Proceedings CPAIOR'03

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 arbitrary 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 areas, such as deadlock breaking [2], program verification [11] or Bayesian inference [15].

There are a few classes of graphs for which the minimum cutset problem has a polynomial time complexity. These classes are defined by certain reducibility properties of

*This work was done while the second author was at INRIA for a summer internship of the Indian Institute of Technology Delhi, New Delhi 110016, India.

the graph. Shamir [11] proposed a linear time algorithm for reducible flow graphs. Rosen [2] modified this algorithm in an approximate algorithm for general graphs. Wang, Lloyd and Soffa [14] found an $O(|E| * |V|^2)$ algorithm for an unrelated class of cyclically reducible graphs. Smith and Walford [13] proposed an exponential time algorithm for general graphs that behaves in $O(|E| * |V|^2)$ in certain classes of graphs. The comparison of these different reducibility properties was done by Levy and Low [8] and Lou Soffa and Wang [9] who proposed an $O(|E| * log|V|)$ approximate algorithm based on a simple set of five graph contraction rules. Pardalos, Qian and Resende [10] used these contraction rules inside a Greedy Randomized Adaptive Search Procedure (GRASP). The GRASP procedure is currently the most efficient approximate algorithm for solving large instances, yet without any guarantee on the quality of the solution found. Exact solving has been tried with polyhedral methods by Funke and Reinelt who presented computational results with a branch-and-cut algorithm implemented in CPLEX [5].

Our aim here is to develop a constraint programming approach to cutset problems and design a global constraint for cutsets. Specialized propagation algorithms for global constraints are a key feature of the efficiency of constraint programming (see [1] for example). The idea of this paper is to embed relevant graph reduction techniques into a global cutset constraint, that can be combined with other boolean constraints, and that can be used within either a branch-and-bound procedure or local search methods.

Our interest for cutset problems, arose from the study of log-based reconciliation problems in nomadic applications [7]. The minimum cutset problem shows up as the central problem responsible for the NP-hardness of optimal reconciliation [3]. In this context however, the cutset constraint comes with other constraints which aggregate vertices into clusters, or more generally, express dependency constraints between vertices. In our previous constraint-based approach [3], the acyclicity constraint was expressed as a scheduling problem mixing boolean and finite domain constraints. We show in this paper that the global cutset constraint provides a more efficient pruning of the acyclicity condition. Moreover it allows for an all boolean modeling of log-based reconciliation problems.

The rest of the paper is organized as follows. In the next section, we define the global cutset constraint, and propose a syntax for its implementation in constraint logic programming (CLP). In section 3, we propose a filtering algorithm based on graph reductions and simple boolean constraint inference. We show its correctness, discuss some implementation issues and prove its $O(|E| + |V|log|V|)$ time complexity. In section 4, we discuss some search heuristics based on the properties of the internal graph managed by the global cutset constraint. Section 5 describes the log-based reconciliation problem in nomadic applications and its modeling with the cutset constraint. Section 6 presents computational results on Funke and Reinelt's benchmarks for pure minimum cutset problems, and on benchmarks of log-based reconciliation problems. The last section presents our conclusion.

2 The Cutset Constraint

Given a graph $G = (V, E)$, we consider the set B of boolean variables, obtained by associating a boolean variable to each vertex in V. A vertex is said to be *accepted* if its boolean variable is true, and is said to be *rejected* if its boolean variable is false. We consider the boolean constraint on B

$$cutset(B, G)$$

which states that the subset of rejected vertices according to B forms a valid cutset of G.

More specifically, we shall consider the implementation of the following constraint logic programming (CLP) predicates:

```
cutset(Variables,Vertices,Edges)
cutset(Variables,Vertices,Edges,Size)
```

where `Variables=[V1,...,Vn]` is the list of boolean variables associated to the vertices, `Vertices = [a1,...,an]` is the list of vertices, `Edges = [ai-aj,... ]` is the list of directed edges represented as pairs of vertices, and `Size` is a finite domain variable representing the size of the cutset, i.e. the number of rejected vertices. The boolean variable V_i equals 0 if the vertex a_i is in the cutset (i.e. rejected 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 rejecting 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:

```
cutset(B,V,E,S),minimize(labeling(B),S).
```

As usual, the cutset constraint does not make any assumption on the other constraints imposed on its variables and hence the user is allowed to qualify the cutset solution he wants with extra constraints. For this reason, the cutset constraint 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 contraction operations to reduce the graph size, check the acyclicity of the graph and bound the size of its cutsets. The graph contraction rules we use are inspired from the rules of Levy and Low [8], and Lloyd, Soffa and Wang [9] for computing one minimum cutset. The graph contraction operations described in this section compute an arbitrary cutset, in a constraint propagation setting.

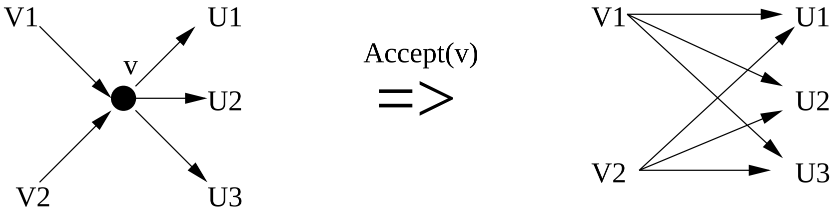
The cutset constraint maintains an internal state composed of an explicit representation of the graph, that is related to the constraints of the constraint store on the boolean variables, $V_1, ..., V_n$, associated to the vertices of the graph. The filtering algorithm tries to convert the information in the graph (about the cycles that have to be cut) to constraints over the boolean variables V_i. On completing such conversion, any valid solution of the constraint store is checked to provide a valid cutset of the original graph. The essential components of the filtering algorithm are the graph contraction operations. They either simplify the graph without loosing any information, or convert some information into explicit constraints and simplify the graph in the process.

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

3.1 Interrnal *Accept* and *Reject* Operations

We consider the two following operations on a directed graph:

1. **Accept(v) :** under the precondition 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_1, v_2) if (v_1, v) and (v, v_2) were edges in the original graph.



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 preclude the instanciation of the boolean variables associated to the vertices of the graph. If a boolean variable is instanciated, the filtering algorithm performs 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 perform *Accept* and *Reject* operations on its internal graph structure without instanciating 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 $v \in V$ be a vertex of the graph such that $(v, v) \notin E$. 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 : ($\Rightarrow$) Let $S \subseteq V$ be a cutset of G and $v \notin S$. Let $G \backslash S$ denote the graph obtained by removing the vertices of S from G. Since S is a cutset, $G \backslash S$ should be acyclic. Now, suppose that S is not a cutset of G'. Therefore, there exists a cycle $v_1, v_2, \ldots, v_n, v_1$ in G' with each vertex in $V' - S$. If this cycle has no edges which came due to the operation *Accept*(v) then this is also a cycle in $G \backslash S$. Hence this cycle has edges induced by the accept operation. By replacing each such edge (v_i, v_{i+1}) by (v_i, v) and (v, v_{i+1}), we again get a cycle in $G \backslash S$. Hence, by contradiction, we have one side of the result.
($\Leftarrow$) Let $S \subseteq V'$ be a cutset of G'. Again, suppose that S is not a cutset of G. Therefore, there exists a cycle $v_1, \ldots, v_n, v_1$ in $G \backslash S$. If none of these vertices is v then this is also a cycle in $G' \backslash S$. Hence, at least one of these vertices is v. If $v_i = v$ then replace the edges (v_{i-1}, v_i) and (v_i, v_{i+1}) by (v_{i-1}, v_{i+1}) to get a cycle in $G' \backslash S$. Again we get a contradiction. $\square$

The *accept* operation can thus be used to check if a given set is a cutset or 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 vertices or edges.

Hence this has a cutset $\emptyset$. Now, by repeated application of proposition 1, $\emptyset$ is also a cutset of G. Hence G is acyclic. The reverse can also be proved similarly by using proposition 1. So if G is a acyclic graph, then it has the cutset $\emptyset$. Now, while accepting the vertices of G, if we get a vertex with a self loop, then that graph cannot have $\emptyset$ as the cutset. However, $\emptyset$ should have been a cutset by proposition 1. Hence by contradiction, we have our result. $\square$

Similarly, 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 $Reject(v)$ 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* Reject(v) *if v has a self loop.*

These propositions show that the *accept* and *reject* operations have the nice property of maintaining any cutset by picking a right vertex to apply the operation on. If there is a minimum cutset that contains the vertex v then after the operation $Reject(v)$, we can still find that cutset but have a smaller graph to work with. Similarly, if there is a minimum cutset that does not contain v then after $Accept(v)$, we can still find that cutset but again in a smaller graph.

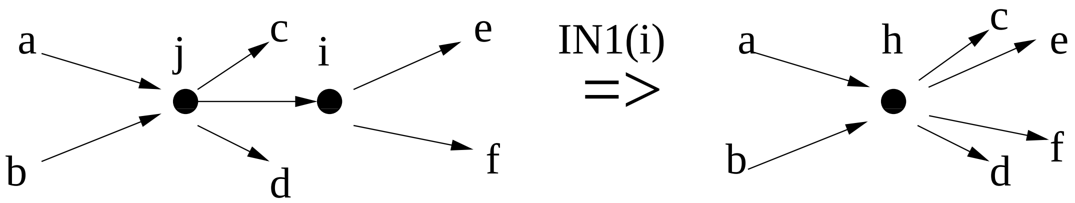
3.2 Graph Contraction Operations

We shall use the following five graph contraction operations :

1. **IN0** (In degree = 0) In case the in degree of a vertex is zero, that vertex cannot be a part of any cycle. Hence its acceptance or rejection will cause no change to the rest of the graph. So, its edges are removed but no constraints are produced since a cutset can exist including or excluding this vertex.

2. **OUT0** (Out degree = 0) In case the out degree of a vertex is zero, the situation is similar to the one above. Again, the edges incident on this vertex are removed and no constraints are produced.

3. **IN1** (In degree = 1) In case a vertex has in degree one, then the situation is as follows,



If a cycle passes through i then it must also pass through j. Hence by merging these two nodes to form the node h, we do not eliminate any cycle in the graph. Along with this reduction, we impose the constraint $V_h = V_i \wedge V_j$ on the variables associated with the vertices. This captures the fact that if h is not a part of any cycle, then both i and j were 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 trading pruning 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.

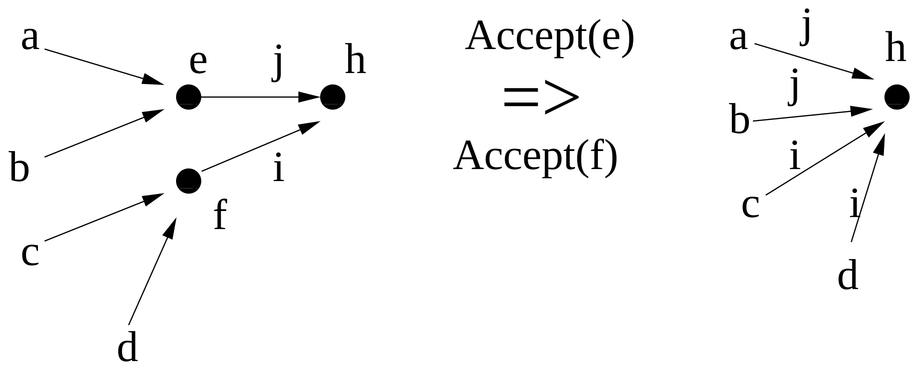
5. **LOOP** (Self loop on a vertex) In case a vertex has a self loop then this vertex is rejected and its boolean variable is set to 0 since no cutset can exist without including this vertex. However, if the vertex is a merged node h then we impose $V_h = 0$ but cannot reject h since that implies rejection of both i and j. So we look at the self loop edge of h and figure out if it came from the loop (j, j) or from $(i, j), (j, i)$. This is done by maintaining history on merged nodes, as described in the next section. Note that there cannot be a loop (i, i) since i had *in* (resp. *out*) degree as one and this edge was not a loop. If the loop came purely from 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 rejected . Choosing to reject either renders 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 procedure by sorting, in $O(|V|log|V|)$, the vertices based on *in* and *out* degrees and store the result in two arrays 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 degrees of affected vertices while maintaining the arrays correctly. Since new edges are not added to the graph at any stage, any number of **IN0** operations interleaved with any number of different reductions can take atmost $O(\sum_v d_v) = O(|E|)$ time. Similarly, any number of **OUT0** reductions can take time $O(|E|)$. For the **LOOP** case, we can see that it too leads to rejection 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 argue since we are not allowing merged nodes to get merged. As a result, we look at a vertex atmost once and do $O(d_v)$ work. Hence these operations, on the whole, can take $O(|E|)$ time. This proves the proposition. □

3.3 Maintaining History and Other Issues with IN1 and OUT1

When a merged node h is rejected, 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 there due to edges from vertex j or from vertex i. Since accept/reject 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 operation since it adds new edges. Consider the following situation where 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 merging was performed (i - in/out degree=1) or the other vertex(j - which gets merged as a result of reduction 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 merging *confluent* with *accept* operations taking place in the rest of the graph.

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

V_i	V_j	*Action*
0	0	Reject h
0	1	Convert h to j and accept
0	X	Convert h to j
1	0	Reject h
1	1	Accept h
1	X	Remove history on edges of h. Now h just represents j
X	0	Reject h
X	1	Convert h back to i and j and accept j

X means unassigned.

We can see from the above discussion that rejection of a merged node does not necessarily mean that the node will disappear. It might get converted to another node. This illustrates why we cannot trivially extend the merging procedure and allow for merging of merged nodes as well. Rejection of ordinary nodes mean that they actually get removed from the graph which is not the case with merged nodes. In order to handle merging of merged nodes, each time an assignment is made on the merged node, we would have to revert back to the original graph and do the changes. Furthermore, the time complexity of the filtering algorithm with complete merges would become in $O(|E| * |V|^2)$. For these reasons, the choice made in our current implementation has been to trade some pruning capabilities for efficiency, so we don't allow merging of merged nodes.

4 Search Heuristics

The internal graph managed by the cutset constraint 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 constraint, are not anymore constrained by the cutset constraint, and can thus be freely accepted or rejected. In pure minimum cutset problems, these vertices should be immediately accepted. On the other hand, in mixed problems where the cutset constraint is combined with other boolean constraints, the labeling of the **IN0** and **OUT0** vertices can be delayed as it no longer affects the graph of the cutset constraint.

The **LOOP** vertices lead to automatically reject vertices of the original graph, except in the case of an ambiguity between the original vertices which are responsible for the loop. The vertices belonging to such loops are constrained by a boolean clause that has for effect to reject at least one of them. In pure minimum cutset problems, there is one labeling which preserves the size of the minimum cutset [8, 9] and which should be immediately done. In mixed problems, the vertices belonging to a loop should be labeled first alltogether.

Concerning the remaining vertices, the vertices with the highest in or out degrees are more 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 degrees provides better results than maximizing the maximum of the in and out degrees, or than maximizing their product [10].

In the experiments reported below on log-based reconciliation problems, we label first the nodes with highest sum of in and out degrees, 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 constraint for cutset prolems arose from the study of log-based reconciliation problems in nomadic applications [7], where the minimum cutset problem shows up as the central 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 divergent 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 reconciler merges the logs according to the schedule, and replays the operations in the merged 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 realized independently, by trying to accept the greatest number of actions possible:

Input: A finite set of L initial logs of actions $\{[T_i^1, ..., T_i^{n_i}] \mid 1 \leq i \leq L\}$, some dependencies between actions $T_i^j \Rightarrow T_k^l$, meaning that if T_i^j is accepted then T_k^l must be accepted, and some precedence constraints $T_i^j < T_k^l$, meaning that if the two actions T_i^j, T_k^l, 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 $T_i^j < ... < T_k^l$ satisfying the precedence constraints.

Note that the output depends solely on the precedence constraints between actions given in the input. In particular 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 forgotten as well without affecting the output. A log-based reconciliation problem over n actions can thus be modeled with n boolean variables, $\{a_1, ..., a_n\}$, associated to each action, satisfying:

- the dependency constraints represented with boolean implications, $a_i \Rightarrow a_j$
- the precedence constraints represented with a *global cutset constraint* over the graph of all (inter-log) precedences 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 $p_1, ..., p_n$, giving the position of the action in the global schedule, whenever the action is accepted,
- by representing the precedence constraints with conditional inequalities

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

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

  $$a_i * a_j * p_i < p_j.$$

In that modeling, the search for 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 principle w.r.t. the number of posted constraints) and trying first 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 series of benchmarks are the set of pure minimum cutset problems proposed by Funke and Reinelt for evaluating their branch-and-cut algorithm implemented in CPLEX [5], see also [10]. The second series of benchmarks is a series of log-based reconciliation problems[1] [3]. We provide the timings obtained with

[1] `http://contraintes.inria.fr/~fages/Reconcile/Benchs.tar.gz`

and without the cutset constraint. The CLP program which does not use the cutset constraint is the one described in the previous section.

The results reported below have been obtained with our prototype implementation of the cutset constraint in Sicstus Prolog version 3.8.5 using the standard interface of Sicstus Prolog for defining global constraints in Prolog [12]. The timings have been measured on a Pentium III at 600 Mhz with 256Mo RAM under Linux. They are given in seconds.

6.1 Funke and Reinelt's benchmarks

		without cutset		*with cutset*	
Bench	*Optimal solution*	*Opt. time*	*Proof time*	*Opt. time*	*Proof time*
r_25_20	14	2.43	8.95	0.22	1.42
r_25_30	13	3.15	5.57	0.53	0.84
r_30_20	19	21.91	48.92	0.71	1.55
r_30_30	14	3.49	16.63	0.95	1.81
r_35_20	18	5.66	214.91	3.12	3.29
r_35_30	14	14.37	167.48	3.45	4.28

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

Table 1 summarizes our computational results on Funke and Reinelt's benchmarks. The first 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 percentile. 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 constraint,

The results on these benchmarks show an improvement by one or two orders of magnitude of the CLP program with the global cutset constraint, 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, their times were in minutes on these benchmarks, and more than one hour on the last two. This shows a much better performance of the cutset constraint over the polyhedral method reported in [5]. On the other hand, it is worth 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 worth implementing in CLP with the cutset constraint for finding first solutions.

6.2 Log-based reconciliation benchmarks

Table 2 shows the running times of the *cutset* constraint 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 constraints. The r series of benchmarks are pure minimum cutset problems containing no dependency constraints. The number in the name of the benchmark is the number of actions (vertices). 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], there is a slow down which is due to the use of Sicstus Prolog instead of GNU-Prolog for making the experiments.

		without cutset		*with cutset*	
Bench	*Optimal solution*	*Opt. time*	*Proof time*	*Opt. time*	*Proof time*
t40v1	36	0.03	3.13	0.03	0.06
t40v2	37	1.44	0.68	0.02	0.02
t40v3	38	0.02	0.07	0.01	0.01
t40v4	37	0.93	0.60	0.08	0.05
t50v1	45	9.90	31.71	0.03	0.11
t50v2	47	1.16	0.09	0.08	0.05
t50v3	44	9.03	44.93	0.04	1.22
t50v4	46	1.10	0.35	0.06	0.02
t70v1	68	2.63	0.34	0.11	0.04
t70v2	67	0.07	1.36	0.05	0.09
t80v1	76	?	?	0.14	0.23
t100v1	94	?	?	19.00	38.10
t200v1	?	?	?	?	?
t500v1	?	?	?	?	?
t800v1	?	?	?	?	?
t1000v1	?	?	?	?	?
r100v1	98	0.10	0.20	0.08	0.04
r100v2	77	0.26	0.48	0.07	0.05
r100v3	95	0.34	0.57	0.10	0.13
r100v4	100	0.08	0.02	0.03	0.01
r100v5	52	0.10	0.06	0.08	0.08
r200v1	65	0.43	0.16	0.11	0.01
r200v2	191	239.77	288.71	2.42	3.27
r500v1	198	1.42	0.99	1.00	0.35
r800v1	?	?	?	?	?
r800v2	318	3.89	12.68	3.85	1.65
r1000v1	389	5.88	3.97	5.54	0.43
r1000v2	?	?	?	?	?

Table 2: Computational results on log-based reconcliation 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 graph reductions could take place, then we might have more reduction in the search space.

For further improvement of the pruning of the global constraint, the IN1 and OUT1 contraction operations should be implemented without restriction. For this, merging of merged node should be allowed and if that is done then care has to be taken that rejection of a vertex would not mean that it will disappear from the graph. The best way to implement this would be to unmerge each time a merged node is assigned and then perform the changes. Also, the cases when external constraints cause those vertices to get assigned which have been merged to form a new vertex, would have to be handled appropriately. The assignment to these vertices would have to be reflected onto the merged node for the program to work properly.

Another improvement that can be made is to change the representation of the graph to speed up the time that the reductions take. The representation 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* constraint we propose is a global boolean constraint defined by a graph $G = (V, E)$. We have provided a filtering algorithm based on graph contraction operations and inference of simple boolean constraints. 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 constraint propagation.

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

As for future work, we expect to further improve the pruning capabilities of our current filtering algorithm while keeping a reasonable amortized complexity. Our implementation will also need to be improved in order to handle very large graphs, and use the cutset constraint in a similar fashion to the GRASP procedure [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 Chrétienne and Francis Sourd for their comments on feedback vertex set problems, Mauricio Resende for providing us Funke and Reinelt's benchmarks, and the reviewers for their comments.

References

[1] N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of same type. In *Proc. of sixth Conference on Principles and*

Practice of Constraint Programming CP, volume 1894 of *Lecture Notes in Computer Science*, pages 52–66, Singapore, September 2000. Springer-Verlag.

[2] B.K.Rosen. Robust linear algorithms for cutsets. *Journal of Algorithms*, 3(1):205–212, 1982.

[3] F. Fages. A constraint programming approach to log-based reconciliation problems for nomadic applications. In *Proceedings of the Ercim/Compulog Net Workshop on Constraints*, June 2001.

[4] P. Festa, P.M. Pardalos, and M.G.C. Resende. Feedback set problems. In *In Handbook of Combinatorial Optimization*, volume 4. Kluwer Academic Publishers, 1999.

[5] M. Funke and G. Reinelt. A polyhedral approach to the feedback vertex set problem. In *Proceedings of the 5th Conference on Integer Programming and Combinatorial Optimization IPCO'96*, pages 445–459, Vancouver, Canada, June 1996.

[6] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations.* Plenum Press, New York, 1972.

[7] A.M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proc. of Twentieth ACM Symposium on Principles of Distributed Computing PODC*, Newport, RI USA, August 2001.

[8] H. Levy and D.W. Low. A contraction algorithm for finding small cycle cutsets. *Journal of algorithms*, 9:470–493, 1988.

[9] E.L. Lloyd, M.L. Soffa, and C.C. Wang. On locating minimum feedback vertex sets. *Journal of Computer and System Science*, 37:292–311, 1988.

[10] P.M. Pardalos, T. Qian, and M.G. Resende. A greedy randomized adaptive search procedure for feedback vertex set. *Journal of Combinatorial Optimization*, 2:399–412, 1999.

[11] A. Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *SIAM Journal of Computing*, 8(4):645–655, 1979.

[12] SICS. Sicstus prolog user's manual, 2001.

[13] W. Smith and R. Walford. The identification of a minimal feedback vertex set of a directed graph. *IEEE Transactions on circuits and systems CAS*, 22(1):9–15, 1975.

[14] C.C. Wang, E.L. Lloyd, and M.L. Soffa. Feedback vertex sets and cyclically reducible graphs. *Journal of the Association for Computing Machinery*, 32(2):296–313, 1985.

[15] B. Yehuda, J. Geiger, J.Naor, and R.M. Roth. Approximation algorithms for the vertex feedback set problem with application in constraint satisfaction and bayesian inference. In *Proc; 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 344–354, 1994.