Partial Constraint Satisfaction *

Eugene C. Freuder and Richard J. Wallace

Computer Science Department, Kingsbury Hall, University of New Hampshire, Durham, NH 03824, USA

Abstract. A constraint satisfaction problem involves finding values for variables subject to constraints on which combinations of values are allowed. In some cases it may be impossible or impractical to solve these problems completely. We may seek to partially solve the problem, in particular by satisfying a maximal number of constraints. Standard backtracking and local consistency techniques for solving constraint satisfaction problems can be adapted to cope with, and take advantage of, the differences between partial and complete constraint satisfaction. Extensive experimentation on maximal satisfaction problems illuminates the relative and absolute effectiveness of these methods. A general model of partial constraint satisfaction is proposed.

1 Introduction

Constraint satisfaction involves finding values for problem variables subject to constraints on acceptable combinations of values. Constraint satisfaction has wide application in artificial intelligence, in areas ranging from temporal reasoning to machine vision. Partial constraint satisfaction involves finding values for a subset of the variables that satisfy a subset of the constraints. Viewed another way, we are willing to “weaken” some of the constraints to permit additional acceptable value combinations. Partial constraint satisfaction problems arise in several contexts:

- The problem is overconstrained and admits of no complete solution.
- The problem is too difficult to solve completely but we are willing to settle for a “good enough” solution.
- We are seeking the best solution obtainable within fixed resource bounds.
- Real time demands require an “anytime algorithm”, which can report some partial solution almost immediately, improving on it if and when time allows.

The utility of some form of partial constraint satisfaction has been repeatedly recognized. A variety of applications has motivated a variety of approaches. As

* This paper is reprinted (with minor changes) from Artificial Intelligence, volume 58, numbers 1-3, E. C. Freuder and R. J. Wallace, Partial constraint satisfaction, pages 21-70, 1992 with kind permission from Elsevier Science - NL, Sara Burgerhartstraat 25, 1055 KV Amsterdam, The Netherlands. (This volume was a special issue that was reprinted as an MIT Press book, Constraint-Based Reasoning.)
AI increasingly confronts real world problems, in expert systems and robotics, for example, we are increasingly likely to encounter situations where, rather than searching for a solution to a problem, we must, in a sense, search for a problem we can solve.

Conflicting constraints have arisen in a variety of domains. Descotte and Latombe made “compromises” among antagonist constraints in a planner for machining problems [10]. Borning used constraint “hierarchies” to deal with situations in which a set of requirements and preferences for the graphical display of a physical simulation cannot all be satisfied [2]; these hierarchies have been imbedded in a constraint logic programming language [3]. Hower used “sensitive relaxation” to resolve conflicts in floor planning [20].


Machine vision has also provided motivation for work on partial constraint satisfaction. Shapiro and Haralick [35] treated inexact matching of structural descriptions using an extension of constraint satisfaction that they called the inexact consistent labeling problem, which sought a solution within a given error bound. Mohr and Masini [31] suggested a modification of local consistency processing to deal with errors, permitting values to fail to satisfy some constraints, in order to cope with noise in domains such as computer vision; Cooper [5] defined an alternative generalization of constraint satisfaction with errors.

The related problem of approximate constraint satisfaction, where weights are assigned to individual combinations of values [34] [36] is motivated by machine vision as well. The expression of preferences in database queries is also a related problem [24]. Note that the concept of optimization can play a role in constraint satisfaction problems even when all constraints are satisfied; there may be an additional criterion to optimize among alternative solutions [5] [7] [37].

Most of this paper will focus on methods for maximal constraint satisfaction, where we seek a solution that satisfies as many constraints as possible. We have systematically reviewed the basic backtracking and local consistency methods for constraint satisfaction [23] [29] [32], and developed analogous methods for maximal satisfaction. The maximal satisfaction context has provided new challenges and new opportunities. The algorithms we formulated were subjected to carefully designed experiments that shed light on both relative and absolute performance as a function of basic structural problem parameters.

Our algorithms also allow for sufficient satisfaction, where we terminate the search if we find a solution which is sufficiently good, in the sense that the number of constraint violations does not exceed some predetermined bound. Our methods easily extend to resource-bounded satisfaction, where we report the best
solution in hand when a resource bound has been reached, and naturally support anytime algorithms.

Maximal satisfaction provides a form of optimization. Sufficient satisfaction incorporates a concept of acceptable error. Our methods clearly generalize to the use of more complex metrics to evaluate proposed solutions than a simple count of the number of violated constraints. In this manner, preferences can be introduced to distinguish among conflicting constraints. However, the simple metric of counting constraint violations facilitates the presentation of our algorithms, and provides a suitable context for an initial evaluation of their performance.

At the end of the paper we develop a still more general model of partial constraint satisfaction [14], in which we compare alternative problems rather than alternative solutions. We suggest viewing partial satisfaction of a problem, $P$, as a search through a space of alternative problems for a solvable problem “close enough” to $P$. We argue that a full theory of partial satisfaction should consider not merely how a partial solution requires us to violate or vitiate constraints, but how the entire solution set of the problem with these altered constraints differs from the solution set of the problem with which we started.

In this paper we will use for pedagogical purposes a simple, toy problem involving a fashion conscious robot seeking to choose matching clothes while getting dressed in the morning. (This could be regarded more seriously as a simple version of a configuration problem [30].) The problem is pictured in Fig. 1. Our robot has a minimal wardrobe: sneakers or Cordovans for footwear, a white and a dark green shirt and three pairs of slacks: denim, dress blue and dress gray. The robot has been told that: the sneakers only go with the denim slacks; the Cordovans only go with the gray slacks and the white shirt; the white shirt will go with either denim or blue slacks; the green shirt only goes with the gray slacks. These are the constraints under which it has to operate.

Section 2 discusses methods for achieving maximal constraint satisfaction. Branch and bound for maximal constraint satisfaction is the natural extension of backtracking for constraint satisfaction. Retrospective and prospective back-
track techniques for constraint satisfaction are shown to have analogues in a branch and bound setting for maximal satisfaction. Local consistency methods for constraint satisfaction have analogues in maximal satisfaction methods. Ordering techniques are if anything likely to be more important for branch and bound than for backtracking.

Section 3 describes extensive testing of maximal constraint satisfaction methods corresponding to several of the most successful constraint satisfaction methods. The results demonstrate the effectiveness of the maximal satisfaction analogues. They also illustrate the importance of taking advantage of the additional information available in the partial satisfaction domain, where the world is not just black and white (consistency or inconsistency), but shades of grey. Furthermore, the experimental design permits insights into the relationship between problem structure and the performance of the different methods.

Section 4 generalizes to other forms of partial satisfaction. Other metrics are briefly discussed. Partially ordered problem spaces are introduced. A partial constraint satisfaction problem is defined as a search through a space of alternative problems. Section 5 contains brief concluding remarks.

2 Methods

2.1 Introduction

A constraint satisfaction problem (CSP) involves a set of problem variables, a domain of potential values for each variable, and a set of constraints, specifying which combinations of values are acceptable. A solution specifies an assignment of a value to each variable that does not violate any of the constraints. We will consider here binary, finite CSPs where the constraints only involve two variables at a time, and the domains are finite sets of values. A constraint can therefore be represented explicitly as a set of permitted pairs of values. (If all pairs of values are allowed between two variables, then there is effectively no constraint between them; we will say that these variables do not share a constraint.)

For our running example: the variables are shoes, slacks and shirt; the values for shoes are Cordovans and sneakers; the constraint between shoes and shirt specifies that the only allowable combination of values is Cordovans and white shirt. Two values, like Cordovans and white shirt, that satisfy the relevant constraint, are consistent. A pair of values that violates a constraint is an inconsistency.

For now we define a partial constraint satisfaction problem (PCSP) as a CSP where we are willing to accept a solution that violates some of the constraints. A more formal approach to PCSPs is developed in Sect. 4.

Backtracking [18] is the classic algorithm for solving CSPs. A number of variations and refinements of backtracking have been developed. Several algorithms, including classical backtracking itself, utilize retrospective techniques, in which a new value selected to try to extend an incomplete solution is tested by “looking back” over the previously chosen values in the incomplete solution, to see if the
new value is consistent with the previously chosen values. By “remembering” more about the course of the search process, some variations reduce redundant testing. Other algorithms employ prospective strategies. Values are tested against the domains of variables that are not yet represented in the incomplete solution, so that inconsistencies can be dealt with before values from these domains are considered for inclusion. Ordering techniques have been used to direct the order in which variables, values or constraints are considered during search.

Our strategy in studying PCSP algorithms was to look for analogues of successful CSP techniques, focusing on backtrack and its variations. Branch and bound [25] [33] [37] is a widely used optimization technique that may be viewed as a variation on backtracking. Thus it was a natural choice in seeking an analogue of backtracking to find optimal partial solutions for PCSPs.

We begin by applying branch and bound to constraint satisfaction problems. Then we set about finding analogues of various CSP retrospective, prospective and ordering techniques, for partial, specifically maximal, constraint satisfaction algorithms. Finding appropriate analogues presents both challenges and opportunities. In presenting these algorithms we will generally begin with a review of the CSP version, then move on to the PCSP version, highlighting the differences. We will present examples and discussions as well as the algorithms themselves.

2.2 Retrospective techniques

2.2.1. Basic branch and bound. Branch and bound for maximal constraint satisfaction is the natural analogue of backtracking for constraint satisfaction. First we will briefly review backtrack search in the context of our running example. Backtrack search will find no solutions; the problem is overconstrained. Then we will use a basic branch and bound algorithm to find a way to dress our robot while violating a minimal number of its esthetic requirements.

![Fig. 2. Backtracking example.](image)

A depth-first traversal of the tree in Fig. 2 traces the progress of a backtrack
search on our running example. First we try Cordovans for shoes, then denims for slacks. According to the given constraints, denim slacks are not consistent with Cordovans, so we try blue slacks. These do not work so we try gray. Gray is good; we can move on to try and find a consistent shirt. The green shirt does not go with the Cordovans so we try the white shirt. It is consistent with the Cordovans but not with the gray slacks (we will assume that consistency is checked “top down” against the already chosen values). At this point we need to back up. We find that we have tried all the values for slacks, so we back up further to try another value for shoes. Ultimately all possibilities fail.

Backtrack search tries all value combinations exhaustively if necessary, but can avoid considering some combinations, by observing that a subset of values cannot be extended to a full solution, thus pruning a subtree of the search space. A standard measure of effort for CSP algorithms is the number of constraint checks. A constraint check occurs every time we ask a basic question of the form: is value $a$ for variable $X$ consistent with value $b$ for variable $Y$? For example, when we ask whether Cordovans are consistent with denims, that is a constraint check. The total number of constraint checks (cc) accumulated when search has reached each leaf node of the search tree is shown in the figure. In total the search required 11 constraint checks to find that there is no solution.

We will be discussing the way different algorithms work through a search tree like this, and will need a suitable vocabulary. We will talk about the levels in the search tree; each level in the search tree corresponds to a problem variable. We will assume the levels are numbered from the top down; higher levels have smaller numbers. A shirt value will be found at a lower level or deeper in the search tree than a slacks value.

The branching corresponds to variable values, e.g. the choice of denim for slacks. The nodes in the search tree represent assignments of values to variables during the search. The green shirt value is always considered at the same level; however it is considered twice in this search. The second time the green shirt is encountered, the search has backed up in the interim to the shoe level. The set of assigned values along a branch of the tree, from the top down to some level, e.g. (sneakers denims) is a search path. The search path leading down to the most recently chosen value is the current search path. It represents the current set of choices of values for variables. It represents a proposed, incomplete solution, unless it includes values for all the variables.

One theme that will recur in deriving our PCSP analogues is the different definition of local failure during CSP and PCSP search. A CSP search path fails as soon as a single inconsistency is encountered. A PCSP search path will not fail until enough inconsistencies accumulate to reach a cutoff bound. Retrospective techniques excel at determining the inconsistencies implied by past choices. Prospective techniques excel at estimating the inconsistencies implied by future choices.

Branch and bound operates in a similar fashion to backtracking in a context where we are seeking a maximal solution, one which satisfies as many constraints as possible. Branch and bound basically keeps track of the best solution found so
far and abandons a line of search when it becomes clear that it cannot lead to a better solution. A version of backtracking that searches for all solutions, rather than the first solution, most naturally compares with the branch and bound extension to find a maximal solution.

Figure 3 traces a branch and bound search for a maximally satisfying solution for our sample problem. Branch and bound is applied in this context by using as an evaluation function a count of the number of violated constraints, or inconsistencies. Where the backtrack search, looking for a perfect solution, that violates no constraints, said that denims were inconsistent with Cordovans and proceeded on to blue slacks, branch and bound, looking for a maximal partial solution, observes that any partial solution containing Cordovans and denims will violate at least one constraint, and proceeds to consider shirts.

Specifically it is noted that any partial solution containing Cordovans and denims will be at a distance, \( d \), of (at least) 1 from a perfect solution. Distance measures the number of constraints violated by the chosen values. By the time we add a green shirt we have violated three constraints. We say that green shirt is an extension of the search path (Cordovans, denims). We will talk about extending a search path by adding one or more values, and, in particular, extending a search path to a complete solution, that contains a value for each variable. The search path leading from Cordovans, through denims, to green shirt now contains three inconsistencies, giving it an associated distance (from a perfect solution) of three. The first inconsistent value in the search path is denims. It is the first value in the search path to be inconsistent with another value in the search path. The last inconsistent value is green shirt.

Cordovans, denims and a green shirt provide a partial solution at a distance of 3 from a perfect solution. This distance is taken as the value of \( N \). \( N \) is used during the search to store the number of inconsistencies in the best solution found up to that point in the search. \( N \) is a necessary bound, which we will often
simply refer to as the bound, in the sense that to do better it is necessary to find a solution with fewer inconsistencies.

The necessary bound $N$ can be set initially based on a priori knowledge that a solution is available that violates fewer than $N$ constraints, or an a priori requirement that we are not interested in solutions that violate more than $N - 1$ constraints. As branch and bound proceeds, if a solution is found that violates $N' < N$ constraints, $N$ is replaced by $N'$.

As the branch and bound search proceeds in our example, it finds a better partial solution, with a single constraint violation (Cordovans, denims, white shirt). This updates $N$ to be 1. Now when it tries Cordovans and blue slacks (hoping that an even better, in this case perfect, solution is to be found), it recognizes that any solution involving Cordovans and blue slacks can be no better than the solution already found. Thus it does not consider matching a shirt to the Cordovans and blue slacks, but proceeds immediately to try gray slacks. As with backtrack search, the basic idea of recognizing defeat early permits pruning of the search space.

Search concludes when we find a perfect solution, not available in this case, or run out of things to try. We could also quit when we reach a preset sufficient bound $S$, which specifies that we will be satisfied if we find a partial solution that violates no more than $S$ constraints. We may know, for example, that no exact solution is possible and thus be able to set $S$ to 1. We may be willing to settle for a “close enough” or sufficient solution. Obviously the larger we set $S$ the easier the problem is likely to be.

Circumstances may also impose resource bounds. In particular, real time processing may require immediate answers, that can be refined later if time allows. The branch and bound process is well suited to providing resource-bounded solutions. We can simply report the best solution available when, for example, a time bound is exceeded. The branch and bound process is also clearly well-suited to support an anytime algorithm, which can repeatedly provide a “best-so-far” answer when queried. It can quickly provide some answer, with a better one perhaps to follow as time allows.

Figure 4 provides a basic branch and bound algorithm for maximal constraint satisfaction. It also provides for a priori sufficient and necessary bounds, $S$ and $N$, on an acceptable solution. If there are no such a priori bounds, $S$ is initially 0 and $N$ “infinity”. The parameters of the P-BB procedure appear in several other algorithms in this paper. The parameter Search-path carries the current search path. Distance carries the number of constraints already violated by the values on the current search path, the number of inconsistencies in the proposed, incomplete solution. Variables carries a list of the variables not assigned values in the current search path, the variables at lower levels in the search tree. Values carries a list of the values not previously tried as extensions of the current search path; the first value in Values is the next value that can be tried as an instantiation of the first variable in Variables.

In this and several subsequent algorithms we will be employing $N$, $S$, and Best-solution as global variables containing the necessary and sufficient bounds.
P-BB\(\text{Search-path, Distance, Variables, Values}\)

if Variables = nil then \{ all problem variables have been assigned values in Search-path \}
  Best-solution ← Search-path
  N ← Distance
  if \(N < S\) then return 'finished' \{ Best-solution is sufficiently close \}
  else return 'keep-searching'
else if Values = nil then \{ tried all values for extending search path \}
  return 'keep-searching' \{ so will back up to see if can try another value for the last variable assigned a value in Search-path \}
else if Distance = N then \{ already extended Search-path to assign values for remaining variables without violating any additional constraints \}
  return 'keep-searching' \{ so will see if can do better by backing up to try another value for the last variable assigned a value in Search-path \}
else \{ try to extend Search-path \}
  Current-value ← first value in Values
  New-distance ← Distance
  try choices in Search-path, from first to last, as as New-distance < N:
    if the choice is inconsistent with Current-value then
      New-distance ← New-distance + 1
    if New-distance < N
      P-BB\(\text{Search-path plus Current-value, New-distance, Variables minus the first variable, values of second variable in Variables}\)
      = 'finished' then return 'finished' \{ Search-path was extended to sufficient solution \}
      else \{ will see if can do better with another value \}
      return P-BB\(\text{Search-path, Distance, Variables, Values minus Current-value}\)

Fig. 4. Branch and bound algorithm.

and the best solution found so far. Other variables in all the algorithms are local, with the exception of some backmark arrays as indicated in Sect. 2.2.3.

The basic recursive structure of P-BB is also common to many other algorithms in this paper. P-BB works sideways in the search tree by recursing through a set of values for a variable, and deeper into the search tree by recursing through the variables. Backing up is implemented through the unwinding of the recursion.

In this algorithm as in all retrospective procedures, a value being considered for inclusion in the solution is compared with values already chosen, to determine whether constraints between the instantiated variables and the current one are satisfied. Each comparison of two values is a constraint check.

Since the total number of constraint checks is a standard measure of CSP algorithm efficiency, we wish to minimize this quantity. To this end, the new
distance is compared with \( N \) after each constraint failure, so that if the bound is reached, the present value is not checked further. A subtle point involves the test to see if the Distance is already \( N \) before trying a new value, \( v \) for \( V \). (Actually our implementation checked for Distance \( \geq N \), but an equality test appears sufficient.) One might wonder, if the number of inconsistencies among the already chosen values stored in Search-path equals the bound, what is the algorithm doing trying to extend Search-path to another variable, \( V' \)? However, when the algorithm began trying to extend the solution, \( N \) may have been larger. A complete assignment of values to variables, extending Search-path and requiring only the current \( N \) constraint violations, may have been found in the interim, using another value for \( V' \), before reaching \( v \).

We have chosen here a depth-first implementation of the branch and bound paradigm. Other branch and bound control structures, notably a best-first approach, are possible. Depth-first is the most direct analogue of backtrack and as such facilitates the development of analogues of backtrack variations. Depth-first also supports an anytime algorithm that will almost immediately have a “best so far” solution to report. (Limited experimentation with a best-first approach was not encouraging with respect to its efficiency, but this should not preclude further study.)

In Fig. 3 the total number of constraint checks (cc) accumulated by the time each leaf node of the search tree has been processed is given at the bottom of the figure. Note that, due to our procedures for minimizing constraint checks, some checks are avoided at many points of the search tree, including at some of the lowest level nodes. In some cases subtrees are pruned; in some cases a value does not need to be checked against the entire preceding search path.

The general worst case bound for this algorithm is of course exponential. However, it is no worse than that for a backtrack algorithm for finding a perfect solution. Both, in the worst case, will end up trying all possible combinations of values, and testing all the constraints among them. On the other hand, the exponential worst case bound is bad enough. We want to consider techniques that may help to avoid achieving that bound. As we have indicated, our strategy is to look for analogues of methods which have already proven successful for finding perfect solutions.

2.2.2. Backjumping. Backjumping [17] remembers information about previous failures to reduce the need for redundant constraint checks to rediscover them. Consider what happens when we test shirts in the branch of the backtrack search tree that begins with sneakers and denims (Fig. 2). Both shirts fail immediately upon being tested against sneakers; there is no need to see if they are consistent with denims. Yet classical, so-called “chronological”, backtracking blindly backs up and tries dress blue slacks. There is no point in doing so; even if the blue slacks went with the sneakers, we would obviously fail again when we reached the shirt level. Backjumping recognizes this, and after the shirts fail to match the sneakers it immediately backtracks to the shoe level. There it tries to consider another type of shoe; since there is none, search terminates. The final
two constraint checks have been avoided. Backjumping generalizes this insight.

We need to recognize that all the values tried for a given variable may not fail against the same previous value. When processing a variable, backjumping remembers the deepest level, \( l \), in the search tree at which any of the values fails. When all the values have been discarded, backtracking can proceed directly to this level. Actually, the algorithm does not “jump” directly to this level. As the recursive calls unwind, they return the depth \( l \); the recursion unwinds until level \( l \) is reached before trying to consider any more values. As we extend search paths we ultimately must reach levels where all value choices for extending the search path do fail to be consistent with previous choices, or else we successfully proceed all the way down to a solution. Of course, the deepest level of failure may simply be the previous level, so no real jumping back need result.

Fig. 5. Backjumping example.

In the backjumping analogue for partial constraint satisfaction, failure does not necessarily occur when an inconsistency is found. Failure occurs only when an inconsistency pushes us too far away from the perfect solution, i.e. when the accumulated number of inconsistencies reaches the necessary bound, \( N \). For example, if search has reached the seventh variable in the sequence, and checking a value of its domain against the value for the fourth variable causes the distance to increase to the bound \( N \), the depth of failure of that variable is 4. Figure 5 shows a trace of backjumping for a maximal solution on our running example. The numbers on the arrows show some of the values returned by P-BJ; notice how the returned depth of 1 at the bottom right of the search tree supports a jump back to the shoes level without considering additional slacks.

Figure 6 contains a backjumping algorithm for partial satisfaction. Aside from the different definition of failure, there is one major difference from the conventional backjumping algorithm. We cannot always jump back all the way to the deepest level of failure. If any values below that level were inconsistent when chosen, i.e. required an increase in Distance when they were chosen, we
can only jump back to the level, \( l \), of the last, deepest, one of these inconsistent values. Otherwise, minimum distance solutions can be missed that are based on other values yet to be tried at level \( l \). This is because alternative values at level \( l \) may involve fewer inconsistencies, adding less to Distance, so that search can proceed from this level without encountering the bound at the same point in the search.

P-BJ(Search-path, Distance, Variables, Values, 
   Current-depth, Return-depth, Inconsistency-depth)
   if Variables = nil then [have new best solution]
      Best-solution ← Search-path
      N ← Distance
   if \( N \leq S \) then return ‘finished’ else return Current-depth – 1
   else if Values = nil or Distance = N then
      return Return-depth {may lead to backjumping}
   else {try to extend Search-path}
      Current-value ← first value in Values
      New-distance ← Distance
   try choices in Search-path, first to last, until New-distance = N or tried all:
      if the choice is inconsistent with Current-value then
         New-distance ← New-distance + 1
         Fail-depth1 ← level of the choice
      if New-distance < N then Fail-depth1 ← Current-depth {did not fail}
      if New-distance < N and
         the value, Fail-depth2, of
         P-BJ(Search-path plus Current-value, New-distance,
            Variables minus the first variable,
            values of second variable in Variables,
            Current-depth + 1, 0,
            Current-depth if New-distance ≠ Distance else Inconsistency-depth),
         is = ‘finished’ {found a sufficient solution}
      or < Current-depth {can backjump!}
      then return Fail-depth2 {backup immediately}
   else {try another value}
      return P-BJ(Search-path, Distance, Variables,
         Values minus the first value,
         Current-depth,
         max(Fail-depth1, Return-depth, Inconsistency-depth),
         Inconsistency-depth)

Fig. 6. Backjumping algorithm.

As an example of this phenomenon, we adapt the matching clothes problem, supposing that there are other values that were tested before, giving a bound
of 2 (Fig. 7). We also change the order in which we examine the variables; the variable search order is indicated by numbers on the nodes. Each domain is checked from left to right, and vertical lines appear to the right of the values currently being considered. Current inconsistencies are indicated by the lines joining two nodes. Search has reached a dead end with the third variable (since the number of inconsistencies equals the bound), with the deepest level of failure (where the total number of inconsistencies became equal to the bound) at the level of variable 1. If this were a CSP, search would now jump back to variable 1 and the next value (gray slacks) would be selected. However, in the present problem the next value associated with variable 2 (white shirt) is compatible with the present value of variable 1 (blue slacks), so in this case no inconsistency is present between variables 1 and 2. Therefore, if search backs up to variable 2, a value can be chosen that leaves the distance at zero. In addition, the Cordovans (at variable 3) are compatible with the white shirt, but not with the blue pants, so it is possible to find a solution including the blue slacks and white shirt that gives a total distance of 1. Since $N$ is currently 2, this is a better solution. However, it would have been overlooked if we had followed the ordinary backjumping procedure used for CSPs.

![Diagram](image)

Fig. 7. Backjumping to the level of the last inconsistent choice.

To handle this situation, the backjumping analogue must keep track of the deepest level associated with an inconsistency (*Inconsistency-depth*), in addition to tracking the depth of failure. In the present implementation, *Inconsistency-depth* is passed along as search proceeds, and updated to *Current-depth*, the level at which we are currently trying to extend the search path, if there is a failure at this level of testing. Obviously, this addition to the procedure will tend to lessen the efficiency of this form of backjumping.

**2.2.3. Backmarking.** Backmarking [16] has the potential to avoid some redundant successful constraint checking, as well some redundant discoveries of
inconsistencies. When trying to extend a search path by choosing a value for a variable $V$, backmarking marks the individual level, $Mark$, in the search tree at which an inconsistency is detected for each value of $V$. For example, if value, $b$, for variable, $U$, is consistent with the first value in the search path, but inconsistent with the second value, the $Mark$ for value $b$ is 2. (If no inconsistency is detected for a value, its $Mark$ is set to the level above the level of the value.) Assuming we cannot successfully extend the search path to a complete solution, and have to back up from $V$, backmarking also remembers the highest level, $Backto$, to which search has backed up since the last time $V$ was considered. When backmarking next considers a value $v$, for $V$, the $Mark$ and $Backto$ levels can be compared. There are two cases:

1. $Mark < Backto$. If the level at which $v$ failed before is above the level to which we have backtracked, we know, without further constraint checking, that $v$ will fail again. The value it failed against is still there.

2. $Backto \leq Mark$. If since $v$ last failed we have backed up to or above the level at which $v$ encountered failure, we have to test $v$; however, we can start testing values against $v$ at level $Backto$. The values above that level are unchanged since we last—successfully—tested them against $v$.

Figure 8 contains a backmarking algorithm for partial constraint satisfaction. Again, for partial satisfaction failure for a value, $v$, does not necessarily occur at the first inconsistency. It occurs at a level, $Lastmark$, where the last inconsistency is found, that which causes us to reach the bound that terminates the search path. We call this level $Lastmark$, rather than $Mark$, because, as we shall see in a moment, we also need to keep track of the level $Firstmark$ where the first inconsistency with $v$ was found. Thus we mark a range, as opposed to a single failure point. As before we store the level, $Backto$, the highest level to which search has backed up since last trying to assign a value to $V$. $Firstmark$, $Lastmark$, $Backto$ and $Inconsistencies$ are arrays, and are global, with the rows associated with the variables and the columns (except for $Backto$) with values. $Firstmark$ elements are initialized to 1, $Lastmark$ and $Inconsistencies$ to 0, $Backto$ to 1.

Again, we have two cases:

1. $Lastmark < Backto$. If the level at which $v$ failed before is above the level to which we have backtracked, we again know, without further testing, that $v$ will fail again. All the values it failed against are still there.

2. $Backto \leq Lastmark$. If since $v$ last failed we have backed up to or above the level at which $v$ encountered failure, we have to test $v$. However, we cannot always start the new testing of values against $v$ at level $Backto$, as in the CSP case. In the CSP case we knew that there were no inconsistencies above the level of failure. Now we only know that there are no inconsistencies above the level where the first inconsistency was found. Thus there are two further cases:

   a. $Backto \leq Firstmark$. If we have backed up to or above the level where the first inconsistency was found, we know that the unchanged values
P-BMK(Search-path, Distance, Variables, Values, Current-depth, Value-index)
if Variables = nil then
    Best-solution ← Search-path
    N ← distance
    if N ≤ S then return ‘finished’
    else return ‘keep searching’
else if Values = nil then
    set array Backto, beginning with current variable, to Current-depth – 1
    return ‘keep searching’
else if ( (Lastmark[Current-depth, Value-index] ≥ Backto[Current-depth]
    {search has backed up to or above the level of the last inconsistency
    marked at the previous encounter with the first of Values}
    and
    adding to Distance the number of inconsistencies found between
    the first of Values and previous values in Search-path starting at level
    min(Backto[Current-depth], Firstmark[Current-depth, Value-index])
    (updating Arrays appropriately)
    produces a New-distance < N)
    {adding the first of Values to the search path
    will not push the number of inconsistencies to the bound}
or
    (Distance + Inconsistencies[Current-depth, Value-index] < N
    {current distance + recorded inconsistencies is less than the bound}
    and
    adding to Distance + Inconsistencies[Current-depth, Value-index]
    the number of inconsistencies found between the first of Values
    and previous values in Search-path starting at level
    Lastmark[Current-depth, Value-index] + 1
    {updating Arrays appropriately}
    produces a New-distance < N )
    {adding the first of Values to the search path
    will not push the number of inconsistencies to the bound}
    and
    P-BMK(Search-path plus Current-value, New-distance,
    Variables minus the first variable,
    values of the second variable in Variables, Current-depth+1, 1)
    = ‘finished’
    then return ‘finished’
else return P-BMK(Search-path, Distance, Variables,
Values minus the first value,
Current-depth, Value-index+1)

Fig. 8. Backmarking algorithm.
above that level are still consistent with \( v \), and we can start the new
testing of \( v \) at level \( \text{Backto} \).

(b) \( \text{Firstmark} < \text{Backto} \). Otherwise we need to start testing at the level of
the first inconsistency, \( \text{Firstmark} \). Above that level, the unchanged values
will still be consistent with \( v \).

Actually the situation is even a bit subtler and more distinct from CSP
backjumping than we have let on. \( \text{Lastmark} \) may not in fact mark a failure
point at all. The previous time we considered \( v \), the last inconsistency may not
have pushed us over the bound (or as in the CSP case, we may have found no
inconsistency at all, in which case \( \text{Lastmark} \) is again set to the level above the
level of \( v \)). On the other hand, even if the inconsistencies with \( v \) did not induce
failure before, they may now, since the distance, the current number of other
known inconsistencies, as well as the bound \( N \), may have changed in the interim.

Accordingly we save another piece of information the number of \( \text{Inconsistencies} \) between \( v \) and the values in the search path down to level \( \text{Lastmark} \).

Case (1) above is really:

\[(1') \text{Lastmark} < \text{Backto}.\]

(a) If the current \( \text{Distance} \) plus \( \text{Inconsistencies} \) is not less than \( N \), we know that
\( v \) will fail again without further testing. The values which produced those
inconsistencies before are still present.

(b) Otherwise, add \( \text{Inconsistencies} \) to the current \( \text{Distance} \), and commence fur-
ther testing of \( v \) at the level below \( \text{Lastmark} \). The values which caused the
inconsistencies before are still there. (We feel that it may be possible to
commence testing at level \( \text{Backto} \), but subtle bookkeeping issues need to be
resolved.)

Several snapshots of the search with the backmarking analogue for the clothes
matching problem are shown in Fig. 9. Each copy of the arrays shows the values
following the portion of search diagrammed to its left; only changed values are
shown. For each array, rows associated with the first level of search are omitted
from the figure because their entries never change.

For example, consider the cells associated with the value green shirt after
the first portion of the search, which is represented at the far left of Fig. 9. The
green shirt value is associated with row three and column one in the arrays. The
value of \( \text{Firstmark}[3,1] \) is 1, since the green shirt does not match the Cordovans
at level one. The value of \( \text{Lastmark}[3,1] \) is 2, due to the mismatch between
the green shirt and the denims. Since there are two mismatches, or constraint
failures, associated with the green shirt, \( \text{Inconsistencies}[3,1] \) is 2.

Note that in the last column of arrays, the cells associated with the green
shirt and the white shirt (row three) all have a value of 1, since the comparisons
with sneakers at level one resulted in a constraint failure and this was sufficient
to attain the bound. In addition, \( \text{Backto}[3] \) has value 2. When search then proceeds
to the next value for the variable pants, (dress blue, not shown), there is no
failure, so the current distance is zero. However, before the green shirt is tested
again, the value of \( \text{Backto}[3] \) is found to be greater than \( \text{Lastmark}[3,1] \) —case (1') above— and the value of \( \text{Inconsistencies}[3,1] \) is 1 while \( N \) is also 1, so we are in case (1'a), and green shirt fails without any further constraint checking. The same situation is found with the white shirt. Search returns to level 2, where the gray pants are tested; these fail to match the sneakers and search ends with a total of 15 constraint checks.

The version of backmarking used in the experiments described in Sect. 4 used an enhancement to minimize the number of constraint checks (though it turned out not to make much difference). Under the second “else if” of the algorithm as described in Fig. 8, the distance was first checked against \( N \); if \( N \) had been reached or exceeded, there was no need for constraint checking. Unfortunately, this tactic entailed further checking after the last “else” and possible revision of the values of \( \text{Firstmark}, \text{Lastmark} \) and \( \text{Inconsistencies} \) if the bound had been reached and the algorithm had since backed up into the recorded range of constraint failures. For example, if the value in \( \text{Backto} \) was now between the corresponding values in \( \text{Firstmark} \) and \( \text{Lastmark} \), \( \text{Lastmark} \) was set to \( \text{Firstmark} \) and \( \text{Inconsistencies} \) set to 1.

Various refinements of PCSP backmarking are possible that have not yet been implemented. Observe that in our running example, the second time we encounter the green shirt, while the \( \text{Lastmark} \) and \( \text{Inconsistencies} \) machinery does not help us, the \( \text{Firstmark} \) value could. Since \( \text{Firstmark} \) occurs at level one, which is above the \( \text{Backto} \) level, and the bound is already down to one at this point, we could infer, without any further constraint checks, that using the green shirt will bring us to the bound. In fact, we could consider storing all levels at which values encountered inconsistencies, and using that additional information for further pruning without further constraint checking.
2.3 Prospective techniques

Prospective techniques “look ahead” to establish some form of local consistency before continuing the search for a global solution. Prospective techniques can prune from consideration values that do not meet local consistency criteria. Consistency techniques can be used as preprocessing methods prior to search (in some situations leaving little if any work for subsequent search). They can be interleaved with backtracking to form hybrid algorithms. There are also methods of applying local consistency techniques repeatedly to subproblems to achieve global solutions.

The most commonly used form of local consistency is arc consistency [26]. A problem is fully arc consistent if every value in the domain of every variable is consistent with, we also say supported by, at least one value in every other variable domain. Arc consistency preprocessing eliminates all unsupported values.

The most familiar hybrid algorithm is forward checking [19]. In forward checking, each assignment of a value, v, to a variable, V, is followed by a limited amount of arc consistency checking, in which the domains of variables that share a constraint with V are tested against v.

It is a standard branch and bound strategy (often discussed in terms of “lower bounds”) to increase pruning by estimating the implications of proceeding on from the current search point. Prospective methods provide a means of implementing this strategy for partial constraint satisfaction. Until now, we have been discontinuing a search path if the number of inconsistencies on that path, the distance \( D_i \), is not less than the bound \( N \). If we had some way of determining that no matter how we sought to continue that path we would encounter at least \( D_i + D_t \) additional inconsistencies, then we could discontinue the search if \( D + D_t \) failed to be less than \( N \). Prospective methods permit us to obtain such \( D_t \) values.

2.3.1. Pruning with arc consistency counts. In the CSP context, arc consistency permits us to eliminate values that arc consistency processing determines cannot participate in any complete solution. In our running example, sneakers can be eliminated because it is not consistent with any shirt. Furthermore the dark green shirt and the dress blue slacks can be eliminated because they are not consistent with any shoes. Notice further that although the denim slacks were originally supported by sneakers, now that sneakers have been eliminated there is no support for denims, and they can be eliminated in turn. Now there is no longer any support for the white shirt. Having eliminated both shirt possibilities arc consistency has, for this problem, discovered that there is no global solution, and no further search is necessary.

In the PCSP context, it is not possible to discard values in this way unless we have an initial value for the necessary bound, \( N \). If we did know, for example, a priori, that there was a solution that violated only one constraint, or that any solution that did violate more than one constraint was unacceptable, we could eliminate a value for a variable that was not consistent with any value for two other variables.
However, it is possible to perform prior calculations regarding the increments in distance associated with specific values. In particular, for each value, the number of domains with no supporting values can be tallied; this number, the arc consistency count, is a lower bound on the increment in distance that will be incurred if this value is added to the solution. An algorithm for computing arc consistency counts is given in Fig. 10.

In the course of subsequent search, the arc consistency count for a proposed search path extension, \( v \), can be added to the distance associated with the search path, and this sum compared with the current bound \( N \). If the sum is not less than \( N \), then we know that any complete solution starting with the current search path and involving \( v \) will violate too many constraints. We can fail at this point without testing \( v \) at all. Of course, \( v \) was tested once during the preprocessing; however, we may encounter \( v \) many times during subsequent search, and on a number of those occasions the preprocessing may save further testing of \( v \). The forward checking algorithm below uses a similar strategy involving a limited, dynamic form of arc consistency count.

P-ACC(Variables, Domains, Counts)
For all \( V_i \) belonging to Variables:
For all \( V_j \neq V_i \) such that there is a constraint between \( V_i \) and \( V_j \):
For each value, \( a \), in the domain of \( V_i \):
    if there is no value, \( b \), in the domain of \( V_j \) such that the pair \((a,b)\)
is allowed by the constraint between \( V_i \) and \( V_j \)
    then increment the count for value \( a \)

Fig. 10. Algorithm for computing arc consistency counts.

If arc consistency checking is used to tally the number of domains that do not support a value, then one constraint failure may be counted twice, once for each value that does not belong to an acceptable pair. This is not a problem during subsequent search because a consistency count is not incorporated into the distance except to check against the bound when first considering that value for extending the search path. (If the value is accepted, then adjustment of the distance through consistency checking is done in the same way as with basic branch and bound.) Suppose a particular value is included, and another value is considered afterwards whose arc consistency count depends in part on a constraint that also affected the former value’s arc consistency count. Since the current distance is based on retrospective checking, it does not yet include this constraint, so the second count can be used without modification in computing a projected distance to compare with the upper bound.

Arc consistency counts have the advantage that they need only be computed once, before search, and can be done in \( O(cd^2) \) time, where \( c \) is the number of
constraints and $d$ the maximum domain size for the variables. They can then be stored in an appropriate form for testing against the current distance. In contrast, both the retrospective and hybrid techniques require more extensive calculation to retain and update information related to distance for specific values.

Because no values are actually discarded, there is no propagation of failure in the manner of arc consistency algorithms for CSPs (e.g., where the removal of sneakers led in turn to the removal of denim). It may be possible to propagate counts in a manner analogous to ordinary constraint propagation. (Hybrid analogues based on this idea may also be possible and were suggested by Shapiro and Haralick [35] for inexact matching.) This might involve retaining information about the conditions of failure, employing conditional counts that can only be used if the supporting values are not used in the solution. In the arc consistency count algorithm, in contrast, which takes one pass through the variables, we are assured that the consistency counts are all unconditional.

2.3.2. Forward checking. Forward checking is a hybrid algorithm that uses a very limited amount of arc consistency checking. Each time a value, $v$, is assigned to a variable, $V$, the algorithm looks ahead to all the variables that currently have not been assigned a value, and that share a constraint with $V$, and removes from the domains of these variables any values inconsistent with $v$. For example, when Cordovans are proposed for shoes, the denim and dress blue slacks will be removed, and the green shirt.

If later we change our mind about $v$, the pruned values have to be restored. E.g., when we move on to consider sneakers, the denim and dress blue slacks and the green shirt must reappear. (In an implementation, recursion can handle the bookkeeping for the variable domains during backup.) Of course, since sneakers are not consistent with any shirt, forward checking with sneakers will reduce the shirt domain to the empty set, signaling a failure point.

Notice that despite the fact that it can be viewed as an integration of consistency processing and backtracking, forward checking really is almost the complement of standard backtracking. Standard backtracking checks a value for consistency against previously chosen values. With forward checking, when we propose a value we already know it is consistent with the previously chosen values (or else the consistency processing would already have pruned it away). Now we test it against the domains of the remaining, uninstantiated variables.

When used for partial satisfaction, forward checking is based on the same type of looking ahead as ordinary forward checking. However, again, the differing definition of failure comes into play. If there is an inconsistency, a value is not rejected unless the total number of currently chosen values with which it is inconsistent is at least as large as the difference between the current distance and the bound $N$. This means that the algorithm must dynamically keep track of the number of times a value has been found to be inconsistent with currently chosen values. This number, a form of dynamic arc consistency count, we will call the inconsistency count for a value.

An example of forward checking beginning to operate on the matching clothes
problem in shown in Fig. 11. In this figure, counts associated with the values
of variables which share a constraint with a variable, $V$, are shown at the point
in the search at which they are calculated. For example, when Cordovans are
chosen, the inconsistency count for green shirt becomes one; it increases to two
when denims is chosen.

\[
\begin{array}{c}
\text{Cordovans} \\
\text{d = 0} \\
\text{d = 1} \\
\text{denims} \\
\text{green} \\
\text{white} \\
\text{d = 3} \\
\text{N = 3} \\
\text{7 cc}
\end{array}
\begin{array}{c}
\text{dress blue} \\
\text{dress gray} \\
\text{green} \\
\text{white}
\end{array}
\begin{array}{c}
\text{green} \\
\text{white}
\end{array}
\begin{array}{c}
\text{green} \\
\text{white}
\end{array}
\begin{array}{c}
\text{Cordovans} \\
\text{d = 0} \\
\text{d = 1} \\
\text{d = 1} \\
\text{N = 1} \\
\text{7 cc}
\end{array}
\begin{array}{c}
\text{dress blue} \\
\text{dress gray} \\
\text{green} \\
\text{white}
\end{array}
\begin{array}{c}
\text{green} \\
\text{white}
\end{array}
\begin{array}{c}
\text{green} \\
\text{white}
\end{array}
\]

Fig. 11. Forward checking example.

Although we see more constraint checks performed in the early stages of
this example than we saw with straightforward branch and bound, the counts
derived from these early checks are used to avoid further constraint checks,
putting forward checking ahead at a later stage.

When a value, $v$, is proposed, its own inconsistency count can be added to
the current distance, and the total used in a manner similar to that proposed for
arc consistency counts in the previous section: if the total is equal to or greater
than the bound, $v$ fails immediately. For example, when Cordovans are tried
with blue slacks, the bound is already one (set by the Cordovans, denim slacks
and white shirt combination), and the inconsistency count of blue slacks is one
(set when Cordovans was chosen). This tells us that we cannot hope to choose
a shirt that will permit us to do any better than our current best solution. The
blue slacks fail, without any further testing.

Notice also that when gray slacks are tried with Cordovans, although they
are consistent, together they eliminate all the shirt values. Cordovans raises the
inconsistency count of green shirt to one, eliminating it, since the bound is one.
Gray slacks raises the inconsistency count of white shirt to one, eliminating it,
and leaving us with an empty domain for shirts. Reducing the domain of an
uninstantiated variable to empty also, of course, signals a failure point.

Shapiro and Haralick [35] generalized the CSP look ahead technique in their
study of the “inexact matching problem”. (The algorithm they call “forward
checking” does more looking ahead than ours; it employs the “extended forward
checking” discussed in Sect. 2.5) They defined the “inexact consistent labeling
problem”, which involves searching for all solutions within a given error bound. The count of violated constraints that we use could be viewed as the “error”. However, Shapiro and Haralick’s algorithms did not seek optimal solutions and were not full branch and bound algorithms in the sense that they did not store and compare with the “best so far” solutions; all comparisons were with the error bound.

P-FC(Search-path, Distance, Variables, Domains, Inconsistency–counts)
{Domains holds values for variables not yet assigned a value in Search-path that are consistent with the values in Search-path}
if Variables = nil then
    Best-solution ← Search-path
    N ← Distance
    if N ≤ S then return ‘finished’
    else return ‘keep searching’
else if first of Domains = nil then
    return ‘keep searching’
else if Distance + first count in Inconsistency–counts < N
    {first count in Inconsistency–counts is the number of inconsistencies between first value in first of Domains and values in Search-path}
    and
    while computing New-inconsistency–counts and New-domains,
        {check consistency, as needed, with first value in first of Domains; details determined by which version of P-FC used}
        New-domains retain at least one value in each domain and
        P-FC(Search-path plus first value in first of Domains,
            Distance + first count in Inconsistency–counts,
            Variables minus first variable,
            New-domains, New-inconsistency–counts) = ‘finished’
        then return ‘finished’
    else
        return P-FC(Search-path, Distance, Variables,
            Domains after removing first value from first of Domains,
            Inconsistency–counts minus count for first value
            from first of Domains)

Fig. 12. Forward checking algorithm.

Different variants of forward checking can be devised, depending on the manner in which the counts for each value are used to minimize constraint checks. In contrast to Shapiro and Haralick, who stored counts in tables and did not discard values, corresponding lists can be used for domain values and associated counts, from which values are discarded if their counts are high enough to raise
the distance to the bound.

In the most straightforward version, referred to as P-FC1, the values of a domain are checked against the latest value proposed, \( v \), and the inconsistency counts associated with values inconsistent with \( v \) are incremented by one. During this revision of the domain, when a count for a value \( u \) is incremented, the incremented count is added to the distance of the search path down to \( v \) and the sum is tested against the bound. If this sum equals or exceeds the bound, it means that the search path down to \( v \) cannot be extended to a solution including \( u \) without reaching the bound. Thus \( u \) can be removed; \( u \) will not be checked again at this or at lower levels of recursion.

In the second version (P-FC2), all counts are tested in this manner against the bound during revision, not just the incremented counts. This eliminates values whose counts might now be too large, not because the count has increased, but because the bound has been lowered since the values were last considered for elimination.

In the third version (P-FC3), we take this a step further. All counts are tested before doing any constraint checking to see if counts can be incremented. Thus values may be deleted simply because the bound has been lowered, without any consistency checking to determine if their counts need to be incremented. Here, of course, any incremented counts of values that survived the first test but fail the consistency test have to be tested again during revision of the list of values.

To summarize, in P-FC1, constraint checks are done on all values and count checks on the failures; in P-FC2, constraint checks and then count checks are done on all values; in P-FC3, count checks are done on all values, constraint checks are done on viable values and then count checks are done on values that failed the consistency test. A general P-FC algorithm that does not specify the details of constraint and count checking is shown in Fig. 12.

2.3.3. Tree-structured problems. Local consistency methods have been used to support polynomial algorithms for CSPs with tree or tree-like structure [8] [15] [27]. When problems do not have such structure it may still be useful to view them as containing or being contained in such structures [6] [8] [9] [13] [15] [28]. A problem is tree-structured if the graph that results from viewing variables as vertices and constraints between variables as edges between vertices (the constraint graph) is tree-structured.

The idea of utilizing a tree which represents a subproblem is particularly attractive in the PCSP domain where the constraints that need to be removed to reduce a problem to the desired structure do not eventually have to be satisfied, but may be written off as unsatisfied constraints in the partial constraint satisfaction process. The algorithm P-T in Fig. 13 obtains an efficient solution for tree-structured maximal constraint satisfaction problems.

Tree-structured CSPs can be solved in time linear in the number of variables and quadratic in the maximum domain size, but at first blush one might suppose that tree-structured maximal constraint satisfaction problems would not admit such a small bound. In fact, however, algorithm P-T does achieve this bound.
Algorithm P-T:
For each variable, \( L \), which is a leaf node of the constraint tree:
    For each value, \( e \), of \( L \):
        Set \( \text{Cost}(e) = 0 \)
For each level in the tree starting at the level above the leaves, and working upwards:
    For each variable, \( V \), at that level:
        For each value, \( v \), of that variable:
            For each child, \( U \), of that variable:
                For each value, \( u \), of that child:
                    If \( v \) is consistent with \( u \)
                        then set \( \text{Cost}(v, u) = \text{Cost}(u) \)
                        else set \( \text{Cost}(v, u) = \text{Cost}(u) + 1 \)
                    Link \( v \) to a \( u \) such that \( \text{Cost}(v, u) \) is minimum
                    Set \( \text{Cost}(v) \) to that minimum \( \text{Cost}(v, u) \)
                    Delete all \( v \) except those with minimal cost at \( V \)
Return as a minimal solution a value for the root with minimal cost, along with the tree of values linked to it at the other variables.

**Fig. 13.** A linear algorithm for a tree-structured maximal constraint satisfaction problem.

**Theorem.** Algorithm P-T finds a maximal solution for a tree-structured maximal constraint satisfaction problem and has an \( O(n^d) \) complexity bound, where \( n \) is the number of variables and \( d \) the maximum number of values in a variable domain.

**Proof.** As we process the tree of variables we associate costs with values. The cost represents the total number of constraints violated if we choose that value and all values we have linked to it at descendant variables. We retain only minimal cost values at each variable. We claim that the cost of a value at a variable represents in fact the minimal number of constraints that we need to violate in order to instantiate that variable and its descendants in the variable tree, while that value and its descendant values represent in fact an optimal solution for the subtree. Thus at the root variable we will have found a minimal cost instantiation, an optimal solution for the complete tree-structured problem.

The claim is trivially true at the leaves. We work our way inductively up the tree to the root. Assume that the claim is true for all the children of a variable \( V \) in the constraint tree. The algorithm only keeps values at \( V \), that minimize the additional cost vis-à-vis previously retained values for the children, i.e. that minimize the number of constraints violated. Changing those previously retained values could not improve matters: changing a value at a child to avoid an inconsistency with the parent value means replacing the child value with one
whose additional cost at the very least offsets the additional consistency. Note also that the only constraints between \( V \) and its descendants are those between \( V \) and its children. Furthermore there are no constraints between variables in different subtrees of \( V \). Thus the cost of a value at \( V \) represents the minimal number of constraints that we need to violate in order to instantiate that variable and its descendants in the variable tree, while that value and its descendant values represents an optimal solution for the subtree rooted at \( V \).

Working up from the leaves, the algorithm builds up optimal solutions for subtrees, all the way to the root. Essentially all \( n - 1 \) edges in the tree have to be processed once, with each processing requiring at most \( d^2 \) consistency checks.

It should be emphasized that these results, indeed more powerful results, have already been obtained in a closely related context [7]. The context is a CSP with multiple solutions, where the objective is to choose a solution which maximizes the value of a criterion function. Though superficially this context appears quite distinct from the PCSP context, where there may not even be a single solution, one could presumably use the criterion function to simulate a maximal satisfaction problem.

### 2.4 Ordering

As usual in branch and bound it is advantageous to order the search to heuristically increase the likelihood that a good, or ideally optimal, solution will be found early. The counts produced by the arc consistency method described in Sect. 2.3.1 can be used to order search. This can be done either by ordering the values in each domain according to their individual counts or by ordering the variables on the basis of some statistic derived from the counts for each domain.

In the tests carried out for this paper, values are ordered by increasing counts. This allows the values most likely to produce a good solution to be tested first, so that a minimum or near-minimum distance solution should be found more quickly on average, yielding a better bound early in search.

In the present work the statistic used for variable ordering is the mean for the counts associated with the values of each domain. (The minimum count was also considered, but, for most problems, there are too many zero counts to make this statistic sufficiently discriminating.) In addition, variables are ordered by decreasing mean count. This is based on the premise that, once a good bound is found, checking domains with less support early in the search will increase the likelihood that the bound will be reached at higher levels of the search tree.

This argument is supported by tests in which the ordering was in the opposite direction; the results were appreciably worse in most cases (and sometimes worse than the basic branch and bound), especially for sets of harder problems.

A variety of variable ordering techniques have been studied for CSPs and could be considered in the PCSP context. More sophisticated cost estimates could be associated with the variables to support additional PCSP-specific techniques. The two forms of ordering based on arc consistency counts, as well as the
pruning based on arc consistency counts described in Sect. 2.3.1, can, of course, be combined in different ways.

2.5 Extensions

The basic techniques discussed above can be extended in a variety of directions, and other techniques considered. Obviously in this paper we can only begin a research program that requires, at the least, a recapitulation of the entire history of progress on CSPs.

One obvious line of inquiry involves combining basic techniques. We emphasize in this paper development and analysis of basic “atomic” techniques. However, we conducted some experiments with an algorithm which combines a retrospective technique—backmarking—a prospective technique—arc consistency count pruning—and an ordering technique—value ordering. We call this algorithm P-RPO.

The branch and bound context suggests looking for tighter lower bounds on the distance of the minimal distance solution that includes a given set of value choices. These can be used to obtain quicker pruning of choices that have no chance of doing better than a solution already found. The arc consistency count pruning described above does this sort of thing in a simple but efficient manner; the arc consistency counts are only computed once in a preprocessing step. The forward checking analogues utilize a kind of dynamic arc consistency count.

Shapiro and Haralick [35] suggest more elaborate lower bound computations, up to dynamically utilizing a complete arc or even path consistency [26] analogue after each value choice, for their inexact matching problems. Of course, there are tradeoffs between consistency check savings and bound computation costs. We tested an analogue, which we call extended forward checking for partial constraint satisfaction, of the most successful algorithm that they implemented.

Forward checking for maximal constraint satisfaction, as described above, assigns an inconsistency count to a value, v, based on the inconsistencies that would be incurred by adding v to the choices, C, which have already been made. Extended forward checking goes further by forming a lower bound estimate of the number of further inconsistencies that would accrue in the course of choosing values for each of the remaining variables. For each of these variables it finds the minimum inconsistency count assigned to the values for that variable. It then adds all of these counts to the inconsistencies incurred in choosing C and v. This sum will be a lower bound estimate on the number of constraints violated by a maximal solution that contains C and v.

Thus we could turn the P-FC algorithm into an extended forward checking algorithm, P-EFC, by changing the test:

\[ \text{Distance + first count in Inconsistency-counts} < N, \]

into the test:

\[ \text{Distance + first count in Inconsistency-counts} \]
+ sum of minimum counts for each variable

in Remaining-variables < N.

The version of extended forward checking we tested included the forward checking refinements implemented in our P-FC3 algorithm, and is thus called P-EFC3.

Essentially we are playing a “what if” game. If we added v to the already chosen values, what would be the best we could hope for? If that is not as good as the best we have already done, we can prune v from consideration (until such time as we may change our choices, C).

3 Experiments

3.1 Overview

The algorithms described in Sect. 2 were tested in a series of experiments with random problems. In the first six experiments, we examined the relative efficiency of these algorithms and the relation between efficiency and problem structure. In these experiments each algorithm was run to completion, to find a maximal solution. Three preliminary studies compared the efficiency of related algorithms. The first included the retrospective algorithms, P-BJ and P-BMK, together with the basic branch and bound (P-BB). The second compared the three versions of P-FC, which differed in the number and placement of tests of distance plus counts against the bound. The third compared procedures based on the arc consistency algorithm, P-ACC. In the fourth and main experiment we tested the most promising algorithms from the first three experiments on a more extensive set of problems, in which basic problem parameters such as domain size were varied systematically. In the fifth experiment, the best algorithms were tested on problems of varying size (number of variables); this experiment also included the algorithms P-RPO and P-EFC3. In the sixth experiment we obtained data on overall efficiency, using time as a measure, with problems selected from the main experiment that presented different levels of difficulty for the algorithms.

For many problems finding a maximal solution may not be feasible because the problem is too hard to solve completely. In these cases a good submaximal solution may still be acceptable. Branch and bound techniques are useful in this situation because of their anytime feature: after an initial solution is found, the algorithm can stop at any time before completion with the best solution found so far. In Experiments 7 and 8, we examined the efficiency with which the different algorithms can find submaximal solutions with distances increasingly close to that of a maximal solution. The problems used in these experiments were larger versions of the random problems used in previous experiments and, in addition, a set of large coloring problems believed to be very hard to solve [4]. In these experiments resource bounds were established by placing a limit on the number of constraint checks that could be performed; the program terminated if this limit was reached.
3.2 Random problem generation

In generating random problems, there are four features to consider:

(1) number of variables, \( n \)
(2) number of constraints, \( c \)
(3) domain size, \( d \)
(4) number of value pairs included in a constraint, \( p \).

In the present work, \( n \) was fixed for each set of problems. Then the values of the other three features were determined with a (constant) probability of inclusion method, that is best explained by example. Consider the choice of number of constraints. For problems of ten variables with a connected constraint graph, up to 36 constraints can be added. In generating a random problem, the probability of inclusion is fixed at, say, 0.3, and each of the possible constraints is considered for inclusion using random methods to simulate this probability. With a probability of 0.3, a set of problems is obtained with an expected value for number of constraints equal to \( 9 + (36 \times 0.3) \approx 20 \). Similar procedures were used to determine \( d \) and \( p \) for each domain and constraint, respectively. The only limitation in these cases was that the value of \( d \) or \( p \) was at least one.

If no element was included (zero value), the procedure was repeated beginning with the first element until a non-zero value was obtained. This method has the advantages that:

(1) each parameter value can be varied in a way that is easily characterized, i.e.,
    by a single probability value
(2) each element has the same probability of inclusion, which makes the sampling
    properties of each possible set of elements relatively easy to characterize.

For these experiments, problems without solutions were required. This limited the range of probability values that could be considered, because as \( d \) or \( p \) increases, it is more likely that problems with solutions will be produced. Values of 0.2, 0.4 and 0.6 were used to determine \( p \), while values of 0.1, 0.2 and 0.3 were used for \( d \), based on a maximum domain size equal to \( 2n \) or twice the number of variables. Values of 0.3, 0.6 and 0.9 were used to determine the number of constraints to be added to a spanning tree (which was itself derived by choosing pairs of variables at random). In the remainder of the paper, these probability values will be designated as \( p_c \), \( p_d \), \( p_p \), for probability of constraint, domain, or constraint pair inclusion, respectively. \( p_c \) is sometimes called the density of the problem, and \( p_p \) the (relative) satisfiability, while the complement of \( p_p \) is sometimes referred to as the tightness of a constraint.) The values chosen covered most of the range of possible values, while allowing a similar degree of variation in each case. After generation, a problem was tested for solutions. If a solution was found, the following strategy was used to obtain an insoluble problem with identical parameter values: a constraint pair that included two values in the solution was chosen at random and discarded, and another pair of values from the same domains was chosen at random as a new constraint pair; this procedure was repeated until a problem with no solutions was found.
3.3 Experimental design

Experiments 1–3. Experiments 1–3 were based on a set of ten-variable problems for which the probabilities of domain and value pair inclusion took on all the values mentioned in the last section, while the density was always 0.3. This gave nine categories of problems. Ten problems were generated for each category, for a total of 90 problems.

In Experiment 1, the algorithms tested were P-BB and two other retrospective algorithms, the backjumping and backmarking analogues, P-BJ and P-BMK. Experiment 2 compared the three variants of P-FC described in Sect. 2.

Experiment 3 compared several variants of P-BB that incorporated different forms of information derived from arc consistency counts, either singly or in combination. These were:

1. pruning based on the count for a given value,
2. ordering of values in each domain by increasing count,
3. ordering of variables by (decreasing) mean count of the values in their domains,
4. a combination of the value and variable ordering strategies,
5. a combination of pruning and value ordering,
6. a combination of pruning, value and variable ordering.

Experiment 4. Experiment 4 compared the most promising algorithms from each of the first three experiments. These were P-BMK, P-FC3, and two varieties of branch and bound that incorporated information from the counts obtained by P-ACC ((5) and (6) above). P-BB was also included for reference. For this experiment the problem set was expanded to include the other two probabilities of constraint inclusion (0.6 and 0.9). It was, therefore, a fully crossed design, with each of the three probabilities of inclusion associated with each parameter, as described in Sect. 3.2. This gave 27 categories; ten problems were generated for each category for a total of 270 problems. (These included the 90 problems used in Experiments 1–3).

Experiment 5. In this experiment the best retrospective and prospective algorithms from Experiment 4, P-BMK and P-FC3, along with P-BB, were compared on problems in which the number of variables, \( n \), was varied. In addition, the extensions P-RPO and P-EFC3 were included. The number of variables ranged between eight and 12, with ten problems for each problem size. Based on preliminary tests of feasibility for higher \( n \), the values of 0.3, 0.2 and 0.4 were chosen for \( pc \), \( pa \), and \( pb \), respectively. (In Experiments 1–4, with \( n = 10 \), problems in this category were relatively easy to solve.).

Experiment 6. In Experiment 6, the following algorithms were compared with respect to run time: P-BB, P-BJ, P-BMK, P-FC2 and P-FC3. Problems were selected from Experiment 4 in which the order of magnitude for constraint checks was three, four or five for branch and bound. (Similar ranges in terms of
order of magnitude were obtained with the other algorithms tested.) Six problems were chosen at each level of difficulty, three for which the density, $p_c$, was 0.3 and three for which it was 0.9, for a total of 18 problems. These problems were also chosen so that other parameter values (probabilities of inclusion) also varied. Run times were obtained with the Lisp time function.

**Experiments 7–8.** In Experiment 7, problems had 12, 16, or 20 variables, with ten problems per group. The values of $p_c$, $p_d$, and $p_r$ were the same as those in Experiment 5; in fact, the same 12-variable problems were used in both experiments. As a consequence, the distance associated with the best solution was known. For the 16-variable problems, an optimal solution was obtained using P-EFC3. This allowed a more complete evaluation of the suboptimal solutions obtained when the number of constraint checks was limited to two million. It was also used as a reference for the results with 20-variable problems, which were too large for an optimal solution to be found when the number of constraint checks was limited to five million. Five algorithms were tested with the 12-variable problems: P-BB, P-BMK, P-FC3, P-RPO and P-EFC3, and all but P-FC3 were tested with the 16- and 20-variable problems. (Since P-FC3 was always bettered by P-EFC3, it will not be discussed further.)

In Experiment 8 a similar procedure was used with nine large, “really hard” coloring problems [4]. These were classic graph coloring problems where the objective is to color every vertex of a graph with a color, chosen from a fixed number of colors, such that no vertices joined by an edge have the same color. Vertices correspond to CSP variables, edges to CSP constraints. These problems involved four colors, 130–144 variables and 620–646 constraints, giving densities of 0.05–0.06. Three algorithms were tested: P-BB, P-BMK and P-EFC3. (P-RPO was not included since arc consistency counts are zero for all values in coloring problems of this sort.)

In all experiments except the sixth, the basic measure was the number of constraint checks, although we also recorded the number of nodes searched. In Experiment 6, the measures were execution time and time per 1000 constraint checks. Garbage collection time and basic I/O and set-up time were subtracted from the total time in calculating these measures.

In these experiments, the analysis of variance (ANOVA) was used to test the statistical significance of differences due to the algorithm used and to variation in each problem parameter, as well as interactions between these factors. In these tests, one to three of the statistical factors were based on problem parameters. If more than one such factor appeared in an experiment, they were fully crossed, with each combination of factors forming a single experimental group. The algorithm used to solve the problem was a separate factor, with all problem groups “repeated” on it. (A simple fixed effects design, in which different problems are chosen from the same category for each algorithm would introduce more variation and is unnecessary, since independence of different treatments (algorithms) is not an issue in this domain.) As an example consider the design of Experiment
1. Here, the factors based on \( p_d \) and \( p_p \) are crossed to form separate categories of problems, all of which are repeated on the factor related to the three algorithms tested. Each of these factors, as well as the first and second order \((p_d \times p_p \times \text{algorithm})\) interactions was tested for statistical significance, using the standard null hypothesis of no differences between groups related to that effect. All analyses were done with log-transformed data, to reduce differences in variance among groups. If the effect of algorithms in the ANOVA was statistically significant, algorithms were compared on their mean performance using Tukey’s \( q \) test for nonorthogonal pairwise comparisons [22].

In Experiment 4, we performed several further analyses to better understand performance characteristics in relation to problem parameters. Standard deviations were obtained for each algorithm on each problem set, as well as measures of skew, or asymmetry in the distributions of performance scores. Pearson product-moment correlations between P-BB and the other algorithms were also derived, using the original scores. And, for each algorithm, multiple regression analysis with respect to the problem parameters was carried out using the log-transformed scores and a zero \( y \)-intercept.

In Experiments 7–8, statistical analysis consisted of paired comparison \( t \) tests between algorithms, beginning with 100 constraint checks and including successive powers of ten up to the highest power within the response bound. (The value of 5 million constraint checks was also tested for 20-variable problems.) For P-RPO the constraint checks required for arc consistency checking were added to the total in each case.

3.4 Results

3.4.1. Experiments 1–3: Preliminary Comparisons of Similar Algorithms. In each of the first three experiments, some of the algorithms were clearly superior to others tested on the same set of problems. In Experiment 1, P-BMK was markedly superior to both P-BB and P-BJ. In Experiment 2, P-FC3 was the most efficient in terms of constraint checks, although all three variants of P-FC were much better than P-BB. In Experiment 3, versions of P-BB that used variable and value ordering together, or pruning and value ordering, or a combination of all three strategies were generally superior to the other three variations as well as to P-BB.

These results were borne out by the statistical analysis. In all three experiments, the effect due to algorithms was statistically significant (Experiment 1, \( F[2,243] = 12.89, p < 0.001 \); Experiment 2, \( F[2,243] = 4.58, p = 0.01 \); Experiment 3, \( F[5,486] = 5.85, p < 0.001 \)). In addition, the factors related to problem parameters, specifically to differences in \( p_d \) and \( p_p \), were highly significant statistically \((F > 50 \text{ always, } p < 0.001)\), as was the interaction between these factors \((F > 10 \text{ always, } p < 0.001)\). With one exception in Experiment 3, none of the interactions between algorithms and the other factors was statistically significant \((F \leq 1)\).

The meaning of the significant effects related to problem parameters can be understood from Fig. 14, which gives the main results for Experiment 1. As
\( q[2.89] = 2.82, \ p < 0.05 \) and \( q[2.89] = 4.20, \ p < 0.01 \), for comparisons with P-FC2 and P-FC1, respectively. All versions were markedly superior to P-BB.

In Experiment 3, improvement in performance due to strategies based on arc consistency counts depended in part on domain size, which was reflected in the statistically significant interaction between algorithms and the factor based on \( p_d \) \( (F[10,486] = 2.95, \ p = 0.001) \). Consider, first, the individual strategies of pruning based on counts, or ordering values, or ordering variables on the basis of counts. For \( p_d = 0.1 \), pruning and value ordering each reduced the number of constraint checks relative to the basic P-BB algorithm, while variable ordering resulted in increases, which were sometimes marked. For \( p_d = 0.2 \) or \( 0.3 \), all three strategies improved the mean performance, although the first two did so most consistently (Fig. 15). Perhaps because of this interaction, none of the comparisons of individual strategies was statistically significant, although the comparison between variable ordering and pruning approached significance.

![Fig. 15. Mean constraint checks required by algorithm P-BB when combined with strategies based on arc consistency counts obtained with P-ACC. Constraint checks shown as a function of constraint satisfiability, \( p_d \); in this slice of the parameter space, \( p_d = 0.2 \). Results for the basic branch and bound are included for reference.](image)

Combining strategies based on arc consistency counts resulted in greater improvement in performance (Fig. 15), and this was reflected in the individual comparisons. When pruning and value ordering were combined, performance of this algorithm was better than for either strategy alone \( (q[2.89] \geq 2.8, \ p < 0.05 \) for both comparisons). Value and variable ordering combined was also better than either strategy alone, although only the latter comparison was statistically significant \( (q[2.89] = 3.43, \ p < 0.05) \), in part because the superiority of the combination was only found for \( p_d = 0.2 \) or \( 0.3 \). Finally, the combination of all three strategies showed the best performance overall, although in comparisons with the combinations of two strategies, only the difference with the “double” ordering of variables and values was statistically significant \( (q[2.89] = 3.30, \ p < 0.05) \). Again, this superiority was found only for larger domain sizes.
3.4.2. Experiment 4: Comparing the best algorithms from Experiments 1–3. As indicated above, the algorithms tested in this experiment in addition to the basic branch and bound (P-BB) were P-BMK, P-FC3, and two branch and bound algorithms that used arc consistency counts: one that used pruning and value ordering and one that used all three strategies. Two algorithms based on arc consistency counts were included because the results from Experiment 3 were not altogether conclusive concerning the best algorithm in this group. Although the procedure that combined the three arc consistency count strategies was best overall, it did not perform well on problems with small domain sizes in comparison with other algorithms based on arc consistency counts, and there was considerable variability in performance even when the mean for this algorithm was better than the other means. The algorithm based on pruning and value ordering did not differ statistically from the full combination algorithm overall; moreover, unlike the combination algorithms that incorporate a variable ordering based on the counts, these strategies are less likely to result in inferior performance with respect to P-BB on individual problems.

In this experiment the factor in the ANOVA for constraint checks associated with the algorithms was highly significant statistically (Table 1; the same pattern of statistically significant results was found in the ANOVA for number of nodes checked). There was, in fact, a fairly consistent ordering over the portions of the parameter space that were tested (Fig. 16). P-BMK and P-BB incorporating arc consistency strategies generally reduced the number of consistency checks made by the basic P-BB algorithm by a factor of two to three. P-FC3 reduced this number by another factor of two. In terms of overall means the ranking of performance from worst to best was: basic branch and bound (P-BB), P-BB with the three arc consistency strategies, P-BB with pruning based on consistency counts and value ordering, P-BMK, and P-FC3. Comparisons of mean performance following the ANOVA showed statistically significant differences between each pair of algorithms, including the two that used arc consistency counts.

The ANOVA for constraint checks also showed statistically significant effects related to each problem parameter (Table 1). In addition, all interactions between these factors were statistically significant. The effects of domain size and satisfiability and the interaction between them can be observed in Fig. 16 and Fig. 14. For all algorithms, problem difficulty increased with increasing domain size and with diminishing satisfiability. The hardest problems were those with the largest domains and the smallest number of acceptable pairs per constraint. These effects were enhanced when the density of the constraint graph increased, which accounts for the interactions that involve this factor.

The interactions between algorithms and $p_c$ and between algorithms and $p_d$ were also statistically significant (Table 1). In contrast, the interaction between algorithms and $p_p$ was not significant ($F \leq 1$); nor were any of the higher-order interactions that involved the algorithms factor. Perusal of performance means suggests that the two statistically significant interactions were due in large part to the P-ACC combination algorithm. For this algorithm, the average number of consistency checks increased more dramatically with increases in $p_c$ or $p_d$
Table 1. Statistically Significant Effects in ANOVA for Experiment 4

<table>
<thead>
<tr>
<th>Factor</th>
<th>df</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
<td>4</td>
<td>43.53</td>
<td>0.0001</td>
</tr>
<tr>
<td>$p_d$</td>
<td>2</td>
<td>2070.00</td>
<td>0.0001</td>
</tr>
<tr>
<td>$p_p$</td>
<td>2</td>
<td>470.91</td>
<td>0.0001</td>
</tr>
<tr>
<td>$p_c$</td>
<td>2</td>
<td>423.81</td>
<td>0.0001</td>
</tr>
<tr>
<td>$p_d \times p_p$</td>
<td>4</td>
<td>81.58</td>
<td>0.0001</td>
</tr>
<tr>
<td>$p_d \times p_c$</td>
<td>4</td>
<td>8.56</td>
<td>0.0001</td>
</tr>
<tr>
<td>$p_p \times p_c$</td>
<td>4</td>
<td>21.19</td>
<td>0.0001</td>
</tr>
<tr>
<td>$p_d \times p_p \times p_c$</td>
<td>8</td>
<td>5.77</td>
<td>0.0001</td>
</tr>
<tr>
<td>alg $\times p_d$</td>
<td>8</td>
<td>4.68</td>
<td>0.0001</td>
</tr>
<tr>
<td>alg $\times p_c$</td>
<td>8</td>
<td>4.39</td>
<td>0.0001</td>
</tr>
<tr>
<td>error</td>
<td>1215</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: df is degrees of freedom associated with each factor, $F$ is the value of the test statistic, $p$ is an upper limit on the probability of obtaining an $F$ greater or equal to this value if there are no differences associated with this factor.

than for any other algorithm. Also of interest is an apparent relation between $p_p$ and the average difference between P-FC3 and P-BMK: for problems with higher satisfiability, the performance of these two algorithms was almost equal and was superior to the other algorithms; but as relative satisfiability decreased, P-FC3 became markedly superior (cf. Fig. 16). This was not reflected as an interaction in the ANOVA because for both algorithms amount of work increased dramatically with a decrease in $p_p$.

Differences in variability of performance within problem sets followed patterns similar to the differences in means. In 17 of the 27 sets of problems, the standard deviation for P-BB was greater than for any other algorithm; in the remaining problem sets the full combination P-ACC algorithm was the most variable. P-FC3 had the smallest standard deviation in performance in all but two sets. Almost all distributions showed a strong positive skew, i.e., the tail of the distribution on the right side of the median value was much longer than the tail on the left side.

Correlations between number of constraint checks performed by each algorithm were very high ($\geq 0.97$), with the exception of the P-ACC algorithm that incorporated the variable ordering; here the correlations were about 0.65. Since this correlation is a measure of the linear relation between two variables, it suggests that all of these algorithms have similar performance characteristics with respect to the problem parameters. Multiple regression analyses were very successful, in terms of accounting for most of the variance. The adjusted $R^2$ value was 97% in each case; in contrast, $R^2$ was about 25% when the original (untransformed) scores were used. Examination of residuals with normalized plots and other measures of influence of individual scores such as DFFITS [1]
3.4.3. Experiment 5: Effect of number of variables. As expected, the differences in algorithm performance found in previous experiments were maintained over the range of problem sizes tested (Fig. 17). From this perspective, it appeared that P-BMK and P-FC3 have similar performance characteristics, reducing the number of constraint checks done by branch and bound by a factor of 2–3 through the range. Combining the best retrospective technique (P-BMK) with prospective and ordering techniques (pruning and value ordering based on arc consistency counts) did not materially affect this result: P-RPO reduced the effort required by P-BMK by about 1/3 throughout the range. It may be noted, however, that with few exceptions this combination outperformed P-FC3 and in a few cases (the easiest) outperformed the fastest algorithm.
significant ($F[4, 60] = 16.46, p < 0.001$). In the individual comparisons of means, the differences between P-BB and either P-BMK or the two forward checking algorithms were statistically significant ($p < 0.01$ in each case). The difference between P-BMK and P-FC3 was also statistically significant ($p < 0.01$), but not the difference between P-BMK and P-FC2. The difference between the two forward checking variants was also not statistically significant ($q[1.17] = 1.58$).

For the average time per 1000 constraint checks, the ranking was considerably different. The following are mean values:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mean Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-BB</td>
<td>1.61 sec</td>
</tr>
<tr>
<td>P-BJ</td>
<td>1.67</td>
</tr>
<tr>
<td>P-BMK</td>
<td>2.52</td>
</tr>
<tr>
<td>P-FC2</td>
<td>1.87</td>
</tr>
<tr>
<td>P-FC3</td>
<td>2.28</td>
</tr>
</tbody>
</table>

In the ANOVA the effect due to algorithms was statistically significant ($F[4, 60] = 9.25, p < 0.001$), as well as the effect of number of constraints ($F[1, 60] = 27.10, p < 0.001$), reflecting an increased efficiency for problems with denser graphs. On this measure, P-BB and P-BJ were both clearly superior to P-BMK and to P-FC3 ($p < 0.01$ for individual comparisons), P-FC2 was also superior to P-BMK ($p < 0.01$) and to P-FC3 ($p < 0.05$).

These results indicate that P-BMK and P-FC3 do incur a relatively large overhead in comparison with the other algorithms. In the case of backmarking, there is similar evidence for the CSP version [19]. But it is obvious that the decrease in number of constraint checks with these more elaborate algorithms yields a greater overall efficiency than the basic P-BB algorithm, reflected in the total time required to solve the problems.

### 3.4.5. Experiments 7–8: Finding submaximal solutions for hard problems

The results of increasing the amount of effort applied to a problem were analyzed in two ways, by considering either, (i) the number of constraint checks required to reduce the distance by a given proportion of the total change possible (the difference between the number of constraints and the best distance), or (ii) the minimum distance found after $k$ constraint checks. As indicated, the former measure could only be derived for 12-variable problems.

For all algorithms, after an initial drop due to the difference between the number of constraints (the initial bound) and the distance of the first solution, the relation between effort and goodness of solution approximated a simple logarithmic function (Figs. 18 and 19). Since P-EFC3 and especially P-RPO required more constraint checks to find an initial solution than P-BB or P-BMK, the latter algorithms were more effective initially ($p < 0.001$ for 100 constraint checks). (In the search phase per se, P-RPO actually found solutions with a given suboptimal distance faster than any other algorithm, but, in terms of constraint checks, this was overwhelmed by the cost of preprocessing.) However, for
Fig. 18. Mean constraint checks required to reduce the distance as a proportion of the difference between the number of constraints and the best distance (12-variable problems).

1000 or 10,000 constraint checks, the best solutions were found by P-RPO or P-EFC3 ($p < 0.05$ for all comparisons with P-BB, for comparisons with P-BMK at 1000 constraint checks, and for the comparison between P-EFC3 and P-BMK at 10,000 constraint checks). Differences between P-RPO and P-EFC3 were never statistically significant. By 100,000 checks all algorithms except P-BB had found maximal solutions for most of the problems.

Fig. 19. Best distance found as a function of number of constraint checks for problems with 12 variables.

For the 20-variable problems (Fig. 20) the effects were similar, although here the superiority of P-EFC3 was more noticeable (and was statistically signifi-
cant for each order of magnitude beginning with 1000 checks). In addition, the difference between P-RPO and P-BMK, while present, was never statistically significant. However, in general the trends found for smaller problems appear to hold as problem size is scaled up. (The results for 16-variable problems were also consistent with these results.)

![Graph](image)

Fig. 20. Best distance found as a function of number of constraint checks for problems with 20 variables.

The results for the hard coloring problems were also similar (Fig. 21). Here, the retrospective algorithm, P-BMK, achieved a better solution after a small number of constraint checks than P-EFC3, although the latter eventually surpassed it. However, the difference between P-BMK and P-EFC3 was never great, although it eventually attained statistical significance at ten million constraint checks ($p < 0.05$). In this case, the degree of local consistency in these problems may have been responsible for the less impressive performance of P-EFC3 relative to P-BMK.

### 3.5 Summary of experimental results

In general, analogues of algorithms that perform well on CSPs performed well on maximal constraint satisfaction problems. Among a basic set of strategies, over the range of parameter values tested, using the measures of constraint checks and total time to obtain an optimal solution, one analogue of forward checking was generally superior. Moreover, this superiority was most evident for parts of the parameter space in which the problems were most difficult. In less extensive testing, more elaborate algorithms reduced the work required even further; here also an extension of forward checking performed better than any other algorithm. On a set of 12-variable problems, for example, the best algorithm
reduced the average number of constraint checks from approximately 300,000 to approximately 20,000.

**Fig. 21.** Best distance found as a function of number of constraint checks for hard coloring problems.

Despite the general superiority of one type of prospective strategy (forward checking), other techniques based on local consistency (arc consistency count preprocessing) or sophisticated retrospective algorithms (P-BMK) also improved on the basic branch and bound algorithm, sometimes by a factor of 4–5. In addition, for some parts of the parameter space, P-BMK was comparable in performance to P-FC3. We conclude that these techniques merit further study, especially if they can be efficiently combined with forward checking methods.

The data from Experiment 3 suggest that for sparse problems, combining arc consistency counts with a variable ordering that puts high counts at the beginning of the search sequence was efficient on average. A key factor appears to be the size of the counts in comparison to the best distance; when the distance is small, putting higher counts at the beginning is very effective in pruning the search tree. This strategy can be incorporated into the P-RPO algorithm, and preliminary data suggest that this may be an extremely effective technique for many, but not all, problems with low densities. \(p_c = 0.1\) for the problems tested.

The main result from the anytime experiments was that solutions within ten percent of the optimum, in terms of the total distance reduction, could be obtained with a reduction in effort equal to or greater than one order of magnitude. As in earlier experiments, P-EFC3 was the best algorithm through most of the range of effort. However, P-BB and P-BMK were more efficient if the criterion of goodness was relaxed sufficiently.

These experiments also show that random problem generation can be parameterized in a straightforward manner to produce subpopulations of problems that vary statistically in their basic features: number of constraints, domain size and satisfiability. This allowed us to examine algorithm performance in terms of
the space of problems defined by these parameters. Regions of this space were

delineated in which the typical problem was either easy or difficult to solve,
and data on the population characteristics of algorithm performance in these
regions was obtained. Such data should aid the potential user in deciding which
algorithms to consider, as well as indicating the costs of such decisions.

4 Partial constraint satisfaction

We focused on maximal satisfaction to facilitate the presentation and testing of
the algorithms. However, there are other forms of partial satisfaction, and to a
large extent the algorithms generalize in obvious ways. In this section we work
toward a general model of partial satisfaction.

4.1 Metrics

Our branch and bound algorithms have sought a solution that violates a minimal
number of constraints. However, the difference between a perfect solution, and
a partial solution can be measured in many different and more subtle ways. As
our use of the term “distance” suggests, all the branch and bound technique
requires is a metric that can compare the values being considered at a given
stage of search with the “best” solution found so far.

Preferences have been expressed by ordering constraints [10], by representing
their importance [12], by organizing constraints into hierarchies [2], by introduc-
ing priorities [11]. Preferences can be reflected in the branch and bound metric
by assigning weights to constraints. Preferences could be associated with subsets
of domains and constraints, individual values or pairs of values, as opposed to
etire constraints. (Wearing a striped tie and a polka dot shirt might not be as
bad as wearing a bow tie with a sweatshirt.)

Constraint deviations have been combined in both local and global fashion
[2]. The branch and bound metric can sum the weights of the violated con-
straints, use the maximum weight, compute the average, as appropriate. The
initial constraints may be viewed as ideal points which we seek to approximate
by some measure.

There may be some “hidden agenda” embodied in the metric. For example,
we may wish to drive a problem toward a weaker version that is easily solvable,
e.g. by removing constraints to yield a tree-structured problem.

4.2 Problem spaces

We have been measuring the success of a partial solution by evaluating the num-
ber, or importance, of the violated constraints. There is another criterion that
we believe should be directly considered. Weakening constraints in effect creates
a different problem. Alternatives for weakening constraints provide alternative
problems. We may well wish to solve a problem that is close to the original in
the sense of having a solution set similar to the original. Removing one set of
constraints might trivialize the problem, allowing thousands of new solutions, while removing another set might allow only a single new solution.

This consideration arises, for example, when viewing constraint knowledge base debugging as a partial constraint satisfaction problem [21]. If the knowledge base is erroneously overconstrained, a change that allows a small number of new solutions is more in keeping with Occam’s Razor than one that allows many.

More formally, we can consider a space of problems with an ordering based on solution sets. A problem space is a partially ordered set, \((PS, \leq)\). PS is a set of constraint satisfaction problems with a partial order, \(\leq\), on PS defined as follows: \(P_1 \leq P_2\) if the set of solutions to \(P_2\) is a subset of the set of solutions to \(P_1\). If the set of solutions to \(P_2\) is a subset of the set of solutions to \(P_1\), but the two sets are not equal, we will write \(P_1 < P_2\) and say that \(P_1\) is weaker than \(P_2\).

One natural problem space for a partial constraint satisfaction problem with initial problem \(P\) consists of all problems \(Q\) such that \(Q \leq P\). This set can be obtained by considering all the ways of weakening the constraints by allowing additional consistent combinations of values. In general, \(PS\) could contain problems \(Q\), which are stronger than \(P\), \(P < Q\), or problems \(Q\) such that neither \(Q \leq P\) nor \(P \leq Q\); \(\leq\) is only a partial order. However, if we collect all the constraints in all the problems in \(PS\) into a single problem \(M\), then all the problems in \(PS\) can be regarded as weakenings of \(M\).

It may be natural to consider a space which does not include all \(Q \leq P\). We may wish to specify how the problem can be weakened. Some weaker problems may make more “semantic sense”. It may not be possible or desirable to violate constraints by arbitrarily allowing individual pairs of values to violate the associated constraint, as we have done until now. For example, in our dressing domain, we might decide that in a fashion emergency, or a burst of avant garde creativity, we would eschew the prohibition against mixing stripes with checks, as opposed to making an individual exception for one striped tie and one checked shirt. We may establish levels of informality in our fashion constraints corresponding to the informality of the occasion for which we are dressing.

It may make more semantic sense to have a preset hierarchy of constraints, where weakening a constraint requires moving upward in the hierarchy, as opposed to making arbitrary individual exceptions. This constraint hierarchy is reminiscent of the concept hierarchies that provide initial bias in machine learning settings, and indeed it is intriguing to think of the constraint satisfaction process as a form of concept learning, synthesizing a relationship from positive and negative information.

The specification of the problem space \(PS\) can clearly affect the efficiency of the search process. One way to specify the problem space is to specify generators, or operators, that take us from one problem \(P\) to a permitted set of problems \(Q_t; Q_t \leq P\). There may be “global” restrictions on these generators, e.g., choose one constraint from column A, one from column B. As we try to move through a problem space in search of a solvable problem, it may prove desirable to take into consideration how many opportunities are opened up by altering a problem in a given way, e.g., removing one fashion restriction may be more liberating than
removing another.

The process of weakening CSP's can now be naturally viewed as involving four options: enlarging a variable domain (buying a new shirt), enlarging a constraint domain (deciding that an old shirt and an old tie can be worn together after all), removing a variable (deciding not to wear any tie), and removing a constraint (deciding not to worry if our socks match). However, all of these can in turn be expressed in terms of the basic process of enlarging constraint domains. We can view variable domains as unary constraints. Enlarging a binary constraint until it contains all pairs of values in the specified domains for the two variables is tantamount to removing the constraint (at least until such time as the variable domains may be enlarged). Removing all the constraints on a variable in this way is tantamount to removing the variable.

4.3 Partial constraint satisfaction problems

A partial constraint satisfaction problem can now be specified more formally by supplying an initial constraint satisfaction problem \( P \), a problem space, \( PS \), containing \( P \), a metric on that space, and necessary and sufficient solution distances, \( N \) and \( S \). A solution to a partial constraint satisfaction problem can be defined as a problem \( P' \) from the problem space \( PS \) along with a solution to that problem where the metric distance of \( P' \) from \( P \) is less than \( N \). A solution is sufficient if the distance is less than or equal to \( S \). An optimal solution is one where the metric distance of \( P' \) from \( P \) is minimal over the problem space. The optimal solution is dominant if there is no optimal solution which involves a problem \( Q \) such that \( P' < Q \).

The metric that evaluates our solution now compares problems rather than counting or otherwise evaluating violated constraints. Of course, one way to compare problems is to compare their constraints. Ideally, however, we might like to define the metric in terms of the partial order, defining the distance between \( P \) and \( P' \) to be the number of solutions not shared by \( P \) and \( P' \). When \( P' \leq P \), this metric, \( M \), measures the number of solutions we have added by weakening \( P \). This is a natural measure of how "good" our partial solution is.

Of course, computing such a metric is not likely to be easy. However, after finding a set of optimal solutions with another metric we might wish to distinguish among these by considering the different problems induced by these solutions. The problem induced by a partial solution \( S \) for a problem \( P \) is the problem obtained by adding to the constraints of \( P \) the inconsistent pairs of values in the partial solution \( S \). We could compute the full solution sets for the induced problems, and look for dominant, optimal solutions using the metric \( M \), which operates on solution sets.

We also may wish to consider how well an alternative metric does tend to reflect this natural metric \( M \). Another natural metric is a count of the number of permitted value combinations not shared by the constraints of \( P \) and \( P' \). To some extent, this metric does reflect the metric \( M \) based on the partial order. If \( P' \) is obtained from \( P \) by weakening the constraints then \( P' \leq P \), because of the monotonic nature of constraint satisfaction problems. In other words, if for
each constraint $C_{ij}$ associated with $P$ and constraint $C'_{ij}$ associated with $P'$,
$C_{ij}$ is a subset of $C'_{ij}$, then $P' \leq P$. In particular, the simple metric used in our
maximality studies, which counts the number of violated constraints, is a metric
of this form.

Viewing partial constraint satisfaction as a search through a problem space
facilitates consideration of integrating branch and bound at different levels of
the search process to produce different algorithms. The natural points at which
to perform this integration are the failure points in a standard backtracking
algorithm.

Our basic branch and bound partial constraint satisfaction algorithm inte-
grates branch and bound at the lowest level of backtrack failure. Whenever a
value $c$ for a variable $V$ would be rejected by normal backtracking, we can view
the algorithm as determining which constraints were violated, and weakening
the problem by adding precisely those constraint elements needed to permit $c$.

In searching through the problem space for a solvable problem, it would be
desirable to avoid changing the problem in ways that do not facilitate progress.
For example, if two problems are equivalent, they both do not need to be consid-
ered. A nice feature of the basic branch and bound algorithm is that for the type
of partial constraint satisfaction problem for which it is designed it is able to
use only the minimally different problem $P$ required to proceed at each problem
choice point.

An even simpler choice for integrating branch and bound would be to add
it to backtracking upon top-level failure of backtracking, when no solution is
found. A branch and bound loop can be added on the outside of the backtracking
algorithm. This loop will run through the problems in the problem space, keeping
track of the closest problem $P'$ to $P$ solved so far. Problems no closer to $P$ than
$P'$ will be rejected immediately.

Whenever alternative problems are generated in an order which reflects their
distance from the original problem, closest to furthest, generation can stop at
that point when the necessary bound $N$ is reached. If we have a top-level in-
tegration of branch and bound, this point marks the termination of the PCSP
algorithm.

Branch and bound can be integrated at failure points in between these ex-
tremes. The natural compromise would occur at the points where all the values
for a given variable have been exhausted in standard backtrack search. At these
times options for alternative problems may be explored.

There is a tradeoff involved in the choice of how to integrate branch and
bound. By integrating at a lower level we take greater advantage of backtrack
pruning to avoid unnecessary effort. On the other hand by integrating at a higher
level we allow greater flexibility in heuristically guiding the search through the
space of alternative problems. Preliminary experiments comparing a high level
and a low level approach to partial satisfaction in the domain of debugging
constraint knowledge bases reflect this tradeoff [21].

In summary, the generalized view we have reached of partial constraint sat-
isfaction as a search through a space of alternative problems has a number of
potential advantages for further work in this area. It facilitates consideration of more global concerns than the suitability of a single solution. Specifically, it encourages us to consider whether our partial solution has been devalued by weakening the problem in a manner that permits too many solutions. It naturally incorporates practical concerns about available constraint modifications. It also facilitates generation of alternate problem solving strategies that take a more global view of effective means of modifying the problem.

5 Conclusion

Standard constraint satisfaction problem (CSP) solution techniques have analogues for solving partial constraint satisfaction problems (PCSPs), which both cope with and take advantage of the differences between CSP and PCSP. Branch and bound is the natural analogue of backtrack search. Local consistency count is an analogue of local consistency. We have found PCSP analogues of retrospective, prospective and ordering techniques for CSPs.

Extensive experimentation over random problems with different structural parameters revealed the effectiveness of a set of PCSP techniques as a function of these parameters. A general model of PCSPs was developed involving a standard CSP together with a partially ordered space of alternative problems and a metric to measure the distances between these problems and the original CSP.

In summary, a firm algorithmic, experimental and theoretical foundation has been laid for the study of problems for which it is impractical or impossible to satisfy fully a set of constraints.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. IRI-8913040 and Grant No. IRI-9207633. Part of this work was done while the first author was a Visiting Scientist at the MIT Artificial Intelligence Laboratory. Portions of this paper were taken from [14], which is copyright (c) 1989 International Joint Conferences on Artificial Intelligence, Inc., and used by permission. We thank Peter C. Cheeseman and Bob Kanefsky for supplying the "really hard" coloring problems.

References


This article was processed using the \LaTeX macro package with LLNCS style