Cycle-cut decomposition and log-based reconciliation

Youssef Hamadi
Microsoft Research Ltd.
7 J J Thomson Avenue
Cambridge CB3 0FB, UK
youssefh@microsoft.com

Abstract

Optimistic reconciliation allows, multiple update of shared data without synchronization. The assumption is that the vast majority of the actions will not conflict. In those systems, write availability is raised in the presence of network failures, high latencies or parallel development. However, in order to remain consistent, optimistic systems repair divergences. To produce a new consistent state, they use the logs of each user in a process called log-based reconciliation. The purpose of an efficient reconciliation engine is then to compute a new consistent state which preserves the maximum of previous actions. This work leverages the efficiency of constraint-based reconciliation. It provides a new efficient two step procedure built by connecting theoretical results on backtrack-free search with this hard optimisation problem.

Keywords: Cycle-cut set decomposition, Reconciliation, Constraint Programming.

Introduction

In many situations, applications perform actions on the same set of objects. In order to keep efficiency, object replication is introduced (Satyanarayanan *et al.* 1990; Petersen *et al.* 1996). When the users or applications are connected (groupware) it is possible to avoid potential conflicts through on-the-fly updates of object states. However, high networks latencies and disconnections make those online operations difficult. Log-based reconciliation is an optimistic technique which assumes that users or applications work in a disconnected environment. They share an initial consistent state of the objects and perform actions locally. Those isolated actions can introduce divergence in object states. When they reconnect, they transfer the log of their local actions to the system which performs reconciliation. The result is a new consistent state for each object. In this process, conflicts can appear between mobile users and the system may have to drop some actions to achieve consistency. The purpose of a log-based reconciliation engine is then to preserve the work done offline while removing any conflicting action. The engine computes a conflict-free plan with users' actions. This plan applied to the previous consistent state of the objects raises a new consistent state for each mobile user.

An efficient reconciliation engine must be able to quickly compute a planning maximizing[1] the number of actions. In this paper we propose such a new engine. Our method uses a problem decomposition based on the theoretical analysis of backtrack-free search (Freuder 1982). The idea is to split the actions in two sets. The first set is made by a cycle-cut. The second set represents an induced forest of actions. The solving process is then decomposed in two stages. A first branch and bound maximizes the number of selected actions in the cycle-cut. A second branch and bound extends the previous solution on the forest and still maximizes the number of non conflicting actions. The previous split makes the search sub-optimal. However it greatly improves the backtrack complexity. Experimental results show a large improvement in time performance with a small loss in solution quality.

The paper is organized as follow. In section 2, we present some previous work. Section 3 presents our initial constraint programming modeling. Problem's decomposition is detailed and discussed in section 4. Finally, before giving a conclusion, section 5 describes experimental results.

Previous work

The vast majority of reconciliation engines use dedicated heuristics to compute a valid plan of actions. For example, (Preguia *et al.* 2003; Kermarrec *et al.* 2001) computes a conflict-free plan of actions through a greedy algorithm successively applying specific heuristic rules to select an action.

More general approaches use generic constraint solvers to perform reconciliation (Fages 2001). The modeling uses the abstraction level of (Kermarrec *et al.* 2001) to define constraints relations. The main advantages of a constraint based approach are optimality combined to modeling simplicity and system integration. The main drawback is combinatorial explosion. In order to improve this last point, (Fages 2001) considered the use of local search. However, his conclusion is that it performs badly, according to the mix between boolean and integer variables.

An interesting challenge is then to raise the performance level of constraint-based reconciliation engines. In the following we show how to exploit theoretical complexity re-

[1] Unlike classical planning, here the new consistent state is reached by using the largest possible set of actions.

sults in order to raise the applicability level of constraint-based reconciliation engines.

Constraint programming modeling

We detail here our constraint modeling of log-based reconciliation. We build on the abstract definition of (Kermarrec *et al.* 2001). This allows an application-independent view of the problem which disconnects the engine from the application. As an immediate result, the applicability of our work is broadened.

The input is made by a set of logs upcoming from m users or applications $\{(a_{i_1}, ..., a_{ik_i}) | 1 \leq i \leq m\}$. Those disjoint plans are individually consistent.

The output is made by the largest subset of users' actions consistent with the following conditions:

1. 'Before' constraints (also called temporal), for any a_i, a_j in the final plan $a_i \rightarrow a_j$, a_i comes before a_j (not necessarily immediately before).
2. 'Must have' constraints (also called dependencies constraints), for any a_i in the final plan $a_i \rhd a_j$ means that a_j is also in the final plan.

The semantic of a large set of applications can be represented through the combination of these low level constraints (Kermarrec *et al.* 2001; Preguia *et al.* 2003). For instance if the method f_m of an object o needs to start with the call to an initialisation method f_i we can use the following constraints:

- $o.f_i \rightarrow o.f_m$
- $o.f_m \rhd o.f_i$

A consistent planning must respect any dependency relation and avoid oriented cycles of 'before' constraints.

The previous problem bounded to 'Before' constraints can obviously be reduced to the search of the largest acyclic network of actions (Karp 1972). The previous observation raises NP-hardness. On the other hand, the addition of the 'must have' dependencies globally reduces the complexity. The reduction is even larger when the relations are symmetricals ($a_i \rhd a_j \rhd a_i$). Indeed, the previous introduces equivalence classes between actions which is equivalent to a problem's size reduction (see, (Fages 2001) and our experimental results).

Invariants

From the previous definitions, we define a Constraint Programming model. The partition of the actions between users is dropped to consider as the input the whole set of n (possibly conflicting) actions. Each action a_i uses two constrained variables:

- $X_i = [0..1]$ where the value 1 is set when the action is included in the final planning.
- $S_i = [0..n-1]$ to represent the position of the action in the solution.

To express any $a_i \rightarrow a_j$ constraint, we put the following invariant:

$$(X_i == 1) \Longrightarrow (S_i < S_j)$$

We do not consider the possible exclusion of action a_j. Since when $X_j == 0$, the entailment of $S_i < S_j$ has no negative impact. Indeed, the pruning of position $n-1$ for S_i is consistent since the schedule can be computed with $n-1$ possible slots according to X_j exclusion. The benefit from the solver point of view is to handle a fewer number of constraints checks.

The representation of any $a_i \rhd a_j$ uses the following constraints:

1. $(X_i == 1) \Longrightarrow (X_j = 1)$
2. $(X_j == 0) \Longrightarrow (X_i = 0)$

The first invariant expresses the fact that the inclusion of action a_i cannot occur without the inclusion of action a_j. The second invariant is redundant, i.e., it does not change the solution space. However we found out that it improves search efficiency at the price of handling more constraints.

Search process

Since the rationale is to keep the largest number of actions, we define the following function which is maximized during the search process:

$$max(\sum_{i=0}^{n-1} X_i)$$

The branch and bound search is performed over the X_i with bound consistency on the S_i. At the end of the exploration, any selected action a_i (s.t., $X_i = 1$) can use the lower bound of the S_i as a valid plan location. The previous is a direct exploitation of the completeness of interval propagation with arithmetic constraints (Hentenryck 1989).

Cycle-cut decomposition

A cycle-cut set is made by a subset of the variables which removal breaks the cycles of a constraint network (Dechter 1990). The remaining network can then be structured as a tree or more generally as a forest induced by the cutset. The previous has an interesting impact on search complexity. Indeed, it is well known that the complexity of a backtrack search is connected to the width of the constraint network (Freuder 1982). More precisely, when the consistency degree achieved during constraint propagation is larger or equal to the width, we can prove that the search is backtrack free (Freuder 1982).

The previous results can greatly benefit to our reconciliation engine. Any cycle-cut computed on the X_i leaves an induced forest of variables. Since the width of any variable of the forest is 1 (i.e., each variable is connected to at most one variable), we know that the interval propagation which ensures bound consistency is backtrack free.

More clearly, the propagation of any selection of actions computed on the cutset allows a backtrack free exploration of the remaining set of variables.

The only pitfall here is that we do not limit our search to the first solution (i.e. satisfaction). Indeed, since we maximize the number of selected actions, we continuously bound the search according to the best quality in order to find a better solution (branch and bound). As an outcome, we will always perform backtracking steps. However from the previous results, we can expect a large reduction of those operations.

A two step search procedure The new search scheme uses the previously defined CP modeling. A preprocessing step computes a partition of the X-variables in two sets. The first set defines a *cycle-cut* of the constraint network. The algorithm starts by solving the cycle-cut subproblem. Each time a solution is found for the first set, it is extended to the remaining variables. The process then continues toward a better solution for the remaining problem. If the second search space is exhausted, the algorithm backtracks to the current solution of the cycle-cut and tries to improve it. The previous process is once again extended on the remaining subproblem. The search is finished when the tree-based exploration over the cycle-cut set is over (see figure 1).

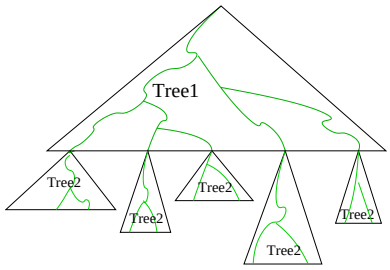

Figure 1: Successive backtrack searches

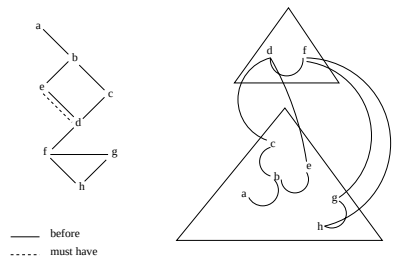

Figure 2: Cycle-cut set decomposition

Figure 2 presents a constraint network and a possible decomposition in two sub-problems. Plain arrows figure 'before' constraints, dashed lines represent 'mustHave' relations. On the left part of the figure the network presents several cycles. A cycle-cut is made of variables whose removal breaks any cycle of the original constraint network. In the figure at least two variables must be excluded. The right part presents the cycle-cut $\{d, f\}$ and its induced forest.

On the second tree several orderings can be used to perform the branch and bound. For instance, a, b, c, e has a width of 1 since in this ordering each variable is connected to at most one previously instanciated variable. The ordering c, e, b, a has width 2 since b is connected to both c and e. The first one is backtrack free, the second one can involve backtracking. It is always possible to compute the best ordering on the second tree, however we decided to approximate this calculation by using a max-degree variable ordering on the second tree. Indeed, ranking the nodes through their graph degree usually results in small ordering widths.

Cycle-cut calculation Finding the smallest cycle-cut set of a given graph is an NP-hard task. However, some heuristics give interesting results. Our preprocessing step computes a cycle-cut with the following method:

- Select a variable V with the highest degree in the constraint network
- Do a DFS from V
- If the DFS shows that V is part of a cycle, remove V and put it in the cycle-cut set
- Repeat this process until no cycles are detected

The previous algorithm returns with a reasonably small cycle-cut set.

Sub-optimality Obviously, our two step procedure is sub-optimal. This comes from the split of the problem in two sub-problems. The reasoning is then optimal according to each sub-part. Indeed, the algorithm starts a full exploration of the second tree each time a so far optimal solution is found in the first tree. But since a problem can have an optimal solution with a poor quality on the first tree, our search scheme can miss it. The following section allows us to quantify the previous observation.

Experiments

The experiments had two main goals. Firstly, assess the time efficiency of the two-step procedure on a large variety of reconciliation problems. Secondly, evaluate the impact on the quality of the sub-optimal exploration. To achieve those two goals, we decided to compare our procedure against a full branch and bound search. This second procedure is complete, i.e. able to find the best quality. Both methods were implemented using Disolver (Hamadi 2003) on a Pentium-4 running at 2Ghz.

We used two sets of problems. The first set is made by purely random problems. The second set was defined in collaboration with the authors of (Kermarrec *et al.* 2001). It figures common features observed on a large set of reconciliation applications.

Each problem is defined with the following parameters:

- number of actions in the initial logs
- tightness of 'before' constraints

- tightness of 'mustHave' constraints

The tightness represents a proportion over the complete graph of constraints. For instance a problem with 40 actions and a tightness of 0.5 has $0.5 \times 40 \times 39 = 780$ constraints.

The measures involve the CPU time, the number of backtrack and the final quality. Figured values represent the median of 50 instances.

Random problems

Randoms problems allow a general characterization of algorithm performances. They are well known to exhibit for some parameters very hard instances (Cheeseman, Kanefsky, & Taylor 1991; Slaney & Thiebaux 1998). Those instances are important to get a clear picture of algorithms performances.

In order to assess the algorithms over a large set of parameters, we decided to limit the size of the problems to 40 actions. This limitation was raised by the poor performance of the full branch and bound exploration used to assess our procedure.

General landscape Figure 3 gives the general landscape of the search effort (time) for both the *global* branch and bound and our *cycle-cut* method. The x-axis reports the percentage of 'before' constraints (from 0 to 80% of the possible constraints between 40 actions). The percentage of 'mustHave' constraints is set on the y-axis (from 0 to 5%). Finally, the time (CPU+kernel) is reported on the z-axis.

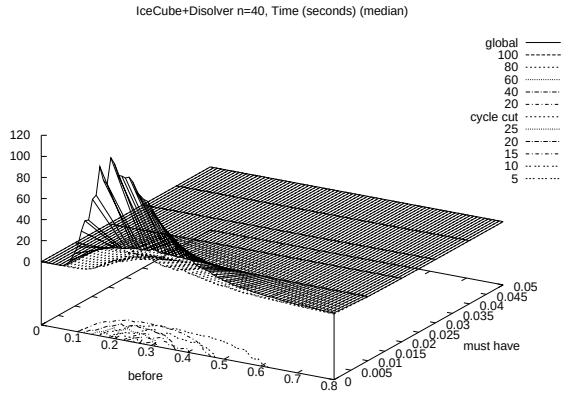

Figure 3: Random problems: time landscape in seconds

This landscape shows the phase transition region occurring at 20% of 'before' (about 300 constraints) and 0% of 'mustHave'. Clearly, the impact of dependency constraints is positive on the search complexity. Those constraints act as equivalence classes between actions. This is equivalent to the reduction of the number of variables (Fages 2001).

To understand this general landscape it is helpful to consider the quality achieved when the problems are free of 'mustHave' constraints (i.e., hardest configuration). This is presented in figure 7. We can observe that before the phase transition (left part of the x-axis), the quality is high. The branch and bound process is able to cut large parts of the space. Indeed, each time a solution is found, the quality is bounded to be higher that the current one. In such under-constrained space, the probability of having a higher quality than the current one is small since the quality is bounded by the number of actions. After the transition, the quality is poor but the number of constraints is large. Then, the probability of having higher quality than the current one is poor since the problem is over-constrained. At the phase transition, the average quality is about $n/2$ (see below) and the search effort of the algorithms is high since the probability of improving the current solution is high.

In order to clearly compare the two methods we decided to focus our presentation to the set of problems without dependency constraints (i.e., hardest problems). The results are presented in the following sections.

Time Figures 4 and 5 respectively show time results and time speed-up with 'mustHave' tightness set to 0.

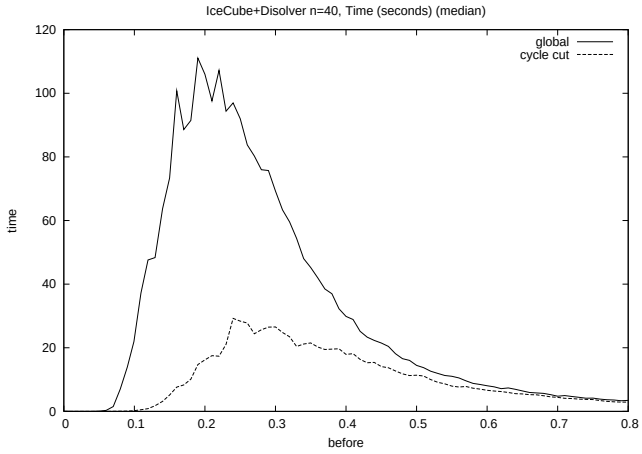

Figure 4: Random problems: time

In figure 4 we clearly see the phase transition behaviour for both methods. For the complete branch and bound search it occurs just before 20% where the median CPU+kernel time exceeds 100 seconds. For the same constraint tightness, the cycle-cut search gives a speed-up of about 6.

The speed-up curve is not defined everywhere. Indeed, the time value of cycle-cut was measured to zero for very small tightness. Those values made speed-up calculation impossible. The largest speed-up exceeds 160 and is achieved for tightness 0.09. Interestingly, the speed-up is never smaller that 1. The conclusion is that on those problems, cycle-cut is always 'as good' as the global search regarding time complexity. With large tightness, the size of the cycle-set increases and at the end, becomes equal to the whole problem. The search is then equivalent to the one done by the global branch and bound since the second tree becomes empty.

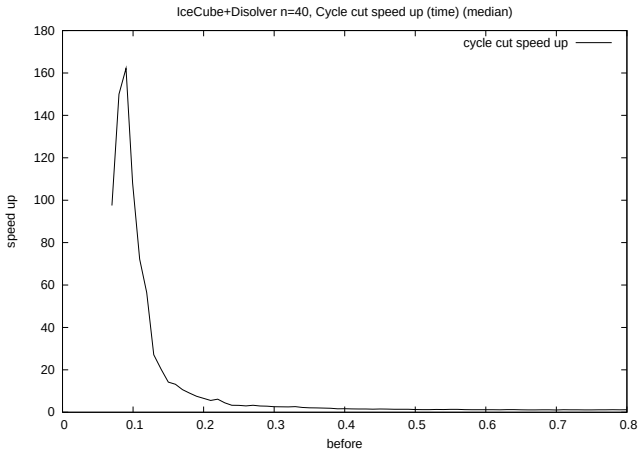

Figure 5: Random problems: time speed-up

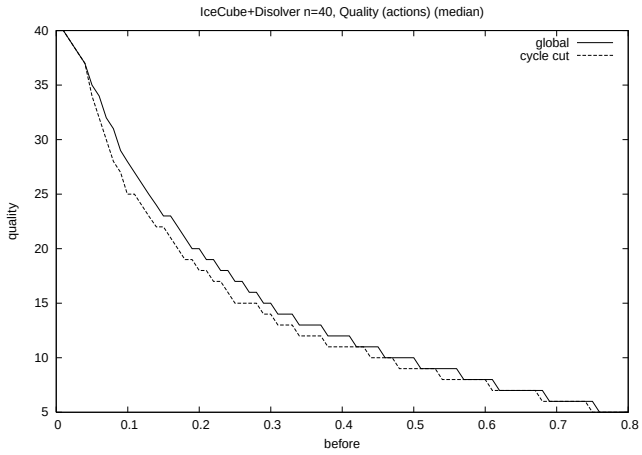

Figure 7: Random problems: quality

Backtracks When we consider backtracking, the results presented in figure 6 are consistent with the previous analysis. However, they confirm our previous observation on the infeasibility of backtrack-free search in an optimisation context. Hence the reported backtracks come from the optimisation process and from the sub-optimality of the orderings used to explore the second tree (see above).

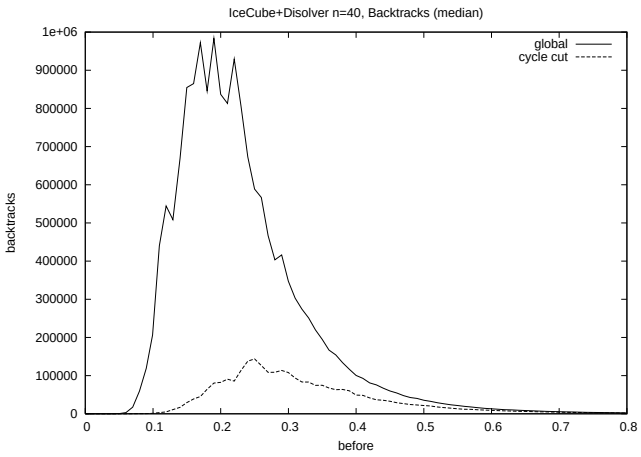

Figure 6: Random problems: backtracking

Quality From the previous results, we know that running a cycle-cut algorithm to perform log-based reconciliation is quite advantageous. A part of this large improvement (up to 160 speed-up) comes from the reduction in backtrack operations. Another part comes from the sub-optimality of the new procedure. Therefore, we can only appreciate observed speed-up in relation to quality loss. Figure 7 presents the quality deprivation.

The gap between the two curves represents the loss. Interestingly this loss is limited to 3 actions (about 7.5%). Not surprisingly the biggest loss occurs with the largest speed-up, i.e., around 0.09. For very low and high tightness there is no loss. The previous reasoning occurs since for such tightness the behaviour of cycle-cut is equivalent to the one of global search. For very low (resp. high) tightness, the first (resp. second) tree is empty.

Pseudo-reals problems

Previous results and analysis are useful to grasp the whole behavior of cycle-cut over a large set of parameters defining log-based reconciliation problems. However, real logs upcoming from distributed collaboration have some structure. In this section we define what we call 'pseudo-real problems' which are random problems respecting some regularities usually observed in log-based reconciliation problems (Kermarrec *et al.* 2001). Then we oppose both algorithms over a set of pseudo-real instances.

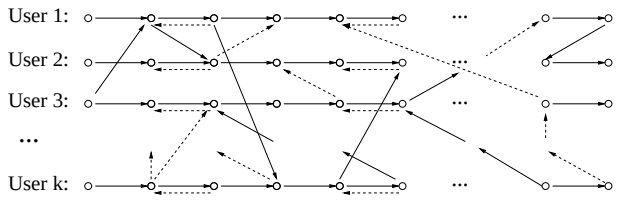

Figure 8: Pseudo-real problems

Figure 8 presents the general structure observed in real reconciliation problems. In those problems, you have a set of k users performing successive actions. Those actions share 'before' constraints stating temporal relationship between them. Sometime, some action is required by another one. We then have some dependency constraints (dashed lines). In addition to that some parcel task can occur. A parcel is made by a set of successive actions all interconnected through pairwise dependency links ('mustHave'). Parcels figure out some complex task involving the succession of

several actions to succeed. Between any couple of users, both precedence and dependency can appear. The figure gives a general characterization of such problems.

From the previous observations we have defined a problem generator which uses the following parameters:

- k the number of users
- n the number of successive actions performed by a user
- $p_{mustHave}$ the probability of having one dependency constraint between two successive actions
- p_{parcel} the probability of having dependency constraints involving any successive pm-set of actions
- For any pair of user, n_{inter} represents the number of inter-users constraints. Those constraints can be with equal probability temporal or dependency constraints.

k	*#actions*	*quality*	*time*	*#backtracks*
3	150	147(147)	0.03(0.01)	6(6)
4	200	193(193)	0.34(0.21)	190(128)
5	250	216(216)	4.70(4.18)	2952(1267)
6	300	104(104)	6.95(2.01)	2647(733)
7	350	123(123)	38.03(6.04)	260937(2452)
8	400	??(111)	??(10.87)	??(30574)

Table 1: Pseudo-real problems with 50 actions per user, $p_{mustHave} = 0.4, p_{parcel} = 0.2, pm = 5, n_{inter} = 10$

Table 1 presents results with pseudo real problems. For each k value, we generated 10 instances with 50 actions per user. The probability of having a dependency between two successive actions was set to 40%. The probability of having a parcel of size 5 was 20%. Finally the number of constraints between any two logs was set to 10. The results for cycle-cut are given in parenthesis.

Interestingly, in those realistic cases, cycle-cut is able to match the optimal quality of global search. This can be explained if we consider that the previous problems are slightly under-constrained. When the size of the problem is rising the speed-up becomes more important. For 8 users, (i.e., 400 actions), the global branch and bound was unable to solve those instances within 1 hour time. Those experiments are quite appealing for both time performance and quality of our cycle-cut set algorithm.

Conclusion

In this work we have presented a new and efficient engine to perform log-based reconciliation. Our choice was to keep the constraint programming framework initially used by (Fages 2001) to solve those problems. The main drawback with this approach was its time complexity. Our solution performs a problem decomposition based on the constraints network topology. Its theoretical drawback is the loss of optimality. Its main advantage is time efficiency. Experimentations used two sets of problems. Random problems showed that according to time, cycle-cut out-performed global search all over the landscape. The largest speed-up is higher than 160. Regarding quality results, the loss is bounded to 7.5%. For the second set of experiments we used designers' knowledge upcoming from real reconciliation applications to define usage patterns. The results confirmed the good behaviour of cycle-cut. Moreover they showed that on those highly structured instances, cycle-cut achieved optimality combined to good execution times.

Our main conclusion is that real problems which are highly structured can greatly benefit from our work. Our next step is to extend this work toward distributed reconciliation where each user owns a reconciliation engine collaborating with remaining users to compute a new consistent state.

References

Cheeseman, P.; Kanefsky, B.; and Taylor, W. M. 1991. Where the really hard problems are. In Mylopoulos, J., and Reiter, R., eds., *Proceedings of IJCAI-91*, 331–337. Morgan Kaufmann.

Dechter, R. 1990. Enhancements schemes for constraint processing: backjumping, learning and cutset decomposition. *AI* 41(3):273–312.

Fages, F. 2001. CLP versus LS onlog-based reconciliation problems. In *6th ERCIM workshop of the Constraint Group, Prague, Czech Rep.*

Freuder, E. C. 1982. A sufficient condition for backtrack-free search. *Jrnl. A.C.M.* 29(1):24–32.

Hamadi, Y. 2003. Disolver: A Distributed Constraint Solver. Technical Report 2003-91, Microsoft Research.

Hentenryck, P. V. 1989. *Constraint Satisfaction in Logic Programming*. The MIT Press.

Karp, R. M. 1972. Reducibility among combinatorial problems. *Complexity of Computer Computations* 85–103.

Kermarrec, A.; Rowstron, A.; Shapiro, M.; and Druschel, P. 2001. The icecube approach to the reconciliation of divergent replicas. In ACM., ed., *Twentieth ACM Symposium on Principles of Distributed Computing PODC, Newport, RI USA*.

Petersen, K.; Spreitzer, M.; Terry, D.; and Theimer, M. 1996. Bayou: Replicated database services for world-wide applications. In *7th ACM SIGOPS European Workshop*.

Preguia, N.; Martins, J. L.; Cunha, M.; and Domingos, H. 2003. Reservations for conflict avoidance in a mobile database system. In *MobiSys*.

Satyanarayanan, M.; Kistler, J. J.; Kumar, P.; Okasaki, M. E.; Siegel, E. H.; and Steere, D. C. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers* 39(4):447–459.

Slaney, J. K., and Thiebaux, S. 1998. On the hardness of decision and optimisation problems. In *European Conference on Artificial Intelligence*, 244–248.