Hard Problems for CSP Algorithms

David G. Mitchell

Department of Computer Science, University of Toronto,
Toronto, Ontario, M5S 3G4, Canada
mitchell@cs.toronto.edu

Abstract

We prove exponential lower bounds on the running time of many algorithms for Constraint Satisfaction, by giving a simple family of instances on which they always take exponential time. Although similar lower bounds for most of these algorithms have been shown before, we provide a uniform treatment which illustrates a powerful general approach and has stronger implications for the practice of algorithm design.

Introduction

Finite-domain constraint satisfaction (CSP) is a popular problem solving technique in AI. A CSP instance is a set of variables, and a set of constraints on values assigned to them. Solving an instance requires finding an assignment which satisfies the constraints, or determining that no such assignment exists. In the usual formalizations this task is **NP**-complete, so a polynomial time algorithm exists if and only if **P**=**NP**. Since many practical problems amount to solving CSPs, we want to find the best algorithms we can, regardless of the truth of this proposition.

The literature on performance of CSP algorithms consists largely of experimental comparisons of variants of backtracking on benchmark instances. A few papers include proofs that one variant is never worse (and sometimes better) than another. These are valuable methods, but it may turn out that we are arguing over the best way to build ladders when we need rocketships. We habitually provide (rough) upper bounds for algorithms, but it is lower bounds that lead us to understand when and why algorithms do badly. Thus, lower bounds are more likely to lead us in fruitful directions, or at least away from some which are futile.

In this paper, we analyze a number of popular techniques for solving CSPs, and give a family of instances for which they take exponential time. We discuss differences between our bounds and some previous results, notably those from (Baker 1995), in Related Work. Our method involves showing a correspondence between the algorithm execution and proofs in the propositional calculus. That such correspondences exist seems to be conventional wisdom in the community, but we have never seen a coherent treatment of this in the literature, suggesting that it is not considered important. We hope with this paper to partially redress the former, and encourage re-consideration of the latter. In particular we will argue that proof-based analysis of algorithms, in addition to being a valuable tool for understanding existing algorithms, tells us something useful about what we ought to be doing next.

The properties we prove are for unsatisfiable instances, but the message about satisfiable instances is essentially the same[1]. When a backtrack-based algorithm takes a long time on a satisfiable instance it must be because, after making some unlucky assignments, it generated an unsatisfiable sub-problem that took a long time. If every unsatisfiable sub-problem generated was solved quickly, the algorithm would find a solution quickly. Our results apply also to unsatisfiable sub-problems generated in the process of solving a satisfiable instance.

Note that our lower bounds apply *independent of variable or value ordering heuristics.* (For unsatisfiable inputs, value ordering does not for matter for backtracking, but it does matter for better algorithms.) For a fully specified algorithm, it is often easy to construct hard inputs by tricking the ordering heuristic. This approach is inadequate in general, and we eliminate the issue by analyzing algorithms as families.

Method and Results

Here we address several of the most prominent methods in the literature. There are too many variants to be comprehensive, but our results apply to many others.

Let **R** be the following set of CSP algorithms.

{ Backtracking (BT)
Backmarking (BM) (Gashnig 1977),
Backjumping (BJ) (Gashnig 1978),
Graph-Based Backjumping (GBJ) (Dechter 1990),
Conflict-Directed Backjumping (CBJ)(Prosser 1993),
Forward Checking (FC) (Haralick and Elliot 1980),
Minimal Forward Checking (MFC)

[1] We consider only complete algorithms. Incomplete methods are useful, but are not our concern here.

(Dent and Mercer 1994),
FC with CBJ (FC-CBJ) (Prosser 1993),
Maintaining Arc Consistency (MAC)
(Sabin and Freuder 1994),
MAC with CBJ (MAC-CBJ) (Prosser 1995),
Learning (Dechter 1990)
Dynamic Backtracking (DB) (Ginsberg 1993) }

Let $Time_A(N)$ be the maximum execution time for algorithm A on an instance of size N. Recall that $\Omega(.)$ is the lower bound analog of $O(.)$. For $f, g : \mathbb{N} \mapsto \mathbb{N}$, $f(N) \in \Omega(g(N)) \Leftrightarrow \exists c > 0 \; s.t. \; \forall N > 0 \; f(N) \geq c \cdot g(N)$.

Theorem 1 *For every $A \in \boldsymbol{R}$, $Time_A(n) \in 2^{\Omega(n)}$.*

Theorem 1 says that for every algorithm in **R** there are infinitely many instances on which it takes takes time at least $2^{cn}, c > 0$ a constant, on every instance of size at least n. In fact, a single family of instances suffices for all the algorithms in **R**, and many others.

Proof: There are two main steps. First we define a class of "resolution bounded" CSP algorithms. The running time of a resolution bounded algorithm on instance Λ is at least as large as the length of a resolution refutation of a related formula $\phi(\Lambda)$. We then exhibit an infinite family of CSP instances, PH_n, such that any refutation of $\phi(\mathrm{PH}_n)$ requires exponentially many clauses. This gives us,

Lemma 1 *Every resolution bounded CSP algorithm takes time at least $2^{\Omega(n)}$ on the instance family PH_n.*

In the second step we show that each algorithm in **R** is resolution bounded. To do this we describe modified versions of the algorithms which have the same time complexity as the originals, but also construct the clauses needed for a refutation of $\phi(\Lambda)$. This step gives us the second lemma.

Lemma 2 *Every $A \in \boldsymbol{R}$ is resolution bounded.*

Theorem 1 follows from Lemmas 2 and 1, which together show that there is an infinite family of instances on which every algorithm in **R** takes time $2^{\Omega(n)}$. ∎

In the next section, we define "resolution bounded", specify the family PH_n, and prove Lemma 1. In the following section, we prove Lemma 2, considering the algorithms case-by-case.

CSP and Resolution Bounds

Finite Constraint Satisfaction

As usual a CSP is a triple $\Lambda = \langle X, D, C \rangle$, but we treat C unconventionally. X is a set of variables. D is a function mapping each variable x to a finite set $D(x)$.

An assignment γ for a set of variables $U \subset X$, is a set of atoms of the form $x{:}a$, where $x \in X$ and $a \in D(x)$, and if $x{:}a$, $x{:}b$ are both in γ, then $a = b$. We write $vars(\gamma)$ for $\{x | \exists a, x{:}a \in \gamma\}$. Let Γ denote the set of all assignments for subsets of X. We model the constraints abstractly, as a function $C : \Gamma \mapsto \Gamma$. For an assignment γ, $C(\gamma)$ is a minimal subset of γ which violates a constraint, or $\emptyset$. ($C(.)$ can easily be defined over the usual formalizations of the constraint set.) We say assignment γ for X satisfies Λ if $C(\gamma) = \emptyset$.

Formula Corresponding to a CSP

We use the following common (e.g., (De Kleer 1989)) reduction from CSP to SAT. For each variable $x \in X$, we have $|D(x)|$ propositional variables to represent assignments of values to x. We write $x{:}a$ for the literal which, when true, corresponds to the CSP variable x being assigned value a. (So $x{:}a$ represents an assignment to a variable in the context of Λ, but a literal in the context of $\phi(\Lambda)$.) Three sets of clauses encode the properties of a CSP solution.

1. For each variable of Λ, a clause saying it is assigned a value from its domain. We will refer to such clauses as *domain clauses*.
2. For each $U \subseteq X$ and each assignment γ to U, if $C(\gamma) \neq \emptyset$, a clause expressing the corresponding restriction on truth assignments.
3. For each variable of Λ, a set of clauses saying that it gets at most one value.

For CSP Λ, we define the CNF formula $\phi(\Lambda)$ to be

$$\phi(\Lambda) = \bigwedge_{x \in X} \left(\bigvee_{a \in D(x)} x{:}a \right) \wedge \bigwedge_{C(\gamma) \neq \emptyset, \gamma \in \Gamma} \left(\bigvee_{x:a \in C(\gamma)} \overline{x{:}a} \right)$$
$$\wedge \bigwedge_{x \in X} \left(\bigwedge_{a \neq c \in D(x)} \left(\overline{x{:}a} \bigvee \overline{x{:}c} \right) \right).$$

Clearly $\phi(\Lambda)$ is satisfiable if and only if Λ is satisfiable.

Resolution Bounded Algorithms

The resolution rule for propositional clauses is

$$\frac{(x \vee X) \quad (\neg x \vee Y)}{(X \vee Y)}.$$

$(X \vee Y)$ is called the resolvent. A *resolution derivation* of a clause c from a set of clauses Δ is a sequence of clauses $c_1 \ldots c_m$ such that $c_m = c$ and each c_i is either in Δ or is a resolvent of two clauses c_j, c_k with $j, k < i$. The derivation is of size m. A *refutation* of Δ is a derivation of the empty clause () from Δ. A CNF formula is unsatisfiable if and only if it has a refutation. Sometimes we say a formula ϕ has a proof of size m, meaning there is a refutation of ϕ of size m.

For our analysis, a CSP algorithm takes a step for each explicit assignment of a value to a variable. If substantial processing occurs between assignments, we also count key steps in the relevant sub-program.

Definition: We say a CSP algorithm A is *resolution bounded* if for every unsatisfiable instance Λ, the number of steps taken by A on instance Λ is at least as large as the number of clauses in the shortest refutation of the formula $\phi(\Lambda)$.

Pigeon Hole Inputs

The pigeonhole principle says that you can't stuff $n+1$ pigeons into n holes, unless you put more than one pigeon in some hole. Writing $[n]$ to denote the set

$\{1 \ldots n\}$, the formal statement is that there is no one-to-one onto function $f : [n+1] \mapsto [n]$. We can encode the statement that such a function does exist as an unsatisfiable CSP PH_n as follows.

$$
\begin{aligned}
X &= \{p_1, p_2, \ldots p_{n+1}\} \\
D(p_i) &= \{h_1, h_2, \ldots h_n\}, \forall p_i \in X \\
C(\gamma) &= \begin{cases} \{p_i{:}h_k, p_j{:}h_k\} & \text{if } \{p_i{:}h_k, p_j{:}h_k\} \subseteq \gamma, \\ & \quad \text{and } i \neq j, \\ \emptyset & \text{otherwise.} \end{cases}
\end{aligned}
$$

We can also encode this statement as a CNF formula, which we will call PHP_n.

$$
\mathrm{PHP}_n = \bigwedge_{1 \leq i \leq n+1} \left(\bigvee_{1 \leq j \leq n} p_i{:}h_j \right) \wedge \bigwedge_{\substack{1 \leq k \leq n \\ 1 \leq i < j \leq n+1}} \left(\overline{p_i{:}h_k} \vee \overline{p_j{:}h_k} \right)
$$

$$
\wedge \bigwedge_{1 \leq i \leq n+1} \left(\bigwedge_{1 \leq j \neq k \leq n} \left(\overline{p_i{:}h_j} \vee \overline{p_i{:}h_k} \right) \right)
$$

This is the same as the formula that we get if we apply the transformation above to the CSP formulation. That is,

$$\mathrm{PHP}_n = \phi(\mathrm{PH}_n)$$

Resolution Lower Bounds

The first super-polynomial lower bounds for unrestricted resolution were given by Haken(Haken 1985), who showed that for large n there are no resolution refutations of PHP_n of size smaller than $2^{\Omega(n)}$. A simplified proof, with a slightly stronger lower bound, was recently given in (Beame and Pitassi 1996)[2].

Theorem 2 (Haken / Beame and Pitassi) *For sufficiently large n, any resolution proof of PHP_n requires at least $2^{n/20}$ clauses.*

An examination of the proof shows that "sufficiently large" means $n \geq 1,867$. Unfortunately, space dictates that we only sketch the barest elements of the proof, which is not likely sufficient for the reader's intuition.

Proof ("Sketch"): Call a clause long if it has at least $n^2/10$ literals. Show that any refutation of PHP_n contains a long clause. Now, assume the existence of a refutation of PHP_n with fewer than $2^{n/20}$ clauses. By a process of successive restrictions it is possible to extract from this refutation a refutation of $\mathrm{PNP}_{n'}$, where $n > n' > 0.671n$, which does not contain a long clause, thereby obtaining a contradiction. ∎

Proof (of Lemma 1): Follows from theorem 2 and the definition of resolution bounded algorithm. ∎

Observe that the lower bound result does not follow merely from Haken's theorem plus a correspondence between the algorithm execution and resolution proofs. Since the mapping of CSP inputs to CNF formulas is not onto, it is not necessarily the case that every unsatisfiable CNF formula has a refutation corresponding to the execution of an algorithm at hand. Thus, one needs to establish a CSP input family that does map onto a suitable CNF formula. It is fortuitous that the PHn family behaves just as we want. To illustrate, if the only known resolution lower bounds were for random CNF formulas, the application of those bounds to CSP algorithms would be non-trivial.

Our lower bounds are expressed as an exponential in n, the number of holes, while the formula PHP_n is of size $\Theta(n^3)$. The exact nature of the lower bound depends on our assumptions about encoding of the input. Using N for input size, the lower bound for resolution proof size is $2^{\Omega(N^{1/3})}$. With a naive encoding, this will be the correct lower bound expressed in terms of input size. Since some more reasonable constraint languages can express PH_n in size linear in n, the implied lower bound for CSP algorithms is indeed $2^{\Omega(N)}$.

Algorithms and Refutations

Most complete CSP algorithms in the literature (including those in our set $\mathbf{R}$), are refinements to backtracking. Backtrack-based algorithms attempt construct a solution by incrementally extending an assignment to a subset of the variables. Whenever a constraint is violated, a recent extension is retracted, and a different extension attempted.

The execution of a backtrack-based CSP algorithm corresponds to a depth-first search of a rooted tree of partial assignments. Each node corresponds to a variable, and each outgoing edge corresponds to one assignment of a value to that variable. Let $\mathrm{T}(A, \Lambda)$ denote the search tree explored by algorithm A executing on unsatisfiable instance Λ. Associated with each node v is partial assignment γ_v, a variable x_v, and for each $a \in D(x_v)$, an outgoing edge corresponding to $x_v{:}a$. If $|D(x_v)| = k$, then v has k children, which we denote $v_1 \ldots v_k$, indexed in the order of execution.

The Refutation Corresponding to BT

The basic backtracking algorithm BT is given in Figure 1. The initial call is assumed to be $\mathrm{BT}(\Lambda, \{\})$. Note that BT is really a family of algorithms, since it does not specify which variable, or which value, to choose next, and each possible way of choosing these induces one particular algorithm.

```
Procedure BT(Λ,γ)
  if C(γ) ≠ ∅ then return UNSAT
  elsif vars(γ) = X, then return SAT
  else
    pick some x ∈ X − vars(γ)
    for a ∈ D(x)
      if BT( Λ, γ ∪ {x:a}) =SAT then return SAT.
    end for
  end if
  return UNSAT
```

Figure 1: Basic Backtracking Procedure, BT

[2]The cited papers use a slightly different formulation of PHP_n than we do, but the proof in (Beame and Pitassi 1996) goes through without modification for our version.

We label each node v of T(BT,Λ) with a clause c_v as follows. If v is a leaf, then γ_v violates a constraint, so let c_v be the clause from $\phi(\Lambda)$ expressing this constraint. If v is an internal node, then consider the k children of v, each of which is labeled with some clause c_i. If each of these clauses mentions x_v, then resolve all of these clauses against the domain clause for x_v and let c_v be the resulting clause. If some c_i does not mention x_v, and therefore the above sequence of resolution steps cannot be carried out, then let c_v be any such c_i, say the first one computed.

Lemma 3 *For every node v of T(BT,Λ), if c_v is the label of node v, and n_v is the number of nodes in the subtree rooted at v, then*

1) There is a resolution derivation of c_v from $\phi(\Lambda)$ of size less than n_v and,

2) $c_v \subseteq \{\overline{x{:}a} | x{:}a \in \gamma_v\}$. (I.e., c_v mentions a subset of assignments in γ_v.)

Proof: If v is a leaf then $C(\gamma_v) \neq \emptyset$. So we have that $c_v = \{\overline{x{:}a} | x{:}a \in C(\gamma_v)\} \subseteq \{\overline{x{:}a} | x{:}a \in \gamma_v\}$. Also $c_v \in \phi(\Lambda)$, so the number of resolution steps needed to derive it is $0 < 1$.
If v is an internal node, then each child v_i of v is labeled with some clause c_i which, by inductive hypothesis, has the desired properties. If each of the c_i includes some $\overline{x_v{:}a_i}$, then resolve all of these clauses against the domain clause for x_v, $(\vee_{a \in D(x_v)}\, x_v{:}a) \in \phi(\Lambda)$ to obtain c_v. This operation can be carried out, because all the c_i contain only negated literals, so all signs match up as required. Now, $c_v = \cup_i (c_i - \overline{x_v{:}a_i}) \subseteq \{\overline{x{:}a} | x{:}a \in \gamma_v\}$, and the number of resolvents is the sum of those to construct the child clauses plus the number of child clauses, which is less than n_v. If one or more of the c_i do not mention any $\overline{x_v{:}a_i}$, then let c_v be such a c_i with smallest index i, and we are done. ∎

Lemma 4 *BT is resolution bounded.*

Proof: Follows from Lemma 3 and the definition of resolution bounded. ∎

Figure 2 shows a modified backtracking algorithm BTR which generates the clauses for our refutation of $\phi(\Lambda)$. On a given instance, the execution of BT and BTR are exactly the same, except that when BT returns UNSAT, BTR returns $\langle$UNSAT; $c\rangle$, where c is the clause labeling the corresponding node in T(BT,Λ).

```
Procedure BTR(Λ,γ)
   if C(γ) ≠ ∅ then return ⟨UNSAT; {‾x:a‾ | x:a ∈ C(γ)}⟩
   elsif vars(γ) = X then return SAT
   else
      pick some x ∈ X − vars(γ)
      α ← (∨_{a∈D(x)} x:a)
      for a ∈ D(x)
         if BTR(Λ, γ ∪ {x:a}) = ⟨UNSAT; β⟩ then
            α ← res(α, β, ‾x:a‾).
         else
            return SAT.
         end if
      end for
   end if
   return ⟨UNSAT; α⟩
```

Where,

$$res(\alpha,\beta,a) = \begin{cases} \text{resolvent of } \alpha \text{ and } \beta, & \text{if } a \in \alpha \text{ and } \overline{a} \in \beta \\ \beta, & \text{if } a \in \alpha \text{ and } \overline{a} \notin \beta \\ \alpha, & \text{if } a \notin \alpha \end{cases}$$

Figure 2: Computing Resolvents while Backtracking

Backmarking

Backmarking (BM) (Gashnig 1977) is a technique to reduce the number of explicit consistency checks performed by backtracking. This does not affect the set of partial assignments examined, so resolution boundedness of BM follows from that of BT. (Explicit consistency checks are hidden in our function $C(.)$, since we can treat them as having no cost.)

Backjumping

Our labeling scheme for node v of the tree for BT allowed that a clause c_i from some child of a node v might not mention x_v. In this case, we labeled v with the first such c_i. Stated another way, at a point where the algorithm BTR is assigning values to variable x_v, one of the recursive calls may return a clause which does not mention any assignment to x_v. This will happen when x did not appear in any of the constraints violated at the leaves in the subtree rooted at v. In this case, none of the other clauses from children of v are used in the refutation. Since such a c_i is a certificate that no extension of γ_v can be a solution, there is no point in exploring other assignments to x_v without first backtracking and reassigning a "previous" variable.

The algorithm RBJ (for Resolvent-Based Backjumping) shown in Figure 3 refines BT in accordance with this observation. If the clause returned by a recursive call does not mention the current variable, then RBJ immediately returns, passing this derived clause up to previous invocations. This passing up continues until a node is reached with a branching variable that occurs in the clause.

```
Procedure RBJ(Λ,γ)
   if C(γ) ≠ ∅ then return ⟨UNSAT; {‾x:a‾ | x:a ∈ γ}⟩
   elsif vars(γ) = X then return SAT
   else
      pick some x ∈ X − vars(γ)
      α ← (∨_{a∈D(x)} x:a)
      for a ∈ D(x)
         R ← RBJ(Λ, γ ∪ {x:a})
         if R = ⟨UNSAT : β⟩ then
            if ‾x:a‾ ∈ β then
               α ← res(α, β, ‾x:a‾)
            else
               return ⟨UNSAT; β⟩
            end if
         else
            return SAT.
         end if
      end for
   end if
   return ⟨UNSAT; α⟩
```

Figure 3: RBJ: Resolvent-Based Backjumping

The strategy RBJ just described sounds like conflict-directed backjumping (CBJ), and indeed they are equivalent. The standard description of CBJ adapts BT by making use of a *conflict set*, conflicts(x), for each variable x. Each time the algorithm reaches a node where x will be assigned values, the set conflicts(x) is set to $\emptyset$. When assigning a value to x violates a constraint with some already-assigned variable y, y is added to conflicts(x), and a new value is tried for x. If all values for x have been tried, then CBJ jumps back to the "latest" (i.e., most recently set) variable in conflicts(x). If this variable is y, then conflicts(y) is set to conflicts(y)$\cup$ conflicts(x) $- y$.

Lemma 5 *RBJ and CBJ are resolution bounded.*

Proof: It is clear that the lemma is true for RBJ, so it remains to show that RBJ and CBJ are equivalent. For this, it is sufficient to show that, when both algorithms are at nodes with the same corresponding truth assignments, they agree on whether to jump back or search another branch.
Label the tree T(CBJ,Λ), as follows. When CBJ backjumps from node v to node h, label each vertex on the path $v \ldots h$ with the set conflicts(x_v). Consider two nodes v, u of T(CBJ,Λ) and T(RBJ,Λ), respectively, such that $x_v = x_u$ and $\lambda_v = \lambda_u$. CBJ and RBJ do the same thing provided the label of v is the same $vars(c_u)$. If v is a leaf, this obvious. Assume v is not a leaf. Then if some child has a label which does not mention x_v, the label of v is the first such label encountered, and otherwise is is the union of all labels from v's children, less x_v. The clause c_u is constructed in the same manner from clauses labeling its children, so by inductive hypothesis we are done. ∎

Lemma 6 *Backjumping (BJ) and Graph-Based Backjumping (GBJ) (Gashnig 1978; Dechter 1990) are resolution bounded.*

Proof: The sub-trees pruned from the BT tree by BJ and GBJ are a subset of those pruned by RBJ. ∎

Forward Checking

Forward Checking (FC) modifies BT to reduce the number of nodes searched by doing extra consistency checks. When a variable is assigned a value, FC checks the constraints of all unassigned variables, and removes from their domains any values not consistent with the current assignment. If the domain of some future variable becomes empty, the current assignment is inconsistent, and backtracking takes place.

Our modified FC works as follows. For each variable x, we maintain a clause δ_x, which corresponds to the domain reductions that FC has performed on x. Initially $\delta_x = (\vee_{a \in D(x)} x{:}a)$. At a node v, when we assign the value a to x_v, we check the consistency of $x_v{:}a$ with each value c available for each unassigned variable y. If $C(\{x_v{:}a, y{:}c\}) \neq \emptyset$, then we let $\delta_y = Res(\delta_y, (\overline{x_v{:}a}, \overline{y{:}c}))$, and remove c from $D(y)$. (These two operations are undone when backtracking occurs.) If at some point $D(y)$ becomes empty, then δ_y no longer mentions y at all, but consists of a set of negated atoms which is falsified by the current assignment. Thus, the algorithm returns $\delta(y)$ as the clause for the current node.

Lemma 7 *FC and MFC are resolution bounded.*

Proof: Our modified FC simulates the original, and constructs resolvents from clauses in $\phi(\Lambda)$ such that whenever FC returns earlier than BT because of an empty domain, a resolvent is available for use in the refutation which is falsified by the current assignment. This clause is one of the possible clauses that could be returned (depending ordering strategy) by BT in searching the eliminated subtree.
The domain reductions made at each node by Minimal Forward Checking are a subset of those made FC, and can be carried out with a similar strategy. ∎

Arc Consistency

A CSP instance is arc consistent if, for every pair of variables x and y, for every $a \in D(x)$, there is at least one $c \in D(y)$ which is consistent, i.e., such that $C(\{x{:}a, y{:}c\}) = \emptyset$. We say that c supports $x{:}a$ at y. If there is no value c that supports $x{:}a$ at y, then no solution gives x the value a. Arc consistency filtering transforms a CSP instance into an arc consistent instance by repeatedly deleting domain elements with no support at some other variable.

We use Mohr and Henderson's AC-4 arc consistency algorithm, which has running time of optimal order(Mohr and Henderson 1986). Figure 4 shows our version, which constructs the set of clauses we need. The scheme is a generalization of that used for FC, and can in fact be further generalized to k-consistency filtering for arbitrary k.

The algorithm uses 4 data structures. $Supports[x{:}a]$ is the set of $y{:}c$ supported by a at x. $Number[x{:}a, y]$ stores the number of values which support $x{:}a$ at y. $Clause[x{:}a, y]$ is a clause which reflects the changes made to $Number[x{:}a, y]$. Initially $Clause[x{:}a, y] = (\vee_{c \in D(y)} y{:}c)$, the domain clause for y. There is also a stack S, which stores derived atoms which must be propagated. We assume the set E of pairs (x, y) of variables between which there is a constraint is available. $Supports[x{:}a]$ and E are used for efficiency, but have no role in constructing resolvents.

The initial pruning stage works as follows. For each $x{:}a$, it checks every $y{:}c$ to see how many values b support $x{:}a$ at y. Whenever $C(\{x{:}a, y{:}c\}) = \emptyset$, c supports $x{:}a$ at y. Whenever $C(\{x{:}a, y{:}c\}) \neq \emptyset$, the clause $Clause[x{:}a, y]$ is resolved with $\{\overline{x{:}a}, \overline{y{:}c}\}$, and the result becomes the new value of $Clause[x{:}a, y]$. After performing all checks for $x{:}a$, $Number[x{:}a, y]$ is the same as the number of occurrences of y in $Clause[x{:}a, y]$. If $Number[x{:}a, y] = 0$, then a is removed from $D(x)$ and $\overline{x{:}a}$ is pushed onto the stack. If this happens, $Clause[x{:}a, y] = (\overline{x{:}a})$, so the new domain clause for x, which does not include a, can be obtained by one more resolution step.

```
Procedure AC-4R(Λ)
  // initialize data structures
  for x ∈ X, a ∈ D(x)
    Supports[x:a] ← ∅
    for (x, y) ∈ E
      Number[x:a, y] ← 0
      Clause[x:a, y] ← (∨_{c∈D(y)} y:c)
    end for
  end for

  // initial pruning
  for (x, y) ∈ E
    for a ∈ D(x), c ∈ D(y)
      if C({x:a, y:c}) ≠ ∅ then
        Number[x:a, y] += 1
        Supports[y:c] ∪= {x:a}
      else
        Clause[x:a, y] ← Res(Clause[x:a, y], (¬x:a ∨ ¬y:c))
      end if
    end for
    if Number[x:a, y] = 0 then
      D(x) ← D(x) − {a}
      S.Push(Clause[x:a, y])
    end for
  end for

  // propagation
  while not(Empty)
    (¬y:c) ← S.Pop()
    for x:a ∈ Supports[y:c]
      Number[x:a, y] −= 1
      Clause[x:a, y] ← Res(Clause[x:a, y], (¬y:c))
      if Number[x:a, y] = 0 and a ∈ D(x) then
        D(x) ← D(x) − {a}
        S.Push(Clause[x:a, y])
      end if
    end for
  end while
```

Figure 4: AC-4R

In the propagation stage, negated atoms – which identify values which have been eliminated from domains – are popped from the stack. When $\overline{y{:}c}$ is popped, $Number[x{:}a, y]$ and $Clause[x{:}a, y]$ are modified for each $x{:}a \in Supports[y{:}c]$. Further domain reductions may occur, with attendant computation of resolvents, and their effects also propagated. When the propagation phase completes, we have one of two possibilities. If some domain has been made empty, then the empty clause has been derived by resolution. Otherwise, we have an arc-consistent version Λ' of the original instance Λ, and a resolution derivation of the clauses in $\phi(\Lambda')$ from the clauses in $\phi(\Lambda)$.

A number of algorithms employ arc consistency filtering during backtracking, either at selected nodes or at all nodes (Nadel 1989; Sabin and Freuder 1994; Prosser 1995, for example). We will consider a generic and unrefined version of this idea, from which it should be clear that the same idea applies to most if not all of the variants in the literature.

Lemma 8 *Any algorithm which consists of backtracking (possibly with backjumping refinements) modified by performing k-consistency filtering before some or all branching operations, then A is resolution bounded.*

Proof (Sketch): Consider a path from the root to a leaf in the search tree for the algorithm. As the search progresses down the path, assignments to variables are made, and simplified instances are created by consistency filtering. The result of consistency filtering is reduced domains for some variables. First prove by induction downward that by carrying out the sequence of resolution steps specified by AC-4R (or a generalization of it for $k > 2$), we can derive shortened domain clauses corresponding to all reductions in domains performed by the consistency filtering operations. Then construct the refutation by induction from the bottom up as before. Because reduced domain clauses are available, there is no need for clauses from the sub-trees pruned by consistency filtering.
Note one subtlety in the downward construction. We cannot derive exactly the clauses that correspond to the reduced instance, since we cannot directly model the assignment operation with a resolution step. We carry out the sequence of resolution steps which correspond to consistency filtering operations as described for AC-4R, but the propositions corresponding to assignments made on the path still appear in the "reduced domain clauses". Fortunately, these are handled by the upward construction of the refutation, as in the versions without consistency filtering. ■

Storing No-goods

A number of CSP algorithms make use of recording no-goods – often called learning – while doing backtracking ((Dechter 1990) for example).

Lemma 9 *Any of the above methods, when enhanced by no-good recording, is resolution bounded.*

Proof (Idea): A no-good corresponds exactly to a negative clause generated while constructing our refutation. Storing no-goods corresponds to caching these resolvents, and checking them against the current assignment each time an extension is made to it. ■

Lemma 10 *Dynamic Backtracking (Ginsberg 1993) is resolution bounded.*

Proof (Idea): Storing no-goods is the primary control mechanism of dynamic backtracking. Space precludes a proof here, but a proof in our style can be easily obtained by adapting Baker's proof of a slightly different property, and Ginsberg's completeness proof (Baker 1995; Ginsberg 1993). ■

Discussion

Related Work

We should begin by noting that even stronger lower bounds than ours (something like $n!$), can be obtained for some of these algorithms by applying known properties of local consistency (Dechter 1992, qv.). Our interest in the proof-based analysis is that we believe it can be more generally applied, and that it tells us something different about the algorithms (even when studying the same inputs).

There are many connections between resolution and backtracking algorithms in the literature. For propositional satisfiability, which is equivalent to CSPs with domain size two, the correspondence between resolution and the Davis-Putnam procedure is well known. de Kleer(De Kleer 1989) points out that (paraphrasing) if Λ' is the result of applying k-consistency filtering to Λ, then $\phi(\Lambda')$ can be obtained from $\phi(\Lambda)$ by resolution. However, his paper does not address complexity. Our point is that carrying out the resolution steps has computationally the same cost as running an optimal consistency filtering algorithm. Mackworth (Mackworth 1991) discusses logical representations of CSP instances and resolution, but does not draw connections to standard CSP algorithms or discuss algorithm complexity. Bayardo and Schrag (Bayardo Jr. and Schrag 1997) employed the relation between resolvents and CBJ for satisfiability, but did not discuss it in the context of general CSPs. The correspondence between no-goods and negative clauses appears, at least implicitly, in many papers.

The work closest to ours in both spirit and technique is in Chapter 5 of Baker's thesis (Baker 1995). There lower bounds are shown for backtracking (plus learning and backjumping) and dynamic backtracking using inputs with bounded induced width (Dechter and Pearl 1988). Since the PH_n inputs have unbounded width, these bounds provide information ours don't. However, the bounds obtained are weaker than ours ($\mathrm{O}(n^{logn})$), and apply to fewer algorithms. (Since the inputs used have short proofs, some resolution bounded procedures will solve them in polytime). Moreover, the inputs used to obtain these results are less natural than PH_n (see below). Baker defines a CSP variant of resolution, based only on no-good clauses, and then uses restricted versions of this system to obtain the lower bounds. The correspondence to propositional resolution is easy to see, not all algorithms will have convenient simulations, and the simulations for some will not be as tight as ours (e.g., the analogous notion of resolution-bounded is weaker), since not all propositional refutations can be represented.

Some Practical Considerations

A reasonable treatment of the relevance of these bounds to actually using algorithms in practice is beyond the scope of this paper, but we have a few brief comments to make along these lines.

Asymptotic results sometimes tell us little about instances of practical size, but these results are not asymptotic (other than that we use the standard asymptotic notation). The lower bound from (Beame and Pitassi 1996) holds for $n > 1,867$, which is not large for a real-world problem. $\mathrm{PH}_{1,867}$ requires at least $2^{94} \approx 10^{29}$ steps, which is beyond hopeless for the for-seeable future. Moreover, this proven bound is likely much smaller than the best that can be achieved in practice. The shortest known refutation for PHP_n requires $(n-1)(n+2)2^{n-3}$ clauses, which even for PH_{100} is more than 10^{33} steps.

While it may be unlikely that one encounters PH_n inputs in practice, the problem is in essence quite natural. Whenever a problem requires finding a matching between two sets, a pigeonhole-like sub-problem can arise. Such matchings are central to a whole range of assignment, timetabling and scheduling problems. In hard instances of such problems, an unsolvable matching problem is exactly what we expect to obtain after a few bad choices have been made. It is important here to realize that the difficulty of the PH_n problems is much more general than this particular encoding of matching. For example, it sounds easier to spot the mistake of trying to squeeze fat too many pigeons into a set of holes, rather than just one too many. But in fact, for resolution-bounded algorithms, it is just as hard (Buss and Turán 1988) Moreover, the same problem arises for more general matching problems to the extent that even having a solution to PH_n supplied for free (as an axiom or sub-routine, say) does not help (Ajtai 1988; Beame and Pitassi 1993).

Determining the extent to which problems of this sort actually occur in applications is an empirical question we cannot address here, but the proliferation of workshops on solving hard instances of the sorts of problems mentioned above is suggestive.

Proofs and Algorithms

Most of this paper is devoted to analysis of common CSP algorithms in terms of propositional calculus and resolution proofs, and using this analysis to prove exponential lower bounds for the algorithms. However, the technique of proof-based analysis can be applied to much more than what we have covered here. One role is a tool in understanding why certain algorithms do badly. Although there is considerable range in sophistication of the algorithms we have looked at, in some sense they all do badly for the same reason. Of course, they have important differences, which are reflected by other inputs, and even here we can learn something from proofs. For example, BT and CBJ correspond to the restriction to tree-like proofs, whereas methods which use consistency filtering or no-good recording have additional power which can be obtained by constructing DAG-like proofs.

From our speculations in the previous subsection it would follow that the resolution-bounded nature of almost all current algorithms is a major obstacle to progress. We conjecture that performance improvements of more than an order of magnitude beyond the current state, on a wide range of instances, will not occur until we break the "resolution barrier". (To see what sort of challenge this is, try to think up an algorithm which could conceivably be practical, but which you are convinced is not resolution bounded.)

A nice thing about viewing algorithms as proofconstructors is that there is a wide range of other proof systems available, most of which are more powerful than resolution. Most of these are not reasonable candidates for practical implementation, but they do provide a suite of potential tools for development.

The prevalence of resolution is in part a result of it's close relationship to backtracking, and in part a result of it being a week enough proof system that it is not too hard to find complete strategies which are can be practically implemented. Much stronger proof systems, such as extended resolution (for which no non-trivial lower bounds are known), do not offer much promise for practical implementation. In the middle ground are systems like the cutting planes system, which is stronger than resolution in that there are polynomial size proofs for the pigeonhole formulas, yet week enough that it is possible to base practical algorithms on it (as demonstrated by the mathematical programming community).

Concluding Remarks

We have shown, for a wide range CSP algorithms, a correspondence between execution on unsatisfiable instances (or sub-instances) and the construction of a resolution refutation for a related propositional formula. Using this property, we have shown that the simple family of instances PH_n, which encodes a version of the pigeonhole principle, induces exponential lower bounds on the runtime of the algorithms. The algorithms addressed include many of the most popular and successful algorithms in the literature.

For veterans in the area none of this will be surprising. Our main goal has been to make more explicit and precise the basics of what we consider to be a powerful but very under-utilized tool. There is probably more to be learned via this route even for existing algorithms. More importantly, we conjecture that what it has told us already has significant implications for how we should be approaching the task of designing better algorithms. That is, that the study of propositional proof systems and their complexity should be an integral part of this activity.

References

M. Ajtai. The complexity of the pigeonhole principle. In *Proc. FOCS-88*, pages 346–355, 1988.

Andrew B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.* PhD thesis, University of Oregon, 1995.

Roberto J. Bayardo Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. AAAI-97*, 1997.

Paul Beame and Toniann Pitassi. An exponential separation between the matching principle and the pigeonhole principle. Technical Report 93-04-07, Dept. of Computer Science and Engineering, University of Washington, 1993.

Paul Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *Proc. FOCS-96*, pages 274–282, 1996.

Samuel R. Buss and György Turán. Resolution proofs of generalized pigeonhole principles. *Theoretical Computer Science*, 62:311–317, 1988.

Johann De Kleer. A comparison of ATMS and CSP techniques. In *Proc. IJCAI 89*, pages 290–296, 1989.

R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

Rina Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.

M.J. Dent and R.E. Mercer. Minimal forward checker. In *Proceedings, 6th INtl. Conf. on Tools with Artificial Intelligence.*, pages 432–438, New Orleans, 1994.

J. Gashnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings,of the Intl. Joint Conf on Artificial Intelligence*, page 457, 1977.

J. Gashnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In *Proc. of the Canadian Artificial Intelligence Conference*, pages 268–277, 1978.

Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satsifaction problems. *Artificial Intelligence*, 14:263–313, 1980.

A.K. Mackworth. The logic of constraint satisfaction. Technical Report 91-26, Department of Computer Science, University of British Columbia, 1991.

R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

Bernard A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).

Patrick Prosser. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report 95/177, Dept. of Comp. Sci., Unversity of Strathclyde, 1995.

Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proc. PPCP-94. Appears as LNCS-874.*, pages 10–19, 1994. Also appears in Proc. ECAI-94.