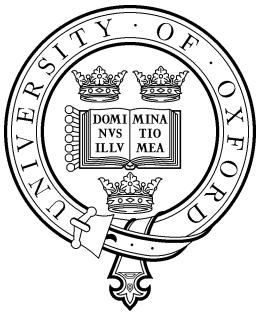
Computing Science

TRACTABLE BENCHMARKS FOR CONSTRAINT PROGRAMMING

Justyna Petke and Peter Jeavons

CS-RR-09-07



Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Tractable Benchmarks for Constraint Programming*

Justyna Petke and Peter Jeavons

Abstract

The general constraint satisfaction problem for variables with finite domains is known to be NP-complete, but many different conditions have been identified which are sufficient to ensure that classes of instances satisfying those conditions are tractable, that is, solvable in polynomial time. Results about tractability have generally been presented in theoretical terms, with little discussion of how these results impact on practical constraint-solving techniques. In this paper we investigate the performance of several standard constraint solvers on benchmark instances that are designed to satisfy various different conditions that ensure tractability. We show that in certain cases some existing solvers are able to automatically take advantage of the problem features which ensure tractability, and hence solve the corresponding instances very efficiently. However, we also show that in many cases the existing pre-processing techniques and solvers are unable to solve efficiently the families of instances of tractable problems that we generate. We therefore suggest that such families of instances may provide useful benchmarks for improving pre-processing and solving techniques.

*Further copies of this Research Report may be obtained from the Librarian, Oxford University Computing Laboratory, Computing Science, Wolfson Building, Parks Road, Oxford OX1 3QD, England (Telephone: +44-1865-273837, Email: library@comlab.ox.ac.uk).

1 Introduction

Software tools for solving finite-domain constraint problems are now freely available from several groups around the world. Examples include the Gecode system developed in Germany and Sweden [2], the G12 solver developed in Australia [1], and the Minion constraint solver developed in the UK [20].

One way to drive performance improvements in constraint solvers, which has proved very successful in the SAT-solving community, is to develop challenging benchmark instances. This approach can also help to drive improvements in the robustness and flexibility of constraint-solving software. For example, several families of benchmark MiniZinc instances have been distributed with G12 [1] since version 0.7, and these have been used to compare the performance of various solvers, and to develop and test an alternative solver, FznTini [26], based on translation to Boolean Satisfiability [27].

How can suitable benchmark instances be obtained? One obvious source of benchmark instances is from practical applications such as scheduling and manufacturing process organisation; the G12 MiniZinc suite includes several examples of this kind, such as "nurse scheduling" problems and "car sequencing" problems. Another common source of benchmark instances is combinatorial problems such as puzzles and games; the G12 MiniZinc suite also includes several examples of this kind, such as "Golomb ruler" problems and "kakuro" puzzles.

In this paper we suggest another important source of useful benchmark instances which has not yet been systematically explored: the theoretical study of constraint satisfaction. From the very beginning of the study of constraint programming there has been a strand of research which has focused on identifying features of constraint problems which make them tractable to solve [10, 13, 15, 17] and this research has gathered pace recently with the discovery of some deep connections between constraint problems and algebra [4, 5, 7, 6], logic [11, 12, 16], and graph and hypergraph theory [9, 24].

This research has focused on two main ways in which imposing restrictions on a constraint problem can ensure that it can be tractably solved. The first of these is to restrict the forms of constraint which are allowed; these are sometimes known as *constraint language* restrictions. For example, it was shown in [10] that certain forms of arithmetic constraint introduced in the CHIP programming language [34] had a property which ensured that they could be efficiently solved no matter how they were combined. It was also shown in [10] that many other forms of constraint also had the same property and could also be combined arbitrarily whilst allowing an efficient solution algorithm. Since then many other classes of so-called tractable constraint languages have been identified, and a sophisticated algebraic theory has been developed which aims to distinguish tractable forms of constraints from those which can lead to intractable problems [5, 6].

The second standard approach to identifying restrictions on constraint problems which ensure tractability has been to consider restrictions on the way in which the constraints overlap; these are sometimes referred to as *structural restrictions*. For example, it was shown in [17] that binary constraint problems where the underlying graph of the constraints has bounded width can be efficiently solved by choosing an appropriate variable ordering. For non-binary constraint problems, where each constraint may involve more

than two variables, the underlying structure is a hypergraph, and certain structural conditions on this hypergraph can again be sufficient to ensure tractability, regardless of the forms of constraint imposed. For example, if this hypergraph is acyclic [14] or has a bounded degree of cyclicity [25] or a bounded hypertree width [21] then the resulting constraint problem has been shown to be tractable. A complete characterisation of the class of hypergraphs which lead to tractable constraint problems was obtained in [24], for problems where the maximum arity of any constraint is bounded. However, some of these structural tractability results rely on the assumption that constraints are represented extensionally, by an explicit table of allowed tuples, and this assumption is often not satisfied in practical constraint problems, where constraints are often represented by special-purpose algorithms known as propagators. A theory of structural tractability for constraints represented by propagators was developed in [22], and results in rather smaller tractable classes.

In this paper we begin the process of translating from theoretical results in the literature to concrete families of instances of constraint problems. We obtain several families which are known to be efficiently solvable by simple algorithms, but which cause great difficulties for some existing constraint solvers. We argue that such families of instances provide a useful addition to benchmark suites derived from other sources, and can provide a fruitful challenge to the developers of general-purpose solvers.

2 Definitions

In the theoretical literature the (finite-domain) **constraint satisfaction problem** (CSP) is typically defined as follows:

Definition 2.1. *A instance of the constraint satisfaction problem is specified by a triple* (V, D, C)*, where*

- V *is a finite set of variables*
- D *is a finite set of values (this set is called the domain)*
- 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* k_i *variables, called the constraint scope;*
  - R_i *is a relation over* D *of arity* k_i*, called the constraint relation.*

In practical constraint problems it is common to assume that each variable has its own specified domain, but this is easily accommodated in the simplified theoretical framework of Definition 2.1 by setting D to be the union of all the individual domains, and then imposing a unary constraint on each variable to restrict the values to the appropriate subset of D.

More significantly, Definition 2.1 says nothing about how the individual constraints are represented in the specification of a particular concrete instance. For example, constraint relations may be specified by explicitly listing all of the allowed tuples of values, or perhaps

all of the disallowed tuples of values, or simply by naming a standard relation such as "all-different". Although it is generally not an issue that is considered in the theoretical literature, it is clearly an important issue in practice to decide how problem instances will be encoded for input to a constraint solver, and the lack of a common agreed standard in this area is one of the difficulties of developing widely-accepted benchmarks.

Two proposed standard higher-level languages for specifying constraint problems in practice are Zinc [30] and Essence [19]. However, both of these languages are considered too abstract and too general to be used directly as the input language for current constraint solvers, so they both have more restricted subsets which are more suitable for solver input: these are called MiniZinc and Essence$'$. There exists a software translation tool, called Tailor [32], which converts from Essence$'$ specifications to the input language for the Minion solver (or Gecode). Another software translation tool distributed with the G12/MiniZinc software [1], converts from MiniZinc to a more restricted language known as FlatZinc, that serves as the input language for the G12 solver; FlatZinc input is also accepted by Gecode. The FznTini solver, developed by Huang, transforms a FlatZinc file into DIMACS CNF format and then uses a Boolean Satisfiability problem (SAT) solver, called TiniSAT, to solve the resulting SAT problem [27].

Definition 2.2. *A* solution *to a CSP instance $P = (V, D, C)$ is an assignment of values from D to each of the variables in V, which satisfies all of the constraints in C simultaneously.*

Formally, a solution is a map $h : V \to D$ such that $h(S_i) \in R_i$, for all i, where the expression $h(S_i)$ denotes the result of applying h to the tuple S_i, coordinate-wise (in other words, if $S_i = \langle\, v_1, \ldots, v_k \,\rangle$, then $h(S_i) = \langle\, h(v_1), \ldots, h(v_k) \,\rangle$).

Note that in the theoretical literature the CSP is generally formalised as a *decision problem*: the question associated with each instance is simply to decide whether a solution exists. In practice, of course, it is often more natural to consider the corresponding *search problem*, which asks us to find a solution if one exists. However, we note that for any class of CSP instances where we are allowed to add unary *constant constraints*, fixing the assignments for some individual variables, we can find a solution with at most $|V| \times |D|$ iterations of the decision algorithm by adding a new unary constant constraint and calling the decision algorithm again on each iteration [8]. In the experimental results recorded here we ask the solvers to solve the search problem.

The time complexity of this search problem is at most exponential in the size of the input, since the size of the total search space for possible solutions is $|D|^{|V|}$. Moreover, if we assume that each constraint is represented in such a way that checking whether a given assignment satisfies a given constraint can be completed in polynomial time, then CSP clearly belongs to the problem class NP, since an assignment can be verified in polynomial-time in the size of the input. The general CSP is easily shown to be NP-complete, since it includes standard NP-complete problems such as Satisfiability and Graph Colouring [5]. However, for certain restricted classes of instances it is possible to find a solution, or verify that no solution exists, in polynomial time. Such restricted classes will be called **tractable**.

Definition 2.3. *A class of CSP instances will be called* tractable *if there exists an algorithm which finds a solution to all instances in that class, or reports that there are no solutions, whose time complexity is polynomial in the size of the instance specification.*

Many examples of tractable classes have been identified in the literature: see [31] for an early survey, and [6, 23] for more recent surveys. In this paper we will focus on some of the simplest and most widely-known examples of tractable classes. In particular, we will construct families of instances that are tractable for each of the following reasons:

- All constraints allow some constant value d to be assigned to every variable (Section 3).
- All constraints are "max-closed constraints" as defined in [29] (Section 4).
- All constraints are "0/1/all constraints" as defined in [10] (Section 5).
- The constraint hypergraph has bounded width (Section 6).

3 Constant-closed constraints

Classes of CSP instances where each constraint allows some constant value d to be assigned to every variable are clearly tractable according to Definition 2.3 because they can be solved by the trivial algorithm that assigns the value d to every variable in the instance. Such classes were included (for completeness) in several early lists of tractable classes, including Schaefer's Dichotomy Theorem for the Boolean satisfiability problem [33] and the first survey of tractable cases identified by the algebraic approach to constraint complexity [28].

Surprisingly, instances with this property do occur in practice: the majority of the satisfiable binary decision diagram instances used in the third CSP solver competition [3] have constant solutions[1].

To investigate whether the presence of a constant solution affects the performance of standard constraint solvers we generated CSP instances with just one solution - the constant one. Our instance generator took as input the number of variables, n, and the number of possible values for each variable, m and created instances with n variables each with domain $0, \ldots, (m-1)$. Each of the generated constraints allowed the value $\lfloor m/2 \rfloor$ to be assigned to all variables[2].

As a simple way to ensure that each instance had only this solution, we generated a line of binary constraints, with one constraint on each successive pair of variables. On the first $n-1$ variables these constraints were obtained by choosing a random list of $m/2$ allowed values for each variable (with repetitions) from the domain $0..(m/2)$ and allowing just those pairs of values formed by the corresponding entries in two successive

[1] This observation was made by Marc van Dongen in a personal communication.

[2] This middle value was chosen as the constant value so that default value orderings which considered the values for each variable in ascending or descending order did not simply happen to consider the constant value first.

MiniZinc

```
array[1..4] of var 0..4: X;
constraint
(
((X[1] = 1)/\(X[2] = 0))
\/((X[1] = 0)/\(X[2] = 0))
\/((X[1] = 2)/\(X[2] = 2))
) /\ (
((X[2] = 0)/\(X[3] = 0))
\/((X[2] = 0)/\(X[3] = 1))
\/((X[2] = 2)/\(X[3] = 2))
) /\ (
((X[3] = 3)/\(X[4] = 1))
\/((X[3] = 4)/\(X[4] = 1))
\/((X[3] = 2)/\(X[4] = 2))
) ;
solve satisfy;
```

Essence′

```
language ESSENCE' 1.b.a
letting D be domain int(1..4)
find X : matrix indexed by [D]
                      of int(0..4)
such that
(
((X[1] = 1)/\(X[2] = 0))
\/((X[1] = 0)/\(X[2] = 0))
\/((X[1] = 2)/\(X[2] = 2))
) /\ (
((X[2] = 0)/\(X[3] = 0))
\/((X[2] = 0)/\(X[3] = 1))
\/((X[2] = 2)/\(X[3] = 2))
) /\ (
((X[3] = 3)/\(X[4] = 1))
\/((X[3] = 4)/\(X[4] = 1))
\/((X[3] = 2)/\(X[4] = 2))
)
```

Table 1: Typical constant-closed CSP instance specifications generated in MiniZinc and Essence′ for $n = 4$ and $m = 5$. All constraints allow the value 2 for all variables.

Domain size (m)	Number of variables (n)	Gecode Time (secs.)	G12 Time (secs.)	FznTini Time (secs.)	Minion Time (secs.)
4	100	$>$ 15 min	0.53	0.04	$>$ 15 min
10	30	41.42	0.47	0.03	5.49
10	100	$>$ 15 min	0.75	0.08	$>$ 15 min
20	20	16.40	0.49	0.04	2.19
100	10	7.96	0.67	0.12	1.48
100	15	254.48	0.91	0.11	38.96
100	20	$>$ 15 min	1.06	0.21	$>$ 15 min
200	10	61.79	1.05	0.29	11.40
200	15	$>$ 15 min	1.80	0.45	344.91
200	20	$>$ 15 min	2.15	0.37	$>$ 15 min

Table 2: Average solution times for Gecode, G12, FznTini and Minion on constant-closed CSP instances of the form shown in Table 1 which have exactly one solution.

lists (together with the pair $(\lfloor m/2 \rfloor, \lfloor m/2 \rfloor)$). The final binary constraint, between the $(n-1)^{th}$ and n^{th} variables, restricted the $(n-1)^{th}$ variable to values in the other half of the domain, thus eliminating all possible solutions except the constant one with value

$\lfloor m/2 \rfloor$. These binary constraints were then expressed in a form of explicit representation, as a disjunction of conjunctions of possible assignments, as shown in Table 1.

We generated instances for various choices of the parameters n and m, and solved these using Gecode (version 1.3), G12 (version 0.8.1), FznTini, and Minion (version 0.8RC1) - see Table 2. As with all of the results presented in this paper, the times given are elapsed times on a Lenovo 3000 N200 laptop with an Intel Core 2 Duo processor running at 1.66GHz an 2GB of RAM. These timings *exclude* the time required to translate the input from MiniZinc to FlatZinc (for input to Gecode, G12 and FznTini) or from Essence′ to Minion input format. (In the special case of FznTini, times *include* the additional time required to translate from FlatZinc to DIMACS CNF format.) Average times over three runs with different generated instances are shown, but the variability was found in all cases to be quite small.

It is clear that by far the most efficient solvers for instances of this kind, when presented in this way, is FznTini, which appears to be able to identify the single constant solution extremely rapidly without any specific tuning. The standard constraint solvers are much less efficient[3] on these instances, which is somewhat surprising since, if the constraints are viewed as binary table constraints, all other values for all of the variables can be eliminated by enforcing arc-consistency, which all of these solvers do by default when propagating constraints. In fact, the translations to FlatZinc and Minion input format do not recognise the constraints as explicit binary constraints, but instead handle the disjunctions by introducing a large number of auxiliary variables, but this is common to all of the solvers tested (including FznTini).

This very simple first set of potential benchmark instances already reveals that there is considerable scope for improving the ability of current CSP solvers to recognise and exploit structure in the constraints, for example by better recognition and translation of table constraints, or by adapting the value ordering.

4 Max-closed constraints

One of the first non-trivial classes of tractable constraint types described in the literature is the class of max-closed constraints introduced in [29].

Definition 4.1 ([29])**.** *A constraint (R, S) with relation R of arity r over an ordered domain D is said to be* max-closed *if for all tuples $(d_1, ..., d_r)$,$(d'_1, ..., d'_r) \in R$ we have $(max(d_1, d'_1), ..., max(d_r, d'_r)) \in R$.*

In particular, one useful form of max-closed constraint is an inequality of the form $a_1X_1 + a_2X_2 + \cdots + a_{r-1}X_{r-1} \geq a_rX_r + c$, where the X_i are variables, c is a constant, and the a_is are non-negative constants [29]. Hence, we constructed a generator which produced random inequalities of this form. An extract from a typical instance specification produced by this generator is shown in Table 3.

[3]The reason that Gecode and Minion are so much less efficient than G12 on these instances appears to be due to the different default value orderings used.

```
array[1..10] of var 1..5: X;
constraint
97*X[6] >= 46*X[3] + 16 /\
81*X[5] +88*X[4] +60*X[2] +92*X[7] +28*X[10] >= 43*X[8] + 4 /\
16*X[3] +78*X[10] +61*X[7] +97*X[5] +50*X[8] +30*X[1] >= 19*X[6] + -51 ;
solve satisfy;
```

Table 3: A generated max-closed instance specification in MiniZinc with 3 inequalities.

Number of inequalities (k)	Satisfiable?	Number of variables (n)	Number of values (m)	Gecode Time (secs.)	G12 Time (secs.)	FznTini Time (secs.)	Minion Time (secs.)
10	yes	10	100	0.01	0.32	9.71	0.01
10	yes	10	200	0.01	0.26	3.49	0.01
10	yes	100	10	0.02	0.41	$>$ 15 min	0.05
100	yes	10	100	0.02	0.41	$>$ 15 min	0.06
100	no	10	100	0.02	0.41	$>$ 15 min	0.03
1000	yes	20	100	0.08	1.34	error	0.38
1000	no	20	100	0.03	0.59	error	0.13
1000	yes	30	200	0.08	1.59	error	0.52
1000	no	30	200	0.03	0.57	error	0.13
1000	yes	200	10	0.46	6.99	error	2.88
1000	no	200	10	0.03	0.59	error	0.14

Table 4: Average solution times for Gecode, G12, FznTini and Minion on max-closed CSP instances of the form shown in Table 3.

To generate *solvable* max-closed CSP instances, we selected a random assignment to all of the variables, and then generated random inequalities of the form above, keeping only those that were satisfied by this fixed assignment. This ensured that the system of inequalities had at least one solution. To generate *unsolvable* max-closed CSP instances, we generated the inequalities without imposing this condition; if the resulting set was solvable, another set was generated. Average times over 3 runs with different instances are shown in Table 4.

The results for these instances are exactly the reverse of those in Section 3 - see Table 2. Predictably, FznTini performs very poorly on these inequalities, which it has to translate into (large) sets of clauses. (For the larger sets of inequalities we considered it simply gave an 'out of memory' error.) Standard CSP solvers should do well on these instances, because the efficient algorithm for solving max-closed instances is based on achieving arc-consistency, and all standard constraint solvers do this by default. Our results confirm that the standard CSP solvers do indeed all perform well on these instances, although this time the G12 solver was noticeably less efficient than the other two.

5 0/1/all constraints

Our final example of a language-based restriction ensuring tractability involves the 0/1/all constraints introduced and shown to be tractable in [10].

Definition 5.1 ([10]). *Let x_1 and x_2 be variables. Let A be a subset of possible values for x_1 and B be a subset of possible values for x_2.*

- *A* complete *binary constraint is a constraint $R(x_1, x_2)$ of the form $A \times B$.*
- *A* permutation *constraint is a constraint $R(x_1, x_2)$ which is equal to $\{(v, \pi(v)) | v \in A\}$ for some bijection $\pi : A \to B$.*
- *A* two-fan *constraint is a constraint $R(x_1, x_2)$ where there exists $v \in A$ and $w \in B$ with $R(x_1, x_2) = (v \times B) \cup (A \times w)$.*

A 0/1/all constraint *is either a complete constraint, a permutation constraint, or a two-fan constraint.*

What is particularly interesting about this form of constraint, for our purposes, is that the efficient algorithm for 0/1/all constraints is based on achieving *path-consistency* [10], which is not implemented in standard constraint solvers.

To investigate whether instances with 0/1/all constraints are solved efficiently in practice by standard constraint solvers, even without explicitly using path-consistency, we wrote a generator for satisfiable CSP instances with 0/1/all constraints of various kinds on n variables. To ensure satisfiability we first generate a random assignment and then add only those 0/1/all constraints that satisfy the initial assignment. An extract from a typical instance specification produced by our generator is shown in Table 5.

We generated instances for various choices of the parameters n and m, and solved these using Gecode, G12, FznTini, and Minion. Average timings over 3 instances are shown in Table 6, but the variability is again very small. All the solvers performed very well, especially Gecode.

We also generated unsatisfiable instances with 0/1/all constraints on just a small number of variables, leaving all other variables unconstrained, see Table 7. FznTini quickly reported 'no solutions', but this solver does not distinguish between unsatisfiable instances, and instances that require too much memory [27], so this answer cannot be relied on. The G12 solver reported 'no solutions' in 0.2 seconds, but Minion and Gecode could not solve this problem within 15 min. When the domain size was decreased to two, none of the solvers could verify that this simple unsatisfiable instance had no solutions within 15 minutes. The problem seems to be the fixed default variable ordering: the solvers try every possible combination of values for the first 49 'unconstrained' variables before they report that the problem does not have a solution. None of the standard solvers focus the search on the few variables that are restricted; having no constraint between two variables is treated in the same way as having a complete constraint. Once the unsatisfiable instances were embedded in satisfiable instances, such as the one shown in Table 5, the performance of all the solvers was as good as before.

```
var {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}: X0;
var {0, 1, 3, 4, 7, 8, 9}: X1;
var {0, 2, 3, 5, 6, 7, 8, 9}: X2;
var {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}: X3;
var {0, 2, 3, 4, 6, 8}: X4;
constraint
((X0 + 7 >= 10) -> (X1 == X0 + 7 - 10)) /\
((X0 + 7 < 10) -> (X1 == X0 + 7)) /\
((X0 == 2) \/ (X2 == 0)) /\
((X1 + 7 >= 10) -> (X3 == X1 + 7 - 10)) /\
((X1 + 7 < 10) -> (X3 == X1 + 7)) /\
((X1 + 9 >= 10) -> (X4 == X1 + 9 - 10)) /\
((X1 + 9 < 10) -> (X4 == X1 + 9)) /\
((X3 == 6) \/ (X2 == 8)) /\
((X3 == 6) \/ (X4 == 6)) ;
solve satisfy;
```

Table 5: A typical CSP instance with 0/1/all constraints specified in MiniZinc. Note that *complete* constraints are imposed by restricting the domains to some subset, *permutation* constraints are imposed by constraints of the form $x = y + k \mod d$, and *two-fan* constraints are imposed by constraints of the form $x = v \vee y = w$.

Number of variables (n)	Number of values (m)	Gecode Time (secs.)	G12 Time (secs.)	FznTini Time (secs.)	Minion Time (secs.)
10	100	0.02	0.41	0.06	0.06
50	100	0.09	1.04	0.61	0.58
100	10	0.33	3.49	1.68	1.51
100	50	0.32	3.53	2.32	2.03
100	100	0.37	3.58	2.42	2.19

Table 6: Average solution times for Gecode, G12, FznTini and Minion on satisfiable 0/1/all CSP instances of the form shown in Table 5.

```
array[0..99] of var 0..3: X;
constraint
((X[50] == 2) \/ (X[51] == 1)) /\
((X[50] == 1) \/ (X[51] == 2)) /\
((X[50] == 2) \/ (X[51] == 2)) /\
((X[50] == 1) \/ (X[51] == 1)) ;
solve satisfy;
```

Table 7: An unsatisfiable CSP instance with 0/1/all constraints on 100 variables with domain size 4 specified in MiniZinc.

These results suggest that standard CSP solvers can handle random collections of 0/1/all constraints very effectively, even without specialised algorithms. However, they appear to be poor at focusing search on more highly constrained regions, which is thought to be one of the strengths of the current generation of SAT-solvers. This suggests an obvious target for improvement in adapting the variable ordering to the specific features of the input instance.

6 Bounded-width structures

For our final example, we consider classes of CSP instances which are tractable because of the way that the constraint scopes are chosen. In other words, we consider structural restrictions.

Definition 6.1. *A hypergraph is a pair $H = (V, E)$, where V is an arbitrary set, called the vertices of H, and E is a set of subsets of V, called the hyperedges of H.*

For any CSP instance, the scopes of all the constraints can be viewed as the hyperedges of an associated hypergraph whose vertices are the variables. This hypergraph is called the structure of the CSP instance. If we impose certain conditions on the kinds of structure we allow an instance to have, then this can be sufficient to ensure tractability for all possible CSP instances with structures satisfying those conditions, regardless of the type of constraints [23]. In particular, one very simple condition which is sufficient to ensure tractability is to require the structure to have a *tree decomposition* [23], with some fixed bound on the maximum number of vertices in any node of the tree. Such structures are said to have *bounded width*.

However, the efficient algorithm for CSP instances with bounded width structures is based on choosing an appropriate variable ordering, and imposing a level of consistency proportional to the width [12, 14, 18]. None of the standard CSP solvers incorporate such algorithms, so it is not at all evident whether they can solve bounded width instances efficiently.

To investigate this question we wrote a generator for a family of specific CSP instances with a very simple bounded-width structure.

The instances we generate are specified by two parameters, w and m. They have $(mw + 1) * w$ variables arranged in groups of size w, each with domain $\{0, ..., m\}$. 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 larger than the sum of the values assigned to the second. This ensures that a solution exists and satisfies the following conditions: the difference between the sum of values assigned to each successive group is 1, and the sum of the values assigned to the last group is zero. An extract from a typical instance specification produced by our generator is shown in Table 8.

When $w = 1$, the generated instances have a single line of binary constraints, so they have a tree structure, and can be efficiently solved using arc-consistency. For this special case, all of the solvers are able to solve the instances very quickly (see the first row of Table 9).

MiniZinc

```
array[1..7] of var 0..2: X1;
array[1..7] of var 0..2: X2;
array[1..7] of var 0..2: X3;
constraint
forall(i in 1..6)(
X1[i]+X2[i]+X3[i]
 >
X1[i+1]+X2[i+1]+X3[i+1]);
solve satisfy;
```

Essence′

```
language ESSENCE' 1.b.a
letting D be domain int(1..7)
letting E be domain int(1..6)
find X1 : matrix indexed by [D] of int(0..2)
find X2 : matrix indexed by [D] of int(0..2)
find X3 : matrix indexed by [D] of int(0..2)
such that
forall i : E. (
X1[i]+X2[i]+X3[i]
 >
X1[i+1]+X2[i+1]+X3[i+1])
```

Table 8: Generated specification in MiniZinc and Essence′ for a CSP instance with bounded-width structure, where $w = 3$ and $m = 2$.

Group size (w)	Maximum value (m)	Number of variables $w(wm+1)$	Gecode Time (secs.)	G12 Time (secs.)	FznTini Time (secs.)	Minion Time (secs.)
1	100	101	0.02	0.41	0.26	0.02
2	5	22	0.57	0.22	0.46	0.10
2	6	26	25.07	0.25	0.36	21.22
2	7	30	$>$ 15 min	0.25	0.11	$>$ 15 min
2	12	50	$>$ 15 min	0.40	2.91	$>$ 15 min
3	2	21	0.01	0.23	0.01	1.45
3	3	30	706.59	0.55	0.07	740.55
3	4	39	$>$ 15 min	103.19	0.23	$>$ 15 min
3	5	48	$>$ 15 min	$>$ 15 min	2.31	$>$ 15 min

Table 9: Solution times for Gecode, G12, FznTini and Minion on bounded-width CSP instances of the form shown in Table 8.

Group size (w)	Maximum value (m)	Number of variables $w(wm+1)$	Gecode Time (secs.)	G12 Time (secs.)	FznTini Time (secs.)	Minion Time (secs.)
1	100	101	0.02	0.37	0.24	0.02
2	5	22	0.01	2.98	0.05	0.06
2	6	26	0.01	166.26	0.03	0.02
2	7	30	0.01	$>$ 15 min	0.12	0.05
2	12	50	0.02	$>$ 15 min	1.66	0.02
3	2	21	0.03	0.49	0.02	0.02
3	3	30	0.07	$>$ 15 min	0.07	0.08
3	4	39	18.51	$>$ 15 min	0.11	10.55
3	5	48	$>$ 15 min	$>$ 15 min	0.15	$>$ 15 min

Table 10: Solution times for Gecode, G12 and Minion on bounded-width CSP instances of the form shown in Table 8, *with inequalities reversed.*

For larger values of w, the generated instances have width $2w$, because their structure has a simple tree-decomposition as a path of nodes, with each node corresponding to a constraint scope. In this case, although the problem is still tractable according to Definition 2.3, it cannot be solved efficiently using standard propagation algorithms. In fact, the runtimes of Gecode and Minion grow rapidly with problem size, as shown in Table 9. The runtimes for the G12 solver do not increase so fast for these specific instances, but if we reverse the inequalities, then they do increase in the same way (see Table 10), although in this case Gecode and Minion perform much better. Somewhat surprisingly, FznTini seems to be able to solve all of these instances fairly efficiently, even though they contain arithmetic inequalities which have to be translated into fairly large sets of clauses.

It is clear from these results that Gecode, G12 and Minion do not take advantage of the simple structure of the instance they are attempting to solve. Hence an important opportunity to improve the performance of CSP solvers would be in finding an efficient way of taking advantage of instance structure by adapting the variable ordering or other aspects of the search process to the particular instance. Moreover, as the ordering can be set in the input file, the question arises as to whether those adjustments could be automatically identified by the translators as part of the pre-processing.

7 Conclusions

We believe that the results presented in this paper have established that the various ideas about different forms of tractable constraint satisfaction instances presented in the theoretical literature can provide a fruitful source of inspiration for the design of challenging benchmark instances.

The initial applications of these ideas, presented in the previous sections, have already identified significant differences between different solvers in their ability to exploit salient features of the problem instances they are given.

There are a number of technical difficulties to overcome in developing useful benchmark instances. First of all, unlike SAT solvers, there is no standard input language for CSP solvers. Some progress has been made in proposing standard specification languages, and in providing automatic translation between different input languages, but these are currently far from complete. We have seen in Section 3 that the translation from MiniZinc to FlatZinc, or from Essence$'$ to Minion, can sometimes obscure the nature of an essentially simple problem, and hence badly affect the efficiency of the solution. We suggest that a better awareness of the factors of a problem specification that can ensure tractability could lead to better translation software, which ensures that such features are preserved. In particular, identifying tractable parts of an instance specification could lead to pre-processing tools that would automatically annotate such features in a way that could be exploited by a solver. This might be a simple matter of identifying useful value- or variable-orderings, or it might mean packaging some parts of the instance into (tractable) global constraints that can then use dedicated propagation algorithms to take advantage of their structure.

SAT solvers avoid many of the difficulties of translating between different input languages by adopting a single standard format for the input: all constraints must be expressed as clauses in CNF. However, the cost of this standardisation is a loss of expressive power. We have seen in Section 4 that translating simple forms of constraints such as linear inequalities into CNF may be very inefficient, and may lose the important features of the constraints which guarantee tractability.

Even when they have been successfully captured in an appropriate specification language, and input to a constraint solver, it can be the case that theoretically tractable instances may still be solved very inefficiently in practice. We have seen in Sections 5, and especially in Section 6, that when the tractability is due to a property that requires a higher level of consistency than arc-consistency to exploit, instances may be very challenging for standard solvers. It may be that this gap between theoretical notions of tractability, as expressed in Definition 2.3, and efficient solvability in practice, can suggest additional refinements that can usefully be added to the constraint-solving armoury (for example, some notion of adaptive consistency that invokes higher-levels of consistency when they can be easily shown to be effective). The dramatic progress in SAT-solving technology that has resulted from the exploitation of heuristics, such as clause learning and random restarts, that serve to focus the search more effectively, is an encouraging precedent. Finding effective automatic ways to improve the variable orderings and value orderings used by a solver according to specific relevant features of the input instance seems a promising first step which has not been sufficiently pursued. For example, the connection between having a bounded-width structure and the existence of a variable ordering with certain favourable properties (such as bounded *induced width*) is well-known [18, 14, 12] but does not seem to have been explored in any of the solvers or translation tools used in this study.

Summing up, in order to improve the performance of constraint solvers we need effective benchmarks which can explore that performance over a range of different problem types with different characteristics. One way to systematically develop such benchmarks is to use the insights from the theoretical study of constraint satisfaction. Benchmarks derived in this way can be simple enough to analyse in detail, and yet challenging enough to reveal specific weaknesses in solver techniques. This paper has begun to explore the potential of this approach, but much remains to be done.

References

[1] G12/MiniZinc constraint solver. Software available at `http://www.g12.cs.mu.oz.au/minizinc/download.html`.

[2] Gecode constraint solver. Software available at `http://www.gecode.org/`.

[3] Third international CSP solver competition. Instances and results at `http://cpai.ucc.ie/08/`.

[4] M. Bodirsky. Constraint satisfaction problems with infinite templates. In *Complexity of Constraints*, volume 5250 of *Lecture Notes in Computer Science*, pages 196–228. Springer Verlag, 2008.

[5] A. Bulatov, A. Krokhin, and P. Jeavons. Classifying the complexity of constraints using finite algebras. *SIAM Journal on Computing*, 34(3):720–742, 2005.

[6] A. Bulatov and M. Valeriote. Recent results on the algebraic approach to the CSP. In *Complexity of Constraints*, volume 5250 of *Lecture Notes in Computer Science*, pages 68–92. Springer, 2008.

[7] Andrei A. Bulatov. A dichotomy theorem for constraint satisfaction problems on a 3-element set. *Journal of the ACM*, 53(1):66–120, 2006.

[8] D. Cohen. Tractable decision for a constraint language implies tractable search. *Constraints*, 9:219–229, 2004.

[9] D. Cohen, P. Jeavons, and M. Gyssens. A unified theory of structural tractability for constraint satisfaction problems. *Journal of Computer and System Sciences*, 74:721–743, 2007.

[10] M.C. Cooper, D.A. Cohen, and P.G. Jeavons. Characterising tractable constraints. *Artificial Intelligence*, 65:347–361, 1994.

[11] N. Creignou, S. Khanna, and M. Sudan. *Complexity Classification of Boolean Constraint Satisfaction Problems*, volume 7 of *SIAM Monographs on Discrete Mathematics and Applications*. Society for Industrial and Applied Mathematics, 2001.

[12] V. Dalmau, Ph. Kolaitis, and M. Vardi. Constraint satisfaction, bounded treewidth, and finite-variable logics. In *Proceedings 8th International Conference on Constraint Programming—CP'02*, volume 2470 of *Lecture Notes in Computer Science*, pages 310–326. Springer-Verlag, 2002.

[13] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55(1):87–107, 1992.

[14] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[15] Y. Deville, O. Barette, and P. van Hentenryck. Constraint satisfaction over connected row convex constraints. *Artificial Intelligence*, 109:243–271, 1999.

[16] T. Feder and M.Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: A study through Datalog and group theory. *SIAM Journal on Computing*, 28:57–104, 1998.

[17] E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32:755–761, 1985.

[18] E.C. Freuder. Complexity of k-tree structured constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 4–9, 1990.

[19] A. Frisch, W. Harvey, C. Jefferson, B. Martnez-Hernndez, and I. Miguel. The essence of ESSENCE: A constraint language for specifying combinatorial problems. In *In Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 73–88, 2005.

[20] I. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In *Proceeedings ECAI 2006, 17th European Conference on Artificial Intelligence*, pages 98–102. IOS Press, 2006. Software available at `http://minion.sourceforge.net/`.

[21] G. Gottlob, L. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124:243–282, 2000.

[22] M. Green and C. Jefferson. Structural tractability of propagated constraints. In *Proceedings 14th International Conference on Constraint Programming—CP'08*, pages 372–386, 2008.

[23] M. Grohe. The structure of tractable constraint satisfaction problems. In *Mathematical Foundations of Computer Science 2006*, volume 4162 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2006.

[24] M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM*, 54:1–24, 2007.

[25] M. Gyssens, P.G. Jeavons, and D.A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66(1):57–89, 1994.

[26] J. Huang. FznTini constraint solver. Software available at `http://users.rsise.anu.edu.au/~jinbo/fzntini/`.

[27] J. Huang. Universal Booleanization of constraint models. In *Principles and Practice of Constraint Programming - CP'08*, volume 5202 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 2008.

[28] P.G. Jeavons, D.A. Cohen, and M. Gyssens. Closure properties of constraints. *Journal of the ACM*, 44:527–548, 1997.

[29] P.G. Jeavons and M.C. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, 79(2):327–339, 1995.

[30] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. MiniZinc: Towards a standard modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming - CP'07*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[31] J.K. Pearson and P.G. Jeavons. A survey of tractable constraint satisfaction problems. Technical Report CSD-TR-97-15, Royal Holloway, University of London, July 1997.

[32] Andrea Rendl. TAILOR - tailoring Essence′ constraint models to constraint solvers. Software available at `http://www.cs.st-andrews.ac.uk/~andrea/tailor/`.

[33] T.J. Schaefer. The complexity of satisfiability problems. In *Proceedings 10th ACM Symposium on Theory of Computing, STOC'78*, pages 216–226, 1978.

[34] P. van Hentenryck, Y. Deville, and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.