Local Consistency and SAT-Solvers

Justyna Petke and Peter Jeavons

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, UK
{justyna.petke,Peter.Jeavons}@comlab.ox.ac.uk

Abstract. In this paper we show that the power of using k-consistency techniques in a constraint problem is precisely captured by using a particular inference rule, which we call positive-hyper-resolution, on the direct Boolean encoding of the CSP instance. We also show that current clause-learning SAT-solvers will deduce any positive-hyper-resolvent of a fixed size from a given set of clauses in polynomial expected time. We combine these two results to show that, without being explicitly designed to do so, current clause-learning SAT-solvers efficiently simulate k-consistency techniques, for all values of k. We then give some experimental results to show that this feature allows clause-learning SAT-solvers to efficiently solve certain families of CSP instances which are challenging for conventional CP solvers.

1 Introduction

One of the oldest and most central ideas in constraint programming, going right back to Montanari's original paper in 1974 [22], is the idea of using *local consistency* techniques to prune the search space [11]. The idea of arc-consistency was introduced in [21], and generalised to k-consistency in [16]. Modern constraint solvers generally employ specialised propagators to prune the domains of variables to achieve some form of generalised arc-consistency, but do *not* attempt to enforce higher levels of consistency, such as path-conistency.

By contrast, the software tools developed to solve propositional satisfiability problems, known as SAT-solvers, generally use logical inference techniques, such as unit propagation and clause-learning, to prune the search space.

One of the most surprising empirical findings of the last few years has been the remarkably good performance of general SAT-solvers in solving constraint satisfaction problems. To apply such tools to a constraint satisfaction problem one first has to translate the instance into a set of clauses using some form of Boolean encoding [26,27]. Such encoding techniques tend to obscure the structure of the original problem, and may introduce a very large number of Boolean variables and clauses to encode quite easily-stated constraints. Nevertheless, in quite a few cases, such approaches have out-performed more traditional constraint solving tools [4,3,24].

In this paper we draw on a number of recent analytical approaches to try to account for the good performance of general SAT-solvers on many forms of constraint problems. Building on the results of [6,7,18], we show that the power of using k-consistency techniques in a constraint problem is precisely captured by using a single inference rule in a standard Boolean encoding of that problem. We refer to this inference rule as *positive-hyper-resolution*, and show that any conclusions deduced by enforcing k-consistency can be deduced by a sequence of positive-hyper-resolution inferences involving Boolean clauses in the original instance and positive-hyper-resolvents with at most k literals. Furthermore, by using the approach of [5] and [25], we show that current clause-learning SAT-solvers will make all such deductions in polynomial expected time, even with a random branching strategy. Hence we show that, although they are not explicitly designed to do so, running a clause-learning SAT-solver on the simplest encoding of a constraint problem efficiently simulates the effects of enforcing k-consistency for *all* values of k.

2 Preliminaries

Definition 2.1. *An instance of the* **Constraint Satisfaction Problem** *(CSP) is specified by a triple* (V, D, C), *where*

- V *is a finite set of* variables*;*
- $D = \{D_v \mid v \in V\}$ *where each set* D_v *is the set of possible values for the variable* v, *called the* domain *of* v*;*
- C *is a finite set of* constraints. *Each constraint in* C *is a pair* (R_i, S_i) *where*
  - S_i *is an ordered list of* m_i *variables, called the constraint* scope*;*
  - R_i *is a relation over* D_v *of arity* m_i, *called the constraint* relation.

Given any CSP instance (V, D, C), a *partial assignment* is a mapping f from some subset W of V to $\bigcup D$ such that $f(v) \in D_v$ for all $v \in W$. A partial assignment *satisfies the constraints* of the instance if, for all $(R, (v_1, v_2, \ldots, v_m)) \in C$ such that $v_j \in W$ for $j = 1, 2, \ldots, m$, we have $(f(v_1), f(v_2) \ldots, f(v_m)) \in R$. A partial assignment that satisfies the constraints of an instance is called a *partial solution*[1] to that instance. The set of variables on which a partial assignment f is defined is called the domain of f, and denoted $Dom(f)$. A partial solution f' *extends* a partial solution f if $Dom(f') \supseteq Dom(f)$ and $f'(v) = f(v)$ for all $v \in Dom(f)$. A partial solution with domain V is called a solution.

One way to derive new information about a CSP instance, which may help to determine whether or not it has a solution, is to use some form of constraint propagation to enforce some level of *local consistency* [11]. For example, it is possible to use the notion of *k-consistency*, as in the next definition. We note that there are several different but equivalent ways to define and enforce k-consistency described in the literature [11,13,16]. Our presentation follows [6], which is inspired by the notion of existential k-pebble games introduced by Kolaitis and Vardi [20].

[1] Note that not all partial solutions extend to solutions.

Definition 2.2. *[6] For any CSP instance P, the* **k-consistency closure of** P *is the set H of partial assignments which is obtained by the following algorithm:*

1. *Let H be the collection of all partial solutions f of P with $|Dom(f)| \leq k+1$;*
2. *For every $f \in H$ with $|Dom(f)| \leq k$ and every variable v of P, if there is no $g \in H$ such that g extends f and $v \in Dom(g)$, then remove f and all its extensions from H;*
3. *Repeat step 2 until H is unchanged.*

Note that computing the k-consistency closure according to this definition corresponds precisely to enforcing *strong $k+1$-consistency* according to the definitions in [11,13,16].

Throughout this paper, we shall assume that the domain of possible values for each variable in a CSP instance is finite. It is straightforward to show that for any fixed k, and fixed maximum domain size, the k-consistency closure of an instance P can be computed in polynomial time [6,13].

Note that any solution to P must extend some element of the k-consistency closure of P. Hence, if the k-consistency closure of P is empty, for some k, then P has no solutions. The converse is not true in general, but it holds for certain special cases, such as the class of instances whose structure has tree-width bounded by k [6], or the class of instances whose constraints are "0/1/all relations", as defined in [14], or "connected row-convex" relations, as defined in [15]. For these special kinds of instances it is possible to determine in polynomial time whether or not a solution exists simply by computing the k-consistency closure, for an appropriate choice of k. Moreover, if a solution exists, then it can be constructed in polynomial time by selecting each variable in turn, assigning each possible value, re-computing the k-consistency closure, and retaining an assignment that gives a non-empty result.

The following result gives a useful condition for determining whether the k-consistency closure of a CSP instance is empty.

Lemma 2.3. *[20] The k-consistency closure of a CSP instance P is non-empty if and only if there exists a non-empty family H of partial solutions to P such that:*

1. *If $f \in H$, then $|Dom(f)| \leq k+1$;*
2. *If $f \in H$ and f extends g, then $g \in H$;*
3. *If $f \in H$, $|Dom(f)| \leq k$, and $v \notin Dom(f)$ is a variable of P, then there is some $g \in H$ such that g extends f and $v \in Dom(g)$.*

A set of partial solutions H satisfying the conditions described in Lemma 2.3 is sometimes called a *strategy* for the instance P [9,20].

One possible approach to solving a CSP instance is to encode it as a propositional formula over a suitable set of Boolean variables, and then use a program to decide the satisfiability of that formula. Many such programs, known as SAT-solvers, are now available and can often efficiently handle problems with thousands, or sometimes even millions, of Boolean variables [29].

Several different ways of encoding a CSP instance as a propositional formula have been proposed [26,27]. Here we consider only a very straightforward encoding, known as the *direct encoding.* In this encoding, for a CSP instance $P = (V, D, C)$ we introduce a set of Boolean variables of the form x_{vi} for each $v \in V$ and each $i \in D_v$. The Boolean variable x_{vi} will be assigned *True* if and only if the original variable v is assigned the value i. To ensure that at least one value is assigned to each variable v, we include the clause $\bigvee_{i \in D_v} x_{vi}$. To ensure that at most one value is assigned to each variable v, we include all binary clauses of the form $\neg x_{vi} \vee \neg x_{vj}$ for all $i, j \in D_v$ with $i \neq j$. Finally, to ensure that each constraint $(R, S) \in C$ is satisfied, we include a clause $\bigvee_{v \in S} \neg x_{vf(v)}$ for each partial assignment f that does *not* satisfy the constraint.

Given any set of clauses we can often deduce further clauses by applying certain *inference rules.* For example, if we have two clauses of the form $C_1 \vee x$ and $C_2 \vee \neg x$, for some (possibly empty) clauses C_1, C_2, and some variable x, then we can deduce the clause $C_1 \vee C_2$. This form of inference is known as *propositional resolution*; the resultant clause is called the *resolvent* [12].

In the next section, we shall establish a close connection between the k-consistency algorithm and a form of inference called positive-hyper-resolution, which we define as follows:

Definition 2.4. *If we have a collection of clauses of the form* $C_i \vee \neg x_i$ *for* $i = 1, 2, \ldots, r$*, where each* x_i *is a Boolean variable, and a purely positive clause* $x_1 \vee x_2 \vee \cdots \vee x_r$*, then we can deduce the clause* $C_1 \vee C_2 \vee \cdots \vee C_r$*.*

We call this form of inference **positive-hyper-resolution** *and the resultant clause* $C_1 \vee C_2 \vee \cdots \vee C_r$ *the* positive-hyper-resolvent*.*

Note that positive-hyper-resolution is equivalent to a sequence of standard resolution steps. The reason for introducing positive-hyper-resolution is that it allows us to deduce the clauses we need in a single step without needing to introduce intermediate clauses (which may be longer than the positive-hyper-resolvent). By restricting the size of the clauses we use in this way we are able to obtain better performance bounds for the SAT-solvers.

A positive-hyper-resolution *derivation* of a clause C from a set of initial clauses Φ is a sequence of clauses $C_1, C_2, \ldots, C_m$, where $C_m = C$ and each C_i follows by the positive-hyper-resolution rule from some collection of clauses, each of which is either contained in Φ or else occurs earlier in the sequence. The *width* of this derivation is defined to be the maximum size of any of the clauses C_i. If C_m is the empty clause, then we say that the derivation is a *positive-hyper-resolution refutation* of Φ.

3 k-Consistency and Positive-Hyper-Resolution

It has been pointed out by many authors that enforcing local consistency is a form of inference on relations analogous to the use of the resolution rule on clauses [8,11,12,18,19]. On the direct encoding of a CSP instance, our positive-hyper-resolution rule corresponds to the “nogood resolution” rule defined in [18].

The precise strength of the standard resolution inference rule on the direct encoding of a CSP instance was considered in [26], where it was shown that *unit* resolution (where one of the clauses being resolved consists of a single literal), corresponds to enforcing a weak form of local consistency known as *forward checking.* In [18] it was pointed out that the standard resolution rule with no restriction on clause length is able to simulate all the inferences made by a k-consistency algorithm. In [7] it was shown that the standard resolution rule restricted to clauses with at most k literals can be characterised in terms of the Boolean existential $(k+1)$-pebble game. It follows that on CSP instances with Boolean domains this form of inference corresponds to enforcing k-consistency.

Here we extend these results a little, to show that for CSP instances with arbitrary finite domains, applying the positive-hyper-resolution rule on the direct encoding to obtain clauses with at most k literals corresponds precisely to enforcing k-consistency. Note that the bound, k, that we impose on the size of the positive-hyper-resolvents, is independent of the domain size. In other words, using this inference rule we only need to consider inferred clauses of size at most k, even though we make use of clauses in the encoding whose size is equal to the domain size, which may be arbitrarily large.

Theorem 3.1. *The k-consistency closure of a CSP instance P is empty if and only if its direct encoding as a set of clauses has a positive-hyper-resolution refutation of width at most k.*

The proof is broken down into two lemmas inspired by Lemmas 2 and 3 in [7].

Lemma 3.2. *Let P be a CSP instance, and let Φ be its direct encoding as a set of clauses. If Φ has no positive-hyper-resolution refutation of width k or less, then the k-consistency closure of P is non-empty.*

Proof. Let V be the set of variables of P, where each $v \in V$ has domain D_v, and let $X = \{x_{vi} \mid v \in V, i \in D_v\}$ be the corresponding set of Boolean variables in Φ. Note that the clauses in Φ are either of the form $\bigvee_{i \in D_v} x_{vi}$ for some $v \in V$, or else consist entirely of negative literals. Let Γ be the set of all clauses having a positive-hyper-resolution derivation from Φ of width at most k. By the definition of positive-hyper-resolution and the observation about Φ, every clause in Γ consists entirely of negative literals.

Now let H be the set of all partial assignments for P with domain size at most $k+1$ that do not falsify any clause in $\Phi \cup \Gamma$ (under the direct encoding).

Consider any element $f \in H$. By the definition of H, f does not falsify any clause of Φ, so by the definition of the direct encoding, every element of H is a partial solution to P. Furthermore, if f extends g, then g is also an element of H, because g makes fewer assignments than f and hence cannot falsify any additional clauses to f.

If Φ has no positive-hyper-resolution refutation of width at most k, then Γ does not contain the empty clause, so H contains (at least) the partial solution with empty domain, and hence H is not empty.

Now let f be any element of H with $|Dom(f)| \leq k$ and let v be any variable of P that is not in $Dom(f)$. For any partial assignment g that extends f and has $Dom(g) = Dom(f) \cup \{v\}$ we have that either $g \in H$ or else there exists a clause in $\Phi \cup \Gamma$ that is falsified by g. Since g is a partial assignment, any clause C in $\Phi \cup \Gamma$ that is falsified by g, must consist entirely of negative literals. Hence the literals of C must either be of the form $\neg x_{wf(w)}$ for some $w \in Dom(f)$, or else $\neg x_{vg(v)}$. Moreover, any such clause must contain the literal $\neg x_{vg(v)}$, or else it would already be falsified by f.

Assume, for contradiction, that H does not contain any assignment g that extends f and has $Dom(g) = Dom(f) \cup \{v\}$. In that case, we have that, for each $i \in D_v$, $\Phi \cup \Gamma$ contains a clause C_i consisting of negative literals of the form $\neg x_{wf(w)}$ for some $w \in Dom(f)$, together with the literal $\neg x_{vi}$. Now consider the clause, C, which is the positive-hyper-resolvent of these clauses C_i and the clause $\bigvee_{i \in D_v} x_{vi}$. The clause C consists entirely of negative literals of the form $\neg x_{wf(w)}$ for some $w \in Dom(f)$, so it has width at most $|Dom(f)| \leq k$, and hence is an element of Γ. However C is falsified by f, which contradicts the choice of f. Hence we have shown that for all $f \in H$ with $|Dom(f)| \leq k$, and for all $v \in V$, there is some $g \in H$ such that g extends f and $v \in Dom(g)$.

We have shown that H satisfies all the conditions required by Lemma 2.3, so we conclude that the k-consistency closure of P is non-empty. □

Lemma 3.3. *Let P be a CSP instance, and let Φ be its direct encoding as a set of clauses. If the k-consistency closure of P is non-empty, then Φ has no positive-hyper-resolution refutation of width k or less.*

Proof. Let V be the set of variables of P, where each $v \in V$ has domain D_v, and let $X = \{x_{vi} \mid v \in V, i \in D_v\}$ be the corresponding set of Boolean variables in Φ.

By Lemma 2.3, if the k-consistency closure of P is non-empty, then there exists a non-empty set H of partial solutions to P which satisfies the three properties described in Lemma 2.3.

Now consider any positive-hyper-resolution derivation Γ from Φ of width at most k. We show by induction on the length of this derivation that the elements of H do not falsify any clause in the derivation. First we note that the elements of H are partial solutions, so they satisfy all the constraints of P, and hence do not falsify any clause of Φ. This establishes the base case. Assume, for induction, that all clauses in the derivation earlier than some clause C are not falsified by any element of H.

Since the clauses in Φ are either of the form $\bigvee_{i \in D_v} x_{vi}$ for some $v \in V$, or else consist entirely of negative literals, it follows that any clause in the derivation obtained by positive-hyper-resolution consists entirely of negative literals.

If $f \in H$ falsifies $C \in \Gamma$, then the literals of C must all be of the form $\neg x_{vf(v)}$, for some $v \in Dom(f)$. Hence we may assume, without loss of generality, that C is the positive-hyper-resolvent of a set of clauses $\Delta = \{C_i \vee \neg x_{vi} \mid i \in D_v\}$ and the clause $\bigvee_{i \in D_v} x_{vi}$. Since the width of the derivation is at most k, C contains at most k literals, and hence we may assume that $|Dom(f)| \leq k$. But then, by the choice of H, there must exist some extension g of f in H such that $v \in Dom(g)$. Any such g will falsify some clause in Δ, which contradicts our inductive hypothesis. Hence no $f \in H$ falsifies C, so C cannot be empty.

It follows that no positive-hyper-resolution derivation of width at most k can contain the empty clause. □

4 Positive-Hyper-Resolution and SAT-Solvers

In this section we adapt the machinery of [5] and [25] to show that for any fixed k, the existence of a positive-hyper-resolution refutation of width k is likely to be discovered by a SAT-solver in polynomial-time using standard clause learning and restart techniques, even with a totally random branching strategy.

Note that previous results about the power of clause-learning SAT-solvers have generally assumed an optimal branching strategy [10,25] - they have shown what solvers are potentially capable of doing, rather than what they are likely to achieve in practice. The exception is [5], which gives an analysis of likely behaviour, but relies on the existence of a standard resolution proof of bounded width. Here we show that the results of [5] can be extended to hyper-resolution proofs, which can be much shorter and narrower than their associated standard resolution proofs.

We will make use of the following terminology from [5]. For a clause C, a Boolean variable x, and a truth value $a \in \{0, 1\}$, the restriction of C by the assignment $x = a$, denoted $C|_{x=a}$, is defined to be the constant $\mathbf{1}$, if the assignment satisfies the clause, or else the clause obtained by deleting from C any literals involving the variable x. For any sequence of assignments S of the form $(x_1 = a_1, x_2 = a_2, \ldots, x_r = a_r)$ we write $C|_S$ to denote the result of computing the restriction of C by each assignment in turn. If $C|_S$ is empty, then we say that the assignments in S *falsify* the clause C. For a set of clauses Δ, we write $\Delta|_S$ to denote the set $\{C|_S \mid C \in \Delta\} \setminus \{\mathbf{1}\}$.

Most current SAT-solvers operate in the following way [5,25]. They maintain a database of clauses Δ and a current state S, which is a partial assignment of truth values to the Boolean variables in the clauses of Δ. A high-level description of the algorithms used to update the clause database and the state, derived from the description given in [5], is shown in Algorithm 1 (a similar framework, using slightly different terminology, is given in [25]).

Now consider a run of the algorithm shown in Algorithm 1, started with the initial database Δ, and the empty state S_0, until it either halts or discovers a *conflict* (i.e., $\emptyset \in \Delta|_S$). Such a run is called a *round started with* Δ, and we represent it by the sequence of states $S_0, \ldots, S_m$, that the algorithm maintains. Note that each state S_i extends the state S_{i-1} by a single assignment to a Boolean variable, which may be either a *decision assignment* or an *implied assignment.*

An initial segment $S_0, S_1, \ldots, S_r$ of a round started with Δ is called an *inconclusive partial round* if $\Delta|_{S_r}$ is non-empty, does not contain the empty clause, and does not contain any unit clauses. Note that for any clause $C \in \Delta$, if $S_0, S_1, \ldots, S_r$ is an inconclusive partial round started with Δ, and S_r falsifies all the literals of C except one, then it must satisfy the remaining literal, and hence satisfy C. This property of clauses is captured by the following definition.

Algorithm 1. Framework for typical clause-learning SAT-solver

```
Input: $\Delta$ : set of clauses;
       $S$ : partial assignment of truth values to variables.

while $\Delta|_S \neq \emptyset$ do
  if $\emptyset \in \Delta|_S$ then                                        CONFLICT
    if $S$ contains no decision assignments then
      print "UNSATISFIABLE" and halt
    else
      apply the learning scheme to add a new clause to $\Delta$
      if restart policy says restart then
        set $S = \emptyset$
      else
        select most recent conflict-causing unreversed decision assignment in $S$
        reverse this decision, and remove all later assignments from $S$
      end if
    end if
  else if $\{l\} \in \Delta|_S$ for some literal $l$ then                 UNIT PROPAGATION
    add to $S$ the implied assignment $x = a$ which satisfies $l$
  else                                                          DECISION
    apply the branching strategy to choose a decision assignment $x = a$
    add this decision assignment to $S$
  end if
end while
print "SATISFIABLE" and output $S$
```

Definition 4.1. *[5] Let Δ be a set of clauses, C a non-empty clause, and l a literal of C. We say that Δ* absorbs *C at l if every inconclusive partial round started with Δ that falsifies $C \setminus \{l\}$ satisfies C.*

If Δ absorbs C at each literal l in C, then we simply say that Δ **absorbs** *C.*

Note that a clause that is *not* absorbed by a set of clauses Δ is referred to in [25] as *1-empowering* with respect to Δ.

Lemma 4.2. *[5] Let Δ and Δ' be sets of clauses, and let C and C' be non-empty clauses.*

1. *If C belongs to Δ, then Δ absorbs C;*
2. *If $C \subseteq C'$ and Δ absorbs C, then Δ absorbs C';*
3. *If $\Delta \subseteq \Delta'$ and Δ absorbs C, then Δ' absorbs C;*
4. *If $\Delta \subseteq \Delta''$ and Δ absorbs C and Δ entails Δ'', then Δ'' absorbs C.*

To allow further analysis, we need to make some assumptions about the *learning scheme*, the *restart policy* and the *branching strategy* used by our SAT-solver.

The learning scheme is a rule that creates and adds a new clause to the database whenever there is a conflict. Such a clause is called a *conflict clause*, and each of its literals is falsified by some assignment in the current state. If a literal is falsified by the *i*-th decision assignment, or some later implied assignment before

$(i+1)$-th decision assignment, it is said to be falsified at level i. If a conflict clause contains exactly one literal that is falsified at the maximum possible level, it is called an *asserting clause* [28,25].

Assumption 1. *The learning scheme chooses an* asserting clause*.*

Most learning schemes in current use satisfy this assumption [28,25], including the learning schemes called "1UIP" and "DECISION" described in [28].

We make no particular assumption about the restart policy. However, our main result is phrased in terms of a bound on the expected number of restarts. If the algorithm restarts after r conflicts, our bound on the expected number of restarts can simply be multiplied by r to get a bound on the expected number of conflicts. This means that the implications will be strongest if the algorithm restarts *immediately after each conflict*. In that case, $r = 1$ and our bound will also bound the expected number of conflicts. Existing SAT-solvers typically do not employ such an aggressive restart policy, but we note the remark in [25] that "there has been a clear trend towards more and more frequent restarts for modern SAT solvers".

The branching strategy determines which decision assignment is chosen after an inconclusive partial round. In most current SAT solvers the strategy is based on some heuristic measure of *variable activity*, which is related to the occurrence of a variable in a conflict clause [23]. However, to simplify the probabilistic analysis, we will make the following assumption.

Assumption 2. *The branching strategy chooses a variable uniformly at random amongst the unassigned variables, and assigns it the value TRUE.*

As noted in [5], the same analysis we give below can also be applied to any other branching strategy that randomly chooses between making a heuristic-based decision or a randomly-based decision, provided that the second case has non-negligible probability p. In that case, the bounds we obtain on the expected number of restarts can simply be multiplied by p^{-k}.

An algorithm that behaves according to the description in Algorithm 1, and satisfies the assumptions above, will be called a *standard randomised* SAT-solver.

Theorem 4.3. *If a set of non-empty clauses Δ over n Boolean variables has a positive-hyper-resolution refutation of width k and length m, where all derived clauses contain only negative literals, then the expected number of restarts required by a standard randomised SAT-solver to discover that Δ is unsatisfiable is less than $mnk^2\binom{n}{k}$.*

Proof. Let $C_1, C_2, \ldots, C_m$ be a positive-hyper-resolution refutation of width k from Δ, where each C_i contains only negative literals, and C_m is the first occurrence of the empty clause. Since each clause in Δ is non-empty, C_m must be derived by positive-hyper-resolution from some collection of negative literals $\neg x_1, \neg x_2, \ldots \neg x_d$ and a purely positive clause $x_1 \vee x_2 \vee \cdots x_d$.

Now consider a standard SAT-solver started with database Δ. Once all of the unit clauses $\neg x_i$ are absorbed by the current database, then, by Definition 4.1, any further inconclusive partial round of the algorithm must assign all variables x_i false, and hence falsify the clause $x_1 \vee x_2 \vee \cdots x_d$. Since this happens even when no decision assignments are made, the SAT-solver will report unsatisfiability.

It only remains to bound the expected number of restarts required until each clause C_i is absorbed, for $1 \leq i < m$. Let each C_i be the positive-hyper-resolvent of clauses $C_{i1}, C_{i2}, \ldots, C_{id}$, each of the form $C'_{ij} \vee \neg x_j$, together with a purely positive clause $C_{i0} = x_1 \vee x_2 \vee \cdots \vee x_d$ from Δ. Assume also that each clause C_{ij} is absorbed by Δ.

If Δ absorbs C_i, then no further learning or restarts are needed, so assume now that Δ does not absorb C_i. By Definition 4.1, this means that there exists some literal l and some inconclusive partial round R started with Δ, that falsifies $C_i \setminus \{l\}$ and does not satisfy C_i. Note that R must leave the literal l unassigned, because one assignment would satisfy C_i and the other would force all of the literals $\neg x_j$ used in the positive-hyper-resolution step to be satisfied, because each C_{ij} is absorbed by Δ, so C_{i0} would be falsified, contradicting the fact that R is inconclusive.

Hence, if the branching strategy chooses to falsify the literals $C_i \setminus \{l\}$ whenever it has a choice, it will construct an inconclusive partial round R' where l is unassigned (since all the decision assignments in R' are also assigned the same values in R, any implied assignments in R' must also be assigned the same values[2] in R, but we have shown that R leaves l unassigned). If the branching strategy then chooses to falsify the remaining literal l of C_i, then the algorithm would construct a complete round R'' where C_{i0} is falsified, and all decision assignments falsify literals in C_i. Hence, by Assumption 1, the algorithm would then learn some asserting clause C' and add it to Δ to obtain a new set Δ'.

Since C' is an asserting clause, it contains exactly one literal, l', that is falsified at the highest level in R''. Hence, any inconclusive partial round R started with Δ' that falsifies $C_i \setminus \{l\}$ will falsify all but one literal of C', and hence force the remaining literal l' to be satisfied, by unit propagation. If this new implied assignment for l' propagates to force l to be true, then R satisfies C_i, and hence Δ' absorbs C_i at l. If not, then the branching strategy can once again choose to falsify the remaining literal l of C_i, which will cause a new asserting clause to be learnt and added to Δ. Since each new asserting clause forces a new literal to be satisfied after falsifying $C_i \setminus \{l\}$ this process can be repeated fewer than n times before it is certain that Δ' absorbs C_i at l.

Now consider any sequence of k random branching choices. If the first $k-1$ of these each falsify a literal of $C_i \setminus \{l\}$, and the final choice falsifies l, then we have shown that the associated round will reach a conflict, and add an asserting clause to Δ. With a random branching strategy, as described in Assumption 2, the probability that this happens is at least the probability that the first $k-1$ random choices consist of a fixed set of variables (in some order), and the final choice is the variable associated with l. The number of random choices that fall in a fixed set follows the hypergeometric distribution, so the overall probability of this is $\frac{1}{\binom{n}{k-1}} \frac{1}{(n-k+1)} = 1/(k\binom{n}{k})$.

[2] See Lemma 3 of [5] for a more formal statement and proof.

To obtain an upper bound on the expected number of restarts, consider the worst case where we require n asserting clauses to be added to absorb each clause C_i at each of its k literals l. Since we require only an upper bound, we will treat each round as an independent trial with success probability $p = 1/(k\binom{n}{k})$, and consider the worst case where we have to achieve $(m-1)nk$ separate consecutive successes to ensure that C_i for $1 \leq i < m$ is absorbed. In this case the total number of restarts will follow a negative binomial distribution, with expected value $(m-1)nk/p$. Hence in all cases the expected number of restarts is less than $mnk^2\binom{n}{k}$. □

A tighter bound on the number of restarts can be obtained if we focus on the Decision learning scheme [5,28], as the next result indicates.

Theorem 4.4. *If a set of non-empty clauses Δ over n Boolean variables has a positive-hyper-resolution refutation of width k and length m, where all derived clauses contain only negative literals, then the expected number of restarts required by a standard randomised SAT-solver using the* Decision *learning scheme to discover that Δ is unsatisfiable is less than $m\binom{n}{k}$.*

Proof. The proof is similar to the proof of Theorem 4.3, except that the Decision learning scheme has the additional feature that the literals in the chosen conflict clause falsify a subset of the current decision assignments. Hence in the situation we consider, where the decision assignments all falsify literals of some clause C_i (in any order), this learning scheme will learn a subset of C_i, and hence immediately absorb C_i, by Lemma 4.2(1,2). Hence the maximum number of learnt clauses required is reduced from $(m-1)nk$ to $(m-1)$, and the probability is increased from $1/(k\binom{n}{k})$ to $1/\binom{n}{k}$, giving the tighter bound. □

Note that a similar argument shows that the standard deviation of the number of restarts is less than the standard deviation of a negative binomial distribution with parameters m and $1/\binom{n}{k}$, which is less than $\sqrt{m}\binom{n}{k}$. Hence, by Chebyshev's inequality (one-tailed version), the probability that a standard randomised SAT-solver using the decision learning scheme will discover that Δ is unsatisfiable after $(m+\sqrt{m})\binom{n}{k}$ restarts is greater than 1/2.

5 *k*-Consistency and SAT-Solvers

By combining Theorem 3.1 and Theorem 4.4 we obtain the following result linking k-consistency and SAT-solvers.

Theorem 5.1. *If the k-consistency closure of a CSP instance P is empty, then the expected number of restarts required by a standard randomised SAT-solver using the* Decision *learning scheme to discover that the direct encoding of P is unsatisfiable is $O(n^{2k}d^{2k})$, where n is the number of variables in P and d is the maximum domain size.*

Proof. The length m of a positive-hyper-resolution refutation of width k is bounded by the number of possible no-goods of length at most k for P, which is $\sum_{i=1}^{k} d^i \binom{n}{i}$. Hence, by Theorem 3.1 and Theorem 4.4 we obtain a bound of $\left(\sum_{i=1}^{k} d^i \binom{n}{i}\right) \binom{nd}{k}$, which is $O(n^{2k}d^{2k})$. □

Hence a standard randomised SAT-solver with a suitable learning strategy will decide the satisfiability of any CSP instance with tree-width k with $O(n^{2k}d^{2k})$ expected restarts, even when it is set to restart immediately after each conflict. In particular, the satisfiability of any tree-structured CSP instance (i.e., with tree-width 1) will be decided by such a solver with at most $O(n^2d^2)$ expected conflicts, which is comparable with the growth rate of an optimal arc-consistency algorithm. Note that this result cannot be obtained directly from [5], because the direct encoding of an instance with tree-width k is a set of clauses whose tree-width may be as high as dk.

Moreover, a standard randomised SAT-solver will decide the satisfiability of any CSP instance, with any structure, within the same polynomial bounds, if the constraint relations satisfy certain algebraic properties that ensure bounded width [9]. Examples of such constraint types include the "0/1/all relations", defined in [14], and the "connected row-convex" relations, defined in [15], which can both be decided by 2-consistency.

6 Experimental Results

The bounds we obtain in this paper are very conservative, and are likely to be met very easily in practice.

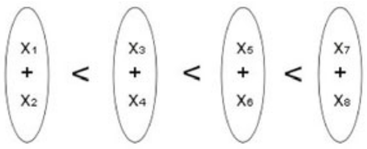
To investigate how an existing SAT-solver performs in practice, we measured the performance of the MiniSAT solver [2] version 2-070721 on a family of CSP instances that can be decided by a fixed level of consistency. We ran the experiments with preprocessing switched off, in order to get a solver that uses only unit propagation and conflict-directed learning with restarts.

We also modified the MiniSAT solver to follow the random branching strategy described above. Our modified solver does not delete any learnt clauses and uses an extreme restart policy that makes it restart whenever it encounters a conflict. We refer to this modified solver as simple-MiniSAT.

For all of the results, the times given are elapsed times on a Lenovo 3000 N200 laptop with an Intel Core 2 Duo processor running at 1.66GHz with 2GB of RAM. For our simple-MiniSAT solver, each generated instance was run three times and the mean times and mean number of restarts are shown.

Example 6.1. We consider a family of instances specified by two parameters, w and d. They have $((d-1)*w+2)*w$ variables arranged in groups of size w, each with domain $\{0,...,d-1\}$. We impose a constraint of arity $2w$ on each pair of successive groups, requiring that the sum of the values assigned to the first of these two groups should be strictly smaller than the sum of the values assigned to the second. This ensures that the instances generated are unsatisfiable. An instance with $w = 2$ and $d = 2$ is shown diagrammatically and defined using the specification language MiniZinc [1] in Figure 1.

The structure of these instances has a simple tree-decomposition as a path of nodes, with each node corresponding to a constraint scope. Hence the tree-width of these instances is $2w-1$, and they can be shown to be unsatisfiable by enforcing $2w-1$ consistency. However, these instances cannot be solved efficiently using standard propagation algorithms which only prune individual domain values.



(a) Graphical representation.

array[1..4] of var 0..1 : $X1$;
array[1..4] of var 0..1 : $X2$;
constraint
forall(i in 1..3)(
$X1[i] + X2[i]$
$<$
$X1[i+1] + X2[i+1]$);
solve satisfy;

(b) Specification in MiniZinc.

Fig. 1. An example of a CSP instance with $w = 2$, $d = 2$ and tree-width $= 3$

Table 1. Performance of CP-solvers and SAT-solvers on instances from Example 6.1

group size (w)	domain size (d)	CSP variables (n)	Minion (sec)	G12 (sec)	MiniSAT (sec)	simple-MiniSAT (sec)	simple-MiniSAT restarts
2	2	8	0.010	0.238	0.004	0.004	17
2	3	12	0.012	0.246	0.007	0.008	175
2	4	16	0.026	0.273	0.021	0.038	866
2	5	20	0.043	0.525	0.052	0.146	2 877
2	6	24	1.040	6.153	0.157	0.626	7 582
2	7	28	47.554	205.425	0.433	2.447	17 689
2	8	32	> 20 min	> 20 min	1.273	10.169	35 498
2	9	36	> 20 min	> 20 min	3.301	44.260	65 598
2	10	40	> 20 min	> 20 min	8.506	135.215	108 053
3	2	15	0.012	0.240	0.005	0.008	176
3	3	24	0.370	1.120	0.103	0.377	4 839
3	4	33	> 20 min	> 20 min	1.942	22.357	43 033
3	5	42	> 20 min	> 20 min	29.745	945.202	209 094

Table 1 shows the runtimes of simple-MiniSAT and the original MiniSAT solver on this family of instances, along with times for two state-of-the-art CP solvers: Minion [17] and G12 [1]. Note that MiniSAT is remarkably effective in solving these instances, compared to the CP solvers, even though they are encoded into a large number of clauses with a much larger tree-width than the original instance. Although our modified SAT solver takes a little longer, it still performs better on these instances than the CP solvers and the number of restarts (and hence the number of conflicts) is much lower than the polynomial upper bound obtained in Theorem 5.1 (see Figure 2).

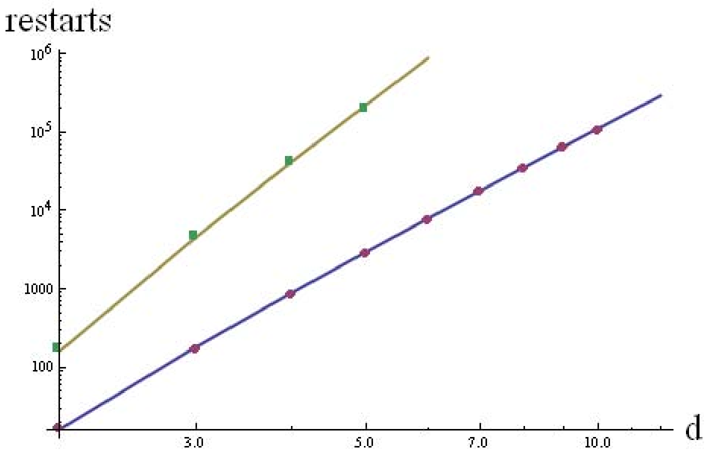


Fig. 2. Log-log plot of the number of restarts/conflicts used by simple-MiniSAT on the instances from Example 6.1. Circles show values for $w = 2$; squares show values for $w = 3$; solid lines show the functions $d^2\binom{n/2}{3}$ (lower line) and $d^4\binom{n/3}{3}$ (upper line). Note that these experimentally determined growth functions are much lower than the worst-case bound calculated in Theorem 5.1.

7 Conclusion

We have shown that the notion of k-consistency can be precisely captured by a single inference rule on the direct encoding of a CSP instance, restricted to deriving only clauses with at most k literals. We used this to show that a clause-learning SAT-solver with a purely random branching strategy will simulate the effect of enforcing k-consistency in expected polynomial time, for all fixed k. This is sufficient to ensure that such solvers are able to solve certain problem families much more efficiently than conventional CP solvers relying on GAC-propagation.

In principle clause-learning SAT-solvers can also do much more. It is known that, with an appropriate branching strategy and restart policy, they are able to p-simulate general resolution [10,25], and general resolution proofs can be exponentially shorter than the negative resolution proofs we have considered here [18]. In practice, it seems that current clause-learning SAT-solvers with highly-tuned learning schemes, branching strategies and restart policies are often able to exploit structure in the encoding of a CSP instance even more effectively than local consistency techniques. Hence considerable work remains to be done in understanding the relevant features of instances which they are able to exploit, in order to predict their effectiveness in solving different kinds of CSP instances.

References

1. G12/MiniZinc constraint solver. Software, `http://www.g12.cs.mu.oz.au/minizinc/download.html`
2. MiniSat solver. Software, `http://minisat.se/MiniSat.html`
3. 2nd internat. CSP solver competition, `http://www.cril.univ-artois.fr/CPAI06/`
4. 3rd international CSP solver competition, `http://cpai.ucc.ie/08/`
5. Atserias, A., Fichte, J.K., Thurley, M.: Clause-learning algorithms with many restarts and bounded-width resolution. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 114–127. Springer, Heidelberg (2009)
6. Atserias, A., Bulatov, A.A., Dalmau, V.: On the power of k-consistency. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 279–290. Springer, Heidelberg (2007)
7. Atserias, A., Dalmau, V.: A combinatorial characterization of resolution width. Journal of Computer and Systems Science 74(3), 323–334 (2008)
8. Bacchus, F.: GAC Via Unit Propagation. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 133–147. Springer, Heidelberg (2007)
9. Barto, L., Kozik, M.: Constraint satisfaction problems of bounded width. In: Proceedings of FOCS 2009, pp. 595–603. IEEE Computer Society, Los Alamitos (2009)
10. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research 22, 319–351 (2004)
11. Bessiére, C.: Constraint propagation. In: Handbook of Constraint Programming, ch. 3, pp. 29–83. Elsevier, Amsterdam (2006)
12. Bordeaux, L., Hamadi, Y., Zhang, L.: Propositional satisfiability and constraint programming: A comparative survey. ACM Computing Surveys 38(4) (2006)
13. Cooper, M.C.: An optimal k-consistency algorithm. Artificial Intelligence 41, 89–95 (1989)
14. Cooper, M.C., Cohen, D.A., Jeavons, P.G.: Characterising tractable constraints. Artificial Intelligence 65, 347–361 (1994)
15. Deville, Y., Barette, O., van Hentenryck, P.: Constraint satisfaction over connected row convex constraints. In: Proceeedings of IJCAI 1997, pp. 405–411 (1997)
16. Freuder, E.C.: Synthesizing constraint expressions. ACM Comm. 21, 958–966 (1978)
17. Gent, I., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: Proceeedings of ECAI 2006, pp. 98–102. IOS Press, Amsterdam (2006)
18. Hwang, J., Mitchell, D.: 2-way vs. d-way branching for CSP. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 343–357. Springer, Heidelberg (2005)
19. Rish, I., Dechter, R.: Resolution versus search: Two strategies for SAT. Journal of Automated Reasoning 24(1/2), 225–275 (2000)
20. Kolaitis, P.G., Vardi, M.Y.: A game-theoretic approach to constraint satisfaction. In: Proceedings of AAAI 2000, pp. 175–181 (2000)
21. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence 8, 99–118 (1977)
22. Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences 7, 95–132 (1974)
23. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: International Design Automation Conference, DAC, pp. 530–535 (2001)

24. Petke, J., Jeavons, P.G.: Tractable benchmarks for constraint programming. Technical Report RR-09-07, Computing Laboratory, University of Oxford (2009)
25. Pipatsrisawat, K., Darwiche, A.: On the power of clause-learning SAT solvers with restarts. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 654–668. Springer, Heidelberg (2009)
26. Walsh, T.: SAT v CSP. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 441–456. Springer, Heidelberg (2000)
27. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints 14(2), 254–272 (2009)
28. Zhang, L., Madigan, C.F., Moskewicz, M.W., Andmalik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD 2001), pp. 279–285 (2001)
29. Zhang, L., Malik, S.: The quest for efficient Boolean satisfiability solvers. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 641–653. Springer, Heidelberg (2002)