A value v is said to *instantiate* a variable x with respect to a partial solution S if $S \cup \{\langle x, v \rangle\}$ is itself a partial solution. We will denote the number of variables in a CSP with n and the number of values in its value domain with k.

In this paper, we investigate the complexity of determining whether a CSP has a solution. The algorithms we present can be easily extended to return a solution when one exists. While deciding CSP solubility is NP-complete, it is possible to define tractable subclasses by restricting constraint graph structure. For example, some algorithms have runtime exponential in the height h of a depth-first search (DFS) tree of the constraint graph (Dechter 1992). We can define a tractable subclass of CSP by restricting attention to those instances that, after arrangement by some specific DFS procedure, have a DFS tree with h bounded by some constant.

We define constraint graph parameters similar to h for reflecting the exponent in the runtime complexity of our restricted learning algorithms. To preserve generality, we state runtime complexity in terms of how many domain values are considered by the algorithm. Specifically, a value is said to be *considered* whenever the algorithm checks to see whether it instantiates some variable. Runtime can be bounded by multiplying the number of domain values considered with the complexity of verifying an instantiation. Complexity of verifying an instantiation depends on the *arity* of the constraints (how many variables mentioned by its nogoods) as well as implementation specific factors. Typically if the instance is *binary* (all constraints are of arity 2), the complexity of verifying an instantiation is $O(n)$. This is because nogoods which map values to the same set of variables can be grouped into a single compatibility matrix to be tested in constant time.

Rooted-Tree Arrangements

We begin by reviewing the concept of rooted-tree arrangements for improving runtime complexity of backtrack search. We use this result as a starting framework to which various learning schemes will be added and evaluated.

A *rooted tree* is a noncyclic graph whose edges are directed away from the *root* vertex and towards the *leaves.* A *branch* of a rooted tree is a path from the root vertex to some leaf. A *rooted-tree arrangement* of a graph (Gavril 1977)[1] is a rooted tree with the same set of vertices as the original graph and the property that adjacent vertices from the original graph must reside in the same branch of the rooted tree. The concept is illustrated in Figure 1. Directed edges represent a rooted-tree arrangement of the vertices. For illustrative purposes, the original constraint graph edges are displayed as dashed arcs to demonstrate that adjacent vertices appear along the same branch.

Backtrack algorithms can exploit rooted-tree arrangements to improve runtime complexity on some instances. Such an algorithm appears in Figure 2. We refer to the *working assignment* as the set of instantiations made to the current subproblem's ancestors. The algorithm traverses the

1 It is not clear whether pseudo-tree as defined in (Freuder & Quinn 1985) is equivalent to a rooted-tree arrangement or depth-first search (DFS) tree. Nevertheless, a rooted-tree arrangement is a slight generalization of DFS tree (Bayardo & Miranker 1995), and the results from (Freuder & Quinn 1985) apply to both.

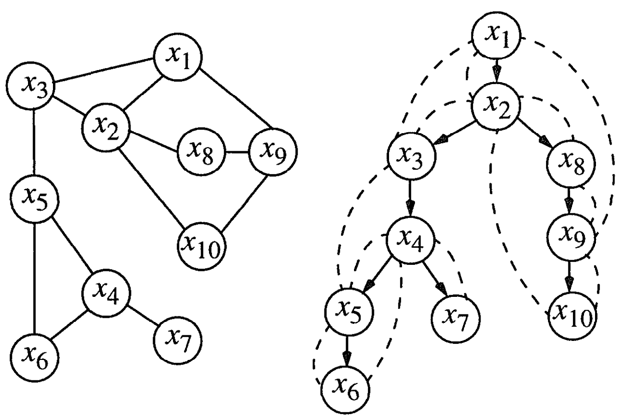


FIGURE 1. A graph and a rooted-tree arrangement of the graph.

rooted-tree arrangement in a depth-first manner as suggested by Dechter (1992) with respect to DFS trees.

```
TREE-SOLVE($P$)
let $x$ denote the root variable of $P$
for each value $v$ instantiating $x$ w.r.t. the working assignment
    for each subproblem $P_c$ corresponding to the subtree
            rooted at a child of $x$
        if TREE-SOLVE($P_c$) = FALSE
            then try another value $v$ (continue outer for loop)
    return TRUE
return FALSE
```

FIGURE 2. An algorithm for backtracking along a rooted-tree arrangement

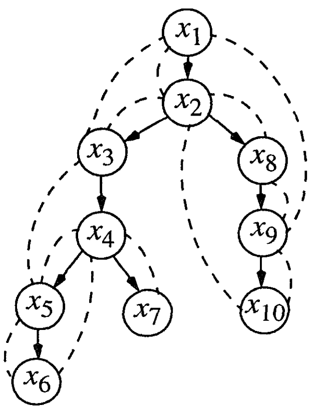
If h denotes the height of the rooted tree, then the total number of values considered by Tree-Solve is $O(nk^h)$ since recursion depth is bounded by h. Correctness of the technique follows from the fact that any subtree of the rooted-tree arrangement corresponds to a *subproblem* whose solubility is determined only by the instantiations of its ancestors -- by definition, there are no constraints between the subproblem variables and those within other branches of the rooted tree.

Unrestricted Learning

Tree-Solve exploits the fact that only the instantiations of a subproblem's ancestors affect its solubility status. However, the constraint graph structure can often be used to further reduce the set of responsible instantiations. For instance, consider variable x_4 from Figure 1. Ancestor x_2 does not connect to any variable in the subproblem rooted at x_4, so the instantiation of x_2 is irrelevant with respect to this subproblem's solubility. More generally, given an ancestor x_a of a subproblem P, if x_a does not connect to any variable within the subproblem P, then the instantiation of x_a is irrelevant with respect to its solubility.

Definition 4.1: The *defining set* of a subproblem is the set of ancestors that are connected to at least one subproblem variable in the constraint graph.

Figure 3 provides the defining sets for the subproblems within the previous example. Subproblems are identified by their root variables. The concept of induced width (Dechter 1992) is equivalent to defining set size, and we elaborate further on the relationship in the appendix. We use this atypical definition since (we feel) it makes the subproblem relationships more explicit, and thereby simplifies the complexity proofs.



$$
\begin{aligned}
ds(x_1) &= \varnothing \\
ds(x_2) &= \{x_1\} \\
ds(x_3) &= \{x_1, x_2\} \\
ds(x_4) &= \{x_3\} \\
ds(x_5) &= \{x_3, x_4\} \\
ds(x_6) &= \{x_4, x_5\} \\
ds(x_7) &= \{x_4\} \\
ds(x_8) &= \{x_1, x_2\} \\
ds(x_9) &= \{x_1, x_2, x_8\} \\
ds(x_{10}) &= \{x_2, x_9\}
\end{aligned}
$$

FIGURE 3. Defining sets of each subproblem.

Suppose Tree-Solve is attempting to solve a subproblem P with defining set X. The working assignment, when restricted to variables within X, is called the *defining set assignment* of P. If P is determined to be unsolvable, we can record its defining set assignment as an additional nogood. Should the assignment resurface, since we have made note of it as a nogood, Tree-Solve(P) can immediately return FALSE instead of attempting the subproblem again. Similarly, if P has been successfully solved given a particular defining set assignment, should the assignment resurface, Tree-Solve(P) can immediately return TRUE. This requires recording the assignment as a *good* (Bayardo & Miranker 1994). Dechter (1990) calls the recording of additional nogoods during backtrack search *learning*. We use this term to apply to the recording of goods as well.

Figure 4 illustrates the described learning extensions. We now consider the effect of these extensions on runtime complexity. First, the size of the largest defining set in the rooted tree will be denoted with the parameter w^*. For example, the problem in Figure 3 has $w^* = 3$. Next, we say the root variable of a subproblem is *visited* whenever the subproblem is attempted. We denote the number of times a variable x is visited by $v(x)$.

Lemma 4.2: Given a subproblem P with root variable x and defining set size s, $v(x) \le k^s$.

Proof: A good or nogood is recorded with each attempt at solving P. There are only k^s unique defining set assignments of P. After k^s visits, every possible defining set assignment is recorded as good or nogood, so the subproblem rooted at x will not be attempted again. □

```
 LEARNING-TREE-SOLVE($P$)
* if the defining set assignment of $P$ is good
*     then return TRUE
* if the defining set assignment of $P$ is nogood
*     then return FALSE
  let $x$ denote the root variable of $P$
  for each value $v$ instantiating $x$ w.r.t. the working assignment
      for each subproblem $P_c$ corresponding to the subtree
              rooted at a child of $x$
          if LEARNING-TREE-SOLVE($P_c$) = FALSE
            then try another value $v$ (continue outer for loop)
*     Record the defining set assignment of $P$ as a good
      return TRUE
* Record the defining set assignment of $P$ as a nogood
  return FALSE
```

FIGURE 4. Tree-Solve extended with unrestricted learning capabilities.

Theorem 4.3: The number of values considered by the unrestricted learning algorithm is $O(nk^{w^*+1})$.

Proof: Each time a variable is visited, at most k values are considered. Recall that the number of variables is n and the largest defining set size is w^*. By lemma 4.2, the algorithm visits at most $O(nk^{w^*})$ variables total, so total values considered is $O(nk^{w^*+1})$. □

Theorem 4.3 is similar to a result in (Frost & Dechter 1994). The difference here is that due to the addition of good recording and a different proof technique, we reduce the bound on domain values considered from exponential in the number of variables to linear. While this improvement may seem minor at this point since we have reduced only the base of the exponent, the algorithmic differences are necessary for the results developed in the following sections.

The space requirement of Learning-Tree-Solve with unrestricted learning is $O(nk^{w^*})$. This is because there are n variables, up to k^{w^*} (no)goods are recorded for each variable, and each (no)good is up to w^* (which we regard as a constant) in size.

Size-Bounded Learning

Since space is often a more precious resource than time, it may be desirable to improve space complexity in exchange for a small runtime penalty. Dechter (1990) suggests recording nogoods of limited size, and defines *i th order [size-bounded] learning* as the scheme in which only those (no)goods of size i or less are recorded, where i is a constant less than w^*. Consider modifying the Learning-Tree-Solve algorithm to perform ith order size-bounded learning. After doing so, learning is only performed at variables with a defining set of size i or less. Its space complexity is thereby $O(nk^i)$. The effect on runtime complexity is as follows:

LEMMA 5.1: Given a variable x whose defining set has size $s \leq i$, $v(x) \leq k^i$.

Proof: See arguments from Lemma 4.2. □

LEMMA 5.2: Given a variable x whose defining set has size $s > i$, $v(x) \leq k^{i+d}$ where d denotes the distance in edges between x and its nearest ancestor where learning is performed.

Proof: Follows from Lemma 5.1 and the fact that $v(x) \leq k \cdot v(x_p)$ where x_p is the parent of x in the tree. □

DEFINITION 5.3: Let d_i denote the maximum of $d + i$ for each variable in the rooted tree.

THEOREM 5.4: The number of values considered by the size-bounded learning algorithm is $O(nk^{d_i+1})$.

Proof: Follows from Lemma 5.2 and the fact that at most k values are considered with each variable visit. □

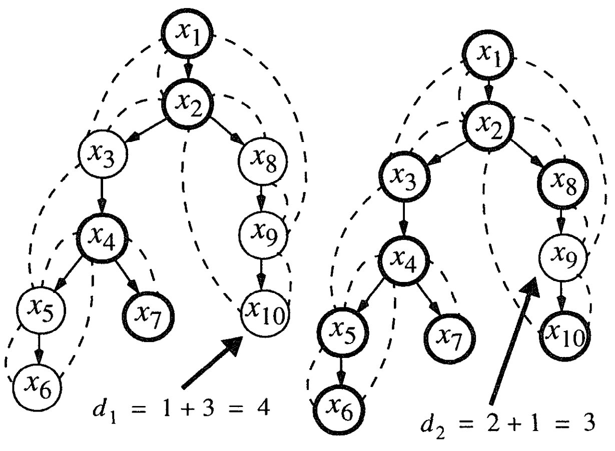


FIGURE 5. Effects of first and second-order size-bounded learning.

Figure 5 illustrates the values of d_1 and d_2 from the example problem. Variables at which learning is performed are highlighted (recall that these are the variables whose defining sets have size i or less). The large arrows point to the variables which have the maximum value of $d + i$. For this instance, second-order size-bounded learning achieves a runtime complexity equivalent to unrestricted learning even though its space requirement is quadratic instead of cubic.

Relevance-Bounded Learning

Another approach for bounding the space consumption of learning is to record (no)goods at every backtrack point, but to also delete (no)goods when they are no longer considered "relevant". For instance, Dynamic Backtracking (Ginsberg 1993) records nogoods of arbitrary size; but, a nogood is deleted once it contains more than one variable-value pair not appearing in the working assignment. We now generalize the notion of relevance to define a class of relevance-bounded learning schemes. A similar generalization appears in (Ginsberg & McAllester 1994).

DEFINITION 6.1: A (no)good is *i-relevant* if it differs from the working assignment in at most i variable-value pairs.

DEFINITION 6.2: An *i th order relevance-bounded learning* scheme maintains only those derived (no)goods that are i-relevant.

Now consider modifying Learning-Tree-Solve to perform i th order relevance-bounded learning. The next theorem establishes that relevance-bounded learning requires an asymptotically equivalent amount of space as size-bounded learning of the same order.

THEOREM 6.3: Space consumption of the relevance-bounded learning algorithm is $O(nk^i)$.

Proof: Consider a variable x with defining set X. If X has size i or less, then the (no)goods recorded at x are always relevant, and occupy $O(k^i)$ space. Suppose now that the size of X is greater than i. For this case, (no)goods recorded at x are sometimes deleted. Let S denote the set of (no)goods recorded at x that exist at some point in time. By definition of relevance and the search order of the algorithm, the (no)goods in S are equivalent when restricted to the uppermost $s - i$ ancestors in X. This implies that S has size at most $|X|k^i = O(k^i)$. Overall space complexity is therefore $O(nk^i)$. □

The effect on runtime complexity of relevance-bounded learning is complicated, so we begin with an example before jumping into formalities. Consider bounding the number of visits to variable x_{10} from the example instance when first-order relevance-bounded learning is being applied. Recall that the defining set of variable x_{10} is $\{x_2, x_9\}$. Note that the subproblem rooted at x_8 cannot affect the instantiation of x_2. Therefore, as long as the algorithm is solving the subproblem rooted at x_8 (call it P_8), all constraints recorded at x_{10} remain 1-relevant. After k visits to variable x_{10} while solving P_8, we therefore have that all possible combinations of x_9 have been recorded as good or nogood. Variable x_{10} will therefore not be visited again until we are done solving P_8. We thus have that $v(x_{10}) \leq k \cdot v(x_8)$. We next generalize this idea to apply to any instance and any order i of relevance-bounded learning.

LEMMA 6.4: Given a variable x whose defining set has size $s \leq i$, $v(x) \leq k^i$.

Proof: Since (no)goods of size i or less are always i-relevant, simply apply the arguments from Lemma 4.2. □

Given a vertex x with defining set X, let $A(x, j)$ denote the member of X that is the jth nearest to x along the branch from x to the root. For instance, from our example problem, $A(x_{10}, 1) = x_9$, and $A(x_{10}, 2) = x_2$. Further, let $C(y, x)$ denote the child of y that resides on the same branch as x. Thus, $C(x_2, x_{10}) = x_8$.

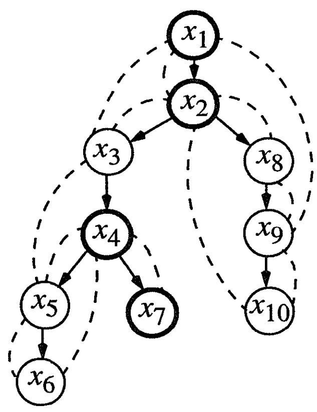
Lemma 6.5: Given a vertex x whose defining set is bigger than i, $v(x) \le k^i \cdot v(C(A(x, i+1), x))$.

Proof: As long as the instantiation of variable $A(x, i+1)$ and the instantiations of its ancestors remain unchanged, by definition, the (no)goods recorded at x are i-relevant. Consider then the subproblem P rooted at $C(A(x, i+1), x)$. Solving this subproblem does not affect the instantiations of $A(x, i+1)$ and its ancestors, so all (no)goods recorded at x while attempting P remain i-relevant until we are done attempting P. The number of visits to variable x while attempting P is bounded by k^i since after k^i visits, all possible instantiations of the defining set members within subproblem P are recorded as good or nogood. The claim follows immediately from this fact. □

Definition 6.6: Let l_i denote the largest exponent from each bound derived from applying lemmas 6.4 and 6.5 to every variable in the problem.

Theorem 6.7: The number of values considered by the relevance-bounded learning algorithm is $O(nk^{l_i+1})$.

Proof: Clear. □



$$\begin{aligned}
v(x_1) &= 1\\
v(x_2) &\le k\\
v(x_3) &\le k \cdot v(x_2) \le k^2\\
v(x_4) &\le k\\
v(x_5) &\le k \cdot v(x_4) \le k^2\\
v(x_6) &\le k \cdot v(x_5) \le k^3\\
v(x_7) &\le k\\
v(x_8) &\le k \cdot v(x_2) \le k^2\\
v(x_9) &\le k \cdot v(x_8) \le k^3\\
v(x_{10}) &\le k \cdot v(x_8) \le k^3
\end{aligned}$$

$$l_1 = 3$$

FIGURE 6. Effects of first-order relevance-bounded learning.

Figure 6 illustrates the value of l_1 for our example problem. Since k^3 bounds the number of visits to any variable, $l_1 = 3$. First-order relevance-bounded learning thereby achieves runtime complexity equivalent to unlimited learning on this instance, while using only $O(nk)$ space.

Comparison of l_i, d_i, h and w^*

Maximum defining set size is a lower-bound for l_i and d_i for any i. It is important to know how close these parameters come to w^* in order to determine whether polynomial space restrictions provide a worthwhile trade-off. This section provides such an evaluation, concluding that l_i provides a close approximation of w^* even when $i = 1$. We also show that usually l_1 is less than d_3, implying relevance-bounded learning provides a better investment of space than does size-bounded learning.

Minimizing w^* by cleverly arranging the variables is NP-hard (Arnborg 1985). It is likely that finding the rooted-tree arrangement that minimizes either h, d_i, or l_i for any i is also NP-hard due to its close relation to this and other NP-hard problems. As a result, a heuristic is often used to arrange the variables of a graph. Then, a structure-based technique is applied with respect to that arrangement. In this section, we evaluate the expected values of the various parameters when using this approach.

Figure 7 plots the expected values of the graph parameters h, d_1, d_3, l_1, l_3 and w^*. The rooted-tree arrangement policy we used was to apply depth-first search to find an arbitrary DFS tree. Each point on the plot represents the average parameter value from 100 connected graphs with 100 vertices. Graphs were generated randomly with the number of edges specified by the horizontal axis. To ensure connectivity, a random spanning tree was greedily generated from a random starting point, and then the remaining edges added in. An additive factor of 1 is applied to each parameter other than h so that it reflects the actual exponent in its algorithm's runtime complexity function.

The figure reveals that relevance-bounded learning is more effective at bounding runtime than size-bounded learning. In fact, we usually have that $l_1 < d_3$. We also see that relevance-bounded learning closely approximates the effects of unrestricted learning: both l_1 and l_3 are within a small constant of w^* throughout the plotted range where on average, $l_3 \le w^* + 5$ and $l_1 \le w^* + 6$. The fewer the edges, the closer the expected values of l_i and w^*. For instance, when the number of edges is 119 or less, $l_1 \le w^* + 3$.

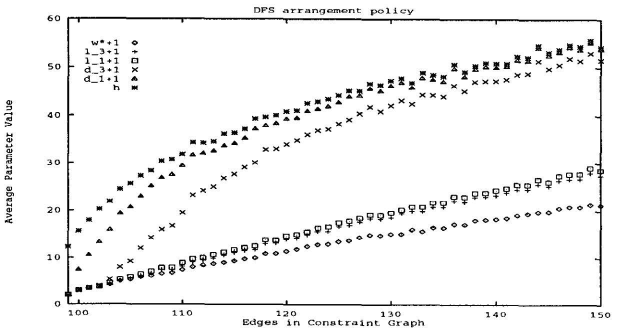


FIGURE 7. Expected parameter values: DFS arrangement

We repeated these experiments with other arrangement policies, and the results appear in Figure 8. Note that now d_3 is sometimes better than l_1, but only for the sparsest graphs. For the maximum degree arrangement policy, all parameters (even h) are reasonably close to induced width. This policy positions several variables with few constraints between them first in the arrangement since it greedily selects the remaining variable with the most edges in the constraint graph. Over all arrangement policies investigated, we see that on average, $l_3 \leq l_1 \leq 2w^*$. Whatever the arrangement heuristic applied, relevance-bounded learning seems to always produce a runtime bound near that of unrestricted learning. The more space dedicated to learning, the more closely the effect of unrestricted learning is approximated. Size-bounded learning, on the other hand, approximates unrestricted learning only for the sparsest graphs. After a certain point, size-bounded learning fails to perform much (if any) learning since the defining sets become too large.

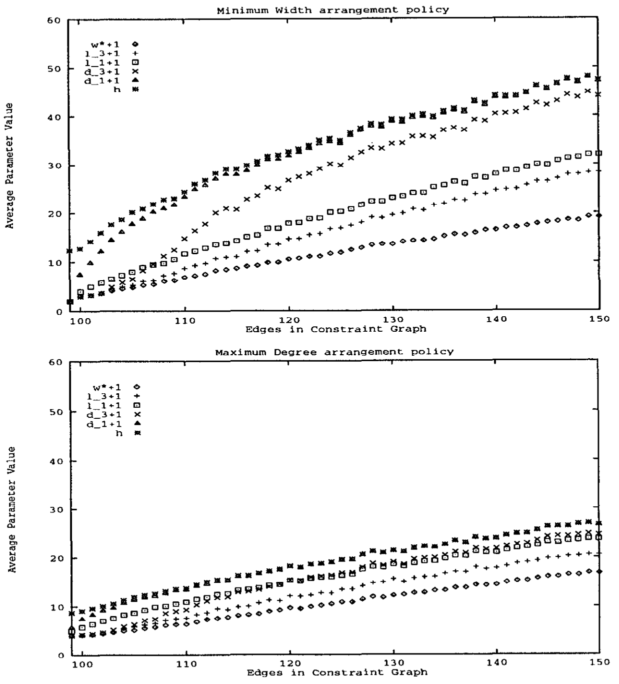


FIGURE 8. Expected parameter values: minimum width and maximum degree arrangements.

It is possible to relate l_i and d_i to other parameters from the literature. For instance, it is easy to define an arrangement policy such that given a cycle-cutset of size c, $l_1 \leq c + 1$. Similarly, given a graph with nonseparable component size of r, we can define an arrangement policy such that $l_1 \leq d_1 \leq r$. Due to length restrictions, we leave the details of these policies as exercises to the reader.

Conclusions and Future Work

We have presented and theoretically evaluated backtrack-based algorithms for tractable constraint satisfaction on structurally-restricted instances. The runtime complexity of backtrack search enhanced with unrestricted learning is known to be exponential in w^*. We defined similar graph parameters for backtrack enhanced with ith order size-bounded learning (d_i) and ith order relevance-bounded learning (l_i). An evaluation of expected values of these parameters reveals that l_i is within a small constant of w^* for sparse instances, and within a factor of 2 for all cases explored. Further, l_1 (linear-space consuming relevance-bounded learning) is often much less than d_3 (cubic space-consuming size-bounded learning). From this we conclude that relevance-bounded learning is a better investment of space resources than size-bounded learning, and that our low-order relevance bounded learning algorithm provides a space-efficient alternative to unrestricted learning in applications requiring tractable constraint satisfaction on structurally-restricted instances. One potential application is real-time multi-join query evaluation since multi-join queries have a graph structure that is typically assumed to be tree or nearly-tree structured (Swami 1989). Heuristic algorithms for the CSP and related problems are prone to pathological behavior (Gent & Walsh 1996), and are thereby inappropriate for real-time domains.

We suspect the theoretical differences between the various learning schemes will hold with respect to average-case performance on denser instances, and leave the issue open to future study. The algorithms from this paper have been given the minimal functionality needed to achieve the stated bounds. However, since they are all variations of standard backtrack search, they remain open to additional backtrack optimizations known to improve average-case performance. For instance, the algorithms use statically-determined culprits for every possible failure. Techniques such as conflict-directed backjumping (Prosser 1993) apply runtime derived information to potentially minimize the set of culprits even further. Dynamic ordering of the variables is another powerful technique for improving average-case performance (Haralick & Elliot 1980). Structure-based techniques impose some restrictions on the variable arrangements, but rooted-tree search procedures are open to a limited form of search rearrangement. The idea is at any point we can attempt the open subproblems in any order (Bayardo & Miranker 1995). Finally, lookahead techniques such as forward checking (Nadel 1988) can help realize failures earlier in the search. Integrating lookahead with our algorithms is straightforward because the defining sets, since they are derived from structure alone, identify a set of culprits that is not affected by the particular mechanism used to identify failure within a given subproblem.

Acknowledgments

This research was supported in part by an AT&T Bell Laboratories Ph.D. fellowship.

Appendix

Here we demonstrate the equivalence of maximum defining set size and induced width. The parameters are different only in that one is defined with respect to a rooted-tree arrangement of the graph, and the other with respect to an ordering of the graph vertices. We demonstrate equivalence by showing (1) a graph with a rooted-tree arrangement can be ordered so that induced width is equivalent to maximum defining set size, and (2) given an ordered graph, we can obtain a rooted-tree arrangement whose maximum defining set size is equal to induced width of the ordered graph.

Induced width is a property of a graph with an ordering imposed on its vertices. A *child* of a vertex v in an ordered graph is a vertex that is adjacent to v and follows v in the ordering. The *induced graph* of an ordered graph G is an ordered graph with the same ordered set of vertices as G and the smallest set of edges to contain the edges of G and enforce the property that any two vertices sharing a child are adjacent. We can build the induced graph of G by iteratively connecting any nonadjacent vertices that share a child. The *width* of a vertex in an ordered graph is the number of parents it has in the graph. The induced width of the ordered graph (w^*) is the maximum of the widths from each vertex in the induced graph.

We can obtain a rooted-tree arrangement T of a graph G from its induced graph G' as follows: Let the first vertex along the ordering be the root of T. A vertex x_b is a child of another vertex x_a if and only if x_b is the first node to follow x_a in the ordering, and have the property that x_a and x_b are adjacent in G' (the induced graph).

Figure 9 illustrates the process. To create the induced graph, we first note that in the original graph, vertices 1 and 4 share the child 5, therefore an induced edge is added between them. After adding this edge, we now have that vertices 1 and 2 share the child 4, so edge (1,2) is added. This completes the process of forming the induced graph. The rooted-tree arrangement of the ordered graph is displayed to the right of the induced graph. To form the tree, we simply keep those edges from the induced graph that connect a vertex its nearest child.

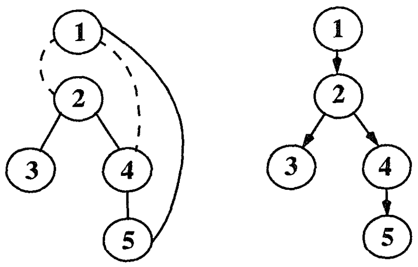


FIGURE 9. An ordered graph, the edges (dashed) added to form the induced graph, and its backtrack tree.

A property of the rooted-tree arrangement provided by the procedure is that the defining set of any subproblem is equivalent to its set of parents in the induced graph. We have therefore described a process for obtaining a rooted-tree arrangement whose largest defining set size is equivalent to induced width. We can similarly obtain an ordered graph from a rooted tree arrangement whose induced width is equivalent to the largest defining set size: simply order the nodes of the graph according to a depth-first traversal of the rooted tree.

References

Arnborg, S. 1985. Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposability--A Survey. *BIT* 25:2-23.

Bayardo, R. J. and Miranker, D. P. 1994. An Optimal Backtrack Algorithm for Tree-Structured Constraint Satisfaction Problems. *Artificial Intelligence* 71(1):159-181.

Bayardo, R. J. and Miranker, D. P. 1995. On the Space-Time Trade-Off in Solving Constraint Satisfaction Problems. In Proc. 14th Intl. Joint Conf. on Artificial Intelligence, 558-562.

Dechter, R. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence* 41(3):273-312.

Dechter, R. 1992. Constraint Networks. In *Encyclopedia of Artificial Intelligence,* 2nd ed., 276-285. New York, NY: Wiley.

Dechter, R. and Pearl, J. 1987. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence* 34:1-38.

Freuder, E. C. 1985. A Sufficient Condition for Backtrack-Bounded Search. *J. Association for Computing Machinery* 32(4):755-761.

Freuder, E. C. and Quinn, M. J. 1985. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In Proc. Ninth Intl. Joint Conf. on Artificial Intelligence, 1076-1078.

Frost, D. and Dechter, R. 1994. Dead-End Driven Learning. In Proc. Twelfth National Conf. on Artificial Intelligence, 294-300.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY: W. H. Freeman and Co..

Gavril, F. 1977. Some NP-complete Problems on Graphs, Proc. 11th Conf. on Information Sciences and Systems, 91-95. Cited through (Garey & Johnson 1979, pg. 201).

Gent, I. P. and Walsh, T. 1996. The Satisfiability Constraint Gap. *Artificial Intelligence* 81.

Ginsberg, M. L. 1993. Dynamic Backtracking. *J. Artificial Intelligence Research* 1:25-46.

Ginsberg, M. L. and McAllester, D. A. 1994. GSAT and Dynamic Backtracking. In Proc. Fourth International Conf. on Principles of Knowledge Representation and Reasoning, 226-237.

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

Nadel, B. 1988. Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms. In *Search in Artificial Intelligence,* 287-342. New York, NY: Springer-Verlag.

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

Swami, A. 1989. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. In Proc. 1989 ACM SIGMOD Intl. Conf. on Management of Data, 367-376.