Theory Learning with Symmetry Breaking

Jacob M. Howe
Dept of Computer Science
City, University of London
j.m.howe@city.ac.uk

Ed Robbins
School of Computing
University of Kent
er209@kent.ac.uk

Andy King
School of Computing
University of Kent
a.m.king@kent.ac.uk

ABSTRACT

This paper investigates the use of a Prolog coded SMT solver in tackling a well known constraints problem, namely packing a given set of consecutive squares into a given rectangle, and details the developments in the solver that this motivates. The packing problem has a natural model in the theory of quantifier-free integer difference logic, a theory supported by many SMT solvers. The solver used in this work exploits a data structure consisting of an incremental Floyd-Warshall matrix paired with a watch matrix that monitors the entailment status of integer difference constraints. It is shown how this structure can be used to build unsatisfiable theory cores on the fly, which in turn allows theory learning to be incorporated into the solver. Further, it is shown that a problem-specific and non-standard approach to learning can be taken where symmetry breaking is incorporated into the learning stage, magnifying the effect of learning. It is argued that the declarative framework allows the solver to be used in this white box manner and is a strength of the solver. The approach is experimentally evaluated.

CCS CONCEPTS

• **Theory of computation** → *Shortest paths*; *Packing and covering problems*; • **Software and its engineering** → *Constraint and logic languages*;

KEYWORDS

SAT solving, Constraints

ACM Reference format:
Jacob M. Howe, Ed Robbins, and Andy King. 2017. Theory Learning with Symmetry Breaking. In *Proceedings of PPDP'17, Namur, Belgium, October 9–11, 2017,* 12 pages.
https://doi.org/10.1145/3131851.3131861

1 INTRODUCTION

Integer difference constraints have the form $x_i - x_j \leq c$, where c is an integer constant. Solving sets or boolean combinations of integer difference constraints is of interest in constraint solving where many positional constraints are naturally described as differences. They are also of interest in formal verification where descriptions of timing constraints and paths can be encoded, and software analysis, for example of concurrent programs [2].

Rectangle packing problems have attracted considerable interest in constraint programming [27] owing to their application in scheduling and layout design. This paper considers the consecutive squares packing problem [17], the problem of placing squares of size $1 \times 1, 2 \times 2, \ldots, n \times n$ into a rectangle of given dimensions. It is argued in [27] that this problem is a good test of search methods for constraint problems. The problem has a natural model as boolean combinations of integer difference constraints, and the disjunctive nature of the model suggests that the problem might be tackled using an SMT solver over the theory of integer difference constraints (often called integer difference logic). This theory has been supported by SMT solvers since their first development [23].

This paper details developments of a declarative SMT solver motivated by tackling this class of problems as a strength test. The solver is coded in Prolog and has at its heart the SAT solver from [12–14], whilst using constraint reification [25] as a mechanism to realise theory propagation. In [26] integer difference logic was integrated into the solver using an incremental variation of the Floyd-Warshall algorithm, together with a structure called a watch matrix that improves propagation from the theory decision procedure into the boolean component of the problem. The solver works with the natural declarative model of the problem, and this paper details enhancements to the solver. In particular, the Floyd-Warshall matrix can be adapted to build unsatisfiable cores, that is, certificates of unsatisfiability, and these can be used to add theory learning (the learning of new boolean constraints from theory failure) to the solver. Most significantly, theory learning is enhanced by exploiting symmetry, so that a single failure can be used to learn multiple clauses. Whilst this specialisation of learning is application specific in its detail, the tactic of specialising learning using problem structure is more generally applicable and promises to be a powerful tactic for SMT solving in general.

In previous work on SAT and SMT solvers built in Prolog it has been argued that the resulting succinct declarative solvers make for good white box solvers, that is, solvers that can easily be deployed and adapted for new applications [13, 14, 25, 26]. This work demonstrates this point – the application of the solver presented motivates a new kind of problem specific theory learning in the form of symmetry breaking theory learning, where one conflict leads to many learnt clauses. The declarative SMT framework naturally accommodates this new instance of learning.

This paper makes the following contributions:

- it is demonstrated that the incremental Floyd-Warshall and watch matrix structure used for integer difference logic can be adapted to enable unsatisfiable theory cores to be built on the fly;

ACM ISBN 978-1-4503-5291-8/17/10…$15.00
https://doi.org/10.1145/3131851.3131861

- it is shown how these cores can be used to add theory learning to an SMT solver;
- it is shown that rectangle packing problems can be naturally expressed in quantifier-free integer difference logic and an experimental evaluation of the solver on this class of problems is given;
- it is demonstrated that theory learning in this solver can be considerably strengthened by adding symmetry breaking in the learning phase, and it is argued that this is a strength of the underlying solver framework.

The remainder of this paper is structured as follows. Section 2 gives background on SAT, SMT and their implementation in Prolog and section 3 describes the rectangle packing problem. Section 4 describes how the incremental Floyd-Warshall approach to solving integer difference logic problems can be used to incorporate theory learning into the solver, then section 5 describes how this might be used with symmetry breaking to make it more powerful. Section 6 gives an experimental evaluation of the resulting solver, Section 7 gives a brief overview of related work, whilst section 8 discusses these results and the structure of the solver.

2 SAT MODULO THEORIES

This section gives background on SAT solving, SAT modulo theories (SMT), quantifier-free integer difference logic and their implementation using logical variables, delay declarations and reification in Prolog.

2.1 SAT solving

The boolean satisfiability problem (SAT) is the problem of determining whether for a given boolean formula, there is a truth assignment to the variables of the formula under which the formula evaluates to true. Most recent SAT solvers are based on the Davis, Putnam, Logemann, Loveland (DPLL) algorithm [8] with watched literals [20].

At the heart of the DPLL approach is unit propagation. Let f be a propositional formula in CNF over a set of propositional variables X. Let $\theta : X \rightarrow \{\text{true}, \text{false}\}$ be a partial (truth) function. Unit propagation examines each clause in f to deduce a truth assignment θ' that extends θ and necessarily holds for f to be satisfiable. For example, suppose $f = (\neg x \vee z) \wedge (u \vee \neg v \vee w) \wedge (\neg w \vee y \vee \neg z)$ so that $X = \{u, v, w, x, y, z\}$ and θ is the partial function $\theta = \{x \mapsto \text{true}, y \mapsto \text{false}\}$. In this instance the clause $(\neg x \vee z)$ has one unassigned literal (is a unit clause), therefore for it to be satisfiable, hence f as a whole, it is necessary that $z \mapsto$ true. Moreover, for $(\neg w \vee y \vee \neg z)$ to be satisfiable, it follows that $w \mapsto$ false. The satisfiability of $(u \vee \neg v \vee w)$ depends on two unknowns, u and v, so no further information can be deduced from this clause. Hence $\theta' = \theta \cup \{w \mapsto \text{false}, z \mapsto \text{true}\}$.

Search for a satisfying assignment proceeds as follows: starting from an empty truth function θ, an unassigned variable x occurring in f is selected and $x \mapsto$ true is added to θ. Unit propagation extends θ until either no further propagation is possible or a contradiction is found. In the first case, if all clauses are satisfied then f is satisfied, else another unassigned variable is selected. In the second case, $x \mapsto$ false is added to θ; if this fails search backtracks to a previous assignment. Search and propagation continues in this manner. Further details can be found in [13, 18, 33].

The solver that provides the SAT engine for this work [13, 14] succinctly implements unit propagation with watched literals in 22 lines of Prolog using logical variables and delay declarations.

2.2 Integer Difference Logic

Difference constraints are a strict subclass of linear constraints where each constraint has the form $x_i - x_j \leq c$, where c is a constant. If x_i, x_j range over the integers and c is constrained to be an integer, then the constraints are called integer difference constraints. Note that bounds on a variable can be expressed in these constraints by introducing an additional variable z which is interpreted as zero. Then $x_i \leq c$ becomes $x_i - z \leq c$ and $x_i \geq c$ becomes $z - x_i \leq -c$.

This class of constraints has been used in SMT solving [22], where the theory of propositional combinations of these constraints is called quantifier-free integer difference logic (QF_IDL, or IDL).

The key result in solving difference constraints is that a system of difference constraints can be modelled as a weighted directed graph where the nodes are the variables occurring in the system and each constraint is encoded as an edge in the graph: $x_i - x_j \leq c$ gives an edge from node x_i to node x_j weighted by c. Satisfiability of the system is equivalent to absence of a negative cycle [5]. The Bellman-Ford single source shortest path algorithm can establish this in $O(mn)$ where n is the number of nodes and m is the number of edges. This algorithm is commonly used in SMT solvers over integer difference logic. However, this means that entailment or disentailment of constraints is not detected as part of the consistency check, meaning that complete propagation from the theory into boolean component is not achieved.

In [26] the Floyd-Warshall algorithm [10, 32] for finding the shortest path between each pair of nodes in a graph is used. Whilst this algorithm is $O(n^3)$ for establishing consistency it allows better propagation to be achieved, as explained in section 2.6.

Consider as an example the system consisting of (the conjunction of) the inequalities:

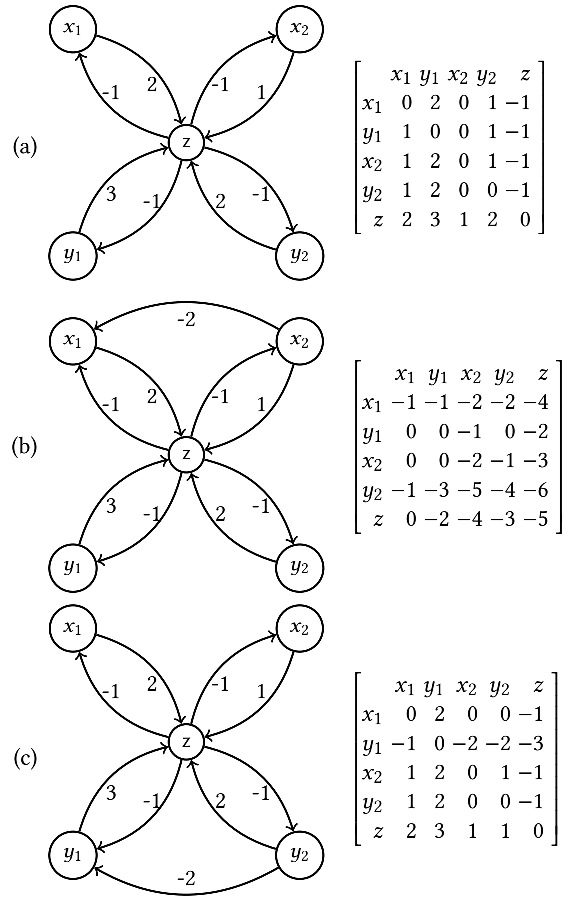
$$\begin{array}{ll} x_1 - z \leq 2 & x_2 - z \leq 1 \\ z - x_2 \leq -1 & z - x_1 \leq -1 \\ y_1 - z \leq 2 & y_2 - z \leq 1 \\ z - y_2 \leq -1 & z - y_1 \leq -1 \end{array}$$

In case (a) of Figure 1 this system is represented as a graph, and the matrix representing this graph after the Floyd-Warshall algorithm has been applied to it is presented.

A key attraction of Floyd-Warshall is that it can be applied incrementally [4], that is, once the algorithm has been applied to a graph, a single new edge can be added to the graph (constraint to the system) and the matrix describing the shortest paths can be updated, rather than recomputed from scratch. This means checking for a shorter path for each entry in the matrix, hence update takes $O(n^2)$ time.

Consider the system above augmented with the inequality $x_2 - x_1 \leq -2$ which is an inconsistent system. This is represented in case (b) of Figure 1. If any of the entries along the diagonal of the matrix is negative then the system is inconsistent. Note that when the system becomes inconsistent this is all the matrix represents –

(a)



$$\left[\begin{array}{c|ccccc} & x_1 & y_1 & x_2 & y_2 & z \\ x_1 & 0 & 2 & 0 & 1 & -1 \\ y_1 & 1 & 0 & 0 & 1 & -1 \\ x_2 & 1 & 2 & 0 & 1 & -1 \\ y_2 & 1 & 2 & 0 & 0 & -1 \\ z & 2 & 3 & 1 & 2 & 0 \end{array}\right]$$

(b)

$$\left[\begin{array}{c|ccccc} & x_1 & y_1 & x_2 & y_2 & z \\ x_1 & -1 & -1 & -2 & -2 & -4 \\ y_1 & 0 & 0 & -1 & 0 & -2 \\ x_2 & 0 & 0 & -2 & -1 & -3 \\ y_2 & -1 & -3 & -5 & -4 & -6 \\ z & 0 & -2 & -4 & -3 & -5 \end{array}\right]$$

(c)

$$\left[\begin{array}{c|ccccc} & x_1 & y_1 & x_2 & y_2 & z \\ x_1 & 0 & 2 & 0 & 0 & -1 \\ y_1 & -1 & 0 & -2 & -2 & -3 \\ x_2 & 1 & 2 & 0 & 1 & -1 \\ y_2 & 1 & 2 & 0 & 0 & -1 \\ z & 2 & 3 & 1 & 1 & 0 \end{array}\right]$$

Figure 1: Illustrating Floyd-Warshall

with a negative cycle any path including this negative cycle can become arbitrarily negative, so the values no longer represent shortest path lengths.

Also consider the original system augmented with $y_2 - y_1 \leq -2$. Here the system is still consistent and the entries of the matrix are updated to record the shortest paths between nodes, as illustrated in case (c) of Figure 1. For example, in the original system the path from x_1 to y_1 had weight 1, the path from x_1 to y_2 had weight 1 and the path from y_1 to itself weight 0. The new inequality gives an edge from y_2 to y_1 with weight -2. The two original paths are linked by this to give a path from x_1 to y_1 with weight -1 which represents the shortest path between these two nodes and the matrix is updated accordingly.

2.3 SMT solving

SAT modulo theories (SMT) gives a general scheme for determining the satisfiability of problems consisting of a formula over atomic constraints in some theory T, whose set of literals is denoted Σ [23, 31]. The scheme separates the propositional skeleton – the logical structure of combinations of theory literals – and the meaning of the literals. A bijective encoder mapping $e : \Sigma \rightarrow X$ associates each literal with a unique propositional variable. Then the encoder mapping e is lifted to theory formulae, using $e(\phi)$ to denote the propositional skeleton of a theory formula ϕ.

The example from section 2.2 is extended with four additional theory literals, encoded by $v_9, \ldots v_{12}$ in the encoded map below:

$$\begin{array}{ll} e(x_1 - z \leq 2) = v_1 & e(x_2 - z \leq 1) = v_2 \\ e(z - x_2 \leq -1) = v_3 & e(z - x_1 \leq -1) = v_4 \\ e(y_1 - z \leq 2) = v_5 & e(y_2 - z \leq 1) = v_6 \\ e(z - y_2 \leq -1) = v_7 & e(z - y_1 \leq -1) = v_8 \\ e(x_1 - x_2 \leq -1) = v_9 & e(x_2 - x_1 \leq -2) = v_{10} \\ e(y_1 - y_2 \leq -1) = v_{11} & e(y_2 - y_1 \leq -2) = v_{12} \end{array}$$

The conjunction from section 2.2 is further conjoined with the disjunction of the four additional inequalities giving the boolean combination of inequalities ϕ, where $e(\phi)$ is:

$$e(\phi) = \begin{array}{l} v_1 \wedge v_2 \wedge v_3 \wedge v_4 \wedge v_5 \wedge v_6 \wedge v_7 \wedge v_8 \wedge \\ (v_9 \vee v_{10} \vee v_{11} \vee v_{12}) \end{array}$$

A SAT solver gives a truth assignment θ satisfying the propositional skeleton. From this, a conjunction of theory literals, $\hat{Th}_\Sigma(\theta, e)$ is constructed. The subscript Σ will be omitted when it refers to all literals in a problem. To construct the conjunction, where $\ell \in \Sigma$, a conjunct is the literal ℓ if $\theta(e(\ell)) =$ true and $\neg\ell$ if $\theta(e(\ell)) =$ false. This problem is passed to a solver for the theory that can determine satisfiability of conjunctions of constraints. Either satisfiability or unsatisfiability is determined, in the latter case the SAT solver is asked for further satisfying truth assignments. This formulation of SMT is known as the lazy-basic approach and details on its Prolog implementation can be found in [14].

2.4 SMT, the DPLL(T) approach

The lazy-basic approach finds complete satisfying assignments to the SAT problem given by the propositional skeleton before computing the satisfiability of the theory problem $\hat{Th}(\theta, e)$. Another approach is to couple the SAT and the theory problems more tightly by determining constraints entailed by the theory and propagating the bindings back into the SAT problem. This is known as theory propagation and is encapsulated in the DPLL(T) approach. Figure 3 gives a recursive formulation of DPLL(T) derived from Algorithm 11.2.3 from [18]. Importantly, on line (7) of propagate this includes a conflict analysis determining why failure has occurred in the theory component $\hat{Th}(\theta, e)$. The conflict analysis can then learn clauses that when conjoined with the boolean component of the problem block this failure from occurring again. This is referred to as theory learning throughout this paper. A more general formulation of DPLL(T) would additionally replace lines (11)-(15) of DPLL(T) with a conflict analysis on the boolean component.

The two functions have access to a boolean formula f (initially f is the propositional skeleton of the input problem, $e(\phi)$, but it may be updated) and an encoder mapping, e. The function DPLL(T) has as its argument a partial truth assignment, θ. DPLL(T) is first invoked with θ empty. It returns a truth assignment if the problem is satisfiable and the constant $\bot$ otherwise.

The call to propagate is the key operation. The function returns a pair consisting of a truth assignment and *res* taking value $\top$ or $\bot$ indicating the satisfiability of f and $\hat{Th}(\theta, e)$. The second argument to propagate is a set of theory literals, D, and the function begins by extending the truth assignment by assigning propositional variables identified by the encoder mapping. Next, unit propagation as described in section 2.1 is applied. The deduction function then infers those literals that hold as a consequence of the extended truth assignment. The function returns a pair consisting of a set of theory literals entailed by $\hat{Th}(\theta_2, e)$ and a flag res whose value is $\perp$ if $\hat{Th}(\theta_2, e)$ or θ_2 are inconsistent and $\top$ otherwise. The function propagate calls itself recursively until no further propagation is possible. After deduction returns, if f is not yet satisfied and no conflict has occurred then a further truth assignment is made and DPLL(T) calls itself recursively.

The DPLL(T) approach incrementally investigates the consistency of the posted constraints as propositional variables are assigned. Further, it identifies literals, ℓ, such that $\hat{Th}(\theta, e) \models \ell$, allowing $e(\ell)$ to be assigned during propagation. The interplay between propositional satisfiability and the consistency of the store $\hat{Th}(\theta, e)$ is important to this approach. Theory conflict analysis can reduce the search space by learning additional boolean clauses that must be satisfied. It could simply return true (in which case it does not reduce the search space), but more powerful analyses will potentially give more and stronger clauses. Incorporating techniques for adding power to theory learning is a key component of this investigation.

2.5 Theory propagation and reification

This section provides a framework for incorporating theory propagation into the propagation framework of the SAT solver from [14]. The approach is based on reifying theory literals with logical variables [25, 26]. As will be illustrated in subsequent sections, this exploits the control provided by delay declarations to realise theory propagation. The integration is almost seamless since the base SAT solver is also realised using logical variables and delay declarations.

There are three major steps in setting up a DPLL(T) solver for some problem ϕ: setting up the encoder map e, linking each theory literal in a problem with a logical variable; posting theory propagators (adding constraints) that reify the theory literals with the logical variables provided by e; posting the SAT problem defined by the propositional skeleton $e(\phi)$, then solving the whole problem. The code in Figure 2 describes the high level call to the solver.

Set up: Where `Prob` is an SMT formula over some theory, let lit(`Prob`) be the set of literals occurring in `Prob`. `TheoryLiteral` is a list of pairs $\ell - e(\ell)$ (or rather, $\ell \leftrightarrow e(\ell)$), where $\ell \in lit$(`Prob`), that defines the encoder mapping e. `Skeleton` represents the propositional skeleton of the problem, e(`Prob`). `Vars` represents the set of variables $e(\ell)$. The role of the predicate `setup(+,-,-,-)` is, given `Prob`, to instantiate the remaining variables.

Theory propagators: The role of `post_theory` is to set up predicates to reify each theory literal. The control on these predicates is key; the predicates need to be blocked until either $e(\ell)$ is assigned, or the literal (or its negation) is entailed by the constraint store $\hat{Th}(\theta, e)$. That is, the predicate for $\ell - e(\ell)$ will propagate in one of four ways:

- If $\hat{Th}(\theta, e) \models \ell$ then $e(\ell) \mapsto$ true
- If $\hat{Th}(\theta, e) \models \neg\ell$ then $e(\ell) \mapsto$ false
- If $e(\ell)$ = true then the store is updated to $\hat{Th}(\theta \cup \{e(\ell) \mapsto \text{true}\}, e)$
- If $e(\ell)$ = false then the store is updated to $\hat{Th}(\theta \cup \{e(\ell) \mapsto \text{false}\}, e)$

Boolean propagators: The role of `post_boolean` is to set up unit propagators for the SAT part of the problem e(`Prob`). This is a call to `problem_setup` as described in [14]. Search is then driven by assignments to the variables using `elim_var`.

Implementing the interface provided by predicates `setup` and `post_theory`, together with the SAT solver from [14] results in a DPLL(T) SMT solver. Note that the propagators posted for the theory and boolean components are intended to capture the spirit of the function propagate from Figure 3. Indeed, the integration between theory and boolean propagation is even tighter than the algorithm indicates. Rather than performing unit propagation to completion, then performing theory propagation, then repeating, here the assignment of a boolean variable is immediately communicated to the theory. This tactic is known as immediate propagation [18] and is a natural consequence of using Prolog's control to implement propagators.

```
dpll_t(Prob):-
    setup(Prob, TheoryLiterals, Skeleton, Vars),
    post_theory(TheoryLiterals),
    post_boolean(Skeleton),
    elim_var(Vars).
```

Figure 2: Interface to the DPLL(T) solver

2.6 SM(IDL) using the Watch Matrix

Improved theory propagation was achieved for SM(IDL) in [26] by pairing the Floyd-Warshall matrix with a new structure called the watch matrix. The desired propagation behaviour is as follows: for each constraint reified by the encoder mapping, if that constraint is entailed its reifying variable should be set to true and if that constraint is disentailed it should be set to false. The watch matrix captures this behaviour by shadowing the Floyd-Warshall matrix. Each entry in the Floyd-Warshall matrix is indexed by a pair of variables (x_i, x_j). Each constraint in Σ has form $x_i - x_j \le c$ and its negation $x_j - x_i \le -(c+1)$. The watch matrix has (x_i, x_j) entries consisting of a pair of lists each element being a constant c (from $x_i - x_j \le c$ or $x_j - x_i \le c$) itself paired with the appropriate reifying variable. The first list corresponds to those inequalites that should be true if entailed, the second those that should be false. These lists are initialised during the setup stage. If an entry in the Floyd-Warshall matrix changes then the corresponding entries in the watch matrix are retrieved, checked for entailment or disentailment and the reifying variable assigned, if appropriate.

For example, returning to the system considered above in Figure 1, case (a), and the inequality $x_1 - x_2 \le -1$. The negation of this (integer) inequality is $x_2 - x_1 \le 0$. Also, inequality $x_2 - x_1 \le -2 \in \Sigma$. Then the watch matrix has (x_2, x_1) entry `[-2-V10]-[0-V9]`. Observe that when incrementally setting up the matrix the (x_2, x_1) entry of the Floyd-Warshall matrix is set to a new value 0. Since $x_2 - x_1 \le 0 \not\models x_2 - x_1 \le -2$, `V10` the reifying variable for $x_2 - x_1 \le -2$ should be set to false.

The code that instantiates the interface for `post_theory` is given across Figures 4, 5 and 6. Figure 4 builds `Store` as AVL trees where

```
input:  f: CNF formula, e : Σ → X

(1)   function DPLL(T)(θ : truth assignment)
(2)   begin
(3)       (θ_3, res) := propagate(θ, ∅);
(4)       if (is-satisfied(f, θ_3)) then
(5)           return θ_3;
(6)       else if (res = ⊥) then
(7)           return ⊥;
(8)       else
(9)           x := choose-free-variable(f, θ_3);
(10)          (θ_4, res) := DPLL(T)(θ_3 ∪ {x ↦ true});
(11)          if (res = ⊤) then
(12)              return θ_4;
(13)          else
(14)              return DPLL(T)(θ_3 ∪ {x ↦ false});
(15)          endif
(16)      endif
(17)  end

(1)   function propagate(θ : truth assignment, D ⊆ Σ)
(2)   begin
(3)       θ_1 := θ ∪ {e(ℓ) ↦ true | ℓ ∈ D ∩ Σ}
                  ∪{e(ℓ) ↦ false | ¬ℓ ∈ D ∧ ℓ ∈ Σ};
(4)       θ_2 := θ_1∪ unit-propagation(f,θ_1);
(5)       ⟨D, res⟩ := deduction(T̂h(θ_2, e));
(6)       if (res = ⊥)
(7)           f′ = analyse-theory-conflict(D);
(8)           f := f ∧ f′
(9)           return (θ_2, res);
(10)      else if (D = ∅)
(11)          return (θ_2, res);
(12)      else
(13)          return propagate(θ_2, D);
(14)      endif
(15)  end
```

Figure 3: Recursive formulation of the DPLL(T) algorithm

```
post_theory(Encoding, Store) :-
    build_store(Encoding, Store),
    setup_reify(Encoding, Queue),
    process_queue(Queue, Store).

build_store([], Store) :-
    empty_avl(Matrix), empty_avl(Watch),
    Store = store(Matrix, 0, Watch).
build_store([(X-Y =< C)-Prop | Rest], Store) :-
    build_store(Rest, StoreRest),
    StoreRest = store(Matrix1, N1, Watch1),
    Store = store(Matrix3, N3, Watch3),
    add_var(X, N1, Matrix1, N2, Matrix2),
    add_var(Y, N2, Matrix2, N3, Matrix3),
    (avl_fetch(X-Y, Watch1, EntL-DisL) ->
        avl_store(X-Y, Watch1,
                  [C-Prop | EntL]-DisL, Watch2)
    ;
        avl_store(X-Y, Watch1, [C-Prop]-[], Watch2)
    ),
    NegC is -(C + 1),
    (avl_fetch(Y-X, Watch2, EntL2-DisL2) ->
        avl_store(Y-X, Watch2,
                  EntL2-[NegC-Prop | DisL2], Watch3)
    ;
        avl_store(Y-X, Watch2,
                  []-[NegC-Prop], Watch3)
    ).
```

Figure 4: Setting up the Floyd-Warshall matrix and the watch matrix

`Matrix3` is the Floyd-Warshall matrix and `Watch1` is the watch matrix. Figure 5 defines `setup_reify` that controls the addition of theory literals to the Floyd-Warshall (hence also watch) matrix. `bool_wait` delays on the instantiation of the reifying variable, if the reifying variable is instantiated the constraint or its negation is added into a queue. If variable `Queue` is instantiated then this resumes the delayed `process_queue`, which in turn updates the matrices using `process_constraint` as given in Figure 6; `floyd_warshall` will fail if a negative cycle is detected.

3 RECTANGLE PACKING

The problem considered in this paper is the consecutive squares rectangle packing problem, where the challange is: given three integers, n, m_1, m_2, can squares of size 1×1, 2×2, ..., $n \times n$ be packed without overlap inside a rectangle of size $m_1 \times m_2$? It might further be asked what are the dimensions of the minimally enclosing rectangle. This problem and similar are of importance since they relate to scheduling and design problems, and have been considered widely in constraint programming [15–17, 27]. The state of the art [16] for an optimal packing solves the $n = 32$ problem solved (in over three weeks of computation time).

This problem has a natural, if naïve, description as an SMT problem and it is this description that is considered in this paper. The positioning of each square i (the $i \times i$ square) is described by two variables x_i, y_i. Here, x_i describes the positioning of the left hand side of the square, and y_i describes the positioning of the bottom side of the square. There are two kinds of contraints. First, bounding constraints which ensure that the squares fit inside the rectangle. So for square i to be placed inside rectangle of dimension $m_1 \times m_2$ the bound constraints are (where z is the zero variable):

$$z - x_i \leq 0 \wedge x_i - z \leq m_1 - i \wedge z - y_i \leq 0 \wedge y_i - z \leq m_2 - i$$

Second, constraints that ensure that each pair of squares does not overlap. That is, for square i and square j:

$$x_i - x_j \leq -i \vee x_j - x_i \leq -j \vee y_i - y_j \leq -i \vee y_j - y_i \leq -j$$

A particularly simple example is to pack the 1×1 and 2×2 squares into a 2×3 rectangle. This is the example given in Section 2.2. The bound constraints are already given above, and to complete the

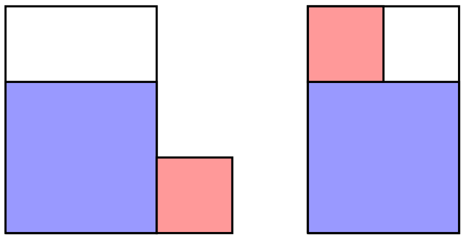
```
setup_reify([], _).
setup_reify([(X-Y =< C)-Prop | Rest], Queue) :-
    bool_wait(Prop, X, Y, C, Queue),
    setup_reify(Rest, Queue).

:- block bool_wait(-, ?, ?, ?, ?).
bool_wait(Prop, X, Y, C, Queue) :-
    Prop == true, !,
    insert_queue(Queue, X, Y, C).
bool_wait(Prop, X, Y, C, Queue) :-
    Prop == false, !,
    NegC is -(C + 1),
    insert_queue(Queue, Y, X, NegC).

insert_queue(Queue, X, Y, C) :-
    var(Queue), !,
    Queue = [(X-Y =< C) | _Cons].
insert_queue([_Con | Cons], X, Y, C) :-
    insert_queue(Cons, X, Y, C).

:- block process_queue(-, ?).
process_queue(Queue, Store1) :-
    nonvar(Queue), !,
    Queue = [(X-Y =< C) | Cons],
    process_constraint(X-Y =< C, Store1, Store2),
    process_queue(Cons, Store2).
```

Figure 5: Propagating from the SAT solver to the theory solver

problem description the following disjunction is required:

$$x_1 - x_2 \leq -1 \vee x_2 - x_1 \leq -2 \vee y_1 - y_2 \leq -1 \vee y_2 - y_1 \leq -2$$

Below are two possible configurations, the first with the second disjunct satisfied (but not the problem), the second with the fourth disjunct (and the problem) satisfied.



In Figure 1, case (a) gives the bound constraints. Case (b) corresponds to the situation where second disjunct is assigned the value true (the first picture above). Case (c) represents the situation where the fourth disjunct is assigned true. The solution to a larger example (15,39,33) is given in Figure 7.

4 ADDING THEORY LEARNING

This section demonstrates how the incremental Floyd-Warshall approach can be developed so that an unsatisfiable core of an unsatisfiable system is determined alongside a theory failure. It further shows how this core can be used within the solver to implement theory learning.

```
process_constraint(X-Y =< C, StoreIn, StoreOut) :-
    StoreIn = store(Matrix1, N, Watch),
    StoreOut = store(Matrix3, N, Watch),
    avl_fetch(X-Y, Matrix1, C_XY),
    min(C_XY, C, Min),
    (C == Min ->
        matrix_update(X-Y, C, Matrix1,
                      Matrix2, Watch),
        floyd_warshall(N, Matrix2, Matrix3, Watch)
    ;
        Matrix3 = Matrix1
    ).

matrix_update(Key, Value, Matrix1, Matrix2, Watch) :-
    (avl_fetch(Key, Watch, EntL-DisL) ->
        true
    ;
        EntL = [], DisL = []
    ),
    entailed(EntL, Value),
    disentailed(DisL, Value),
    avl_store(Key, Matrix1, Value, Matrix2).

entailed([], _).
entailed([C-Prop | Rest], Min) :-
    (Min =< C -> Prop = true ; true),
    entailed(Rest, Min).

disentailed([], _).
disentailed([C-Prop | Rest], Min) :-
    (Min =< C -> Prop = false ; true),
    disentailed(Rest, Min).
```

Figure 6: Propagating from the theory solver to the SAT solver

The key operation of the incremental Floyd-Warshall algorithm is that if a new inequality is added to the system – and this new inequality gives a lower entry in the matrix than the existing one – then for each pair of nodes it is determined whether using this new edge leads to a shorter path [4]. Suppose that the (x_i, x_j) entry is c (the shortest path from x_i to x_j has length c) and that the entry for (x_k, x_ℓ) is updated to value d. Then the values for (x_i, x_k) and (x_ℓ, x_j) are looked up finding values c_1, c_2. If $c' = c_1 + c_2 + d < c$ then the (x_i, x_j) entry is updated to c' since a new shortest path has been found using the new edge between x_k and x_ℓ.

To find an unsatisfiable core, the Floyd-Warshall matrix is augmented so that each entry is a pair consisting of an integer (the length of the shortest path between two nodes, as before) and a list (explicitly describing the shortest path; it is well known that algorithms such as Floyd-Warshall can be made self-certifying in this way). Returning to the above, the entry for (x_i, x_j) entry is now a pair $c - L$, where L is a list. If an inequality is added to the system, then the shortest path is the direct one, so the entry for (x_k, x_ℓ) is $d - [(x_k, x_\ell)]$. Looking up the values for (x_i, x_k) and (x_ℓ, x_j) gives $c_1 - L_1, c_2 - L_2$. If $c' = c_1 + c_2 + d < c$ then the (x_i, x_j)

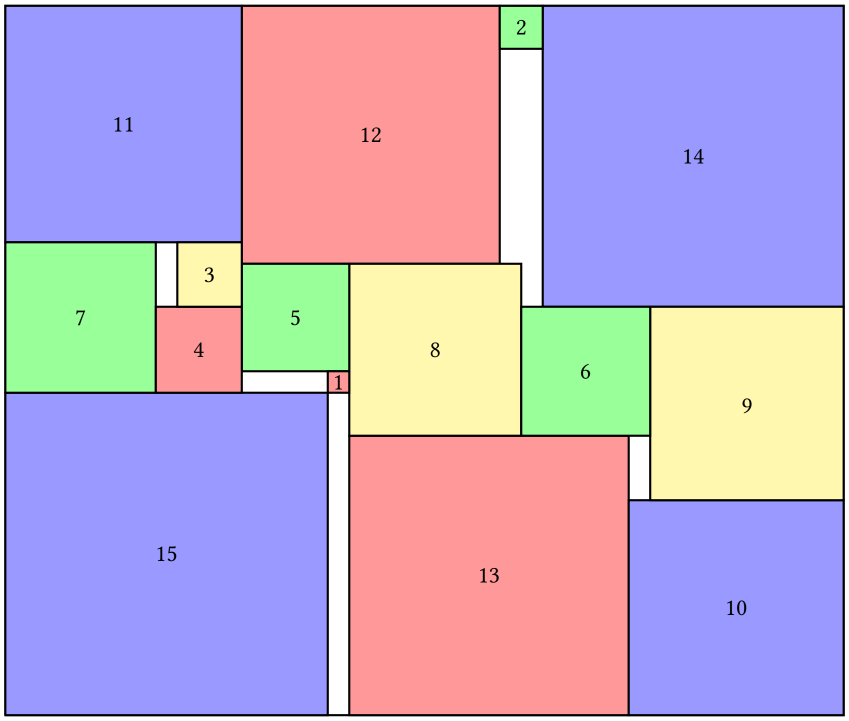


Figure 7: Solution for (15,39,33)

entry is updated to $c' - L'$, where $append(L_1, [(x_k, x_\ell)|L_2], L')$. The previous path L is discarded, and L' represents the new shortest path via the new edge.

The system is unsatisfiable if for some i the entry for (x_i, x_i) is negative. With the augmented matrix, when a negative entry is found it comes paired with a path describing the negative cycle. This cycle represents an unsatisfiable core of inequalities. At least in the current set up, the length of this path is typically short (since the number of squares that can be placed next to each other is small) and the overhead in maintaining this structure is minimal.

Having found an unsatisfiable core, to achieve theory learning a clause needs to be added to the problem expressing the unsatisfiability of this core. Where $\ell_i \in \Sigma$, suppose that a negative cycle has been found consisting of the edges given by inequalities $\ell_1, \ldots, \ell_n$ posted positively and $\ell_{n+1}, \ldots, \ell_m$ negatively. Then the learned clause $\neg e(\ell_1) \vee \ldots \vee \neg e(\ell_n) \vee e(\ell_{n+1}) \vee \ldots \vee e(\ell_m)$ is added to the boolean part of the problem.

To implement this in a Prolog backtracking solver requires an approach that will not undo the posting of the learned clause on backtracking. This is achieved using state restoration following an approach similar to that discussed in [14]. Each logical variable is indexed by a unique integer. Variable assignment includes maintaining an assignment history on the blackboard, which can be retrieved on backtracking. As shown in Figure 8, assignment consists of four clauses. The first skips past already assigned variables (`edit_restore` catches the possibility that a learnt clause means that a variable is assigned before restoration), the second and third assign to true and false recording this to the blackboard and the fourth clause allows this history to be popped on backtracking during search. The blackboard call `bb_get(core,[])` enforces backtracking once a theory failure has occurred. To suspend search (in this instance because of theory failure that is captured by the non-empty value stored on the blackboard under `core`) `assign_vars` fails to root, that is, all variable assignments are undone. To resume, a copy of the history is used to ensure that variables are assigned the values they previously held before search continues as before. There is a cost associated with this: the (re-)assignment and propagation for the current path, but search itself is not repeated.

To enable theory learning the blackboard is used again. The call to `floyd_warshall/4` (see Figure 6) fails when a negative cycle is detected. The watch matrix allows the encoder variables for the constraints to be found. The indices for these variables together with the values assigned to them are then written to the blackboard (under `core`). As described above search fails to root. The predicate `elim_var` that controls variable assignment is redefined as in Figure 8. The first clause attempts to assign values to the variables. If this fails (in this instance a theory failure), the second clause says that learning is applied (adding a blocking clause) before `elim_vars` calls itself recursively on this augmented program. In this way, the boolean component of the problem is extended monotonically. Note that the core is empty when the search space has been exhausted and that this leads to failure. As shown in Figure 9 predicate `post_new_boolean` builds a new clause and uses `learnt` to post this to the boolean component.

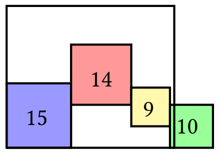
```
elim_var(Vars):-
  assign_vars(Vars).
elim_var(Vars):-
  learn(Vars),
  elim_var(Vars).

learn(Vars):-
    bb_get(core, Core), Core \== [],
    post_new_boolean(Core, Vars).

assign_vars([]).
assign_vars([V | Vs]) :-
    label_var(V),
    assign_vars(Vs).

label_var(V-N) :-
    ground(V), !,
    edit_restore(V-N).
label_var(V-N) :-
    bb_get(core, [])
    restore_from_history(N),
    update_history_true(N),
    V = true.
label_var(V-N) :-
    bb_get(core, []),
    update_history_false(N),
    V = false.
label_var(_-N) :-
    bb_get(core, []),
    update_history_back(N),
    fail.
```

Figure 8: Variable assignment

As an example, suppose that when searching for a solution to the $(15, 39, 33)$ problem, the following negative cycle is encountered $[(z, x_{15}), (x_{15}, x_{14}), (x_{14}, x_9), (x_9, x_{10}), (x_{10}, z)]$. Since this has negative weight a theory failure occurs. This is recorded by writing `[478-true, 1-true, 65-true, 82-true, 457-true]` to the blackboard where the integers are the indices for the encoder variables for inequalities, that is:

$$\begin{aligned} e(z - x_{15} \leq 0) &= v_{478} \\ e(x_{15} - x_{14} \leq -15) &= v_1 \\ e(x_{14} - x_9 \leq -14) &= v_{65} \\ e(x_9 - x_{10} \leq -9) &= v_{82} \\ e(x_{10} - z \leq 29) &= v_{457} \end{aligned}$$

This describes the following failure (the y positions are not being considered and could be any consistent set of values):



At this point the solver fails to the root of the search. Learning uses the variable indices to find their associated logical variables and

```
post_new_boolean(Core, Vars, _, _) :-
    build_clause(Core, Vars, Clause), !,
    learnt(Clause),
    bb_put(core, []).

build_clause([], _, []).
build_clause([N-B|Ns], Vs, [V-N-NegB | Rest]) :-
    lookup(Vs, N, V),
    negateB(B, NegB),
    build_clause(Ns, Vs, Rest).

negateB(true,false).
negateB(false,true).

lookup([V-N | _], N, V):- !.
lookup([_ | Vs], N, V) :-
    lookup(Vs, N, V).
```

Figure 9: Learning a single clause

the learnt blocking clause is added to prevent this from occurring again. That is, clause

$$\neg v_{478} \lor \neg v_1 \lor \neg v_{65} \lor \neg v_{82} \lor \neg v_{457}$$

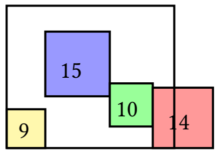
is added to the problem. Variables v_{478}, v_{457} correspond to the bounds on the rectangle and by the problem construction these must always be assigned true. If the following partial assignment is made or inferred $\{v_1 \mapsto \text{true}, v_{65} \mapsto \text{true}\}$ the new clause infers that $v_{82} \mapsto$ false, hence the theory failure above can never again be encountered (note that in the implementation the learnt clause is specialised so that the bound variables v_{478}, v_{457}, that must be assigned true, are dropped from the learned clause, so the posted learned clause is $\neg v_1 \lor \neg v_{65} \lor \neg v_{82}$).

To conclude this section is should be noted that it is possible to find more than one core at a failure point – look along the matrix diagonal and every negative entry defines a core. Anecdotal experiment suggests that on the benchmark suite it is rare to find more than a single core.

5 SYMMETRY BREAKING THEORY LEARNING

In the consecutive squares packing problem, an unsatifiable theory core describes a configuration of squares that will not fit inside the given rectangle. The diagnosis stage analyses the failure and adds an appropriate blocking clause. However, observe that once such a core has been found other unsatisfiable configurations can also be deduced.

For example, consider again the failure to place the 15, 14, 9 and 10 squares each to the right of the last. If this is not possible, then it is also not possible to place these squares in order 9, 15, 10 and 14, each to the right of the previous. This leads to the following theory failure (again, note that the y positioning is not important):



This second failure is again made up of constraints in Σ, that is, with their encoder variables:

$$e(z - x_9 \le 0) = v_{454} \quad e(x_9 - x_{15} \le -9) = v_{62}$$
$$e(x_{15} - x_{10} \le -15) = v_{41} \quad e(x_{10} - x_{14} \le -10) = v_{46}$$
$$e(x_{14} - z \le 25) = v_{475}$$

Adding the blocking clause:

$$\neg v_{454} \vee \neg v_{62} \vee \neg v_{41} \vee \neg v_{46} \vee \neg v_{475}$$

will block this theory failure from ever occurring.

This symmetry under permutation gives 4!=24 ways in which the 9, 10, 14 and 15 square can be placed with each one to the right of the previous. Once the first of these theory failures has been detected, the other 23 can be deduced and it would be good to block all 24 from occurring without further search.

This is problematic in a standard SMT solver used as a black box. However, with the reification-based Prolog white box solver used here, the learning step can be specialised by the programmer to capture problem specific features – in this case symmetry between failures. This has been implemented in the solver, so that on failures such as the one above all the symmetric clauses are also learnt.

As described in the previous section, the unsatisfiable theory core is expressed as the indices of the encoder variables, and the value assigned to these variables. To find the symmetric cores involves a number of stages: using the inverse of the encoder mapping the constraints in the core are looked up; the set of integer variables occurring in these constraints is found; all possible permutations of these variables are generated (excluding the zero variable) and for each of these the constraints describing a symmetric unsatisfiable core are produced; finally the encoder mapping is used to build a boolean clause for each core, which is posted, as before.

Returning to the example, the theory failure involved the integer variables $[x_{15}, x_{14}, x_9, x_{10}]$ (as well as z). One of the 24 permutations of these variables is $[x_9, x_{15}, x_{10}, x_{14}]$ and this permutation is used to generate the symmetric constraint, hence blocking clause, above.

The code follows the scheme in the previous section, with a new implementation of `post_new_boolean` that posts many new boolean constraints rather than just one. As previously described, the learning mechanism uses the blackboard to pass back the indices for the encoder variables. `lookup/4` uses the encoder mapping to find the integer variables occurring in the constraints (if a constraint contains the zero variable then it is a bounding constraint that is always true, hence it is omitted from the construction). `construct_pairs` finds the variable pairs for each of the symmetric unsatisfiable cores, then `find_clauses` constructs the boolean blocking clauses. The clauses are built using `build_clause` given previously and `all_learnt` posts these new boolean clauses. The definition of predicate `permutations` that generates a list of all permutations of a list is omitted, as are `merge` and `memchk`.

The approach described here is applied only to cores that correspond to a series of squares each to the right of the previous (or

```
post_new_boolean(Core, Vars, Encoding, Zero) :-
    lookup(Core, Encoding, Zero, Out),
    construct_pairs(Out, Pairs),
    find_clauses(Pairs, Encoding, Extras),
    build_clauses(Extras, Vars, Clauses), !,
    all_learnt(Clauses),
    bb_put(core, []).

lookup([], _, _, []).
lookup([C-_ | Cs], Encoding, Zero, VarsOut) :-
    get_vars(Encoding, C, Zero, Vs),
    lookup(Cs, Encoding, Zero, Vars),
    merge(Vs, Vars, VarsOut).

get_vars([], _, _, []).
get_vars([I-_-C | _Rest], C, Zero, VsOut) :- !,
    term_variables(I, VsTmp),
    (memchk(VsTmp,Zero) -> VsOut=[]; VsOut=VsTmp).
get_vars([_ | Rest], C, Zero, Vs) :-
    get_vars(Rest, C, Zero, Vs).

construct_pairs(Vars, Pairs) :-
    permutations(Vars, [], Perms),
    find_pairs(Perms, Pairs).

find_pairs([],[]).
find_pairs([Perm|Perms], [Pair|Pairs]):-
    perm_to_pair(Perm,[],Pair),
    find_pairs(Perms,Pairs).

perm_to_pair([_V | []], Pairs, Pairs).
perm_to_pair([V1,V2 | Rest], Pairs, AllPairs) :-
    perm_to_pair([V2|Rest],[V1-V2|Pairs],AllPairs).

find_clauses([], _, []).
find_clauses([Pair | Pairs], Enc, [E | Extras]) :-
    find_clause(Pair, Enc, E),
    find_clauses(Pairs, Enc, Extras).

find_clause([], _, []).
find_clause([P | Ps], Encoding, [N-true | Ns]) :-
    from_encoding(Encoding, P, N),
    find_clause(Ps, Encoding, Ns).

from_encoding([], _, _). %never occurs
from_encoding([E | _Es], V1-V2, N) :-
    E = I-_-M, I =.. [_, U1-U2, _],
    V1 == U1, V2 == U2,!,
    M = N.
from_encoding([_ | Es], V1-V2, N) :-
    from_encoding(Es, V1-V2, N).

build_clauses([], _, []).
build_clauses([E | Es], Vars, [C|Cs]):-
    build_clause(E, Vars, C),
    build_clauses(Es, Vars, Cs).
```

Figure 10: Posting Symmetric Constraints

each above the previous). However, it is also possible to extract symmetries from more complicated cores.

Note that an alternative to this demand driven approach to symmetry breaking is possible: add symmetry breaking clauses up front before search. However, the number of possible symmetries is high leading to an infeasibly large problem specification.

6 EXPERIMENTAL RESULTS

This section presents empirical results on using the solver on the consecutive squares packing problem. The variable ordering for the search is such that variables relating pairs of squares covering greater area are assigned before pairs covering lesser area. In addition, variables are assigned so that assigning two variables to true in a clause from the original problem description is avoided. There is a single specialisation, breaking the symmetry between the placement of the $n \times n$ and the $n - 1 \times n - 1$ square.

The benchmarks include the known optimal packings for each n, and a range of problems selected to test the solver. The problem is considered at its most difficult when close to the optimum area, and when the shape of the rectangle is itself close to square. A selection of benchmarks fitting this description have been chosen. In particular, unsatisfiable problems have been chosen to test exploration of the entire search space. The solver has been built in SICStus Prolog 4.2.3, and the experiments have been run on a single core of a Dell Precision T7610 workstation with two 8 core Intel Xeon E5-2650v2 CPUs @ 2.6 to 3.4GHz and 128GB RAM.

The results are presented in Table 1. The first column gives the instance (n, m_1, m_2) where n is the size of the largest of the squares to be packed into an $m_1 \times m_2$ rectangle. The second column says whether or not the instance is satisfiable. The next three columns give measures of the search using the SMT solver without learning from [26]: the number of theory failures, the number of boolean assignments and the (cpu) time in milliseconds (using the statistics predicate with first argument runtime; time given is that for the fastest of five runs). The next three columns give the same measures for the solver with theory learning and the final three for the solver with symmetry breaking theory learning. Timeout was set to 60 minutes. The fastest time for each benchmark is highlighted.

As a further comparison against off-the-shelf solvers, Table 2 gives similar data for a selection of the benchmarks (the largest three satisfiable problems, plus the slowest unsatisfiable problem for $n \in \{12, 13, 14, 15\}$) solving using the CVC3 (whose integer decision procedure is based on the Omega test [24]) and CVC4 (whose integer decision procudure is based on mixed integer linear programming) [3] SMT solvers.

6.1 Discussion

Table 1 shows that the number of theory fails decreases across the columns from left to right; this is expected since potential theory fails are blocked by clauses learnt from previous theory fails. The decrease in theory fails is significant and indicates that theory learning and symmetry breaking theory learning are working well.

Adding theory learning pushes failure from the theory component to the boolean component, so the second column shows an increase in boolean assignment (it also includes the state restoration overhead). With the reduction in search space achieved by using symmetry breaking theory learning the number of boolean assignments uniformly decreases.

In terms of time, the solver with symmetry breaking theory learning is the fastest in all but three benchmarks. Two of these have small search spaces and the overhead of constructing the symmetric constraints does not pay. The other is (16, 28, 54), where the search space is reduced (indicated by the smaller values for theory fails and boolean assignments), but despite this the additional learned inequalities are slowing the solver.

The comparison against off-the-shelf solvers given in Table 2 firstly shows that the differing underlying techniques used by the theory solver component leads to varying results. The symmetry breaking theory learning solver has considerably fewer theory failures than the off-the-shelf solvers for all examples. The results for CVC3 show that the underlying SAT solver performs fewer boolean assignments than the Prolog coded solver used is this work, whilst the results for CVC4 show that it uses many more boolean assignments. In terms of speed, CVC4 is fastest for all but one example reflecting the underlying speed of its operations, whilst CVC3 and the symmetry breaking theory learning solver vary considerably in comparison across the benchmarks.

7 RELATED WORK

As already discussed, many SMT solvers use the Bellman-Ford single source shortest path algorithm for integer differences. Other approaches have been taken, for example [9] where simplex is used.

In solving the consecutive squares problem, [16] uses a dedicated solver for the problem, whilst [27] uses the off-the-shelf SICStus CLP(FD) solver arguing that this has all the machinery to realise a good model for the problem. Recently, when considering compilation of tabled constraints into SAT instances, [1] gives a model of the problem using tabled constraints (similar to that used in this paper); the largest non-trivial examples considered have $n = 14$ and optimal solutions are not addressed.

Compiling constraint problems into SAT instances has recently attracted attention, with [19] giving compilation strategies for finite domain constraints and the Sugar and Azucar finite domain solvers [29, 30] using SAT as their search engine. Modelling languages such as MiniZinc aim to decouple modelling and search, so that the constraint model can be compiled for a variety of different solvers, including SAT [21, 28].

Symmetry breaking has long been acknowledged as an important problem in constraint solving [11]. More recently, constraint solvers incorporating learning have been developed, such as Chuffed [6, 7]. The current work can be viewed as a learning constraint solver over the class of integer difference constraints.

8 CONCLUSIONS

This paper has shown how theory learning can be added to a general purpose declarative SMT solver over quantifier-free integer difference logic. In addition it has shown how such a solver can be enhanced with symmetry breaking theory learning, in a problem specific context, which promises to be a powerful new tactic. The white box nature of the solver allows this kind of novel specialisation to be integrated with ease.

Table 1: Benchmarking on the consecutive squares packing problem

instance	result	core			learn			learn + sym		
(n, m_1, m_2)		th	bool	time	th	bool	time	th	bool	time
(10,15,27)	SAT	0	15	120	0	15	**30**	0	15	**30**
(10,20,20)	UNSAT	66	158	520	25	289	280	14	225	**210**
(10,21,19)	UNSAT	76	158	680	41	356	440	22	258	**250**
(10,22,18)	UNSAT	50	98	360	32	231	340	18	170	**190**
(10,23,17)	UNSAT	50	98	520	40	332	450	24	242	**330**
(11,19,27)	SAT	148	323	1310	74	939	970	43	671	**790**
(11,23,23)	UNSAT	66	158	530	25	289	280	14	225	**200**
(11,24,22)	UNSAT	82	182	700	42	398	420	22	290	**280**
(11,25,21)	UNSAT	106	214	1040	56	497	550	26	340	**340**
(11,26,20)	SAT	83	183	940	68	727	830	40	534	**700**
(11,27,19)	SAT	109	239	1240	61	759	820	35	532	**580**
(12,23,29)	SAT	37	109	**640**	32	690	920	26	576	730
(12,26,26)	UNSAT	66	158	590	25	289	450	14	225	**250**
(12,27,25)	UNSAT	82	182	840	42	398	480	22	290	**350**
(12,28,24)	UNSAT	124	256	1370	58	565	730	26	386	**510**
(12,29,23)	SAT	244	509	3130	153	2273	2770	80	1486	**1850**
(12,30,22)	UNSAT	1066	2218	11680	186	3879	4220	63	2759	**2830**
(12,31,21)	UNSAT	1408	2986	16890	276	6074	6840	92	3977	**4460**
(13,22,38)	SAT	401	816	6200	165	2967	4570	67	1629	**2440**
(13,29,29)	UNSAT	66	158	710	25	289	440	14	225	**290**
(13,30,28)	UNSAT	82	182	1010	42	398	580	22	290	**400**
(13,30,30)	SAT	6	40	340	6	95	250	5	81	**150**
(13,31,27)	UNSAT	124	256	1910	58	565	1060	26	386	**540**
(13,32,26)	UNSAT	1782	3834	26140	501	10320	14080	139	5538	**6990**
(13,33,25)	UNSAT	4265	9456	58820	451	14532	16830	92	10380	**11300**
(14,23,45)	SAT	98	214	2150	64	1182	2650	35	731	**1730**
(14,32,32)	UNSAT	66	158	830	25	289	450	14	225	**340**
(14,33,31)	UNSAT	82	182	1200	42	398	680	22	290	**470**
(14,33,33)	SAT	8	45	390	8	145	300	7	131	**270**
(14,34,30)	UNSAT	124	256	2270	58	565	980	26	386	**630**
(14,34,32)	SAT	50	128	1190	22	356	810	17	321	**600**
(14,35,29)	UNSAT	1740	3714	30680	486	9918	15540	132	5307	**7630**
(14,35,31)	SAT	35	96	900	32	484	930	27	442	**810**
(14,36,30)	SAT	42	110	1190	38	716	1390	31	599	**1050**
(14,37,29)	SAT	116	266	2130	66	1068	1830	37	742	**1170**
(14,38,28)	SAT	3432	7991	59930	383	13615	18980	126	9795	**13320**
(14,39,27)	SAT	4888	11047	76800	600	20134	28600	176	13643	**18890**
(14,40,26)	UNSAT	7496	16334	112960	925	29425	43040	234	19379	**27180**
(14,41,25)	UNSAT	5448	11542	86290	772	22982	34600	233	14923	**21940**
(15,23,55)	SAT	204	431	4720	109	2088	5690	45	1066	**4170**
(15,36,36)	SAT	14	57	**600**	14	330	740	12	286	720
(15,37,35)	SAT	52	135	1480	24	382	820	19	347	**760**
(15,38,34)	SAT	296	644	6290	114	2912	6120	73	2041	**4380**
(15,39,33)	SAT	503	1105	11470	233	4746	9730	112	2886	**6110**
(15,40,32)	SAT	2685	6060	56640	351	10936	17350	118	7813	**11980**
(15,41,31)	UNSAT	13997	33374	287770	783	44687	65580	204	36258	**51880**
(15,42,30)	UNSAT	12655	29654	232960	769	40713	60030	192	32267	**46110**
(15,43,29)	UNSAT	9059	20794	144650	634	28627	42680	161	22614	**32280**
(16,27,56)	SAT	8823	17788	209770	1557	48118	112410	286	23014	**64510**
(16,28,54)	SAT	96322	195088	2441900	3421	265286	**521430**	471	203966	572230
(17,39,46)	SAT	12338	29224	358030	1297	55636	119120	383	37481	**77680**
(18,31,69)	SAT	6847	14715	220530	1100	41968	111610	230	20130	**59140**

Table 2: Benchmarking on consecutive squares for CVC

		CVC3			CVC4		
instance	result	th	bool	time	th	bool	time
(12,31,21)	UNSAT	1046	2404	6354	5980	30716	3620
(13,33,25)	UNSAT	549	2342	16988	5015	24533	4194
(14,40,26)	UNSAT	1753	7819	48507	7863	41365	6879
(15,41,31)	UNSAT	863	6186	38665	8505	60134	9134
(16,28,54)	SAT	9872	35771	262132	25280	141959	32919
(17,39,46)	SAT	664	1941	4595	20243	143473	17249
(18,31,69)	SAT	8820	42261	526683	33781	175353	52735

The solver has been applied to the consecutive squares packing problem and the results compare well to an approach using a similar straightforward model [1]. Results for constraint solving are often highly model dependent and approaches with more sophisticated modelling [16, 27] solve more problems in this class. Future work is to consider how such models might interact with the restricted syntax of an SMT solver over integer differences.

A weakness of the solver is the state restoration approach to learning that brings with it an overhead proportional to the number of theory fails. However, as the problem size increases this number drops as a proportion of the total number of fails and the overhead becomes less problematic.

Symmetry breaking is a well recognised technique in constraint modelling and the symmetry breaking theory learning approach to SMT solving should be applicable to a wide range of problems as well as the one tackled in this paper. This paper has taken a problem specific approach, but future work would allow classes of symmetries to be identified and broken. The comparison with CVC3 and CVC4 shows that symmetry breaking theory learning can reduce the number of theory failures considerably compared to standard SMT solvers. Considering the limited range of heuristics applied in the underlying Prolog SAT solver the comparison of the solver times is pleasing. The symmetric clauses learned can often block failures that would never be encountered and this is in part the cause of the modest improvement in times compared to the learning only solver. Improving learning so that only the most effective clauses are learned and retained is also future work.

In conclusion, a powerful, flexible white box SMT solver over integer difference constraints has been coded in only a few hundred lines of Prolog code.

Acknowledgments

The authors thank Peter Nightingale for useful discussions. The first author would like to thank the University of St Andrews for support during a sabbatical and Dagstuhl seminar 14351 on decision procedures and abstract interpretation.

REFERENCES

[1] O. Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Exploiting Short Supports for Improving Encoding of Arbitrary Constraints into SAT. In *Constraint Programming*, volume 9892 of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2016.

[2] J. Alglave, D. Kroening, and M. Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[4] C. Baykan and M. Fox. Spatial Synthesis by Disjunctive Constraint Satisfaction. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 11(4):245–262, 1997.

[5] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[6] G. Chu. *Improving Combinatorial Optimization.* PhD thesis, University of Melbourne, 2011.

[7] G. Chu, M. Garcia de la Banda, C. Mears, and P. J. Stuckey. Symmetries, Almost Symmetries, and Lazy Clause Generation. *Constraints*, 19(4):434–462, 2014.

[8] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[9] B. Dutetre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[10] R. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962.

[11] I. P. Gent and B. M. Smith. Symmetry Breaking in Constraint Programming. In *European Conference on Artificial Intelligence*, pages 599–603. IOS Press, 2000.

[12] J. M. Howe and A. King. Positive Boolean Functions as Multiheaded Clauses. In *International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 2001.

[13] J. M. Howe and A. King. A Pearl on SAT Solving in Prolog. In *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 165–174. Springer, 2010.

[14] J. M. Howe and A. King. A Pearl on SAT and SMT Solving in Prolog. *Theoretical Computer Science*, 435:43–55, 2012.

[15] E. Huang and R. E. Korf. New Improvements in Optimal Rectangle Packing. In *International Joint Conference on Artificial Intelligence*, pages 511–516. AAAI Press, 2009.

[16] E. Huang and R. E. Korf. Optimal Rectangle Packing: An Absolute Placement Approach. *Journal of Artificial Intelligence Research*, 46:47–87, 2012.

[17] R. E. Korf. Optimal Rectangle Packing: New Results. In *International Conference on Automated Planning and Scheduling*, pages 142–149. AAAI, 2004.

[18] D. Kroening and O. Strichman. *Decision Procedures.* Springer, 2008.

[19] A. Metodi and M. Codish. Compiling finite domain constraints to SAT with BEE. *Theory and Practice of Logic Programming*, 12(4-5):465–483, 2012.

[20] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535. ACM Press, 2001.

[21] 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*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[22] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In *Computer-Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, 2005.

[23] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[24] W. Pugh. A Practical Algorithm for Exact Dependency Analysis. *Communications of the ACM*, 35(8), 1992.

[25] E. Robbins, J. M. Howe, and A. King. Theory Propagation and Rational-trees. In *Principles and Practice of Declarative Programming*, pages 193–204. ACM Press, 2013.

[26] E. Robbins, J. M. Howe, and A. King. Theory Propagation and Reification. *Science of Computer Programming*, 111:3–22, 2015.

[27] H. Simonis and B. O'Sullivan. Search Strategies for Rectangle Packing. In *Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2008.

[28] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer. The MiniZinc challenge 2008-2013. *AI Magazine*, 35(2):55–60, 2014.

[29] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling Finite Linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[30] T. Tanjo, N. Tamura, and M. Banbara. Azucar: A SAT-Based CSP Solver Using Compact Order Encoding. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 456–462. Springer, 2012.

[31] C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In *European Conference on Logics in Artificial Intelligence*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 308–319. Springer, 2002.

[32] S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[33] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2002.