


Bibliography


Appendix

Here we show that given an ordered CSP with induced width \( w^* \), we can construct another ordering of the CSP such that pseudo-tree height and reset distance are within a logarithmic factor of \( w^* \). The proof idea is to apply a splitting technique from [Freuder & Quinn, 85] to a k-tree embedding of the induced graph, using a construction from [Freuder, 90] demonstrating that graphs with induced width \( k \) are partial graphs of \( k \)-trees. The splitting technique is used to create a pseudo-tree with height equal to the maximum recursion depth of the splitting procedure.

A graph is a k-tree [Beineke & Pippert, 71] if: (1) the graph has \( k \) vertices and is complete, or (2) there is a vertex of degree \( k \) whose neighborhood induces a complete graph, and the graph obtained by removing the vertex is a \( k \)-tree. For example, an noncyclic graph is a 1-tree.

**Lemma A.1:** A connected, ordered graph of induced width \( w^* \) can be broken into two or more components of size at most \( \left\lceil \frac{n-w^*}{2} \right\rceil \) by removing \( w^* + 1 \) vertices from the graph, where size is the number of vertices.

**Proof:** Freuder [1990] provides a construction demonstrating that graphs with induced width \( k \) are partial graphs of \( k \)-trees. Now, note that a removal of any \( k+1 \) clique from a \( k \)-tree disconnects the \( k \)-tree just as the removal of a single-vertex disconnects a noncyclic graph. It is easy to show that a noncyclic graph can be broken into components of size at most \( \left\lceil \frac{n-1}{2} \right\rceil \) by removing a single vertex. A straightforward generalization reveals that a \( k \)-tree can be broken into components of size at most \( \left\lceil \frac{n-k}{2} \right\rceil \) by removing \( k+1 \) vertices. The claim follows immediately from these facts. □

**Theorem A.2:** For an ordered binary CSP of \( n \) variables with induced width \( w^* \), there exists an ordering of the same problem with \( h = O(w^* \log \frac{n}{w^*}) \).

**Proof:** By lemma 4.1, given a problem with induced width \( w^* \), we can find \( w^* + 1 \) vertices from its constraint graph whose removal split the graph into components of size at most \( \left\lceil \frac{n-w^*}{2} \right\rceil \). Because the resulting components must have induced width of \( w^* \) or less, we can then recursively apply this process on the components until no component has more than \( w^* \) vertices. Recursion depth of this procedure is \( O(\log \frac{n}{w^*}) \). Because at most \( w^* + 1 \) vertices are removed at each split, this process can be used to build a pseudo-tree of height \( h_o = O(w^* \log \frac{n}{w^*}) \) (see [Freuder & Quinn, 85] for more detail). The theorem thus follows from the procedure described previously for constructing an ordering from a pseudo-tree with a backtrack tree of equivalent height. □

**Corollary A.3:** For an ordered binary problem of \( n \) variables with induced width \( w^* \), there exists an ordering of the same problem with \( d = O(w^* \log \frac{n}{w^*}) \).
tively on problems of bounded reset distance, but they also perform well on average. While FC+DSR has the best median performance on problems from suite 2, the probability of failure is relatively high and its mean performance poor on both problem suites. The variable order restrictions imposed by jump-ahead backtrack do not seem to be a problem as long as the reset distance of the problem is sufficiently bounded.

9. Conclusions

We have evaluated the implications of purely structure-based techniques for informed backtracking and linear-space-bounded learning. A new graph parameter, reset distance, has been shown to reflect the exponent in the runtime complexity function of a new CSP algorithm, jump-ahead backtrack. Reset distance can be made to match and often improve upon other parameters including cycle-cutset size, pseudo-tree height, and maximum nonseparable component size. While reset distance does not improve on induced width, for a common effective static variable ordering policy, it comes closer on average than the others. By applying a technique to make reset distance match or improve upon maximum nonseparable component size, we have duplicated the previous runtime complexity results based on maximum nonseparable component size, but without exponential space or preprocessing requirements.

For two suites of randomly generated problems with bounded reset distance, jump-ahead backtrack and its forward-checking-enhanced variant performed well on average and succeeded in solving all instances. A commonly employed technique, forward checking with dynamic search rearrangement, failed to solve a substantial percentage of the same instances. Work still remains to determine the natural classes of problems on which jump-ahead backtrack is most effective. Certainly, given equivalent variable and value orderings, it performs no more constraint tests than chronological backtrack on any problem instance. Furthermore, since most information required by the enhancements applied by jump-ahead backtrack can be precomputed from the induced graph, and since no operations are added to the inner-loop which finds an instantiation for the current variable, its overhead beyond that of chronological backtrack is small. We lastly note that the technique is compatible with most look-ahead schemes such as forward checking and certain schemes which further improve the quality of the conflict sets identified by the algorithm.

Whether or not a polynomial space algorithm can achieve a runtime bound exponential in the induced width of the ordering remains an open question. Current approaches fundamentally require exponential space since they record high-arity constraints or generate all solutions to certain subproblems. We therefore suspect that if it is indeed possible, it will be through some new technique that is radically different from those discussed here and in previous work.
One thousand problem instances with reset-distance of seven after minimum-width ordering were taken to form each suite, and each instance was solved by three algorithms: jump-ahead backtrack (JABT), jump-ahead backtrack with a forward checking lookahead phase (JABT+FC), and chronological backtrack with forward checking and dynamic search rearrangement (FC+DSR). Forward checking [Haralick & Elliot, 80] is a polynomial time-bounded lookahead phase that is invoked after each variable instantiation to filter out incompatible values from the domains of future variables. Dynamic search rearrangement [Haralick & Elliot, 80] chooses the variable with the fewest remaining values (after forward checking has performed its filtering) to be instantiated next. Both these techniques have been shown to greatly improve backtrack search in a variety of settings. The dynamic search rearrangement algorithm ignores the minimum width ordering already imposed on the variables except when choosing the variable to instantiate first. While forward checking is compatible with the techniques for bounding worst-case performance used by jump-ahead backtrack, dynamic search rearrangement is not since jump-ahead backtrack requires a fixed variable ordering.

Our performance measure was chosen to be the number of constraint tests performed in order to eliminate implementation specific factors. We argue that this measure is appropriate because each algorithm performs a relatively small amount of work that does not primarily involve constraint testing. For our implementations, we found the number of constraint tests per second to be roughly equivalent across all algorithms, with jump-ahead backtrack performing the most per second, and forward checking with dynamic search rearrangement performing the least. Each algorithm was implemented to stop and report “failure” in solving a particular instance if over 100,000,000 constraint tests were performed. Mean and median values in the data reflect performance of the algorithm on the instances which did not produce a failure.

**FIGURE 10. Empirical Results for Problem Suite 1 -- 52.8% Solvable**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>% Failure</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC+DSR</td>
<td>11%</td>
<td>3839554</td>
<td>15594</td>
</tr>
<tr>
<td>JABT+FC</td>
<td>0.0%</td>
<td>14348</td>
<td>5330</td>
</tr>
<tr>
<td>JABT</td>
<td>0.0%</td>
<td>18012</td>
<td>6216</td>
</tr>
</tbody>
</table>

**FIGURE 11. Empirical Results for Problem Suite 2 -- 97.4% Solvable**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>% Failure</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC+DSR</td>
<td>6.4%</td>
<td>1467128</td>
<td>1104</td>
</tr>
<tr>
<td>JABT+FC</td>
<td>0.0%</td>
<td>8773</td>
<td>1774</td>
</tr>
<tr>
<td>JABT</td>
<td>0.0%</td>
<td>11844</td>
<td>1909</td>
</tr>
</tbody>
</table>

The data in Figure 10 and Figure 11 suggests that jump-ahead backtrack and its forward-checking enhanced variant not only bound worst-case performance quite effec-
For the randomly generated graphs, induced width was lowest on average, followed by reset distance, cycle-cutset size, pseudo-tree height, and maximum nonseparable component size. Even without any of the reordering techniques discussed in the theorems above, reset distance tends to better the other parameters except for induced width. While induced width is always less than or equal to reset distance given any ordering policy, reset distance appears to come close on average. A more intelligent variable ordering scheme that heuristically attempts to minimize reset distance would likely reduce the gap between the induced width and reset distance plots.

8. Empirical Evaluation

To empirically evaluate the algorithms, we randomly generated two suites of test problems with sixty variables, sixty-eight binary constraints, and ten domain values per variable. The first suite of instances were generated such that a pair of values were allowed by a particular constraint with probability 0.225. This value was chosen so that approximately fifty percent of the instances were solvable. Problems falling in this range of solvability have been found by [Cheeseman et al., 91] and [Mitchell et al., 92] to be harder, on average, than those in other solvability ranges. For the second suite of problems, the probability of compatibility was set to 0.26 in order to produce instances that are almost always solvable. [Smith, 94] has noted that exceptionally difficult problems sometimes arise in this region.
Minimizing induced width by cleverly ordering the variables and finding a minimum size cycle-cutset are well-known NP-hard problems. It is likely that minimizing pseudo-tree height and reset distance are also NP-hard due to their close relation to the NP-hard problem of minimizing the height of a DFS tree for a given graph. As a result, typically a heuristic is used to order the variables of a graph. Then, a structure-based technique is applied with respect to that ordering. Induced width and reset distance of a particular ordered graph are well defined. We can define the cycle-cutset size of an ordered graph as the number of vertices removed along the ordering before breaking all cycles. Since a backtrack tree is a pseudo-tree, we define the pseudo-tree height for a particular ordered graph as the height of its backtrack tree. Computing the maximum size of a non-separable component can be done by a variant of depth-first search [Even, 79]. The value of this parameter is independent of the ordering.

The plot in Figure 9 demonstrates the average value taken on by the various parameters discussed so far for randomly generated graphs. The minimum-width variable ordering policy [Freuder, 82] was used to order the variables of the graph. Dechter has demonstrated that this policy is among the most effective static variable ordering policies of those analyzed with respect to average-case performance of various CSP algorithms [Dechter & Meiri, 92]. Each point on the plot represents the average parameter value from 1000 connected graphs with sixty 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 is applied to each parameter so that it reflects the actual exponent in its algorithm’s worst case complexity function. We therefore plot the average values taken by $h, c + 2, w^* + 1, r$ and $d + 2$. 


is a graph with a vertex for each nonseparable component in $G$, and two vertices are adjacent if and only if the components they represent share a vertex. Any component tree is a noncyclic graph [Even, 79]. Freuder’s and Dechter’s algorithms achieve a runtime bound exponential in $r$, where $r$ is the size of the largest nonseparable component of the constraint graph. However, they also require space and a preprocessing phase exponential in $r$.

**Theorem 7.3:** Given a CSP with a maximum nonseparable component size of $r$, an ordering of the CSP exists with $d < r - 1$.

**Proof:** Consider the ordering policy defined by the order in which the following procedure visits each vertex of the constraint graph: Perform a DFS traversal of the component tree of the constraint graph. Before going to the next vertex in the component tree, “visit” each unvisited constraint graph vertex within the current component. The ordering defined by the order in which this procedure visits each constraint graph vertex has a reset distance less than $r - 1$ because if we look at the reset graph, no reset edges can connect a vertex $x$ to another vertex belonging to a nonseparable component that doesn’t contain $x$. 

The proof of the above theorem provides the necessary variable ordering technique for using jump-ahead backtrack to duplicate the runtime complexity of previous algorithms exploiting nonseparable components while improving space complexity from exponential in $r$ to $O(n^2 + nk)$. Further, it eliminates the need for an exponential time preprocessing phase.

The *induced width* of an ordered graph ($w^*$) is the maximum number of parents to any vertex in the induced graph (alternatively, $w^*$ is the size of the largest static conflict set). Dechter [Dechter, 92] has conjectured that this is an intrinsic measure of a problem’s difficulty since it is a lowerbound for many other structure based parameters when considering the additive factors. Reset distance is no exception, since it is easy to see $w^* \leq d + 1$ for any particular ordering of a constraint graph. The appendix shows, however, that pseudo-tree height and reset distance can always be made within a logarithmic factor of induced width by a reordering technique.

Adaptive consistency [Dechter & Pearl, 88], tree-clustering [Dechter & Pearl, 89], and other CSP algorithms have runtime complexity exponential in $w^*$. While these techniques are fundamentally more effective in bounding worst-case runtime than jump-ahead backtrack, they have a space complexity that is also exponential in $w^*$, compared to the quadratic space requirement of jump-ahead backtrack. They are furthermore unlikely to perform well in practice, not only due to their space requirements, but also due to the fact that they induce additional constraints which need to be tested and/or require exponential-time preprocessing phases.
induced width, while more effective than jump-ahead backtrack in bounding worst-case performance, require exponential space. Previous algorithms with a worst-case complexity function exponential in maximum non-separable component size [Freuder, 85; Dechter, 87] also have exponential space complexity. A contribution of this section is therefore a technique for achieving this complexity result while only consuming polynomial space. Because minimizing most of these parameters by choosing the best possible variable ordering is often NP-hard, we lastly compare the expected values of these parameters given a common variable ordering heuristic.

The cycle-cutset method [Dechter, 90] is a technique that involves finding a set of vertices in the constraint graph (called a cutset) whose removal from the graph breaks every cycle. A backtrack algorithm using the technique has a runtime that is exponential in \( c + 2 \), where \( c \) is the size of the cutset. Space required by this technique is polynomially bounded.

**Theorem 7.1:** Given a CSP and a cutset of size \( c \), an ordering of the CSP exists with \( d \leq c \).

**Proof:** Consider the variable ordering where the cutset variables are positioned first, and the remaining variables are ordered by DFS traversing the remaining nodes in the constraint graph. The claim follows from the fact that all reset edges in the reset graph of this ordered CSP must connect to at least one cutset vertex. \( \square \)

Another polynomially space-bounded technique is pseudo-tree search [Freuder & Quinn, 85]. This algorithm involves backtracking along a tree arrangement of the constraint graph vertices called a pseudo tree. A pseudo tree has the property that vertices in different branches of the tree are nonadjacent in the constraint graph.\(^*\) The technique is exponential in the parameter \( h \), where \( h \) is the height of this tree.

**Theorem 7.2:** Given a CSP and a pseudo-tree arrangement of its variables with height \( h \), an ordering of the CSP exists with \( d < h - 1 \).

**Proof:** Consider the variable ordering where we order the variables according to a DFS traversal of the pseudo-tree. We are left with an ordered graph whose backtrack tree is of height \( h \) or less. The theorem follows from the fact that reset distance must be less than the height of the backtrack tree minus one. \( \square \)

Now we consider a graph decomposition technique exploited in [Freuder, 85] and [Dechter, 87]. A *separation vertex* of a connected graph \( \langle V, E \rangle \) is a vertex whose removal induces a disconnected graph, and a *nonseparable component* of a graph \( G \) is a maximal connected subgraph of \( G \) with no separation vertices. The *component tree* for a graph \( G \)

\(^*\) Note that a depth-first search tree (whose properties with respect to CSP are mentioned in [Dechter 90] and [Gashnig 79]) or a backtrack tree is a pseudo tree.
that subtree since $v$ will be permanently marked as “good” w.r.t. $x$. It is therefore impossible for more than one reset of $I$ to take place for each value in the domain of $x_p$.

**INDUCTION:** Let us now assume the claim holds for any variable with reset distance $i - 1$. Consider a variable $x$ with reset distance $i$. While some value instantiates $x_r$, the remaining variables form a subproblem where $x$ has reset distance 0. We can therefore apply the logic from the base case to conclude that each value in the domain of $x$ can be considered at most $k$ times while the same value instantiates $x_r$.

Now, consider the variable $x_a$ that is, in the backtrack tree, both an ancestor of $x$ and a descendent of $x_r$. By definition, $x_a$ has reset distance $i - 1$. We can apply the inductive hypothesis to conclude each value in the domain of $x_a$ is considered at most $k^i$ times. It follows that $x_r$ can be instantiated at most $k^i$ times. The number of times a value in the domain of a variable with reset distance $i$ can be considered is therefore $k^{i+1}$. □

**Theorem 6.1:** Jump-ahead backtrack runs in time $O(dnk^{d+2})$ where $d$ is the largest reset distance of any variable and $n$ is the number of variables.

**Proof:** It is sufficient to bound the number of constraint tests performed by jump-ahead backtrack in order to bound runtime spent in the main body of the algorithm. Each variable $x$ has at most $d_x + 1$ parents in the ordered constraint graph, where $d_x$ is its reset distance. This implies that $O(d)$ constraint tests are performed for each value considered by the algorithm. By lemma 6.1, the total number of domain values considered is $O(nk \cdot k^{d+1}) = O(nk^{d+2})$. Time spent in the main body is therefore $O(dnk^{d+2})$.

The time spent in the ADVANCE, RETREAT, and MAKE SOLUTION procedures can be bounded similarly to complete the proof. □

**Theorem 6.2:** Jump-ahead backtrack requires $O(n^2 + nk)$ space beyond the problem input.

**Proof:** The $O(n^2)$ term arises from space required to construct the induced graph. The $O(nk)$ term arises from the “good” markings since there are a total of $n-1$ backtrack children, each capable of marking up to $k$ values at any one point in time. Other space consumers such as the iterators can be stored in $O(n)$ space. □

**7. Complexity Comparison with Other Work**

Let the reset distance of an ordered CSP (denoted $d$) be the largest reset distance of any variable in the problem. Ignoring the additive factor of 2, $d$ reflects the exponent in the runtime complexity function of jump-ahead backtrack. In this section, we compare $d$ to several other parameters in the literature that reflect the exponents in the complexity functions of other CSP algorithms: cycle-cutset size, pseudo-tree height, maximum non-separable component size, and induced width. We first show that $d$ can be made less than or equal to each of these other parameters except for induced width, even when additive factors are considered. Algorithms with complexity functions that are exponential in
edge connecting $x$ to its reset parent $x_r$, then traverses the backtrack edge connecting $x_r$ to the backtrack child $x_c$ of $x_r$ that is also an ancestor of $x$ in the backtrack tree, and lastly traverses the edges in the reset path of $x_c$. Vertices without reset parents have zero-length reset paths. Let the reset distance of a vertex be the number of reset edges in its reset path. Figure 8 illustrates these definitions. Dashed edges in the reset graph are the reset edges. Arrows represent the reset path for the highlighted vertex.

**FIGURE 8. Example of Reset Graph and Reset Distance**

![Diagram showing ordered graph, backtrack tree, reset graph, and reset distances]

**Lemma 6.1:** Jump-ahead backtrack considers each value in the domain of a variable with reset distance $d$ at most $k^{d+1}$ times, where $k$ bounds the size of any domain.

**Proof: (by induction on reset distance)**

BASE: For the initial variable in the ordering (which has reset distance 0), each of its values are considered at most once since its iterator is reset exactly one time. Any other variable $x$ with reset distance 0 must have no reset parent by definition. We show that the iterator $I$ for such a variable $x$ can be reset at most once for each value in the domain of the parent $x_p$ of $x$. The base case of the claim immediately follows from this fact since there are at most $k$ values in the domain of $x_p$, and each iterator reset allows each value to be considered at most once:

Consider a particular value $v$ to instantiate $x_p$. While $v$ instantiates $x_p$, there are two possible scenarios. First, the algorithm may eventually backtrack from $x$ to $x_p$, at which point $v$ will be permanently removed from the problem. Second, the algorithm may advance beyond all variables in the subtree of the backtrack tree rooted at $x$. For this case, subsequent instantiations of $x_p$ with $v$ will cause ADVANCE to skip over the variables in
FIGURE 7. Jump-Ahead Backtrack

RETREAT()
1 let $x_j$ denote the backtrack parent of $x_i$
2 move the value instantiating $x_j$ into the elimination list of $x_i$ and remove its “good” marking made
   with respect to variable $x_i$
3 $i \leftarrow j$

ADVANCE()
1 for each reset child $x_c$ of $x_j$ do
2 empty the elimination list of $x_c$ into the domain of its backtrack parent $x_p$
3 remove from the values of $x_p$ any “good” markings made with respect to variable $x_c$
4 $i \leftarrow i + 1$
5 if the value $v$ instantiating $x_j$ is marked “good” with respect to variable $x_i$
6 then let $x_s$ denote the skip point of $x_i$
7 $i \leftarrow s$ ; jump ahead in the variable order
8 goto 5
9 else mark $v$ as “good” with respect to variable $x_i$

MAKESOLUTION()
1 recursively run jump-ahead backtrack on the skipped subtrees
2 return the set of current instantiations

The jump-ahead backtrack algorithm (Figure 7) assumes the variables have been
DFS reordered if necessary. Each value in the domain of a variable $x$ is provided with a
slot for each backtrack child of $x$. This slot is used to insert the “good” markings men-
tioned in the pseudo-code. These good markings could take the form of a timestamp
which is to be compared against another timestamp stored by the variable performing the
marking. This way, when it is time to remove the good markings (line 9 of ADVANCE()), a
simple $O(1)$ timestamp increment is sufficient to invalidate the markings instead of scan-
ning the entire variable domain.

A domain value $v$ is marked as good with respect to some variable $x$ immediately
when $x$ is made current. This marking persists until either a backtrack takes place from $x$
to its backtrack parent before control advances beyond all of $x$’s descendents in the back-
track tree, or the resettor of $x$ is revisited. Since variables may have been skipped (and
therefore left uninstantiated) when control reaches the final variable in the ordering,
MAKESOLUTION() must go back and instantiate them. This is done by recursively running
jump-ahead backtrack on the skipped sub-trees of the backtrack tree.

6. Complexity Analysis

We now define a new ordered graph parameter that dictates the exponential com-
plexity of jump-ahead backtrack. Let the reset graph be the graph formed by taking the
backtrack tree and adding a reset edge between each vertex and its reset parent. The non-
reset edges will be referred to as backtrack edges. For a particular vertex $x$, we recur-
sively define its reset path as the path in the reset graph that initially traverses the reset
Figure 6. Example of DFS Reordering

The algorithm in Figure 7 exploits the notion of learning from success by marking domain values as “good” with respect to particular variables. When the backtrack parent of a variable \( x \) is instantiated with a value marked as good with respect to variable \( x \), then the algorithm knows a priori that variables in the subtree of the backtrack tree rooted at variable \( x \) can be successfully instantiated. This determination can be made by the algorithm because the good marking implies it has already successfully solved that subproblem in the past. Because the static conflict set of variable \( x \) determines the behavior of a backtracking algorithm when instantiating variables in the subtree of the backtrack tree rooted at \( x \), the values instantiating the variables in this static conflict set are used to determine when good values are valid. For example, suppose the algorithm instantiates variables \( x_d \) and \( x_g \) in the DFS reordered backtrack tree displayed in Figure 6. The static conflict set of variable \( x_d \) is \( \{ x_d, x_c \} \). Therefore, the value \( v \) instantiating \( x_c \) can be marked good with respect to \( x_d \) at least until variable \( x_a \) is revisited. Subsequent instantiations of \( x_c \) with \( v \) before \( x_a \) is revisited imply variables \( x_d \) and \( x_g \) can be successfully instantiated once again. This enables the algorithm to skip ahead to variable \( x_e \) without sacrificing completeness or correctness, assuming \( x_d \) and \( x_g \) are reinstantiated before a solution is returned if the end of the variable order is reached.

Let the skip point of a vertex \( x \) be the first non-descendent variable of \( x \) in the backtrack tree to follow it in the ordering (if there is one). For the DFS-reordered graph in Figure 6, only the vertices \( x_d \) and \( x_g \) have skip points, and they are both \( x_e \).
FIGURE 5. Learning Backtrack

RETREAT()
1 let $x_j$ denote the backtrack parent of $x_i$
2 move the value assigned to $x_j$ into the elimination list of $x_i$
3 $i \leftarrow j$

ADVANCE()
1 for each reset child $x_c$ of $x_i$ do
2 empty the elimination list of $x_c$ into the domain of its backtrack parent
3 $i \leftarrow i + 1$

Each variable now maintains an “elimination list” of values that originally resided in the domain of its backtrack parent. Eliminated values are not returned by the iterators until they are restored by line 2 of ADVANCE.

5. Learning from Successes

In this section, we extend the previous learning backtrack algorithm in order to prevent it from repeatedly solving certain solvable subproblems. The same idea was exploited in [Bayardo & Miranker, 94] to produce an optimal backtrack algorithm for tree-structured problems. While the previous enhancements work correctly given any variable ordering, this enhancement requires that the backtrack tree be ordered according to the order in which a depth-first search procedure, starting at the root of the backtrack tree, visits each vertex (as is the case in Figure 3). Using a depth-first search procedure to order variables in a graph is termed DFS ordering.

Now, given a constraint graph ordering that lacks a DFS-ordered backtrack tree, we can reorder the variables of the constraint graph by DFS ordering (starting from the root) its backtrack tree, and reordering the constraint graph respectively. This DFS reordering step preserves the original induced graph, thus all static conflict sets and backtrack/reset parent/child relationships are preserved. Figure 6 illustrates an ordered graph and its backtrack tree, and the result of DFS reordering the backtrack tree and the respectively reordered graph. Dashed arcs are those added to form the induced graph, and the ordering is assumed to be from top to bottom.

Note that a DFS-reordered graph does not necessarily correspond to a DFS-ordered graph, which is one in which a DFS traversal was used to order the nodes of the constraint graph instead of the backtrack tree. This is because the backtrack tree is constructed from the induced graph, which may have more edges than the constraint graph. By DFS ordering the constraint graph to produce the variable order, no DFS reordering is required. However, DFS orderings have been shown to perform poorly on average [Dechter & Meiri, 92], so this technique is not recommended.
**THEOREM 3.1:** In static graph-based backtrack, at the time a call is made to RETREAT, the static conflict set of variable $x_i$ is a conflict set.

The proof idea is that the static conflict set of a variable $x$ contains any variable whose assignment could have affected the algorithm’s behavior while trying to instantiate $x$ and its descendents in the backtrack tree. This is because variables in different branches of the backtrack tree must be nonadjacent in the constraint graph, implying each branch can be solved independently as discussed in [Freuder & Quinn, 85]. What is essentially this same idea has been exploited by various schemes including graph-based backjumping [Dechter, 90], conflict-directed backjumping [Prosser, 93], and pseudo-tree search [Freuder & Quinn, 85].

4. Learning from Failures

Dechter [Dechter, 90] introduces various polynomial space-bounded schemes which remove domain values and/or record additional constraints during backtracking. These techniques are termed *learning* since they prevent certain failures from repeating. Here, we restrict attention to learning schemes that remove domain values from the problem since they introduce no significant space overhead and can be applied easily to schemes where constraints may be represented implicitly (e.g. through comparison operators such as <, >, and =).

When a conflict set is found that contains a single variable $x$, the value instantiating $x$ can be removed from the problem since it belongs to no solution. This is termed “first-order” learning in [Dechter, 90]. We further note that any value instantiating a variable in a conflict set can be removed from the problem, as long as it is restored if another variable in the conflict set is revisited by backtrack. For instance, suppose the algorithm instantiates the variable $x_4$ from the constraint graph in Figure 3 with a value $v$. Suppose the algorithm then attempts to instantiate variable $x_5$ but fails. A conflict set at this point, according to theorem 3.1, is $\{x_1, x_4\}$. As long as the value instantiating variable $x_4$ remains its instantiation, subsequent instantiations of variable $x_4$ with $v$ will lead to the exact same failure. We can thus remove $v$ from the problem, at least until a new value is made to instantiate variable $x_1$. An idea similar to this is exploited in [Bruynooghe, 81] and [Ginsberg, 93].

We define the *reset parent* of a vertex as the second-deepest element in its static conflict set. Vertices with less than two elements in their static conflict sets do not have reset parents. In the algorithm presented in Figure 5, when a backtrack takes place from a variable $x$, the value instantiating its backtrack parent is removed from the problem until the reset parent of $x$ is revisited. If $x$ has no reset parent, the value is removed permanently. Let *reset child* denote the inverse of the reset parent relationship.
connecting any nonadjacent vertices that share a child. An $O(n^2)$ method for constructing
the induced graph appears in [Tarjan & Yannakakis, 84].

**FIGURE 3. Static conflict sets for vertices in an example ordered graph**

Let the **static conflict set** of a vertex in an ordered graph be the set of its parents in
the induced graph. Figure 3 displays the static conflict sets of each vertex in an example
graph whose vertices are ordered according to the subscripts of the vertex labels. The
dashed edge is the only edge added to form the induced graph. The **backtrack parent** of a
vertex is the vertex in its static conflict set with the highest position along the ordering. If
the static conflict set is empty, then the vertex does not have a backtrack parent. Since we
assume the graph is connected, all but the initial vertex in the ordering have backtrack par-
ten. Let **backtrack child** denote the inverse of the backtrack parent relationship. The **backtrack tree** of an ordered graph $G$ is a rooted tree with the same set of vertices as $G$, the first vertex in $G$ being the root. Two vertices are adjacent in the backtrack tree iff one vertex is the backtrack parent of the other. For example, for the ordered graph in Figure 3, if we remove the edges $(x_1, x_4)$ and $(x_1, x_5)$, and root the resulting noncyclic graph at $x_1$, we are left with its backtrack tree. The backtrack tree provides us a “map” of each potential backtrack point chosen by the subsequent algorithms. From now on, we refer to a vertex in a constraint graph and the variable it represents interchangeably.

**FIGURE 4. Static Graph-Based Backtrack**

```
RETREAT()
1    let $x_j$ denote the backtrack parent of $x_i$
2    $i \leftarrow j$
```

We redefine the RETREAT method in Figure 4 to create the static graph-based back-
track algorithm. **ADVANCE** and **MAKESOLUTION** are the same as those from chronological
backtrack. Correctness of the scheme can be proven from the following theorem:
\[
\{ (x_j, v_j) \mid 1 \leq j < i \} \cup \{ (x_j, v_j) \}, \text{ where } v_j \text{ is the value most recently to instantiate } x_j, \text{ is a partial solution.}
\]

3. Graph-based Backtracking

Many methods improving on chronological backtrack determine better backtrack points during backtrack search [Bruynooghe, 81; Dechter, 90; Freuder & Quinn, 85; Gashnig, 79; Ginsberg, 93; Prosser, 93]. A conflict set is a set of variables whose current assignments, taken together, are not part of any solution. In chronological backtrack, when the iterator of a variable \( x_j \) exhausts its domain values, it has effectively determined that the set of variables \( \{ x_1, x_2, \ldots, x_{i-1} \} \) is a conflict set. A backtrack is then made to the highest positioned variable in this conflict set \( x_{i-1} \). Some backtrack algorithms apply techniques which minimize the size of conflict sets so that when an iterator exhausts its values, it may be possible to backtrack further than the previous variable in the ordering. Suppose, for the case above, the algorithm determines a conflict set that does not contain variable \( x_{i-1} \). Backtracking to variable \( x_{i-1} \) would then lead to the exact same failure since a subset of variables whose positions precede that of \( x_{i-1} \) are instantiated with values that are not part of any solution. For this reason, it is a simple matter to prove that completeness of backtrack search will be preserved if the algorithm backtracks to the highest positioned variable of any conflict set it identifies.

We now improve on chronological backtrack by providing it with a mechanism for possibly identifying smaller conflict sets. The technique is a purely graph-based technique in that it uses only information garnered from the constraint graph to minimize conflict sets. The technique is similar, but not identical, to Dechter’s graph-based backjumping [Dechter, 90]. The difference lies in the fact that Dechter’s scheme requires maintaining a set of variables during backtracking, while most of the work for the technique presented here can be done prior to backtracking. The goal of our minimalistic approach is to simplify the complexity proofs. More sophisticated techniques for identifying conflict sets can be applied to improve average-case performance without sacrificing the improved worst-case bounds to be demonstrated in a later section. For instance, Prosser’s conflict-directed backjumping [1993] provides a technique for identifying conflict sets that are always subsets (though not necessarily proper) of those identified here assuming equivalent variable orderings.

We first introduce the necessary terminology. A child of a vertex in a graph with an ordering of its vertices (an ordered graph) is an adjacent vertex that follows it in the ordering. A parent is an adjacent vertex that precedes it. The induced graph [Dechter, 92] 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 recursively
A common technique for solving constraint satisfaction problems is backtracking. Figure 1 displays pseudo-code for an incompletely specified backtracking algorithm for solving constraint satisfaction problems. Throughout this paper, the functions ADVANCE, RETREAT, and MAKE SOLUTION will be (re)defined to produce various CSP solving algorithms. For example, chronological backtrack, a naive method for solving constraint satisfaction problems that advances and retreats over one variable at a time along the variable ordering, is defined in Figure 2 below.

**FIGURE 1. Generic Backtrack**

```plaintext
BACKTRACK(P)
1 a variable ordering \( x_1, \ldots, x_n \) is assumed
2 initialize an iterator over each variable domain: \( I_1, \ldots, I_n \)
3 \( i \leftarrow 1 \)
4 while true
5 do \( v \leftarrow \text{NEXT}(I_i) \)
6 if \( v = \text{nil} \)
7 then if \( i = 1 \)
8 then return \( \text{nil} \) ; no solution exists
9 else \text{RETREAT}()
10 else if \( v \) instantiates \( x_i \)
11 then if \( i = n \)
12 then return \text{MAKE SOLUTION}() ; solution found
13 else \text{ADVANCE}()
14 \text{RESET}(I_i)
```

We assume backtracking takes place on a static variable ordering imposed before backtracking commences. A CSP with an ordering imposed on its variables is termed an ordered CSP. The subscript of a variable in the pseudocode reflects its position along the variable ordering. An iterator over a variable domain returns each domain value exactly once in an arbitrary order, but starts over again each time it is reset (line 14). A domain value is said to be considered if it is returned by the iterator scanning its domain. The nil flag is returned by an iterator if all the values in its domain have been considered since its last reset. In the pseudocode, an iterator’s subscript refers to the position of the variable whose values it considers.

Backtracking algorithms attempt to incrementally extend partial solutions along the variable ordering. A value \( v \) is said to instantiate a variable \( x_i \) (line 10) if the mapping
Quinn, 85], and maximum nonseparable component size [Freuder, 85; Dechter, 87]. The algorithms that most effectively exploit constraint graph structure have worst-case time and space complexities that are exponential in a constraint graph parameter known as induced width [Dechter, 92]. While reset distance does not improve on induced width, we demonstrate that it comes close on average. In light of recent scalings of backtrack-based applications to disk-resident data [Chimenti et al., 89; Miranker & Brant, 90] and the poor performance of memory-intensive algorithms due to processor speeds outpacing improvements in average memory access times, our approach is likely to be a more practical alternative to bounding the worst-case performance of backtrack search.

We begin by introducing necessary terminology and the standard chronological backtrack algorithm for solving constraint satisfaction problems. We then apply three enhancements to chronological backtrack. The first two have been well explored, involving graph-based backtracking and learning from failure. The third enhancement, which we call “learning from success”, prevents the algorithm from repeatedly solving certain solvable subproblems. The performance of the culminating algorithm, jump-ahead backtrack, is theoretically and experimentally analyzed and compared to various other schemes.

2. Constraint Satisfaction Problems and Backtracking

A constraint satisfaction problem (CSP) consists of a set of variables and a set of constraints. Each variable is associated with a finite value domain, and each constraint consists of a subset of the problem variables called its scheme and a set of mappings of domain values to variables in the scheme. An assignment is a mapping of values to a subset of the problem variables such that a value mapped to a particular variable belongs to the domain of the variable. An assignment satisfies a constraint with scheme if restricted to the variables in is a mapping in . A partial solution to a CSP is an assignment that satisfies every constraint whose scheme consists entirely of variables mentioned in the assignment. A solution to a CSP is a partial solution mentioning every variable.

A binary constraint satisfaction problem is a CSP in which each constraint has a scheme of cardinality 2. Such constraints are called binary constraints. The constraint graph of a binary CSP has a vertex representing each variable and an edge connecting any pair of variables contained in the scheme of some constraint. Vertices in a constraint graph will be used to denote the variables they represent. To simplify the presentation, this paper assumes that all problems are binary. The techniques can be easily generalized to higher order problems as discussed in [Dechter, 90]. Without loss of generality, we also assume problems have connected constraint graphs.
Backtrack-Bounded Search in Polynomial Space*

Roberto J. Bayardo Jr. (bayardo@cs.utexas.edu)
Daniel P. Miranker (miranker@cs.utexas.edu)
Department of Computer Sciences, University of Texas at Austin

Abstract

We present and analyze a polynomially space-bounded backtrack algorithm for solving constraint satisfaction problems. We show the algorithm is capable of bounding worst-case runtime almost as effectively as the best exponential space-consuming schemes, and more effectively than various other schemes including the cycle-cutset method [Dechter, 90], pseudo-tree search [Freuder & Quinn, 85], and techniques exploiting nonseparable component decomposition of the constraint graph [Freuder, 85; Dechter, 87]. Experiments on randomly generated problems show the algorithm is capable of solving classes of problems on which a forward checking algorithm with dynamic search rearrangement [Haralick & Elliot, 80] often fails.

1. Introduction

The constraint satisfaction problem (CSP) is a combinatorial search problem occurring in the domains of vision processing, scheduling, and many others. Briefly, a CSP consists of a set of variables to which values must be assigned without violating constraints that disallow certain value combinations. In the general case, the CSP is NP-hard, and worst-case runtime of any CSP algorithm is, at best, exponential in the number of variables. Improved bounds are achieved on some problems by algorithms that exploit problem specific features. A common technique is to exploit the structure of the problem’s constraint graph (see [Dechter, 92] for a survey), where a constraint graph has a vertex for each variable and an edge between each pair of vertices representing variables on which a constraint is defined. In general, the more sparse the constraint graph, the greater the improvement in worst-case bounds. For instance, problems with noncyclic constraint graphs (“tree-structured” problems) can be solved in time linear in the size of the problem [Dechter & Pearl, 88].

This paper presents and evaluates a new algorithm for solving CSP that exploits constraint graph structure for informed backtracking and linear-space-bounded learning. The algorithm runs in worst-case time that is exponential in a constraint graph parameter called reset distance, and requires $O(n^2 + nk)$ space, where $n$ is the number of variables and $k$ bounds the number of values in any domain. We demonstrate that reset distance is less than or equal to other graph parameters dictating the exponential runtime of several other schemes including cycle-cutset size [Dechter, 90], pseudo-tree height [Freuder &

---

*This research was supported in part by grants from the State of Texas Advanced Technology Program, the Applied Research Laboratories Internal Research and Development Program, and ARPA, grant number DABT63-92-0042.