The University of York Department of Computer Science

Submitted in part fulfilment for the degree of MEng.

Comparing SAT and SMT Encodings of All-Different and Global Cardinality Constraints

Charles Turner

06/05/2014

Supervisor: Alan M. Frisch

Number of words = 17836, as counted by `texcount -inc`. The count includes words in the body of the text, headings and captions.

Abstract

Satisfiability (SAT) solving technology has improved dramatically since the 1960's. New techniques have allowed industrial problems involving tens of thousands of variables and millions of clauses to be solved, which is astounding given the difficulty of SAT. With these advances in solving technology, the desire to use languages richer than propositional logic has grown. Satisfiability Modulo Theories (SMT) builds on SAT by incorporating supporting theories for concepts like arithmetic, arrays, bit-vectors and uninterpreted functions. SMT has already enjoyed success in hardware and software verification tools, and more recently it has shown promise for constraint satisfaction problems (CSPs.) Extensive research has been invested in how best to attack constraint satisfaction problems with SAT, but this question has barely been addressed for SMT. Hence, this project investigates a particular aspect of solving constraint satisfaction problems, the encoding of problems into SMT compared to encodings of problems into SAT. Sophisticated encodings are compared against simple encodings. These sophisticated encodings are found to reduce search significantly in both SAT and SMT. Surprisingly, despite the reduction in search causing a corresponding reduction in run-time for SAT, the same is not observed in SMT. The experiments in this report suggest that the cost of this reduction in search is greater in SMT than it is in SAT. The report concludes with suggestions based on the experiments of when a sophisticated encoding is preferred in SMT, and puts forward some suggestions for why the cost of reduced search is greater in SMT.

Acknowledgements

Many thanks go out to my supervisor, Alan Frisch, for guiding me on my first foray into academic research. His humour and ability elucidate otherwise enigmatic topics with the lightest of touch kept me motivated throughout the duration of my project.

I would also like to extend thanks to the team working on the Z3 theorem prover, whose responsiveness to bug reports and questions was highly appreciated in times of desperation.

Finally, I am eternally grateful to my loving family for their kindness and support.

Statement of Ethics

This project is concerned with theoretical Computer Science, and no Human participants (other than the author and supervisor) were involved in the creation of this report. No personal data is disclosed in this report, all referenced works are published material and thus confidentiality violations were not applicable. Results were not doctored, work was not plagiarised. I have endeavoured to write as true an account of what I did as my expository capabilities permit. The Microsoft Office program was used to develop the figures in this report. This is a licensed copy used on the Computer Science lab machines. The other software used for this report was distributed under various Open Source and Free Software licenses, and thus further licensing issues were not relevant. I have no control over the ethical implications of any use of this work, but due to its theoretical nature the chances of harm being caused to the health and/or welfare of other organisms in the universe are beyond my comprehension. There is little I can do to even *attempt to ensure* that the results of this project will be used in socially irresponsible ways, as I have no control over the desires of other people. So as not to discriminate, I have used gender neutral terminology when appropriate, and have avoided the use of cultural and religious references in my work.

Contents

1 Introduction

This project investigates the suitability of a particular technology (SMT) for tackling combinatorial optimisation (CO) problems. Despite the somewhat scholastically opaque name, CO appears in many pragmatic problems, the rapid solution of which can be the keystone of a company's success.

With the availability and affordability of ever more sophisticated computing resources becoming available year after year, and the bounding algorithmic advancements in CO software, many industries and researchers are turning to systems that enable them to solve hard problems in a declarative manner, relying on the sophistication of CO software and available computer resources[1]. "Hard problems" typically means those problems which are NP-complete, and the CO community has amassed general techniques for tackling such problems. The grand challenge of such pursuits, termed the "Holy Grail" of programming[2], where one merely states the problem and the computer solves it, efficiently, could be considered the limit point of all research in this area.

From its humble description, propositional satisfiability (SAT) is one such technique that has enjoyed many years of fruitful academic endeavour and industrial success[3, 4]. In SAT, one expresses their problem as conjunctions of disjunctions of Boolean variables. While this may sound intimidating and devoid of application, SAT is typically used as a destination for one of Computer Science's most fundamental ideas: problem reduction. Users reduce their problem of interest or necessity into the SAT framework, and those many years of advancements in SAT technology can be applied, frequently to the immense satisfaction of the user, relieved from the burdensome proposition of having to implement a bespoke, efficient software for the task at hand.

SAT, with its language of propositional logic, has been found somewhat restrictive by people wishing to reduce ever more sophisticated problems into it[5]. SAT Modulo Theories (SMT) is a generalisation of SAT that combines theories of linear integer arithmetic, equality and uninterpreted functions, and several other formalisms into one coherent whole; allowing the user to utilise a more expressive language to describe their problem. The increased generality of SMT can come at cost to efficiency, and on the contrary, the ability to describe your problem at a higher level of abstraction can allow the solver to exploit structure of the problem that a further reduction into SAT may obfuscate. The various theories packaged by SMT have also enjoyed success in industrial settings. The theory of equality and uninterpreted functions ($\mathcal{EUF}$) has been deployed in the verification of pipelined microprocessors[6], the theory of bit-vectors ($\mathcal{BV}(\mathbb{Z})$) has been deployed in the verification of register-transfer level (RTL) circuit designs[7], the theory of arrays ($\mathcal{AR}(\mathbb{Z})$) has been studied in the verification of software[8] and the list goes on. Needless to say, SMT is a very general framework, with new theories being integrated as applications call for them. This report will be primarily concerned with the theory of (quantifier-free) linear-integer arithmetic ($\mathcal{LIA}$) in SMT.

At yet a higher level of abstraction comes Constraint Programming (CP). In CP, the problem to be solved and the means of solving it are decoupled from one another. The user specifies their problem in (typically) a declarative manner, called the model, in which they must describe the variables of the domain, the values applicable to those variables and the constraints among

variables which must be accommodated for a solution to be of interest. The coupling of the model is with a search procedure. The CP system considers the constraints and systematically searches, with the use of logical inference procedures (termed propagators), for a solution that satisfies the requirements ordained by the modeller. To be effective, these propagators must be carefully constructed to make the most out of the information about the problem they are given. They must reduce from the typically exponential amount of options those options which are implied by the statement of the problem, and guide the search procedure to lucrative avenues, not dead-ends. For these reasons, propagators have been extensively studied, and many efficient algorithms now exist for their resolution[9].

When modelling in the declarative CP language, the modeller has at their disposal a collection of general constraints. Such constraints have been made available due to years of modelling experience where their utility has been verified. These constraints are often called global constraints, and by issuing such global constraints, we can in effect describe yet a higher-resolution picture of our problem to a solver, which ideally could utilise this information to most intelligently conclude a solution rapidly. Capturing these commonly occuring patterns can tremendously improve efficiency[10, 11].

Here then, are three approaches to solving these "hard problems." SAT, SMT and CP. It might seem that with more abstraction comes a longer time to find a solution, due to the added sophistication of the abstractions. Such questions of efficiency are of course dependent on a nexus of factors, and statements such as "SAT is faster than SMT which is faster than CP" are not particularly meaningful. The more relevant question is: "When is one technique more appropriate than another in a given situation?" In [12], benchmarks were carried out demonstrating that SMT was a viable technology for the solution of CSPs. In particular, their experiments showed that over a suite of constraint satisfaction problems, their translation of these problems into SMT performed more favourably over a significant number of problems that the state-of-the-art CP systems, despite little effort being expended on finding the best encodings in SMT. Further evidence was provided by [13] who developed a compiler from CSPs to SMT and demonstrated its excellent performance compared to specialised systems built for CSPs. [13] explain that despite naïve encodings, their compiler still performs very well and suggest that a study of appropriate encodings in SMT could help make the technology even more competitive in CSPs. As applied to CO problems, the questions of when is SAT effective and when is CP effective at solving CO problems has been extensively studied, but the same question for SMT has barely been addressed.

This project investigates a small piece of this larger research question. In particular, the effectiveness of constraint reformulation[2], also known as decomposition, in SMT. A reformulation of a global constraint is its substitution with a simpler set of constraints that preserves the logical meaning of the global constraint. Such decompositions have been found to be highly effective in SAT and CP[14–17] for several reasons [18]

1. Global constraints, because of their need to be highly efficient, are often implemented as a special built-in procedures of the solver being used. By using a reformulation, also known as a decomposition, we can reuse existing infrastructure and thus obviate the need to design and implement a special-purpose propagator.

2. Some decompositions are designed to simulate a special-purpose propagator[14]. By decomposing the mechanics of such a propagator, its internal state can be exposed to the rest of the problem, which can yield performance improvements because of the extra information. It can also yield performance regressions, because of *too much* information.

3. By decomposing a set of a global constraints, their internal states may be communicated to another, and this added mutual information can again be a great help, or a monstrous hindrance.

The decompositions of two global constraints are studied in this report. The "All Different" (AllDifferent) global constraint and the "Global Cardinality Constraint" (GCC). The decompositions of these two constraints have a parameter which can be altered to change their behaviour. This parameter is called a Hall interval and will be introduced when required. The change in behaviour is rather mysterious, with no previous experiments studying the change of the parameter over the parameter's entire domain for a given problem. In this report such a study is carried out, but a theoretical explanation as to the reason for the observations has escaped the author and hence is not presented.

The decomposition studied in this work for AllDifferent was found to perform very well in SAT by [14]. The reason being that the decomposition is highly informative, reducing the amount of search the SAT solver has to perform. This reduction in search is due to the extra *propagation* entailed by the decomposition. However, while in SAT this extra propagation appears to be handled expediently, my experiments suggest that in SMT the cost of propagation is much higher than the cost in SAT.

This report also concludes with some observations of the effect that "more intelligent" propagators have on the problems studied in SMT. I conjecture that some measure of problem difficulty should be developed in SMT and that the more intelligent propagators should be applied in SMT only when the problem is, for some definition, very difficult.

The relevant background is presented in Chapter. 2. In Chapter. 3 formal accounts and detailed discussions of all the decompositions studied are presented. Then, in Chapter. 4, the decompositions introduced are experimentally studied over two problems selected to exercise the decompositions of global constraints employed in them. Finally the conclusions of these experiments are drawn in Chapter. 5, where future work is also considered.

2 Background

2.1 Satisfiability

Satisfiability is expressed in terms of propositional logic. Propositional logic is a language for dealing with 'propositions', which are viewed as atoms impervious to being broken down for further analysis. The language consists of *primitive symbols* and *operator symbols*. The primitive symbols of interest in this report are *propositional variables*, which can refer to Boolean truth values such as *true* (often denoted with the numeral 1), and *false* (often denoted with the numeral 0). The operator symbols of interest in this report are the logical NOT ($\neg$), logical AND ($\wedge$), logical OR ($\vee$), logical implication ($\implies$), logical equivalence ($\iff$), and finally parentheses used for grouping. A *formula* is a syntactic notion that determines whether a string of symbols has been constructed according to the rules of composition in propositional logic. There are two rules which define propositional formulas recursively. In particular (1) each primitive symbol is a formula, and (2) if ϕ and ψ for formulas, then so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$ and $\phi \implies \psi$. If these topics are unfamiliar to the reader, then a reference on logic[19] should be consulted prior to this report.

The propositional satisfiability problem (SAT) is that given a propositional formula $F(b_1, b_2, \ldots, b_n)$ in so-called conjunctive-normal form (CNF), which is a conjunction ($\wedge$) of disjunctions ($\vee$) find an assignment of Boolean values to $b_1, b_2, \ldots, b_n$ such that $F(b_1, b_2, \ldots, b_n) = 1$, or prove that no such assignment exists[20]. A Boolean value is either true (1) or false (0). For example, the formula

$$F(b_1, b_2, b_3) = (\neg b_1 \vee \neg b_2) \wedge (\neg b_2 \vee b_3) \wedge (b_1 \vee \neg b_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \tag{2.1}$$

is satisfiable under the assignment $b_1 b_2 b_3 = 101$, if we rule out this solution, as in the formula

$$G(b_1, b_2, b_3) = F(b_1, b_2, b_3) \wedge (\neg b_1 \vee b_2 \vee \neg b_3) \tag{2.2}$$

then we say that G is *unsatisfiable*, it has no satisfying assignments.

There are several other bits of nomenclature worth reviewing here. The terms b_i above are called *variables*. A variable can be written with different letters, so both x_2 and ζ_{100} would be considered variables. It also often convenient to denote variables by their numerical subscripts. So 'x_2' might be denoted by '2', and '$\neg b_5$' might be denoted '$\neg 5$'. Such usage should be unambiguous from the context, and the notation is mostly used for convenience of typing and economy of space. A variable can be *assigned* a Boolean value, or a variable can be *unassigned*, that is, having not yet been considered in the search for a solution. If all the variables in a formula F have been assigned Boolean values, then we call this situation a *complete assignment*. If only a strict subset of the variables have been assigned Boolean values, then we call it a *partial assignment*.

A *literal* is a variable or its negation. So both b_2 and $\neg b_2$ are literals. As are 10 and $\neg 10$.

A *clause* is a formula such as $l_1 \vee \ldots \vee l_m$ where each l_k, $1 \leq k \leq m$ is a literal[21]. A formula is in CNF if it is a conjunction of clauses. In other words, a formula is said to be in CNF when it is composed a conjunction of zero or more clauses, where a clause is disjunction of zero or more literals, where a literal is a boolean variable or its negation[20]. CNF is often referred to a "Product of Sum" in the digital logic community.

Example 1. The formula

$$(x \vee \neg y) \wedge z \tag{2.3}$$

is in CNF, but $(x \wedge y) \vee z$ and $x \wedge \neg(y \vee z)$ are not.

A CNF formula with zero clauses is logically equivalent to true. A CNF formula containing an empty clause (i.e., with no literals) is unsatisfiable since this is an impossible requirement. □

Using well known logical equivalences such as De Morgan's laws, every propositional formula can be converted into CNF, such a transformation allows us to deal with the satisfiability of a set of clauses, rather than a general formula, which is an easier problem to deal with. The idea of reducing formulas to ones with fewer connectives is commonly applied in automated logical reasoning, which is discussed further in §2.4.1.

As a final notation example, clauses are often represented as sets of literals. Recalling how variables such as 'x_j' can be written as 'j', a possible rewriting of Equation. 2.1 is

$$F(b_1, b_2, b_3) = \{\{\neg 1 \neg 2\}, \{\neg 23\}, \{1 \neg 3\}, \{1 \neg 23\}\} \tag{2.4}$$

where the logical operators have been elided in the notation, the assumption being that this set of sets is representing a formula in CNF.

SAT is a celebrated NP-complete problem[22, p. 260], but despite this very efficient procedures exist to solve the problem in practice[20, 23], and some of these will be discussed in §2.3. Because of these advances, SAT solvers have been used in a wide variety of practical applications often providing remarkable performance improvements[24]. For vivid examples of encoding interesting puzzles and practical problems such as graph colouring, factoring integers, fault testing circuits, bounded-model checking and digital tomography, see §7.2.2.2 of [25].

Solving SAT problems is the duty of *SAT solvers*. In this report I will be using the Z3 theorem prover, which comes with a state-of-the-art SAT solver built in. A language is required to issue SAT problems to the SAT solver. A standard input language by the name of DIMACS is commonly used in the SAT community.

A variation on the SAT problem that will be considered in this report is the *Pseudo-Boolean Satisfiability* problem (PB-SAT). PB-SAT can be viewed as a generalisation of clauses in SAT[26]. In PB-SAT, the clauses are linear combinations of boolean variables with integer coefficients. In particular, a PB-clause looks like

$$C_1 b_1 + C_2 b_2 + \ldots + C_{n-1} b_{n-1} \bowtie C_n \tag{2.5}$$

where $b_i \in \{0, 1\}$ and $\bowtie \in \{<, >, \leq, \geq, =\}$. Analogously to the SAT problem, PB-SAT problems are solved by PB-SAT solvers. Also analogously to the SAT problem, a language is required to issue PB-SAT problems to a PB-SAT solver. For lack of a standard name for this format, in this report I shall overload the meaning of PB-SAT to refer both to this generalisation of propositional clauses and to refer to a specific file format used for issuing PB-SAT problems to PB-SAT solvers. This use should be unambiguous from context.

In this report, I will be using an off-the-shelf tool called MiniSat+ to perform the translation from PB-SAT problems to equisatisfiable SAT problems. MiniSat+ translates problems expressed in PB-SAT into propositional CNF formulae. The first step in this translation is to normalise the PB-SAT constraints. For example, coefficients are divided by their greatest common divisor, and trivial propagations are performed: if there exists a formula of the form $x > 1$, then MiniSat+ will immediately return *unsatisfiable*, since no 0/1-variable x can be greater than 1. Furthermore, from constraints such as $2x + y + z >= 3$, MiniSat+ will propagate

that $x = 1$, since if $x = 0$, then $y + z >= 3$, which is clearly not possible for the 0/1-variables y and z.

Once normalisation has been performed, MiniSat+builds a *binary decision diagram* (BDD). This BDD can then be converted into what is called a *reduced boolean circuit* (RBC), which can be represented as another tree-like structure whose nodes are logical operators like AND, OR and ITE (if-then-else). The nodes of the RBC are then converted in SAT using a known technique called the Tsietin encoding[27]. Further details of how MiniSat+works can be found in [26].

2.2 Satisfiability Modulo Theories

SMT is the problem of deciding the satisfiability of a first-order formula with respect to some decidable first-order background theory $\mathcal{T}$[5]. Propositional satisfiability is therefore a special case of SMT, where $\mathcal{T}$ is a so-called "empty theory", since propositional logic is subsumed by SMT. In SMT, we might be interested in solving formulas like

Example 2.

$$g - b \geq 7 \quad \wedge \tag{2.6}$$
$$F(G(\mathbf{read}(A, g))) + g^2 = G(b - 4) \quad \wedge \tag{2.7}$$
$$F(b + g - 1) \neq 0 \tag{2.8}$$

□

where theories such as (linear) arithmetic (2.6), arrays and non-linear arithmetic (2.7), uninterpreted functions (2.8) and propositional logic ($\wedge$) are all at play in composition. A detailed walk-through of how an SMT solver might accomplish this is given in section §2.4.2.

2.2.1 Example Encoding

To ground the discussion of SMT, I shall encode a simple scheduling problem suggested to me by my supervisor, Alan Frisch. The problem will demonstrate using one particular background theory available in SMT, the linear integer arithmetic theory ($\mathcal{LIA}$).

In this problem we have five tasks each with an associated duration, or time to complete. For this example, we will have $\{A : 2, B : 1, C : 2, D : 2, E : 7, F : 5\}$ meaning that task A requires two time units to complete. To make the problem more interesting, there are some constraints on the order in which we may execute these tasks:

- Tasks A and C may not overlap.
- No two of tasks B, D or E can overlap.
- Tasks D and E must be completed before task F starts.
- Task A must complete before B starts.

We can introduce integer variables X, X_t, where X will hold the time at which task X starts, and X_t will hold the (constant) time task X requires to complete.

The overlapping constraints are quite easy to encode in SMT. If task A may not overlap with task C, then one of two conditions must be true:

1. Task A must complete before task C.

2. Task C must complete before task A.

Expressing this as a disjunction of linear inequalities is sufficient,

$$A + A_t \leq C \vee C + C_t \leq A \tag{2.9}$$

In an analogous manner, the remaining constraints from the problem specification can be expressed,

$$\begin{aligned}
&\textit{B, D and F can not overlap} \\
&B + B_t \leq D \vee D + D_t \leq B \\
&B + B_t \leq E \vee E + E_t \leq B \\
&D + D_t \leq E \vee E + E_t \leq D \\
&\textit{D and E must complete before F} \\
&D + D_t \leq F \wedge E + E_t \leq F \\
&\textit{A must complete before B} \\
&A + A_t \leq B
\end{aligned} \tag{2.10}$$

If we presented the above constraints to an SMT solver, we would likely not receive a satisfactory result. We must also ensure that the start times are non-negative and that our domains are appropriately constrained. For each variable X, we must say that $X \geq 0$. If *End* is the end time of the schedule, then for each variable X and its corresponding end-time X_t, we must post that $X \leq End - X_t$.

The mathematics presented here are actually rather close to a concrete implementation ready for consumption by an off-the-shelf SMT solver. There are some syntactic differences, however. I will be using the SMT solver packaged with the Z3 theorem prover in my experiments, and like several other solvers, one of the formats it accepts problem descriptions in is defined by a project called SMT-LIB2[28].

In the SMT-LIB2 format, expressions are entered using parenthesised prefix notation, the same notation used in the Lisp programming language. The concrete encoding into SMT-LIB2 of the formulas in Equations. 2.9 2.10 is,

```
(set-logic QF_LIA)
(set-option :produce-models true)

(declare-fun A () Int)
(assert (>= A 0))
(declare-fun At () Int)
(assert (= At 2))
(declare-fun B () Int)
(assert (>= B 0))
(declare-fun Bt () Int)
(assert (= Bt 1))
(declare-fun C () Int)
(assert (>= C 0))
(declare-fun Ct () Int)
(assert (= Ct 2))
(declare-fun D () Int)
(assert (>= D 0))
```

```
(declare-fun Dt () Int)
(assert (= Dt 2))
(declare-fun E () Int)
(assert (>= E 0))
(declare-fun Et () Int)
(assert (= Et 7))
(declare-fun F () Int)
(assert (>= F 0))
(declare-fun Ft () Int)
(assert (= Ft 5))

(declare-fun End () Int)
(assert (= End 14))

(assert (or (<= (+ A At) C) (<= (+ C Ct) A)))
(assert (or (<= (+ B Bt) D) (<= (+ D Dt) B)))
(assert (or (<= (+ B Bt) E) (<= (+ E Et) B)))
(assert (or (<= (+ D Dt) E) (<= (+ E Et) D)))
(assert (and (<= (+ D Dt) F) (<= (+ E Et) F)))
(assert (<= (+ A At) B))
(assert (<= A (- End At)))
(assert (<= B (- End Bt)))
(assert (<= C (- End Ct)))
(assert (<= D (- End Dt)))
(assert (<= E (- End Et)))
(assert (<= F (- End Ft)))

(check-sat)

(get-value (A B C D E F))
```

The first line of this listing sets the background logic to be used by Z3. In this case, quantifier-free linear integer arithmetic (`QF_LIA`) has been selected. The second line asks Z3 to keep the model it generates for subsequent printing. The first block of code is declaring the function symbols of our problem. The domains of these symbols must be nonnegative, and furthermore, the time required for each task is asserted. The `End` function symbol is what must be minimised in this problem. The next block encodes the constraints of this example, and finally Z3 can be asked check whether the program is satisfiable and if it is, to print out the values of starting times for each variable A through F (formally called the *model*).

If this code is saved in a file called `alan.smt2`, then a command such as `z3 alan.smt2` will give us an answer:

```
sat
((A 0)
 (B 9)
 (C 2)
 (D 0)
 (E 2)
 (F 9))
```

Z3 has found a solution in which task A starts at time zero, B at time 9 and so on. This is the minimal make-span (difference between end time and start time) of the problem as specified above, which can be verified by replacing `(assert (= End 14))` with `(assert (= End 13))`. With this replacement, Z3 will respond `unsat`, for unsatisfiable, meaning there exists no model

that satisfies the formula.

2.3 Solving SAT

Fundamentally, to solve the SAT problem for some formula $F(x_1, \ldots, x_n)$ a complete assignment α must be found in the search space of 2^n complete assignments such that α satisfies F. A simple way of achieving this would be to evaluate $F(x_1, \ldots, x_n)$ for each of the 2^n complete assignments over $x_1, \ldots, x_n$ until either an assignment that makes F equal true is found, or there are no further complete assignments to explore. An improvement to this scheme is backtracking, which may find satisfying assignments much faster, but falls back to simple enumeration to prove unsatisfiability. An algorithm is shown in Algorithm. 1. This algorithm works by branching on the assignments to literals. Formulas in this algorithm are represented as sets of clauses. So $x \wedge (\neg y \vee z)$ is represented as $\{\{x\}, \{\neg y, z\}\}$, and for computational reasons, we normally assign a unique integer to each variable, and represent a variable's negation by the negative of its associated integer, such as $\{\{1\}, \{-2, 3\}\}$. If the formula is the empty set, then we have satisfied every clause and return true. If the empty set is contained in the formula, then a clause could not be satisfied and false is returned. The EVAL function deletes literals that have been assigned false from every clause they appear in, and removes clauses containing a literal that has been assigned true. After the termination checks, we pick an unassigned literal from the set of clauses, and branch on the result of recursively calling the BACKTRACK function with the resulting assignment. An example run of this algorithm is shown in Fig. 2.1, where the formula to check for satisfiability is $F = (1 \vee \neg 2) \wedge (1 \vee 3) \wedge (\neg 1 \vee \neg 3) \wedge (3 \vee 4) \wedge (\neg 4 \vee \neg 1)$. In this example, CHOOSE-NEXT-LITERAL simply picks the first unassigned variable in the set of clauses, but in general the goal is to chose a variable which simplifies the problem instance as much as possible[29]. A common heuristic in this spirit is to choose the most constrained variable first.

Algorithm 1 A simple backtracking algorithm for SAT

```
1:  function BACKTRACK(assignment, F)
2:      nextF ← EVAL(assignment, F)
3:      if nextF = ∅ then
4:          return true
5:      end if
6:      if ∅ ∈ nextF then
7:          return false
8:      end if
9:      l ← CHOOSE-NEXT-LITERAL(nextF)
10:     if BACKTRACK(ADD(assignment, l), nextF) then
11:         return true
12:     else
13:         return BACKTRACK(ADD(assignment, ¬l), nextF)
14:     end if
15: end function
```

The backtracking algorithm in Algorithm. 1 is far from optimal. There are various optimisations that can be made. The *DPLL*-procedure[30] incorporates a couple of heuristics to the BACKTRACK algorithm that greatly improve search times. The DPLL-procedure extends the BACKTRACK algorithm with two further checks before branching on a literal,

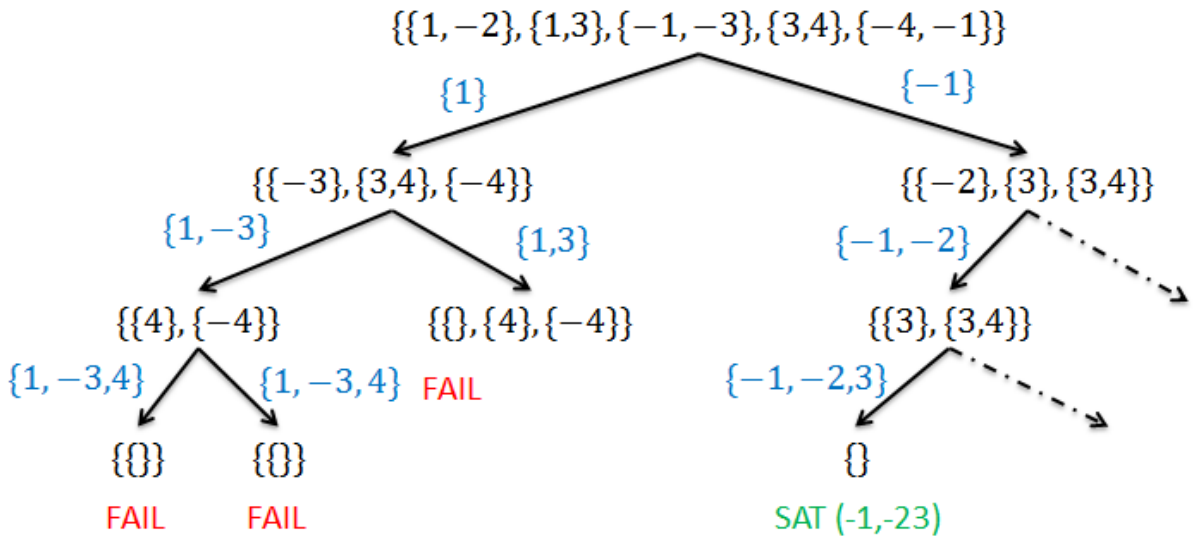

Figure 2.1: A run-through of the backtracking algorithm. Negative numbers represent negated literals. Dashed lines represent choice points unexplored. The blue sets represent the current assignment. Note that the traversal is left-to-right depth-first. This pattern of backtracking through previous assignment is referred to as *chronological* backtracking, the word "chronological" appealing to exploration of events as they happened in the past. This method of backtracking is contrasted with *non-chronological* backtracking, which will be described later in this section.

- **Unit clause elimination**. A *unit clause* is a clause which contains only one literal $\{p\}$. In this case, the algorithm recurs with the current assignment extended by p. The propagation of this rule, that is, its repeated application, is referred to as *unit propagation* or *Boolean constraint propagation* (BCP).
- **Pure literal elimination**. A literal p is called *pure* in a formula F if there is no additional occurrence of its negation $\neg p$ in F. If F contains a pure literal, the algorithm recurs with the current assignment extended by p.

Pure literal elimination is rarely implemented as check in this recursive procedure due to efficiency reasons[31, p. 943]. It is often implemented as one of the pre-processing steps SAT solvers perform before the actual search. Moreover, a distinction between how literals are given values must now be drawn. Either a value for a variable is explicitly chosen in the branching stage, or a variable's value is forced by the two heuristics mentioned above, such a forced variable is commonly called an *implied* variable. An assignment of the first variety is called a *decision assignment*. The implicit search tree may now be decorated with such decision assignments. When a conflict (an unsatisfiable clause) is detected, instead of weaving back up the tree in a left-to-right depth-first manner as done in the Backtrack algorithm, the algorithm *jumps* back up the tree to the last decision assignment. This technique is called *backjumping*[31], and also speeds up the search process for practical problems.

Example 3. To demonstrate the DPLL-procedure, I will walk through the example given for the Backtrack algorithm above using notation developed in [31]. In this notation, a *state* of the search process is denoted by $M \| F$, where M is a (partial) assignment and F is a formula in CNF. States are related to one another by a relation, denoted $\Rightarrow$. We transition between states using one of the four rules in the DPLL procedure. These rules are Decide, UnitPropagate, PureLiteral, and Backtrack, corresponding to the stages of the algorithms presented above. A decision assignment for a variable x is denoted x^{d}.

$$\begin{aligned}
\emptyset \| 1 \vee \neg 2, \neg 1 \vee \neg 3, 3 \vee 4, \neg 4 \vee \neg 1 &\Rightarrow (\mathsf{Decide}) \\
1^{\mathsf{d}} \| 1 \vee \neg 2, \neg 1 \vee \neg 3, 3 \vee 4, \neg 4 \vee \neg 1 &\Rightarrow (\mathsf{UnitPropagate}) \\
1^{\mathsf{d}} \neg 4 \| 1 \vee \neg 2, \neg 1 \vee \neg 3, 3 \vee 4, \neg 4 \vee \neg 1 &\Rightarrow (\mathsf{UnitPropagate}) \\
1^{\mathsf{d}} \neg 43 \| 1 \vee \neg 2, \neg 1 \vee \neg 3, 3 \vee 4, \neg 4 \vee \neg 1 &\Rightarrow (\mathsf{UnitPropagate}) \\
\neg 1^{\mathsf{d}} \| 1 \vee \neg 2, \neg 1 \vee \neg 3, 3 \vee 4, \neg 4 \vee \neg 1 &\Rightarrow (\mathsf{Backtrack}) \\
\neg 1^{\mathsf{d}} \neg 2 \| 1 \vee \neg 2, \neg 1 \vee \neg 3, 3 \vee 4, \neg 4 \vee \neg 1 &\Rightarrow (\mathsf{UnitPropagate}) \\
\neg 1^{\mathsf{d}} \neg 23 \| 1 \vee \neg 2, \neg 1 \vee \neg 3, 3 \vee 4, \neg 4 \vee \neg 1 &\ (\textit{Final state})
\end{aligned}$$

Note here that only made two decision assignments on variable 1. The rest of the decisions were inferred by the heuristics. Also notice how the application of the Backtrack rule jumped past two assignments, an example of backjumping.

The satisfying assignment in this example is $\{\neg 1, \neg 2, 3\}$, as before with the Backtrack algorithm. □

It has been estimated that SAT solvers spend more than 90% of their run-time in the BCP step[23]. It is therefore not hyperbolic to say that optimising this procedure is of great importance. The reason so much time is spent in BCP can be seen by asking the question: "How does the solver find out when a clause becomes unit?" A naïve approach would be as follows: every time an assignment is made, the solver scans each clause in the database to see if the assignment

caused it to become unit. Since there can be millions of clauses in a given problem, this is clearly not a winning strategy. The CHAFF solver employed an optimisation referred to as *watched literals*[23] to address this need for optimisation. For each clause in the problem, two literals are selected to play the role of watched literals. Note that all clauses of interest in this optimisation must be of length greater than two, since unit clauses are exactly what BCP removes. The two literals are chosen from those variables which are either currently unassigned or false. Hence, when *one* of these two watched literals becomes true, two possibilities remain: either the clause is still not unit, in which case another unassigned or false variable is selected to join the remaining watched literal, or the clause has indeed become unit and the variable implying this unit clause removal is the remaining watched literal. This scheme has several other algorithmic pleasantries that all competitive modern SAT solvers enjoy. This optimisation does not effect the search space explored by the SAT solver, only the time efficiency of the SAT solver. I thought the reader may find it interesting to see just one of many clever implementation tricks employed to make SAT solvers so useful for many practical problems.

A common technique that **does** alter the search space explored by SAT solvers is *conflict driven clause learning*[20] (CDCL). The fundamental trick here is to exploit the structure of BCP. When BCP is applied to a formula, often several variables will become assigned, and certain clauses will become satisfied. Some clauses will also become unsatisfied and hence conflict with the current assignment. This process implying (through BCP) variables' assignments and the status of previously unresolved clauses reveals a graph. The edges in such a graph encode which variables' assignments were implied from other assignments, and through which clauses the implications arose.

Example 4. (**Adapted from [32]**.) In this example, I will show how a clause can be learnt from a conflict, so as to avoid making the same mistake in the future. As Santayana once said: "Those who cannot remember the past are condemned to repeat it."

Consider the formula,

$$\begin{aligned}\phi &= \omega_1 \wedge \omega_2 \wedge \omega_3 \wedge \omega_4 \wedge \omega_5 \wedge \omega_6 \\ &= (x_1 \vee x_{31} \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \\ &\quad \wedge (x_{21} \vee \neg x_4 \vee \neg x_6) \wedge (x_5 \vee x_6),\end{aligned} \tag{2.11}$$

and assume that the assignment $x_{21} = 0$ was decided at level 2 of the backtracking search (denoted $x_{21} = 0@2$), that $x_{31} = 0@3$ and that the current assignment decision $x_1 = 0@5$. What will happen next? To find out, BCP must be applied. Running through each clause, the following assignments are implied,

$$\begin{aligned} x_1 = 0@5 \underset{\text{BCP}}{\Longrightarrow} \quad & x_2 = 0@5 \quad (\omega_1) \\ & x_3 = 0@5 \quad (\omega_2) \\ & x_4 = 1@5 \quad (\omega_3) \\ & x_5 = 0@5 \quad (\omega_4) \\ & x_6 = 0@5 \quad (\omega_5). \end{aligned}$$

The bracketed ω_i expressions denote in which clause the implication was made. Note that if BCP is applied a decision level d, then all implied decisions are also considered to be made a decision level d, not $d + 1$ as you might possibly expect. This set of implicates has caused a

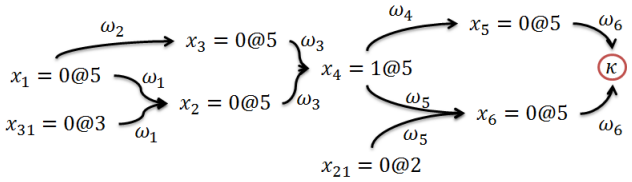

Figure 2.2: The implication graph after applying BCP to Equation. 2.11. An arc from a decision node x_i to a decision node x_j with edge label ω_k means that x_j was inferred from x_i in clause ω_k. The circled node on the far right is the conflict clause $\kappa = (x_5 \vee x_6)$.

conflict. For $\omega_6 = (x_5 \vee x_6)$ to be true, at least one of x_5 and x_6 must be true, but the implications have made both of them false. The conflict clause in this case is denoted $\kappa = (x_5 \vee x_6)$. The graph structure of this example is shown in Fig. 2.2

Consider what can be deduced from this conflict. Recall the conflicted clause $\kappa = (x_5 \vee x_6)$. The way in which a new clause is learned from this conflicted clause is to repeatedly apply an operator that transforms the conflicted clause, by following the reverse implications starting from κ into a clause that will ensure the search never considers this erroneous partial assignment again,

$$(x_5 \vee x_6) \underset{\omega_4}{\rightarrow}^{1} (\neg x_4 \vee x_6) \underset{\omega_5}{\rightarrow}^{2} (\neg x_4 \vee x_{21}) \underset{\omega_3}{\rightarrow}^{3}$$
$$(x_3 \vee x_2 \vee x_{21}) \underset{\omega_2}{\rightarrow}^{4} (x_1 \vee x_2 \vee x_{21}) \underset{\omega_1}{\rightarrow}^{5} (x_1 \vee x_{31} \vee x_{21}).$$

This set of steps is showing what the reverse traversal through the implication graph of Fig. 2.2 looks like. Starting from the conflict, κ, each literal in the clause is replaced by the negation of all the literals that implied it. The result of applying this process until a fixed-point is reached is $(x_1 \vee x_{31} \vee x_{21})$. This is the learnt clause; after being added to the clause database, the search can never explore this dead-end again. □

The learnt clauses may be later deleted by a clause deletion policy, since some instances can cause the solver to learn a vast number of clauses which may become a burden on the SAT solver's memory limitations. The conflict clauses can also be used to speed-up backtracking. By considering the level at which the conflict was originally initiated, the solver can non-chronologically backtrack to this epoch[20]. This overview of CDCL was necessarily brief, for more detail and an excellent historical perspective, see §2.2.3 of [33].

Finally, most contemporary SAT solvers also incorporate randomness in their search procedures[33]. The randomness can be employed in the choice of which variable to branch on next (the branching heuristics), or more commonly, when to stop exploring the current assignment and start afresh; known as a *random restart*. The random restart works by defining a particular cut-off depth in the backtracking search. If the solver backtracks to depths beyond the cut-off point, it will randomly restart itself with a different assignment, keeping the clauses learnt from the previous forays. This strategy has been shown to be very effective at reducing search times[34] and the method can be implemented to ensure that the completeness guarantee of DPLL is maintained.[35]

The SAT solver used in this report for conducting experiments in §4.1 is part of the Z3 theorem prover system. This SAT solver is state-of-the-art and incorporates CDCL, BCP and non-chronological backtracking, amongst other techniques not covered in this report[36].

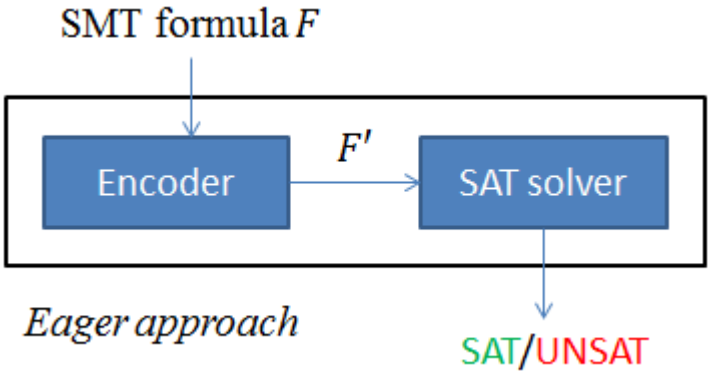

Figure 2.3: The Eager approach to solving SMT.

2.4 Solving SMT

There are two general approaches to solving an SMT problem. In the first, so called "Eager Approach", the problem of solving in some background-theory is reduced to that of solving an equisatisfiable propositional formula which SAT technology can handle. In the second so called "Lazy Approach", the formula in some background-theory is abstracted to a formula which contains only Boolean variables. SAT technology can then find models for these Boolean variables, and the validity of these models is verified by the underlying decision procedure the for original terms of the background theory. An overview of Eager approach is depicted in Fig. 2.3 and an overview of the Lazy approach is depicted in Fig. 2.4.

2.4.1 Eager SMT Approach

A surprisingly large amount of reduction can be performed on a formula in some variant of first-order logic to achieve an equisatisfiable formula in propositional logic. For this reason, early approaches to SMT solving focused on the translation to SAT. This was attractive because so much research had already been invested into SAT technology, and efficient decision procedures for the various background theories in SMT had not yet been developed.

One such reduction that can be performed is called Ackermann's reduction[37]. This reduction was first presented in [38], but I do not have access to this work. This procedure reduces formulas with uninterpreted function symbols to formulas of equality logic. The reason for performing this reduction is to make the decision problem easier. For example, proving the equivalence of two functions $F(x, y) = x * 1/y$ and $G(x, y) = x/y$ over 64-bit integers x and y is computationally very difficult. We can abstract the details of these functions into "uninterpreted functions" F^{UF} and G^{UF} and then just impose functional consistency requirements which state that the function returns the same output for the same inputs. Such abstractions as part of a larger model can turn an intractable search problem into a tractable one. The general idea of the reduction is to take a formula F^{UF} with uninterpreted functions in it and convert it to a formula F^{E} with the uninterpreted functions removed . Using notation in[37], the reduction looks like,

$$F^{E} := FC^{E} \rightarrow \text{flat}^{E},$$

where FC^{E} are the functional consistency constraints and flat^{E} is a so-called "flattening" of F^{UF} where the uninterpreted function symbols have been replaced by newly generated variables.

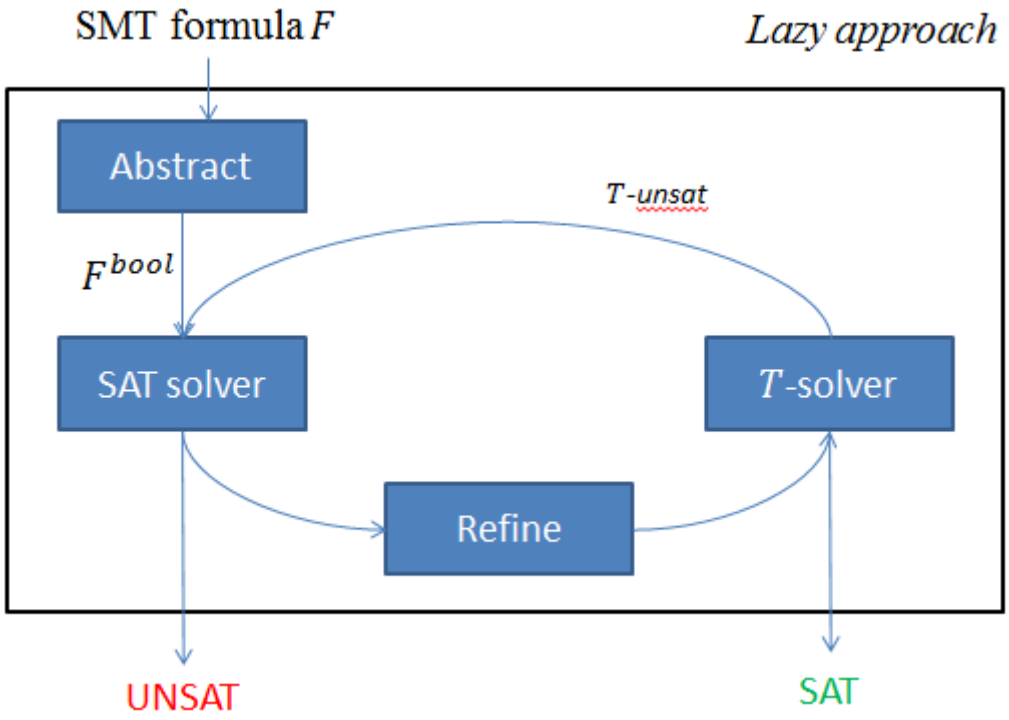

Figure 2.4: The Lazy approach to solving SMT.

Example 5. Consider the following formula,

$$F^{UF} := (x \neq z) \wedge (G(x) \neq G(y)) \wedge (G(y) = G(z))$$

In this example we have four uninterpreted function symbols (two of which are the same). For each of these distinct occurrences, a new variable is introduced. For this example: $g_1 = G(x)$, $g_2 = G(y)$ and $g_3 = G(z)$.

The flattening of F^{UF} is,

$$\text{flat}^E := (x \neq z) \wedge (g_1 \neq g_2) \wedge (g_2 = g_3)$$

and now functional consistency constraints are imposed,

$$FC^E := \begin{array}{rl} & (x = y \implies g_1 = g_2) \wedge \\ & (x = z \implies g_1 = g_3) \wedge \\ & (y = z \implies g_2 = g_3) \end{array}$$

The result of applying Ackermann's reduction is that F^{UF} is valid if, and only if, F^E is valid, where

$$F^E := FC^E \rightarrow \text{flat}^E.$$

□

The nice property about the result of applying the Ackermann reduction (or any reduction that removes uninterpreted function symbols) is that the resulting formula is in the language of

equality over symbols, and hence satisfies the small model property, which using the definition of R. Ramanujam[39] states that "a formula α is satisfiable iff it is satisfiable in a model whose size is bounded by some (at most) exponential function of $|\alpha|$." Using this property, a finite domain can be constructed such that the original formula is valid iff it is valid over all interpretations of this finite domain. It is this insight, and a common trick of letting each Boolean domain variable range over $[1..n]$, where n is the number of arguments taken by the function being operated on by the Ackermann reduction, that drives the so-called "Small Domain Instantiation" method for deciding formulas in equality logic by reduction to SAT.

Small Domain Instantiation[40]. Given a formula ϕ in equality logic, the Small Domain Instantiation method first maps each variable $x_i \in \phi$ into a *finite* set of integers. This mapping is denoted by $R(x_i)$. The goal is to minimise the size of this set of integers. The constraint on what this set of integers can be is that the original formula ϕ must be satisfiable if and only if ϕ is satisfiable over an interpretation with respect to $R(x_i)$, for $i \in [1, n]$. The intended meaning here is that ϕ is satisfied iff ϕ is satisfied after replacing each variable $x_i \in \phi$ with every $k \in R(x_i)$, its finite domain. With these finite domain so constructed, they are encoded into a suitable enumerated type (for example, a bit vector). With such an encoding, a binary decision diagram B_ϕ may be constructed. Formula ϕ is satisfiable iff B_ϕ is not identical to 0.

Binary decision diagrams (BDDs) are beyond the scope of this report, but they are introduced in[41]. In brief, they are concise graphical models of Boolean functions. The important point here is that testing whether a BDD for a formula F is not identical to 0 is equivalent to asking whether F is satisfiable. The equivalence is explained in more detail in[42, §6]. The Small Domain Instantiation as explained here is not small at all. I have simplified the presentation by using a trivial upper bound on the size of the set of integers to which each variable is mapped, that being n, the number of variables in the formula being reduced. [40] introduce a more sophisticated approach that shrinks this bound, and research in this area is concerned with further shortening the size of this set of integers.

What has been achieved here? From a formula ϕ with uninterpreted function symbols and equality, this exposition of the Eager Approach has reduced, through the use of Ackermann's reduction, ϕ to a formula ϕ' with uninterpreted function symbols removed, and finally the small domain instantiation method has reduced ϕ' to a object in a SAT-equivalent formalism, which can be solved using well established techniques.

The small domain instantiation method is just for reducing the formulas in the theory of uninterpreted functions with equality to SAT. There are several other theories that must be reduced. Propositional logic is not expressive enough for representing many real-world problems[5], and furthermore, eager encodings often result in more deduction steps than necessary[43]. This is because we have to encode the entire formula in one pass, and such an approach often encodes more than is required to show that a formula is unsatisfiable. The eager approach is not useless, and is used by the UCLID system, which has enjoyed success in the verification of pilelined microprocessors. However, eager approaches to theories involving arithmetic suffer from exponential encodings, and as such UCLID has been dominated by DPLL-based lazy SMT methods[5, 31] as described in §2.4.2.

2.4.2 Lazy SMT Approach

The lazy SMT approach[5] combines the success of DPLL-based SAT solvers with theory specific decision procedures. This approach frequently outperforms the eager approach, even by orders of magnitude[31], this is one of the reasons why most modern SMT-solvers use the lazy approach rather than the eager approach.

One reason why the lazy approach is efficient is that it works using layers of increasing complexity. The lowest layer in this framework is a DPLL procedure, as described in §2.3. If this layer can not find a satisfiable assignment, the lazy approach will immediately fail. Further layers are added on this propositional DPLL layer, incrementally considering harder decision problems. Hence, the more expensive decision procedures are called upon only when necessary by the current sub-problem being analysed. This approach was a precursor to what is now referred to as the DPLL(T), and is described more rigorously in [44].

In the lazy approach, the DPLL-algorithm described in §2.3 is extended to communicate directly with a theory solver (T-solver), this extension is denoted DPLL(T)[31]. The DPLL(T) procedure is general, any T-specific decision procedure can be plugged in and used, provided that it meets some implementation-defined interface for doing so. Contrast this with the *ad-hoc* nature of the eager approach. An overview of the approach is shown in Fig. 2.3. The conversation between the SAT solver and the theory-specific solver is orchestrated by two functions. The first function $T2\mathcal{B}$ transforms a T-formula F containing T-terms into a Boolean formula F^{bool} containing only propositional atoms. This mapping is called *abstraction*. The second function $\mathcal{B}2T$ is the inverse of $T2\mathcal{B}$, mapping a Boolean formula back into its T-formula. This mapping is called *refinement*.

Example 6. Consider the T-formula ϕ in a combination of the theories of arrays and linear arithmetic,

$$\begin{aligned}\phi := &(\text{read}(a, x_1) = x_3)\\ &\wedge (A_1 \vee \neg A_2 \vee 3x_3 \leq x_1)\\ &\wedge (\neg(3x_3 \leq x_1) \vee 2x_1 + 5x_3 = 21 \vee A_3).\end{aligned}$$

The result of abstraction is,

$$\begin{aligned}T2\mathcal{B}(\phi) := &B_1\\ &\wedge (A_1 \vee \neg A_2 \vee B_2)\\ &\wedge (\neg B_2 \vee B_3 \vee A_3).\end{aligned}$$

Note how the two occurrences of $3x_2 \leq x_1$ have been assigned the same Boolean variable B_2. Two T-terms are considered the same if they are syntactically identical[5], not if they are logically identical, so $2x_1 = 10$ and $x_1 = 5$ would not be assigned the same Boolean variable.

With reference to Fig. 2.4, the SAT-solver receives a Boolean abstraction and then uses a backtracking algorithm such as the one in Algorithm. 1 to present an assignment of the Boolean variables. Assume the first assignment is

$$\mu := \{B_1, \neg A_2, B_3\}.$$

The refinement of this assignment now passed to the T-solver,

$$\mathcal{B}2T(\mu) := \{(\text{read}(a, x_1) = x_3), \neg A_2, 2x_1 + 5x_3 = 21\}$$

The T-solver then uses its decision procedures for the theory of arrays and linear arithmetic to decide whether this set of formulas is satisfiable. If they are, then the result is true, otherwise we inform the SAT-solver that this Boolean abstraction is not satisfiable and then the SAT-solver either presents us with another Boolean assignment, or determines that no more assignments are possible, resulting in false being returned.

I have mentioned in passing how SMT often solves problems involving a "combination" of theories. Modern techniques for combining theories follow the theoretical work described in

[45]. One problem with this combination is that of theory-propagation. Recall that SAT-solvers employ heuristics to figure out how to proceed, such as how many times a variable occurs in distinct clauses, or whether a variable appears only positively, etc. Such decisions can be made in part because of the consistent nature of the representation, namely, propositional logic. The matter is complicated when using combined theories in SMT[46]. Consider the combination of the array theory and the linear arithmetic theory, how can solving a sub-goal in theory of arrays help the solver learn something about solving another sub-goal in theory of arithmetic, even if both sub-formulas contain the same variables? This is one potential drawback of the lazy approach compared to eager approaches, but it is noted in[5] that such drawbacks are often compensated for by other benefits of the lazy approach.

2.5 Constraint Programming

Constraint programming (CP) is a programming paradigm. Other examples of programming paradigms are procedural programming (C), functional programming (Haskell) and logic programming (Prolog). There are various constraint programming languages in which it is relatively easy to describe and solve constraint satisfaction problems (CSPs) in a declarative manner. What, then, is a CSP?

A constraint satisfaction problem (CSP) is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$[47] where $\mathcal{X}$ is an n-tuple of variables $\mathcal{X} = \langle x_1, x_2, \ldots, x_n \rangle$, $\mathcal{D}$ a corresponding n-tuple of finite domains[1] $\mathcal{D} = \langle D_1, D_2, \ldots, D_n \rangle$ such that each x_i takes its value from the corresponding D_i. The notation $\min(X)$ will be used to denote the minimum element in the domain of variable X, and the maximum will be denoted as $\max(X)$. $\mathcal{C}$ is a t-tuple of constraints $\mathcal{C} = \langle C_1, C_2, \ldots, C_t \rangle$[47].Let $\mathcal{V} = v_1, v_2, \ldots, v_k$ be a finite sequence of variables, then a constraint C on $\mathcal{V}$ is defined as a subset of the Cartesian product of the domains of the variables in $\mathcal{V}$[48]. More formally, $C \subseteq D_1 \times D_2 \times \ldots \times D_k$. This is often written as $C(\mathcal{V})$. Constraints constrain the domain D_i from which the variables take their values. When a constraint C equals $D_1 \times D_2 \times \ldots \times D_k$ and is non-empty, the constraint C is said to be *solved*.

Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP. A tuple $(d_1, \ldots, d_n) \in D_1 \times D_2 \times \ldots \times D_n$ is said to *satisfy* a constraint $C \in \mathcal{C}$ if $(d_1, \ldots, d_n) \in C$. If no such tuple satisfies a constraint C, the the constraint C is said to be inconsistent. A tuple $(d_1, \ldots, d_n)$ is called a *solution* to P if and only if it satisfies every constraint in $\mathcal{C}$[49]. A CSP is *consistent* if a solution exists for the CSP.

A distinction is often made in the literature between *unary constraints*, *binary constraints* and *n-ary constraints*. A unary constraint is a constraint over one variable, a binary constraint is over two variables, that is, $C(\mathcal{V})$ where $|\mathcal{V}| = 2$, and an n-ary constraint if over n variables. Such distinctions are unnecessary in theory, but the terms are occasionally convenient when referring to certain constraints. One final piece of terminology related to constraints is that of a *global constraint*. Some authors define this as being an n-ary constraint where $n > 2$[50, 51], but the definition used in this report, following [52], is to define it as a constraint which encapsulates other constraints, for example by expressing the global constraint as a conjunction of constituent constraints. Global constraints are heavily used in CP, and will be the focus of this report. Examples of global constraints are the "All Different" constraint introduced in §2.5.2 and the "Global Cardinality Constraint" intrduced in §2.5.2. Global constraints can in a loose sense provide more information about the structure of the problem being solved than single constraints alone. This higher-resolution picture of the problem structure is often found to improve solve times dramatically. A small example demonstrating this is presented in §2.5.1.

[1] In this report only finite-domain CSPs are considered, infinite-domain CSPs do also exist.

Given an abstract description of a problem to be attacked by CP, the constraint programmer must reformulate the abstract problem in a CSP. That is, the constraint programmer must find appropriate variables, domains and constraints that when solved yield a solution to the original problem. This is often a highly non-trivial task, and the process of reformulation is called *modelling*. Models are expressed in a modelling language. In this report, I have used the MiniZinc language which is a medium-level declarative modelling language[53]. The solver I have coupled with this modelling language is the G12 finite domain solver[54]. MiniZinc was chosen because of its easy-to-use declarative modelling language and large library of built-in global constraints.

What has this got to do with SAT or SMT? SAT and CSPs are two closely related problems with several interesting mappings between the two formalisms[55–57]. The most obvious of which is that both SAT and CSP are NP-complete problems, and hence by definition must be translatable to one-another in polynomial time. It stands to reason therefore that advancements in either SAT or CSP solving technology is of mutual benefit.

A popular encoding of CSP into SAT is the *direct encoding* [56]. For each CSP variable X_i, propositional variables x_{ik} are introduced, where $k \in \{1 \dots m\}$ and $m = |\mathcal{D}(X_i)|$. If $x_{ik} = 1$ in the SAT encoding, then variable X_i takes value k in the CSP encoding. Furthermore, to ensure that each CSP variable X_i is actually given a value, clauses such as $x_{i1} \vee x_{i2} \vee \ldots \vee x_{im}$ are posted. For this clause to be satisfied, at least one of x_{ik} must be true, that is, the CSP variable X_i must take at least one value. If we wish that each CSP variable X_i may take only one value, then we can enforce this with a binary clique of inequalities over the corresponding propositional variables: $\neg x_{ij} \vee \neg x_{ik}$ for all i, j and k such that $j \neq k$. This ensures that only one of x_{ik} can be true.

The literature contains many encodings of CSP into SAT, and [56] lists several of them. The above discussion should have made it clear how a CSP can be viewed as a SAT problem; and hence, it should be now be clear how we might view it as an SMT problem. CSPs can be solved using either SAT, SMT or CP systems. The choice of which to use is application-specific and beyond the scope of this report. Solving CSPs using SAT and/or CP has been extensively studied, but the problem has barely been addressed for SMT.

CSPs nearly[2] always have an element of *search* in them. In the most abstract sense, we define a finite-domain for each of our variables X_i, and ask the computer to find a subset of these domains that satisfy the constraints we have specified.

Example 7. To make these formalisms more palpable, consider a rather contrived example which will hopefully demonstrate how a model of a "real-world" scenario can be formulated as a CSP.

Imagine that you own a pottery. Each day, your pottery gets a number of requests for ceramic objects. You have a potting wheel and an oven. While clay is baking in the oven, you can be throwing clay on the wheel in parallel. For each request that comes in, an expected time of completion is given for both stages in the potting process, and as the benefactor of the pottery, you're interested in minimising the time taken to complete all the requests. Your first set of variables might be T_{sr} which represent the estimated time stage s takes for request r,

$$\{T_{sr} | s \in \{\text{Wheel}, \text{Oven}\}, r \in \text{Requests}\} \subseteq X.$$

The corresponding domain for T_{sr} would just be a finite set of integers. This is what you already know, you also make some variables that the constraint programming system will assign values

[2]Some CSPs can be solved without *any* search and just by inference. An example is the Sudoku problem, where AllDifferent constraints can reduce the domain of each variable to a single value, or determine that no such assignment exists by propagation alone, with no search

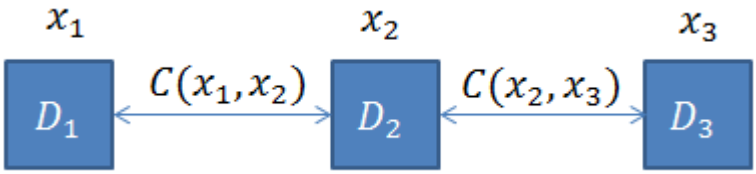

Figure 2.5: An example constraint network

to; these are the variables whose values you wish to find out. In this case, you want to know when you should start each stage of production for each request, let S_{sr} represent the start time of stage s for request r,

$$\{S_{sr} | s \in \{\text{Wheel}, \text{Oven}\}, r \in \text{Requests}\} \subseteq X.$$

The corresponding domain for S_{sr} is trickier and must be bounded for finite-domain CSP models. This can be done by choosing the upper bound to be $U = \sum_{s,r} T_{sr}$, which would be the time taken if you just sequentially performed all the requests, one at a time. Specifically, $\mathcal{D}(S_{sr}) = \{1, \ldots, U\}$. This is safe, but it is hoped that the constraint system will be able to find a sequence of actions that reduce this trivial upper bound.

Less formally now, some constraints over S_{sr} can be formulated, which are members of $\mathcal{C}$,

- You can't mould two different clays at the same time on the wheel
- A maximum of 2 distinct clays can be baked in the oven at the same time
- You must mould a clay before you bake it
- A clay must rest for M minutes after moulding before it is baked.

and so on. These constraints will squash the domains for each start time S_{sr}. To pick an appropriate set of domains, you must instruct the system to make a choice. You want to minimise the end time of all these jobs, so you introduce a new variable **end**, that represents this,

$$\textbf{constraint } \forall s, r \; S_{sr} + T_{sr} \leq \textbf{end}.$$

A constraint that **end** is less than 6-pm must also be added, so that you can eventually go home! Finally, the system can be asked to satisfy these constraints by **minimising** the **end** variable. The result will be an assignment to each of the variables S_{sr}, or a message stating that no assignment exists for this set of requests, indicating that you can't do this much work in one day. □

2.5.1 Consistency

We can view CSPs as *constraint graphs*. In such a graph, a node represents a variable, and an arc represents a *binary constraint*, which is simply a constraint between two variables. Such graphs can be generalised to contain *hyper-edges* which then represent constraints between more than two variables, but I shall ignore this in the presentation. The notation used is shown in Fig. 2.5. Boxes represents domains of variables, and arcs between the boxes represent (binary) constraints.

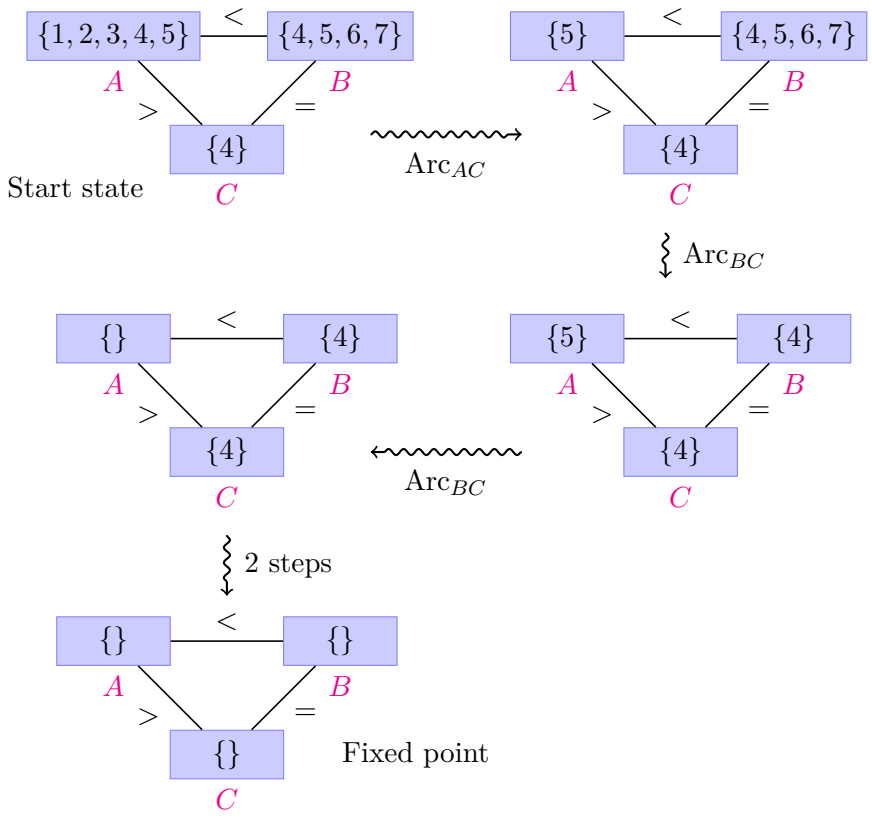

Figure 2.6: Overview of Arc Consistency Arc$_{AB}$ means arc consistency has been performed from A to B in the diagram. In the first transition, the only value in D_A that satisfies C_{AC} is 5, and therefore $D_A = \{5\}$. Skipping ahead to the second-to-last diagram, D_A has become empty, now there are no solutions to the constraints, and this information is eventually propagated around the network until all domain become empty.

The reason these constraint graphs have been introduced is to talk about how certain constraints can be verified as being *locally consistent* with the domains of our variables. Such consistencies are *local* because the checks in this section work constraint by constraint, rather than checking the consistency of the entire CSP in one fell swoop.

Arc Consistency. A constraint C on variables x_i and x_j is arc consistent *from* x_i to x_j if every value $x \in D_i$ takes part in some solution of C. A constraint C between x_i and x_j is arc consistent, written $AC(C)$, if it is both arc consistent from x_i to x_j and from x_j to x_i. A CSP is arc consistent if every binary constraint is arc consistent.

There are various algorithms for achieving arc consistency[58] and an optimal bound of $\Theta(ed^2)$ where e is the number of constraints in a CSP, and d is an upper bound on the size of a variable's domain. To give a demonstration of how domains can be reduced (and in this example, a CSP proven unsatisfiable), arc consistency has been performed between several pairs of variables in turn, shown in Fig. 2.6. The drawing finishes when we reach a so-called *fixed-point* where no further changes to the domains of any variable occur. This is not demonstrating any particular consistency algorithm, just the mechanics of performing arc consistency.

Generalised Arc Consistency. A binary constraint $C(x_1, x_2, \ldots, x_k)$ is generalised arc consistent from x_1 if and only if every value in the domain of x_1 takes part in some solution of C. A constraint C on a set $\mathcal{X}$ of variables is generalised arc consistent, written GAC(C), if and only

if it is generalised arc consistent from every $x_i \in \mathcal{X}$. A CSP is generalised arc consistent if and only if for every $C \in \mathcal{C}$, GAC(C) is true. This is called *hyper-arc consistency*, *domain consistency* or *generalised arc consistency* by various authors. In this report, the property is referred to as GAC. Note that GAC really is a generalised notion. A binary constraint is arc consistent if and only if it is generalised arc consistent. Checking GAC is in general NP-hard[59], which is why several other weaker forms of consistency (defined below) are used by the community, since enforcing GAC for all constraints is often too expensive.

It is important to realise that a CSP exhibiting GAC does not in general mean that the CSP is consistent. For example, the CSP depicted in Fig. 2.7 is GAC but not consistent. Consider x_1 and x_2: if $x_1 = 1$, then $x_2 = 2$ satisfies $\neq (x_1, x_2)$, similarly, if $x_1 = 2$, then $x_2 = 1$ satisfies $\neq (x_2, x_1)$. Hence $\neq (x_1, x_2)$ is arc consistent. The same holds for $\neq (x_1, x_3)$ and $\neq (x_2, x_3)$. However, the CSP is clearly **not** consistent. In essence, an attempt is being made to solve an instance of the graph-colouring problem in which the graph has a clique of three nodes with two colours.

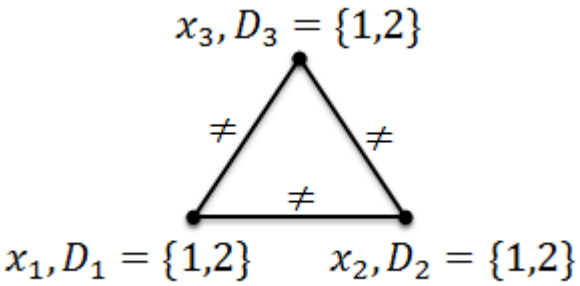

Figure 2.7: A CSP that is GAC but not consistent.

A slight extension to the situation shown in Fig. 2.7 which demonstrates the potential effectiveness of global constraints over binary constraints is shown in Fig. 2.8. It was mentioned in §2.5 that global constraints can evince a deeper structure of the problem being solved than binary constraints. In Fig. 2.8, notice that arc consistency on the binary constraints can not remove any values from any variables' domain, since all values are mutually consistent. However, imagine there exists a global constraint called All Different (see §2.5.2) that parametrises the three disequalities. By reasoning globally across all three binary constraints, an appropriate propagation algorithm can efficiently deduce that $x_3 = 3$, as required.

GAC is a very expensive level of consistency to maintain, especially on *arithmetic* CSPs, that is, CSPs with arithmetic constraints. For example, imagine applying GAC to a constraint such as $430A - 103B + 9C + 17D = 213$, where the domains of A, B, C and D are all $\{1, 2, \ldots, 10000\}$. In general, is it NP-hard to decide if these arithmetic constraints are GAC[60]; the same hardness as solving the whole CSP.

For these reasons, two relaxations of GAC designed for use in arithmetic constraints were developed: bounds and range consistency.

Bounds consistency[50]. A constraint $C(x_1, x_2, \ldots, x_n)$ $(n > 1)$ is *bounds consistent* (BC) if and only if for all $i \in \{1, 2, \ldots, n\}$ and each value $d_i \in \{\min(\mathcal{D}(x_i)), \max(\mathcal{D}(x_i))\}$, there exist **integer** values $d_j \in [\min(\mathcal{D}(x_j)), \max(\mathcal{D}(x_j))]$ for all $j \in \{1, 2, \ldots, n\} \setminus i$ such that $(d_1, d_2, \ldots, d_n) \in C$. There are some subtle differences in definitions of bounds consistency throughout the literature, [59] explain the differences, using their taxonomy, the flavour of bounds consistency used in this report is bounds($\mathcal{Z}$). While there may be dramatic differences

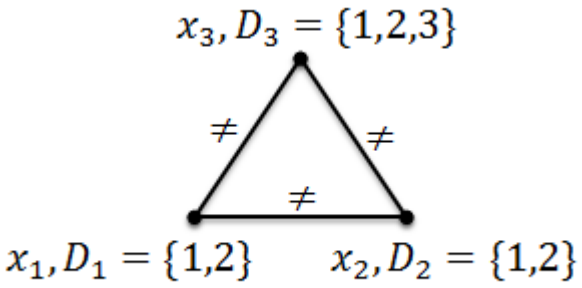

Figure 2.8: A consistent CSP which demonstrates the effectiveness of global constraints versus binary constraints.

between these flavours of bounds consistency when applied to linear constraints (bounds($\mathcal{D}$) and bounds($\mathcal{Z}$) are NP-complete, whereas bounds($\mathcal{R}$) is only linear time), the definitions are equivalent for the propagators studied in this report.

Range consistency[50]. *Range consistency* (RC) is a stronger form of consistency than BC, but a relaxation of GAC. Specifically, a constraint $C(x_1, x_2, \ldots, x_n)$ $(n > 1)$ is RC if and only if for all $i \in \{1, 2, \ldots, n\}$ and for all values $d_i \in \mathcal{D}(x_i)$, there exist **integer** values $d_j \in [\min(\mathcal{D}(x_j)), \max(\mathcal{D}(x_j))]$ for all $j \in \{1, 2, \ldots, n\} \setminus i$ such that $(d_1, d_2, \ldots, d_n) \in C$. Note that instead of considering the bounds $\{\min(\mathcal{D}(x_i)), \max(\mathcal{D}(x_i))\}$ as in BC, we consider the entire domain.

To summarise the various consistency notions described in this section, a partial order $\preceq$ will now be defined over them. Let P and P' be two CSPs, and let $|P| = |\bigcup_i \mathcal{D}(X_i)|$ be the size of the combined variables' domain in the CSP. Then $P \preceq P'$ if and only if $|P| \leq |P'|$. It can now be proven that $\text{GAC}(P) \preceq \text{RC}(P) \preceq \text{BC}(P)$.

To prove $\text{GAC}(P) \preceq \text{RC}(P)$, consider the constraint $X = 2Y + Z$ where $\mathcal{D}(X) = \{3, 4\}$, $\mathcal{D}(Y) = \{1, 3\}$ and $\mathcal{D}(Z) = \{1, 3\}$. This is RC since $X = 3$ is satisfied by $Y = 1$, $Z = 1$ and $X = 4$ is satisfied by $Y = 1, Z = 2$ where $2 \in [1, 3]$. $X = 4$ is not valid for GAC however, since there does not exist values for $Y \in \{1, 3\}$ or values for $Z \in \{1, 3\}$ to make $2Y + Z = 4$.

To prove $\text{RC}(P) \preceq \text{BC}(P)$, consider the constraint $X \neq Y$ where $\mathcal{D}(X) = \{1, 2, 3\}$ and $\mathcal{D}(Y) = \{2\}$. This is BC since for $x_1 = \min(\mathcal{D}(X)) = 1$, there exists a value $2 \in \mathcal{D}(Y)$ such that $x_1 \neq 2$. Similarly for $x_3 = \max(\mathcal{D}(X)) = 3$. However, this constraint is not RC, since we consider all domain values of X, not just its bounds, and for $x_2 = 2 \in \mathcal{D}(X)$, there does not exist a corresponding value $y \in \mathcal{D}(Y)$ such that $x_2 \neq y$.

The inequalities above follow from the arguments for $\text{GAC}(P) \preceq \text{RC}(P)$ and $\text{RC}(P) \preceq \text{BC}(P)$ since the transitivity of $\leq$ implies the transitivity of $\preceq$.

2.5.2 Global Constraints

This section motivates two examples of global constraints studied in Chapter. 4, the "all different" (AllDifferent) and "global cardinality" (GCC) constraints. Also discussed are the propagator algorithms for the AllDifferent and GCC constraints from which the respective decompositions in [14] are directly derived.

The propagator algorithms utilise the notion of a *Hall interval*. Therefore, a Hall interval is first defined.

Hall intervals are exploited in [61] to develop a propagation algorithm for AllDifferent that maintains RC. They are also exploited in [62] develop a propagation algorithm for AllDifferent that maintains BC.

Consider a problem with variables $X_1, X_2, \ldots, X_n$. A Hall interval is an interval of k values from $\mathcal{D}(X_1) \cup \mathcal{D}(X_2) \cup \ldots \cup \mathcal{D}(X_n)$ which forms of superset of the domains of k variables. Formally, an interval $[l, u]$ is a Hall interval if and only if $|\{i|\mathcal{D}(X_i) \subseteq [l, u]\}| = u - l + 1$.

Example 8. In this example the Hall intervals for the set variables depicted in Fig. 2.9 are identified.

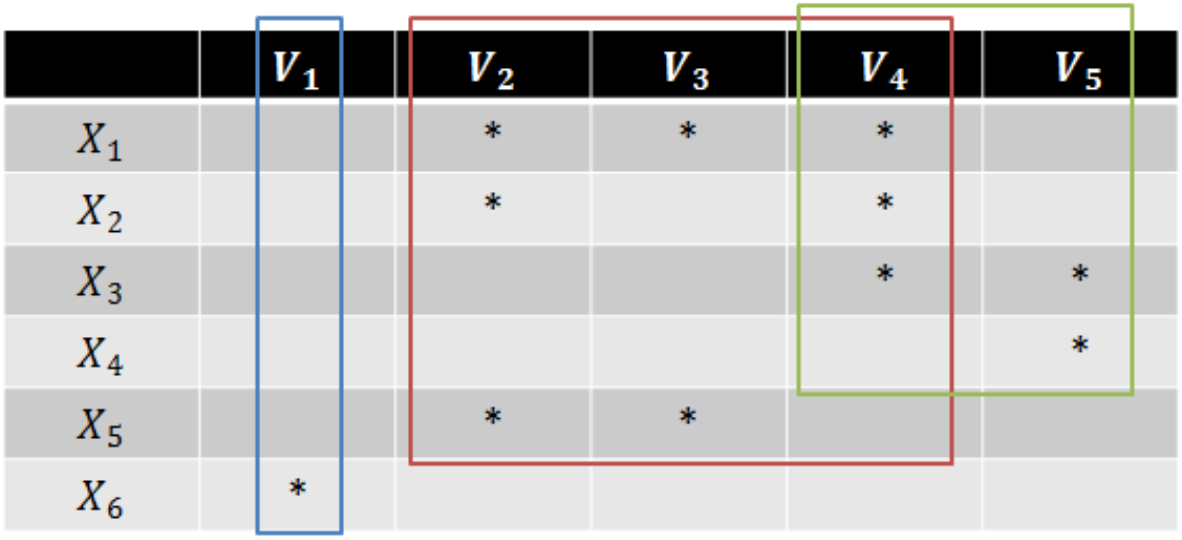

Figure 2.9: Fictitious setup demonstrating Hall intervals.

The boxes indicate which Hall intervals exist in this example. The blue box is a Hall interval of size 1, since exactly one variables' (X_6) domain is contained in the set of domain values $\{V_1\}$. Similarly, the red box is a Hall interval of size 3, since exactly three variables' (X_1, X_2 and X_5) domains are contained in the set of domain values $\{V_2, V_3, V_4\}$. □

All Different

The AllDifferent constraint takes a set of variables $\{x_i\}$, each with a corresponding finite domain, and says that for each pair of variables x_i and x_j, where $i \neq j$ that $x_i \neq x_j$. In other words, AllDifferent$(x_1, x_2, \ldots, x_n)$ is logically equivalent to $\bigwedge_{1 \leq i < j \leq n} x_i \neq x_j$[63]. The AllDifferent constraint is one of the most frequently used global constraints in constraint programming due its wide applicability. For instance, [64] use the AllDifferent constraint to optimally schedule air traffic flows. In their work, n aircraft wish to travel to and from a set of destinations along the straight-line paths (termed "flows") between those locations. Since more than one aircraft might be travelling in the same flow, a set of flight levels are provided. These are horizontal levels vertically separated by about 2000ft, allowing multiple aircraft to travel along the same straight line. If r_i are variables representing the route for aircraft n_i and L_i are variables representing the level at which aircraft n_i flies in, then part of the model looks like,

$$\forall i, j \quad 1 \leq i < j \leq n \quad \text{intersect}(r_i, r_j) \implies L_i \neq L_j, \tag{2.12}$$

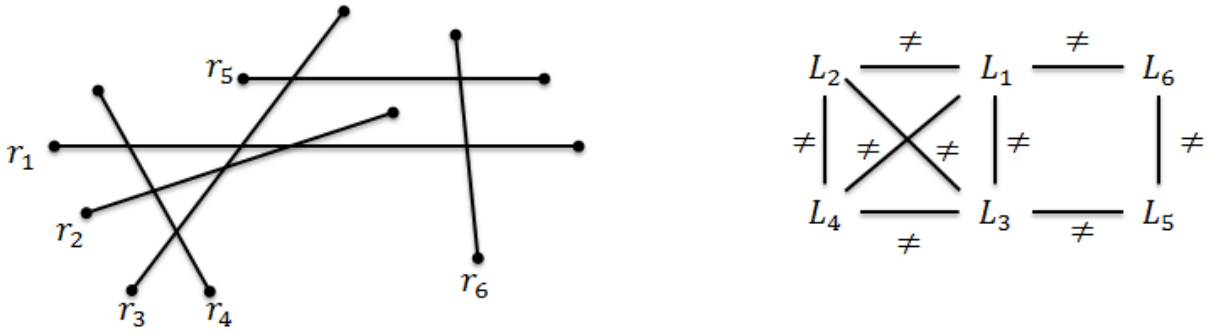

Figure 2.10: The picture on the left shows some fictitious flight routes represent by solid lines. Points of intersection imply that aircraft travelling along the intersecting routes must be in different vertical levels. The picture on the right represents the corresponding level constraints as local disequalities.

meaning that if two routes intersect, then the airplanes must be in different horizontal levels L_i, L_j to avoid a collision. An example flight network given in [64] is shown in Fig. 2.10.

In the constraint network on the right of Fig. 2.10, the only constraints are local disequalities. The method of [64] was to find all maximal cliques in such a constraint network and then post an AllDifferent over that clique. Such a method was found to enormously cut down on the search required to prove optimality of the scheduled flight networks. For example, in Fig. 2.10, the maximal clique is just $\{L_1, L_2, L_3, L_4\}$, from which an AllDifferent constraint can be posted: AllDifferent(L_1, L_2, L_3, L_4). Once that constraint is posted, the disequalities can be removed, since they are subsumed by AllDifferent.

As another example, AllDifferent was employed by [65] as part of the solution to the problem of scheduling personnel in a work rota, in particular, it was used to constrain that no-one can work two jobs simultaneously. Another reason why AllDifferent is so widely known by the constraint programming community is because many global constraints can be expressed as generalisations or extensions of the AllDifferent constraint. The Global Cardinality Constraint is one such constraint of this type, and it will be explained in §2.5.2. Other examples of global constraints that can be considered extensions of AllDifferent include sort, diffn and cycle[50]. For an encyclopedic overview of the AllDifferent constraint, consult the entry in the Global Constraint Catalogue [66].

Hopefully the reader is convinced that AllDifferent is a widely useful constraint with many applications. It is therefore important to have an efficient algorithm for such a frequently used constraint. There exist propagator algorithms for AllDifferent enforcing GAC[10], RC[61] and BC[62, 67, 68]. The decomposition for AllDifferent presented in [14] is inspired by the propagator algorithm in [61], and so it is this algorithm briefly considered in this report. It might be unclear why the decomposition selected achieves mere RC instead of the stronger GAC. Why not just decompose the GAC propagator? By an intriguing result, there does not exist a polynomial size decomposition of the AllDifferent constraint that maintains GAC[69]. This means that it would be intractable to try and use such a decomposition for non-toy problems.

Example 9. Before getting to the meat of the algorithm, an illustration of what a RC propagator (or decomposition) must do is first presented. This example is taken from [14], however, they have not correctly enforced RC, in this example, RC is correctly applied. Consider the setup in

Table. 2.1.

	1	2	3	4	5
X_1			*	*	
X_2	*	*	*	*	
X_3			*	*	
X_4		*	*	*	*
X_5	*				

Table 2.1: Set of variables

First off, $[1, 1]$ is a Hall interval since X_5 is the only variable whose domain in completely contained in this interval. Therefore, no other variables can take support in this domain, in particular, we can remove 1 from X_2's domain. The situation now as depicted in Table. 2.2.

	1	2	3	4	5
X_1			*	*	
X_2		*	*	*	
X_3			*	*	
X_4		*	*	*	*
X_5	*				

Table 2.2: The state after 1 has been removed from X_2's domain.

From Table. 2.2 the next Hall interval to be seen is $[3, 4]$. Exactly two variables, X_1 and X_3, take their support in this Hall interval of size 2. So again, no other variables can take a value in this interval. Hence, the values 3 and 4 must be removed from the domains of X_2 and X_4, and the situation is now as depicted in Table. 2.3.

	1	2	3	4	5
X_1			*	*	
X_2		*			
X_3			*	*	
X_4		*			*
X_5	*				

Table 2.3: The state after 3 and 4 have been removed from X_2 and X_4's domain.

From Table. 2.3, a new Hall interval, $[1, 1]$ has shown up. In particular, X_2 is the only variable whose domain is not contained in this interval. Hence, all other variables must take their values outside of this interval. Therefore, we must now remove 2 from X_4's domain, finally yielding the situation in Table. 2.4.

The domains in Table. 2.4 are now RC. This example has demonstrated the sort of reasoning employed by the RC propagator algorithms in [61] and [62], as well as the result after applying GAC to the decomposition presented in [14]. □

Pseudo code for the algorithm proposed in [61][3] is shown in Algorithm. 2. This algorithm works

[3]The description here is based on pseudo-code from [70], since I could not obtain access to [61]. The pseudo-code has been significantly modified since it did not appear correct, or I misunderstood its purpose. I have an implementation in Python available for inspection[71]

	1	2	3	4	5
X_1			*	*	
X_2		*			
X_3			*	*	
X_4					*
X_5	*				

Table 2.4: The state after 3 and 4 have been removed from X_2 and X_4's domain.

by first sorting the variables in two ways: by their maximum domain values and their minimum domain values. We can explicitly store these two sorted instances, or, as in Algorithm. 2, implicitly store them in the looping constructs. The iteration order ensures that Hall intervals are detected from right-to-left, from smallest to largest. The inner-loop in lines 6–23 is detecting the Hall intervals in the domains and removing those values from which a variable must not take support from its domain. [61] proved that this algorithm is optimal for RC.

Algorithm 2 A propagator algorithm that achieves RC.

```
1: function LECONTE-ALLDIFF-RC([X_1, X_2, ..., X_n]) return The X_i with RC domains.
2:     while Fixed-point over X_i not reached do
3:         for X_j in decreasing order of max(D(X_j)) do
4:             count← 0
5:             k ← NIL.
6:             for X_i in decreasing order of min(D(X_i)) do
7:                 if k is not NIL then
8:                     D(X_i) ← D(X_i) \ [min(k), max(D(X_j))]
9:                 end if
10:                if max(D(X_i)) ≤ max(D(X_j)) then
11:                    count ← count + 1
12:                    if count = max(D(X_j)) − min(D(X_j)) + 1 then
13:                        k ← D(X_i)
14:                    end if
15:                end if
16:            end for
17:            if k is not NIL then
18:                for X_i such that max(D(X_j)) < max(D(X_i)) do
19:                    if min(k) ≤ min(D(X_i)) then
20:                        Remove min(k) from D(X_i)
21:                    end if
22:                end for
23:            end if
24:        end for
25:    end while
26: end function
```

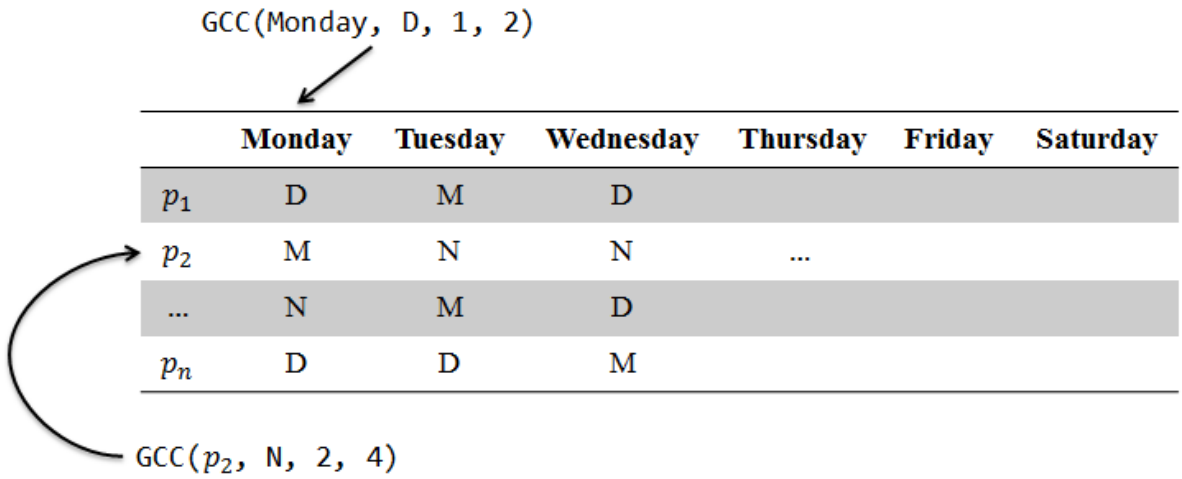

Figure 2.11: Example of the sorts of GCC involved in scheduling problems.

Global Cardinality

Like AllDifferent, GCC has numerous applications, particularly in scheduling problems[72]. The example given in [72] is that a crew-scheduling problem. In this problem, a set A of activities is given, which denote the shift a worker is assigned to. Three shifts occur in each day of the week. Three of the shifts are morning, day and night. Also given is a set P of persons that can be scheduled to work, and a set W denoting the days of the week.

The sorts of constraints involved in scheduling problems like this are of the following form: A person $p \in P$ must work at least j day shifts per week and at most k day shifts per week. Or there must be at least m night shifts on a Wednesday and at most n night shifts on a Wednesday. A pictorial description of such a scenario is shown in Fig. 2.11. The obvious local constraints for such a scenario are of the form `atleast(n, Night, Wednesday)` and `atmost(m, Night, Wednesday)`. But for the same reasons as were outlined in §2.5.1, a GCC can filter the domains of such a problem more effectively with its global approach.

For an encyclopedic overview of the GCC constraint, consult the entry in the Global Constraint Catalogue [73].

More formally, the GCC can be specified as follows: given a set of variables $X = \{x_1, \ldots, x_m\}$ where each x_i takes its values from a subset of $V = \{v_1, \ldots, v_n\}$, the GCC constrains the number of times some value v_i can be assigned to a variable in X to be between l_i and u_i, where L is the set of lower bounds $\{l_1, \ldots, l_n\}$ and U is the corresponding set of upper bounds $\{u_1, \ldots, u_n\}$. In other symbols, GCC may be written as,

$$GCC(X, V, L, U) \equiv \bigwedge_{s \in [1,n]} l_s \leq \sum_{i=1}^{m} [x_i = v_s] \leq u_s \tag{2.13}$$

Where $[B]$ is the Iverson bracket notation, meaning $[B]$ is one if B is true, and zero otherwise. As a quick example, $\text{GCC}(\{x_1, x_2, x_3\}, \{1, 2, 3\}, \{1, 2, 0\}, \{2, 3, 3\})$ would be equivalent to,

$$\begin{aligned} &(1 \leq [x_1 = 1] + [x_2 = 1] + [x_3 = 1) \leq 2) \wedge \\ &(2 \leq [x_1 = 2] + [x_2 = 2] + [x_3 = 2] \leq 3) \wedge \\ &(0 \leq [x_1 = 3] + [x_2 = 3] + [x_3 = 3] \leq 3) \end{aligned} \tag{2.14}$$

Example 10. The GCC is rather subtle, so in this example several sets of bounds are shown and the question of whether they are satisfiable or unsatisfiable is explored, so as to give a flavour of the nature of GCC constraints. Consider the set of variables in Table. 2.5,

	1	2	3
X_1	*	*	
X_2		*	
X_3	*	*	*
L	l_1	l_2	l_3
U	u_1	u_2	u_3

Table 2.5: Set of variables

Scenario 1: $\mathbf{L} = \{\mathbf{0}, \mathbf{2}, \mathbf{0}\}, \mathbf{U} = \{\mathbf{3}, \mathbf{2}, \mathbf{3}\}$. The lower bounds for our three variables are 0, 2, and 0 respectively. The upper bounds are 3, 2, and 3 respectively. Said another way, this set of bounds is enforcing that between 0 and 3 variables can take the value 1, *exactly* 2 variables take the value 2, and between 0 and 3 variables take the value 3. This case is easily seen to be satisfiable. X_2 has to equal 2 as before, but since there must be exactly two variables taking the value 2, one of X_1 or X_3 must take value 2. Assume that $X_3 = 2$. Now there are no choices and is must be the case that $X_1 = 1$. Hopefully it is clear that AllDifferent is just GCC with $l_i = 1$ and $u_i = 1$ for all value of i. Hence GCC is a generalisation of AllDifferent.

Scenario 2: $\mathbf{L} = \{\mathbf{1}, \mathbf{2}, \mathbf{1}\}, \mathbf{U} = \{\mathbf{2}, \mathbf{3}, \mathbf{1}\}$. It is immediately seen that this is not a satisfiable case. Notice that $\sum_i l_i = 4$, which is larger than the number of variables in the problem, and hence unsatisfiable.

Scenario 3: $\mathbf{L} = \{\mathbf{2}, \mathbf{0}, \mathbf{1}\}, \mathbf{U} = \{\mathbf{2}, \mathbf{3}, \mathbf{3}\}$. This case is more subtle. Since there must be two variables taking the value 1, $X_1 = X_3 = 1$ is forced. But now is must also be the case that at least one variable takes the value 3. The only candidate for this is X_3, but the assignment $X_3 = 1$ has already been forced. Hence this case is unsatisfiable.

Scenario 4: $\mathbf{L}\{\mathbf{0}, \mathbf{0}, \mathbf{0}\}, \mathbf{U} = \{\mathbf{3}, \mathbf{0}, \mathbf{3}\}$. This case is slightly pathological, but important to realise. It is requested that no variable take a value of 2. Unfortunately, X_2 can only take the value 2. It is forbidden for a variable to have no viable, hence this is also unsatisfiable.

Hopefully this example has made clear the sort of reasoning employed by the GCC. □

2.5.3 Symmetry

In Chapter. 4 an encoding of a problem in SMT is presented which makes use of *symmetry breaking constraints*. This section very briefly motivates the need for such constraints and what they do.

To pictorially motivate what is meant by symmetry, consider the 8 symmetric solutions of the 8-queens problem depicted in Fig. 2.12. From this example, it is easier to see that by "symmetry" the meaning is an "operation which changes the positions of the pieces, but whose end-state obeys all the constraints if and only if the start-state does"[74]. The symmetries dealt with in this report are called *static symmetry breaking constraints*. They are static because the constraints are posted statically, along with the other constraints of the problem being modelled, as opposed to being dynamically generated by the solver and they *break* symmetry in the sense that for the constraint to be satisfied, only one solution in a class of symmetric solutions can be used. Another example of a problem with inherent symmetry is the AllDifferent constraint. In AllDifferent(X_1, X_2, X_3), where each X_i has a domain of $\{1, 2, 3\}$, there is no requirement

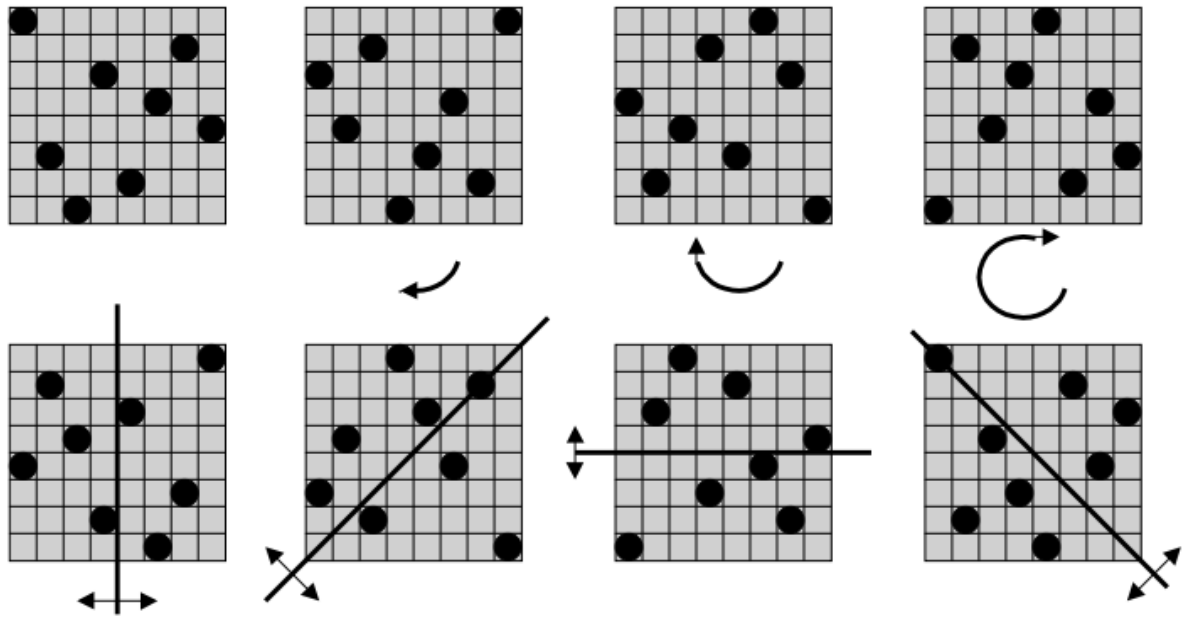

Figure 2.12: The 8-fold symmetry of the square. This is an example of one solution to the 8-queens problem. In this problem, 8 queens must be placed on a chessboard such that no two queens attack one another. The symmetry operators are the identity, rotation by 90°, 180° and 270° and reflection vertically, horizontally and diagonally. Figure taken from [75].

as to which value a variable must take. In terms of the search space, there are $3! = 6$ complete assignments that satisfy this constraint. The symmetries can be broken by adding a second constraint of the form $X_1 < X_2 < X_3$ which forces the assignment $X_1 = 1, X_2 = 2$ and $X_3 = 3$ to be made.

This presentation of symmetry was necessarily brief as symmetry considerations are not the focus of this report, but the concept is lightly used in Chapter. 4. [74] give a comprehensive introduction to the topic.

3 Decompositions

Decompositions, also referred to as *reformulations*[2] are translations of constraints into sets of simpler constraints that preserve logical equivalence at the very least, and typically in addition ensure that a certain level of consistency is achieved on the set of simpler constraints. The motivation for a decomposition is described in Chapter. 1. In this report decompositions that provide a less powerful consistency guarantee than built in propagators are studied; a natural question is therefore: when are decompositions *not appropriate*? It turns out that there are several constraints that are not *effectively decomposable*[76], meaning that their decompositions can not be bounded by a polynomial in size, and hence are not practical for large instances. There are therefore limits to the approach of reformulation. For the cases where reformulation is theoretically sensible, this report is concerned with the practical benefits of such reformulations in SMT.

In what follows, decompositions of the AllDifferent and GCC constraints will be presented. Each decomposition will come with a code name, introduced in bold font at the start of the decompositions' explanation. The code names are used in Chapter 4, where experiments are undertaken to compare the performance of the encodings introduced in this section.

Then, for each decomposition introduced, the concrete SAT encodings are presented in §3.3 and concrete SMT encodings are presented in §3.4 for the decompositions introduced.

3.1 AllDifferent Decompositions

BI. For AllDifferent there is at least one very obvious decomposition: decompose AllDifferent$(X_1, X_2, \ldots, X_n)$ into a clique of binary disequalities, namely,

$$\textsc{AllDifferent}(X_1, X_2, \ldots, X_n) \equiv \bigwedge_{i=1}^{n} X_i \neq X_j \quad j \in [i+1, n]. \tag{3.1}$$

DI. This encoding, suggested to me by my supervisor, is a variation of **BI** wherein pairwise inequalities are encoded as disjunctions of negated equalities over the intersection of two variables' domains. For example, consider three variables X_1, X_2 and X_3 with corresponding domains $\mathcal{D}(X_1) = \{1, 2\}, \mathcal{D}(X_2) = \{2\}$ and $\mathcal{D}(X_3) = \{1, 2, 3\}$. Now consider the pairwise intersection of the variables' domains. $\mathcal{D}(X_1) \cap \mathcal{D}(X_2) = \{2\}$, $\mathcal{D}(X_1) \cap \mathcal{D}(X_3) = \{1, 2\}$ and $\mathcal{D}(X_2) \cap \mathcal{D}(X_3) = \{2\}$. From this, we can assert the following, which is equivalent to the pairwise clique, but more informative:

$$\begin{aligned} &\neg(X_1 = 2) \vee \neg(X_2 = 2) \\ &\neg(X_1 = 1) \vee \neg(X_3 = 1) \\ &\neg(X_1 = 2) \vee \neg(X_3 = 2) \\ &\neg(X_2 = 2) \vee \neg(X_3 = 2) \end{aligned} \tag{3.2}$$

This encoding, while it may seem quite ridiculous at first sight, has some interesting solving statistics which will be explored in Chapter. 4.

HI(RC) ([14]). A far more sophisticated encoding than **BI** is now presented. The general idea is to decompose the logic of Algorithm. 2. In particular, this decomposition simulates the search for Hall intervals and the corresponding pruning of domains.

Let n be the number of variables X_i in our problem. Let d be the maximum value any variable can take and then for $1 \leq l \leq u \leq d$ and $u - l < n$, post the following constraints:

$$A_{ilu} = 1 \iff X_i \in [l, u] \quad \text{for } i \in [1, n] \tag{3.3}$$

$$\sum_{i=1}^{n} A_{ilu} \leq u - l + 1 \tag{3.4}$$

Less formally, Equation. 3.3 is a channelling constraint between the original variables X_i and the variables A_{ilu}, and Equation. 3.4 is counting how many variables take support inside each interval $[l_i, u_i]$. When the number of variables taking a support in $[l_i, u_i]$ becomes equal to $u - l + 1$, then the remaining unset variables A_{ilu} will be forced to zero, and the channelling constraints will take care of setting the corresponding X_i variables. Clearly this decomposition introduces $O(n^2 d)$ extra variables A_{ilu}. [14] prove that enforcing GAC on Equations. 3.3 3.4 enforces RC on the original AllDifferent constraint, which is quite remarkable given the simplicity of this decomposition.

HI(BC) ([14]) **HI(RC)** can be easily converted in a decomposition achieving the weaker BC guarantee. New variables B_{ij} are introduced for the truth of $X_i \leq j$. It was shown in [14] that enforcing BC on this version of the decomposition enforces BC on the original AllDifferent. Specifically, Equation. 3.3 is rewritten into,

$$B_{il} = 1 \iff X_i \leq l \tag{3.5}$$

$$A_{ilu} = 1 \iff B_{i(l-1)} = 0 \wedge B_{iu} = 1, \tag{3.6}$$

where Equation. 3.6 is defined for $l > 1$. When $l = 1$, define by convention that $B_{i(l-1)} = 0$ (that is, false.) In summary, **HI(BC)** is defined as,

$$B_{il} = 1 \iff X_i \leq l \tag{3.7}$$

$$A_{ilu} = 1 \iff B_{i(l-1)} = 0 \wedge B_{iu} = 1 \tag{3.8}$$

$$\sum_{i=1}^{n} A_{ilu} \leq u - l + 1. \tag{3.9}$$

3.2 GCC Decomposition

EHI(RC) ([14]). This is an extension of the **HI(RC)** decomposition which is inspired by another propagator algorithm for GCC acheiving bounds consistency[77]. The decomposition as described in [14] for GCC is a straight-forward generalisation of the **HI(RC)** decomposition, so its connection with the propagator algorithm was not presented in §2.5.2.

Let n be the number of variables X_i in our problem. Let d be the maximum value any variable can take. Now introduce $O(d^2)$ variables, N_{lu}, to represent the number of variables whose domains are subsets of the interval $[l, u]$. It is immediately seen that $\sum_{i=l}^{u} l_i \leq N_{lu} \leq \sum_{i=l}^{u} u_i$ and that $N_{1d} = n$. Now post the following constraints for $1 \leq l \leq u \leq d, 1 \leq k \leq u$,

$$A_{ilu} = 1 \iff X_i \in [l, u] \quad \text{for } i \in [1, n] \tag{3.10}$$

$$N_{lu} = \sum_{i=1}^{n} A_{ilu} \tag{3.11}$$

$$N_{1u} = N_{1k} + N_{(k+1)u}. \tag{3.12}$$

As before, the constraints in Equation. 3.10 are channelling constraints between the variables A_{ilu} and the original variables X_i. The constraint in Equation. 3.11 counts the number of variables taking support in each interval $[l, u]$ and the constraint in Equation. 3.12 provides an interaction constraint between the N_{ij} variables by expressing the count of each interval $[1, u]$ in terms of the counts for sub-intervals $[1 + k, u]$ for $k \in [0, u - 1]$. It was shown in [14] that enforcing GAC on Equation. 3.10 and BC on Equations. 3.11 3.12 achieves RC on the original GCC problem.

EHI(BC). As with the **HI(RC)** decomposition, Equation. 3.10 can be reformulated using Equations. 3.5 3.6 to achieve BC, then it was also shown in [14] that enforcing BC on this reformulation of **EHI** enforces BC on the original GCC problem. Specifically, the decomposition for **EHI(BC)** is,

$$B_{il} = 1 \iff X_i \leq l \tag{3.13}$$

$$A_{ilu} = 1 \iff B_{i(l-1)} = 0 \wedge B_{iu} = 1 \tag{3.14}$$

$$N_{lu} = \sum_{i=1}^{n} A_{ilu} \tag{3.15}$$

$$N_{1u} = N_{1k} + N_{(k+1)u}. \tag{3.16}$$

3.3 SAT Encodings

The encodings into SAT are indirectly achieved by first encoding in PB-SAT and then using MiniSat+ to translate the PB-SAT into SAT as described in §2.1.

Encoding BI into SAT. This encoding was not part of my experiment in SAT, since [14] demonstrated that the **HI(RC)** encoding performed more favourably than **BI** in SAT. Hence the obvious encoding of **BI** into SAT is not covered in this report.

Encoding HI(RC) into PB-SAT. Despite [14] presenting the RC decomposition of AllDifferent they did not explain how it can be encoded into SAT as stated. In particular, it was not clear to me how Equation. 3.3 could be encoded. In [14], they included a "hack" to increase propagation in which a direct encoding is employed with literals Z_{ij} for the truth of $X_i = j$ and clauses of the form $(A_{ilu} = 0) \implies (Z_{ij} = 0)$, $j \in [l, u]$. They claim (without proof) that this moves the consistency of the decomposition between BC and RC. Their experiments suggest this version of the decomposition reduces the number of decisions taken dramatically, and that the runtime is also much better than **BI** in SAT. Given these results, experiments in SAT for this decomposition are not conducted in this report. Experiments on this encoding in SMT will be conducted (see the next section).

Encoding HI(BC) into PB-SAT. First consider the constraints in Equations. 3.5 3.6. since $X_i \leq l \equiv \left(\sum_{k=1}^{l} x_{ik} \geq 1\right)$, then is should be clear that in PB-SAT, Equation. 3.5 can be

encoded as

$$(B_{il} = 1 \iff X_i \leq l) \equiv \left(B_{il} - \sum_{k=1}^{l} x_{ik} = 0 \right) \tag{3.17}$$

Example 11. It must be ensured that when a variable X_i takes a value k less than the lower bound of an interval, that $B_{ik} = 1$. This is just what Equation. 3.17 is doing. Consider a problem with three variables X_1, X_2 and X_3, each with the same domain $\mathcal{D}(X_i) = \{1, 2, 3\}$, and hence $d = 3$. Then the bound constraints for this example are,

$$\begin{aligned}
B_{11} - x_{11} &= 0 \\
B_{12} - x_{11} - x_{12} &= 0 \\
B_{13} - x_{11} - x_{12} - x_{13} &= 0 \\
B_{21} - x_{21} &= 0 \\
B_{22} - x_{21} - x_{22} &= 0 \\
B_{23} - x_{21} - x_{22} - x_{23} &= 0 \\
B_{31} - x_{31} &= 0 \\
B_{32} - x_{31} - x_{32} &= 0 \\
B_{33} - x_{31} - x_{32} - x_{33} &= 0
\end{aligned}$$

It is important to keep in mind that our variables can only take on values of zero or one. So in the first line above, if $x_{11} = 1$, then variable X_1 has taken the value one, and hence B_{11} must equal one. Similarly, in the third line above, if any of x_{11}, x_{12} or x_{13} are set true, then B_{13} must equal one. Also recall the constraint that $\sum_i x_{ki} = 1$, for any variable X_k. This means that only one of x_{11}, x_{12} or x_{13} can be equal to one. □

In a similar vein, Equation. 3.6 may be encoded as

$$\left(A_{ilu} = 1 \iff B_{i(l-1)} = 0 \wedge B_{iu} = 1 \right) \equiv \left(A_{ilu} + B_{i(l-1)} - B_{iu} = 0 \right). \tag{3.18}$$

This is only valid for $l > 1$. If $l = 1$, then let $B_{i(l-1)} = 0$.

Example 12. Encoding the interval constraints for the above three-variable example, I get:

$$\begin{aligned}
A_{111} - B_{11} &= 0 \\
A_{112} - B_{12} &= 0 \\
A_{113} - B_{13} &= 0 \\
A_{122} + B_{11} - B_{12} &= 0 \\
A_{123} + B_{11} - B_{13} &= 0 \\
A_{133} + B_{12} - B_{13} &= 0 \\
A_{211} - B_{21} &= 0 \\
A_{212} - B_{22} &= 0 \\
A_{213} - B_{23} &= 0 \\
A_{222} + B_{21} - B_{22} &= 0 \\
A_{223} + B_{21} - B_{23} &= 0 \\
A_{233} + B_{22} - B_{23} &= 0 \\
A_{311} - B_{31} &= 0 \\
A_{312} - B_{32} &= 0 \\
A_{313} - B_{33} &= 0 \\
A_{322} + B_{31} - B_{32} &= 0 \\
A_{323} + B_{31} - B_{33} &= 0 \\
A_{333} + B_{32} - B_{33} &= 0
\end{aligned}$$

The first line is stating that for the variable A_{111} to be true (that is, the numeral 1,) there must be variable taking a value less than or equal to 1 (B_{11}.) Since 1 is the minimum domain value, the "less than or equal to" is actually strict equality. The fifth line is stating that for the variable A_{123} to be true, some variable must take a value less than or equal to 2 (B_{12}) but not less than 1 (B_{11}). □

Finally, the constraint in Equation. 3.4 can be directly posted into PB-SAT.

If it is the case that some of our variables do not have a domain of $\{1, 2, \ldots, d\}$, that is, a domain spanning the largest space in our model, then we can greatly improve unit propagation by further specifying which values our variables *can not* take. For example, assume that a variable X_1 has domain $\mathcal{D}(X_1) = \{1, 2\}$, then we can assert that $x_{13} = 0$.

Encoding EHI(RC) and EHI(BC) into SAT. Although [14] introduced these decompositions for the GCC constraint, they did not run any experiments in their paper testing the encoding. This is most likely because the encoding is not suitable for PB-SAT, which was the formalism of choice in their paper. To see why, note that the N_{ij} variables in the **EHI** decomposition are not 0/1-variables in general. However, such a generalisation is not a problem for SMT's linear-integer arithmetic theory. Therefore, a SAT encoding of this decomposition will not be considered in this report.

3.4 SMT Encodings

In §3.3 it was explained that the **EHI(RC)** and **EHI(BC)** decompositions are not suitable for SAT. They are suitable for SMT, and in this section SMT encodings used to experiment with

varying the Hall interval sizes for problems with GCC constraints will be described. Also shown in this section is how the **BI**, **DI**, **HI(RC)** and **HI(BC)** decompositions are encoded into SMT.

Encoding BI into SMT. There is nothing fancy about this encoding, a list of (`assert` (`not` (= `X_i X_j`))) are posted in SMT-LIB2.

Encoding DI into SMT. This is also an unceremonious encoding, A direct transliteration into SMT-LIB2.

Encoding HI(RC) into SMT. HI(RC) uses pseudo-boolean variables, that is, variables whose value is zero or one. In SMT, the closest data type to this is the integers. I expected that explicitly asserting the valid domain members, such as $x_{ij} = 0 \vee x_{ij} = 1$ would be a benefit to the LIA theory, but it turns out that such restrictions do not help this theory at all. The SMT translation of [14] using a direct encoding performs very poorly compared to SAT.

Instead of using variables x_{ij} for each value in each variables' domain (i.e., the direct encoding), the LIA theory can be exploited more effectively if instead variables x_k for $k \in [1, d]$ are considered. The domain constraints then become $x_k \geq 1 \wedge x_k \leq d$. It is this encoding that will be used in the SMT experiment.

Now the **HI(RC)** decomposition must be encoded into SMT. Firstly Equation. 3.3 must be decomposed into SMT. There are two ways to achieve this in SMT-LIB2. One is by rewriting the biconditional,

$$\begin{aligned}&(A_{ilu} = 0 \vee (X_i \geq l \wedge X_i \leq u)) \wedge \\ &(A_{ilu} = 1 \vee (X_i < l \vee X_i > u))\end{aligned} \tag{3.19}$$

and decomposing Equation. 3.19 directly into SMT-LIB. The alternative method is to use the `ite` (if-then-else) procedure exposed in Z3. Specifically, I post

$$(\texttt{assert}\,(= A_{ilu}\ (\texttt{ite}\,(\texttt{and}\,(>= A_{ilu}\,l\,)\,(<= A_{ilu}\,u\ ))\,\texttt{1}\,\texttt{0}))) \tag{3.20}$$

in SMT-LIB2. Both of these decompositions perform similarly. Equation. 3.20 was selected since it is slightly easier to target programmatically.

As with the SAT encoding, Equation. 3.4 can be decomposed directly into SMT.

Encoding EHI(RC) and EHI(BC) into SMT. The only new bits in these decompositions are the constraints in Equations. 3.11 3.12, both of which are trivially encoded in SMT-LIB2.

4 Comparison of Encodings

In this chapter the performance of the decompositions presented in Chapter. 3 for the AllDifferent and GCC constraints are examined.

The questions the experiments in this report are designed to answer are,

1. Is it practically feasible to use decompositions with consistency as strong as BC and RC in SMT?

2. When is it appropriate to use a BC decomposition versus a RC decomposition in SMT?

3. How does varying the Hall interval size effect the conflicts, decisions, propagations and run-time in SAT & SMT?

4. Are decompositions which theoretically espouse improvements really better than simpler local encodings in SMT?

To try and answer these questions, two problems to apply decompositions of the AllDifferent and GCC constraints have been selected,

- **Sudoku**. Sudoku is a natural candidate for testing the AllDifferent constraint. The Sudoku problem can be solved using AllDifferent alone (see §4.1). The limitation of this choice is that the effects of different decompositions of constraints interacting with one another are not studied.

- **Covering Arrays**. The Covering Array Problem will be described in §4.2, it was chosen because in addition to GCCs playing a vital role in the Covering Array Problem, several other constraints are used and their interaction can be studied, at least for this one problem.

For both problems the Hall size is varied over the appropriate range and data are collected to find out what effect the larger Hall interval sizes have. In addition, both RC and BC encodings of the AllDifferent and GCC encodings are deployed to see how they compare to each other for these problems.

The experiments were conducted on a computer with two Intel Xeon E5520 processors, both clocked at 2.27GHz with 8 MiB cache sizes. This computer had 16GiB of RAM. The operating system was Ubuntu 12.04 LTS 64-bit edition. It should be noted that this computer is a shared resource of the Computer Science undergraduates, and hence experiments were occasionally conducted under heavy load from the many users. Therefore, wall clock time is not measured, rather the "user" time is measured using the GNU/Linux `time` command. User time measures the total number of CPU-seconds that the process used directly. I used MiniSat+ version 1.0, and Z3 version 4.3.2.

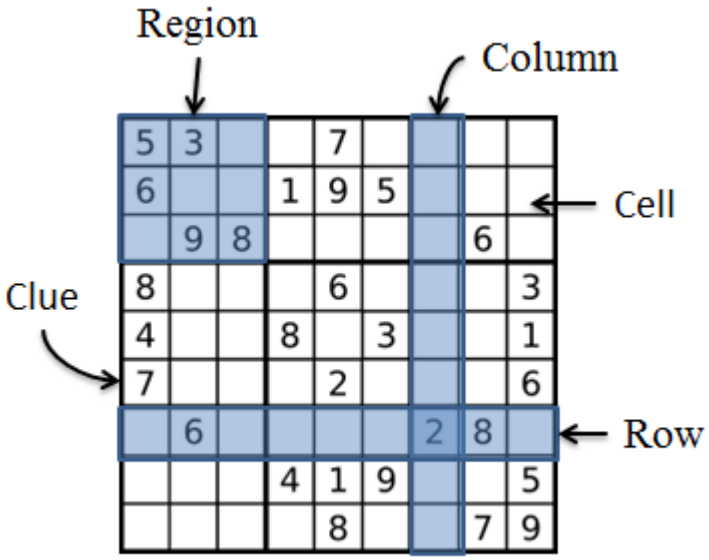

Figure 4.1: Overview of Sudoku terminology.

4.1 AllDifferent constraint

The requisite Sudoku terminology is presented in Fig. 4.1. For an $N \times N$ Sudoku problem, sometimes referred to as an order-$\sqrt{N}$ Sudoku problem, we have N rows, columns and regions each containing N cells. The values given to cells in the problem are called clues. Each cell in each of these rows, columns and regions must take on a distinct value from $[1, N]$, that is, they must be AllDifferent.

My variables are X_{ij} representing the cell at row i and column j on the Sudoku board. The domain of each variable is $\mathcal{D}(X_{ij}) = [1, N]$ and the constraints are AllDifferent's over the variables in each row, column and region.

A representative sample of "hard" Sudoku problems was required. Preliminary experiments suggested that benchmarking over an adequately (> 20) representative sample of order-4 Sudoku problems would require too much time in SMT. It was feasible however in SAT. So in addition to the order-4 problems, a large sample of order-3 Sudoku problems were curated for comparing SAT and SMT. The order-3 Sudoku instances were curated from [78], who compiled a list of minimum instances with 18 clues. A minimum instance is one in which no clues can be removed leaving the puzzle with only one solution. The order-4 Sudoku instances were curated from [79]. The Sudoku node in Fig. 4.2 denotes these files.

To adequately test different encodings of the AllDifferent constraint, a framework was designed to decompose Sudoku problems into both SAT and SMT. An overview of the entire framework is shown schematically in Fig. 4.2. Two translator programs were written that take as input a Sudoku file, and produce as output a file in the SMT-LIB2 language and a file in the PBO file format language respectively. Z3 directly supports input in the SMT-LIB2 language. The PBO file needs to be translated into the DIMACS format before it can be fed to Z3's SAT solver. This translation is performed by MiniSat+.

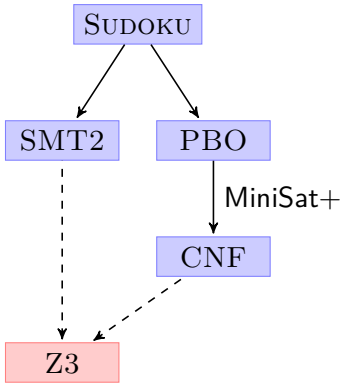

Figure 4.2: Overview of how the instances were generated. The blue boxes represent file format in the tool-chain, and the red boxes represent solver programs. The source node of a dashed line represents the input to the target node. Notice that I am using an off-the-shelf translator to minimise the PBO file format into CNF. A brief introduction to this tool is given in §2.1. The solid lines represent translators that I wrote for this project. The source code is available at [80]

4.1.1 SAT Experiment

To explore the effect of the Hall interval size in SAT, experiments were run on both order-3 and order-4 boards for Sudoku. The results of the order-3 Sudoku experiments are shown in Figs. 4.3 (conflicts),4.4 (decisions),4.5 (propagations),4.6 (run-time). The results of the order-4 Sudoku experiments are shown in Figs. 4.7 (propagations),4.8 (time). Not shown for the order-4 Sudoku experiments are the conflict and decision statistics. The reason is nearly all boards produced zero conflicts and one decision (with a few small exceptions.) This is a remarkable property of the encoding using MINISAT+'s minimisation procedure: for the order-4 board, most were solved with unit propagation alone!

What is clear from these experiments in SAT is that the detection of larger Hall intervals provides far more propagation, and correspondingly fewer decisions being made. This was already demonstrated in [14] for the decomposition between BC and RC (see §3.3), but the experiments confirm the trend for **HI(BC)**. This does seem to correlate with a slightly increased run-time, as the cost of propagation is noticeable but scales well in SAT. Is is also clear that smaller Hall intervals produce more conflicts, which in turn reduces run-time. I conjecture this is due to the CDCL SAT implementation in Z3.

4.1.2 SMT Experiment

Only the **HI(BC)** decomposition was considered in my SAT experiment. In these SMT experiments, I will be examining both **HI(BC)** and **HI(RC)**, the latter of which has not yet been experimented with to the best of my knowledge. To answer the question about how these more sophisticated decompositions compare and interact with simpler, more local encodings, I will also be experimenting with the **BI** and **DI** both in isolation of and in combination with **HI(BC)** and **HI(RC)**.

The results of the **BI** experiment are shown in Table. 4.1 and the results of the **DI** experiment are shown in Table. 4.2. It is interesting that despite the **BI** decomposition entailing a large number of propagations, **DI** still pips **BI** on run-time because the cost of propagation in SMT has slightly dominated for these order-3 Sudoku problems. Another factor in favour of **DI** is its

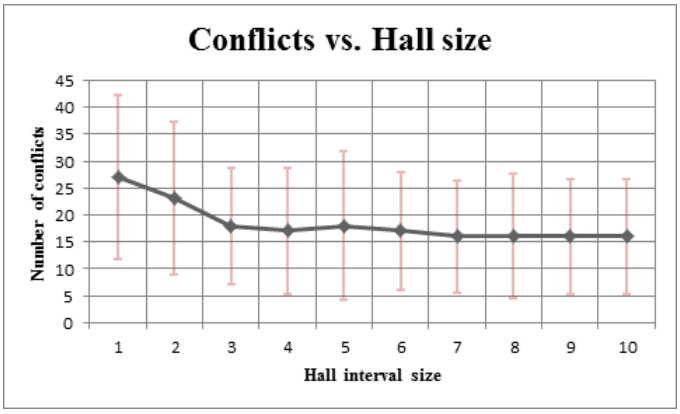

Figure 4.3: Conflicts recorded over the 87 order-3 Sudoku instances. Detecting smaller Hall intervals appears to generate more conflicts, which when taken into account with the run-time (see Fig. 4.6), helps the solver. I would conjecture this is due the conflict-driven clause learning. There is also a noticeable trend that increasing the Hall interval size reduces the number of conflicts; this can be neatly contrasted with the extra propagation larger Hall intervals entail (see Fig. 4.5.)

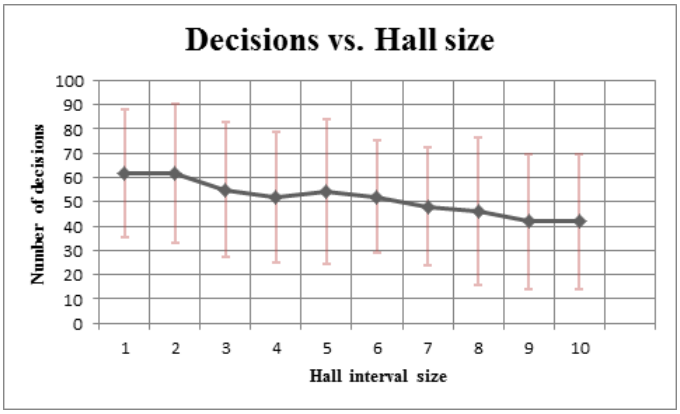

Figure 4.4: Decisions recorded over the 87 order-3 Sudoku instances. Decisions almost inversely correlate with propagations, as I would expect. Larger Hall intervals provide more information to the solver and hence require it to make less decisions of its own. Clearly decisions are not that expensive given the run-time results in Fig. 4.6.

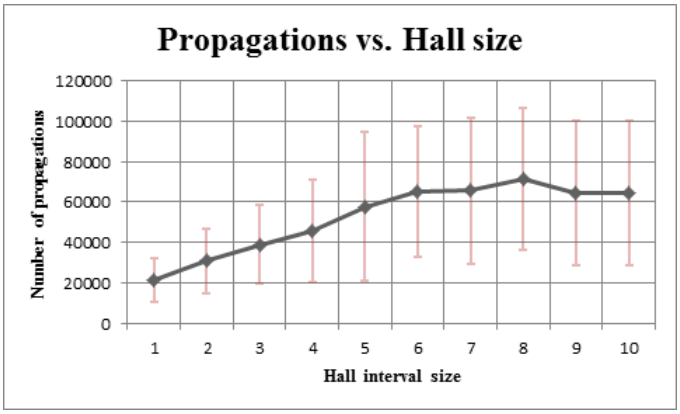

Figure 4.5: Propagations recorded over the 87 order-3 Sudoku instances. There is a clear trend of larger Hall intervals entailing more propagation, as was noticed in [14]. There is a noticeable, almost negligible, increase in run-time (see Fig. 4.6); propagation in SAT can be seen to scale very well with time taken.

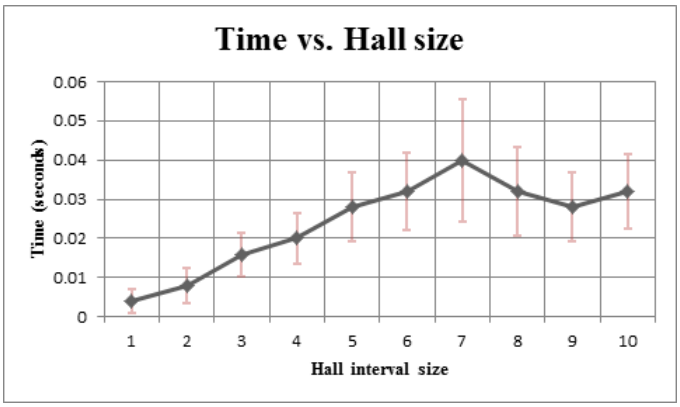

Figure 4.6: Time taken recorded over the 87 order-3 Sudoku instances. Generally, larger Hall intervals require more time. There is however a strange ridge at Hall sizes of 7. I can not explain this behaviour, since the other statistics I collected demonstrated nothing unusual about the Hall size of 7.

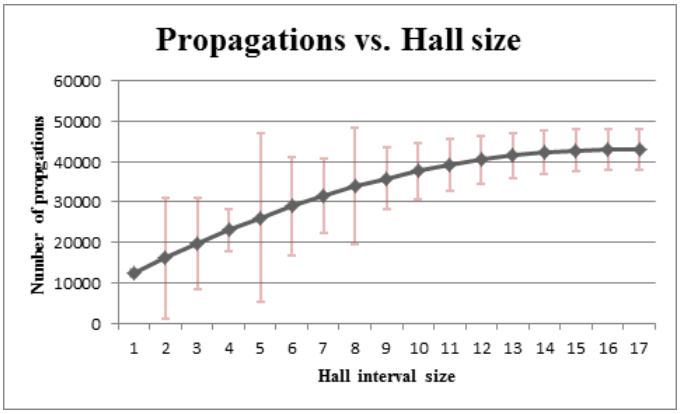

Figure 4.7: Propagations recorded over 25 order-4 Sudoku instances. It is more easily noticeable how larger Hall intervals yield far more propagations. In fact, for several of the instances unit propagation was sufficient to satisfy the instance alone. It is also again evident that extra propagation does not effect runtime in a significant manner.

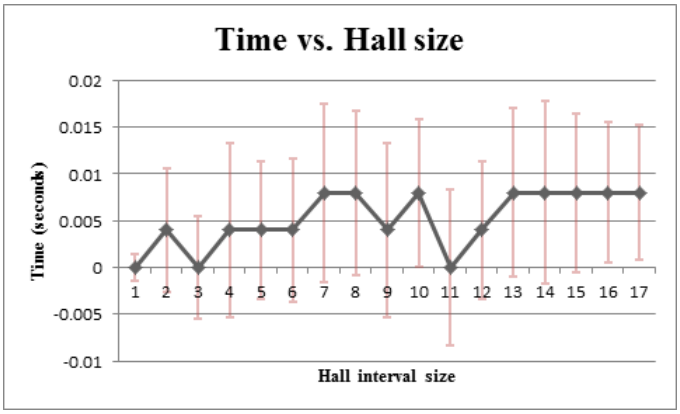

Figure 4.8: Time taken recorded over the 25 order-4 Sudoku instances. The differences in solve time are barely noticeable, despite enormous amount of extra propagations as the Hall sizes increase.

Statistic	Median	Standard Deviation
Conflicts	101	0.3
Decisions	1992	318.84
Propagations	13294	1995
Time (seconds)	0.505	0.1862

Table 4.1: Experiment results over the 87 order-3 Sudoku instances using the **BI** decomposition.

Statistic	Median	Standard Deviation
Conflicts	101	0.6
Decisions	3479	598.61
Propagations	1518.5	285.03
Time (seconds)	0.225	0.059

Table 4.2: Experiment results over the 87 order-3 Sudoku instances using the **DI** decomposition.

heavy use of logical operators. It was demonstrated in [12] that introducing clauses instead of expressing disjunctions arithmetically benefits SMT.

The results of the combination experiments for BC are shown in Fig. 4.9 and those for the RC experiments are shown in Fig. 4.10. The combination of **BI** and **HI(BC)** is denoted **HI(BC)+BI**, and similarly for the other 3 combinations. The **A** decomposition is combination of **HI(BC)**, **DI** and **BI**, and similarly for **HI(RC)**.

I was shocked by the order of magnitude differences in decisions and propagations between SAT and SMT. In SAT, with the **HI(BC)** decomposition, the number of decisions was on the order of 50; in SMT the number of decisions with the **HI(BC)** decomposition was on the order of 5000. Similarly, in SAT the number of propagations was on the order of 30,000; in SMT the number of propagations was on the order of 3000. It is not clear what has caused this order of magnitude switch in statistics with the two technologies.

My conclusions from these experiments are as follows,

- Larger Hall intervals invariably imply longer run-times. The trend is exacerbated when using a more consistent decomposition.
- Combination decompositions appear to have pleasant linearity properties, in that they subtractively decide and additively propagate.
- Combination decompositions seem to dampen the cost of propagation in more consistent decompositions, for reasons that are not clear to me. Perhaps there are other factors I have not measured in these experiments causing this observation.
- The cost of propagation in SMT appears to be comparatively high compared to the cost in SAT. When choosing a decomposition, you must be careful to avoid the cost of propagation dominating in the search for a solution to your problem.
- I conjecture that the choice of when to use a more consistent decomposition depends on some measure of "problem difficulty".
- Most disappointingly (or most encouraging, depending on your persuasion,) with the caveat that my experiments were run on a set of fairly small Sudoku instances, these

more sophisticated decompositions do not perform competitively with the simpler local decompositions of **DI** and **BI**.

4.2 Global Cardinality Constraints

In this section experiments are conducted with the **EHI(BC)** and **EHI(RC)** decompositions in SMT. These decompositions will be compared by application to the Covering Array problem.

The questions (further) addressed by these experiments are what effects larger Hall interval sizes have in SMT on a much harder problem than Sudoku, and when is it appropriate to use a more consistent decomposition in SMT.

Covering Arrays. A covering array $CA(t, k, g)$ of size b and strength t, is a $k \times b$ array $A = (a_{i,j})$ over $Z_g = \{0, 1, 2, \ldots, g-1$[81], where g is the alphabet size. $CA(t, k, g)$ has the property that for any t distinct rows $1 \leq r_1 \leq r_2 \leq \ldots \leq r_t \leq k$, and any member $(x_1, x_2, \ldots, x_t)$ of Z_g^t there is at least one column c such that $x_i = a_{r_i,c}$ for all $1 \leq i \leq t$. In general finding optimal covering arrays is NP-complete[81].

Example 13. Consider the covering array in Fig. 4.11. This example is $CA(3, 5, 2)$ and $b = 10$. In fact, to simplify the task in this evaluation, the strength *will always be* $t = 3$. Two coverings are shaded in this figure. For each cover, you can run your finger down the rows and count off the g-ary numbers from 0 to g^3 (those in Z_g^t.) There are $\binom{5}{3} = 10$ such coverings in this array of 10 rows, only two are shown in Fig. 4.11.

In [82], a CP model for generating covering arrays was presented. For my experiments in this section, I have implemented the constraints described in [82] in MiniZinc[1]. Inspired from my MiniZinc model, I wrote another translator tool[2] which would take parameters (k, g, b) (recall that t is fixed at 3 in these experiments) and generate SMT-LIB2 code to decide if the problem was satisfiable. In this section I will describe the CP model used for solving the Covering Array problem, and in §4.2.1 I will describe the SMT translation for this model into SMT-LIB2.

The model in [82] uses two matrices of decision variables,

- A test vector matrix I will denote by $\boldsymbol{X}$ $(b \times k)$ where X_{ij} is the value of the variable in row i and column j, with the meaning that if $X_{ij} = m$, then the value of parameter j in test vector i is m. The domain of m is $[0, g-1]$.
- A channelling matrix I will denote by $\boldsymbol{Y}$ $(b \times \binom{k}{3})$. The channelling matrix is cleverly designed to express the coverage constraints on $\boldsymbol{X}$, those being that every subset of 3 parameters must be combined in all possible g^3 ways.

The construction of the channelling matrix $\boldsymbol{Y}$ is rather subtle. Using the setup in Fig. 4.11 as a running example, there are 10 triples of parameters representing the columns of $\boldsymbol{Y}$. Hence each variable Y_{ij} represents a tuple of 3 variables in $\boldsymbol{X}$. For example, $Y_{rt} = 5$ with $t = \langle 1, 3, 5 \rangle$ implies that $X_{r1} = 1$, $X_{r3} = 0$ and $X_{r5} = 1$ since $5_{10} = 101_2$. The domain of a value Y_{ij} is $[0, 2^3 - 1] = [0, 7]$.

I have defined the variables used and their domains, now for the constrains over these variables. The details of why these constraints are posted are beyond the scope of this report but can be looked up in [82],

[1] Source code available from https://github.com/switicus/constraints/blob/master/cover_test.mzn

[2] Source code available from https://github.com/switicus/smt_research/blob/master/ex/cover_test/cover_test.py

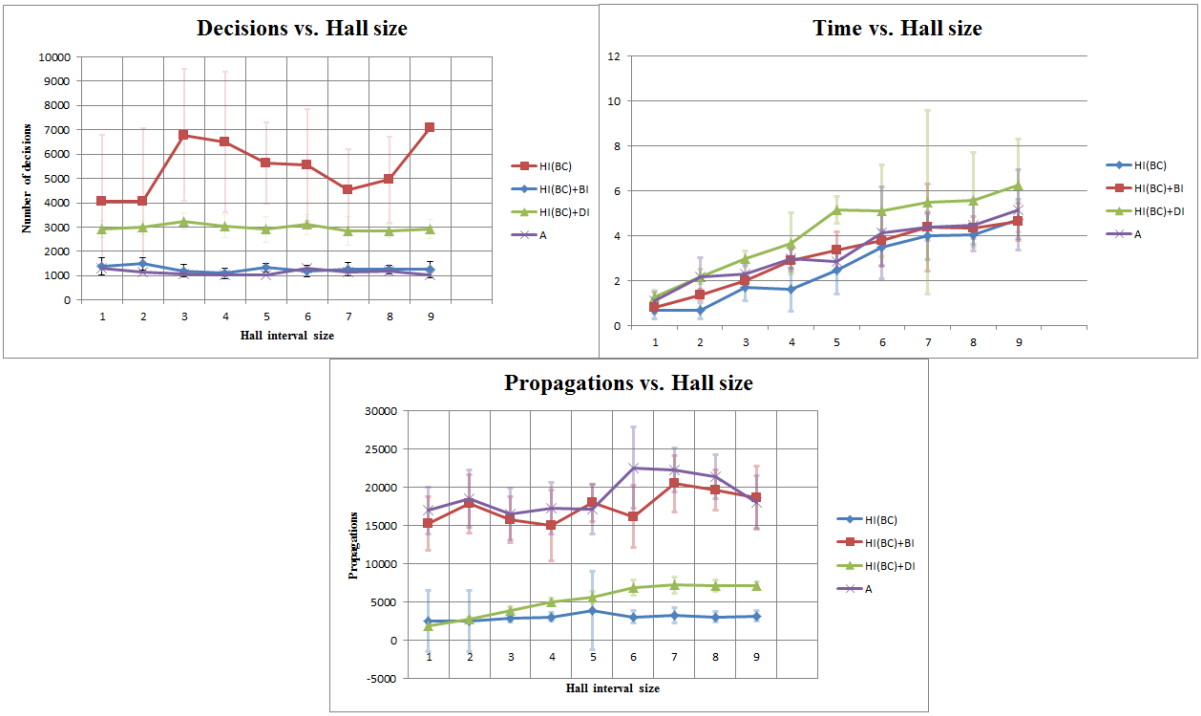

Figure 4.9: **HI(BC)** decomposition applied to the 87 order-3 Sudoku instances. Conflicts are not displayed since all decompositions produced around 101 conflicts. Compared to the SAT results on these Sudoku instance, it is amazing to me that there are orders of magnitude difference between the number of decisions and the amount of propagation performed by the two technologies. Clearly SAT has exhibited better run-time for this problem. It is not obvious from these statistics what exactly is causing the one decomposition to exhibit a better run-time than another. **HI(BC)** has performed most favourably on its own in this experiment, exhibiting the most decisions taken and the fewest propagations. It is also interesting that combination decompositions subtractively combine their decompositions, yet additively combine their propagations. Furthermore, despite **DI** performing slightly more favourably that **BI** in isolation, clearly **BI** is a better team-player than **DI** in this experiment.

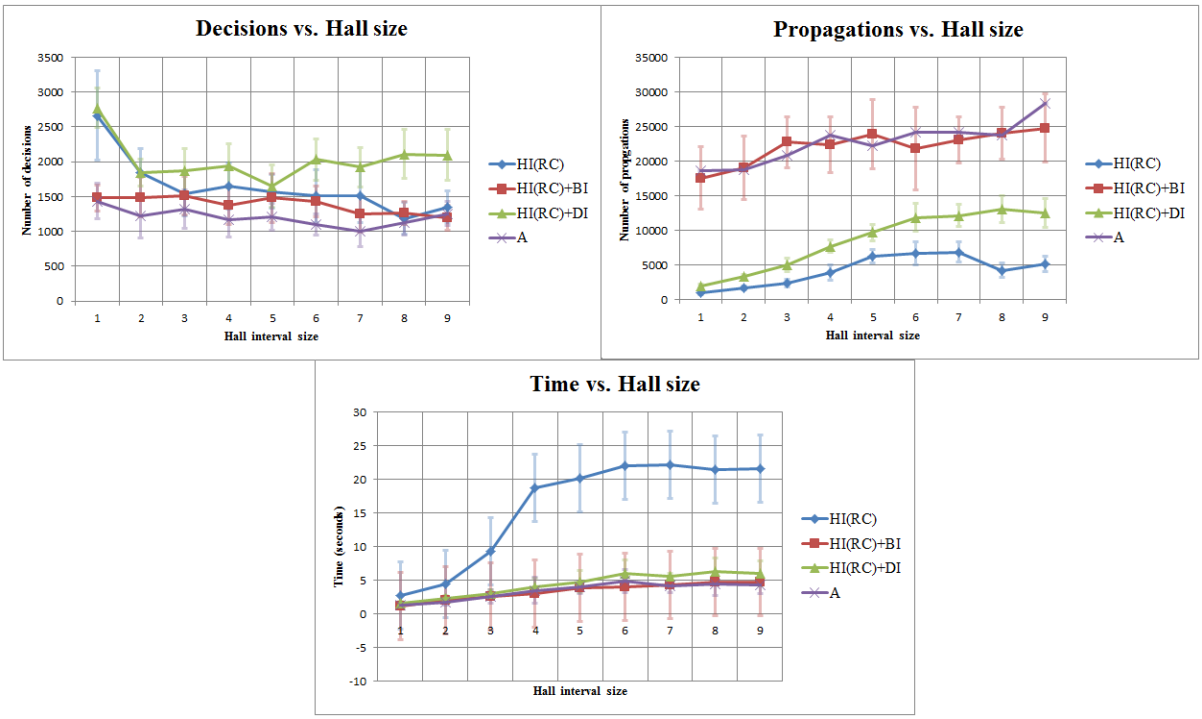

Figure 4.10: **HI(RC)** decomposition applied to the 87 order-3 Sudoku instances. Conflicts are not displayed since all decompositions produced around 101 conflicts. An interesting observation here is how the slightly increased propagation has killed the run-time for the **HI(RC)** decomposition in isolation. Apart from this drastic difference between **HI(RC)** and **HI(BC)**, many of the characteristics exhibited in Fig. 4.9 are exhibited here. I will conjecture that these Sudoku instances are just not "hard" enough for RC's strength to be of use.

0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 4.11: Example of a $CA(3, 5, 2)$ with $b = 10$. The shaded region show two coverings. The remaining $\binom{5}{3} - 2 = 8$ coverings are not shown.

- The channelling constraint linking the variables between $\boldsymbol{Y}$ and $\boldsymbol{X}$. Specifically,

$$Y_{rc} = g^2 X_{rt_1} + g X_{rt_2} + X_{rt_3}, \tag{4.1}$$

  where t_i denotes the i^{th} element of the tuple t.

- The coverage constraint. For each column, c, of the channelling matrix, I post,

$$\text{GCC}(c, [0, 1, \ldots, 7], 1, b - g^t + 1). \tag{4.2}$$

This model works correctly, but it works slowly due to the symmetry inherent in this problem. I have added symmetry breaking constraints to my MiniZinc model and I have translated these symmetry breaking constraints into SMT. In doing so, I now have several decompositions of constraints working together.

- An obvious constraint we can post that will cut out useless search is that of enforcing the first row of $\boldsymbol{X}$ to be full of zeros, i.e., $X_{1c} = 0$ for $c \in [1, 2, \ldots, b]$.

- There are in fact symmetries in the test vectors of the array $\boldsymbol{X}$. In particular, we can swap values in any column with one another and still have a valid solution. [82] suggested letting n_{is} denote the number of occurrences of values s in column i for $1 \leq i \leq k$ and $0 \leq s \leq g - 1$ and then imposing an order $n_{i0} \leq n_{i1} \leq \ldots \leq n_{i(g-1)}$. I encoded this using the Exactly global constraint[9] (called Count-Eq in MiniZinc) which in $\text{Exactly}(N, [X_1, X_2, \ldots, X_n], V)$ states that the value V must occur exactly N times in the X_i. Explicitly, assuming that we having counter variables n_i for $i \in [1, 2, \ldots, g]$ with constraints such that $n_i \leq n_{i+1}$ for $i \in [1, 2, \ldots, g - 1]$ then for each column c of $\boldsymbol{X}$ and for each $s \in [0, 1, \ldots, g - 1]$ I post $\text{Exactly}(n_{s+1}, c, s)$.

- A final source of symmetry discussed in [82] is that permuting the rows and/or columns of $\boldsymbol{X}$ in a given solution yields another valid solution, so we would like to break this symmetry as well. A common way of achieving this is to use the global lexicographic ordering constraints introduced in [51] on all pairs of rows and columns in $\boldsymbol{X}$. When two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of n variables $\langle x_0, x_1, \ldots, x_{n-1} \rangle$ and $\langle y_0, y_1, \ldots, y_{n-1} \rangle$ are lexicographically ordered (denoted by $\boldsymbol{x} \leq_{\text{lex}} \boldsymbol{y}$) it ensures that: $x_0 \leq y_0$; $x_1 \leq y_1$ when $x_0 = y_0$; $x_2 \leq y_2$ when $x_0 = y_0$ and $x_1 = y_1$; $\ldots$; $x_{n-1} \leq y_{n-1}$ when $x_0 = y_0$, $x_1 = y_1, \ldots$, and $x_{n-2} = y_{n-2}$[51]. MiniZinc by default posts the lexicographic orderings between all pairs of *adjacent* rows and columns, which, interestingly, for $\boldsymbol{X}$ achieves the same propagation guarantees as posting them for *all* pairs of rows and columns since the variables of $\boldsymbol{X}$ have 0/1-values[51], however in general this decomposition will hinder propagation.

4.2.1 SMT Experiment

In this section I will describe the encoding in SMT of the CP model presented in §4.2. Experiments shall be conducted on various instances of the Cover Array problem, and conclusions from these experiments will be drawn.

The variables of both the $\boldsymbol{X}$ and $\boldsymbol{Y}$ matrices are straightforwardly encoded as integers with domains $[0, g - 1]$. Similarly, Equation. 4.1 is trivial to express in SMT when using the linear integer arithmetic theory.

To encode the test-vector symmetry breaking constraints I need a decomposition of $\text{Exactly}(N, [X_1, X_2, \ldots, X_n], V)$. To encode this in SMT, I introduce a `count` variable which

denotes the number of variables X_i taking on value V. This is achieved by $\texttt{count} = \sum_{i=1}^{n}[X_i = V]$ where the Iverson bracket is again implemented using the handy `ite` procedure. Given the `count`, the EXACTLY constraint can be completed by asserting that $N = \texttt{count}$.

The final constraint I needed to encode for this problem was the lexicographic constraint over a matrix. I decomposed the lexicographic ordering constraints using an alternative arithmetic encoding given in [51]. In particular $\boldsymbol{x} \leq_{\text{lex}} \boldsymbol{y}$ is encoded as

$$g^{n-1}x_0 + g^{n-2}x_1 + \ldots + g^0 x_{n-1} \leq g^{n-1}y_0 + g^{n-2}y_1 + \ldots + g^0 y_{n-1}. \tag{4.3}$$

It is mentioned in [51] that as g and the length of the vectors n grows, this decomposition may not be suitable since some systems have machine-limited integer sizes. Z3 does not suffer from this problem as it utilises "bignum" packages on the platforms it supports. This constraint is posted for all pairs of adjacent rows and columns. MINIZINC uses a different decomposition for the lexicographic constraint, also described in [51],

$$\begin{aligned}\{x_0 \leq y_0,\, x_0 = y_0 \implies x_1 \leq y_1,\, x_0 = y_0 \vee x_1 = y_1 \implies x_2 \leq y_2,\, \ldots,\\ x_0 = y_0 \wedge x_1 = y_1 \wedge \ldots \wedge x_{n-2} = y_{n-2} \implies x_{n-1} \leq y_{n-1}\}\end{aligned} \tag{4.4}$$

Three instances of the Covering Array problem were selected for my SMT experiment. CA($g = 2, k = 4, b = 10$), CA($g = 2, k = 5, b = 12$), and CA($g = 2, k = 7, b = 11$). They are in increasing order of difficulty, with the final one being an unsatisfiable instance.

The results of the CA($g = 2, k = 4, b = 10$) experiment are shown in Fig. 4.12, the results of the CA($g = 2, k = 5, b = 12$) experiment are shown in Fig. 4.13 and the results of the CA($g = 2, k = 7, b = 11$) experiment are shown in Fig. 4.14. Each problem was solved 10 times, and the median value of these runs, along with error bars of one standard deviation are presented in the timing results.

My conclusions from these experiments are as follows,

- Detecting larger Hall intervals again correlates with longer run-times.
- I conjecture that the conflicts induced by a decomposition can seriously hamper run-time, indirectly through extra propagation from the learnt clauses.
- The conjecture that for more consistent decompositions to be useful, they need to be applied to "hard" problems was supported. Unfortunately, experiments on even harder Covering Array problems, particularly those for which $g > 2$, where **EHI(RC)** might perform much better than **EHI(BC)**, took longer than the 2 minute time-out I enforced.
- It was hard to pry out anything meaningful about how the various constraint decompositions in this problem interacted with one another. I suspect the conflicts inducing large amounts of propagation may be partly explained by the internals of the global constraint decompositions interacting with one another, but this is hard to demonstrate experimentally, or to prove theoretically.
- While it is not fair to make a comparison of my SMT encoding's performance to the performance of my CP model using the MINIZINC solver, it was non-the-less of note how large the performance gap is due to the sophistication of CP technology.

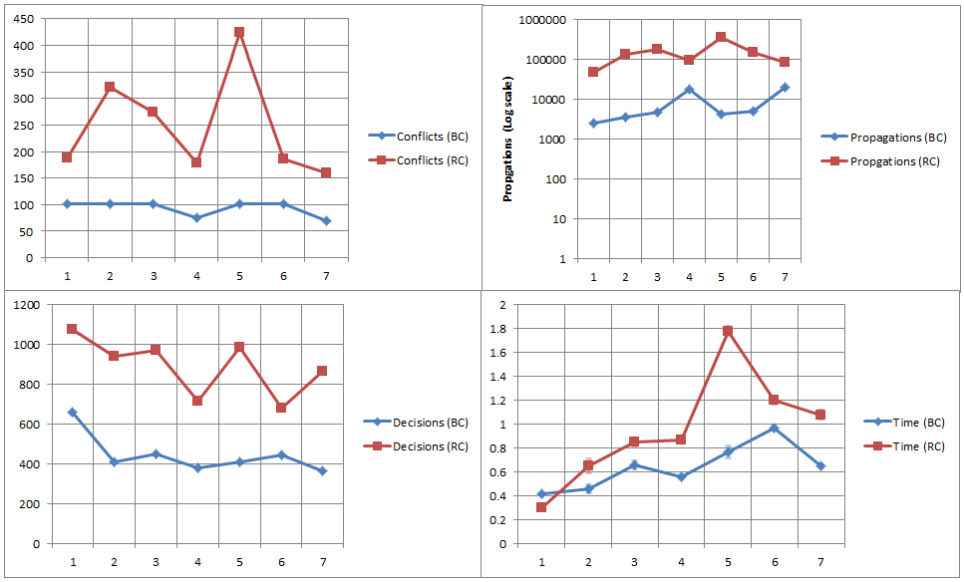

Figure 4.12: **EHI(BC)** and **EHI(RC)** decompositions of GCC applied to CA($g = 2, k = 4, b = 10$). The x-axis measures Hall interval size. Time is measured in seconds. The **EHI(RC)** decomposition has more of everything, and a highly curious peak in each graph for Hall intervals of size 5. It must be the case that size 5 Hall intervals interact uniquely on this particular problem. For this problem, the difference in entailed propagation is very large, notice that the propagation y-axis is a log scale, despite the extra propagation, the run-time is scaling more favourably for this Covering Array problem that it did for Sudoku. This being a "harder" problem supports my original conjecture that the more consistent decomposition starts to pay for itself as the problem gets harder. I further conjecture that the better scaling run-time has a lot to do with the conflicts encountered with **EHI(RC)** because CDCL is so effective.

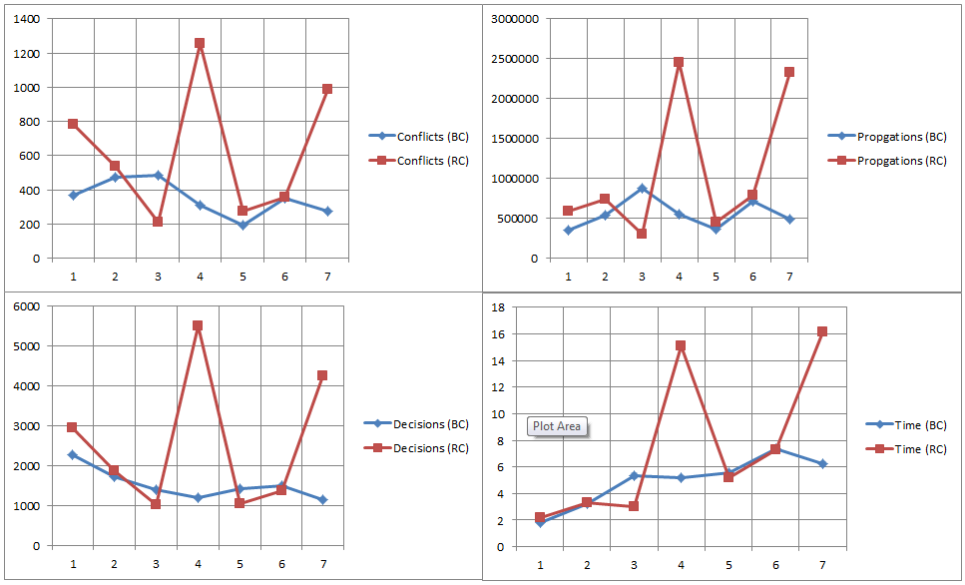

Figure 4.13: **EHI(BC)** and **EHI(RC)** decompositions of GCC applied to CA($g = 2, k = 5, b = 12$). The x-axis measures Hall interval size. Time is measured in seconds. There are two highly curious peaks for this problem, one at 4 and one at 7. There seems to be no rhyme or reason to where one of these mountainous Hall sizes will present themselves in a given problem. The conflicts generated at Hall sizes of 4 and 7 must have entailed many learnt clauses that cause the extra propagation and correspondingly increased run-time. The jumps for this problem are far more vicious than those of Fig. 4.12, which I conjecture is why, despite this being a harder instance of the Covering Array problem, the RC decomposition performs much worse than the BC decomposition than that observed in the easier problem shown in Fig. 4.12.

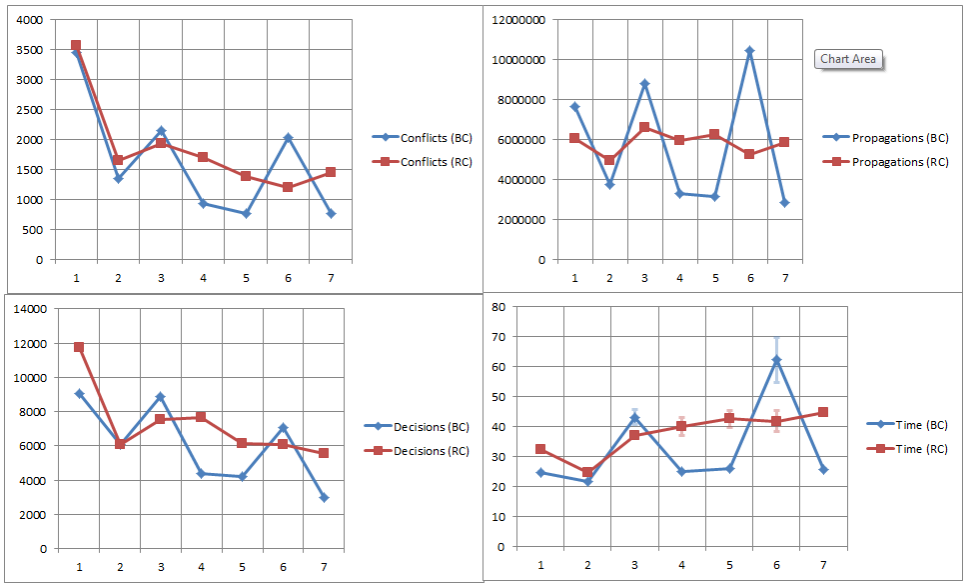

Figure 4.14: **EHI(BC)** and **EHI(RC)** decompositions of GCC applied to the unsatisfiable CA($g = 2, k = 7, b = 11$). The x-axis measures Hall interval size. Time is measured in seconds. This set of charts are particularly interesting, and support my conjecture that RC decompositions become more useful as the problem gets, for some definition, harder. The curious peaks in this instance have actually *switched* from **EHI(RC)** to **EHI(BC)**. It's as though the two decompositions' characteristics have been swapped. The peaks for the BC decomposition are at Hall sizes of 3 and 6 this time, and for these problems, the RC decomposition has a lower run-time.

5 Conclusion

This project has investigated the effectiveness of reformulations of the AllDifferent and GCC global constraints in SAT and SMT. Despite the **HI(BC)** and **HI(RC)** decompositions being found to perform well in [14], it was shown in this report that the improvements do not translate into SMT, at least for the problems considered here. Even though it was confirmed that increasing the size of Hall intervals in **HI(BC)**, **HI(RC)**, **EHI(BC)** and **EHI(RC)** reduces the amount of search required in SMT, as it does in SAT, the experiments suggest that the reason these decompositions perform poorly in SMT is because of the higher cost of propagation as compared to SAT.

Therefore, if a problem is to be deployed in SMT, the modeller must carefully experiment with the decompositions selected to make sure their propagation does not dominate in the search for a solution to the problem. The allure of blindly employing the most consistent decomposition to all problems must be avoided. Measures of problem difficultly should be used to assess what level of consistency is appropriate. These measures of difficultly are likely going to be informal, but the importance of experimentation with the consistency levels of decompositions in SMT seems more important than the same considerations in SAT. The guidance from the experiments in this report would be to prefer simple, local encodings at first and only when it can be measured that the traversal of the search space is becoming a bottleneck should the modeller consider employing a more consistent decomposition. This advice mirrors the concept of premature optimisation in software engineering.

A further caveat to these decompositions is the unpredictability of conflicts. I have conjectured that bursts of extra propagation are induced by many conflict clauses being generated from the decompositions. This pattern appears exacerbated when decompositions are composed. Due to the seemingly high cost of propagation in SMT, this pattern is particularly pernicious. This observation seems to be a contradiction to [12] who conjectured and statistically supported that the greater the Boolean component, the better the SMT solver is expected to perform.

Another modelling observation is that the direct encoding performed terribly in SAT modulo $\mathcal{LIA}$ for the experiments considered in this report. Is is far better to use single variables constrained over the entire domain than Boolean variables denoting individual values in a domain.

An unexpected discovery, somewhat unrelated to the main thread of my report, was a very effective decomposition for Sudoku using the MiniSat+ to translate into SAT, and then Z3 to solve the resulting SAT problem. This decomposition proved to be the fasted method I came across in the SAT, SMT and CP communities. Given the sophistication of global constraints and symmetry breaking constraints in CP, I was surprised to find that this decomposition into SAT performed so well.

This project has presented more questions that it has answered, and a new avenue for research has suggested itself. A problem I routinely faced was not having a deep enough understanding of the internals of SMT solvers and their associated theory procedures. There are many moving parts in such a system, and a "black-box" understanding of the various components will only take the analysis so far. I would like to spend some time studying and perhaps trying to implement an SMT solver with an emphasis on helping the experimenter understand what is going on rather

than being a competitive system in terms of run-time and number of features. A limitation of this report was my shallow understanding of SMT solvers, which I would like to rectify in the future so as to give myself a better chance at understanding the effects of decompositions interacting with one another.

5.1 Further Work

There are likely opportunities available in developing new encodings for common global constraints using the primitives available in SMT-LIB. There has been little work invested into developing specialised encodings for SMT.

Some constraints are not effectively decomposable, and for those, it is likely that an encoding, in SAT or SMT, would not be profitable. Instead propagation algorithms are developed. However, there are a number of constraints that are effectively decomposable [76], including the Sequence, Precedence, Roots, Slide and Table constraints. An investigation of novel SMT encodings for any one of these constraints would be interesting in comparison to encodings used for SAT. In [18], decompositions of many effectively decomposable constraints were analysed. A project wishing to complete a systematic evaluation in SMT would be at a great advantage starting with the work in [18].

There appears to be much work left to do investigating the implementation of global propagation algorithms in SMT. A propagator for AllDifferent was developed in [83] and shown experimentally to perform very favourably when compared to state-of-the-art SAT propagation algorithms. It seems inevitable that other global constraints can be efficiently implemented in SMT.

One last scrap of an idea for further work is to explore the effect of lazy clause generation, as introduced in [17] and [16], more closely in SMT. This seems like a particularly prudent strategy in SMT due to the higher cost of propagation identified in this report.

Bibliography

[1] E. R. Bixby, "Robert e. bixby returns to cornell to deliver the fulkerson lectures on optimisation," http://www.orie.cornell.edu/news/index.cfm?news_id=62130&news_back=news_archive%26y%3D2008, accessed: 25/04/2014.

[2] E. Freuder, "In pursuit of the holy grail," *Constraints*, vol. 2, no. 1, pp. 57–61, 1997. [Online]. Available: http://dx.doi.org/10.1023/A%3A1009749006768

[3] K. Claessen, N. Een, M. Sheeran, and N. Sorensson, "Sat-solving in practice," in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on.* IEEE, 2008, pp. 61–67.

[4] J. Marques-Silva, "Practical applications of boolean satisfiability," in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on.* IEEE, 2008, pp. 74–80.

[5] R. Sebastiani, "Lazy satisfiability modulo theories," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, pp. 141–224, 2007. [Online]. Available: http://jsat.ewi.tudelft.nl/content/volume3/JSAT3_9_Sebastiani_.pdf

[6] M. N. Velev and R. E. Bryant, "Exploiting positive equality and partial non-consistency in the formal verification of pipelined microprocessors," in *Design Automation Conference, 1999. Proceedings. 36th.* IEEE, 1999, pp. 397–401.

[7] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani, "A lazy and layered smt (\ mathcal {BV}) solver for hard industrial verification problems," in *Computer Aided Verification.* Springer, 2007, pp. 547–560.

[8] R. Jhala and K. L. McMillan, "Array abstractions from proofs," in *Computer Aided Verification.* Springer, 2007, pp. 193–206.

[9] N. Beldiceanu, M. Carlsson, and J.-X. Rampon, "Global constraint catalog," *SICS Research Report*, 2005.

[10] J.-C. Régin, "A filtering algorithm for constraints of difference in CSPs," in *AAAI*, vol. 94, 1994, pp. 362–367.

[11] K. Stergiou and T. Walsh, "The difference all-difference makes."

[12] M. Bofill, M. Palahí, J. Suy, and M. Villaret, "Solving constraint satisfaction problems with SAT modulo theories," *Constraints*, vol. 17, no. 3, pp. 273–303, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10601-012-9123-1

[13] M. Bofill, M. Palahi, J. Suy, and M. Villaret, "Simply: a compiler from a csp modeling language to the smt-lib format," in *Proceedings of the 8th Intl. Workshop on Constraint Modelling and Reformulation*, 2009, pp. 30–44.

[14] C. Bessiere, G. Katsirelos *et al.*, "Decompositions of all different, global cardinality and related constraints."

[15] O. Ohrimenko, P. J. Stuckey, and M. Codish, "Propagation via lazy clause generation," *Constraints*, vol. 14, no. 3, pp. 357–391, 2009.

[16] T. Feydy and P. J. Stuckey, "Lazy clause generation reengineered," in *Principles and Practice of Constraint Programming-CP 2009*. Springer, 2009, pp. 352–366.

[17] O. Ohrimenko, P. J. Stuckey, and M. Codish, "Propagation= lazy clause generation," in *Principles and Practice of Constraint Programming–CP 2007*. Springer, 2007, pp. 544–558.

[18] N. Narodytska, "Reformulation of global constraints," Ph.D. dissertation.

[19] J. Truss, *Discrete mathematics for computer scientists*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[20] J. P. Marques-Silva and K. A. Sakallah, "Grasp: A search algorithm for propositional satisfiability," *Computers, IEEE Transactions on*, vol. 48, no. 5, pp. 506–521, 1999.

[21] V. Marek, *Introduction to Mathematics of Satisfiability*, ser. Chapman & Hall/CRC Studies in Informatics Series. Taylor & Francis, 2010. [Online]. Available: http://books.google.co.uk/books?id=6uWxLgKRO9gC

[22] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[23] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, pp. 530–535.

[24] J. Marques-Silva, "Practical applications of boolean satisfiability," in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*. IEEE, 2008, pp. 74–80.

[25] D. E. Knuth, *The Art of Computer Programming, Volume 4, Pre-Fascicle 6a: Satisfiability*. Addison-Wesley Professional, 2014.

[26] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into SAT." *JSAT*, vol. 2, no. 1-4, pp. 1–26, 2006.

[27] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning*. Springer, 1983, pp. 466–483.

[28] C. Barrett, A. Stump, C. Tinelli *et al.*, "The SMT-LIB standard: Version 2.0," Tech. Rep., 2010.

[29] R. Williams, C. P. Gomes, and B. Selman, "Backdoors to typical case complexity." Citeseer.

[30] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962. [Online]. Available: http://doi.acm.org/10.1145/368273.368557

[31] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, Nov. 2006. [Online]. Available: http://doi.acm.org/10.1145/1217856.1217859

[32] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning sat solvers," *Handbook of satisfiability*, vol. 185, pp. 131–153, 2009.

[33] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," *Handbook of Knowledge Representation*, vol. 3, pp. 89–134, 2008.

[34] L. Baptista and J. Marques-Silva, "Using randomization and learning to solve hard real-world instances of satisfiability," in *Principles and Practice of Constraint Programming–CP 2000*. Springer, 2000, pp. 489–494.

[35] A. S. Fukunaga, "Complete restart strategies using a compact representation of the explored search space."

[36] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[37] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[38] W. Ackermann, *Solvable cases of the decision problem*, ser. Studies in logic and the foundations of mathematics. North-Holland Pub. Co., 1954. [Online]. Available: http://books.google.co.uk/books?id=YTk4AAAAMAAJ

[39] R. Ramanujam, "Temporale logik: Model checking," 2000.

[40] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, "Deciding equality formulas by small domains instantiations," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds. Springer Berlin Heidelberg, 1999, vol. 1633, pp. 455–469. [Online]. Available: http://dx.doi.org/10.1007/3-540-48683-6_39

[41] S. B. Akers, "Binary decision diagrams," *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 509–516, 1978.

[42] M. R. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?" 1999.

[43] D. Kroening and O. Strichman, "A framework for satisfiability modulo theories," *Formal Aspects of Computing*, vol. 21, no. 5, pp. 485–494, 2009. [Online]. Available: http://dx.doi.org/10.1007/s00165-009-0105-z

[44] G. Audemard, P. Bertoli, and A. Cimatti, "A SAT based approach for solving formulas over boolean and linear mathematical propositions," *... Deduction—CADE-18*, pp. 195–210, 2002. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-45620-1_17

[45] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 2, pp. 245–257, 1979.

[46] O. Strichman, S. Seshia, and R. Bryant, "Deciding separation formulas with sat," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, E. Brinksma and K. Larsen, Eds. Springer Berlin Heidelberg, 2002, vol. 2404, pp. 209–222. [Online]. Available: http://dx.doi.org/10.1007/3-540-45657-0_16

[47] E. Freuder and A. Mackworth, "Constraint satisfaction: An emerging paradigm," in *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Elsevier, 2006, ch. 2.

[48] K. R. Apt, "The essence of constraint propagation," *Theoretical computer science*, vol. 221, no. 1, pp. 179–210, 1999.

[49] K. Apt, *Principles of constraint programming*. Cambridge University Press, 2003.

[50] W.-J. van Hoeve, "The alldifferent constraint: A survey," *arXiv preprint cs/0105015*, 2001.

[51] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh, "Global constraints for lexicographic orderings," in *Principles and Practice of Constraint Programming-CP 2002*. Springer, 2002, pp. 93–108.

[52] J.-C. Régin, "Global constraints and filtering algorithms," in *Constraint and Integer Programming*. Springer, 2004, pp. 89–135.

[53] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "Minizinc: Towards a standard cp modelling language," in *Principles and Practice of Constraint Programming–CP 2007*. Springer, 2007, pp. 529–543.

[54] P. Stuckey, M. Banda, M. Maher, K. Marriott, J. Slaney, Z. Somogyi, M. Wallace, and T. Walsh, "The g12 project: Mapping solver independent models to efficient solutions," in *Logic Programming*, ser. Lecture Notes in Computer Science, M. Gabbrielli and G. Gupta, Eds. Springer Berlin Heidelberg, 2005, vol. 3668, pp. 9–13. [Online]. Available: http://dx.doi.org/10.1007/11562931_3

[55] C. Bessiere, E. Hebrard, and T. Walsh, "Local consistencies in SAT," in *Theory and Applications of Satisfiability Testing*. Springer, 2004, pp. 299–314.

[56] T. Walsh, "SAT v CSP," in *Principles and Practice of Constraint Programming–CP 2000*. Springer, 2000, pp. 441–456.

[57] I. P. Gent, "Arc consistency in SAT," 2002.

[58] A. K. Mackworth, "Consistency in networks of relations," *Artificial intelligence*, vol. 8, no. 1, pp. 99–118, 1977.

[59] C. W. Choi, W. Harvey, J. H.-M. Lee, and P. J. Stuckey, "Finite domain bounds consistency revisited," in *AI 2006: Advances in Artificial Intelligence*. Springer, 2006, pp. 49–58.

[60] K. Marriott and P. Stuckey, *Programming with Constraints: An Introduction*, ser. Adaptive Computation and Machine. MIT Press, 1998. [Online]. Available: http://books.google.co.uk/books?id=jBYAleHTldsC

[61] *A Bounds-Based Reduction Scheme for Difference Constraints*, 1996.

[62] J.-F. Puget, "A fast algorithm for the bound consistency of alldiff constraints," 1998.

[63] N. Downing, T. Feydy, and P. J. Stuckey, "Explaining alldifferent," in *Proceedings of the Thirty-fifth Australasian Computer Science Conference-Volume 122*. Australian Computer Society, Inc., 2012, pp. 115–124.

[64] N. Barnier and P. Brisset, "Graph coloring for air traffic flow management," 2002.

[65] E. Tsang, J. Ford, P. Mills, R. Bradwell, R. Williams, and P. Scott, "Zdc-rostering: A personnel scheduling system based on constraint programming," Technical Report CSM, Tech. Rep., 2004.

[66] "Global Constraint Catalogue: `alldifferent`," http://www.emn.fr/z-info/sdemasse/gccat/Calldifferent.html#uid9995, accessed: 24/04/2014.

[67] K. Mehlhorn and S. Thiel, "Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint," in *Principles and Practice of Constraint Programming–CP 2000*. Springer, 2000, pp. 306–319.

[68] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. Van Beek, "A fast and simple algorithm for bounds consistency of the alldifferent constraint."

[69] C. Bessiere, G. Katsirelos, N. Narodytska, T. Walsh *et al.*, "Circuit complexity and decompositions of global constraints."

[70] C.-G. Quimper, "Efficient propagators for global constraints," Ph.D. dissertation, University of Waterloo, 2006.

[71] C. Turner, "Python implementation of Leconte's RC propagator," https://github.com/switicus/constraints/blob/master/leconte.py.

[72] Y. Caseau, P.-Y. Guillo, and E. Levenez, "A deductive and object-oriented approach to a complex scheduling problem," *Journal of Intelligent Information Systems*, vol. 4, no. 2, pp. 149–166, 1995.

[73] "Global Constraint Catalogue: `global_cardinality_low_up`," http://www.emn.fr/z-info/sdemasse/gccat/Cglobal_cardinality_low_up.html#uid17272, accessed: 24/04/2014.

[74] I. P. Gent, K. E. Petrie, and J.-F. Puget, "Symmetry in constraint programming," *Handbook of Constraint Programming*, pp. 329–376, 2006.

[75] J.-F. Puget, "Symmetry breaking revisited," in *Principles and Practice of Constraint Programming-CP 2002*. Springer, 2002, pp. 446–461.

[76] T. Walsh, "Exploiting constraints," in *Proceedings of Inductive Logic Programming 2011*. Springer, 2012.

[77] C.-G. Quimper, P. Beek, A. LÃşpez-Ortiz, A. Golynski, and S. Sadjad, "An efficient bounds consistency algorithm for the global cardinality constraint," in *Principles and Practice of Constraint Programming âĂŞ CP 2003*, ser. Lecture Notes in Computer Science, F. Rossi, Ed. Springer Berlin Heidelberg, 2003, vol. 2833, pp. 600–614. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45193-8_41

[78] G. Royle, "Minimum sudoku," http://school.maths.uwa.edu.au/~gordon/sudokumin.php, accessed: 6/12/2013.

[79] "The 2009 international conference on fpga technology: Design competition," http://fpt09.cse.unsw.edu.au/comp/benchmarks.html, accessed: 15/01/2014.

[80] C. Turner, "Source code for Sudoku experiments," https://github.com/switicus/smt_research/tree/master/ex/sudoku.

[81] A. Hartman and L. Raskin, "Problems and algorithms for covering arrays," *Discrete Mathematics*, vol. 284, no. 1, pp. 149–156, 2004.

[82] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, no. 2-3, pp. 199–219, 2006.

[83] M. Bankovic and F. Maric, "An Alldifferent constraint solver in SMT," 2010.