ELSEVIER Artificial Intelligence 71 (1994) 159–181

Artificial Intelligence

An optimal backtrack algorithm for tree-structured constraint satisfaction problems[☆]

Roberto J. Bayardo Jr[a,*], Daniel P. Miranker[b,1]

[a]Department of Computer Sciences, University of Texas at Austin, Taylor Hall, Rm. 141, Mail Code 60500, Austin, TX 78712, USA
[b]Department of Computer Sciences, University of Texas at Austin, Taylor Hall, Rm. 2.124, Austin, TX 78712, USA

Received July 1993; revised November 1993

Abstract

This paper presents and evaluates an optimal backtrack algorithm for solving tree-structured constraint satisfaction problems—a subset of constraint satisfaction problems which can be solved in linear time. Previous algorithms which solve these problems in linear time perform expensive preprocessing steps before attempting solution. The work presented here resolves the open problem posed by Dechter (1990) on the development of an algorithm which avoids this preprocessing. We demonstrate significant improvements in average-case performance over the previous state of the art, and show the benefits provided to backtrack enhancement schemes exploiting the easiness of tree-structured problems such as the cycle-cutset method (Dechter and Pearl, 1987).

1. Background and motivation

Tree-structured constraint satisfaction problems are a subset of constraint satisfaction problems which can be solved in optimal worst-case time $O(nk^2)$, where n is the number of variables in the problem, and k is the size of the largest variable domain [6]. This good bound has motivated the use of algorithms for solving tree-structured constraint satisfaction problems to improve the average-case (and often worst-case) performance of general backtrack schemes. For

[☆] 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.
[*] Corresponding author. E-mail: bayardo@cs.utexas.edu.
[1] E-mail: miranker@cs.utexas.edu.

instance, the cycle-cutset method [3, 7] enhances backtrack algorithms by identifying and quickly solving tree-structured subproblems encountered during backtrack execution. Another technique uses solutions to tree-structured approximations of the problem being solved to guide the order of value assignments made by backtrack [6].

Previous optimal algorithms for solving tree-structured problems perform expensive preprocessing on the problem input in order to reduce the search space explored by a naive backtrack procedure. Because naive backtrack can often find a solution quickly without any preprocessing, for many problem instances, preprocessing actually hurts performance. This behavior is typical of most preprocessing schemes, and has been demonstrated throughout the literature [3, 5, 10].

In order to exploit the topological structure of constraint satisfaction problems without costly preprocessing phases, various enhancements to naive backtrack have been suggested. Enhancements that are capable of improving the average-case performance of naive backtrack without substantially degrading performance on any problem instance include graph-based backjumping and shallow forms of learning [3]. Graph-based backjumping uses constraint connectivity information to improve the accuracy of backtrack points. Shallow forms of learning delete a domain value or add a limited amount of constraint information after particular dead ends in the search space in order to prevent the same failure from recurring.

While often performing poorly on average, preprocessing algorithms typically have better bounds on worst-case performance than backtrack algorithms using enhancements such as graph-based backjumping and shallow learning. With respect to tree-structured problems, we demonstrate that this is no longer the case. First, we show that by carefully ordering the variables of the problem, a backtrack procedure using known enhancements is within $\log n$ of optimal on tree-structured problems. We next introduce a new enhancement that involves recording the "good" domain values encountered during backtrack. Instantiating a variable with a good domain value allows skipping ahead in the variable search order to avoid solving the same subproblem multiple times. We apply this enhancement to obtain an optimal backtrack algorithm for tree-structured problems.

Through experiments on randomly generated problems, we demonstrate that the average-case performance of our algorithm is significantly better than that of a directional arc consistency preprocessing algorithm, the previous tree algorithm of choice. We also determine the benefits of using our algorithm with the cycle-cutset method for enhancing backtrack. Specifically, we show that it provides better mean speedups, and that it greatly reduces the probability of the cycle-cutset enhanced algorithm performing worse than the unenhanced algorithm on a randomly chosen problem instance.

1.1. Constraint satisfaction problems

A *constraint satisfaction problem* (CSP) consists of a set of variables, a finite and discrete value domain for each variable, and a set of constraints. A constraint is defined on a subset of the variables called its *constraint subset*, and contains a *relation* which is a subset of the Cartesian product of these variables' domains. Each tuple within the relation holds values which, when assigned to their respective variables, are said to *satisfy* the constraint. A *solution* to a CSP is an assignment of domain values to variables such that every constraint is satisfied, or nil if no such assignment exists.

Without loss of generality, we assume that each constraint has a unique constraint subset. If it were otherwise, we could combine the constraints with equivalent constraint subsets into a single constraint by taking the intersection of their relations.

Constraints with constraint subsets of cardinality 2 are *binary* constraints, and a *binary constraint satisfaction problem* is one in which all constraints are binary. The *constraint graph* for a binary CSP has a node representing each variable, and an edge connecting each pair of nodes representing variables on which a binary constraint is defined. A *tree-structured constraint satisfaction problem* is a binary CSP whose constraint graph is acyclic.

1.2. CSP algorithms

Backtrack algorithms for solving constraint satisfaction problems incrementally expand a partial solution to the problem, backing up to a previous state whenever a dead end in the search space is encountered. Backtrack algorithms differ primarily in the amount of work applied to reducing the search space. *Preprocessing* algorithms perform all search space reducing work prior to a backtrack phase. The best-case performance of a preprocessing algorithm can only be as good as the minimum time required by the preprocessing step since no attempts are made at a solution until preprocessing terminates.

The following subsections present an example of a pure backtrack algorithm and a preprocessing algorithm. While meant primarily to provide background information, we present these algorithms in detail in order to establish terminology used in subsequent sections. We use pseudocode based on the style used in [2].

1.2.1. Naive backtrack

One of the simplest backtrack algorithms is *naive backtrack*, which is also referred to in the literature as chronological or pure backtrack. Naive backtrack attempts to successively instantiate variables along an arbitrarily imposed variable ordering. A variable is said to be *instantiated* if it is assigned a domain value which satisfies all constraints defined over it and the variables previous to it along the ordering. If at any point the algorithm fails to instantiate a variable, the algorithm selects the previous variable in the ordering as the *backtrack variable* and attempts to instantiate it with a different domain value.

Fig. 1 presents the naive backtrack algorithm. The algorithm first orders the problem's variables and initializes an iterator to scan over each variable domain. While naive backtrack works correctly on any ordering of the variables, most implementations use specific ordering heuristics to improve performance. An experimental evaluation of common variable ordering policies with respect to

```
   SOLVE-NAIVE($P$)
1  impose an ordering on the variables: $X_1, X_2, \ldots, X_n$
2  initialize iterators $I_{1\ldots n}$ for scanning the domain of each variable
3  $i \leftarrow 1$
4  while true do
5      $v \leftarrow$ NEXT($I_i$)
6      if $v = nil$ then
7          if $i = 1$ then
8              return nil
9          else $i \leftarrow i - 1$
10     else if $v$ instantiates $X_i$ then
11         if $i = n$ then
12             return current variable assignments
13         else $i \leftarrow i + 1$
14             RESET($I_i$)
```

Fig. 1. The naive backtrack algorithm for solving constraint satisfaction problems.

naive backtrack and other CSP algorithms is provided in [5]. For succinctness, a variable at position i along an imposed ordering will be referred to simply as variable i. We also call the first variable in an ordering the *root* variable, and the others *non-root* variables.

1.2.2. *Arc-consistency-based algorithms*

Arc consistency [11, 12] is a preprocessing scheme which accepts a CSP and removes all unsupported values from its domains. Using the terminology from [11], an *unsupported* value is one whose assignment alone prevents the satisfaction of some binary constraint involving the assigned variable. We say a value *supports* another value with respect to a particular constraint if their assignments satisfy the constraint. A *width*-1 variable ordering is one where every node in the problem's ordered constraint graph connects to at most one node preceding it in the ordering. For any tree-structured problem, a width-1 ordering exists and can be found in time O(n) by performing a depth-first traversal of the constraint graph.

Width-1 orderings are of interest because determining whether a value instantiates a variable along a width-1 ordering requires testing at most one constraint. After achieving arc consistency, we are guaranteed that given any variable assignment, any constraint involving the assigned variable can be satisfied by some other assignment. Invoking naive backtrack along a width-1 ordering after arc consistency preprocessing therefore achieves a solution backtrack-free, or in time O(nk). By using an arc consistency algorithm that is O(nk^2)-bounded (such as the one from either [1] or [12]), this approach to solving tree-structured problems is optimal in the worst case.

A formal proof that $O(nk^2)$ is optimal for tree-structured problems is provided in [6]. The proof idea is simply that each constraint in the problem must be examined at least once, and each such examination may require checking order k^2 assignments.

Directional arc consistency [6] is a more limited form of preprocessing that accepts a CSP and a fixed variable ordering, and removes from the variable domains any value whose assignment alone prevents the satisfaction of some binary constraint defined on the assigned variable and any other variable following it in the ordering. Directional arc consistency therefore removes only a subset of the domain values removed by arc consistency, enabling both simpler and faster implementations. After directional arc consistency preprocessing given a width-1 ordering, naive backtrack along that ordering achieves a solution backtrack-free. Because directional arc consistency can be performed in $O(nk^2)$ time [6], this scheme is also optimal. Due to its lower constant, this approach is preferable to full arc consistency preprocessing, and has been the algorithm of choice for use in backtrack enhancement schemes which exploit the easiness of tree-structured problems [3, 6].

An algorithm for solving tree-structured problems that performs directional arc consistency preprocessing (DAC) is provided in Fig. 2. This algorithm is essentially a translation of the tree algorithm from [3], although it is slightly simplified due to our additional assumptions: without loss of generality, we from this point on assume a tree-structured problem has a connected constraint graph. This allows us to assume that every variable except the root has exactly one parent, where the *parent* of a variable is a variable both preceding it in the ordering and connecting to it in the constraint graph. A width-1 ordering can be

```
   SOLVE-DAC($P$)
1  impose a width-1 variable order: $d = X_1, X_2, \ldots, X_n$
2  for $i = n$ downto 2 do
3    $p \leftarrow$ position of $X_i$'s parent
4    REVISE($X_i, X_p$)
5    if $X_p$'s domain is empty then
6      return nil
7  return SOLVE-NAIVE($P, d$)

   REVISE($X_c, X_p$)
8  for each value $v$ in $X_p$'s domain do
9    if there is no value in $X_c$'s domain supporting $v$ then
10     delete $v$ from $X_p$'s domain
```

Fig. 2. A directional arc-consistency-based algorithm for solving tree-structured problems.

thought of as imposing directionality on the edges of an acyclic constraint graph, where an edge is directed towards the node deeper in the ordering. This allows the use of terms such as parent, child, subtree, etc. for denoting nodes in the constraint graph, and hence variables in the tree-structured CSP.

We assume the position of each variable's parent has been precomputed by the variable ordering step. After preprocessing completes, the algorithm invokes a naive backtrack algorithm that accepts an already imposed variable order to solve the modified problem (line 7).

2. Backtrack algorithms for tree-structured problems

While DAC exhibits significantly better constants than algorithms using full arc consistency for solving tree-structured problems, much of the work performed by DAC is often unnecessary, particularly when searching for a single solution. Consider a tree-structured problem with constraints that allow any assignment. On this particular instance, naive backtrack returns a solution in $\Theta(n)$ steps (assuming a simple variable ordering heuristic), whereas DAC requires $\Theta(nk)$ steps to ensure directional arc consistency (which in fact already holds) before even attempting a solution. A better approach is to apply simple runtime enhancements to naive backtrack in an effort to obtain a good bound on worst-case performance without requiring expensive preprocessing. Following this approach, we have developed an algorithm with the same worst-case complexity as DAC along with significant improvements in performance on average.

In this section, we begin by applying known enhancements and a special variable ordering policy to naive backtrack to obtain TreeTracker-1, an algorithm with a bound that is near-optimal on tree-structured problems. We present this algorithm because its combination of simplicity and performance may make it preferable for some applications. We then present a new enhancement involving the recording of "good" domain values identified during backtrack, where instantiating a variable with a good domain value may allow advancing beyond more than one variable in the ordering. We apply this enhancement to obtain TreeTracker-2, an algorithm that is optimal on tree-structured problems. Lastly, we empirically compare both these algorithms with DAC.

2.1. TreeTracker-1

We obtain TreeTracker-1 (TT1) through minor modifications to naive backtrack. It incorporates into backtrack search a simplified implementation of graph-based shallow first-order learning [3] that unconditionally deletes a domain value from the problem with each backtrack. It also uses a depth-first search variable ordering procedure that applies a novel tie-breaking rule to help minimize the distance between children and their parents in the ordering. The intent of minimizing these distances is to reduce the amount of work undone with

```
     SOLVE-TT1($P$)
 *1  impose TT1 variable order: $X_1, X_2, \ldots, X_n$
  2  initialize iterators $I_{1\ldots n}$ for scanning the domain of each variable
  3  $i \leftarrow 1$
  4  while true do
  5    $v \leftarrow$ NEXT($I_i$)
  6    if $v = nil$ then
  7      if $i = 1$ then
  8        return nil
 *9      else $i \leftarrow$ position of $X_i$'s parent
*10        delete the value assigned to $X_i$ from its domain
 11    else if $v$ instantiates $X_i$ then
 12      if $i = n$ then
 13        return current variable assignments
 14      else $i \leftarrow i + 1$
 15          RESET($I_i$)
```

Fig. 3. The TreeTracker-1 algorithm.

each backtrack. Not only does the tie-breaking rule improve worst-case complexity, it also helps average-case performance.

Fig. 3 presents pseudocode for TT1, and flags with an asterisk the lines which were either changed or do not appear in naive backtrack. First, a special depth-first traversal of the constraint graph is used to determine the variable order. This traversal starts at an arbitrary node and simply ensures that the largest subtree of any node is visited last. The order in which the nodes are first visited defines the TT1 variable order. Note that since the ordering is depth-first, it must also be width-1.

Fig. 4 illustrates the TT1 variable ordering process on an example acyclic constraint graph. From a particular starting point, there may be more than one legal ordering since the smaller subtrees of a node can be visited by the traversal algorithm in arbitrary order. We found that visiting the subtrees of a node in order from smallest to largest results in the best average-case performance in terms of constraint tests performed. In order to maintain a linear complexity for the variable ordering step and still get some of the benefits of a strict ordering, our implementation (evaluated later) performs a single pass of pairwise swaps over the edge list of each node to ensure the largest subtree is positioned at the end.

Lines 9 and 10 of Fig. 3 contain the only remaining differences between TT1 and naive backtrack. With each backtrack from a non-root variable, instead of naively selecting the most recently instantiated variable as the backtrack variable, TT1 selects the current variable's parent. Then, to prevent the same failure from repeating, the value assigned to the parent is deleted from its domain.

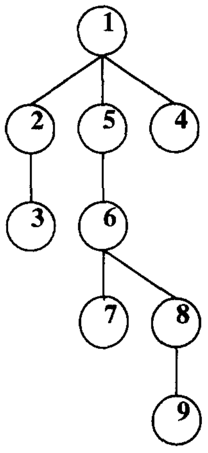

Fig. 4. A TT1 ordering of an acyclic constraint graph.

2.1.1. Correctness

We define the *run* of a particular iterator to be the set of values returned by the iterator since its most recent reset, excluding those values deleted from the problem and any value currently assigned to its variable. Note that all values in the run of a variable's iterator become members of the run while the same value is assigned to the variable's parent. This is because after reassignment of the parent variable, before the iterator is advanced again, it gets reset by line 15. We refer to the value assigned to the parent during the course of a variable's iterator run as the *parent value*.

Lemma 2.1. *At any point during TT1 execution, the values in the run of any iterator over a non-root variable domain do not support the parent value.*

Proof. Given the width-1 ordering, if any value v supporting the parent value is encountered by the iterator of a non-root variable, the variable is instantiated with v and control progresses to the remaining variables. For the current run of this iterator to be extended, a backtrack must take place to the iterator's variable, which causes v to be deleted from the problem. It is therefore impossible for a value supporting the parent value to become a member of an iterator's run. □

Theorem 2.2. *TT1 returns a non-nil solution if a non-nil solution exists, and returns nil otherwise.*

Proof. Besides the specific variable ordering policy required by TT1, the only difference between it and naive backtrack is the backtrack scheme which is invoked when an iterator exhausts the domain values of a non-root variable (line 9). At this point, the run of the current variable's iterator contains every value remaining in the domain of the current variable. Thus by Lemma 2.1, the value assigned to the parent is unsupported, and hence the problem cannot be solved unless the parent variable is instantiated with another value. The theorem follows from this fact and the correctness of naive backtrack. □

2.1.2. Complexity analysis

Assuming TT1 borrows as much of the original problem representation as possible, the only additional space requirement beyond the problem input is O(n) to store the iterators and variable positioning information.

Note that with the exception of variable ordering and returning the solution (both requiring O(n) steps), all other primitives used by TT1 can be implemented as O(1) operations. The runtime of TT1 is thus O(T), where T is the number of calls to the iterator function NEXT(). We can naively bound the number of such calls by first noting that at most nk backtracks are possible, since with each backtrack a different domain value is eliminated from some variable's domain. Next, there can be no more than nk calls between backtracks, otherwise we could show a solution was found. This leaves a bound of O(n^2k^2), which would be tight for some problems if TT1 selected arbitrary width-1 variable orderings.

The purpose of TT1's variable ordering policy is to reduce the occurrence of backtracks which back up over many variables. Note that because each iterator RESET() allows at most $k+1$ calls to NEXT(), we can also denote algorithm runtime by counting the number of iterator resets. Without any backtracks, exactly n resets (including the implicit reset resulting from initializing the first domain iterator) are performed. Each backtrack allows at most r more resets, where r equals the number of variables in the ordering between the failure variable and its parent, including the failure variable itself. If we let R equal the sum of all such values for all backtracks performed during the course of the algorithm, then runtime can be denoted as

$$\mathrm{O}(Rk + nk)\,. \tag{1}$$

Let r_i bound the contribution to R resulting from a backtrack to variable i. Since each variable can be backtracked to at most k times, we have the following inequality:

$$k \cdot \sum_{i=1}^{n} r_i \geqslant R\,. \tag{2}$$

Now, suppose d_i equals the number of descendents of a non-leaf variable i, and l_i equals the number of variables in the largest subtree rooted at one of its children. For example, the problem illustrated in Fig. 4 has $d_1 = 8$ and $l_1 = 5$. Because the variable ordering policy is a depth-first search procedure, only descendents of variable i could exist between it and the failure variable. Furthermore, because a backtrack to variable i can take place only from one of its children, and the tie-breaking rule ensures that variables in the largest subtree must follow all children in the ordering, it follows that $r_i = d_i - l_i + 1$. Since leaf variables cannot be backtrack variables, for them $r_i = 0$. We can now prove the following:

Lemma 2.3. $\sum_{i=1}^{n} r_i \leq n \lg n$.

Proof (by induction on the number of variables).

Basis. For a tree-structured problem with only one variable, the claim clearly holds.

Inductive step. Suppose the theorem holds for any tree-structured problem of less than n variables. Now, consider the root variable in the ordering of an n-variable tree-structured problem. Because we assume a connected constraint graph, all variables following the root in the ordering must be its descendents. Let d equal the number of children of the root. Also, let $n_1, n_2, \ldots, n_d$ denote the number of variables in each of the subtrees rooted at its children, and n_m denote the maximum.

The contribution of the root variable to the sum is $(n-1) - n_m + 1 = n - n_m$, and by the inductive hypothesis, the contribution to the sum of the nodes in each subtree j, where $1 \leq j \leq d$, is $n_j \lg n_j$. The entire summation can therefore be bounded by:

$$\begin{aligned} & n_1 \lg n_1 + n_2 \lg n_2 + \cdots + n_d \lg n_d + (n - n_m) \\ & \leq (n_1 + n_2 + \cdots n_d) \lg n_m + n - n_m \\ & \leq n \lg n_m + n - n_m \\ & \leq n \lg n . \end{aligned}$$

(The last step follows from a lemma proved in the appendix.) □

Theorem 2.4. *TT1 runs in worst-case time* $O(n \log n \cdot k^2)$.

Proof. Follows directly from Lemma 2.3 and Eqs. (1) and (2). □

It is easy, albeit tedious, to prove the tightness of this bound by constructing a class of problems, each with a constraint graph that is a perfectly balanced binary tree (with respect to the node chosen as the root), which exhibit a worst-case number of long backtracks.

2.2. TreeTracker-2

TreeTracker-1 uses a special variable ordering policy to reduce the amount of work required to recover from each backtrack. Here we present the optimal TreeTracker-2 (TT2), which instead uses one additional enhancement to better accomplish the same goal. This enhancement keeps track of domain values that were previously used to completely instantiate an entire subtree of the problem. We refer to these domain values as good domain values, and they are stored in goodlists associated with each variable domain. TT2 exploits the fact that when a variable is instantiated with a good domain value, the variables in the subtree rooted at the variable can be instantiated later without backtracking. By skipping over the variables in these subtrees and quickly instantiating them only after reaching the end of the variable order, TT2 avoids solving the same subproblems over and over again.

Fig. 5 provides a pseudocode description of TT2 and flags with an asterisk the differences from TT1. The first difference is that TT2 can use an arbitrary depth-first variable order. Next, any domain value successfully used to instantiate a variable is moved from the variable's domain to the variable's *goodlist*. A backtrack results in removal of the value last added to the backtrack variable's goodlist, instead of removing anything from the variable domain as is done by TT1.

After moving a value to a goodlist, TT2 increments the current variable index

```
     SOLVE-TT2($P$)
 *1  impose depth-first variable order: $X_1, X_2, \ldots, X_n$
  2  initialize iterators $I_{1\ldots n}$ for scanning the domain of each variable
  3  $i \leftarrow 1$
  4  while true do
  5    $v \leftarrow$ NEXT($I_i$)
  6    if $v = nil$ then
  7      if $i = 1$ then
  8        return nil
  9      else $i \leftarrow$ position of $X_i$'s parent
*10        delete the value last added to $X_i$'s goodlist
 11    else if $v$ instantiates $X_i$ then
 12      if $i = n$ then
*13        instantiate skipped variables with goodlist values
 14        return current variable assignments
*15      else move $v$ from $X_i$'s domain to its goodlist
 16          $i \leftarrow i + 1$
*17          TRY-GOOD-LISTS($i$)

  *  TRY-GOOD-LISTS($i$)
     ($i$ is passed by reference)
 18  for each value $v$ in $X_i$'s goodlist do
 19    if $v$ instantiates $X_i$ then
 20      $i \leftarrow$ position of next non-descendent of $X_i$ in the ordering
 21      TRY-GOOD-LISTS($i$)
 22      return
 23  RESET($I_i$)
```

Fig. 5. The TreeTracker-2 algorithm.

and calls TRY-GOOD-LISTS(), which simply attempts to instantiate the new current variable with a value from its goodlist alone. If successful, this function jumps ahead to the next variable in the ordering that is not a descendent of the current variable. The function is then called recursively until failure, at which point the iterator of the current variable is reset and control is returned to the main procedure.

Because TRY-GOOD-LISTS() only explicitly instantiates root variables to particular subtrees of the problem, after instantiating the final variable in the ordering, TT2 must instantiate any skipped variables before a solution can be returned (line 13). For each variable instantiated by TRY-GOOD-LISTS(), this process successively instantiates the descendents of the variable along the existing variable ordering. By considering only values within the goodlists, we show in the next section these instantiations are found backtrack free.

2.2.1. Correctness

We show that once TT2 instantiates the last variable, any skipped variables can be successfully instantiated by the process described above. Correctness follows from this fact and the correctness of TT1.

We say that TT2 *advances* over a variable if it either instantiates or skips over the variable. Note that once a value is used to instantiate a variable by line 11, it is moved to the goodlist and it will remain there until algorithm termination if and only if the algorithm subsequently advances over all variables in the variable's subtree. We call the values which TT2 permanently keeps in a variable's goodlist the *good* values for that variable. We can imagine the algorithm as explicitly marking a domain value as good immediately after advancing over the subtree rooted at the variable instantiated with the value. We now prove an invariant from which correctness will be argued.

Lemma 2.5. *At any point during TT2 execution, for any good value v identified for a variable, TT2 has identified good values supporting v for each of the variable's children.*

Proof. (by induction on the number of good values identified by TT2).

Basis. The first good values identified are those first used to instantiate the initial subtree in the ordering, if that ever happens. By definition of instantiation the claim trivially holds.

Inductive step. Assuming all good values identified by TT2 have the stated property, it is easy to see that any good values subsequently identified by the algorithm must also have this property: Any subtree advanced over by the algorithm must either be completely instantiated by line 11 of the algorithm, or it may have the root nodes to some of its own subtrees instantiated by line 19. The values used to instantiate variables in the subtree by line 11 are precisely the ones identified as good once the subtree has been advanced over, and it thus follows that for any such value v, for each child of the variable instantiated by v, either a good value supporting v is already identified for the child, or one is identified as good at the same time. □

Theorem 2.6. *TT2 returns a non-nil solution if a non-nil solution exists, and returns nil otherwise.*

Proof. It should be clear that TT2 always terminates. To prove the claim, we demonstrate that each termination state returns an appropriate result.

Consider line 14 which returns the variable assignments. Because the goodlists of any skipped variable must be non-empty, Lemma 2.5 implies that any skipped variables can be instantiated with goodlist values backtrack-free along the existing width-1 variable order. The assignments returned by line 14 must therefore constitute a solution to the CSP.

Now consider line 8 which returns nil. Like TT1, TT2 deletes only unsupported values from the problem. For control to reach line 8, TT2 must first delete all values in the domain of the root variable. If an entire domain consists of values deemed unsupported by the algorithm, then only the nil solution exists. □

2.2.2. Complexity analysis

The only additional space required beyond the problem input is $O(n)$ to store the iterators and variable positioning information, assuming the space required by the goodlists is obtained by reclaiming space relinquished by the variable domains. We now prove a bound on runtime.

Theorem 2.7. *TT2 runs in worst-case time* $O(nk^2)$.

Proof. Besides the one implicit reset of the iterator for the domain of the root variable, all other iterator resets take place on line 23. Exactly one reset is made for each call to TRY-GOOD-LISTS() by line 17, and there can be no more than one such call for every domain value in the problem. This implies a bound of $O(nk^2)$ on the time spent in the main procedure without counting the time spent in TRY-GOOD-LISTS().

For each possible parameter value j, TRY-GOOD-LISTS() is called with i set to j at most once for each value that gets assigned to variable j's parent by line 11. This implies there can be no more than nk total calls to TRY-GOOD-LISTS() since line 11 assigns each value at most once. Because each call to TRY-GOOD-LISTS() requires only $O(k)$ steps, the total time spent in it is also $O(nk^2)$. □

2.3. Empirical comparison

2.3.1. Implementation

To test the average-case performance of our algorithms, we implemented a random tree-structured problem generator accepting three parameters, n, k, and p. The generator works by first randomly constructing a constraint graph such that out of all the possible constraint graphs that are trees for a given set of nodes of size n, each one is generated with equal probability. Variable domains are then filled with exactly k values, and for each constraint, a truth table is randomly generated according to parameter p, where p is the probability that two values, one from each domain on which the constraint is defined, satisfy the constraint.

We next implemented TT1, TT2 and DAC. Both TT2 and DAC provide a good amount of leeway in choosing variable orderings. We tried several schemes, and interestingly, we found that the variable ordering policy required for TT1 results in the best performance on average for both TT2 and DAC. TT2 benefits from the TT1 variable ordering policy because it minimizes the time required to recover from backtracks which are not soon after followed by a variable instantiation with a good domain value. DAC benefits from the policy because it ensures the algorithm propagates up the most constrained paths from leaves to root early in the search, thereby increasing the probability that an empty domain will be quickly encountered. Because of the negligible amount of additional overhead required by the TT1 ordering process over other schemes, we therefore suggest it be used by all of these tree algorithms. The following experimental results are for all of the algorithms using TT1's variable ordering policy.

2.3.2. Experimental results

We computed the average number of constraint tests performed by each algorithm on problems generated with particular settings of n, k, and p to compare algorithm performance. Each algorithm performs roughly the same amount of work as the others within its inner loop. Since a count of constraint tests reflects the number of times an algorithm's inner loop is executed, this value provides a good relative performance indicator independent of the machine being used to conduct the experiments.

Each of our experiments fixed n and k, and varied p from 0 to 1 in increments of 0.005. A data point produced by an experiment for a particular algorithm is the average number of constraint tests over 5000 runs on different problems. For the first experiment, we fixed both n and k at 10. We next investigated the effects of varying k by increasing its value to 20 while leaving n at 10. Our final experiment shows the effect of varying n by setting n to 20 and k to 10.

Even for the modest parameter values chosen for the initial experiment, Fig. 6 shows that TT2 performs substantially better than DAC across the entire spectrum of p. Figs. 7 and 8 demonstrate that for larger values of either n or k, the improvement in performance becomes even more substantial. We anticipated this result because as problem size increases, there exists more opportunity for the TreeTracker algorithms to avoid search space reducing work which DAC has no choice but to perform.

What was somewhat surprising was the negligible difference in performance between TT1 and TT2. The similarity between their data points precluded the plotting of TT1 data in the figures. Most of the data points produced by the two algorithms were almost identical, and any differences were small. Respective data points differed by no more than 3 on the y-axis for the first experiment, and no more than 8 and 13 for the second and third experiments, with TT2 always

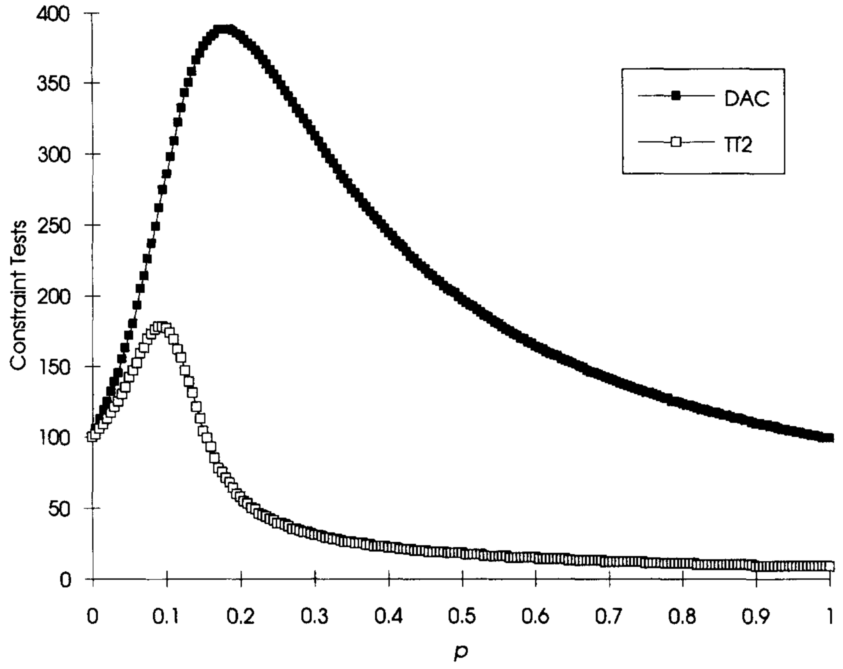

Fig. 6. Average number of constraint tests performed by DAC and TT2 on randomly generated tree-structured problems when n and k are fixed at 10.

performing better than or equal to TT1. To elicit a non-negligible difference, we ran an additional experiment with n set to 100 and k to 10. For this experiment, the points differed by a maximum of 171 at the peaks of the curves. However, TT1 still outperformed DAC considerably, the peaks of each curve reaching 3709, 1094, and 923 for DAC, TT1, and TT2 respectively. We conclude that despite its suboptimal runtime bound, the simplicity of TT1 may make it the algorithm of choice for many applications.

3. Re-evaluating the cycle-cutset method

The *cycle-cutset method* [7] is an enhancement to backtrack algorithms for solving any constraint satisfaction problem. For a binary CSP, consider maintaining the set of constraint graph nodes representing variables currently instantiated by a backtrack algorithm attempting to solve it. The cycle-cutset method exploits the fact that when the removal of this set from the constraint graph renders the graph acyclic, a tree algorithm can be invoked to attempt instantiation of the remaining variables after performing a limited amount of work required to fully switch to a tree representation. The cycle-cutset method applies to non-binary problems as well, only requiring that a Hyper-graph representation of the problem be used instead [13].

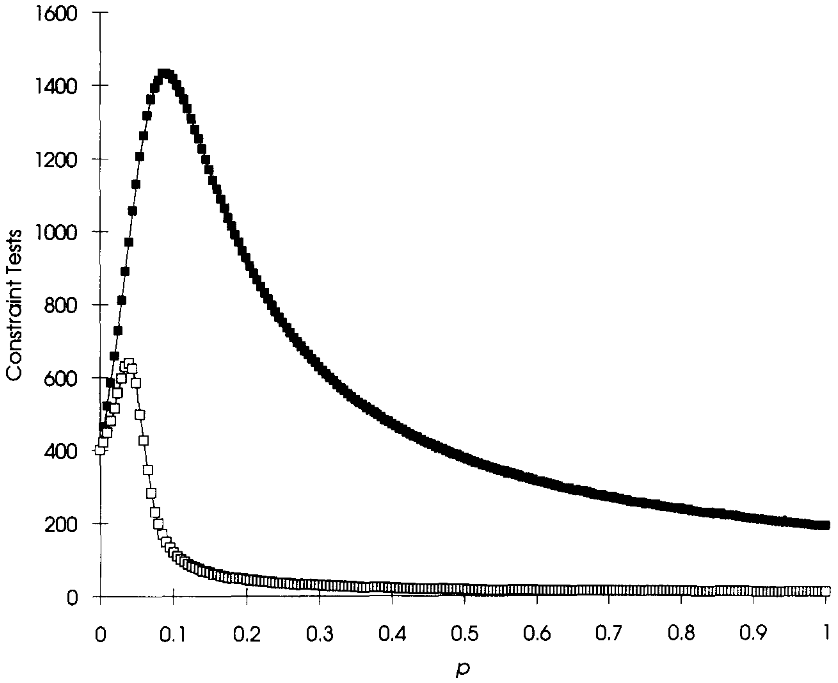

Fig. 7. Average number of constraint tests performed by DAC and TT2 on randomly generated tree-structured problems when $n = 10$ and $k = 20$.

The set of variables instantiated before the tree algorithm is invoked is called the *cutset*. Once these variables have been instantiated, the domains of the variables outside the cutset are filtered. The filtering process removes all domain values whose assignments would be in conflict with the assignments made to the cutset variables. More formally, to fully switch to a tree representation once the cutset variables are instantiated, for each variable V outside the cutset and each cutset variable C connecting to it in the constraint graph, any value in V's domain must be removed if it and the value currently assigned to C fail to satisfy the constraint on V and C. This work can be performed either as a preprocessing step prior to invoking the tree algorithm, or lazily during tree algorithm execution.

Work required to switch to a tree representation requires $O(n^2k)$ steps, and the tree algorithm requires $O(nk^2)$ steps. Because the tree algorithm can be invoked with each possible combination of cutset variable instantiations, the runtime of naive backtrack enhanced with the cycle-cutset method is $O(k^c(nk^2 + n^2k))$ on binary CSPs, where c denotes the cardinality of the cutset. More importantly, due to the speed of algorithms for tree-structured problems, an improvement in average-case performance is often realized. The amount of improvement resulting from the cycle-cutset method is clearly related to the performance of the tree algorithm it employs. In [3] Dechter expresses that tree algorithms without intentional preprocessing would be very beneficial to the cycle-cutset scheme. We

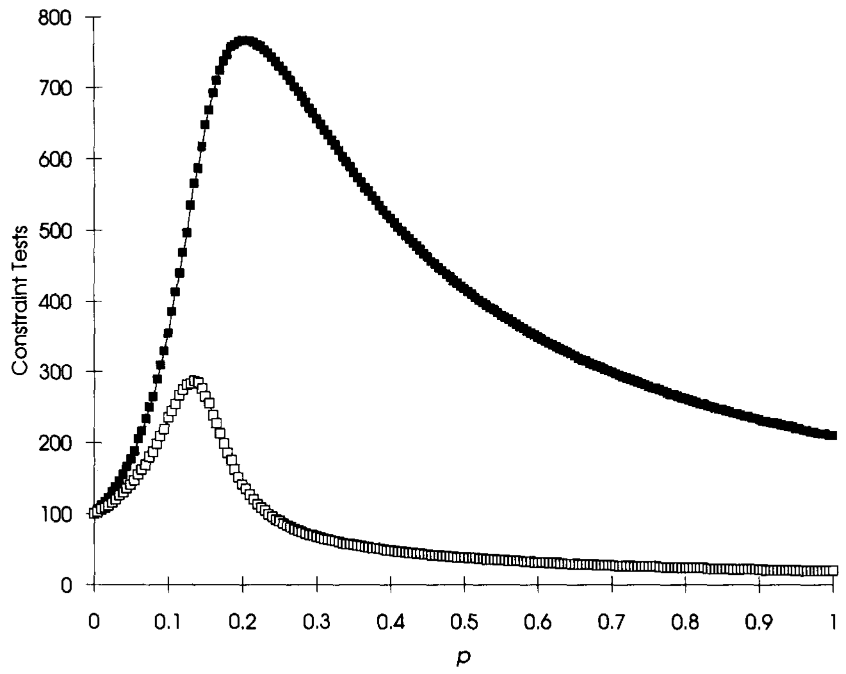

Fig. 8. Average number of constraint tests performed by DAC and TT2 on randomly generated tree-structured problems when $n = 20$ and $k = 10$.

therefore performed experiments similar to those in [7] in order to determine the benefits, and to further compare the performance of the TreeTracker algorithms with that of DAC. Due to the near-identical performance of the two TreeTracker algorithms, here we only evaluate TT2.

3.1. Implementations

3.1.1. Algorithms

We implemented a naive backtrack algorithm for solving binary CSPs, along with cycle-cutset enhanced versions employing DAC and TT2. The variable ordering policy used by our implementation of naive backtrack is known as the minimum-width ordering policy [8], and was shown in [5] to result in better performance than other common static variable ordering policies with respect to a naive backtrack algorithm. We found the maximum-degree variable ordering policy, which was used in the experiments performed in [7], to provide the smallest cutset sizes on average. The maximum-degree variable ordering policy orders the variables in decreasing order of their degree in the constraint graph. Despite the appeal of smaller cutsets, we chose not to use this policy because it was shown in [5] to be clearly inferior to minimum-width with respect to average-case performance of naive backtrack.

3.1.2. Switching to tree representations

The cycle-cutset method performs work required to switch to a tree representation once the cutset variables have been instantiated. This work filters from the domains of the variables outside the cutset all values which are incompatible with the cutset variable assignments. At first glance, it might seem that performing this work as lazily as possible might be best. The laziest implementation avoids filtering any domain value that the tree algorithm is never required to fetch. Much unnecessary work is therefore avoided every time the tree algorithm terminates without reaching all the variables, and/or all domain values. On the other hand, because this work alone often empties a variable domain, if performed somewhat eagerly, control is often returned to backtrack without involving the tree algorithm.

We found the scheme resulting in best performance on average, regardless of the tree algorithm being used, to be the following: Before invoking the tree algorithm, perform the minimum amount of work required to determine whether an empty domain results from the filtering of incompatible domain values alone. If a domain turns up empty, simply return control to backtrack. Otherwise, invoke a tree algorithm which is modified to filter out any remaining incompatible domain values encountered during its execution. (To avoid an additional test in the innermost loops of the tree algorithms, our implementations use separate loops for already filtered and unfiltered domains). Our finding that this limited lookahead step best improves performance is in line with those in [10], where the authors evaluate various lookahead schemes and find that forward checking, the most limited form of lookahead investigated, performs best.

3.1.3. Problem generation

For the experiments, we implemented a random binary CSP generator accepting parameters n, k, p_1 and p_2, where p_1 is equivalent to the parameter p described earlier, and the last is the probability that a constraint exists over a pair of variables. We first generated problems for n and k both set to 10, p_1 chosen uniform randomly from the range (0.001, 0.999), and p_2 chosen uniform randomly from the range (0.1, 0.5). These ranges were chosen so that problems with widely varying difficulties and cutset sizes would be generated. Problems were then generated similarly with $n = 10$, $k = 20$ and $n = 20$, $k = 10$. Each problem with a connected constraint graph was then solved using naive backtrack, and both the DAC and TT2 cycle-cutset enhanced versions, counting the number of constraint tests required to solve each problem by each algorithm.

3.2. Experimental results

3.2.1. General effects of cycle-cutset enhancement

Figs. 9 and 10 demonstrate the general effect of enhancing naive backtrack with the cycle-cutset method employing DAC and TT2 respectively. Each point in the

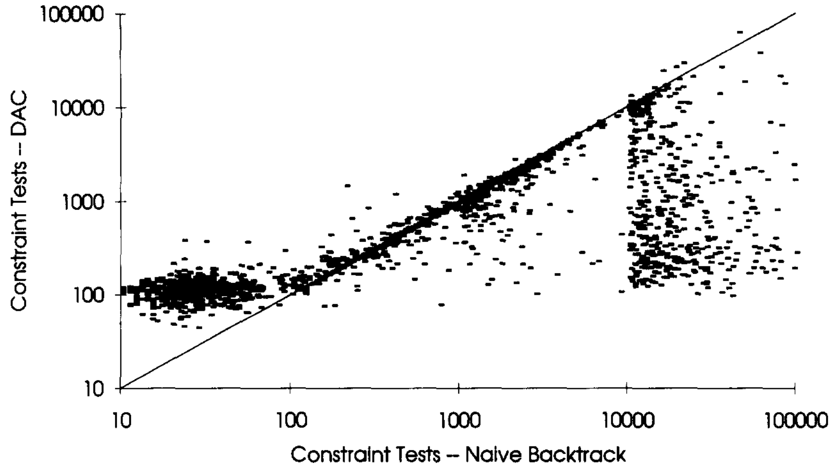

Fig. 9. Effects of cycle-cutset enhancement using DAC on randomly generated problems.

figures represents a problem instance such that its position on the horizontal axis reflects the number of constraint tests required to solve the instance using naive backtrack, and its position on the vertical axis reflects the number of constraint tests required to solve the instance with a cycle-cutset enhanced version. Points falling below the diagonal are instances for which the cycle-cutset method improved performance. Points lying exactly on the diagonal are most likely instances for which backtrack failed to advance beyond the variables within the cutset.

We intentionally selected problems such that exactly 500 points exist in each horizontal axis range $[10^x, 10^{x+1})$ in order to best demonstrate the effects of

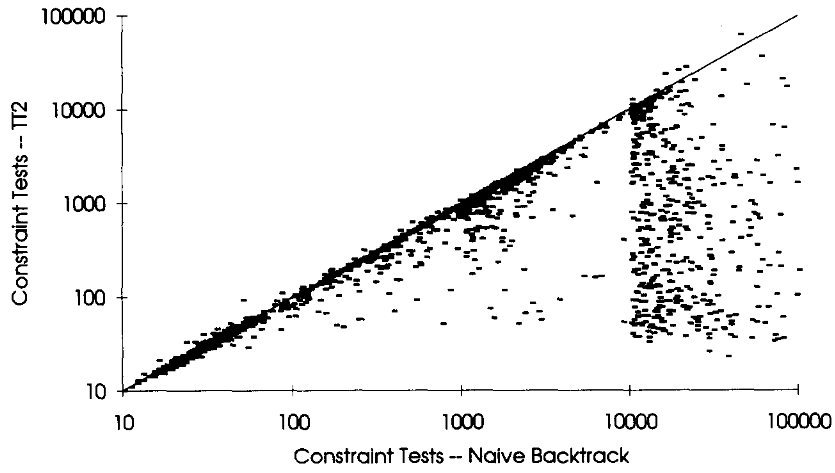

Fig. 10. Effects of cycle-cutset enhancement using TT2 on randomly generated problems.

cycle-cutset enhancement across all problem difficulties. From each such range the instances were chosen at random. Both figures plot the same set of instances chosen from the $n = 10$, $k = 10$ category.

The figures show that on average the cycle-cutset method yields substantial performance improvements, whether using DAC or TT2. Note, however, that for the easiest problems, the DAC enhanced algorithm is much slower than the unenhanced version. Even for more difficult classes, a considerable number of instances were solved faster by the unenhanced algorithm than the DAC enhanced algorithm. The TT2 enhanced version has the desirable property that the performance of the enhanced algorithm is infrequently worse than that of the unenhanced version, and when it is, the degradation is usually small.

3.2.2. *Mean speedups*

The speedup one can expect from utilizing the cycle-cutset method depends not only on the difficulty of the problem being solved and the tree algorithm employed, but on the size of the cutset as well. In order to quantify mean speedups when enhancing naive backtrack, we analyzed our data to determine the average speedup resulting from the cycle-cutset method given a particular tree algorithm, and a problem of a particular difficulty and cutset size. Tables 1–3 summarize these results. Each cell was calculated by averaging the speedups resulting from runs on over 1000 problem instances which were generated at random as described previously. Difficulty is defined as the number of constraint tests required by naive backtrack to solve the instance. Speedup is the number of constraint tests performed by naive backtrack divided by the number of tests performed by a cycle-cutset enhanced version. Instances on which the enhanced algorithm performs better produce a speedup of greater than one.

In general, we see that the smaller the cutset or the more difficult the problem, the better the mean speedup resulting from cycle-cutset enhancement employing

Table 1
Mean speedups from enhancing naive backtrack with the cycle-cutset method on randomly generated problems when $n = 10$ and $k = 10$

Problem difficulty	Tree algorithm	Cutset size						
		1	2	3	4	5	6	7
$[10, 10^2)$	TT2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	DAC	0.15	0.21	0.25	0.29	0.34	0.40	0.46
$[10^2, 10^3)$	TT2	1.8	1.3	1.2	1.2	1.1	1.1	1.0
	DAC	1.0	1.0	1.0	1.0	1.0	1.0	1.0
$[10^3, 10^4)$	TT2	19	8.5	4.0	2.2	1.5	1.2	1.0
	DAC	7.2	4.6	1.8	1.8	1.3	1.1	1.0
$[10^4, 10^5)$	TT2	500	320	180	78	26	6.4	2.2
	DAC	140	130	95	54	22	6.1	2.1

Table 2
Mean speedups from enhancing naive backtrack with the cycle-cutset method on randomly generated problems when $n = 10$ and $k = 20$

Problem difficulty	Tree algorithm	Cutset size						
		1	2	3	4	5	6	7
$[10, 10^2)$	TT2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	DAC	0.082	0.12	0.14	0.18	0.21	0.26	0.31
$[10^2, 10^3)$	TT2	1.5	1.3	1.3	1.2	1.1	1.1	1.0
	DAC	0.60	0.76	0.85	0.88	0.90	0.92	0.92
$[10^3, 10^4)$	TT2	4.5	2.3	1.8	1.5	1.2	1.1	1.0
	DAC	1.6	1.4	1.4	1.3	1.1	1.1	1.0
$[10^4, 10^5)$	TT2	180	120	44	8.4	1.9	1.2	1.1
	DAC	41	40	20	5.2	1.7	1.2	1.1

either TT2 or DAC. Note that only TT2 exhibits speedups greater than or equal to one for almost every problem class investigated. Furthermore, for all problem classes, the TT2 enhanced algorithm exhibits equal or better mean speedups than the DAC enhanced algorithm. The tables also reveal that increasing domain size or number of variables does not significantly affect the improvements one can expect from the TT2 enhanced algorithm, while the performance of the DAC enhanced algorithm on easier problems degrades noticeably with the larger domain size.

Table 3
Mean speedups from enhancing naive backtrack with the cycle-cutset method on randomly generated problems when $n = 20$ and $k = 10$

Problem difficulty	Tree algorithm	Cutset size						
		4–5	6–7	8–9	10–11	12–13	14–15	16–17
$[10, 10^2)$	TT2	1.0	0.99	1.0	1.0	1.0	1.0	1.0
	DAC	0.18	0.22	0.26	0.30	0.34	0.40	0.48
$[10^2, 10^3)$	TT2	1.2	1.1	1.1	1.1	1.0	1.0	1.0
	DAC	0.90	0.91	0.91	0.90	0.87	0.87	0.88
$[10^3, 10^4)$	TT2	5.5	3.3	2.3	1.7	1.3	1.2	1.0
	DAC	2.9	2.2	1.8	1.3	1.2	1.1	1.0
$[10^4, 10^5)$	TT2	79	42	18	5.4	2.2	1.3	1.1
	DAC	36	23	11	4.2	2.9	1.2	1.1
$[10^5, 10^6)$	TT2	1300	780	360	93	11	2.0	1.1
	DAC	500	390	210	67	10	2.0	1.1

3.2.3. Summary

The experimental results show that the cycle-cutset method is more effective when employing TT2 instead of DAC. In particular, the chance of a cycle-cutset enhanced naive backtrack algorithm performing worse than the unenhanced algorithm is significantly reduced when employing TT2. Furthermore, when the TT2 enhanced algorithm does performs worse, it is usually by only a small amount. The use of TT2 through the cycle-cutset method yields better mean improvements in performance than when using DAC in almost every problem class investigated, and the enhancement remains effective when increasing either domain size or the number of variables.

4. Conclusions

This paper presents and evaluates TreeTracker-2, an optimal backtrack algorithm for tree-structured constraint satisfaction problems that avoids expensive preprocessing phases performed by previous algorithms. TreeTracker-1, a near-optimal but simpler algorithm is presented first, and experiments demonstrate that the two algorithms perform almost identically. Experiments also reveal that the TreeTracker algorithms are faster on average than DAC, the previous algorithm of choice for solving tree-structured problems, and that the amount of improvement increases with problem size.

The efficiency of the TreeTracker algorithms enables greater effectiveness of enhancement schemes exploiting the easiness of tree-structured problems such as the cycle-cutset method. For all problem classes investigated, a naive backtrack algorithm enhanced with the cycle-cutset method employing TreeTracker-2 yields better mean improvements than when employing DAC. Furthermore, given a randomly chosen problem instance, the chance of a degradation in performance when using the TreeTracker enhanced naive backtrack algorithm over the unenhanced algorithm is small, whereas the DAC enhanced algorithm quite often performs worse than the unenhanced algorithm, especially on easy problems.

We did not re-evaluate the enhancement scheme presented in [6] which counts all solutions to tree-structured approximations of the original problem and uses the results to guide backtrack in ordering value assignments. While the TreeTracker algorithms we present here find only first solutions to tree-structured problems, it is not difficult to extend them to generate or count all solutions. It would be interesting to determine whether the appropriately modified TreeTracker algorithms would improve this enhancement scheme as well.

Lastly, we suspect that the technique involving the identification of good domain values during backtrack can provide benefits to CSP algorithms which are not restricted to tree-structured problems. We hope that this or some similar enhancement will encourage the development of backtracking algorithms that rarely perform substantially worse than unenhanced backtrack on any problem instance, while running in worst-case times equivalent or comparable to those of the best known preprocessing-based schemes.

Appendix A

Lemma A.1. *For any integers n and k such that* $1 \leqslant k < n$, $n \lg n \geqslant n \lg k + n - k$.

We prove this through the equivalent lemma:

Lemma A.2. *For any integers n and k such that* $1 \leqslant k < n$, $n^n \geqslant k^n 2^{n-k}$.

Proof (by induction on n).

Basis. For $n = 2$, $2^2 \geqslant 1^2 \cdot 2$ clearly holds.

Inductive step. Supposing the lemma holds for all $n < m$, we show it also holds when $n = m$.

$$\begin{aligned} m^m &= m^{m-1} m \\ &\geqslant (m-1)^{m-1} \cdot 2m \quad \text{(since } m > 1) \\ &\geqslant k^{m-1} 2^{m-1-k} \cdot 2m \\ &\quad \text{(for integers } k \text{ where } 1 \leqslant k < m-1 \qquad \text{(inductive hypothesis))} \\ &\geqslant k^m 2^{m-k}. \end{aligned}$$

Lastly, consider the case where $k = m - 1$: $m^m \geqslant (m-1)^m \cdot 2$, which must hold since $m > 1$. □

References

[1] C. Bessière and M. Cordier, Arc-consistency and arc-consistency again, in: *Proceedings AAAI-93*, Washington, DC (1993) 108–113.

[2] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms* (MIT Press, Cambridge, MA, 1990).

[3] R. Dechter, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, *Artif. Intell.* **41** (3) (1990) 273–312.

[4] R. Dechter, Constraint networks, in: S. Shapiro, ed., *Encyclopedia of Artificial Intelligence* (Wiley, New York, 2nd ed., 1992) 276–285.

[5] R. Dechter and I. Meiri, Experimental evaluation of preprocessing algorithms for constraint satisfaction problems, *Artif. Intell.* **68** (1994) 211–241.

[6] R. Dechter and J. Pearl, Network-based heuristics for constraint-satisfaction problems, *Artif. Intell.* **34** (1987) 1–38.

[7] R. Dechter and J. Pearl, The cycle-cutset method for improving search performance in AI applications, in: *Proceedings 3rd IEEE on AI Applications*, Orlando, FL (1987) 224–230.

[8] E.C. Freuder, A sufficient condition for backtrack-free search, *J. ACM* **29** (1) (1982) 24–32.

[9] J. Gashnig, Performance measurement and analysis of certain search algorithms, Technical Report CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, PA (1979).

[10] B. Nadel, Tree search and arc consistency in constraint-satisfaction algorithms, in: L. Kan and V. Kumar, eds., *Search in Artificial Intelligence* (Springer-Verlag, New York, 1988) 287–342.

[11] A.K. Mackworth, Consistency in networks of relations, *Artif. Intell.* **8** (1977) 99–118.

[12] R. Mohr and T.C. Henderson, Arc and path consistency revisited, *Artif. Intell.* **28** (2) (1986) 225–233.

[13] E. Tsang, *Foundations of Constraint Satisfaction* (Academic Press, San Diego, CA, 1993).