Constraints

Explaining Circuit Propagation

--Manuscript Draft--

Manuscript Number:	CONS286R3
Full Title:	Explaining Circuit Propagation
Article Type:	Original Research
Keywords:	circuit; Constraint propagation; explanation
Corresponding Author:	Kathryn Glenn Francis University of Melbourne Parkville, VIC AUSTRALIA
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	University of Melbourne
Corresponding Author's Secondary Institution:	
First Author:	Kathryn Glenn Francis
First Author Secondary Information:	
Order of Authors:	Kathryn Glenn Francis
	Peter Stuckey
Order of Authors Secondary Information:	
Abstract:	The circuit constraint is used to constrain a graph represented by a successor for each node, such that the resulting edges form a circuit. Circuit and its variants are important for various kinds of tour-finding, path-finding and graph problems. In this paper we examine how to integrate the circuit constraint, and its variants, into a lazy clause generation solver. To do so we must extend the constraint to explain its propagation. We consider various propagation algorithms for circuit and examine how best to explain each of them. We compare the effectiveness of each propagation algorithm once we use explanation, since adding explanation changes the trade-off between propagation complexity and power. Simpler propagators, although less powerful, may produce more reusable explanations. Even though the most powerful propagator considered for circuit and variants creates huge explanations, we find that explanation is highly advantageous for solving problems involving this kind of constraint.

We have corrected the two errors mentioned by reviewer three.
Thank you for bringing these to our attention.

Constraints manuscript No.
(will be inserted by the editor)

Explaining Circuit Propagation

Kathryn Francis · Peter J. Stuckey

Received: date / Accepted: date

Abstract The *circuit* constraint is used to constrain a graph represented by a successor for each node, such that the resulting edges form a circuit. Circuit and its variants are important for various kinds of tour-finding, path-finding and graph problems. In this paper we examine how to integrate the *circuit* constraint, and its variants, into a lazy clause generation solver. To do so we must extend the constraint to explain its propagation. We consider various propagation algorithms for *circuit* and examine how best to explain each of them. We compare the effectiveness of each propagation algorithm once we use explanation, since adding explanation changes the trade-off between propagation complexity and power. Simpler propagators, although less powerful, may produce more reusable explanations. Even though the most powerful propagator considered for *circuit* and variants creates huge explanations, we find that explanation is highly advantageous for solving problems involving this kind of constraint.

1 Introduction

The *circuit* constraint is used to constrain a graph represented by a successor for each node, such that the resulting edges form a Hamiltonian circuit. Circuit and its variations are important for various kinds of tour-finding, path-finding and graph problems. The *circuit* constraint is notable in the sense that we are not aware of any decompositions of the constraint which are effective, principally because of the exponential number of possible illegal subtours which must be eliminated. Hence we must rely on global propagators for *circuit*.

The $circuit(\bar{v})$ global constraint requires the variables in its argument list $\bar{v}$ to take values such that each variable's value indicates the index of its successor in a tour visiting all variables. If we consider the graph G with a vertex u_i for each variable v_i and edges (u_i, u_j) where j is in the domain of v_i, a solution to the *circuit* constraint

K. Francis and P. J. Stuckey
National ICT Australia, Department of Computing and Information Systems, The University of Melbourne, Victoria 3010, Australia
Tel.: +61-3-83441300
Fax: +61-3-93481184
E-mail: {k.francis@pgrad,pjs@}csse.unimelb.edu.au

is a Hamiltonian cycle of G. The *circuit* constraint is a special case of the *cycle* constraint [5]. The constraint $cycle(n, \bar{u})$ holds if each node in $\bar{u}$ has a distinct successor, and the number of different (including self) cycles is n, hence $circuit(\bar{v}) = cycle(1, \bar{v})$.

The *circuit* constraint has a number of closely related variants: *path* looks for a Hamiltonian path in a graph; *subcircuit* looks for a simple cycle in a graph not including all nodes; while *subpath* looks for a path in the graph. All the propagation algorithms for these constraints are highly related (and indeed we can use the propagators for *circuit* and *subcircuit* for *path* and *subpath* respectively). Hence we examine propagation for each of these variants as well.

Lazy clause generation [15] is a hybrid constraint solving technique combining finite domain propagation with Boolean satisfiability techniques. A propagator in a lazy clause generation solver reports a reason for every propagation step in the form of a Boolean clause. These clauses can then be used to generate nogoods and guide search in the manner of a SAT solver. For more background on propagation based solvers, SAT solvers, and lazy clause generation, see [18], [13], and [15], respectively.

Clearly the definition of *circuit* implies that the *alldifferent* constraint must also hold for *circuit* to be satisfied, since every variable must have a different successor in the circuit. Propagators for the *circuit* constraint typically re-use *alldifferent* propagation algorithms, using a separate algorithm to prevent subtours. For the purpose of this article we focus on this circuit specific part of the propagation, as *alldifferent* propagation is already well studied, including an investigation into its interaction with lazy clause generation (see [21] and [10,6]).

The next section gives a brief description of propagation based constraint solving and how this is enhanced with nogood learning. Section 3 discusses modelling with *circuit* and introduces the example used for experiments. Section 4 discusses variants on *circuit* and their relationship with *circuit*. Section 5 introduces different propagators for *circuit* and its variants, and explores how best to explain the resulting propagation. In Section 6 we examine the trade-off between propagation complexity and pruning strength, and compare the propagators with and without explanation. Section 7 discusses related constraints to the *circuit* constraint. Finally Section 8 concludes.

2 Propagation-based constraint solving and Learning

In this section we briefly introduce propagation-based constraint solving and how it is extended to add learning in a lazy clause generation solver [15].

2.1 Propagation-based constraint solving

We consider a set of integer variables $\mathcal{V}$. A *domain* D is a complete mapping from $\mathcal{V}$ to finite sets of integers. Let D_1 and D_2 be domains and $V \subseteq \mathcal{V}$. We say that D_1 is *stronger* than D_2, written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in \mathcal{V}$. Similarly if $D_1 \sqsubseteq D_2$ then D_2 is *weaker* than D_1. We use *range* notation: $[\, l \,..\, u \,]$ denotes the set of integers $\{d \mid l \leq d \leq u, d \in \mathcal{Z}\}$. We assume an *initial domain* D_{init} such that all domains D that occur will be stronger i.e. $D \sqsubseteq D_{init}$.

A *valuation* θ is a mapping of variables to values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. We extend the valuation θ to map expressions or constraints involving the variables in the natural way. Let *vars* be the function that returns the set of variables appearing in

an expression, constraint or valuation. In an abuse of notation, we define a valuation θ to be an element of a domain D, written $\theta \in D$, if $\theta(v) \in D(v)$ for all $v \in vars(\theta)$. We say a variable v is *fixed* by domain D if $|D(v)| = 1$, since it implies that v takes the same value for any valuation $\theta \in D$.

A constraint c is a set of valuations over $vars(c)$ which give the allowable values for a set of variables. In finite domain propagation constraints are implemented by propagators. A propagator f for c is a contracting and weakly monotonic [19] function over domains such that for all domains $D \sqsubseteq D_{init}$: $f(D) \sqsubseteq D$ and no solutions are lost, i.e. $\{\theta \in D \mid \theta \in c\} = \{\theta \in f(D) \mid \theta \in c\}$. The power of propagation-based constraint solving arises from the fact that a propagator for each constraint is completely independent of other propagators. The focus of this paper will be on propagators for the *circuit* constraint.

2.2 Lazy Clause Generation

Lazy clause generation [15] works as follows. Propagators are considered as clause generators for an underlying SAT solver. Instead of applying propagator f to domain D to obtain $f(D)$, whenever $f(D) \neq D$ we build a clause that encodes the change in domains. In order to do so we must link the integer variables of the finite domain problem to a Boolean representation.

We represent an integer variable x with domain $D_{init}(x) = [\, l \,..\, u \,]$ using the Boolean variables $[\![x = l]\!], \ldots, [\![x = u]\!]$ and $[\![x \leq l]\!], \ldots, [\![x \leq u - 1]\!]$. The variable $[\![x = d]\!]$ is true if x takes the value d, and false for a value different from d. Similarly the variable $[\![x \leq d]\!]$ is true if x takes a value less than or equal to d and false for a value greater than d. The inclusion of these bounds literals is the main feature distinguishing lazy clause generation from other attempts to incorporate nogood learning in an FD solver, such as [9]. Note we will use the notation $[\![x \neq d]\!]$ as shorthand for the literal $\neg[\![x = d]\!]$.

Not every assignment of Boolean variables is consistent with the integer variable x, for example $\{[\![x = 3]\!], [\![x \leq 2]\!]\}$ (i.e. both Boolean variables are true) requires that x is both 3 and ≤ 2. In order to ensure that assignments represent a consistent set of possibilities for the integer variable x we add to the SAT solver the clauses $DOM(x)$ that encode $[\![x \leq d]\!] \rightarrow [\![x \leq d + 1]\!], l \leq d < u$, $[\![x = l]\!] \leftrightarrow [\![x \leq l]\!]$, $[\![x = d]\!] \leftrightarrow ([\![x \leq d]\!] \wedge \neg[\![x \leq d - 1]\!]), l < d < u$, and $[\![x = u]\!] \leftrightarrow \neg[\![x \leq u - 1]\!]$ where $D_{init}(x) = [\, l \,..\, u \,]$. We let $DOM = \cup\{DOM(v) \mid v \in \mathcal{V}\}$.

Any assignment A on these Boolean variables can be converted to a domain: $domain(A)(x) = \{d \in D_{init}(x) \mid \forall [\![c]\!] \in A, vars([\![c]\!]) = \{x\} : x = d \models c\}$, that is, the domain includes all values for x that are consistent with all the Boolean variables related to x. It should be noted that the domain may assign no values to some variable.

Example 1 Assume $D_{init}(x_1) = D_{init}(x_2) = [\, 0 \,..\, 10 \,]$. The assignment $A = \{[\![x_1 \leq 5]\!]$, $\neg[\![x_1 \leq 1]\!]$, $\neg[\![x_1 = 4]\!]$, $[\![x_2 \leq 7]\!]$, $\neg[\![x_2 \leq 3]\!]$ $\}$ is consistent with $x_1 = 2, x_1 = 3$ and $x_1 = 5$. Hence $domain(A)(x_1) = \{2, 3, 5\}$. Similarly $domain(A)(x_2) = [\, 4 \,..\, 7 \,]$. □

In lazy clause generation a propagator is extended to not only map from domains to domains but also to generate clauses describing the reasons for the propagations. When $f(D) \neq D$ we assume the propagator f can determine a set of clauses C which explain the domain changes.

Example 2 Consider the propagator f for $x_1 \leq x_2 - 3$. When applied to domain D of Example 1 it obtains $f(D)(x_1) = \{2, 3\}$, $f(D)(x_2) = [4..7]$. The clausal explanation of the change in domain of x_1 is $[\![x_2 \leq 7]\!] \rightarrow [\![x_1 \leq 4]\!]$, This becomes the clause $\neg[\![x_2 \leq 7]\!] \vee [\![x_1 \leq 4]\!]$. □

In practice the lazy clause generation solver keeps track of the domains D of variables $\mathcal{V}$ and the equivalent state A of the Booleans in DOM ($D = domain(A)$). When a propagator detects an inconsistency and triggers failure, it provides an explanation $c \rightarrow \mathit{false}$ where c is a conjunction of true literals. This initial nogood c is transformed by the learning process into an equivalent nogood containing at most one literal which became true at the current decision level. This is achieved by repeatedly selecting from c a literal l which was set true by propagator p, asking p to provide an explanation in the form of a conjunction of literals $l_1 \wedge \cdots \wedge l_n$ which imply l, and then replacing l in c with the conjunction $l_1 \wedge \cdots \wedge l_n$. Once the learning process has derived a final nogood, the solver adds this nogood to the constraint store and backtracks or backjumps.

Example 3 Consider a problem containing constraints $x_1 \leq x_2 - 3$, $x_1 \leq x_4$, $x_1 + x_2 + x_3 \geq 14$ with initial domains $D_{init}(x_1) = [0..7]$, $D_{init}(x_2) = D_{init}(x_3) = D_{init}(x_4) = [0..10]$. Suppose search chooses to set $[\![x_3 \leq 2]\!]$, there is no propagation. Then if search sets $[\![x_4 \leq 6]\!]$, this then propagates $[\![x_1 \leq 6]\!]$ using $x_1 \leq x_4$, and then $[\![x_2 \geq 6]\!]$ using $x_1 + x_2 + x_3 \geq 14$. Now assume search sets $[\![x_2 \leq 7]\!]$, this propagates $[\![x_1 \leq 4]\!]$ using $x_1 \leq x_2 - 3$ and then fails using $x_1 + x_2 + x_3 \geq 14$. The initial nogood is $[\![x_3 \leq 2]\!] \wedge [\![x_2 \leq 7]\!] \wedge [\![x_1 \leq 4]\!] \rightarrow \mathit{false}$ returned by the propagator for $x_1 + x_2 + x_3 \geq 14$. To remove $[\![x_1 \leq 4]\!]$ we ask the propagator that set it ($x_1 \leq x_2 - 3$) for an explanation $[\![x_2 \leq 7]\!] \rightarrow [\![x_1 \leq 4]\!]$ and replace obtaining $[\![x_3 \leq 2]\!] \wedge [\![x_2 \leq 7]\!] \rightarrow \mathit{false}$. Since this only contains one literal true at the last level we add it to the constraint store and backtrack.

□

The advantages of lazy clause generation over a standard FD solver (e.g. [18]) are that we automatically have the nogood recording and backjumping ability of the SAT solver applied to our FD problem. We can also use activity counts from the SAT solver to direct the FD search.

3 Modelling with *circuit*

The *circuit* constraint has a Zinc/MiniZinc [12,14] definition of the form

```
predicate circuit(array[int] of var int: succ);
```

where the array of integer variables `succ` represents nodes in a graph numbered from 1 to n, where n is the length of the array. The initial domain of variable `succ`$[i]$ represents the possible successors of node i, and hence is a subset of $1..n$. The constraint ensures that the edges represented by the values of the successor variables form a Hamiltonian cycle of the nodes, that is a complete circuit visiting every node once.

Consider the problem of designing a tour of a set of locations. We imagine a tour company has selected important sites which should be included in the tour, and wishes to find an order to visit the sites so that each site is visited exactly once, and the length of the longest leg is minimised (as their clients do not like sitting in a bus for a long time). We assume a transport network which is not complete. That is, it is not always

```
include "circuit.mzn";
int: n;              % number of locations
set of int: Locations = 1..n;
int: maxLegLen;      % length of longest edge in network

% travel times between locations
% -1 means no direct connection exists
array[Locations,Locations] of int: travelTime;

% successor variables
array[Locations] of var Locations: succ;

% only use allowed legs
constraint forall(loc1, loc2 in Locations)
  ( travelTime[loc1,loc2] < 0 -> succ[loc1] != loc2 );

% successors must form a circuit
constraint circuit(succ);

% variable for the length of the longest leg
var 1..maxLegLen: maxleg;
constraint forall(loc1, loc2 in Locations)
  ( succ[loc1] == loc2 -> maxleg >= travelTime[loc1,loc2] );

solve minimize maxleg;
```

Fig. 1: MiniZinc model for tour design problem

possible to travel from one site to another directly, without passing through one or more other sites. Even passing through an already visited site is not allowed, as this would be frustrating to clients.

A MiniZinc [14] model for this problem is shown in Figure 1. The only global constraint is the *circuit* constraint. All other constraints are (reified) binary constraints.

We shall use this model to explore design alternatives for *circuit* and its variants. Since it is dominated by the global *circuit* constraint, it is an effective model for such evaluation.

The underlying graph used for the transport network is important. Completely random graphs do not make very realistic transport networks, because there is no consistency in the distances between nodes, and in which nodes are connected to each other. We have used a more realistic technique to generate networks for our benchmarks. We first randomly distributed the locations in a two dimensional space, and calculated the Euclidean distance between each pair. Edges were then added so that every node was connected to its seven closest neighbours. To keep the instances more similar in difficulty we then added extra edges to ensure the existence of at least one Hamiltonian circuit (to make the instance satisfiable). We achieved this by performing a random walk of the graph, adding a randomly chosen new edge whenever all existing edges from the current node lead to already visited nodes, and then adding an edge from the final node back to the initial node. All data files used in our experiments are available online.[1]

[1] www.cs.mu.oz.au/~pjs/circuit

4 Variations of *circuit*

In this paper we consider not only the *circuit* constraint, but close variations, each of which is introduced below along with a corresponding variation to our example problem. The propagation algorithms for all variants are similar. Figure 3 provides an illustration of solutions satisfying each variation of the constraint.

4.1 The *path* constraint

The *circuit* constraint can be trivially extended to implement a *path* constraint. The *path* constraint ensures that when the value of each variable is interpreted as the index of its successor, the result is a single path including all variables.

The *path* constraint is defined as

```
predicate path(array[int] of var int: succ,
               var int: start, var int: end) =
          circuit(succ ++ [start]) /\
          succ[end] = length(succ)+1;
```

where the `succ` array is a successor relationship, `start` is the number of the first node in the path, and `end` is the number of the last node in the path. Each variable ranges over $1..n+1$ where n is the length of the array `succ`. The successor relationship defines a single path from node `start` to node `end`. This is achieved by adding a dummy node (with index $n+1$) to the graph. Its successor, given by variable `start`, is added at the end of the array of original variables (using MiniZinc syntax `succ ++ [start]`), and we then apply the standard *circuit* constraint. In a solution, the value of the `start` variable gives the start of the path, and the successor of the final node in the path `end` is the dummy node (so index $n+1$, as this is the index of the `start` variable).

Note that this *path* constraint is a special case of the path constraint listed in the global constraints catalog [5]. The more general constraint also accepts an argument which is the number of paths required (in our case this is always 1).

Returning to our tour design example, the *path* constraint is applicable when the tour is not required to start and finish in the same location. The start and end locations for the tour may be unconstrained, they may be fixed, or there may be a limited number of options (e.g. cities with international airports). For simplicity we consider the case where the start and end locations are completely open, so the MiniZinc model is exactly the same, except that the call to *circuit* is replaced with a call to *path* with `start` and `end` unconstrained, and the domain of the `succ` variables extended by one.

4.2 The *subcircuit* constraint

The *circuit* constraint is only applicable when all variables are required to participate in the circuit. We next consider a variation of *circuit*, which we call *subcircuit*, which drops this requirement.

The *subcircuit* constraint has a Zinc/MiniZinc [12,14] definition of the form

```
predicate subcircuit(array[int] of var int: succ);
```

where the argument succ is the same as for *circuit*.

In *subcircuit*, a variable excluded from the circuit is required to take its own index as its value, hence it in effect appears in a self cycle. This means that the *alldifferent* constraint still holds for *subcircuit* and prevents a variable not included in the circuit from becoming the successor of another variable, since a variable v having its own index as a value prevents any other variable having v as a successor.

Changes are required to the *circuit* propagation algorithms (and therefore also the explanations generated) to make them applicable for *subcircuit*. These will be discussed in Section 5 where we describe propagation algorithms in detail.

The *subcircuit* constraint applies to our tour design problem when instead of visiting every location in the network, a tour is required to visit some subset of the locations satisfying further constraints. For example, say we have a set of activities required to be included in the tour (e.g. shopping, beach visit, museum), and each activity is only available in some locations. The task is to find a tour covering at least one location providing each activity, while minimising the length of the longest leg. This is the problem we will use to evaluate *subcircuit* propagation and explanation. Our MiniZinc implementation is provided in Figure 2.

4.3 The *subpath* constraint

The *subpath* constraint is an extension of *subcircuit* equivalent to the *path* extension for *circuit*. The successor variables are required to form a path, and any node not in the path must have itself as its successor. As with *path*, *subpath* is implemented by adding a dummy node $n + 1$ with successor variable start, and then applying the *subcircuit* constraint. For *subpath* we also need to also ensure that the dummy node is included in the circuit. This is easily achieved by limiting the domain of the start variable to only the original nodes.

The *subpath* constraint is defined analogously to the *subcircuit* constraint, that is

```
predicate subpath(array[int] of var int: succ,
                  var int: start, var int: end) =
          subcircuit(succ ++ [start]) /\
          start <= length(succ) /\
          succ[end] = length(succ)+1;
```

For a version of the tour design problem appropriate for *subpath*, we again simply drop the requirement that the tour must start and finish in the same location.

5 Propagating and explaining *circuit*

In the following sections we consider three complementary algorithms for propagating circuit, in order of increasing complexity. For each we define the algorithm and then examine various alternatives for adding explanations, using experimental results to justify our final decisions.

In order to reduce the risk of making design decisions which are only justified for a specific search strategy, the experiments are repeated using two different search strategies. The first is in-order labelling of variables using minimum values first. This simple fixed strategy allows us to more easily isolate the effect of different design

```
include "subcircuit.mzn";
int: n;              % number of locations
set of int: Locations = 1..n;
int: m;              % number of activities
set of int: Activities = 1..m;
int: maxLegLen;      % length of longest edge in network

% travel times between locations
% -1 means no direct connection exists
array[Locations,Locations] of int: travelTime;

% activity locations
array[Activities,Locations] of bool: activityAvailable;

% successor variables
array[Locations] of var Locations: succ;

% only use allowed legs
constraint forall(loc1, loc2 in Locations)
  ( travelTime[loc1,loc2] < 0 -> succ[loc1] != loc2 );

% visit at least one location with every activity
constraint forall(act in Activities)
  ( exists(loc in Locations where activityAvailable[act,loc])
          (succ[loc] != loc) );

% successors must form a circuit
constraint subcircuit(succ);

% variable for the length of the longest leg
var 1..maxLegLen: maxleg;
constraint forall(loc1, loc2 in Locations)
  ( succ[loc1] == loc2 -> maxleg >= travelTime[loc1,loc2] );

solve minimize maxleg;
```

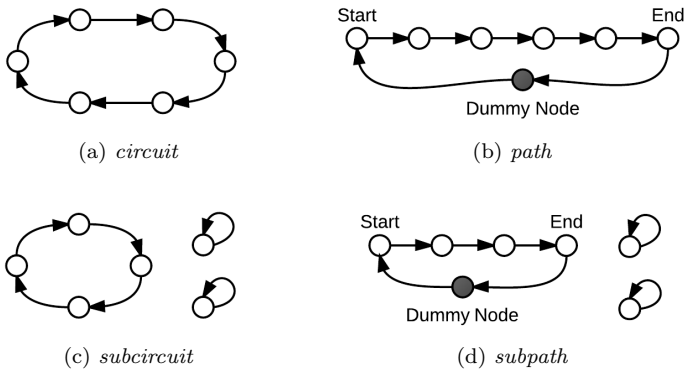
Fig. 2: MiniZinc model for a version of the tour design problem where not all locations are required to be visited as long as at least one location is visited which provides each of a set of activities

decisions, as the interaction between the design decision and the search strategy is relatively easy to understand. It also allows us to compare versions of the propagator with and without learning. The second strategy is activity based dynamic search. This is the most effective autonomous search for lazy clause generation solvers, and is highly dynamic.

All experiments in the paper were carried out on a 2.8GHz AMD 6-Core Opteron 4184 CPU with 64GB of memory, using (except where otherwise stated) the lazy clause generation solver Chuffed.

Throughout this section we use the following notation: V is the set of all nodes (or variable indices), $n = |V|$, x_i where $i \in V$ is the variable holding the successor of node i, and $value(x_i)$ is the (fixed) value of variable x_i in the current domain D.

Note that all of the algorithms discussed below assume the *alldifferent* constraint has already been propagated. We use the existing implementation of domain consistent *alldifferent* for all experiments. Since it is not possible to explain only some propaga-



(a) *circuit* (b) *path*

(c) *subcircuit* (d) *subpath*

Fig. 3: Solutions to variations of the *circuit* constraint.

tions in CHUFFED, whenever explanations are used for *circuit* they are also used for *alldifferent* (and all other constraints in the model).

5.1 The *check* algorithm

A very simple and cheap propagator for the *circuit* constraint fires only on variable fixing, and simply follows the chain of fixed variables starting at this newly fixed variable, until an unfixed variable is found, or a loop is detected. If a loop is detected with length less than the number of nodes, the propagator reports a conflict. We call this propagation algorithm *check*.

For *subcircuit* (and *subpath*), a cycle which excludes some nodes is allowed if all excluded nodes have themselves as successor. Therefore when a cycle of fixed variables is found in the *subcircuit* version of the *check* algorithm, instead of reporting failure we set the value of all excluded variables to their own indices. If this is not possible for some variable, then a conflict is reported.

5.1.1 Explanations for check

We consider two alternative explanations for *check* propagation, shown below. Here C is the set of nodes included in the small cycle.

$$\bigwedge_{i \in C} [\![x_i = value(x_i)]\!] \rightarrow false \tag{1}$$

$$\bigwedge_{i \in C, j \in V \setminus C} [\![x_i \neq j]\!] \rightarrow false \tag{2}$$

Clause 1 says that the successor variables for nodes in the cycle taking their current values leads to failure. The second option is more general (but also larger - $O(n^2)$ rather than $O(n)$), indicating that the fact that no node inside the cycle has a successor outside the cycle is sufficient to cause failure.

For *subcircuit* we can use very similar clauses. The two options are shown below.

$$\bigwedge_{i \in C} [\![x_i = value(x_i)]\!] \rightarrow [\![x_k = k]\!], \quad k \in V \setminus C \tag{3}$$

$$\bigwedge_{i \in C, j \in V \setminus C} [\![x_i \neq j]\!] \rightarrow [\![x_k = k]\!], \quad k \in V \setminus C \tag{4}$$

Table 1 shows experimental results comparing these two alternatives for each variation of *circuit*. It seems clear from the table that the second alternative (clauses 2 and 4) is better for all variations of our problem, as for both search strategies the number of failures, number of propagations, and execution time are all much smaller. This result demonstrates that general clauses can be more effective than smaller, more specific clauses. We use clauses 2 and 4 in the remainder of our experiments.

		Inorder Search			VSIDS Search		
Problem	Clause	Fails	Props	Time	Fails	Props	Time
circuit	1	2071	14567	199.8 (151)	903	4928	78.2 (52)
	2	399	2088	105.9 (75)	1	18	0.3
subcircuit	3	5179	39758	429.1 (325)	2394	13263	225.7 (142)
	4	657	3495	215.5 (122)	5	62	0.9
path	1	2301	15584	264.1 (192)	1161	6312	102.5 (70)
	2	505	2993	205.5 (139)	2	26	0.6
subpath	3	4606	28863	443.1 (328)	2507	13880	211.2 (122)
	4	871	4297	409.5 (269)	8	89	1.8

Table 1: Comparison of alternative explanation clauses for the *check* algorithm. Each figure is the average for 500 instances with 50 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

5.1.2 Choosing an explanation for failure

When not using explanation, it makes sense to exit a propagator as soon as failure is detected. However, when using lazy clause generation the explanation we give for failure will affect the clause which is subsequently learned. Therefore if a constraint is violated in multiple ways it may be worth discovering all of these and using a heuristic to choose which violation to report.

For *subcircuit*, when a small cycle is found we report failure if there exists a node outside that cycle which cannot be a self cycle (that is, a node whose successor variable's current domain does not include its own index). By default we have reported failure using the first such node encountered. However, it is possible to instead collect all nodes for which this condition holds, and then make a deliberate selection.

The clause we produce to explain *subcircuit check* conflicts is shown below, where C is the set of nodes in the cycle, and k is the chosen node outside the cycle.

$$\left([\![x_k \neq k]\!] \wedge \bigwedge_{i \in C, j \in V \setminus C} [\![x_i \neq j]\!] \right) \rightarrow false$$

Note that for each node we could choose, the clause will include a different first literal but will otherwise be the same. We can use the properties of the corresponding literals to make a good choice for k.

We know that this clause will cause a conflict, and at that point the solver will compute a learned clause and backjump to some earlier part of the search tree. The position in the search tree we jump to will depend on the levels at which literals involved in the failure became fixed. We would like to backjump as far as possible, as this way we exclude more of the search tree. Therefore it makes sense to choose to include in our explanation of failure the literal which became fixed highest (earliest) in the search tree. This information is already available as it is used when deriving learned clauses.

We tested this theory experimentally, comparing four different selection heuristics for the node to be used to explain a *subcircuit check* conflict:

1. The *first* applicable node. Note that this option entails less overhead as we can report failure immediately upon finding an appropriate node.
2. The *last* applicable node.
3. The node k whose corresponding literal $x_k \neq k$ became fixed *highest* in the search tree. In other words, the node whose successor as itself was excluded the earliest.
4. The node k whose corresponding literal became fixed *lowest* in the search tree (the node whose successor as itself was excluded most recently).

The results are shown in Table 2. As expected, choosing the literal which was fixed highest (earliest) in the search tree is the best option for both *subcircuit* and *subpath*, and choosing the literal which was fixed lowest in the tree is worse than all other options for both problems.

When using inorder search for *subcircuit*, choosing the first applicable node is better than choosing the last. This is probably because an inorder search means successor variables for nodes earlier in the list will often become fixed higher in the search tree.

For *subpath*, choosing the last node works quite well (and better than choosing the first). This is surprising until you realise that the last node in the list is the dummy node (`start`), which is never allowed to have itself as a successor. This means that whenever a small cycle is found which does not include the dummy node, choosing the last node gives an ideal explanation for the failure, because the corresponding literal is fixed at the root node.

In all further experiments, we use the highest node heuristic to choose a literal for *subcircuit* and *subpath* failure explanations.

5.2 The *prevent* algorithm

We now consider a slightly stronger propagation algorithm, described in [4], which we call *prevent*. This algorithm finds the start and end of each chain of nodes with fixed successors, and removes the first node of each chain from the domain of the successor variable for the end node of that chain (unless the chain includes all variables). This prevents the chain from becoming a subcycle.

The circuit propagator in the open source constraint solver Gecode (3.5.0) [17] uses a clever technique to find distinct chains, exploiting the fact that a chain must start with a node which is not the fixed value of any other successor variable. Assuming that *alldifferent* has been propagated, the set of all possible starts of chains is the union

		Inorder Search			VSIDS Search		
Problem	Heuristic	Fails	Props	Time	Fails	Props	Time
subcircuit	first	1220	6564	274.5 (159)	7	82	1.3
	last	1259	6733	430.7 (288)	7	82	1.2
	lowest	1183	6030	481.0 (330)	16	139	2.3
	highest	922	5737	212.8 (123)	5	70	1.0
subpath	first	1095	5541	501.2 (359)	9	108	2.0
	last	1041	5325	458.0 (321)	5	68	1.5
	lowest	1189	5488	561.2 (430)	21	174	4.1
	highest	922	5419	397.3 (260)	4	59	1.2

Table 2: Comparison of alternative heuristics for choosing an explanation of failure detected by the *check* algorithm for *subcircuit* and *subpath*. Each figure is the average for 500 instances with 55 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

of the domains of the unfixed successor variables. We used this method in our implementation as well, with the only drawback being that because chains that are already cycles are not explored, this algorithm is not complete and must be accompanied by either the *check* algorithm from above, or the stronger algorithm described in the next section. In our experiments, the *prevent* algorithm is always accompanied by *check*.

5.2.1 Applying prevent *to* subcircuit

The *subcircuit* version of the *prevent* algorithm cannot perform any propagation unless there exists outside the chain a node k which must be included in the circuit (because its successor variable's current domain does not include itself). We call this node the *evidence* node, and for *evidence* node k we refer to x_k as the *evidence* variable, and $[\![x_k \neq k]\!]$ as the *evidence* literal.

5.2.2 Explaining prevent

For the *circuit* version of *prevent* we again have two different options for explanations. In the following, C is the set of nodes in the fixed chain, a is the first node, and z is the last node in the chain (so its successor variable is not fixed).

$$\bigwedge_{i \in C, i \neq z} [\![x_i = value(x_i)]\!] \rightarrow [\![x_z \neq a]\!] \tag{5}$$

$$\bigwedge_{i \in C, i \neq z, j \in V \setminus C} [\![x_i \neq j]\!] \rightarrow [\![x_z \neq a]\!] \tag{6}$$

The first clause (5) says that the last node is not allowed to have the first node as a successor because of the current choice of successor for the other nodes in the chain. The second clause (6) says that the last node is not allowed to have the first as a successor because none of the other nodes in the chain have a possible successor outside the chain. If none of the other nodes lead outside the set included in the chain, then the final node must do so, and therefore it cannot have the first node in the chain as a successor.

		Inorder Search			VSIDS Search		
Problem	Clause	Fails	Props	Time	Fails	Props	Time
circuit	5	354	2434	72.8 (50)	0.3	11.8	0.25
	6	316	2148	75.0 (50)	0.3	12.0	0.28
path	5	533	4402	212.0 (145)	0.4	18.2	0.37
	6	490	3960	220.1 (151)	0.4	18.5	0.44
subcircuit	5	677	5250	166.6 (86)	1.5	38.3	0.55
	6	708	5458	181.4 (94)	1.5	38.5	0.56
subpath	5	863	6262	377.8 (236)	1.2	36.7	0.67
	6	835	5935	384.3 (236)	1.2	37.0	0.77

Table 3: Comparison between alternative explanation clauses for the *prevent* algorithm. Each figure is the average for 500 instances with 55 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

For the *subcircuit* version of the algorithm, we need to also specify that one of the nodes outside the chain cannot have itself as a successor. The two possible explanations are therefore the same as above but with an extra literal on the left hand side $[\![x_k \neq k]\!]$, where k is a node outside the chain. This is the *evidence* literal mentioned previously.

Clauses 5 and 6 have the same size complexity as the *check* explanation clauses - $O(n)$ and $O(n^2)$ respectively.

The results of experiments evaluating the two options are shown in Table 3. In contrast with the results for *check* propagation, this time the second clause was not more effective. Instead for all problems and for both search strategies the first clause (5) resulted in roughly equal or slightly better performance.

Although the clauses presented for *prevent* seem analogous to the *check* clauses, the second *prevent* explanation (6 above) is less general than the corresponding *check* explanation would be if the chain was later closed into a circuit. The *prevent* explanations both refer specifically to the final successor variable being equal to the first node, whereas only the first *check* explanation does this. The second *check* explanation would simply state that no node inside the cycle reaches any node outside. As stated previously we always use *prevent* in combination with *check* as it is not complete on its own, and in these experiments we used the best discovered options for *check*, which means the more general explanations. Perhaps this is why the more general explanation for *prevent* did not perform all that well.

Since there was very little difference between the two options, and the first option is simpler, we chose to use the first option (Clause 5) in all further experiments.

5.2.3 Choosing an evidence *literal*

For the *subcircuit* version of the *prevent* algorithm, we need to choose an evidence literal to include in our propagation explanation. We experimented with the same four options used for choosing a literal to include in *check* failure clauses. That is, the first appropriate literal, the last, the literal fixed highest (earliest) in the search tree, and the literal fixed lowest (most recently) in the search tree. The results are shown in Table 4.

Although the differences are much smaller than those observed in the *check* case, we again find that the best option is to choose the literal highest in the search tree. Recall that when using inorder search we expect the first and highest options to perform

		Inorder Search			VSIDS Search		
Problem	Heuristic	Fails	Props	Time	Fails	Props	Time
subcircuit	first	677	5250	166.6 (86)	1.5	38.3	0.55
	last	708	5375	172.2 (92)	1.5	37.9	0.53
	lowest	706	5470	181.2 (92)	1.5	38.1	0.54
	highest	668	5211	166.5 (85)	1.4	37.7	0.53
subpath	first	863	6262	377.8 (236)	1.2	36.7	0.67
	last	878	6351	392.6 (243)	1.2	36.9	0.66
	lowest	893	6348	399.6 (244)	1.3	37.3	0.67
	highest	868	6233	375.8 (240)	1.1	36.2	0.64

Table 4: Comparison of alternative heuristics for selecting an evidence literal for *subcircuit prevent* explanations. Each figure is the average for 500 instances with 55 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

similarly. In future experiments we use the *highest* selection method to choose evidence literals for *prevent* explanation.

5.3 The *scc* algorithm

As mentioned previously, a solution to the *circuit* constraint is a Hamiltonian cycle of the graph G where each node v has an edge to all nodes $u \in D(x_v)$. This implies that for *circuit* to be feasible G must have a single strongly connected component, as there is no Hamiltonian cycle in a graph with multiple strongly connected components.

The final propagation algorithm we consider, which we call *scc*, takes advantage of this property. The algorithm is based on Tarjan's depth first search algorithm for finding strongly connected components [20]. If multiple strongly connected components are discovered, the algorithm reports failure. With only minor changes to Tarjan's algorithm, it is also possible to discover further propagation in some cases. As discussed in [19], depending on the current domains of variables and the root node chosen, depth first search may explore multiple disjoint subtrees below the root. Figure 4 shows an example of this, with nodes numbered in the order they are visited, and the subtrees shown as triangles. When this happens, reasoning can be applied to prune links between subtrees which cannot form part of a valid solution, and to enforce links which must be part of any solution.

The following two observations are made in [19].

1. There must be an edge from each subtree to its predecessor subtree, and an edge from the first subtree to the root. Hence no edge to the root from a subtree other than the first can be used.
2. Edges between non-adjacent subtrees are not allowed. That is, if A, B and C are subtrees such that A was visited before B and B was visited before C, then any edge that leads from a C to A is forbidden, because if such an edge were used in the circuit there would be no way to get in and out of B without visiting the root twice.

We add two more observations, both of which provide further opportunity for propagation with only minor alterations to the algorithm.

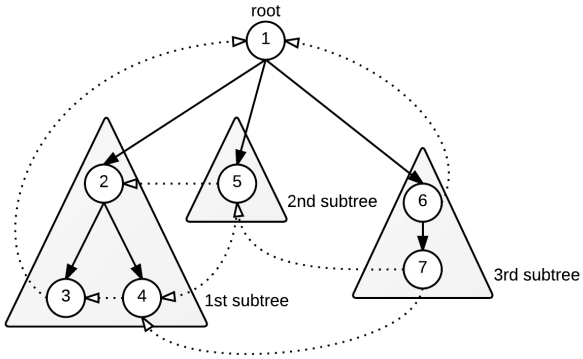
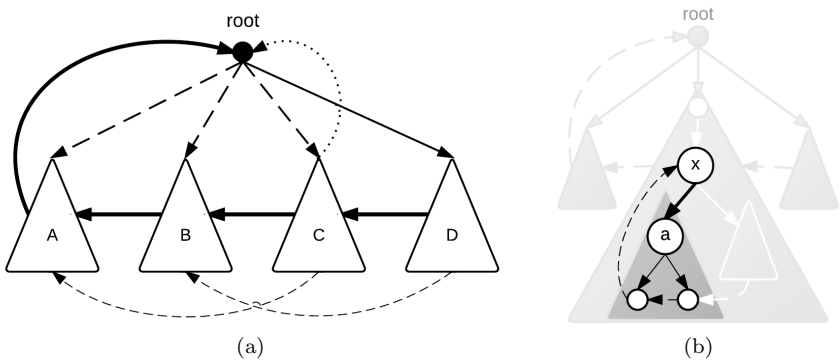


Fig. 4: Example of a case where depth first search explores multiple disjoint subtrees. Nodes are numbered in the order they are visited. Edges followed during the search are solid, while edges leading to already visited nodes are dotted. The three subtrees are shown as triangles.

3. Any solution must include an edge leading from the root node to the last subtree (as there is no other way to reach this subtree). Therefore, since a solution can only include one edge originating at the root node (because the root successor variable can only take one value), any edge leading from the root to earlier subtrees can be pruned.
4. At any node x within the search tree, if exploration of x's first child a does not reach any node above x, then the edge leading from x to a can be pruned. This is because the only way out of the subtree rooted by a is through x, so the circuit must not enter this subtree through x (as x cannot be visited twice). The circuit must instead enter the subtree rooted at a via a back edge from a later subtree.

In the remainder of this paper we will refer to these new opportunities for propagation as *prune root* (rule 3) and *prune within* (rule 4) respectively. Figure 5 illustrates the pruning rules.

We now describe the *scc* algorithm. Pseudo-code for this algorithm is included in Figure 6. The key idea is to keep track of the search index of the first and last node in the previous subtree explored, in order to be able to detect for every back edge found whether the destination is within the current subtree, in the previous subtree, or in an earlier subtree. Any edge to an earlier subtree can be pruned. Edges to the previous subtree are counted and the most recently found is stored. After a subtree has been explored a conflict is reported if there were no back edges, and if there was only one then this edge can be enforced. Removal of edges leading from the root to subtrees before the last is done after exploration is complete, while *prune within* is handled by testing the lowpoint of the first child at each node, and if this lowpoint is not less than the index of the current node, the edge to that child is pruned. Obviously if a strongly connected component is found below the root, or if the search does not reach all variables, a conflict is reported.



(a) (b)

Fig. 5: (a) The SCC exploration graph for *circuit* starting from root. At least one (thick) edge from A to the root, from D to C, C to B, and B to A must exist (rule 1). Backwards (dotted) edges to the root from B, C or D cannot be used (rule 1). The (thin-dashed) edges from C to A and D to B cannot be used (rule 2). The (thick-dashed) edges leading from root to A, B and C cannot be used (rule 3). (b) Illustration of *prune-within* (rule 4). The edge from x to a cannot be used otherwise we cannot escape the subtree rooted at a (dark grey). We need to enter the subtree from elsewhere.

5.3.1 Adjusting the scc *algorithm for* subcircuit

Converting the *scc* algorithm to apply to *subcircuit* is a little more difficult than for the previous two algorithms. Much of the reasoning for this algorithm depends on the fact that every subtree must be visited. Our general approach is to perform the algorithm as per *circuit*, but before a propagation is performed or conflict reported, an extra checking step is required as follows.

For *subcircuit*, it is not necessarily true that there must be an edge from each subtree to its predecessor subtree. This is only true if both of these subtrees are required to be included in the circuit (at least one node from each). Similarly, we only require an edge from the first subtree to the root if there exist nodes both inside and outside that first subtree that must be in the circuit. Otherwise it doesn't matter if there is no way out of the first subtree. So in each of these situations, before pruning or fixing any edges we ensure that we can find an evidence node (whose successor variable's current domain does not include itself) within these subsets of the nodes.

The rule concerning edges skipping subtrees is similarly qualified. Such edges are only prohibited if the skipped subtrees contain a node which is required to be included in the circuit. Note that there is no need to ensure that the origin and destination subtrees of the edge in question are required to be included. If either of these subtrees is not included, then no edge between them can be used.

Edges from the root to subtrees before the last can be allowed if no node in the last subtree is required to be included in the circuit. It would be possible to extend this to say that an edge from the root to a subtree can be pruned if any of the later subtrees contains a node which must be included in the circuit, but we did not implement that extra reasoning.

At a node x within the search tree, if exploration of x's first child a does not reach any node above x, then the edge from x to a can be pruned only if nodes both inside and outside the part of the tree rooted at a are required to be included in the circuit.

If the search discovers a strongly connected component below the root, a conflict is reported only if there exist nodes both inside and outside that component which must be included in the circuit.

If the search does not reach all nodes, instead of reporting a conflict, we first check that at least one reached node is required to be included in the circuit, and if such a node can be found we fix all successor variables of nodes not reached by the search to form self-cycles.

The only other change required to the algorithm is that self-cycle edges must be handled carefully. These edges are ignored when finding the children of a node, and edges from the root to itself are not removed as *prune root* propagations.

5.3.2 Explaining scc *propagation*

In this section we provide the explanation clauses used for *scc* propagation. All of the explanations for this algorithm have size complexity $O(n^2)$, as they all include at least one statement that no (or only one) edge exists between two subsets of the nodes.

When discussing *subcircuit* we use the notation $in(a)$ to mean that node a must be included in the circuit, making it a possible evidence node (i.e. $a \notin D(x_a)$ for the current domain D).

There are several different propagation rules to consider.

1. A strongly connected sub-component exists.
   For *circuit*, on discovery of a strongly connected component made up of a strict subset of the nodes S, a conflict is reported with explanation
   $$\bigwedge_{i \in S, j \in V \setminus S} [\![x_i \neq j]\!] \rightarrow \mathit{false}.$$
   For *subcircuit*, if there exists a node $a \in S$ where $in(a)$ holds, then for each node $b \in V \setminus S$ we set $x_b = b$ with explanation
   $$\left([\![x_a \neq a]\!] \wedge \bigwedge_{\substack{i \in S \\ j \in V \setminus S}} [\![x_i \neq j]\!] \right) \rightarrow [\![x_b = b]\!].$$
2. Only one edge leads from the first subtree to the root.
   Let r be the root node, a be the unique node reaching the root from the first subtree, and A be the set of all nodes in the first subtree.
   Then for *circuit* the clause generated is
   $$\bigwedge_{\substack{i \in A, j \in V \setminus A \\ i \neq a \vee j \neq r}} [\![x_i \neq j]\!] \rightarrow [\![x_a = r]\!].$$
   For *subcircuit* it is
   $$\left([\![x_b \neq b]\!] \wedge [\![x_c \neq c]\!] \wedge \bigwedge_{\substack{i \in A, j \in V \setminus A \\ i \neq a \vee j \neq r}} [\![x_i \neq j]\!] \right) \rightarrow [\![x_a = r]\!],$$

```
propagateSCC(variables) {
  root ← selectRoot(variables);
  root.index ← 0;
  root.lowlink ← 0;
  nodesSeen ← 1;
  // original subtree is just the root node (index 0)
  startSubtree ← 0;
  endSubtree ← 0;
  foreach (v in root.successors) {
    if (v.index is undefined) {  // haven't explored this yet
      backedges ← explore(v, startSubtree, endSubtree);
      if (size(backedges) == 0) fail; // no edge to previous subtree
      if (size(backedges) == 1) requireEdge(backedges[0]);
      startSubtree ← endSubtree + 1;
      endSubtree ← nodesSeen - 1;
    }
  }
  if (nodesSeen ≠ size(variables)) fail;  // graph not connected
  // prune edges from root to all except last subtree
  if (startSubtree > 1) {  // if there was more than one subtree
    foreach (v in root.successors)
      if (v.index < startSubtree) pruneEdge((root,v));
  }
}

explore(v, startPrevSubtree, endPrevSubtree) {
  v.index ← nodesSeen;
  v.lowlink ← nodesSeen;
  nodesSeen ← nodesSeen + 1;
  foreach (w in v.successors) {
    if (w.index is undefined) { // haven't already visited w
      w_backedges ← explore(w, startPrevSubtree, endPrevSubtree);
      add all edges in w_backedges to backedges;
      v.lowlink ← min(v.lowlink, w.lowlink);
    } else { // w has been seen before
      if (w.index ≥ startPrevSubtree and w.index ≤ endPrevSubtree) {
        // w in previous subtree
        add edge (v,w) to backedges
      } else if (w.index < startPrevSubtree) // edge to w skips a subtree
        pruneEdge((v,w));
      v.lowlink ← min(v.lowlink, w.index);
    }
  }
  if (v.lowlink == v.index) fail; // scc rooted at v
  return backedges;
}
```

Fig. 6: Pseudo code for *scc* propagation algorithm.

where $b \in A, c \in V \setminus A$, $in(b)$ and $in(c)$.

3. Only one edge leads from subtree C to the previous subtree B.
   In this case the reason the edge is required depends on the structure of the tree. Let B be the set of nodes in B, and C be the set of nodes in C. Also let A be the set of nodes in subtrees before B, and D be the set of nodes which were included in subtrees after C or not reached in the search at all. Let c be the unique node in subtree C that reaches node b in subtree B.

For *circuit* the clause is

$$\left(\bigwedge_{\substack{i\in A \\ j\in B\cup C\cup D}} [\![x_i \neq j]\!] \wedge \bigwedge_{\substack{i\in B \\ j\in C\cup D}} [\![x_i \neq j]\!] \wedge \bigwedge_{\substack{i\in C, j\in B\cup D \\ i\neq c \vee j\neq b}} [\![x_i \neq j]\!] \right) \rightarrow [\![x_c = b]\!].$$

For *subcircuit*, where $p \in B$ and $q \in C$, $in(p)$ and $in(q)$, the clause is

$$\left([\![x_p \neq p]\!] \wedge [\![x_q \neq q]\!] \wedge \bigwedge_{\substack{i\in A \\ j\in B\cup C\cup D}} [\![x_i \neq j]\!] \wedge \bigwedge_{\substack{i\in B \\ j\in C\cup D}} [\![x_i \neq j]\!] \wedge \bigwedge_{\substack{i\in C, j\in B\cup D \\ i\neq c \vee j\neq b}} [\![x_i \neq j]\!] \right) \rightarrow [\![x_c = b]\!].$$

4. No edges lead from sub-tree C to the previous subtree B.
   This case is very similar to the above case.
   For *circuit* the clause is

$$\left(\bigwedge_{\substack{i\in A \\ j\in B\cup C\cup D}} [\![x_i \neq j]\!] \wedge \bigwedge_{\substack{i\in B \\ j\in C\cup D}} [\![x_i \neq j]\!] \wedge \bigwedge_{\substack{i\in C \\ j\in B\cup D}} [\![x_i \neq j]\!] \right) \rightarrow \mathit{false}.$$

   For *subcircuit*, where $p \in B$ and $q \in C$, $in(p)$ and $in(q)$, the clause is

$$\left([\![x_p \neq p]\!] \wedge [\![x_q \neq q]\!] \wedge \bigwedge_{\substack{i\in A \\ j\in B\cup C\cup D}} [\![x_i \neq j]\!] \wedge \bigwedge_{\substack{i\in B \\ j\in C\cup D}} [\![x_i \neq j]\!] \wedge \bigwedge_{\substack{i\in C \\ j\in B\cup D}} [\![x_i \neq j]\!] \right) \rightarrow \mathit{false}.$$

5. An edge skips one or more subtrees.
   In this case reasoning again depends on the structure of the tree. Let c be the origin of the edge and a its destination. Take A as the set of nodes in the same or an earlier subtree to that of a, B as the set of nodes in subtrees between that of a and c (of which there is at least one), and C as the set of nodes in the same or later subtree as that of c plus nodes not reached by the search.
   For *circuit* the clause generated is

$$\left(\bigwedge_{i\in A, j\in B\cup C} [\![x_i \neq j]\!] \wedge \bigwedge_{i\in B, j\in C} [\![x_i \neq j]\!] \right) \rightarrow [\![x_c \neq a]\!].$$

   For *subcircuit* it is

$$\left([\![x_b \neq b]\!] \wedge \bigwedge_{i\in A, j\in B\cup C} [\![x_i \neq j]\!] \wedge \bigwedge_{i\in B, j\in C} [\![x_i \neq j]\!] \right) \rightarrow [\![x_c \neq a]\!],$$

   where $b \in B$ and $in(b)$.
6. Edges leading from the root to a subtree other than the last.
   Let E be the set of nodes in subtrees before the last, L be the set of nodes in the last subtree or not reached by the search, and r be the root node. An edge from r to e where $e \in E$ is pruned with the following explanation.

For *circuit*,

$$\bigwedge_{i \in E, j \in L} [\![x_i \neq j]\!] \rightarrow [\![x_r \neq e]\!].$$

For *subcircuit*,

$$\left([\![x_l \neq l]\!] \wedge \bigwedge_{i \in E, j \in L} [\![x_i \neq j]\!] \right) \rightarrow [\![x_r \neq e]\!],$$

where $l \in L$ and $in(l)$.

7. Non-viable edge discovered within a subtree.
   This is the case where the first child of a node within the traversal tree does not reach any node above its parent. The edge from parent to child is pruned, because there is no way to reach outside the child's part of the tree without going through the parent node. The explanations are therefore as follows, where c is the child node, C is the set of nodes in the subtree rooted at c, p is the parent node, and $A = V \setminus (C \cup \{p\})$ is the set of all nodes excluding that of the parent and nodes in the subtree rooted at the child.
   For *circuit*,

$$\bigwedge_{i \in C, j \in A} [\![x_i \neq j]\!] \rightarrow [\![x_p \neq c]\!],$$

   and for *subcircuit*,

$$\left([\![x_a \neq a]\!] \wedge [\![x_b \neq b]\!] \wedge \bigwedge_{i \in C, j \in A} [\![x_i \neq j]\!] \right) \rightarrow [\![x_p \neq c]\!],$$

   where a and b are nodes such that $a \in A, b \in C$, $in(a)$ and $in(b)$.

5.3.3 Root node selection

In [19], it was shown that the choice of root node can have a significant impact on the performance of the *scc* algorithm. In particular, choosing a random root was very successful. We are interested in whether or not choosing the root randomly is still beneficial when using explanations. The effectiveness of lazy clause generation depends on the opportunity to make use of the learned clauses, so randomness may be detrimental if it results in a more varied exploration which does not often reach similar nodes.

We consider five options for the root selection strategy.

1. Always choose the first node.
2. Choose the first node with unfixed successor variable.
3. Choose a random node.
4. Choose a random node with unfixed successor variable.
5. Run the algorithm on every node.

If the root's successor is fixed, then there will be exactly one subtree below it and therefore no opportunities for propagation unless multiple strongly connected components are discovered, or the prune-within rule fires. For this reason it is probably better to choose an unfixed root. By including this option separately we can more accurately judge the benefit of making a random selection.

For *subcircuit*, the first, third and fifth strategies are modified slightly to avoid choosing a root which is fixed in a self-cycle, as this is guaranteed not to produce any

propagation. The other strategies already avoid this by choosing a node with unfixed successor. In the case where all successors are fixed these strategies also choose a non-self-cycle node as the root.

		Inorder Search			VSIDS Search		
Problem	Root	Fails	Props	Time	Fails	Props	Time
circuit	always first	196	1099	71.3 (48)	0.9	17.9	0.64
	first unfixed	118	678	42.3 (26)	0.8	16.9	0.60
	random	89	524	33.9 (22)	0.8	16.9	0.60
	random unfixed	77	453	30.4 (21)	0.7	15.7	0.55
	all roots	45	277	52.5 (33)	0.5	12.5	2.65
path	always first	350	2242	257.4 (180)	1.5	26.8	1.04
	first unfixed	313	2027	244.6 (174)	1.3	25.0	0.90
	random	211	1320	162.2 (103)	1.2	25.0	0.85
	random unfixed	205	1279	152.4 (95)	1.1	23.4	0.79
	all roots	125	869	211.1 (138)	0.7	18.3	3.98
subcircuit	always first	385	2636	171.6 (105)	3.2	57.7	1.50
	first unfixed	311	2144	149.0 (72)	3.3	60.8	1.52
	random	81	509	37.6 (15)	2.1	36.4	1.09
	random unfixed	78	487	35.6 (15)	2.2	37.6	1.17
	all roots	29	236	79.8 (39)	1.0	25.1	8.88
subpath	always first	623	4488	596.6 (494)	5.3	74.2	2.48
	first unfixed	628	4411	598.2 (492)	5.3	77.7	2.36
	random	271	1706	220.5 (64)	3.4	50.8	1.76
	random unfixed	257	1801	251.2 (95)	3.8	53.3	1.68
	all roots	120	1034	286.3 (140)	1.6	34.8	10.90

Table 5: Comparison of root selection strategies for the *scc* algorithm. Each figure is the average for 500 instances with 65 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

Table 5 shows the results of our root selection experiments. In most cases, choosing as root the first node with unfixed successor was better than always choosing the very first node. Surprisingly that didn't seem to be the case for *subcircuit* and *subpath* when using VSIDS search. This could be because a node with fixed successor is guaranteed to be in the circuit (except if it is a fixed self-cycle, but we never choose such a node as the root). Choosing a root which is fixed and therefore in the circuit may allow inconsistencies to be detected earlier, even though little other propagation will be possible. Failing early can be beneficial for VSIDS since it quickly learns how to escape the inconsistency. Note that we can't make a meaningful comparison between these two options for *subpath* using inorder search because in both cases almost all instances timed out.

Choosing a random root was clearly beneficial for all versions of the problem, and for both search strategies. This is in agreement with the results in [19], which is quite encouraging as it suggests that techniques involving randomness which are effective without explanation can still be beneficial when using explanation. Running the algorithm for every potential root did reduce the search space significantly, but due to the very high overhead this strategy was much slower than random root selection for all problems.

For *circuit* and *path*, choosing randomly from among the nodes with unfixed successors appeared to be the best strategy. For *subcircuit* and *subpath* the results are not

so clear. For *subcircuit* using VSIDS search and for *subpath* using inorder search, it was better to only eliminate nodes fixed in self-cycles, again probably due to the fact that in order for conflicts to be detected by the *subcircuit* propagator the exploration must reach at least one node which is required to be included in the circuit. However, for *subcircuit* using inorder search, excluding nodes with fixed successors still resulted in slightly better results, and for *subpath* using VSIDS search, although excluding these nodes resulted in higher numbers of failures and propagations, the execution time was slightly shorter.

In the remainder of our experiments we use random root selection. The *circuit* version of the propagator excludes nodes with fixed successor, while the *subcircuit* version only excludes self-cycle nodes (as this appeared to be the better choice overall for *subcircuit* and *subpath*).

5.3.4 Additional pruning rules

We would also like to discover the impact of the two extra pruning rules we have suggested (*prune root* and *prune within*). In the previous experiments we used the original version of the *scc* algorithm which excludes these pruning rules. Table 6 shows the impact of adding one or both of the additional rules for each version of our problem.

It is clear that for all versions of our problem both *prune within* and *prune root* are beneficial. For *circuit* and *path*, *prune root* was more effective, reducing the average execution time by around 30-40%, while *prune within* gave a more modest improvement of between 7% and 30%. For *subcircuit* and *subpath* however, *prune within* performed better than *prune root*, most obviously in the case of *subpath* using inorder search where adding this rule reduced the execution time by almost 85%.

Combining the two rules was the best (or tied best) option in all cases, whether considering failures, propagations, or execution time. Therefore in the remainder of this paper we include both *prune within* and *prune root* whenever *scc* is used.

The decision of whether or not to use *prune root* will clearly affect the choice of root selection strategy. We conducted further experiments to verify that using both additional propagation rules with random root selection is indeed the best combination for our problems.

5.3.5 Choosing evidence literals

As with *prevent* explanations, the *subcircuit* versions of clauses used to explain *scc* propagation require the inclusion of evidence literals (literals $[\![x_k \neq k]\!]$ for a node k whose successor variable's current domain does not include k). In many cases there are several appropriate literals which could be chosen. We experimented with the same options for selecting evidence literals as discussed for propagation of *prevent*.

As can be seen in Table 7, in most cases choosing the literal which became fixed highest in the search tree made very little difference compared with simply choosing the first applicable literal. However, when using VSIDS search for *subpath* it did appear to be beneficial. Since the *highest* heuristic was never far from the best option, and was in most cases significantly better than the opposite strategy of choosing the literal fixed lowest in the search tree, we decided to use the *highest* heuristic for *scc* evidence literal selection in further experiments (making this the same as *prevent* evidence literal selection).

		Inorder Search			VSIDS Search		
Problem	Pruning Rules	Fails	Props	Time	Fails	Props	Time
circuit	original scc	77	453	30.4 (21)	0.7	15.7	0.55
	scc+within	128	696	28.2 (18)	0.6	14.8	0.41
	scc+root	75	447	18.2 (10)	0.6	14.8	0.37
	scc+within+root	77	412	16.8 (9)	0.5	13.9	0.35
path	original scc	205	1279	152.4 (95)	1.1	23.4	0.79
	scc+within	251	1606	128.1 (79)	0.9	21.9	0.56
	scc+root	192	1206	92.0 (51)	1.0	22.3	0.56
	scc+within+root	148	968	76.1 (40)	0.9	20.9	0.52
subcircuit	original scc	81	509	37.6 (15)	2.1	36.4	1.09
	scc+within	47	280	12.7 (5)	1.6	33.2	0.80
	scc+root	46	299	13.5 (6)	2.0	35.5	0.82
	scc+within+root	26	186	7.4 (3)	1.6	32.4	0.75
subpath	original scc	271	1706	220.5 (64)	3.4	50.8	1.76
	scc+within	71	484	33.7 (7)	2.5	44.7	1.23
	scc+root	274	1745	157.0 (35)	3.3	49.4	1.35
	scc+within+root	62	425	30.1 (5)	2.4	43.9	1.17

Table 6: Additional propagation rules for the *scc* algorithm. Each figure is the average for 500 instances with 65 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

		Inorder Search			VSIDS Search		
Problem	Heuristic	Fails	Props	Time	Fails	Props	Time
subcircuit	first	26	186	7.4 (3)	1.6	32.4	0.75
	last	26	183	7.4 (2)	1.6	32.2	0.73
	lowest	36	240	9.8 (3)	1.8	30.2	0.73
	highest	23	160	7.5 (3)	1.6	32.4	0.73
subpath	first	62	425	30.1 (5)	2.4	43.9	1.17
	last	81	554	44.5 (8)	2.0	39.8	1.06
	lowest	113	758	65.5 (13)	2.7	42.2	1.22
	highest	61	418	29.1 (5)	2.0	39.8	1.05

Table 7: Comparison of heuristics for evidence literal selection for the *scc* algorithm. Each figure is the average for 500 instances with 65 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

6 The effect of explanation

In this section we explore the effect of explaining circuit. We first investigate which propagation algorithm performs the best with and without explanation, and then go on to compare explaining and non-explaining propagators.

6.1 Propagation complexity trade-off

In all propagators there is a trade off between the complexity of the algorithm and its power. In lazy clause generation propagators we also need to consider the size and generality of the explanations produced. When using explanation, a weakly propagating algorithm which produces short highly reusable explanations, may be able to compete

with a stronger propagator whose explanations are much more complex and large and not very reusable.

We therefore wish to investigate whether adding explanation changes the relative effectiveness of the different circuit propagation algorithms. We consider four different versions of the propagator, as follows.

1. The *check* algorithm only.
2. Both *check* and *prevent* (recall *prevent* cannot be used alone).
3. The *scc* algorithm alone.
4. All three algorithms in combination.

When multiple algorithms are used, we apply the least expensive first, and then continue with each more expensive algorithm if no conflict has been discovered. For *scc* we used random root selection and both extra propagation rules. Experimentation showed that just as this is the best combination when using explanation, it is also the best combination when not using explanation.

Table 8 shows the results without explanation. For all versions of the problem the *prevent* algorithm is an improvement over *check* alone. The *scc* algorithm used alone performs better than *check* for *circuit* and *path*, but worse than *check* for *subcircuit* and *subpath*, and always worse than *check* plus *prevent*. It appears that the *scc* algorithm is too expensive to pay off, as using all algorithms together is better than *scc* alone, but still slower than *check* and *prevent* without *scc*.

For *circuit* and *path* we include execution times for Gecode (version 4.0.0) using the default circuit implementation which includes all three algorithms (although obviously without our modifications to *scc*). It is clear that without explanation the Gecode implementation is much faster than Chuffed.

		Problem Size		
Problem	Algorithm	15	20	25
circuit	*check*	14.9	355.6 (219)	567.7 (459)
	check +prevent	7.2	244.0 (129)	536.0 (415)
	scc	11.4	289.4 (170)	556.2 (442)
	check +prevent +scc	9.2	277.2 (162)	542.3 (423)
	Gecode	0.1	4.4	33.5 (20)
path	*check*	25.7 (5)	259.4 (143)	548.0 (426)
	check +prevent	15.1 (1)	236.5 (127)	521.0 (398)
	scc	22.4 (4)	258.9 (144)	546.2 (421)
	check +prevent +scc	20.2 (3)	265.5 (148)	536.7 (414)
	Gecode	0.3	25.8 (11)	136.0 (76)
subcircuit	*check*	15.1 (1)	409.9 (269)	578.9 (470)
	check +prevent	10.0	318.0 (189)	555.7 (439)
	scc	17.0 (1)	396.9 (254)	575.8 (463)
	check +prevent +scc	12.5	359.2 (224)	564.8 (452)
subpath	*check*	24.1 (2)	375.7 (237)	576.0 (462)
	check +prevent	17.6 (3)	327.2 (194)	556.3 (440)
	scc	29.9 (5)	394.9 (253)	576.3 (464)
	check +prevent +scc	28.2 (4)	363.8 (224)	570.9 (456)

Table 8: Comparison of propagation algorithms without explanation, using inorder search. Each figure shown is the average execution time (secs) over 500 instances of the given size. Where at least one instance reached the 10 minute time limit, the number of timeouts is given in brackets.

		Inorder Search			VSIDS Search		
Problem	Algorithm	55	60	65	55	60	65
circuit	*check*	99 (66)	132 (96)	166 (122)	0.42	0.63	2.37 (1)
	check +prev	73 (51)	114 (86)	128 (96)	0.26	0.31	0.35
	scc	5 (2)	10 (4)	17 (9)	0.24	0.27	0.35
	all	11 (8)	19 (13)	22 (14)	0.25	0.30	0.31
path	*check*	265 (184)	305 (227)	353 (271)	0.69	1.00	2.69 (1)
	check +prev	214 (146)	274 (201)	314 (233)	0.37	0.48	0.56
	scc	48 (22)	48 (20)	76 (40)	0.47	0.41	0.52
	all	44 (22)	72 (39)	97 (56)	0.33	0.40	0.46
subcircuit	*check*	216 (127)	195 (100)	270 (154)	1.04	1.74	2.19
	check +prev	169 (86)	155 (72)	242 (133)	0.54	0.74	0.94
	scc	3 (1)	2	7 (3)	0.45	0.58	0.73
	all	7 (3)	2	9 (2)	0.41	0.50	0.61
subpath	*check*	400 (262)	410 (250)	511 (371)	1.17	1.71	2.64
	check +prev	382 (245)	393 (245)	494 (343)	0.63	0.85	1.17
	scc	19 (3)	19 (4)	29 (5)	0.64	0.82	1.05
	all	19 (3)	17 (1)	26 (6)	0.58	0.78	1.04

Table 9: Comparison of propagation algorithms when using explanation. Each figure is the average execution time (secs) over 500 instances of the given size. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

We now compare this with the results using explanation shown in Table 9. The addition of *prevent* to *check* was still clearly beneficial. The striking difference is that with explanation the *scc* algorithm actually performs very well. This is interesting as it suggests that the explanations produced by *scc*, although large, are sufficiently general to be effective. In most cases when using inorder search *scc* alone was the best performing algorithm, but for the hardest problems (large sizes of *subpath*) it was better to use all three algorithms together. For VSIDS search, using all three algorithms together seemed to be the best option.

6.2 Benefit of Explanation

Having investigated the best choice of algorithm when using and not using explanation, we now consider the impact of explanation on the performance of our circuit propagator. In order to make a direct comparison, we need to use the same circuit algorithm with and without explanation. When not using explanation, the best choice of algorithm was *check* plus *prevent* without *scc*, but when using explanation the *scc* algorithm was vital for good performance. We have chosen to use all three algorithms together, since this gives reasonable performance both with and without explanation, and is the best option when using VSIDS search. We also include results for Gecode, which was found earlier to be much faster than Chuffed without explanation regardless of the choice of algorithm.

Table 10 shows the average execution time and number of failures for increasing problem sizes, giving a comparison between Chuffed without explanation (inorder search), Chuffed with explanation using inorder search, Chuffed with explanation and VSIDS search, and Gecode (inorder search). Comparing Chuffed with explanation to Chuffed without explanation, the smallest improvement was for *circuit* problems. For the smallest size (15) these were solved almost 80 times faster when using explanation.

		Execution Time (secs)			Fails (000s)		
Problem	Solver	15	30	60	15	30	60
circuit	No expl	9.16	594.4 (492)	600.0 (500)	704	21462	13517
	Expl	0.12	0.6	19.1 (13)	0.04	1.8	80.5
	VSIDS	0.04	0.1	0.3	0.03	0.1	0.1
	Gecode	0.09	111.1 (75)	465.4 (370)	3.97	5362	17821
path	No expl	20.19 (3)	597.3 (495)	600.0 (500)	1746	23874	12481
	Expl	0.05	2.6 (1)	72.1 (39)	0.12	4.7	130.2
	VSIDS	0.04	0.1	0.4	0.06	0.1	0.3
	Gecode	0.33	344.3 (249)	586.8 (488)	18.19	15365	23197
subcircuit	No expl	12.47	598.6 (498)	600.0 (500)	837	20808	12800
	Expl	0.04	0.5	2.4	0.06	3.3	6.9
	VSIDS	0.04	0.1	0.5	0.04	0.2	0.5
subpath	No expl	28.19 (4)	598.2 (497)	600.0 (500)	1640	20850	12392
	Expl	0.04	0.6	16.6 (1)	0.16	2.8	35.0
	VSIDS	0.04	0.2	0.8	0.06	0.3	0.8

Table 10: Experiment showing the effect of explanation, comparing Chuffed without explanation, Chuffed with explanation, Chuffed with explanation using VSIDS search, and Gecode. Each figure is the average for 500 instances of the given size. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

For size 30 these problems were solved around 1000 times faster. Looking at size 15, the improvement was greatest for *subpath* problems, which are also the hardest. For *subpath* problems of this size, using explanation was already more than 700 times faster than not using explanation. For the larger sizes too many instances timed out when not using explanation to accurately estimate the improvement factor.

Using VSIDS rather than inorder search was a further improvement over just adding explanation, and this improvement also increased with increasing problem size. For example, solving *circuit* problems using VSIDS search was 3, 6 and 60 times faster than inorder search (with explanation) for sizes 15, 30 and 60 respectively.

Although Gecode was much faster than Chuffed without explanation, it was able to compete with Chuffed with explanation only for the smallest size of *circuit* problems. By size 60, Gecode timed out at 10 minutes for 75% of *circuit* problems and almost all *path* problems, while Chuffed with explanation took on average 19 and 72 seconds, and Chuffed with explanation and VSIDS averaged less than half a second for both problems.

The failure counts also given in Table 10 show a similar pattern to the times.

These results make it very clear that explanation is highly effective for circuit (and subcircuit) propagation. It is also apparent that larger and harder problems benefit more. That is, those instances with greater average search time without explanation also have a larger improvement factor. This is probably because when more search is required, there are more opportunities to make use of learned clauses.

7 Related work

The purpose of this paper is to investigate how explanation can be used, and its effect on the *circuit* constraint and its variants. We are unaware of any other work on *circuit*

with explanation. However, for context we briefly describe some related constraints and propagation algorithms.

In this work we considered three propagation algorithms for *circuit*. Another (incomplete) *circuit* propagation algorithm was suggested in [11]. This algorithm uses graph separators to detect nonhamiltonian edges which can then be removed. While of theoretical interest, the algorithm is very complex and appears very slow to propagate, which is why we have not included it in our study.

The *circuit* constraint can clearly be implemented using the more general *cycle* constraint[1], by fixing the number of cycles to one. This is also possible for *subcircuit*, although an extra constraint is required to ensure that the total number of cycles is one more than the number of self-cycles, and this means it will not propagate as strongly as the propagator we describe. Since *path* and *subpath* are implemented using *circuit* and *subcircuit* respectively, these can be implemented using *cycle* as well.

We have already mentioned the more general version of *path* which enforces n disjoint paths where n is a variable. A further generalisation of this is the *tree* constraint, introduced in [2] and extended in [3]. The extended version of *tree* includes precedence, incomparability and degree constraints, and can be used to implement *subpath* as well as *path* constraints.

Another graph-based constraint of particular relevance is the DomReachability constraint, which uses reasoning based on node dominance and reachability and can be used to solve the ordered simple path with mandatory nodes problem [16]. This problem is equivalent to that solved by *subpath* with the additional requirement of enforcing an order between certain pairs of nodes (which is not possible using our *circuit*-based implementation).

While this paper focuses on propagation algorithms designed specifically for *circuit*, using a simple transformation to allow these to be applied to *path* problems, it is also possible to make the opposite transformation and implement *circuit* using a *path* constraint (or a *tree* constraint with the ability to restrict the degree of nodes). This is achieved by selecting an arbitrary node to be the start and end of the path, and splitting this node into two - the start node keeps all outgoing edges and the end node keeps the incoming edges.

The same technique can be used for *subcircuit* as long as there is at least one required node. A general propagator for *subcircuit* using the *path* formulation is possible, but it would need to wait until at least one node became mandatory before it could begin propagating, using this node as the start and end of the path.

Actually, the *scc* algorithm is closely related to a propagation algorithm for *path* called Reduced Path [8]. This algorithm finds strongly connected components of the graph and enforces that they form a chain with exactly one edge between neighbouring components and no edges skipping components. If the root node from the *scc* algorithm were the one to be split during the conversion to a *path* formulation, then each subtree would become one or more strongly connected components in this chain. The propagations performed by *scc* are therefore a subset of those performed by Reduced Path.

The *prune within* improvement to *scc* is also covered by existing *path* propagation algorithms. This rule is actually a restricted form of dominator based pruning, similar to that discussed in [7] but only detected when it is convenient to do so as part of the *scc* algorithm.

Although *path* propagators can potentially remove more edges than the *scc* algorithm, there is a disadvantage to using a *path* formulation for *circuit*, which is that the node to split is chosen up front. This is equivalent to always choosing the same root

node for the *scc* algorithm, and our experimental results show that it is beneficial to choose the root randomly. An interesting avenue for future work would be a propagator for *circuit* based on *path* propagation algorithms, but which does not commit to a node to split up front, instead selecting this node each time propagation is performed, perhaps randomly as we have done for *scc*.

8 Conclusion

We have investigated how best to add explanation to the global constraint *circuit* and its variants. Our results show that explanation is highly beneficial for problems involving these constraints. The resulting propagators compare very favourably against the state-of-the-art *circuit* implementation in Gecode, one of the fastest available constraint programming solvers.

Somewhat surprisingly the complex *scc* propagator which creates very large explanations, is not dominated by the cheaper propagators, whose explanations are typically very much smaller. Indeed it is certainly not always the case that once learning is used that simpler propagators are preferable. But just as without learning, sometimes weaker propagators are preferable to strong propagators. A full CP system must support both kinds.

Perhaps more surprising is the fact that adding randomness to an explaining propagator (the choice of root for *scc*) is beneficial. Usually explaining propagators want to be deterministic so that they tend to reuse earlier explanations again and again. It appears for *circuit* that the benefit of random root selection is substantial enough to overcome this disadvantage

Acknowledgments

We are thankful to the reviewers for their helpful comments and suggestions. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, Dec. 1994.
2. N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524, pages 64–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
3. N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, and incomparability constraints. *Constraints*, 13(4):459–489, Feb. 2008.
4. Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In *Proceedings of the 14th International Conference on Logic Programming (ICLP97)*, pages 316–330, 1997.
5. *cycle* constraint. Global constraint catalog: http://www.emn.fr/z-info/sdemasse/gccat/Ccycle.html.
6. N. Downing, T. Feydy, and P. Stuckey. Explaining alldifferent. In M. Reynolds and B. Thomas, editors, *Proceedings of the Australasian Computer Science Conference (ACSC 2012)*, volume 122 of *CRPIT*, pages 115–124, Melbourne, Australia, 2012. ACS.

7. J. Fages and X. Lorca. Revisiting the tree constraint. In *Principles and Practice of Constraint Programming – CP 2011*, volume 6876, pages 271–285. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
8. J. Fages and X. Lorca. Improving the asymmetric TSP by considering graph structure. *arXiv preprint arXiv:1206.3437*, 2012.
9. G. Katsirelos. *Nogood processing in CSPs*. PhD thesis, University of Toronto, 2008.
10. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 390–396, 2005.
11. L. Kaya and J. Hooker. A filter for the circuit constraint. In F. Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, volume 4204, pages 706–710. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
12. K. Marriott, N. Nethercote, R. Rafeh, P. Stuckey, M. Garcia de la Banda, and M. Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
13. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of 38th Conference on Design Automation (DAC'01)*, pages 530–535, 2001.
14. N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer-Verlag, 2007.
15. O. Ohrimenko, P. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
16. L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 of *Lecture Notes in Computer Science*, pages 73–87, 2006.
17. C. Schulte, M. Lagerkvist, and G. Tack. GECODE - an open, free, efficient constraint solving toolkit. http://www.gecode.org/.
18. C. Schulte and P. Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems*, 31(1):Article No. 2, 2008.
19. C. Schulte and G. Tack. Weakly monotonic propagators. *Principles and Practice of Constraint Programming-CP 2009*, page 723–730, 2009.
20. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.
21. W. van Hoeve. The alldifferent constraint: A survey. *http://arxiv.org/abs/cs/0105015*, May 2001.