Explaining alldifferent

Nicholas Downing **Thibaut Feydy** **Peter J. Stuckey**

National ICT Australia* and the University of Melbourne, Victoria, Australia
Email: {ndowning@students.,tfeydy@,pjs@}csse.unimelb.edu.au

Abstract

Lazy clause generation is a powerful approach to reducing search in constraint programming. For use in a lazy clause generation solver, global constraints must be extended to explain themselves. Alternatively they can be decomposed into simpler constraints which already have explanation capability. In this paper we examine different propagation mechanisms for the *alldifferent* constraint, and show how they can be extended to explain themselves. We compare the different explaining implementations of *alldifferent* on a variety of problems to determine how explanation changes the trade-offs for propagaton. The combination of global *alldifferent* propagators with explanation leads to a state-of-the-art constraint programming solution to problems involving *alldifferent*.

1 Introduction

Lazy clause generation (Ohrimenko et al. 2009) is a hybrid approach to constraint solving that combines features of finite domain propagation and Boolean satisfiability. Finite domain propagation is instrumented to record the reasons for each propagation step. This creates an implication graph like that built by a SAT solver, which may be used to create efficient nogoods that record the reasons for failure. These learnt nogoods can be propagated efficiently using SAT unit propagation technology.

The resulting hybrid system combines some of the advantages of finite domain constraint programming (CP): high level model and programmable search; with some of the advantages of SAT solvers: reduced search by nogood creation, and effective autonomous search using variable activities. Lazy clause generation provides state of the art solutions to a number of combinatorial optimization problems.

The *alldifferent* global constraint is one of the most common global constraints appearing in constraint programming models. *alldifferent*$(x_1, \ldots, x_n)$ requires that each of the variables $x_1, \ldots, x_n$ takes a different value. It is logically equivalent to $\bigwedge_{1 \leq i < j \leq n} x_i \neq x_j$. It succinctly encodes assignment subproblems occurring in a model. Such assignment subproblems occur frequently in real-world scheduling and rostering problems, such as the INSN_SCHED, TALENT_SCHED and SOCIAL_GOLFER problems discussed in the experiments section of this paper.

Various implementations for the *alldifferent* constraint are available, some relying on specific propagation algorithms that enforce VALUE, BOUNDS and DOMAIN consistency, and some relying on decomposition of *alldifferent* into simpler constraints.

Learning changes the trade-offs for propagation. It may well be worth spending more time calculating stronger propagation, if the results can be reused elsewhere using learning, thus amortizing the cost over multiple uses. Conversely, it may be worth spending less time on propagation, if we can rely on the globality of learning to learn the stronger consequences of a constraint that are useful to the search in any case. Hence it is worthwhile studying what form of propagation of *alldifferent* is best for a learning solver.

Propagation algorithms for *alldifferent* have been quite well-studied, see the survey of van Hoeve (2001) for details. But until recently, global *alldifferent* propagators have not been used in learning solvers. Katsirelos (2008) describes a method for implementing the domain-consistent algorithm (Régin 1994) with explanations for use in a learning solver, but without experiments. We present for the first time an implementation of the method (with slight enhancements), and also we describe and implement for the first time an explained version of the bounds-consistent propagator (Lopez-Ortiz et al. 2003).

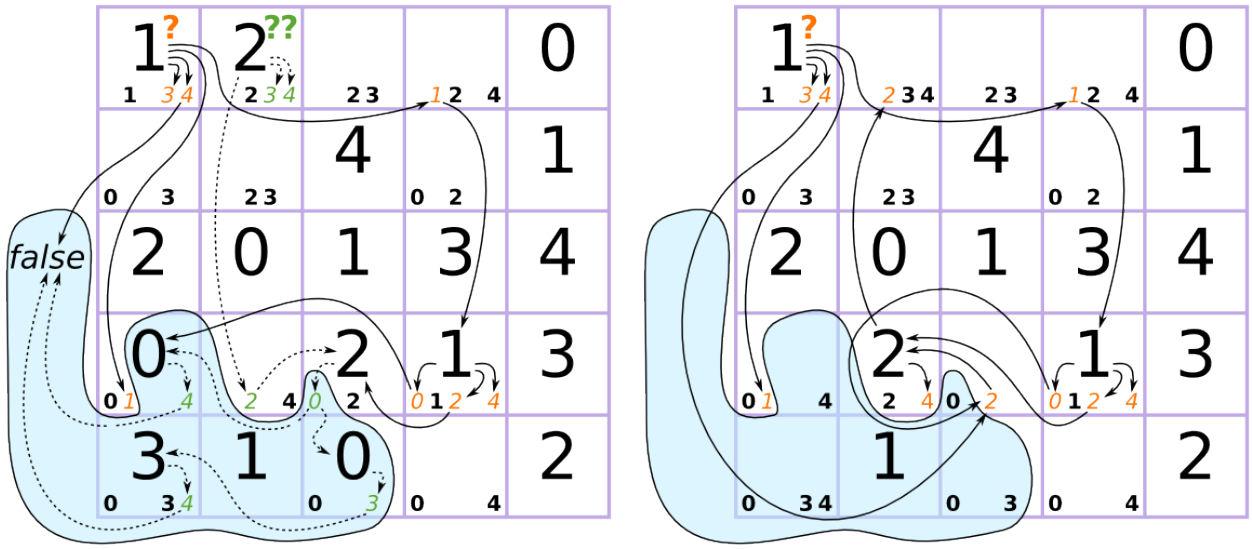
As well as the new propagators a further important contribution is a comprehensive suite of experiments using over 4000 hours of computer time to compare the learning vs. non-learning and global vs. decomposition approaches over a large set of structured problems that use *alldifferent*, and to find the best search strategy and solver combination for each problem, with comparison to previous state-of-the-art CP approaches to verify our results. We find learning to be enormously beneficial, so much so that new harder problems needed to be created to exercise our propagators, and that using the correct global or decomposed constraint is important on most models.

2 Lazy clause generation

We give a brief description of propagation-based solving and lazy clause generation, for more details see Ohrimenko et al. (2009). We consider constraint satisfaction problems (CSPs), consisting of constraints over integer variables $x_1, \ldots, x_n$, each with a given finite domain $D_{\text{orig}}(x_i)$. A feasible solution is a valuation to the variables such that each x_i is within its allowable domain and all constraints are satisfied.

A propagation solver maintains a domain restriction $D(x_i) \subseteq D_{\text{orig}}(x_i)$ for each variable and consid-

*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.



(a) Pre analysis (b) Post analysis

Figure 1: Conflict analysis on the implication graph of a 5×5 QG_COMPLETION problem

ers only solutions that lie within $D(x_1) \times \cdots \times D(x_n)$. Solving interleaves propagation, which repeatedly applies propagators to remove unsupported values, and search which splits the domain of some variable and considers the resulting sub-problems. This continues until all variables are fixed (success) or failure is detected (backtrack and try another subproblem).

Lazy clause generation is implemented in the above framework by defining an alternative model for the domains $D(x_i)$, which is maintained simultaneously. Specifically, Boolean variables are introduced for each potential value of a variable, named $[x_i = j]$ and $[x_i \geq j]$. Negating them gives the opposite, $[x_i \neq j]$ and $[x_i \leq j-1]$. Fixing such a *literal* modifies domains to make the corresponding fact true in $D(x_i)$ and vice versa. Hence these literals give an alternate Boolean representation of the domain, which can support SAT reasoning.

In a lazy clause generation solver, the actions of propagators (and search) to change domains are recorded in an *implication graph* over the literals. Whenever a propagator changes a domain it must *explain* how the change occurred in terms of literals, that is, each literal l that is made true must be explained by a clause $L \rightarrow l$ where L is a (set or) conjunction of literals. When the propagator causes failure it must explain the failure as a *nogood*, $L \rightarrow \mathit{false}$, with L a conjunction of literals which cannot hold simultaneously. Conflict analysis reduces L to a form suitable to use as a clausal propagator to avoid repeating the same search (Moskewicz et al. 2001).

Example 2.1 (conflict analysis) *Figure 1a shows a simple 5×5 Quasigroup Completion problem. Initially 11 of the 25 cells are filled in. The learning solver attempts to fill in the remaining 14 cells in such a way that the same digit does not appear twice in any row or column. In the (initially) blank cells is depicted a domain representation '1 2 3 4 5' which shows the possible values for the cell.*

A value can be removed from a domain (shown in lightweight italics) when that value is assigned to another cell in the same row or column. A value can be assigned to a cell when (i) the domain of the cell has been reduced to a single possibility, or (ii) it is the only cell in the same row (or column) that can take this value. Such reasoning is depicted graphically by arrows showing, for each assignment/removal, its preconditions (a set of previous assignments/removals which must hold simultaneously).

Since the original problem was at fixed-point with respect to the above reasoning, search had to 'pencil in' the value 1 in the top-left corner, depicted '?' (first decision level). Resulting implications are shown as solid arrows. Fixed-point being reached again (under this assumption), search pencilled in the value 2 in the next cell, depicted '??' (second decision level). Resulting implications are shown as dotted arrows, to show they occurred at the second level.

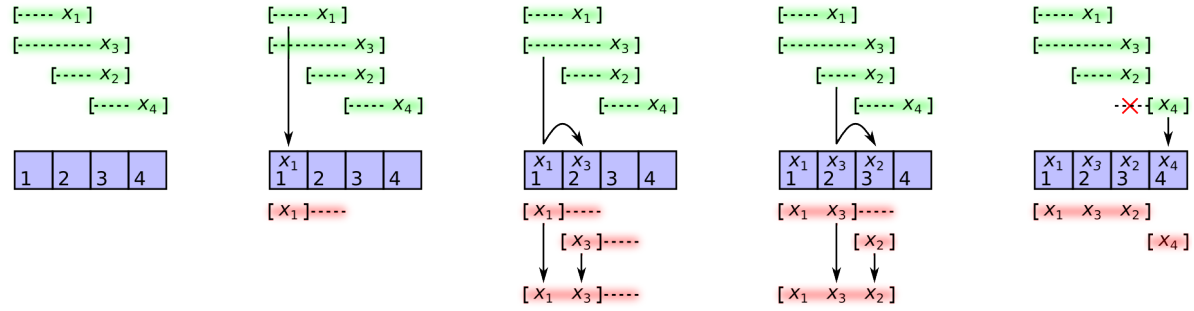
These assumptions (1 and 2 in the initial cells), lead to a conflict because no cell in the first column can now take the value 4. The resulting conflict clause $L \rightarrow \mathit{false}$ is depicted graphically. L simply expresses a rule of the puzzle and hence is useless as a learnt clause, so we have to look back in the implication graph to see the underlying causes of the conflict.

We use 1UIP conflict analysis (Moskewicz et al. 2001) to find the 'cut', depicted in the figures, which (i) contains the conflict, (ii) is as small as possible, and (iii) traces the conflict back to a single precondition at the current decision level, here $[x_{43} = 2]$. Observe that there are 3 implication arrows entering the cut (of which only one can be dotted). It is easy to see that only these preconditions need to exist simultaneously for failure to be inevitable.

The learnt nogood is simply a list of preconditions to the cut, here $[x_{11} \neq 4] \wedge [x_{44} \neq 0] \wedge [x_{43} = 2]$. After undoing all work at decision level 2, this new clause must propagate, as $[x_{11} \neq 4] \wedge [x_{44} \neq 0] \rightarrow [x_{43} \neq 2]$, which removes the immediate reason for the conflict. Inevitably this also propagates back as shown in Figure 1b, to undo the bad decision marked '??'. Due to this clause learning mechanism, search never makes the same mistake again.

3 Hall Sets

The *alldifferent* constraint requires that each argument takes a different value. The key to all propagation algorithms for *alldifferent* is the detection of *Hall sets* (Hall 1935). Given a constraint *alldifferent*$(x_1, \ldots, x_n)$, $H \subseteq \{1, \ldots, n\}$ is a Hall set if $|H| \geq |V|$ where $V = \cup_{h \in H} D(x_h)$. If the inequality



(a) Initial domains (b) Singleton interval (c) Merging intervals (d) Finding Hall interval (e) Pruning lower bound

Figure 2: Example of bounds-consistent propagator execution for pruning lower bounds

holds strictly, that is $|H| > |V|$, then the constraint is unsatisfiable. If it holds as an equality, $|H| = |V|$, then no variable $x_i, i \notin H$ can take a value from V.

Example 3.1 (Hall sets) *Given $x_1 \in \{1,2\}$, $x_2 \in \{1,3\}$ and $x_3 \in \{1,3\}$ are all different, $H = \{2,3\}$ is a Hall set with $V = \{1,3\}$. Since 3 different variable-values can't fit in a domain containing only 2 values, x_1 must be outside this domain, that is $x_1 \neq 1$.*

In the next sections we examine various propagators for *alldifferent* and how they can be extended to explain their propagations. The explanation clauses are essentially descriptions of the well-known conditions for pruning. Usually these clauses also suffice to describe failure (because they wake up implicit clauses requiring domains to be non-empty) but in some cases explicit failure nogoods can also be produced.

4 Global value-consistent propagator

The simplest form of *alldifferent*$(x_1, \ldots, x_n)$ is a decomposition that enforces $x_i \neq x_j$ for all $1 \leq i < j \leq n$. Let $E = \cup_{i=1}^{n} D_{orig}(x_i)$ be the union of the domains of all variables appearing in the *alldifferent* constraint. An equivalent decomposition based on a *linear* constraint is $\sum_{i=1}^{n} bool2int([x_i = v]) \leq 1$ for all $v \in E$. Since the size of the decomposition is $O(n|E|)$ we implement this as a single global propagator that wakes upon variable fixing, i.e. when $D(x_h) = \{v\}$ for some h, v, it prunes all $D(x_i), i \neq v$ with explanation

$$[x_h = v] \rightarrow [x_i \neq v].$$

The complexity of this propagator is $O(n|E|)$.

When $|E| = n$, there are no spare values and we also enforce the clauses $\bigvee_{i=1}^{n} [x_i = v]$ for all $v \in E$, equivalent to changing the upper bound of 1 to equality with 1 in the above *linear* constraints. These clauses are standard in the SAT community (e.g. in the CNF output of Gomes's *lsencode* generator for QG_COMPLETION problems) but their importance isn't widely recognised for CSPs.

5 Global bounds-consistent propagator

Given the constraint *alldifferent*$(x_1, \ldots, x_n)$ over domains $D(x_1)$, $\ldots$, $D(x_n)$, bounds consistency ensures for each x_i, both $a_i = \min(D(x_i))$ and $b_i = \max(D(x_i))$ have a support over $\Pi_{j \neq i} a_j..b_j$, i.e. a solution to the constraint relaxed to range domains, which uses the value $x_i = a_i$ or b_i.

The best bounds-consistent *alldifferent* propagator is by Lopez-Ortiz et al. (2003). It rests on two key observations, (i) a solution to the constraint may be found greedily, if one exists, by allocating each variable its minimum possible value, treating variables in the order most- to least-constrained; and (ii) a union-find data structure (Tarjan 1975) can efficiently encode the dependencies between interval domains, to build Hall intervals incrementally and inform us when a complete Hall interval has been identified.

Example 5.1 (pruning bounds) *Suppose $x_1 \in 1..2$, $x_2 \in 2..3$, $x_3 \in 1..3$ and $x_4 \in 3..4$. These intervals, along with a representation of the overall domain 1..4, are shown in Figure 2a. Initially, all cells of the domain representation are unoccupied. The variables are sorted in order of increasing upper bound, which is the criterion for constrainedness, since for example it would not make sense to allocate $x_3 = 1$, $x_2 = 2$ and then find all possibilities for x_1 occupied.*

The first variable to allocate is $x_1 = 1$, shown in Figure 2b. A new singleton interval is created in the union-find data structure, shown below the domain-representation. The endpoints of the interval are indicated by [ ], whereas the upper bound of the contained variable extends further, shown shaded and dotted. At present the new interval is not Hall; when its right-hand endpoint increases to take in the shaded region then it will become a Hall interval.

Referring to Figure 2c, we next allocate $x_3 = 2$. Because we had to jump over the value 1 to allocate x_3, the interval containing 1 is merged in the union-find data structure with the newly created interval. This merging preserves the invariant that for each interval in the data structure, a value in the interval can be freed up if and only if *one of the variables in the interval has its bounds relaxed. The merged interval is still not a Hall interval since it contains x_3 which has the highest upper bound, 3, shown.*

Then, allocating $x_2 = 3$ discovers a Hall interval (Figure 2d). Since the value 2 was passed over, the corresponding interval is merged with the new interval, including the value 1 which could only affect the new allocation indirectly. The upper bound of the newly merged interval has caught up with the upper bounds of the variables in it, so the interval is marked 'Hall'. Then when processing x_4 (Figure 2e), we notice its lower bound falls into a Hall interval, and should be pruned before attempting any allocation.

Due to the order of discovering Hall intervals relative to the processing of variables, the algorithm as described above will only prune lower bounds. We use the original algorithm with minimal change, which includes a second pass of recomputing all Hall intervals to prune the upper bounds, though there is no reason in principle why the information discovered on the first pass should not be reused to save effort.

The algorithm of Lopez-Ortiz et al. (2003) is $O(n \log n)$ to sort the variables, plus $O(n \log n)$ to

scan variables and construct/maintain their (specialized) union-find data structures, overall $O(n \log n)$.

The only changes we made to the algorithm were (i) to use insertion sort for the variables at cost $O(n^2)$, in practice this takes only linear time since the variables are already sorted, and the algorithm is dominated by the union-find operations hence still $O(n \log n)$; and (ii) to collect the set H, required for explanations. Given H with $V = a..b$ we can explain the increased lower bound for a variable $x_i \notin H$ as

$$[x_i \geq a] \wedge \bigwedge_{h \in H} ([x_h \geq a] \wedge [x_h \leq b]) \rightarrow [x_i \geq b + 1].$$

This requires $O(n)$ literals per explanation.

6 Global domain-consistent propagator

For *alldifferent*$(x_1, \ldots, x_n)$ over domains $D(x_1), \ldots, D(x_n)$, domain consistency ensures that for each x_i, each value in $D(x_i)$ has a support, i.e. a solution to the entire constraint, which uses the value.

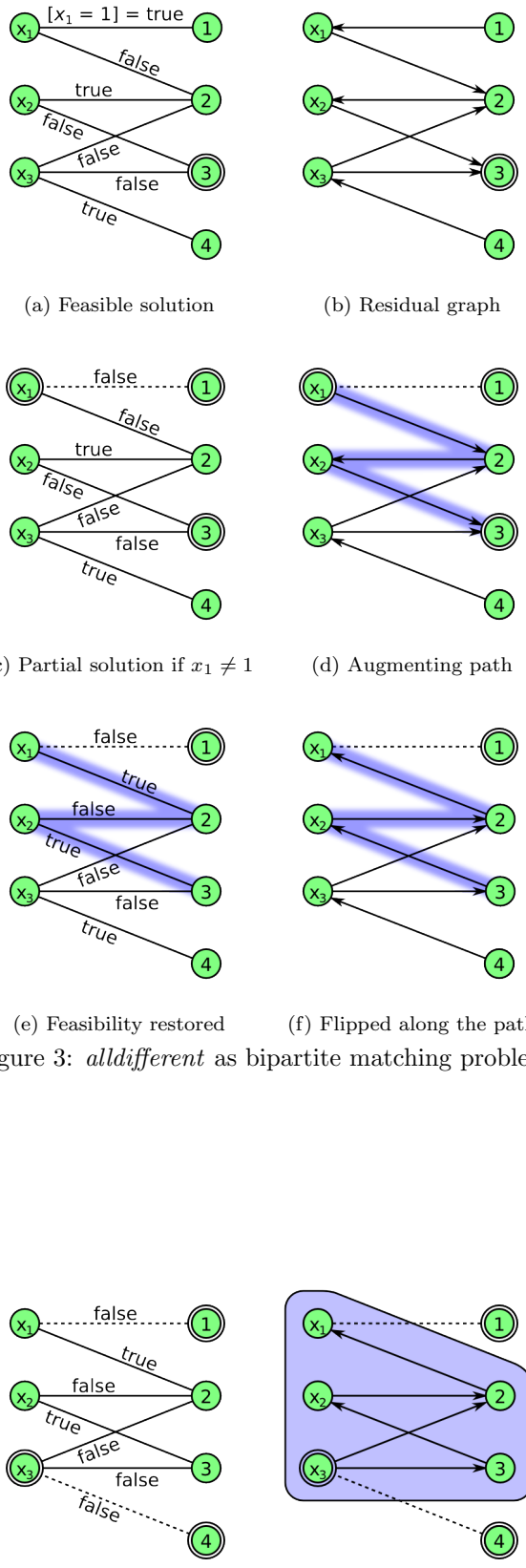
The best domain-consistent *alldifferent* propagator is by Régin (1994) with improvements by Gent et al. (2008). For *alldifferent* as bipartite graph matching problem, we can find a feasible solution (or prove that none exists) using Ford & Fulkerson's (1956) *augmenting paths* algorithm. The arcs (variable-value pairs) used in this solution are obviously supported. Support for another arc depends on whether there exists an *augmenting cycle* containing it, which we can check efficiently using Tarjan's (1972) *strongly connected components* (SCC) algorithm.

Example 6.1 (augmenting paths) *Suppose $x_1 \in \{1, 2\}$, $x_2 \in \{2, 3\}$, $x_3 \in \{2, 3, 4\}$. Then a feasible solution is $x_1 = 1$, $x_2 = 2$, $x_3 = 4$, illustrated in Figure 3a. The corresponding residual graph, shown in Figure 3b, has a forward arc where a variable/value pair could be added to the matching or a backward arc where a variable/value pair could be removed.*

Now suppose $x_1 \neq 1$. The alldifferent propagator wakes up and removes the illegal assignment from the graph as shown in Figure 3c, where unmatched nodes are double-circled. To repair the matching, a path is found in the residual graph (Figure 3d), from the unmatched variable x_1 to an unmatched value 3. Augmenting along this path means adding to the matching when traversing forward arcs or removing for backward arcs (Figures 3e and 3f), so that x_1 becomes 2 and x_2 moves onto 3 in the proposed solution.

The actions of Régin's propagator may be explained by Hall sets. When Régin's propagator fails or prunes we can easily identify the failure set or Hall set which caused it. For infeasibility, the set of nodes searched for an augmenting path (the *cut*) consists of $H \cup V$ and is a failure set as necessarily $|H| > |V|$. For pruning, the most recently discovered SCC consists of $H \cup V$ and is a Hall set. We instrumented the propagator to use this knowledge to explain its failures and prunings. Note that we use SCC-splitting (Gent et al. 2008), and we generate explanations lazily.

Example 6.2 (failure) *Continuing example 6.1, suppose $x_1 \neq 1$ and also $x_3 \neq 4$. When the propagator wakes up it can repair x_1 as shown previously (Figure 4a), but there is no augmenting path from x_3 to an unused value, which the algorithm proves by searching the nodes indicated in Figure 4b before concluding that no further search is possible.*



(a) Feasible solution (b) Residual graph

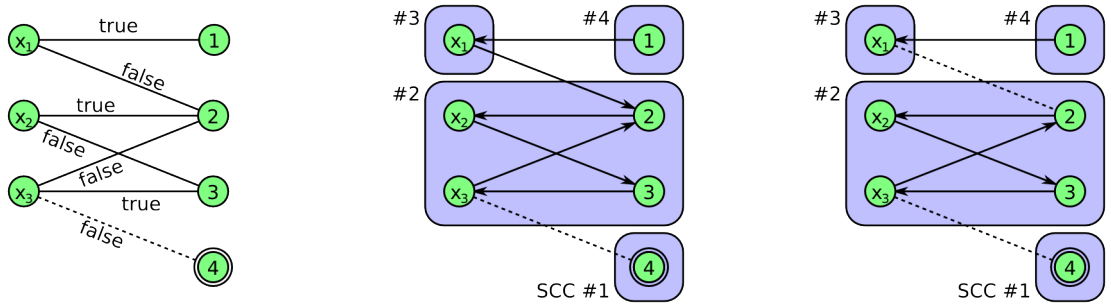
(c) Partial solution if $x_1 \neq 1$ (d) Augmenting path

(e) Feasibility restored (f) Flipped along the path

Figure 3: *alldifferent* as bipartite matching problem

(a) Partial solution if $x_3 \neq 4$ (b) Set of nodes searched

Figure 4: Deriving an explanation for failure



(a) Feasible solution if $x_3 \neq 4$ (b) SCC connectivity (c) After pruning $[x_1 = 2]$

Figure 5: Isolating SCCs in depth-first manner while deriving explanations for the prunings

The resulting cut-set, partitioned into variables $H = \{1, 2, 3\}$ and values $V = \{2, 3\}$, proves infeasibility since $|H| > |V|$, and suggests the failure nogood $\bigwedge_{h\in\{1,2,3\}}(x_h \in \{2, 3\})$. Since we do not have literals to express that $x_h \in \{2, 3\}$, we use an equivalent clausal representation $x_h \neq 1 \wedge x_h \neq 4$. By removing the literals that are false in the original domains, we obtain the nogood $[x_1 \neq 1] \wedge [x_3 \neq 4] \rightarrow$ false. This final nogood is simply the list of dotted arcs leaving the cut; this is intuitive since the search stops precisely because those arcs are dotted.

Example 6.3 (pruning) *Alternatively, suppose $x_3 \neq 4$ while $x_1 = 1$ remains possible. The propagator wakes up and repairs x_3, resulting in the feasible solution of Figure 5a. Using Tarjan's algorithm it determines the SCCs of the resulting residual graph, shown in Figure 5b. The arc $x_1 = 2$ crosses SCCs, so can't be augmented (it is not part of any augmenting cycle in the residual graph), and may be removed.*

The target of the arc being pruned is SCC #2 which gives the Hall set $H = \{2, 3\}$, $V = \{2, 3\}$ as evidence for the pruning, suggesting the explanation $\bigwedge_{h\in\{2,3\}}(x_h \in \{2, 3\}) \rightarrow [x_1 \neq 1]$. Once again we can express this as the list of dotted arcs leaving the SCC, giving $[x_3 \neq 4] \rightarrow [x_1 \neq 1]$.

For the sake of simplicity we glossed over the distinction between augmenting paths and cycles. An arc may be augmented if it is part of any augmenting *path*, whereas the SCC-algorithm can only eliminate its appearing in an augmenting *cycle*. We get around this difficulty with a slight modification to Tarjan's algorithm which makes used values reachable from unused values, so that freeing up a value by taking another value is considered to be a cycle.

In the worst case the edges are removed from the graph one by one, so there are $n|E|$ propagator executions, each computing a single augmenting path at cost $O(n|E|)$ and re-running the SCC-algorithm at cost $O(n)$, so the cost is $O(n^2|E|^2)$ down a branch. In practice, repairing the matching is very fast (because it seldom explores the whole graph), and most time is spent in the SCC-algorithm.

Given H and V the explanation that we use for pruning values in $j \in V$ from $x_i \notin H$ is

$$\bigwedge_{h\in H, d\in E\setminus V} [x_h \neq d] \rightarrow [x_i \neq j] \qquad (1)$$

It requires $|H|(|E| - |V|)$, or $O(n|E|)$ literals per explanation in the worst case.

Our explanations are the same as Katsirelos's (2008) except that, our explanations based on the list of dotted arcs leaving an SCC are quite general, so we naturally deduce and propagate equalities, rather than just disequalities as Katsirelos does.

Example 6.4 (deducing equalities) *Pruning the arc from SCC #2 → #3 as described in Example 6.3 gives the residual graph of Figure 5c. Further pruning is possible: x_1 may be fixed to 1, using SCC #3 as evidence, yielding explanation $[x_1 \neq 2] \rightarrow [x_1 = 1]$.*

7 *alldifferent* by decomposition

alldifferent can also be implemented by decomposition into simpler constraints which already have explanation capability. The obvious decomposition is the conjunction of disequalities discussed in Section 4. A decompositions which prunes the same as the bounds-consistent global propagator is available (Bessiere et al. 2009*b*). There is no polynomially sized decomposition of *alldifferent* into clauses which enforces domain consistency (Bessiere et al. 2009*a*).

The attraction of decomposition is the ease of understanding, implementation and experimentation. We also cannot rule out that decompositions may perform better, by channelling the problem into more appropriate variables, or by making intermediate variables and implications available for conflict analysis which would otherwise have remained implicit.

A new decomposition of *alldifferent*, as a special case of *gcc* (Global Cardinality Constraint), was introduced by Feydy & Stuckey (2009). It is defined in *MiniZinc* (Nethercote et al. 2007) as follows:

```
predicate alldifferent_feydy_decomp(
                array[int] of var int: x) =
  let { int: L = lb_array(x),
        int: C = ub_array(x) + 1 - L,
        int: N = length(x),
        array[1..C] of var 0..1: c,
        array[0..C] of var 0..N: s } in
  s[0] = 0 /\ s[C] = N /\
  forall (i in 1..C) (
    s[i] = s[i - 1] + c[i] /\
    c[i] = sum (j in 1..N) (bool2int(x[j] = i)) /\
    s[i] = sum (j in 1..N) (bool2int(x[j] <= i)));
```

The new decomposition is efficient because it uses the literals $[x_i = v]$ and $[x_i \leq v]$, which are native in a lazy clause generation solver, as they are part of the integer variable encoding.

To see how this works consider the simplest case when there are no spare values. Then the constraints $s_i = s_{i-1} + c_i$ simplify to $c_i = 1$ and $s_i = i$. This gives the consistency level described in Section 4, plus the detection of Hall intervals aligned to the start or end of the domain interval $\min(E)..\max(E)$ where E is the union of the domains of the variables.

Example 7.1 *Suppose $x_1 \in 1..2$, $x_2 \in 1..3$, $x_3 \in 2..3$. No propagation is possible, e.g. considering the*

constraint $s_1 = \sum_{i=1}^{3} \mathit{bool2int}([x_i \leq 1] = 1$*, we find*

$$\begin{aligned}&\min(\mathit{bool2int}([x_1 \leq 1]))\\ &\quad + \max(\mathit{bool2int}([x_2 \leq 1]))\\ &\quad + \max(\mathit{bool2int}([x_3 \leq 1])) = 0 + 1 + 0 \geq 1\end{aligned}$$

which supports $\mathit{bool2int}([x_1 \leq 1]) = 0$*. But if the interval for* x_2 *becomes* $2..3$*, then the preceding test gives* $0 + 0 + 0$ *which is no longer* ≥ 1*, therefore* $\mathit{bool2int}([x_1 \leq 1]) = 0$ *is unsupported, and is pruned with explanation* $[x_2 \geq 2] \wedge [x_3 \geq 2] \rightarrow [x_1 \leq 1]$.

8 Experiments

We implemented the *alldifferent* constraint in *Chuffed*, a state-of-the-art lazy clause generation solver. Hardware was a cluster of Dell PowerEdge 1950 with 2 × 2.0 GHz Intel Quad Core Xeon E5405, 2×6MB Cache, and 16 GB RAM. Timeouts were 1800s, and each core was limited to 1 GB RAM in our experiments. Data files are available from `http://www.csse.unimelb.edu.au/~pjs/alldifferent`.

The *alldifferent* implementations we compare in *Chuffed* were VALUE, the global value consistent propagator described in Section 4; BOUNDS, the global bounds consistent propagator of Section 5; DOMAIN, the global domain consistent propagator of Section 6; and FEYDY, the *gcc*-based decomposition of Section 7. We also examined value-consistent *alldifferent* by decomposition to disequalities or *linear* constraints, and the bounds consistent decomposition of Bessiere et al. (2009*b*), but found them universally worse than the corresponding globals of equal propagation strength.

The search strategies were: IO, input order, an appropriate static search depending on the model; DWD, *dom/wdeg* search (Boussemart et al. 2004); and ACT, activity-based (VSIDS) search (Moskewicz et al. 2001). We use Luby restarts (Luby et al. 1993) for dynamic search strategies (DWD and ACT) if learning. Note that ACT is inapplicable without learning since it is the process of conflict analysis which collects activity counts, hence was only run with learning.

In the first experiment we take all benchmarks involving the global *alldifferent* constraint from the Third International CSP Solver Competition (CSP 2008) plus the CSP2008 QCP and QWH benchmarks which were only available in extensional form and had to be converted to use *alldifferent* directly. On these benchmarks the leading CSP2008 solvers were CPHYDRA, MISTRAL and SUGAR, and we compare against the published results of the competition, noting that they use an older Xeon architecture, but as they run at 3.0 GHz the performance should be comparable. We excluded the trivial PIGEONS instances, and certain BQWH instances where for some reason published results were not available. The CSP2008 solvers use their default strategy as in the published results, shown as IO in the table.

Table 1 reports the geometric mean of runtimes for the first experiment (using the timeout for timed-out instances), with the number of timeouts appearing as a superscript. For each model we show how many solved instances were unsatisfiable or satisfiable and how many were indeterminate as not solved by any solver. These latter instances aren't included in the runtime or timeouts statistics. In each block the solver with the fewest timeouts is highlighted, with ties broken by runtimes. Memory-outs were treated as timeouts (these occur on FEYDY only). Referring to Table 1 our solver is clearly far superior to the winners of the CSP2008 competition (PATAT is an exception which arises because *MiniZinc* produced a particularly poor decomposition for the $(x_0 \neq x_1) \vee (x_2 \neq x_3)$ constraints appearing in this model, a problem we did not address due to time constraints).

Since our runtimes were so small as to be barely measurable in most cases, we compiled a new set of much harder problems, based on the *MiniZinc* 1.1.6 benchmark suite (Nethercote et al. 2007) and the suite of Gent et al. (2008), with some additional models and additional harder instances. We compare on these models versus the state-of-the-art constraint programming system *Gecode* (Gecode Team 2006), running the same *MiniZinc* models as *Chuffed*. *Gecode* has won every *MiniZinc* Challenge (G12 Project 2010) run so far! The models are:

GOLOMB_RULER (prob006 in CSPLib (Gent & Walsh 1999)), $n = 8..11$ is a problem of placing n marks on a rule so that all the distances between the marks are distinct.

INSN_SCHED, instruction scheduling for single-issue pipelined CPUs, similar to Lopez-Ortiz et al. (2003). Instances were obtained by compiling MediaBench benchmarks with *gcc* 4.5.2, switches *-O3 -march=barcelona -fsched-verbose=5*, and taking all sequences with 250..999 instructions. These problems are interesting as the AMD Barcelona-core CPUs have instructions with various latencies. These CPUs are multiple-issue, requiring a *gcc* constraint, so we consider a hypothetical single-issue version of the CPU requiring only *alldifferent*. We omit redundant constraints, they can improve performance, but their number grows quadratically, hurting scalability.

KAKURO, a grid-based puzzle similar to a crossword but with a numeric grid and arithmetic clues. We used the grid generator at `http://www.perlmonks.org/?node_id=550884` to generate 10 puzzles of size 25×25 with coverage 50%. We use the redundant *alldifferent-sum* constraints of Simonis (2008), but not Simonis's *interact* constraints.

KNIGHTS_TOUR, finding a cyclic knight's tour of length 56, 58, ..., 64 on an 8×8 chessboard.

LANGFORD, Langford's number problem (prob024 in CSPLib) which is to sequence k sets of numbers $1..n$ such that each occurence of a number i is i numbers apart from the next i in the sequence. The selected instances are from $k = 2..4 \times n = 3..24$, taken from the *MiniZinc* benchmarks.

QG_COMPLETION, Quasigroup Completion (QCP) is given an $n \times n$ array where some cells are filled in with numbers from 1 to n, fill in the rest of the cells so that each row and each column contains the set of numbers $1..n$. We used Gomes's *lsencode* generator, `http://www.cs.cornell.edu/gomes/SOFT/lsencode-v1.1.tar.Z`, to generate 160 problems of sizes 30×30..45×45. An arbitrary random problem generated in this way is also usually too easy, so we used *picoSAT* 936 from `http://fmv.jku.at/picosat` to test the problems, keeping only those which were still being solved after 10 seconds on all of 10 randomized attempts. For example, on size 30×30 we generated 27855 instances to find 10 which were hard enough.

QG_EXISTENCE, Quasigroup Existence (prob003 in CSPLib) looks for a quasigroup of size n which satisfies various other criteria. We use sizes 8×8..13×13, variants QG3..7 × {idempotent, nonidempotent}, except that QG6..7 are always idempotent. Redundant constraints are from Colton & Miguel (2001).

SOCIAL_GOLFER (prob010 in CSPLib) is to schedule a golf tournament for $n \times m$ golfers over p weeks playing in groups of size m so that no pair of golfers plays twice in the same group. We use instances taken from `http:`

model		CSP2008, *solver*=				NOLEARN, *alldiff* =				LEARN, alldiff=			
unsat, sat, ?		CPHYDRA	MISTRAL	MISTRAL′	SUGAR	VALUE	BOUNDS	DOMAIN	FEYDY	VALUE	BOUNDS	DOMAIN	FEYDY
BQWH	IO	0.09s	0.16s	**0.08s**	1.29s	0.24s	0.53s	**0.00s**	0.01s	0.02s	0.04s	0.01s	0.02s
	DWD					0.02s	0.04s	0.00s	0.01s	0.01s	0.02s	0.00s	0.01s
0, 20, 0	ACT									0.02s	0.04s	**0.00s**	0.02s
COSTAS-	IO	6.28s[1]	5.42s[2]	**1.92s**[1]	21.23s[2]	1.49s[1]	3.12s[1]	8.16s[2]	12.23s[2]	5.34s[2]	9.94s[3]	11.38s[3]	24.97s[4]
ARRAY	DWD					**0.77s**[1]	1.60s[1]	2.18s[1]	38.16s[4]	**1.78s**[1]	1.25s[2]	2.31s[1]	17.51s[3]
0, 10, 1	ACT									5.24s[2]	11.34s[2]	15.68s[2]	19.19s[3]
LATIN-	IO	6.27s[5]	4.49s[5]	4.35s[5]	**2.09s**[1]	0.99s[5]	1.42s[5]	1.62s[5]	1.75s[5]	0.16s	0.49s[1]	0.55s[1]	2.18s[3]
SQUARE	DWD					**0.74s**[4]	1.09s[4]	1.32s[4]	1.78s[5]	0.11s[1]	0.27s[1]	0.49s	0.97s[2]
7, 3, 0	ACT									**0.00s**	0.01s	0.01s	0.03s
MAGIC-	IO	53.35s[9]	**11.22s**[6]	36.46s[9]	65.72s[9]	165.73s[14]	170.30s[14]	172.82s[14]	179.22s[14]	170.85s[14]	190.90s[14]	172.51s[14]	194.31s[14]
SQUARE	DWD					**103.31s**[13]	121.10s[13]	120.84s[13]	130.61s[13]	15.97s[9]	27.82s[9]	107.06s[12]	48.64s[11]
0, 17, 1	ACT									2.81s[1]	5.86s[4]	**5.08s**	36.64s[9]
ORTHO-	IO	6.18s[2]	**2.72s**[1]	3.91s[2]	25.77s[1]	0.28s[1]	0.82s[1]	2.07s[1]	0.88s[1]	0.58s[1]	0.94s[1]	0.88s[1]	1.07s[1]
LATIN	DWD					**0.34s**	1.02s[2]	1.16s[2]	1.26s[2]	0.15s	0.24s	**0.06s**	0.54s[1]
1, 3, 5	ACT									0.22s	0.20s[1]	0.33s[1]	0.98s[1]
PATAT	IO	272.80s[2]	351.82s[15]	**59.18s**[1]	375.50s[18]	1800.00s[42]	1800.00s[42]	1800.00s[42]	1800.00s[42]	771.11s[35]	845.78s[35]	921.16s[35]	1228.48s[38]
	DWD					1272.96s[40]	1377.46s[41]	1406.26s[41]	**1216.18s**[39]	**170.75s**[12]	230.99s[13]	429.64s[20]	831.64s[33]
0, 42, 4	ACT									518.98s[31]	632.88s[29]	532.62s[32]	1348.48s[36]
QCP	IO	10.48s[5]	9.44s[18]	11.50s[21]	**6.08s**	0.05s[2]	0.06s[3]	0.05s	0.22s[5]	0.02s	0.03s	0.03s	0.14s
	DWD					**0.01s**	0.02s	0.02s	0.06s	0.01s	0.01s	0.01s	0.06s
20, 40, 0	ACT									**0.01s**	0.01s	0.01s	0.05s
QUASI-	IO	0.86s[2]	0.87s[3]	0.53s[2]	**2.91s**[1]	0.06s	0.06s	0.05s	0.07s	0.08s[1]	0.08s[1]	0.07s[1]	0.09s[1]
GROUP	DWD					0.03s	0.03s	**0.03s**	0.03s	0.03s[1]	**0.03s**[1]	0.03s[1]	0.05s[1]
18, 12, 5	ACT									0.06s[2]	0.06s[2]	0.05s[2]	0.08s[2]
QWH	IO	3.68s	2.10s[5]	2.68s[10]	**2.85s**	0.03s	0.04s	0.03s	0.14s	0.02s	0.02s	0.02s	0.10s
	DWD					**0.01s**	0.01s	0.01s	0.04s	0.01s	0.01s	0.01s	0.04s
0, 40, 0	ACT									**0.01s**	0.01s	0.01s	0.05s
OTHER	IO	2.28s	9.75s[1]	**0.64s**	17.28s	**6.34s**[1]	7.40s[1]	13.17s[1]	54.72s[2]	0.38s[1]	5.00s[1]	10.38s[1]	62.23s[2]
	DWD					12.22s[1]	15.18s[1]	21.63s[1]	70.45s[2]	**0.02s**	0.22s	0.18s	81.70s[2]
0, 3, 0	ACT									0.03s	0.42s	1.22s	92.14s[2]

Table 1: Models from the CSP2008 solver competition, against published results

`//www.cs.brown.edu/~sello/solutions.html` and `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`. The model has 2 *alldifferent* constraints, referred to as *alldiff0* between players in a group, and *alldiff1* between pairs of players overall. Symmetries are broken lexicographically. We use integer variables, whereas most CSP approaches use set variables, e.g. Gange et al. (2010), and we don't claim state-of-the-art results, only that the model exercises our propagators.

TALENT_SCHED (prob039 in CSPLib) schedules the scenes in a film to minimize the cost of the schedule in terms of actors' fees, where an actor is paid for the time from the first scene they are in until the last scene they are in. We use the three instances from CSPLib (a rehearsal problem plus *film1* and *film2* based on real data), plus the randomly generated *film1??* instances of Smith (2005). The input-order (static) search is based on Smith's but due to time constraints we haven't yet implemented the redundant constraints described by Smith.

Since *Gecode* supports the search strategies IO and DWD we also give the geometric means for conflict counts for each benchmark under the times in Tables 2 and 3 where we comparing against *Gecode*.

Since the higher-consistency propagators can be slow to execute, our default approach is to include a VALUE propagator at the same time, at a high priority (including, when there are no spare values, the clauses described in Section 4, noting that clauses have higher priority than propagators). So the strong propagator only executes after obvious propagation has been done, and is avoided entirely if failure is obvious.

To see whether these default redundant propagators were really an improvement, we took the best solver for each model and tried removing each type of redundant constraint (NOVALUE and NOCLAUSE), at least where it made sense to do so, which resulted in the matrix of Table 4. Note that the 'sat, unsat, ?' summary numbers do not exactly match Table 2 since the set of solveable instances is recomputed for each table based on the solvers attempted.

9 Discussion

In the first experiment *Chuffed* comprehensively beats the CSP2008 solver competition winners. The improvement is partly just from adding learning, but *Sugar* also has learning, and the improvement here is from lazy clause generation, because *Sugar* is based on SAT decompositions which are large on global constraints such as *element*, and also because *Sugar* has only bounds literals rather than the more advanced dual model discussed in Section 2. The new explained global propagators for *alldifferent* also played an important role in defeating the CSP2008 solvers.

In the second experiment *Chuffed* without learning is equivalent to, or slightly better than, the state-of-the-art publically available solver, *Gecode*. With learning it is comprehensively better. All problems except GOLOMB_RULER, LANGFORD and QG_EXISTENCE benefit from learning in our experiments. When learning, all problems except KAKURO and QG_COMPLETION benefit from higher consistency levels (BOUNDS, DOMAIN or FEYDY). Thus, on 4 of the 9 problems selected, we demonstrate the usefulness of explained higher-consistency propagators.

Learning changes the tradeoffs for propagation. DWD allows the best comparison. For the models QG_COMPLETION (Table 2), and SOCIAL_GOLFER (in particular the constraint *alldiff1*; Table 3), a strong propagator (DOMAIN) was best without learning but a simpler decomposition (VALUE) was best with learning, suggesting that learning can recover some of the global knowledge lost through decomposition.

Indeed for constraint *alldiff1* of SOCIAL_GOLFER the conflict count was *worse* with the stronger propagator, which we do not normally expect. The issue seems to be nogood reuse: The strong DOMAIN propagator produces a weaker nogood than VALUE, since the nogood involves many variables and describes a situation that might not recur often enough to pay

model unsat, sat, ?		*Gecode* *alldiff* =VALUE	BOUNDS	DOMAIN	FEYDY	NOLEARN *alldiff* =VALUE	BOUNDS	DOMAIN	FEYDY	LEARN *alldiff* =VALUE	BOUNDS	DOMAIN	FEYDY
GOLOMB_RULER 0, 3, 1	IO	30.90s 746078	**14.40s** **219092**	22.28s 219092	147.10s[1] 195952	66.73s 794358	**29.36s** **243769**	46.89s 243769	111.99s 243840	274.48s[1] 412471	**103.95s**[1] 170590	175.98s[1] 159359	146.08s[1] **134791**
	DWD	59.91s 1551794	22.58s 404733	42.88s 469285	303.10s[1] 402804	152.03s 1917948	49.66s 471899	98.85s 561992	206.39s[1] 527333	514.76s[2] 521536	183.76s[1] 282786	413.46s[2] 299639	245.19s[1] 208272
	ACT									1022.28s[2] 1299326	560.32s[2] 740216	976.18s[2] 712173	637.83s[2] 609045
INSN_SCHED 0, 39, 0	IO	839.72s[36] 23022833	**159.72s**[29] 1019960	279.15s[29] **951303**	678.11s[32] 140	841.90s[36] 25271476	**195.67s**[29] 492194	285.44s[29] 554869	284.57s[29] **22943**	20.05s[14] 6227	5.62s[7] 601	11.31s[9] 741	30.65s[12] 239
	DWD	1011.38s[37] 6176314	219.15s[30] 617280	368.15s[30] 138367	817.12s[33] 920	1027.00s[37] 14483805	287.06s[30] 706914	417.93s[30] 190637	490.81s[30] 44543	42.48s[15] 19867	**4.11s** **574**	30.67s[9] 2263	37.66s[8] 463
	ACT									18.65s[12] 20939	5.06s 1083	7.01s[1] 1673	12.55s[4] 304
KAKURO 0, 10, 0	IO	**1800.00s**[10] 27111480	1800.00s[10] 22150945	1800.00s[10] 21166803	1800.00s[10] **770872**	1800.00s[10] 82297506	1800.00s[10] 62950017	1800.00s[10] 48485283	1800.00s[10] 9276694	473.32s[7] 580547	508.85s[7] 584233	514.28s[7] 614998	745.40s[7] 366761
	DWD	1800.00s[10] 29817082	1800.00s[10] 23819512	1800.00s[10] 23343828	1800.00s[10] 1325438	1571.18s[9] 57417794	1800.00s[10] 52387830	1800.00s[10] 37955394	**1061.99s**[9] **4974804**	**1.90s** **21101**	3.64s 32441	3.34s[1] 28992	8.29s[1] 17943
	ACT									24.13s 200766	27.14s 178232	27.12s 153574	234.38s[2] 412243
KNIGHTS_TOUR 0, 5, 0	IO	**256.59s**[2] 2676127	271.01s[2] **2528878**	264.78s[2] 2530965	511.85s[3] 1268024	205.73s[2] 3182194	218.41s[2] 3093521	216.60s[2] 3077334	328.51s[2] 2525015	244.66s[3] 116045	263.62s[3] 113638	270.15s[3] 112789	316.48s[3] 122205
	DWD	554.63s[4] 9628143	561.59s[4] 7930382	655.84s[4] 8386774	1024.31s[4] 2603590	**102.23s**[2] 1616868	117.12s[2] 1517141	108.57s[2] 1560930	164.25s[2] **1274243**	55.74s[2] 90951	84.00s[2] 118841	10.57s[2] 26154	28.26s[1] 42700
	ACT									45.64s[1] 170283	20.56s[1] 60285	42.61s[1] 133384	**31.79s** **83068**
LANGFORD 5, 16, 4	IO	0.47s[3] 6057	0.31s[2] 2601	0.32s[2] 2596	0.70s[3] 2257	0.33s[2] 6752	0.25s[1] 2679	0.26s[1] 2678	0.41s[2] 2558	1.08s[4] 2433	0.62s[3] 1340	0.74s[4] 1341	0.88s[4] 1300
	DWD	**0.05s** 379	0.06s 359	0.06s **355**	0.05s[1] 72	**0.03s** 297	0.04s **288**	0.04s 288	0.07s 288	0.05s[1] 194	0.05s[1] **182**	0.06s[1] 189	0.09s[1] 193
	ACT									0.04s[1] 241	**0.04s**[1] 187	0.05s[1] 214	0.06s[1] 205
QG_COMPLETION 21, 64, 75	IO	388.57s[76] 3604285	385.38s[76] 1545976	309.90s[74] 996245	455.85s[75] 57285	364.26s[74] 2871125	366.36s[74] 1907175	390.85s[74] 1350303	645.99s[75] 167908	358.64s[73] 138630	357.83s[73] 134574	389.95s[73] 119224	659.25s[74] 56068
	DWD	345.13s[75] 2778555	342.16s[75] 1121675	**220.43s**[55] **608231**	451.20s[74] 51376	273.61s[54] 1576137	285.31s[57] 1003851	**270.96s**[46] **713505**	633.44s[72] 128872	225.25s[38] 85790	190.24s[43] 73486	188.42s[36] 59615	478.37s[61] 40346
	ACT									**89.01s**[11] **43014**	90.49s[15] 41402	84.06s[13] 30812	357.25s[38] 29410
QG_EXISTENCE 12, 23, 13	IO	127.01s[19] 370842	93.57s[17] 181646	81.87s[16] 133805	110.09s[17] 125406	73.49s[17] 228031	72.09s[17] 188657	68.04s[16] 165335	82.41s[17] 169940	28.08s[13] 16970	24.75s[13] 14204	24.00s[13] 12711	28.17s[13] 14281
	DWD	9.25s[10] 20285	10.27s[9] 18175	**9.08s**[8] **13802**	16.49s[10] 14731	3.21s[5] 7132	3.43s[5] 6916	3.43s[5] 6182	**4.04s**[4] **6166**	6.16s[7] 4621	5.08s[7] 3933	5.72s[7] 3975	**4.83s**[6] **3082**
	ACT									7.15s[8] 5086	10.05s[11] 6301	8.76s[10] 5046	13.19s[11] 7698
TALENT_SCHED 0, 9, 1	IO	709.06s[7] 13900566	708.35s[7] 11745423	**665.89s**[7] 11203913	715.40s[7] 8134702	**614.28s**[7] 21095605	631.69s[7] 19307512	625.50s[7] 20216040	649.70s[7] 17975676	521.24s[7] 823504	538.38s[7] 713159	549.09s[7] 714077	532.20s[7] 733577
	DWD	716.46s[7] 13479876	678.88s[7] 10230692	685.64s[7] 10779862	775.07s[7] **7344706**	623.25s[7] 15472903	631.93s[7] 15187825	623.32s[7] 14844274	656.48s[7] **14253332**	568.22s[7] 605667	582.31s[7] 660109	585.74s[7] 612372	546.25s[7] 695941
	ACT									341.45s[3] 576270	107.07s 185405	**52.51s** **104562**	114.05s 190925

Table 2: Models containing one *alldifferent* constraint, per propagator and search strategy

0, 39, 7 unsat, sat, ?		*Gecode*				NOLEARN				LEARN			
		alldiff0=VALUE	BOUNDS	DOMAIN	FEYDY	*alldiff0*=VALUE	BOUNDS	DOMAIN	FEYDY	*alldiff0*=VALUE	BOUNDS	DOMAIN	FEYDY
IO	*alldiff1*=VALUE	8.61s^{14}	1.26s^{9}	**0.97s**^{9}	1.62s^{10}	3.78s^{13}	0.93s^{9}	**0.88s**^{9}	1.09s^{9}	0.91s^{7}	0.60s^{8}	**0.49s**^{7}	0.70s^{7}
		13686	1066	660	722	10087	1010	848	978	725	233	193	231
	BOUNDS	9.20s^{14}	1.39s^{9}	1.04s^{9}	1.71s^{9}	4.29s^{13}	1.08s^{9}	0.93s^{9}	1.20s^{9}	1.05s^{7}	0.65s^{8}	0.55s^{7}	0.77s^{7}
		13060	1045	**639**	715	9202	983	793	952	738	235	193	239
	DOMAIN	10.57s^{15}	1.82s^{10}	1.40s^{10}	2.05s^{10}	5.58s^{13}	1.35s^{9}	1.21s^{9}	1.55s^{9}	1.37s^{8}	0.88s^{8}	0.75s^{7}	1.02s^{8}
		7512	918	572	652	7099	867	**726**	856	707	228	**186**	236
	FEYDY	39.95s^{19}	12.35s^{14}	9.46s^{13}	11.93s^{14}	10.18s^{16}	4.01s^{13}	3.51s^{13}	4.10s^{13}	4.67s^{11}	3.49s^{12}	2.83s^{11}	3.80s^{12}
		494	119	77	93	490	98	80	97	81	38	30	38
DWD	*alldiff1*=VALUE	93.47s^{26}	1.48s^{7}	1.18s^{7}	2.18s^{8}	10.91s^{14}	10.76s^{15}	9.35s^{10}	9.71s^{12}	0.38s^{3}	**0.33s**^{1}	0.30s^{2}	0.37s^{2}
		149011	1747	1327	1610	24281	22795	18686	16773	283	206	177	192
	BOUNDS	109.26s^{26}	1.58s^{7}	1.36s^{7}	2.13s^{7}	12.14s^{14}	11.41s^{15}	10.45s^{10}	10.12s^{12}	0.42s^{3}	0.34s^{1}	0.33s^{2}	0.40s^{1}
		154582	1622	1375	1486	23564	21319	17903	15573	282	191	171	185
	DOMAIN	100.61s^{26}	2.03s^{6}	**1.14s**^{5}	1.92s^{5}	4.85s^{8}	4.29s^{9}	**3.40s**^{6}	5.02s^{7}	0.71s^{3}	0.58s^{1}	0.57s^{1}	0.57s^{1}
		96630	1462	**819**	1026	7342	6590	**4916**	6895	296	200	**183**	183
	FEYDY	177.35s^{27}	20.88s^{13}	14.64s^{12}	22.90s^{13}	29.74s^{17}	47.34s^{20}	46.13s^{19}	47.11s^{20}	3.15s^{9}	2.76s^{9}	2.71s^{9}	2.55s^{8}
		2173	240	154	222	2489	2841	2824	2186	50	43	39	38
ACT	*alldiff1*=VALUE									0.38s^{2}	0.31s^{2}	**0.27s**^{1}	0.36s^{2}
										382	213	**182**	202
	BOUNDS									0.39s^{1}	0.36s^{2}	0.33s^{2}	0.37s^{1}
										350	202	202	201
	DOMAIN									0.68s^{2}	0.41s^{2}	0.54s^{2}	0.52s^{2}
										396	162	219	200
	FEYDY									4.24s^{9}	3.93s^{9}	3.72s^{9}	3.32s^{8}
										108	95	83	95

Table 3: Model SOCIAL_GOLFER, propagator matrix for the two *alldifferent* constraints, by search strategy

model solver unsat, sat, ?		NOVALUE		VALUE
GOLOMB_RULER	NOCLAUSES	32.94s	**243769**	**29.67s**	243769
NOLEARN, BOUNDS, IO 0, 3, 1	CLAUSES				
INSN_SCHED	NOCLAUSES	**3.46s**	**417**	4.41s	573
LEARN, BOUNDS, DWD 0, 39, 0	CLAUSES				
KAKURO	NOCLAUSES			1.19s	14787[1]
LEARN, VALUE, DWD 0, 10, 0	CLAUSES			**1.90s**	**21101**
KNIGHTS_TOUR	NOCLAUSES	34.46s	**77418**	32.87s	77418
LEARN, FEYDY, ACT 0, 5, 0	CLAUSES	34.00s	83068	**31.79s**	83068
LANGFORD	NOCLAUSES			0.06s	416
NOLEARN, VALUE, DWD 5, 16, 4	CLAUSES			**0.03s**	**297**
QG_COMPLETION	NOCLAUSES			341.63s	152155[62]
LEARN, VALUE, ACT 21, 53, 86	CLAUSES			**56.93s**	**28702**
QG_EXISTENCE	NOCLAUSES	2.13s	3318	2.20s	3378[1]
NOLEARN, FEYDY, DWD 12, 19, 17	CLAUSES	1.68s	2892[1]	**1.84s**	**2989**
TALENT_SCHED	NOCLAUSES	79.24s	175839	66.76s	164128[1]
LEARN, DOMAIN, ACT 0, 9, 1	CLAUSES	**38.73s**	105665	52.51s	**104562**
SOCIAL_GOLFER	NOCLAUSES	**0.18s**	170	0.23s	183[1]
LEARN, DOMAIN/VALUE, ACT 0, 38, 8	CLAUSES	0.15s	146[1]	0.21s	**151**

Table 4: Removing default redundant propagators

the nogood's propagation cost. The cases where DOMAIN is beneficial, such as the constraint *alldiff0* of SOCIAL_GOLFER, tend to be those where the domain size is very small, so the nogoods are always highly reuseable and also cheap to produce.

For INSN_SCHED where the bounds propagator is best, we see the opposite effect than for DOMAIN, the nogoods produced by BOUNDS are stronger than VALUE because the domains are large and sparse, so collisions between values (pruned by the VALUE propagator) are unlikely, whereas many different pruning opportunities are compactly described by a BOUNDS nogood. These nogoods are also symmetric between variables, since they describe a Hall interval rather than any specific pruning resulting from the existence of the Hall interval, which further promotes reuse.

For most models ACT is the best. In our experience the models for which ACT is the wrong approach, tend to be those with a good static search order (here INSN_SCHED), where gaps in the sequence of variable assignments are difficult to resolve later on. In such cases DWD is a useful compromise between activity-based search (*weighted degree* is similar to activity) and sequential search (*domain* size causes a domino effect which makes the search ripple outwards from previously fixed variables). But where ACT works it is almost always significantly better, and allowing the use of activity based search with strong propagators is an important contribution of our work.

Our KNIGHTS_TOUR model relies on *linear* constraints propagated to bounds consistency which creates a 5×5 bounding box for each knight move. This causes the most propagation at the edges of the board, so it is intuitive that FEYDY, which efficiently detects Hall intervals aligned to the edges of the board, should perform best on this model. We don't claim state-of-the-art results since specialized techniques based on lookahead can solve the problem greedily (von Warnsdorff 1823), but the model is still very useful in demonstrating that the FEYDY decomposition may be best despite its (relative) simplicity.

Referring to Table 4, for most models it is important, indeed essential, to add the redundant VALUE propagator and the extra clauses if possible.

The exceptions are: INSN_SCHED, where VALUE does not prune very well as discussed above; and SOCIAL_GOLFER (in particular the *alldiff0* constraint) and TALENT_SCHED, where domains are small and the domain propagator is cheap as discussed above.

10 Conclusions and further work

We have shown how to extend propagators for *alldifferent* to explain their propagation, in order to use them in a lazy clause generation solver. We see that for problems involving *alldifferent*, learning is usually hugely beneficial. Each of the different propagation methods is best for some problems, so having a range of different propagators (or decompositions) that can explain their propagation is valuable. Overall combining learning and *alldifferent* leads to a highly competitive approach to these problems. The combination of global *alldifferent* constraints with explanation leads to a state-of-the-art constraint programming solution to problems involving *alldifferent*.

In further work we would like to investigate why learning isn't effective on some of the models. We conjecture it may be because the instances become too hard too quickly, and that indeed learning may be better on harder instances but this is academic when the instances are out of reach for any solver. Further work could also examine hybrid methods to see if our work can be incorporated into the specialized solvers for TALENT_SCHED (Garcia de la Banda et al. 2010) or SOCIAL_GOLFER (Gange et al. 2010).

References

Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.-G. & Walsh, T. (2009*a*), Circuit Complexity and Decompositions of Global Constraints, *in* 'Procs. of IJCAI-2009', pp. 412–418.

Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.-G. & Walsh, T. (2009*b*), Decompositions of All Different, Global Cardinality and Related Constraints, *in* 'Procs. of IJCAI-2009', pp. 419–424.

Boussemart, F., Hemery, F., Lecoutre, C. & Sais, L. (2004), Boosting Systematic Search by Weighting Constraints, *in* 'Procs. of ECAI04', pp. 146–150.

Colton, S. & Miguel, I. (2001), Constraint Generation via Automated Theory Formation, *in* 'Procs. of CP01', pp. 575–579.

CSP (2008), 'Third international csp solver competition'. `http://www.cril.univ-atrois.fr/CPAI08`.

Feydy, T. & Stuckey, P. (2009), Lazy Clause Generation Reengineered, *in* 'Procs. of CP2009', pp. 352–366.

Ford, L. & Fulkerson, D. (1956), 'Maximal flow through a network', *Canad. J. Math.* **8**, 399–404.

G12 Project (2010), 'MiniZinc Challenge'. `http://www.g12.cs.mu.oz.au/minizinc/challenge2010/challenge.html`.

Gange, G., Stuckey, P. & Lagoon, V. (2010), 'Fast set bounds propagation using a BDD-SAT hybrid', *JAIR* **38**, 307–338.

Garcia de la Banda, M., Stuckey, P. J. & Chu, G. (2010), 'Solving Talent Scheduling with Dynamic Programming', *INFORMS J. on Computing (preprint)* .

Gecode Team (2006), 'Gecode: Generic constraint development environment'. `http://www.gecode.org`.

Gent, I., Miguel, I. & Nightingale, P. (2008), 'Generalised arc consistency for the AllDifferent constraint: An empirical survey', *AI* **172**(18), 1973 – 2000.

Gent, I. P. & Walsh, T. (1999), CSPLIB: A Benchmark Library for Constraints, *in* 'Princ. and Prac. of CP', pp. 480–481. `http://www.csplib.org`.

Hall, P. (1935), 'On Representatives of Subsets', *J. London Math. Soc.* **s1-10**(1), 26–30.

Katsirelos, G. (2008), Nogood processing in CSPs, PhD thesis, University of Toronto, Canada.

Lopez-Ortiz, A., Quimper, C.-G., Tromp, J. & Van Beek, P. (2003), A fast and simple algorithm for bounds consistency of the all different constraint, *in* 'Procs. of IJCAI-2003', pp. 245–250.

Luby, M., Sinclair, A. & Zuckerman, D. (1993), 'Optimal speedup of Las Vegas algorithms', *Inf. Proc. Let.* **47**(4), 173 – 180.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. & Malik, S. (2001), Chaff: engineering an efficient SAT solver, *in* 'Procs. of DAC01', pp. 530–535.

Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G. & Tack, G. (2007), MiniZinc: Towards a Standard CP Modelling Language, *in* 'Procs. of CP2007', Vol. 4741 of *LNCS*, Springer-Verlag, pp. 529–543.

Ohrimenko, O., Stuckey, P. & Codish, M. (2009), 'Propagation via lazy clause generation', *Constraints* **14**, 357–391.

Régin, J.-C. (1994), A filtering algorithm for constraints of difference in CSPs, *in* 'Procs. of AAAI-1994', pp. 362–367.

Simonis, H. (2008), Kakuro as a Constraint Problem, *in* P. Flener & H. Simonis, eds, 'Procs. of MODREF08', Uppsala University, Computing Science.

Smith, B. (2005), Caching Search States in Permutation Problems, *in* P. van Beek, ed., 'Procs. of CP2005', Vol. 3709 of *LNCS*, Springer Berlin / Heidelberg, pp. 637–651.

Tarjan, R. E. (1972), 'Depth-First Search and Linear Graph Algorithms', *SIAM J. Computing* **1**(2), 146–160.

Tarjan, R. E. (1975), 'Efficiency of a Good But Not Linear Set Union Algorithm', *J. ACM* **22**, 215–225.

van Hoeve, W. J. (2001), The alldifferent Constraint: A Survey, *in* 'Procs. of the 6th ERCIM Working Group on Constraints Workshop', Vol. cs.PL/0105015.

von Warnsdorff, H. C. (1823), *Des Rösselsprungs einfachste und allgemeinste Lösung*, Th. G. Fr. Varnhagenschen Buchhandlung, Schmalkalden.