Explaining flow-based propagation

Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey

National ICT Australia* and the University of Melbourne, Victoria, Australia
{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. In this paper we present two new generic flow-based propagators (for hard and soft flow-based constraints) with several novel features, and most importantly, the addition of explanation capability. We discuss how explanations change the tradeoffs for propagation compared with the previous generic flow-based propagator, and show that the generic propagators can efficiently replace specialized versions, in particular for *gcc* and *sequence* constraints. Using real-world scheduling and rostering problems as examples, we compare against a number of standard Constraint Programming implementations of these contraints (and in the case of soft constraints, Mixed-Integer Programming models) to show that the new global propagators are extremely beneficial on these benchmarks.

1 Introduction

Lazy clause generation [16] is a hybrid approach to constraint solving that uses a traditional DPLL or 'propagation and search' constraint solver as the outer layer which guides the solution process, plus an inner layer which lazily decomposes the Constraint Program (CP) to a Boolean satisfiability problem (SAT) and applies the latest SAT solver technology to prune the search [15].

gcc and *sequence* are two of the most important global constraints. They occur frequently in scheduling and rostering problems.

The *gcc* constraint takes the form $gcc([x_1, \ldots, x_n], [c_1, \ldots, c_m])$ and says that each value $v \in 1..m$ occurs c_v times in the list of x-values. If only the domains of the c are interesting, we write their intervals directly e.g. 1..2 instead of c_1.

The *sequence* constraint takes the form $sequence(l, u, w, [y_1, \ldots, y_n])$ and says that every consecutive w-window of y-variables sums to $l..u$.

Earlier work has shown flow-based propagation can be used to efficiently implement these constraints [4, 18]. The previous generic flow-based propagator by Steiger et al. [22] is promising but does not incorporate the work on *gcc* and nor does it produce explanations for use in a learning solver. Ideas on flow-based explanations have been proposed for generic flow networks [20] and for the special cases of *alldifferent* and *gcc* [12].

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

In this paper we present a new generic flow-based propagator which replaces all specialized flow-based propagators, supports soft constraints, and produces explanations for use in a lazy clause generation solver.

We take a fairly different approach to the previous work [22] because the previous propagator relied on a form of lookahead, which is not advantageous in a learning context, since simply searching on the lookahead value will add a *nogood* which will have the same effect for the remainder of search. This effect is well understood for SAT solvers [10] and confirmed by our early experiments.

The contributions of this paper are:

- We implement for the first time explanations for flow-based propagation.
- We give a systematic approach to pruning flow bounds, as opposed to the existing previous methods [22] which relied on explicit testing.
- We give a specialized method for deriving infeasibility from a spanning tree solution which is simpler and more efficient than the existing method for general linear programs [1] applied to network flow.
- We give new flow network encodings for *sequence* and *gsc* constraints.
- We define a new search strategy for CP optimization problems with flow networks, combining pseudocost [7] and reduced cost.
- We provide experiments showing that flow propagators with explanation can produce state-of-the-art results for problems encodable using flow networks.
- We show that learning is advantageous for flow propagation, even though explanations (particularly for soft constraints) can be large.

2 Lazy clause generation

We give a brief description of propagation-based solving and lazy clause generation, for more details see [16]. We consider constraint satisfaction problems, 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 considers 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 by introducing Boolean variables for each potential value of a CP variable, named $[x_i = j]$ and $[x_i \geq j]$. Negating them gives $[x_i \neq j]$ and $[x_i \leq j-1]$. Fixing such a *literal* modifies $D(x_i)$ to make the corresponding fact true, and vice versa. Hence the 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 conjunction of literals. When the propagator detects failure it

must explain the failure as a *nogood*, $L \rightarrow \mathit{false}$, with L a conjunction of literals which cannot hold simultaneously. Then L is used for conflict analysis [15].

3 Flow networks

A *flow network* is a graph (N, A) which models a system where flow is conserved, e.g. the pipes in a refinery, or the truck routes in a distribution network. It consists of nodes N and arcs $A = \{(u, v) : \text{there is a directed arc } u \rightarrow v\}$. Flow in the graph is represented by a vector $\mathbf{f}$ with bounds vectors $\mathbf{l}, \mathbf{u}$ such that $l_{uv} \leq f_{uv} \leq u_{uv}$ for all arcs (u, v). Flow conservation at each node requires that $\mathit{outflows} - \mathit{inflows} = \mathit{supply}$, or more technically

$$\forall n \in N, \sum_{v \in N:(n,v) \in A} f_{nv} - \sum_{u \in N:(u,n) \in A} f_{un} = s_n, \tag{1}$$

where the supply ($s_n > 0$) or demand ($s_n < 0$) is a constant taken from a vector $\mathbf{s}$ whose entries sum to 0. The network may also have a cost vector $\mathbf{c}$ which associates a cost per unit flow with each arc, such that $\mathbf{c}^T\mathbf{f}$ is the cost of solution $\mathbf{f}$. Further discussion of the cost vector is deferred to Section 5. Note that there may be parallel arcs of different cost (Section 7) but we only discuss the case without parallel arcs because the notation is much simpler.

Example 1. Figure 1 shows a simple flow network with nodes representing nurses (x = Xavier, y = Yasmin), shifts (d = day, n = night), and a sink t. A feasible (integer) assignment to $\mathbf{f}$ gives a solution to a nurse rostering problem:

- 1 or 2 nurses on day shift,
- 0 or 1 nurses on night shift,
- $f_{ij} = 1$ if nurse i works shift j, 0 otherwise.

Flow conservation ensures the validity of the solution, for

- nurse i works only one of the shifts, because $f_{i\mathsf{d}} + f_{i\mathsf{n}} = 1$ at node i,
- the number of nurses on shift j is $f_{j\mathsf{t}}$, because $f_{\mathsf{x}j} + f_{\mathsf{y}j} = f_{j\mathsf{t}}$ at node j, and
- the staffing requirement for shift j is expressed as the bounds on $f_{j\mathsf{t}}$.

This illustrates Régin's [18] encoding of the constraint $\mathit{gcc}([x, y], [1..2, 0..1])$, with $x, y = 1$ (day) or 2 (night) being the shift worked by Xavier (x) and Yasmin (y). Using the coercion function *bool2int*, the 'working arc' flows are expressed directly as domain literals which are intrinsic in a Lazy Clause Generation solver, e.g. $f_{\mathsf{xd}} = \mathit{bool2int}([x = 1])$, where $\mathit{bool2int}(\mathit{false}) = 0$ and $\mathit{bool2int}(\mathit{true}) = 1$.

3.1 Ford and Fulkerson's algorithm

We define the residual graph as summarizing, based on some current solution $\mathbf{f}$, the allowable neighbouring solutions. Where an arc $(u, v) \in A$ appears in the residual graph it means $f_{uv} < u_{uv}$ and can be increased. Where the reverse arc (v, u) appears in the residual graph it means $f_{uv} > l_{uv}$ and can be decreased. If neither arc appears, f_{uv} is fixed. If both arcs appear, f_{uv} is at neither bound.

From a solution $\mathbf{f}$ which respects the bounds but not the flow conservation constraints (hence certain nodes have an *excess* of flow and certain nodes a *deficit*), we can approach feasibility using Ford and Fulkerson's algorithm [6]. We

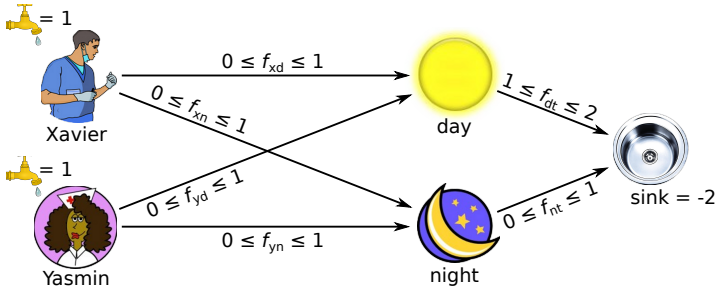


Fig. 1: Example flow network encoding a *gcc* constraint

augment, that is, increase the flow, along paths of the residual graph (each time updating the solution and corresponding residual graph). The *augmenting path* chosen is always from a node in excess to a node in deficit, which systematically reduces the infeasibility until feasibility is achieved. The only paths considered are those in the residual graph, ensuring that flows stay within their bounds.

Example 2. Continuing Example 1, Figure 2 shows the residual graph of the feasible solution which has Xavier on night shift and Yasmin on day shift, that is $f_{\mathsf{xn}} = 1$, $f_{\mathsf{yd}} = 1$, and so on. Since this is the graph of a *gcc* constraint, for simplicity we label certain arcs directly with their Boolean literals, understanding that *false* is a flow of 0 and *true* is a flow of 1. The bounds $\mathbf{l}, \mathbf{u}$ are as illustrated in the earlier Figure 1, so the *false* arcs are drawn in a forward orientation (can be increased to *true*) whereas the *true* arcs are drawn reverse (can be decreased to *false*). The staffing-level arcs f_{it} are also re-oriented as appropriate.

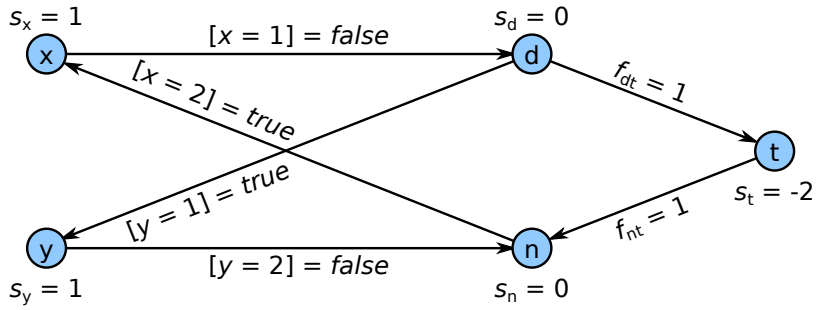
Suppose the flow bound u_{yd} is externally tightened to 0, that is Yasmin is no longer available for day shift (this could occur through search or as a result of side constraints). Before applying Ford and Fulkerson's algorithm we have to put flows into range, so f_{yd} is reduced to 0, equivalently $[y = 1]$ is set to *false*, creating an excess at node y and a deficit at node d, shown in Figure 3. f_{yd} is now fixed so removed from the residual graph, shown as the dotted line from node y to d. An appropriate augmenting path is identified in Figure 4. After augmenting along this path, feasibility is restored as shown in Figure 5.

4 Network flow propagator

We define the new constraint *network_flow*$(N, A, \mathbf{s}, \mathbf{f})$ which enforces the flow conservation constraints (1) on $\mathbf{f}$ according to the graph (N, A) and supplies $\mathbf{s}$, where $l_{uv}, u_{uv} = \min, \max D(f_{uv})$. The propagator maintains a (possible) solution to the flow graph at all times. It wakes up with low priority when any flow bound is tightened and attempts to repair its solution for the new bounds.

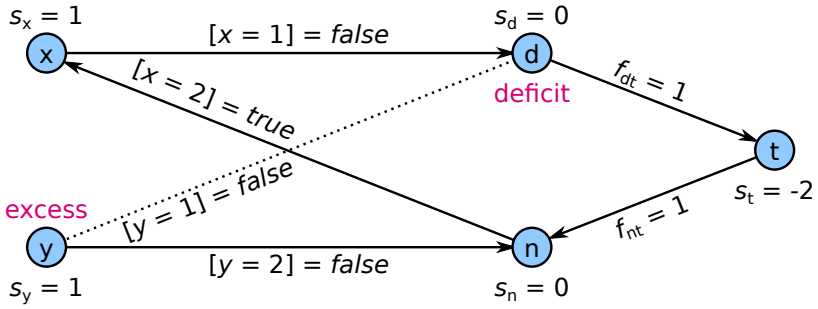
4.1 Explaining failure

Suppose there is no feasible solution. Let C, the 'cut', be the set of nodes searched for an augmenting path. It contains node(s) in excess but none in deficit. Then according to the current flow bounds, more flow enters C than can leave it, taking



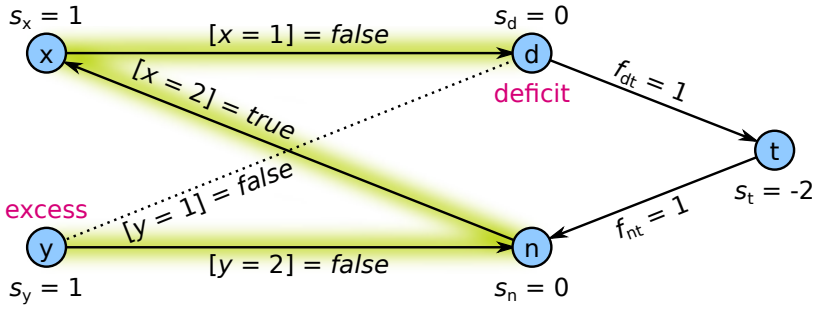
.5

Fig. 2: Residual graph ($x = 2$, $y = 1$)



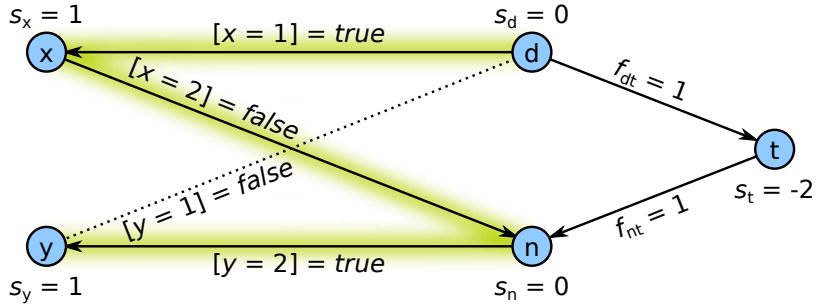
.5

Fig. 3: After an external pruning



.5

Fig. 4: Found an augmenting path



.5

Fig. 5: Feasibility restored

Fig. 6: Ford and Fulkerson's algorithm to find feasible flows

into account the arcs crossing C and the net supply/demand of C. Summing the equations (1) over $n \in C$ gives flow conservation for the cut,

$$\sum_{(u,v) \text{ leaves } C} f_{uv} - \sum_{(u,v) \text{ enters } C} f_{uv} = \sum_{n \in C} s_n. \tag{2}$$

Given C that proves infeasibility, we explain equation (2) as a *linear* constraint, using a standard linear explanation for LHS $\leq$ RHS [16]. Even if outflows are at minimum for outgoing arcs and inflows are at maximum for incoming arcs, minimizing the net flow leaving the cut, the net flow is still greater than the net supply/demand of the cut. The explanation of failure is the conjunction of literals $[f_{uv} \geq l_{uv}]$ for outflows and $[f_{uv} \leq u_{uv}]$ for inflows, using current $\mathbf{l}$, $\mathbf{u}$. Similar explanations were proposed by Rochart [20]. For the special case of *gcc* they reduce to those proposed by Katsirelos [12]. We can improve the base explanation by using lifting methods [1, 5, 16] to create a stronger explanation.

Example 3. Continuing Example 2, suppose search sets $f_{\mathsf{xd}} = f_{\mathsf{yd}} = 0$, equivalently $x, y \neq 1$, so that insufficient nurses are available for day shift. Figure 7 shows the residual graph of a partial solution with flows in range but not conserved. Attempting to resolve the excess, breadth-first search explores nodes $C = \{\mathsf{x}, \mathsf{n}, \mathsf{y}\}$. Cut-conservation (2) requires *bool2int*$([x = 1])$ + *bool2int*$([y = 1])$ + $f_{\mathsf{nt}} = 2$, unachievable since both literals are *false* and $f_{\mathsf{nt}} \leq 1$. Hence the *network_flow* propagator fails with nogood $[x \neq 1] \wedge [y \neq 1] \wedge [f_{\mathsf{nt}} \leq 1] \rightarrow$ *false*.

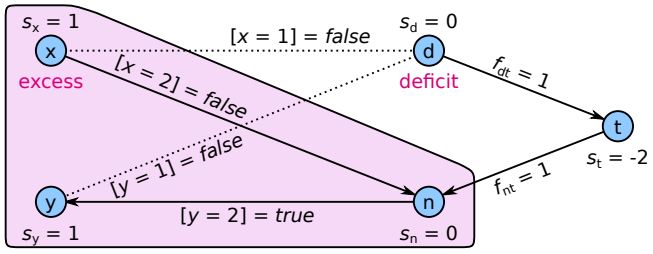


Fig. 7: Example residual graph showing infeasibility of the *gcc* constraint

4.2 Explaining pruning

Suppose that, on wakeup, there is a feasible solution to the network under the tightened bounds. Pruning is possible, if some f_{uv} can no longer reach its minimum or maximum due to the externally-tightened bounds that caused the wakeup. Régin describes a method based on Strongly Connected Components (SCCs) for *gcc* constraints [18], which we generalize to any flow network to find all arcs fixed at a bound, that is $f_{uv} = l_{uv}$ (resp. u_{uv}) which cannot increase (resp. decrease). For Boolean flow variables, bound-tightening implies fixing at a bound and vice versa, giving bounds-consistency on Boolean-valued arcs.

It is easy to see that the flow along an arc can only increase/decrease if an augmenting cycle can be found in the residual graph, that passes through the

arc in the appropriate direction (and does not pass back through the same arc). To check this we compute the SCCs of the residual graph, which can be done in linear time by Tarjan's algorithm [23]. An arc $u \to v$ with u, v in different SCCs can never be augmented since by definition u is not reachable again from v.

The explanation for pruning is the same as for failure, except that an SCC is used as the cut-set C instead of an infeasible set. Once again we treat equation (2) as a *linear* '$\leq$' constraint. This relies on the SCC acting as a 'trap' for incoming flow, to prune an incoming flow the bounds on outgoing flows must be tight.

Example 4. Consider *alldifferent*(x_1, x_2, x_3), expressed as the usual *gcc* network of *gcc*$([x_1, x_2, x_3], [c_1, c_2, c_3, c_4])$ where $c_i \in 0..1$. If $x_1 \in \{1, 2\}$, $x_2 \in \{2, 3\}$, $x_3 \in \{2, 3, 4\}$, then a solution is $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, as shown in Figure 8. The residual graph of this solution is shown in Figure 9. Due to the cycle $t \to 1 \to x_1 \to 2 \to x_2 \to 3 \to x_3 \to 4 \to t$ every node is reachable from each other, the entire graph is a single SCC, and no pruning is possible.

Now suppose $x_3 \neq 4$, that is, the arc $x_3 \to 4$ is pruned externally, as shown in Figure 10. Tarjan's algorithm executes starting from node t and proceeds through nodes 1 and x_1 before discovering SCC #1. Then the arc $x_1 \to 2$ may be pruned due to cut-conservation (2) for SCC #1: *bool2int*$([x_3 = 4]) + c_2 + c_3 -$ *bool2int*$([x_1 = 2]) = 2$ and hence *bool2int*$([x_1 = 2]) = 0$ since $[x_3 = 4] =$ *false*, $c_2 \leq 1$, and $c_3 \leq 1$. The explanation is $[x_3 \neq 4] \wedge [c_2 \leq 1] \wedge [c_3 \leq 1] \to [x_1 \neq 2]$ or after removing redundant bounds $[x_3 \neq 4] \to [x_1 \neq 2]$. Having pruned all arcs leaving SCC #2, that SCC is closed, allowing the arc $x_1 \to 1$ to be fixed to *true* using $[x_1 \neq 2]$ as justification and so on.

5 Minimum cost flow networks

When there is a cost vector $\mathbf{c}$ for the network, instead of just solving for any feasible flow we have to solve the following optimization problem,

$$\min \mathbf{c}^T\mathbf{f} \text{ s.t. } \mathbf{Af} = \mathbf{s},\ \mathbf{f} \geq \mathbf{l},\ \mathbf{f} \leq \mathbf{u}, \tag{3}$$

where each row of $\mathbf{A}$ corresponds to a flow conservation equation (1). This is a Linear Program (LP) and may be solved by the well-known Simplex method. Since the column of A corresponding to a flow f_{uv} consists of a difference of unit vectors $e_u - e_v$, this LP is a network LP and may equivalently be solved by Network Simplex, which is usually faster, because operations on general matrices reduce to a series of operations on spanning trees and augmenting paths.

In a network flow problem a basic solution is a spanning tree of the graph (N, A), directed in the sense that the root is distinguished and all tree-arcs point upwards to the root (requiring us to correct for the current tree-direction of an arc when referring to its flow variable). Non-tree arcs are set to a fixed flow value, which may be either the lower or upper bound of the associated flow variable. This gives the tree-arc flows, as the outgoing (i.e. upwards) flow of a node is its supply plus incoming flows (i.e. $\mathbf{Af} = \mathbf{s}$ has $|A| - |N|$ degrees of freedom).

Each node n is assigned a potential g_n which is the cost of taking a unit of flow from that node to the root (via the tree). Then the reduced cost h_{uv} for

each arc says how much the overall cost would change if a unit of flow from u to the root at cost g_u, were re-routed via the arc (u, v), i.e. from u to v and then to the root at cost $c_{uv} + g_v$. Taking the difference in cost, $h_{uv} = c_{uv} + g_v - g_u$.

5.1 Dual Network Simplex

Since we use the (lesser known) Dual Network Simplex method we cannot avoid a brief discussion of duality. Let $\mathbf{y}$ be a vector with one entry per constraint called the *row costs*, indicating the local change in the objective per unit change in the right-hand side of the constraint. For problem (3) this is simply the node potentials (row costs generalize this concept). Now rewrite the primal (3) as

$$\min \mathbf{c}^T\mathbf{f} \text{ s.t. } \begin{bmatrix} \mathbf{A} \\ -\mathbf{A} \\ I \\ -I \end{bmatrix} \mathbf{f} \geq \begin{bmatrix} \mathbf{s} \\ -\mathbf{s} \\ \mathbf{l} \\ -\mathbf{u} \end{bmatrix}, \text{ row costs } \mathbf{y} = \begin{bmatrix} \mathbf{g}^+ \\ \mathbf{g}^- \\ \mathbf{h}^+ \\ \mathbf{h}^- \end{bmatrix}. \tag{4}$$

Then the node potentials and reduced costs discussed earlier become $\mathbf{g} = \mathbf{g}^+ - \mathbf{g}^-$ and $\mathbf{h} = \mathbf{h}^+ - \mathbf{h}^-$. The standard dual is an LP over the row costs vector $\mathbf{y}$, obtained by transposing the constraint matrix, costs, and right-hand sides,

$$\max \begin{bmatrix} \mathbf{s}^T & -\mathbf{s}^T & \mathbf{l}^T & -\mathbf{u}^T \end{bmatrix} \mathbf{y} \text{ s.t. } \begin{bmatrix} \mathbf{A}^T & -\mathbf{A}^T & I & -I \end{bmatrix} \mathbf{y} = \mathbf{c},\ \mathbf{y} \geq \mathbf{0},\ \text{row costs } \mathbf{f}. \tag{5}$$

Solving the dual problem to optimality yields variables $\mathbf{y}$ and row costs $\mathbf{f}$ which also solve the primal and vice versa. After bound tightenings as in the earlier Examples 2 to 4, the previous solution resident in the dual solver remains feasible (since modifying $\mathbf{l}, \mathbf{u}$ only changes the objective) so allows a warm start.

Dual Network Simplex, as opposed to the Ford and Fulkerson method, takes a solution where flows are conserved but may violate flow bounds, and 'pivots' to reduce the bounds violation while maintaining dual feasibility, that is, arcs at their lower (resp. upper) bounds have positive (resp. negative) reduced costs.

The dual pivot consists of choosing an arc to leave the spanning tree whose flow violates its bounds, then choosing the appropriate entering arc that maintains dual feasibility. The subtree or 'cut' under the leaving arc has its potentials updated and all arcs crossing the cut have their reduced costs updated accordingly. The entering arc must cross the cut, its reduced cost must have the correct sign, and when added to the other reduced costs it must not cause them to cross 0, hence its absolute value must be minimal among the possible arcs.

Example 5. Figure 12 shows a simplified underground mining network, which is convenient since all flows are naturally upwards, otherwise the example is more complicated. Supplies/demands are shown in bold beside the nodes, the mining areas at the leaf nodes supply one tonne of ore each ($s_{\mathsf{e}} = s_{\mathsf{f}} = s_{\mathsf{g}} = s_{\mathsf{i}} = s_{\mathsf{j}} = s_{\mathsf{k}} = 1$) which has to be moved to the mine portal ($s_{\mathsf{a}} = -6$). Beside each arc is shown in lightweight italic the cost c_{uv} per tonne moved through the arc.

Figure 13 shows a dual feasible tree for the network, with potentials in bold, flows and reduced costs in italics, and non-tree arcs dotted, of which f_{hg} is at its upper bound $u_{\mathsf{hg}} = 3$, others are at their lower bounds. f_{hd} violates its lower bound and will leave the tree. The cut shows nodes under the leaving arc.

The leaving arc must be augmented by 1 tonne to leave the tree at its lower bound, so the entering arc must provide 1 extra tonne into the cut, while the objective either increases or stays the same (the dual is a maximization problem). So we can either increase an inflow with reduced cost ≥ 0, or decrease an outflow with reduced cost ≤ 0. Then the possibilities are f_{hg} or f_{ej}, we have to choose the former because $|h_{\mathsf{hg}}| < |h_{\mathsf{ej}}|$. Figure 14 shows the result of the pivot.

6 Minimum cost network flow propagator

We define the new constraint *min_cost_network_flow*$(N, A, \mathbf{s}, \mathbf{f}, \mathbf{c}, z)$ which is the same as *network_flow* except that $\min D(z)$ increases to track the objective, hence fathoming occurs when $\mathbf{c}^T\mathbf{f} > \max D(z)$. The propagator wakes up with low priority upon bound tightening, re-optimizes from warm-start, and may fail/fathom, or perform its normal pruning plus additional pruning based on the objective. Explaining failure/fathoming depends on the fact that any solution to the dual gives an upper bound on the primal objective (weak duality).

6.1 Explaining failure

If the dual is unbounded then eventually after choosing a leaving arc no entering arc will have the correctly signed or zero reduced cost. Because $h_{uv} > 0$ implies $f_{uv} = l_{uv}$ and $h_{uv} < 0$ implies $f_{uv} = u_{uv}$, the leaving arc cannot be augmented because all other arcs crossing the cut (i.e. the subtree under the leaving arc) are tight at the appropriate bound, so there is too much flow attempting to cross the cut, and we can simply explain failure as in Section 4.1.

Example 6. Continuing Example 5, suppose $u_{ba} = 3$. Then f_{ba} violates its upper bound and is selected as the leaving arc. Figure 15 shows the resulting cut. To reduce the outflow on f_{ba} in a favourable way we look for inflows with $h_{uv} \leq 0$ or outflows with $h_{uv} \geq 0$ but find none. Increasing the potentials inside the cut by $\alpha > 0$ gives $h_{\mathsf{ba}} = \alpha$ and h_{eb}, h_{ej}, $h_{\mathsf{fc}} = 1 + \alpha$ which is dual feasible. The objective would increase by 4α. So the cut encodes an unbounded dual ray.

6.2 Explaining fathoming

Given flow bounds $\mathbf{l}$, $\mathbf{u}$, and an optimal flow $\mathbf{f}$ with reduced costs $\mathbf{h}$ and objective value m, the explanation for fathoming is

$$\bigwedge_{h_{uv}>0} [f_{uv} \geq l_{uv}] \wedge \bigwedge_{h_{uv}<0} [f_{uv} \leq u_{uv}] \rightarrow [z \geq m], \tag{6}$$

which causes a conflict when $\max D(z) < m$. This is after all intuitive because the flows being tight against their lower (resp. upper) bounds when reduced costs are positive (resp. negative) is what prevents us improving the solution.

To see this algebraically take also the potential vector $\mathbf{g}$ from the optimal solution considered above, and substitute $\mathbf{g}$, $\mathbf{h}$ into the *linear* constraint

$$\mathbf{g}^T\mathbf{s} + \mathbf{h}^T\mathbf{f} \leq z, \tag{7}$$

which is best considered as constraining the bounds $\mathbf{l}$, $\mathbf{u}$ on $\mathbf{f}$ rather than $\mathbf{f}$ itself,

$$\mathbf{g}^T\mathbf{s} + \sum_{h_{uv}>0} h_{uv}l_{uv} + \sum_{h_{uv}<0} h_{uv}u_{uv} \leq \max D(z).$$

when this is violated we know that

$$\text{primal objective} \geq \text{dual objective} = \left[\, -\mathbf{s}^T \ \mathbf{s}^T \ -\mathbf{l}^T \mathbf{u}^T \,\right] \mathbf{y} > \max D(z),$$

where $\mathbf{y}$ is any feasible solution of (5). Upon backtracking and trying a new subproblem, the feasible region of problem (5) is unaffected by any changes to $\mathbf{l}$, $\mathbf{u}$, hence $\mathbf{y}$ remains dual feasible even though its dual objective may change. By weak duality the new dual objective still provides a lower bound on $\mathbf{c}^T\mathbf{f}$.

So when (7) is violated we can fathom with the usual explanation of the *linear* constraint, essentially the clause (6), but treating as a *linear* constraint confers some advantages, (i) we can use a lifting algorithm [1, 5, 16], which in our implementation is naive but nevertheless effective, and (ii) we can propagate (7) to bounds consistency in the usual way each time the propagator executes, an idea known to the MIP community as reduced-cost variable fixing.

Whilst the dual optimal $\mathbf{y}$ provides the tightest bound and the most likelihood of detecting failure or pruning, we do not necessarily need the tightest bound. Explanations of fathoming from optimizing earlier subproblems have a good chance of being applicable on a new subproblem if it is similar enough.

Similar schemes for explaining failure and fathoming in general linear programs were given by Davey et al. [5] for problems involving 0-1 variables, and later by Achterberg [1] for general integer variables.

7 New *sequence* and *gsc* encodings

We give a new flow-based encoding for *sequence*, as a flow network, similar to [14] but simpler and using fewer arcs. Referring to Figure 17, a flow f_i along the spine corresponds to a sum of y_j over the w-window $i \leq j < i + w$, which we may show by a series of cuts, e.g. the cut illustrated shows by cut-conservation (2) that $f_3 = y_3 + y_4 + y_5$. Constraining the f_i-flows to $l \leq f_i \leq u$ enforces *sequence*. Tarjan's algorithm propagates the y to domain consistency if they are 0..1 valued (the common case). The f are only opportunistically pruned, but this does not matter as they are only introduced for the sake of the decomposition.

Régin and Puget's $gsc(l, u, w, [x_1, \ldots, x_n], [(v_1, c_1), \ldots, (v_m, c_m)])$ says that $x_i \in \{v_1, \ldots, v_m\}$ occurs $l..u$ times per w-window and that $x_i = v_j$ occurs c_j times overall [19]. In their experiments they reduced *gsc* to *gcc* for which a flow based propagator was available, at the expense of adding side constraints. Our network is equivalent but modifies the *gcc* instead of needing side constraints.

Referring to Figure 18, nodes x_i, v_j represent variables and values as in a standard *gcc* network. Nodes w_k ensure that $x_k, \ldots, x_{k+w-1}$ meet the $l..u$ constraint by setting the flow from the overall source s to those variable nodes. As windows w_k do not overlap, there are w different window alignments, hence w *network_flow* propagator instances.

We encode a soft version of each constraint as *min_cost_network_flow* by adding in parallel to each arc of capacity $l..u$, two additional 'violation arcs', one with capacity $-\infty..0$ and cost -1, the other with capacity $0..\infty$ and cost 1.

8 Experiments

We implemented the *network_flow* and *min_cost_network_flow* propagators in *Chuffed*, a state-of-the-art lazy clause generation solver. We used the MCF 1.3 Dual Network Simplex code [13]. We evaluated the new propagators on car sequencing and nurse rostering problems. Hardware was a Dell PowerEdge R415 cluster with dual-processor 2.8GHz AMD 6-Core Opteron 4184 nodes. Timeouts were 3600s and memory limit was 1.5Gb per core. *Minizinc* models and instances are available from `http://www.csse.unimelb.edu.au/~pjs/flow`.

Car sequencing. Car sequencing (prob001 in CSPLib [9]) is a problem of scheduling a day's production in an assembly plant. We consider instance set 1 consisting of 9 'classic' instances which are extremely difficult, some feasible and some infeasible, based on real data from Renault, and set 2 consisting of 70 randomly generated instances of increasing difficulty, all feasible. The first set is somewhat of a stress test, at least one instance has never been solved by CP or MIP methods to our knowledge [8]. The second set, although random, may be more realistic, as the usefulness of our technology in practice is defined by its ability to produce solutions to feasible problems in a reasonable time.

Nurse rostering. Nurse rostering is a problem of assigning shifts to nurses on consecutive days such that each shift has at least the required number of nurses (a *gcc* constraint per day) and that each individual nurse has an acceptable work pattern (*sequence* and clausal constraints). Symmetries are broken by lexicographic ordering (using a clausal decomposition). We use a version of models 1 and 2 described by Brand et al. [4], which are simple but plausible. Unlike those authors we keep the (clausal) 'no isolated shifts' constraints as well as adding *sequence*; their model was less realistic, and also less interesting for a propagation solver since *sequence* was essentially the only constraint.

The first 50 instances from NSPLib [24] (disregarding nurse preferences) were solved with each model. Model 1 is over-constrained, because we kept the 'no isolated shifts' rule, and all instances are infeasible, but since infeasibility forces a complete search, model 1 is the most useful for measuring the pruning power of each propagation method. Model 1 is also useful for testing soft-*sequence* and hence *min_cost_network_flow*, since any solution will be a compromise. Model 2 is more realistic and checks that we can find a useful roster in practice.

In the first experiment we compare the *network_flow* encoding of *gcc* and *sequence* with traditional approaches. We ran each combination of *gcc* implementation, *sequence* implementation, and search strategy chosen from STATIC: an appropriate fixed variable order for each problem [21], DOM/WDEG: domain size / weighted degree [3], IMPACT: impact-based search similar that of Refalo [17] but using log-impacts, and VSIDS: activity based search from SAT [15]. We try with and without learning, except for VSIDS which requires learning. We use geometric restarts, except with STATIC where restarting is not sensible.

		set 1: 2 sat, 0 unsat, 7 ?				set 2: 70 sat, 0 unsat, 0 ?			
		not learning		learning		not learning		learning	
		gcc=LIN	FLOW	*gcc*=LIN	FLOW	*gcc*=LIN	FLOW	*gcc*=LIN	FLOW
STATIC	*seq*=DPS	—	—	3456.2s^{1}	—	1758.5s^{38}	1782.4s^{37}	1466.5s^{43}	1646.4s^{40}
	REG	3292.0s^{1}	3294.4s^{1}	3263.7s^{1}	3321.4s^{1}	1362.9s^{46}	1357.4s^{47}	1252.0s^{47}	1442.8s^{44}
	FLOW	3257.1s^{1}	3244.7s^{1}	3250.1s^{1}	3273.6s^{1}	1284.9s^{49}	1217.8s^{50}	1173.0s^{49}	1449.8s^{44}
	gsc=FLOW	2800.1s^{2}		**2800.0s**^{2}		244.3s^{66}		206.3s^{66}	
DOM/	*seq*=DPS	—	—	—	—	3139.9s^{9}	3139.8s^{9}	181.5s^{67}	122.2s^{68}
WDEG	REG	—	—	—	—	622.5s^{58}	623.9s^{58}	87.4s^{69}	2.6s^{70}
	FLOW	—	—	—	—	892.3s^{53}	888.3s^{53}	114.0s^{68}	143.5s^{68}
	gsc=FLOW	3200.0s^{1}		2800.1s^{2}		215.7s^{66}		**0.8s**^{70}	
IMPACT	*seq*=DPS	—	—	—	—	—	—	3533.6s^{2}	3550.1s^{2}
	REG	—	—	—	—	1685.8s^{39}	1109.1s^{52}	1496.9s^{45}	1563.3s^{44}
	FLOW	—	—	—	—	1581.2s^{42}	1645.4s^{41}	1302.7s^{48}	1610.0s^{42}
	gsc=FLOW	3324.1s^{1}		3515.2s^{1}		1150.0s^{53}		951.3s^{55}	
VSIDS	*seq*=DPS			—	—			3504.1s^{2}	—
	REG			—	—			1097.3s^{51}	2191.0s^{29}
	FLOW			—	—			2493.2s^{24}	2908.0s^{14}
	gsc=FLOW			2885.3s^{2}				1522.2s^{47}	

Table 1: Car sequencing results

		model 1: 0 sat, 50 unsat, 0 ?				model 2: 37 sat, 10 unsat, 3 ?			
		not learning		learning		not learning		learning	
		gcc=LIN	FLOW	*gcc*=LIN	FLOW	*gcc*=LIN	FLOW	*gcc*=LIN	FLOW
STATIC	*seq*=DPS	3364.0s^{4}	3351.1s^{4}	9.7s^{50}	103.0s^{50}	3458.4s^{3}	3464.3s^{2}	1435.3s^{32}	1489.4s^{32}
	REG	3350.7s^{4}	3346.0s^{4}	6.5s^{50}	23.3s^{50}	3427.3s^{3}	3438.5s^{3}	1389.5s^{32}	1403.3s^{33}
	FLOW	3216.2s^{6}	3237.2s^{6}	0.6s^{50}	11.0s^{50}	3326.9s^{5}	3314.5s^{5}	1269.9s^{34}	1211.5s^{36}
DOM/WDEG	*seq*=DPS	799.9s^{39}	79.3s^{49}	0.9s^{50}	0.8s^{50}	3528.8s^{1}	3050.0s^{8}	1656.3s^{30}	1543.7s^{30}
	REG	2125.3s^{22}	2132.7s^{22}	8.5s^{50}	91.8s^{49}	—	—	1774.2s^{28}	1842.3s^{26}
	FLOW	72.1s^{49}	0.1s^{50}	0.2s^{50}	**0.0s**^{50}	—	—	1441.8s^{34}	1532.1s^{30}
IMPACT	*seq*=DPS	2498.4s^{17}	2504.7s^{19}	8.3s^{50}	10.3s^{50}	2176.6s^{22}	1942.8s^{25}	1045.7s^{37}	1106.6s^{36}
	REG	1693.5s^{31}	1932.0s^{30}	7.9s^{50}	8.1s^{50}	2079.5s^{24}	1555.8s^{31}	1095.5s^{36}	1034.5s^{36}
	FLOW	2084.5s^{25}	724.7s^{47}	3.1s^{50}	1.1s^{50}	1535.9s^{32}	1596.9s^{29}	834.7s^{39}	748.9s^{41}
VSIDS	*seq*=DPS			1.0s^{50}	0.7s^{50}			620.7s^{43}	753.6s^{41}
	REG			0.8s^{50}	0.4s^{50}			575.1s^{44}	512.4s^{45}
	FLOW			0.0s^{50}	0.1s^{50}			468.5s^{44}	**432.7s**^{47}

Table 2: Nurse rostering results

gcc$([x_1, \ldots, x_n], [c_1, \ldots, c_m])$ is implemented as LIN: decomposition into *linear* constraints $\sum_{i=1}^{n}$ *bool2int*$([x_i = j]) = c_j \ \forall j \in 1..m$, or FLOW: Régin's domain-consistent flow-based encoding using our new *network_flow* propagator.

sequence$(l, u, w, [y_1, \ldots, y_n])$ is implemented as DPS: difference of partial sums, where $s_i = \sum_{j=1}^{i} y_j \ \forall i \in 0..n$ and $l \leq s_{i+w} - s_i \leq u \ \forall i \in 0..n-w$ (both implemented as *linear* constraints), REG: *regular* decomposition into *table* constraints over allowable state change tuples (q_{i-1}, y_i, q_i) and thence to SAT, or FLOW: the new domain-consistent flow-based encoding described in Section 7.

On car sequencing, as well as the standard *gcc*+*sequence*+*table* model, we evaluate Régin and Puget's specialized *gsc* constraint [19], which is applied once per option instead of *sequence* and subsumes all other constraints.

Tables 1 and 2 report the number of instances solved and the mean of the runtime (or timeout). '—' indicates all instances timed out. The heading shows how many solved instances were unsatisfiable or satisfiable and how many were indeterminate as not solved by any solver. The solver which solves the most instances is highlighted, falling back to comparing runtimes.

		set 1: 9 sat, 0 unsat, 0 ?		set 2: 70 sat, 0 unsat, 0 ?	
		not learning	learning	not learning	learning
		opt,sol,s,obj,inf	opt,sol,s,obj,inf	opt,sol,s,obj,inf	opt,sol,s,obj,inf
seq	*SCIP*=MIP	4,9,2441s, 5.2,0	4,9, 2245s, 3.7,0	69,70, 350s, 0.0,0	70,70, 301s, 0.0,0
	CPLEX=MIP	5,9,1995s, 1.7,0		70,70, 8s, 0.0,0	
	Chuffed=LIN	—	0,9, 3600s,87.6,0	—	0,70,3600s,118.5,0
	FLOW	0,9,3600s,63.2,0	0,9, 3600s,66.4,0	10,70,3208s,50.9,0	21,70,2597s, 44.0,0
gsc	*SCIP*=MIP	4,9,2194s, 1.7,0	**5**,**9**,**1763s**, **1.7**,**0**	70,70, 33s, 0.0,0	70,70, 35s, 0.0,0
	CPLEX=MIP	4,9,2038s, 1.7,0		**70**,**70**, **3s**, **0.0**,**0**	
	Chuffed=LIN	0,9,3600s,73.1,0	0,9, 3600s,68.3,0	5,70,3394s,69.4,0	27,70,2335s, 48.2,0
	FLOW	0,9,3600s,57.0,0	0,9, 3600s,58.0,0	9,70,3412s,39.9,0	13,70,3285s, 38.1,0

Table 3: Car sequencing with soft-*sequence* and soft-*gsc*

	model 1: 45 sat, 5 unsat, 0 ?		model 2: 45 sat, 5 unsat, 0 ?	
	not learning	learning	not learning	learning
	opt,sol,s,obj,inf	opt,sol,s,obj,inf	opt,sol,s,obj,inf	opt,sol,s,obj,inf
SCIP=MIP	0,12,3242s,516.0,5	0,18, 3242s, 475.0,5	1,33,3240s,178.2,5	1,39, 3237s,115.9,5
CPLEX=MIP	0, 6,3241s,456.0,5		14,37,2485s, 10.6,5	
Chuffed=LIN	—	0, 0, 3407s, —,3	—	0,43, 3460s,250.5,2
FLOW	0,45,3528s,383.0,1	**0**,**45**,**3348s**,**327.0**,**4**	9,45,3102s, 56.1,3	**35**,**45**,**1135s**, **4.8**,**3**

Table 4: Nurse rostering with soft-*sequence*

The results for car sequencing show that flow based propagators are almost always preferable to other approaches for propagating *sequence* and *gcc*. While REG is better than FLOW for propagating *sequence* overall, the *gsc* approach is clearly the best on this problem, showing the benefit of a generic propagator.

These results clearly show that learning is strongly beneficial, even though explanations from flow networks can be large (usually hundreds of literals for nurse rostering, more for car sequencing). The only counter example is IMPACT which does not create reuseable nogoods, since the search is driven by domain reductions instead of failure. For the difficult problems the programmed search is preferable, while for the easier problems DOM/WDEG is clearly the best.

VSIDS was not the best strategy for this problem which we think is because it pays no attention to locality in the schedule so has trouble when partially filled areas meet, whereas DOM/WDEG tends to propagate outwards from partially filled areas since these are where the domain sizes are smallest.

The results for nurse rostering again reinforce that flow based approaches are preferable to other methods of propagating *sequence* and *gcc*. Learning is even more important on these examples regardless of the search strategy. The best approach overall is VSIDS with *sequence* and *gcc* encoded using flow, though DOM/WDEG was also competitive, at least on model 1.

In the second experiment we consider the same problems with the *sequence* constraints relaxed to soft-*sequence*, optimizing over the sum of violations. Unfortunately we could not obtain the instances used by Steiger et al. in their previous work on soft constaints, precluding a proper comparison with their explicit arc-testing algorithm. Since these are optimization problems we compare against MIP solvers: CPLEX 12.2 which does not use learning (although it does use other cutting plane methods) and SCIP 2.1.1 with or without learning.

Starting with the flow-based encodings of the hard-*gcc* and soft-*sequence* constraints, we prepared three different models, MIP: an Integer Program with all

constraints decomposed to *linear*, LIN: a Constraint Program with *(min_cost_) network_flow* constraints decomposed to *linear*, and FLOW: a Constraint Program utilizing the new *(min_cost_)network_flow* global propagators.

The MIP solvers use their default search strategy. For the CP-optimization problems we use a novel search strategy PSEUDOREDCOST inspired by Gauthier and Ribière [7]. None of the strategies used on the satisfaction problems compete with PSEUDOREDCOST on these problems. Pseudocosts are computed by sampling $\min D(z)$ before and after each decision (the latter sample takes into account the resulting propagation but is not re-sampled after backtracking to the same decision level later on), and averaging the differences with period 25. Failure counts as 25 objective units (a manual setting for now). Here z is the model objective and need not be associated with any *min_cost_network_flow*.

We compute the variable ordering based on pseudocosts plus reduced costs rather than pseudocosts alone. Higher (absolute) reduced cost indicates a more important variable. If multiple *min_cost_network_flow* propagators can provide a reduced cost (e.g. on soft-*gsc*) then their absolute values are summed.

Tables 3 and 4 give the results showing number of instances where an optimal solution was proved, number for which at least one solution was found, mean of elapsed time or timeout, mean objective of the best solution found (using only those instances for which data was available from all solvers that solved > 0 instances), and finally the number of instances proved unsatisfiable.

For *Chuffed*, the flow-based propagator was typically better, or much better, than *linear* constraints (*gsc* is an exception), and learning was clearly beneficial, in some cases highly beneficial. For *SCIP*, learning gave only a modest improvement. Our understanding is that *SCIP* is not optimized to propagate nogoods quickly. Also, their conflicts involve the entire LP rather than a network subproblem, probably resulting in longer and less reuseable explanations.

For car sequencing the results show that the MIP model, particularly with *CPLEX*'s excellent cutting plane methods and heuristics, is unbeatable, perhaps unsurprising since excellent results were reported earlier with MIP [11]. The results for nurse rostering are quite different. On model 1 the MIP solvers can only prove unsatisfiability and they (particularly *SCIP*) also have difficulty with model 2. The CP approach is far superior in finding good solutions quickly, and with both learning and the new propagators enabled, it clearly improves on *CPLEX* in the number of solutions proved optimal (35 vs 14).

We can explain the difference between car sequencing and nurse rostering by considering the clausal side constraints that accompany the flow networks. For car sequencing there are typically 10000 binary clauses and 1000 longer clauses. For nurse rostering there are typically 50000 binary clauses and 10000 long clauses. Having more clauses altogether, and in a greater ratio of long to binary, weakens the LP relaxation of nurse rostering. Binary clauses are easy for MIP as they have a special encoding $x \geq y$ whereas long clauses have a relatively weak encoding e.g. $x + y + z \geq 1$ (consider setting all variables to 0.5).

Another experiment confirmed the importance of using the most specialized algorithm for the flow subproblems. Over the suite of satisfaction problems, at

least where ratios could be calculated in the absence of timeouts, and using a static search to reduce measurement noise: Replacing Ford and Fulkerson's algorithm by Network Simplex caused a mean 7.8× slowdown, disabling Tarjan's algorithm cost another 17× slowdown, then replacing Network Simplex by Simplex cost a further 3.0× slowdown (noting that we haven't implemented Tarjan's for the general LP propagator since it requires network structure).

9 Conclusions

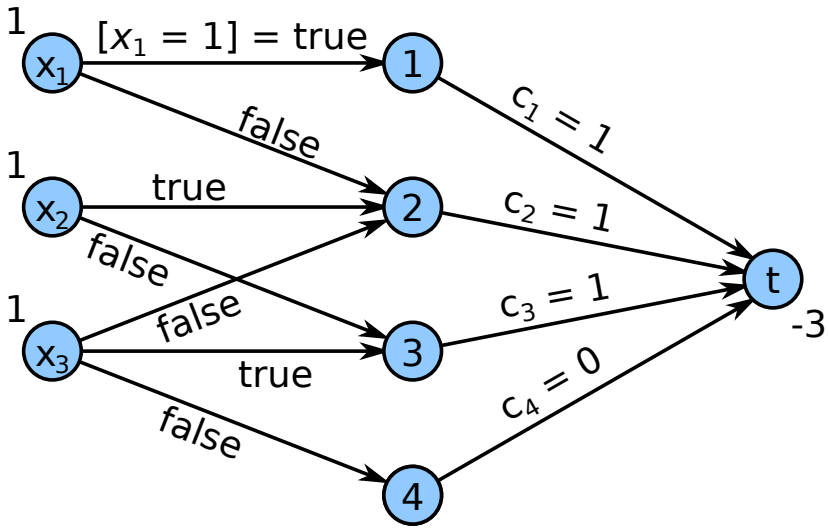
It is by now established that learning changes the tradeoffs for propagation and search. Despite the fact that learning tends to favour decomposition into smaller constraints (even if they propagate to a weaker consistency), we found that our monolithic network flow propagator worked extremely well on the problems considered here, in particular problems which decompose into flow networks over equality literals $[x = k]$, where our methods enforce domain consistency.

This research aimed at drawing together the previous work on flow-based *alldifferent* and *gcc* constraints [18], generic flow networks [2, 22] and explanations for flows [12, 20] and general LPs [1, 5], into a unified, state-of-the-art, propagator. Our results show that enabling all features together gives improvement on problems which are good for CP. Our methods are also more competitive than traditional CP on problems good for MIP, and in some cases execute faster and/or produce better solutions than the best MIP solver.

References

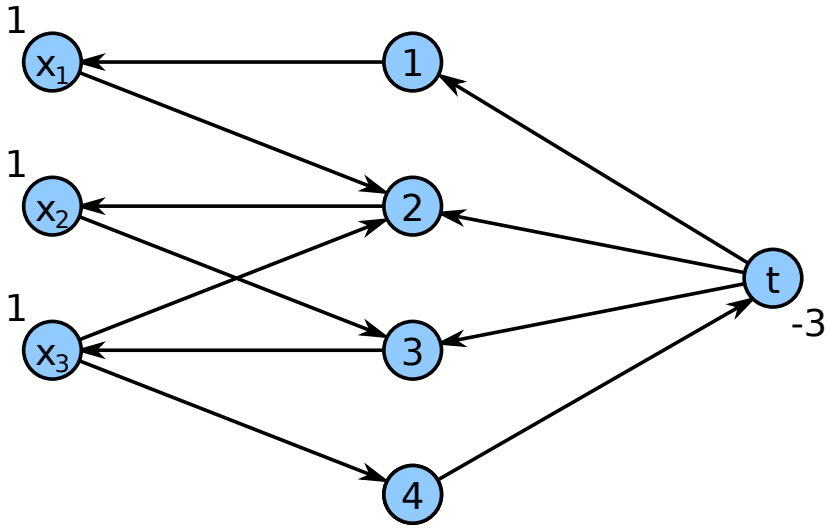
1. Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optimization 4(1), 4–20 (2007)
2. Bockmayr, A., Pisaruk, N., Aggoun, A.: Network Flow Problems in Constraint Programming. In: Walsh, T. (ed.) Proc. CP01, LNCS, vol. 2239, pp. 196–210 (2001)
3. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting Systematic Search by Weighting Constraints. In: Proc. ECAI04. pp. 146–150 (2004)
4. Brand, S., Narodytska, N., Quimper, C.G., Stuckey, P., Walsh, T.: Encodings of the Sequence Constraint. In: Bessière, C. (ed.) Proc. CP07, LNCS, vol. 4741, pp. 210–224 (2007)
5. Davey, B., Boland, N., Stuckey, P.: Efficient Intelligent Backtracking Using Linear Programming. IJOC 14(4), 373–386 (2002)
6. Ford, L., Fulkerson, D.: Maximal flow through a network. Canad. J. Math. 8, 399–404 (1956)
7. Gauthier, J.M., Ribière, G.: Experiments in mixed-integer linear programming using pseudo-costs. Mathematical Programming 12, 26–47 (1977)
8. Gent, I.P.: Two Results on Car-sequencing Problems. Technical report APES-02-1998, Dept. of CS, University of Strathclyde, UK (1998)
9. Gent, I.P., Walsh, T.: CSPLIB: A Benchmark Library for Constraints. In: Princ. and Prac. of CP. pp. 480–481 (1999)
10. Giunchiglia, E., Maratea, M., Tacchella, A.: (In)Effectiveness of Look-Ahead Techniques in a Modern SAT Solver. In: Rossi, F. (ed.) Proc. CP03, LNCS, vol. 2833, pp. 842–846 (2003)
11. Gravel, M., Gagné, C., Price, W.L.: Review and Comparison of Three Methods for the Solution of the Car Sequencing Problem. J.O.R.Soc. 56(11), 1287–1295 (2005)

12. Katsirelos, G.: Nogood processing in CSPs. Ph.D. thesis, University of Toronto, Canada (2008)
13. Löbel, A.: MCF 1.3 - A network simplex implementation (2004), available free of charge for academic use. `http://www.zib.de/loebel`
14. Maher, M., Narodytska, N., Quimper, C.G., Walsh, T.: Flow-Based Propagators for the SEQUENCE and Related Global Constraints. In: Stuckey, P. (ed.) Proc. CP08, LNCS, vol. 5202, pp. 159–174 (2008)
15. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proc. DAC01. pp. 530–535 (2001)
16. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14, 357–391 (2009)
17. Refalo, P.: Impact-Based Search Strategies for Constraint Programming. In: Wallace, M. (ed.) Proc. CP04. LNCS, vol. 3258, pp. 557–571 (2004)
18. Régin, J.C.: Generalized arc consistency for global cardinality constraint. In: Proc. AAAI96. pp. 209–215 (1996)
19. Régin, J.C., Puget, J.F.: A filtering algorithm for global sequencing constraints. In: Smolka, G. (ed.) Proc. CP97, LNCS, vol. 1330, pp. 32–46 (1997)
20. Rochart, G.: Explications et programmation par contraintes avancée (in French). Ph.D. thesis, Université de Nantes, France (2005)
21. Smith, B.: Succeed-first or Fail-first: A Case Study in Variable and Value Ordering. In: Proc. PACT97. pp. 321–330 (1997)
22. Steiger, R., van Hoeve, W.J., Szymanek, R.: An efficient generic network flow constraint. In: Proc. SAC11. pp. 893–900 (2011)
23. Tarjan, R.E.: Depth-First Search and Linear Graph Algorithms. SIAM J. Computing 1(2), 146–160 (1972)
24. Vanhoucke, M., Maenhout, B.: NSPLib – A Nurse Scheduling Problem Library: A tool to evaluate (meta-)heuristic procedures. In: Proc. ORAHS05 (2005)



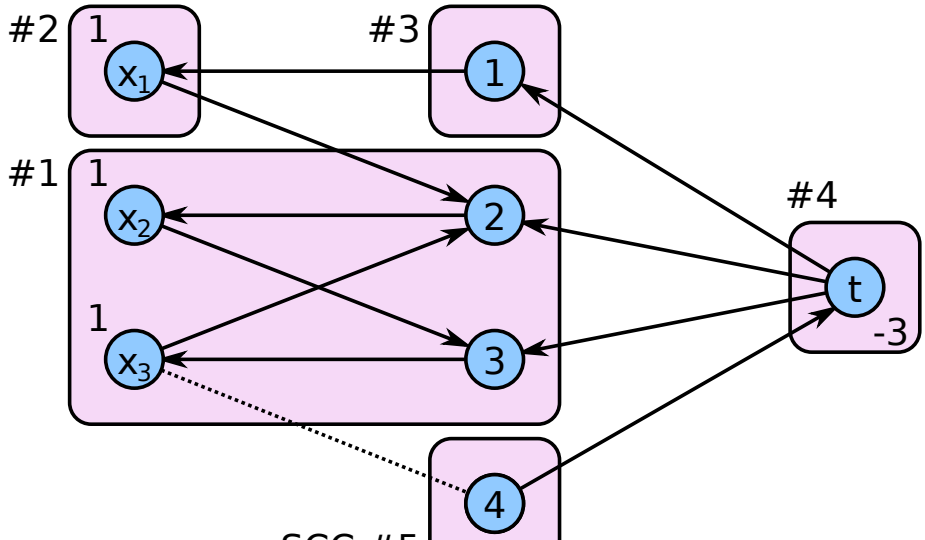
.33

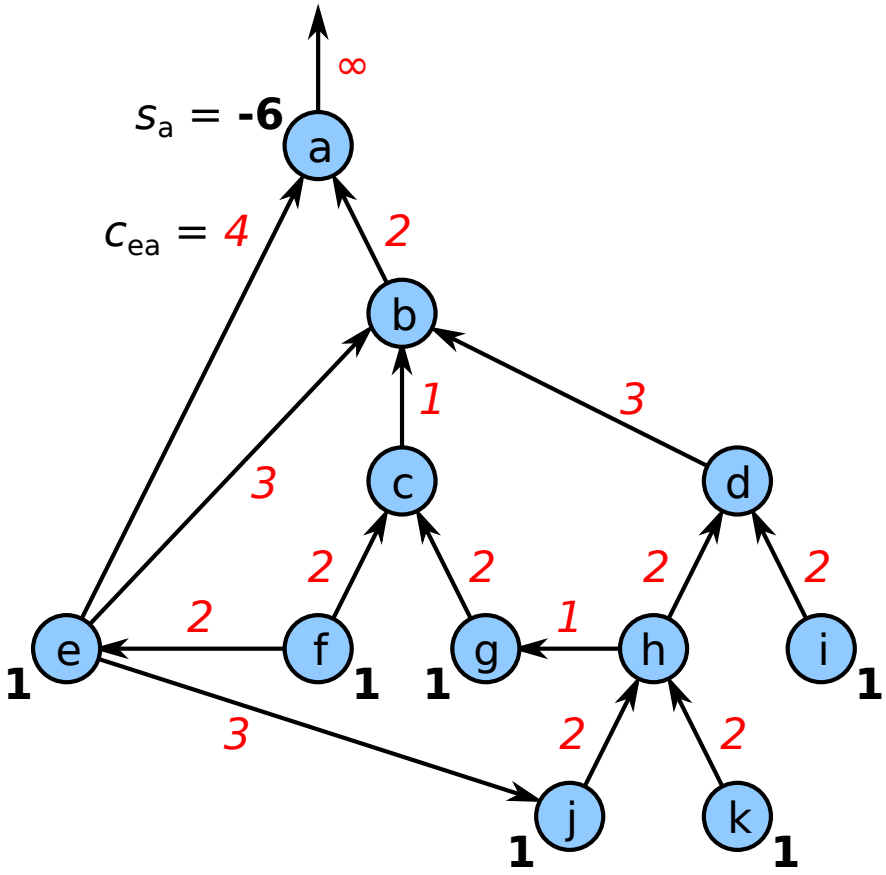
Fig. 8: *alldifferent* network



.33

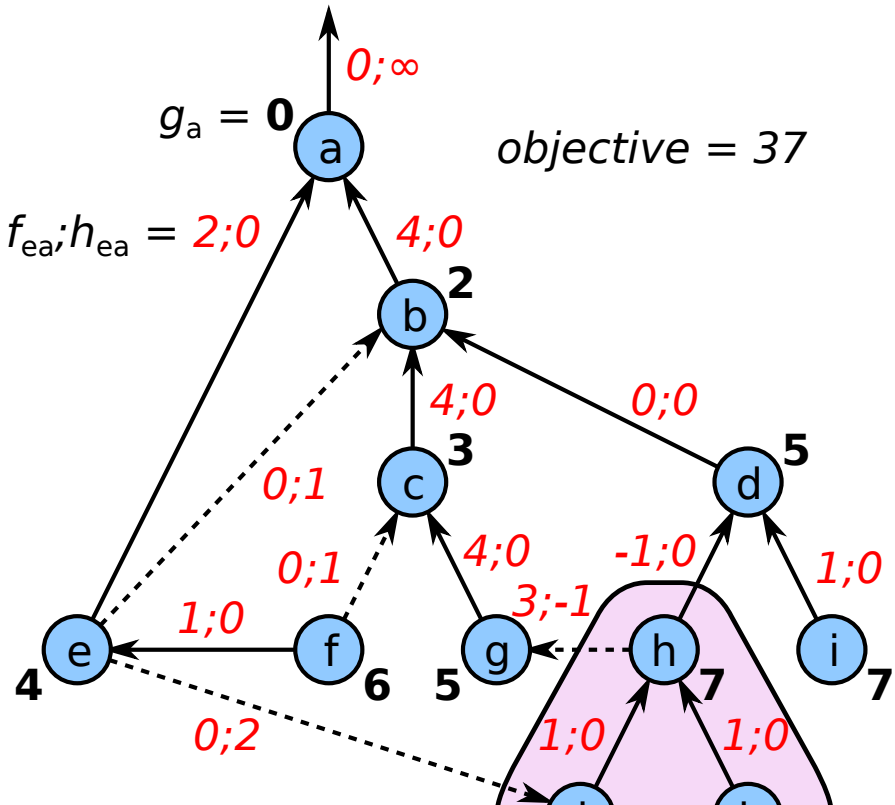
Fig. 9: Residual graph





.33

Fig. 12: Underground network



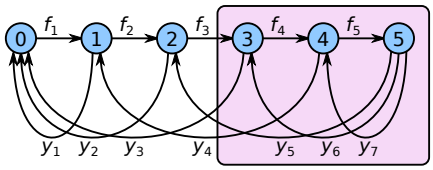


Fig. 17: Flow network encoding a $w = 3, n = 7$ *sequence* constraint

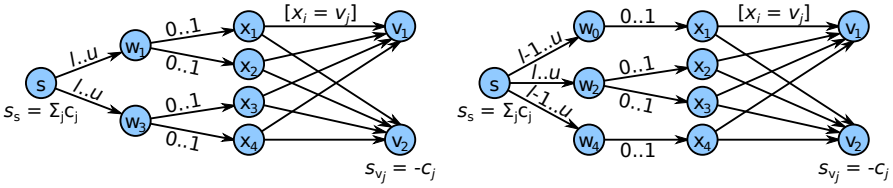


Fig. 18: The w flow networks encoding a $w = 2, n = 4, m = 2$ *gsc* constraint