A Global Constraint for Cutset Problems

François Fages and Akash Lal*
Projet Contraintes, INRIA Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France,
francois.fages@inria.fr
akashlal@cse.iitd.ernet.in

Abstract
We consider the problem of finding a cutset in a directed graph \( G = (V,E) \), i.e. a set of vertices that cuts all cycles in \( G \). Finding a cutset of minimum cardinality is NP-hard. There exist several approximate algorithms and exact algorithms, most of them using graph reduction techniques. In this paper we propose a global constraint for cutset problems. The cutset constraint is a boolean constraint over variables associated to the vertices of a given graph, that states that the subgraph restricted to the vertices having their boolean variable set to true is acyclic. We propose a filtering algorithm based on graph contraction operations and inference of simple boolean constraints, that has an \( O(|E| + |V| \log |V|) \) time complexity. We discuss search heuristics based on graph properties provided by the cutset constraint, and show the efficiency of the cutset constraint on benchmarks of the literature for pure minimum cutset problems, and on an application to log-based reconciliation problems where the global cutset constraint is mixed with other boolean constraints.

1 Introduction
Let \( G = (V,E) \) be a directed graph with vertex set \( V \) and edge set \( E \). A cycle cutset, or cutset for short, of \( G \) is a subset of vertices \( V' \subseteq V \), such that the subgraph of \( G \) restricted to the vertices belonging to \( V \setminus V' \) is acyclic. Deciding whether an a bit a y graph admits a cutset of a given cardinality is an NP-complete problem [6]. The minimum cutset problem, i.e. finding a cutset of minimum cardinality (also called a feedback vertex set [4]), is thus an NP-hard problem. This problem has found applications in various contexts such as deadlock avoidance [2], diagnosis and reconciliation [11] or Bayesian inference [15].

The e a e a few classes of graphs for which the minimum cutset problem has a polynomial time complexity. These classes are defined by certain edge connectivity properties.
the g a ph. Shami [11] p oposed a linea time algo ithm fo educible flow g a phs. Rosen [2] modifed this algo ithm in an app oximate algo ithm fo gene al g a phs. Wang, Lloyd and Soffa [14] found an $O(|E| \cdot |V|^2)$ algo ithm fo an unlated class of cyclically educible g a phs. Smith and Walfo d [13] p oposed an exponential time algo ithm fo gene al g a phs that behaves in $O(|E| \cdot |V|^2)$ in cee time classes of g a phs. The comparison of these diff ent educibility p ope ties was done by Levy and Low [8] and Lou Soffa and Wang [9] who p oposed an $O(|E| \cdot \log|V|)$ app oximate algo ithm based on a simple set of five g a ph cont action ules. Pa dalos, Qian and Resende [10] used these cont action ules inside a G ee dy Randomized Adaptive Sea ch P ocetu e (GRASP). The GRASP p ocetu e is cu ently the most efficient app oximate algo ithm fo solving la ge instances, yet without any gua antee on the quality of the solution found. Exact solving has been t ied with polyhed al methods by Funke and Reinelt who p esented computational esults with a b anch-and-cut algo ithm implemented in CPLEX [5].

Our aim he e is to develop a const aint p og amming app each to cutset p obles and design a global const aint fo cutset ules. Specialized p ogagation algo ithms fo global const aints a e a key featu e of the efficiency of const aint p og amming (see [1] fo example). The idea of this pape is to emed ele vant g a ph eduction techniques into a global cutset const aint, that can be combined with the boolean const aints, and that can be used within either a b anch-and-bound p ocetu e o local sea ch methods.

Ou inte est fo cutset p obles, a ose f om the study of log-based ecollision p obles in nomadic applications [7]. The minimum cutset p oblem shows up as the cent al p oblem esponsible fo the NP-ha dness of optimal ecollision [3]. In this context however, the cutset const aint comes with the boolean const ains which ag egate ve ties into clusters fo mo e gene ally, express dependency const ains between ve ties. In ou p evious const aint-based app each [3], the acyclicity const aint was expressed as a scheduling p oblem mixing boolean and finite domain const aints. We show in this pape that the global cutset const aint p ovides a mo e efficient p uning of the acyclicity condition. Mo eove it allows fo an all boolean modeling of log-based ecollision p obles.

The est of the pape is o ganized as follows. In the next section, we define the global cutset const aint, and p opose a syntax fo its implementation in const aint logic p og amming (CLP). In section 3, we p opose a filtr ing algo ithm based on g a ph eductions and simple boolean const aint inference. We show its co ectness, discuss some implementation issues and p ove its $O(|E| + |V|\cdot \log|V|)$ time complexity. In section 4, we discuss some sea ch heuistics based on the p ope ties of the inte nal g a ph managed by the global cutset const aint. Section 5 describes the log-based ecollision p oblem in nomadic applications and its modeling with the cutset const aint. Section 6 p esents computational esults on Funke and Reinelt’s benchma ks fo pu minimum cutset p obles, and on benchma ks of log-based ecollision p obles. The last section p esents ou conclusion.

2 The Cutset Constraint

Given a g a ph $G = (V, E)$, we conside the set $B$ of boolean va iables, obtained by associating a boolean va iable to each ve tex in $V$. A ve tex is said to be accepted if its boolean va iable is t ue, and is said to be rejected if its boolean va iable is false. We
consider the boolean constant on \( B \)

\[
\text{cutset}(B,G)
\]

which states that the subset of edges \( B \) forms a valid cutset of \( G \).

More specifically, we shall consider the implementation of the following constant logic program (CLP) predicate:

\[
\text{cutset}(\text{Variables, Vertices, Edges})
\]

\[
\text{cutset}(\text{Variables, Vertices, Edges, Size})
\]

where \( \text{Variables} = [V_1, \ldots, V_n] \) is the list of boolean variables associated to the vertices, \( \text{Vertices} = [a_1, \ldots, a_n] \) is the list of vertices, \( \text{Edges} = [a_i-a_j, \ldots] \) is the list of directed edges incident to pairs of vertices, and \( \text{Size} \) is a finite domain variable corresponding to the size of the cutset, i.e., the number of directed edges. The boolean variable \( V_i \) equals 0 if the vertex \( a_i \) is in the cutset (i.e., ejected from the graph) and equals 1 if the vertex \( a_i \) is not in the cutset (i.e., accepted to be in the graph).

For the purpose of the minimum cutset problem, that is, ejecting a minimum number of vertices, the branch-and-bound minimization predicate of CLP can be used. So, essentially one expresses a minimum cutset problem with the CLP query:

\[
\text{cutset}(\text{B, V, E, S}), \text{minimize} \text{(labeling}(B)) \text{, S)}.
\]

As usual, the cutset constant does not make any assumption on the other constants imposed on its variables and hence the use is allowed to qualify the cutset solution he wants with an aspect constant. For this reason, the cutset constant has to be general enough to allow the possibility of finding any cutset of the graph.

3 Filtering Algorithm

The filtering algorithm we propose uses constant action operations to reduce the graph size, check the acyclicity of the graph and bound the size of its cutsets. The graph constant action rules use a constant if on the rules of Levy and Low [8], and Lloyd, Soffa and Wang [9] for computing one minimum cutset. The graph constant action operations described in this section compute an aspect by applying a cutset in a constant aspect propagation setting.

The cutset constant maintains an internal state composed of an explicit presentation of the graph, that is, related to the constant states of the constant state on the boolean variables, \( V_1, \ldots, V_n \), associated to the vertices of the graph. The filtering algorithm tries to converge the information in the graph (about the cycles that have to be cut) to constant states on the boolean variables \( V_i \). On completing such convergence, any valid solution of the constant state is checked to provide a valid cutset of the original graph. The essential components of the filtering algorithm are the graph constant action operations. They either simplify the graph without losing any information or converge some information into explicit constant states and simplify the graph in the process.

Below we present two basic Accept and Reject operations and the graph constant action operations performed by the filtering algorithm.

3.1 Internal Accept and Reject Operations

We consider the following operations on a directed graph:

\[
\text{cutset}(B,G)
\]
1. **Accept**(*v*): under the condition that *v* has no self loop, i.e., (*v*, *v*) is not an edge, this operation removes the vertex *v* along with the edges incident on it and adds the edges (*v*, *v*<sub>1</sub>) if (*v*, *v*) and (*v*, *v*<sub>2</sub>) are edges in the original graph.

![Diagram](attachment:image.png)

2. **Reject**(*v*): This operation removes the vertex *v* along with the edges incident on it.

Note that these operations on the internal graph of the cutset constraint do not Pclude the instantiation of the boolean variables associated to the vertices of the graph. If a boolean variable is instantiated, the filtering algorithm returns the corresponding Accept or Reject operation. On the other hand, we shall see in the next section that the filtering algorithm of the cutset constraint can produce Accept and Reject operations on its internal graph structure without instantiating the boolean variables associated to the original graph.

We shall use the following:

**Proposition 1** Let *G* = (*V*, *E*) be a directed graph with vertex set *V* and edge set *E* and let *v* ∈ *V* be a vertex of the graph such that (*v*, *v*) ∉ *E*. Also let *G'* = (*V'*, *E'*') be the graph obtained by performing Accept(*v*) on *G*. Then any cutset of *G* which does not have *v* is also a cutset of the graph *G'* and vice versa.

**Proof**: (⇒) Let *S* ⊆ *V* be a cutset of *G* and *v* ∉ *S*. Let *G*\*S denote the graph obtained by removing the vertices of *S* from *G*. Since *S* is a cutset, *G*\*S should be acyclic. Now, suppose that *S* is not a cutset of *G'*.

![Diagram](attachment:image.png)

Due to the operation Accept(*v*), the edge (*v*, *v')* exists in *G'* with each vertex in *V'*\*S. If this edge has no edges which came due to the operation Accept(*v*), then this is also a cycle in *G*\*S. Hence this cycle has edges induced by the accept operation. By replacing each such edge (*v*, *v')* by (*v*, *v*) and (*v', *v'†*), we again get a cycle in *G*\*S. Hence, by contradiction, we have one side of the result.

(⇐) Let *S* ⊆ *V* be a cutset of *G'*.

Again, suppose that *S* is not a cutset of *G*. The edge *e*, the edge exists a cycle *v*, ..., *v*, *v* in *G'*\*S. If none of these vertices is *v* then this is also a cycle in *G*\*S. Hence, at least one of these vertices is *v*. If *v* = *v* then replace the edges (*v*, *v*<sub>i</sub>) by (*v*, *v*<sub>i+1</sub>) to get a cycle in *G*\*S. Again we get a contradiction.

The accept operation can thus be used to check if a given set is a cutset of not:

**Corollary 2** A given directed graph *G* = (*V*, *E*) is acyclic provided we can accept all vertices in it, i.e., while accepting the vertices one by one, no vertex gets a self loop.

**Proof**: Suppose that while accepting the vertices in *G*, no vertex gets a self loop. Then after accepting all the vertices, the graph that remains has no vertex of edges.
Hence this has a cutset \( \emptyset \). Now, by repeated application of p oposition \( 1, \emptyset \) is also a cutset of \( G \). Hence \( G \) is acyclic. The same can also be proved similarly by using p oposition \( 1 \). So if \( G \) is a acyclic g aph, then it has the cutset \( \emptyset \). Now, while accepting the ve ties of \( G \), if we get a ve tex with a self loop, then that g aph cannot have \( \emptyset \) as the cutset. However, \( \emptyset \) should have been a cutset by p oposition \( 1 \). Hence by cont adion, we have ou esult.

Simila ly, we have:

**Proposition 3** Let \( G = (V, E) \) be a directed graph and \( v \in V \) be a vertex of the graph. Also let \( G' = (V', E') \) be the graph obtained by performing \( \text{Reject}(v) \) in \( G \). If \( S \) is a cutset of \( G \) which contains \( v \) then \( S - \{v\} \) is a cutset of \( G' \) and vice versa.

**Corollary 4** The set of all cutsets of a graph remains invariant under the operation \( \text{Reject}(v) \) if \( v \) has a self loop.

These p opositions show that the accept and reject ope ations have the nice p ope ty of maintaining any cutset by picking a ight ve tex to apply the ope ation on. If the e is a minimum cutset that contains the ve tex \( v \) then after the ope ation \( \text{Reject}(v) \), we can still find that cutset but have a smalle g aph wo k with. Simila ly, if the e is a minimum cutset that does not contain \( v \) then after \( \text{Accept}(v) \), we can still find that cutset but again in a smalle g aph.

### 3.2 Graph Contraction Operations

We shall use the following five g aph cont action ope ations:

1. **IN0** (In deg ee = 0) In case the in deg ee of a ve tex is ze o, that ve tex cannot be a pa t of any cycle. Hence its acceptance o ection will cause no change to the est of the g aph. So, its edges a e emoved but no const aints a e p oduced since a cutset can exist including o excluding this ve tex.

2. **OUT0** (Out deg ee = 0) In case the out deg ee of a ve tex is ze o, the situation is simila to the one above. Again, the edges incident on this ve tex a e emoved and no const aints a e p oduced.

3. **IN1** (In deg ee = 1) In case a ve tex has in deg ee one, then the situation is as follows,

\[
\begin{array}{ccc}
\text{a} & \text{j} & \text{c} \\
\text{i} & \text{e} & \\
\text{b} & \text{d} & \text{f}
\end{array}
\]

\[
\Rightarrow
\begin{array}{ccc}
\text{a} & \text{h} & \text{c} \\
\text{d} & \text{f} & \text{e}
\end{array}
\]

If a cycle passes th ough \( i \) then it must also pass th ough \( j \). Hence by me ging these two nodes to fo m the node \( h \), we do not eliminate any cycle in the g aph. Along with this ection, we impose the const aint \( V_h = V_i \cup V_j \) on the va iables associated with the ve ties. This captu es the fact that if \( h \) is not a pa t of any
cycle, then both \(i\) and \(j\) are not part of any cycle and vice versa. The rest of this paper will use the names \(i\) and \(j\) in the context that vertex \(i\) has in (or out) degree 1 and vertex \(j\) is the predecessor (or successor) of \(i\). Note that, as a compromise tading precision for efficiency, we do not perform this operation if it leads to merging two nodes that have themselves come due to the merging of other nodes. This restriction is justified in the next section.

4. **OUT1** (Out degree = 1) This case is similar to the above case.

5. **LOOP** (Self loop on a vertex) In case a vertex has a self loop then this vertex is ejected and its boolean variable is set to 0 since no cutset can exist without including this vertex. However, if the vertex is a merged node \(h\) then we impose \(V_h = 0\) but cannot eject \(h\) since that implies ejection of both \(i\) and \(j\). So we look at the self loop edge of \(h\) and figure out if it came from the loop \((i, j)\) of \((i, j), (j, i)\). This is done by maintaining history on merged nodes, as described in the next section. Note that the edge cannot be a loop \((i, j)\) since \(i\) had in (or out) degree 1 as one and this edge was not a loop. If the loop came from edge \(i\) to \(j\), then we impose \(V_j = 0\) and remove \(h\) from the graph. Otherwise, we just convert \(h\) to \(j\) i.e., remove edges corresponding to \(i\). This conversion is done because we know that the loop came due to a cycle involving edges between \(i\) and \(j\). Hence at least one of \(i\) and \(j\) should be ejected. Choosing to eject either ends the edges of \(i\) useless.

**Proposition 5** The complexity of the reduction algorithm (repeated application of contraction operations till no more can be applied) is \(O(|E| + |V| \log |V|)\).

**Proof:** The proof is very easy and comes from the fact that we look at an edge only \(O(1)\) times and don’t add new edges. Let \(d_v\) denote the in + out degree of vertex \(v\). Start the process by setting \(\forall |V| \log |V|\), the vertex degrees based on in and out degrees are stored in a way indexed by the vertex degree. Each time an operation is performed, we will update these arrays. First consider the IN0 operation. Using the arrays just created, we can find in \(O(1)\) time, a vertex to apply the reduction on. Reduction on vertex \(v\) will take \(O(d_v)\) time and will lead to deletion of all edges on it. Along with this deletion, update the degree of affected vertices while maintaining the above arrays. Since new edges are not added to the graph, any reduction of IN0 operations is performed by leaving any number of different reductions can take at most \(O(\sum d_v) = O(|E|)\) time. Similarly, any number of OUT0 reductions can take time \(O(|E|)\). For the LOOP case, we can see that it too leads to the reduction of edges of some vertex and hence satisfies the same bound (history lookup is \(O(1)\)). The case for IN1 and OUT1 is easy to prove since we are not allowing merged nodes to get merged. As a result, we look at a vertex at most once and do \(O(d_v)\) work. Hence these operations, on the whole, can take \(O(|E|)\) time. This proves the proposition. \(\square\)

3.3 **Maintaining History and Other Issues with IN1 and OUT1**

When a merged node \(h\) is ejected, we might need to convert it back to \(j\). For this purpose, more information is maintained by keeping the history of each edge along with it. This history tells if the edge is due to edges from vertex \(j\) or from vertex \(r\). Since accept/eject operations on the neighbors of a vertex cause the edges on the vertex to
get changed, the history needs to be maintained dynamically. The problem is only with the accept action since it adds new edges. Consider the following situation when a label on an edge denotes the vertex it came from. We only use i and j as the labels since we only need to know if an edge came from the vertex on which a ging was performed (i - in/out degree = 1) or the other vertex (j - which gets a ged as a result of deducing on another vertex).

When vertex e is accepted, the history of the new edges (a, h) and (b, h) is determined by the history on the edge (e, h). One can easily verify that such a simple system of maintaining history makes the action of a ging confluent with accept operations taking place in the rest of the graph.

Another issue we had to consider was that due to the constaints on e, a variable might get assigned due to assignments to other variables. This causes a problem with the me ged nodes since the constraints imposed on i and j are not reflected entirely on h by the me ging p ocedu e. To take care of this, we look at the nodes i and j for such assignments and effect them on the me ged node h. The following is done if any of i or j are both assigned.

<table>
<thead>
<tr>
<th>V_i</th>
<th>V_j</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Reject h</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Convent h to j and accept</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>Convent h to j</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Reject h</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Accept h</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>Remove history on edges of h. Now h just represents j</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>Reject h</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>Convent h back to i and j and accept</td>
</tr>
</tbody>
</table>

X means unassigned.

We can see from the above discussion that ejection of a me ged node does not necessarily mean that the node will disappear. It might get conveted to another node. This illustrates why we cannot trivially extend the me ging p ocedu e and allow for me ging of me ged nodes as well. Rejection of o dina y nodes mean that they actually get removed from the graph which is not the case with me ged nodes. In order to handle me ging of me ged nodes, each time an assignment is made on me ged node, we would have to evert back to the original graph and do the changes. For me e,
the time complexity of the filtering algorithm with complete guesses would become in \( O(|E| \times |V|^2) \). For these reasons, the choice made in our current implementation has been to trade some pruning capabilities for efficiency, so we don’t allow guessing of logical nodes.

4 Search Heuristics

The internal graph managed by the cutset constant p provides interesting information which can be used to build heuristics for guiding the search in cutset problems.

In particular we know that the IN0 and OUT0 vertices, that have been deleted from the internal graph managed by the cutset constant, are not anymore cutset vertices by the cutset constant, and can thus be easily accepted or rejected. In practice minimum cutset p problems, these vertices should be immediately accepted. On the other hand, in mixed p problems whose cutset constant is combined with other boolean constants, the labeling of the IN0 and OUT0 vertices can be delayed as it no longer affects the graph of the cutset constant.

The LOOP vertices lead to automatically eject vertices of the original graph, except in the case of an ambiguity between the original vertices which a vertex responsible for the loop. The vertices belonging to such loops a cutset attended by a boolean clause that has effect to eject at least one of them. In practice minimum cutset p problems, the e is one labeling which sees the size of the minimum cutset \([8, 9]\) and which should be immediately done. In mixed p problems, the vertices belonging to a loop should be labeled first altogether.

Conceiving the remaining vertices, the vertices with the highest input degree are likely to break cycles in the graph. The experience with the GRASP procedure suggests that the selection of the vertex which maximizes the sum of the in and out degree p provides better results than maximizing the maximum of the in and out degree, or than maximizing the product \([10]\).

In the experiments reported below on log-based reconciliation problems, we label first the nodes with highest sum of in and out degree, and label at the end the nodes having an in or out degree equal to zero.

5 Log-based reconciliation

Our interest in the design of a global constant for cutset p problems a possible from the study of log-based reconciliation p problems in nomadic applications \([7]\), where the minimum cutset p problem shows up as the central p problem responsible for the NP-hardness of optimal reconciliation \([3]\). Nomadic applications create replicas of shared objects that evolve independently while they are disconnected. When reconnecting, the system has to reconcile the different replicas. Log-based reconciliation is a novel approach in which the input is a common initial state and logs of actions that were performed on each replica \([7]\). The output is a consistent global schedule that maximises the number of accepted actions. The reconcile use the logs according to the schedule, and replays the operations in the node log against the initial state, yielding to a reconciled common final state. We thus have to reconcile a set of logs of actions that have been executed independently, by trying to accept the greatest number of actions possible:
Input: A finite set of $L$ initial logs of actions $\{[T^1_i, \ldots, T^n_i] \mid 1 \leq i \leq L\}$, some dependencies between actions $T^j_i \Rightarrow T^k_i$, meaning that if $T^j_i$ is accepted then $T^k_i$ must be accepted, and some precedence constraints $T^j_i < T^k_i$, meaning that if the two actions $T^j_i$, $T^k_i$ are accepted, they must be executed in that order. The precedence constraints are supposed to be satisfied inside the initial logs.

Output: A subset of accepted actions, of maximal cardinality, satisfying the dependency constraints, given with a global schedule $T^j_i < \ldots < T^k_i$ satisfying the precedence constraints.

Note that the output depends solely on the precedence constraints between actions given in the input. Particularly, the output is independent of the precise structure of the initial logs. The initial consistent logs, that can be used as starting solutions in some algorithms, can be found as well without affecting the output. A log-based reconciliation problem over $n$ actions can thus be modeled with $n$ boolean variables, $\{a_1, \ldots, a_n\}$, associated to each action, satisfying:

- the dependency constraints are respected with boolean implications, $a_i \Rightarrow a_j$
- the precedence constraints are respected with a global cutset constraint over the graph of all (integrity) precedence constraints between actions.

In the next section, we compare this modeling with our previous modeling without the cutset constraint [3], where the precedence constraints were handled as in a scheduling problem, that is:

- by associating to the actions $n$ integer variables $p_1, \ldots, p_n$, giving the position of the action in the global schedule, whenever the action is accepted,
- by respecting the precedence constraints with conditional inequalities

$$a_i \land a_j \Rightarrow (p_i < p_j)$$

- or equivalently, assuming false is 0 and true is 1,

$$a_i \land a_j \land p_i < p_j.$$ 

In that modeling, the search solutions went through an enumeration of the boolean variables $a_i$'s, with the heuristic of instantiating first the variable $a_i$ which has the greatest number of constraints on it (i.e., first fail p incase w.r.t. the number of posted constraints) and then fixing it to the value 1 (i.e., best-first search for the maximization problem) [3].

6 Computational Results

In this section, we provide some computational results which show the efficiency of the global cutset constraint. The first set of benchmarks are the set of public minimum cutset problems posed by Funke and Reinelt for evaluating their branch-and-cut algorithms implemented in CPLEX [5], see also [10]. The second set of benchmarks is a set of log-based reconciliation problem problems [3]. We provide the timings obtained with

<http://constraintes.imria.fr/~fages/Reconcile/Benches.tar.gz>
and without the cutset const aint. The CLP p og am which does not use the cutset const aint is the one desc ibed in the p evious section.

The results e p oted below have been obtained with our prototype implementation of the cutset const aint in Sictus P olog ve sion 3.8.5 using the standa d inte face of Sictus P olog fo de ning global const aints in P olog [12]. The timings have been measu ed on a Pentium III at 600 Mhz with 256Mo RAM unde Linux. They a e given in seconds.

### 6.1 Funke and Reinelt’s benchmarks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>solution</td>
<td>time</td>
<td>time</td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>2520</td>
<td>14</td>
<td>2.43</td>
<td>8.95</td>
<td>0.22</td>
<td>1.42</td>
</tr>
<tr>
<td>2530</td>
<td>13</td>
<td>3.15</td>
<td>5.57</td>
<td>0.53</td>
<td>0.84</td>
</tr>
<tr>
<td>3020</td>
<td>19</td>
<td>21.91</td>
<td>48.92</td>
<td>0.71</td>
<td>1.55</td>
</tr>
<tr>
<td>3030</td>
<td>14</td>
<td>3.49</td>
<td>16.63</td>
<td>0.95</td>
<td>1.81</td>
</tr>
<tr>
<td>3520</td>
<td>18</td>
<td>5.66</td>
<td>214.91</td>
<td>3.12</td>
<td>3.29</td>
</tr>
<tr>
<td>3530</td>
<td>14</td>
<td>14.37</td>
<td>167.48</td>
<td>3.45</td>
<td>4.28</td>
</tr>
</tbody>
</table>

Table 1: Computational results on Funke and Reinelt’s benchmarks.

Table 1 summ aizes our computational results on Funke and Reinelt’s benchmarks. The fi st number in the name of the benchmark indicates the number of vertices. The second number in the name indicates the density of the graph, as a percentage. The second column gives the number of accepted vertices in the optimal solution. The following columns indicate the CPU time for finding the optimal solution, and the CPU time for the proof of optimality, for each of the two CLP programs without and with the cutset const aint.

The results on these benchmarks show an impovement by one to two orders of magnitude of the CLP program with the global cutset constant, especially on the CPU time for proving the optimality of solutions. It is difficult to make precise comparisons with the results obtained by Funke and Reinelt with CPLEX because their experiments were done on a SUN Sparc 10/20. Nevertheless, they times we e in minutes on these benchmarks, and more than one hour on the last two. This shows a much better performance of the cutset constant over the polyhedral method e p oted in [5]. On the other hand, it is wo th noting that the GRASP method remains much faster for finding good solutions that are in fact optimal in these benchmarks [10]. The GRASP metaheuristic would thus be wo th implementing in CLP with the cutset constant fo finding the first solutions.

### 6.2 Log-based reconciliation benchmarks

Table 2 shows the running times of the cutset constant on the benchmarks of reconciliation problems described in [3]. These problems have been generated with a low density of 1.5 for precedence and dependency constants. The sets of benchmarks are pu e minimum cutset problems containing no dependency constants. The number in the
name of the benchmark is the number of actions (vertexes). The table gives the number of accepted actions in the optimal solution, and for each version of the CLP program, without and with the global cutset constraint, we indicate the CPU time for finding the optimal solution, and for making the proof of optimality. Compared to our previous results without the global cutset constraint reported in [3], the e is a slow down which is due to the use of Siestus P olog instead of GNU-P olog for making the experiments.

<table>
<thead>
<tr>
<th>Bench</th>
<th>Optimal solution</th>
<th>without cutset</th>
<th>Opt. time</th>
<th>Proof time</th>
<th>with cutset</th>
<th>Opt. time</th>
<th>Proof time</th>
</tr>
</thead>
<tbody>
<tr>
<td>t40v1</td>
<td>36</td>
<td>0.03</td>
<td>3.13</td>
<td>0.03</td>
<td>0.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t40v2</td>
<td>37</td>
<td>1.44</td>
<td>0.68</td>
<td>0.02</td>
<td>0.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t40v3</td>
<td>38</td>
<td>0.02</td>
<td>0.07</td>
<td>0.01</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t40v4</td>
<td>37</td>
<td>0.93</td>
<td>0.60</td>
<td>0.08</td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t50v1</td>
<td>45</td>
<td>9.90</td>
<td>31.71</td>
<td>0.03</td>
<td>0.11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t50v2</td>
<td>47</td>
<td>1.16</td>
<td>0.09</td>
<td>0.08</td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t50v3</td>
<td>44</td>
<td>9.03</td>
<td>44.93</td>
<td>0.04</td>
<td>1.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t50v4</td>
<td>46</td>
<td>1.10</td>
<td>0.35</td>
<td>0.06</td>
<td>0.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t70v1</td>
<td>68</td>
<td>2.63</td>
<td>0.34</td>
<td>0.11</td>
<td>0.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t70v2</td>
<td>67</td>
<td>0.07</td>
<td>1.36</td>
<td>0.05</td>
<td>0.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t80v1</td>
<td>76</td>
<td>?</td>
<td>?</td>
<td>0.14</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t100v1</td>
<td>94</td>
<td>?</td>
<td>?</td>
<td>19.00</td>
<td>38.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t200v1</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t500v1</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t800v1</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t1000v1</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100v1</td>
<td>98</td>
<td>0.10</td>
<td>0.20</td>
<td>0.08</td>
<td>0.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100v2</td>
<td>77</td>
<td>0.26</td>
<td>0.48</td>
<td>0.07</td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100v3</td>
<td>95</td>
<td>0.34</td>
<td>0.57</td>
<td>0.10</td>
<td>0.13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100v4</td>
<td>100</td>
<td>0.08</td>
<td>0.02</td>
<td>0.03</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100v5</td>
<td>52</td>
<td>0.10</td>
<td>0.06</td>
<td>0.08</td>
<td>0.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200v1</td>
<td>65</td>
<td>0.43</td>
<td>0.16</td>
<td>0.11</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200v2</td>
<td>191</td>
<td>239.77</td>
<td>288.71</td>
<td>2.42</td>
<td>3.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500v1</td>
<td>198</td>
<td>1.42</td>
<td>0.99</td>
<td>1.00</td>
<td>0.35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>800v1</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>800v2</td>
<td>318</td>
<td>3.89</td>
<td>12.68</td>
<td>3.85</td>
<td>1.65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000v1</td>
<td>389</td>
<td>5.88</td>
<td>3.97</td>
<td>5.54</td>
<td>0.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000v2</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Computational results on log-based reconciliation benchmarks.

6.3 Discussion

The advantage of the heuristic selecting the highest degree vertex is reflected both in the first solution found which is accurate and takes little time, and in the total execution
of the program i.e., including the proof of optimality. We could also look into some modifications of this heuristic. Low degree vertices cause a little change in the graph, so if we could select those vertices that would change the graph enough so that more edges could take place, then we might have more reduction in the search space.

For the pruning of the global constant, the IN1 and OUT1 constant actions should be implemented without estimation. For this, mea ging of me ged node should be allowed and if that is done then case has to be taken that ejection of a vertex would not mean that it will disappear from the graph. The best way to implement this would be to unmea ge each time a me ged node is assigned and then perform the changes. Also, the cases when external const aints cause those vertices to get assigned which have been me ged to form a new vertex, would have to be handled app op iately. The assignment to these vertices would have to be effected onto the me ged node for the program to work properly.

Another improvement that can be made is to change the presentation of the graph to speed up the time that the reductions take. The presentation can be changed from maintaining adjacency lists to maintaining an adjacency matrix, as done in GRASP implementation. This will make lookups like finding self loops, constant time.

7 Conclusion

The cutset constant we propose is a global boolean constant defined by a graph $G = (V, E)$. We have provided a filtering algorithm based on graph constant actions and inference of simple boolean constants. The time complexity of this algorithm is $O(|E| + |V| \log |V|)$, thanks to a trade-off between the pruning capabilities and the efficiency of one cutset constant program operation.

The computational results we have presented on benchmarks of the type attack and on log-based reconciliation problems, shows a speed-up by one to two orders of magnitude thanks to the global cutset constant, and shows much better performance than polyhedra methods for program optimality of solutions.

As for future work, we expect to further improve the pruning capabilities of our current filtering algorithm while keeping a reasonable amount of complexity. Our implementation will also need to be improved to handle larger graphs, and use the cutset constant in a similar fashion to the GRASP program [10] for finding first solutions.

Acknowledgement

The first author would like to thank Marc Shapiro and his team for interesting discussions on log-based reconciliation, Philippe Chétienne and Francisco Souza for their comments on feedback vertex set problem, Mauic Resende for providing us Funke and Reinelt’s benchmarks, and the reviewers for their comments.

References

[1] N. Beldiceanu. Global constant aints as g apgh p ope ties on a st uctu ed netwo k of elementa y const aints of same type. In Proc. of sixth Conference on Principles and


