Generating Implied Boolean Constraints via Singleton Consistency

Roman Barták

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
roman.bartak@mff.cuni.cz

Abstract. Though there exist some rules of thumb for design of good models for solving constraint satisfaction problems, the modeling process still belongs more to art than to science. Moreover, as new global constraints and search techniques are being developed, the modeling process is becoming even more complicated and a lot of effort and experience is required from the user. Hence (semi-) automated tools for improving efficiency of constraint models are highly desirable. The paper presents a low-information technique for discovering implied Boolean constraints in the form of equivalences, exclusions, and dependencies for any constraint model with (some) Boolean variables. The technique is not only completely independent of the constraint model (therefore a low-information technique), but it is also easy to implement because it is based on ideas of singleton consistency. Despite its simplicity, the proposed technique proved itself to be surprisingly efficient in our experiments.

Keywords: implied constraints, reformulation, singleton consistency, SAT.

1 Introduction

Problem formulation is critical for efficient problem solving in formalisms like SAT (satisfiability testing), LP (linear programming), or CS (constraint satisfaction). LP and SAT formalisms are quite restricted, to linear inequalities in LP and logical formulas in SAT. Hence problem formulation is studied for a long time in LP and SAT because it is not always easy to express real-life constraints using linear inequalities or logical formulas. We cay say that the problem formulation is the core of courses for normal users of LP and SAT, while the solving techniques are studied primarily by experts and researchers contributing to improving the solving techniques. Opposite to SAT and LP, the CS formalism is very rich concerning its expressivity (any constraint can be directly modeled there). Hence the users get a big freedom in expressing their problems as constraint satisfaction problems which has some negative consequences. First, because the solvers need to cover the generality of the problem formulation, it is hard to improve their efficiency, and, actually, we have not observed the dramatic increase of speed of constraint solvers similar to SAT and LP solvers. Second, the main burden on efficient problem solving is on the user who must understand the details of the solving process to formulate the problem in an

efficient way. Note that sometimes a small change in the model, such as adding a single constraint, may dramatically influence efficiency of problem solving which makes the modeling task even more complicated. There exist some rules of thumb how to design efficient constraint models [10,13], but constraint modeling is still assumed to be more art than science. There exist some automated techniques for on-fly problem re-formulation such as detecting and breaking symmetries during search (for a short survey see [13]) or no-good recording (introduced in [14] and formally described in [6]). Usually the problem (re-)formulation is up to the user by using techniques such as adding symmetry breaking or implied constraints, encoding parts of the problem using specialized global constraints, or adding dominance rules.

In this paper, we address the problem of fully automated generation of useful implied constraints in constraint satisfaction problems. Informally speaking, by a useful implied constraint we mean a constraint than is deduced from the existing model (hence implied) and that positively contributes to faster problem solving (hence useful). A fully automated technique means that the implied constraints are generated for any given constraint model without any user intervention. According to the principle that the best constraint model will be the one in which information is propagated first [10] we are trying to generate implied constraints that propagate more than the existing constraints (remove more inconsistencies from the model). Recall that more inconsistencies can be easily removed from any constraint model by applying a stronger consistency technique, for example by using path consistency instead of arc consistency. However, the main problem with stronger consistency techniques is their time and space complexity which disqualifies these techniques from being used repeatedly in the nodes of the search tree. Naturally, stronger consistency techniques can be applied once before the search starts but then their effect is limited to removing initially inconsistent values from variables' domains. We propose to exploit information from these stronger consistency techniques in the form of implied constraints that are deduced during the initial consistency process and added to the constraint model. In particular, we propose to use singleton arc consistency [5] to deduce new constraints between Boolean variables in the problem. The rationale for using singleton arc consistency (SAC) is that this meta-technique is easy to implement on top of any constraint model (singleton consistency is a meta-technique because it works on top of other "plain" consistency techniques such as arc consistency or path consistency). The reasons for restricting to Boolean variables are twofold. First, singleton consistency is an expensive technique especially when applied to variables with large domains so Boolean variables seem to be a good compromise. Second, we need to specify the particular form of constraints that we are learning, which is easier for Boolean variables. To be more specific, at this stage we are learning only the equivalence, implication, and exclusion constraints. In [1] we already showed that SAC over Boolean variables contributes a lot to removing initial inconsistencies so our hope is that the constraints derived from SAC can further help in problem solving.

The paper is organized as follows. After giving the initial motivation for our work, we will define more formally the used notions and techniques. Then we will present the core of the proposed technique and the paper will be concluded by an experimental section showing the benefits and detriments of the proposed method.

For now, we can reveal that despite the simplicity of the proposed method, the experiments showed surprising speed-ups for some problems.

2 Motivation

In [2] we proposed a novel constraint model for description of temporal networks with alternative routes similar to [4]. Briefly speaking, this model consists of a directed acyclic graph or in general a Simple Temporal Network [7], where the nodes are annotated by Boolean validity variables. There are special constraints between the validity variables describing logical relations between the nodes (we call them parallel and alternative branching). These constraints specify which nodes should be selected together to form one of the possible alternative routes through the network. Figure 1 shows an example of alternative branching together with a constraint model describing the relations between the validity variables

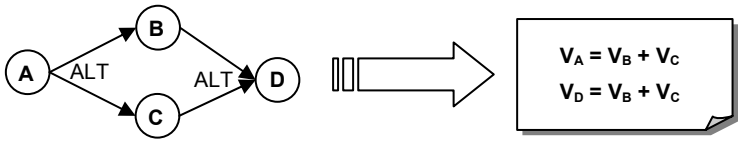

Fig. 1. A simple graph with alternative branching (left) and its formulation as a constraint satisfaction problem (left) over the validity variables.

The above model is useful for description of manufacturing scheduling problems, but it suffers from several drawbacks. The main issue is that the problem of deciding which nodes are valid in the network is NP-complete in general [2]. Hence, opposite to Simple Temporal Networks [7] we cannot expect a complete polynomial constraint propagation technique that removes all inconsistencies from the constraint model. For example, the constraint model in Figure 1 cannot discover, using standard (generalized) arc consistency, that $V_A = 1$ if V_D is set to 1 (and vice versa). In [3] we proposed some pre-processing rules that can deduce implied constraints improving the filtering power of the constraint model. In particular, we focused on discovering (some) equivalent nodes, that is, the nodes whose validity status is identical in all feasible solutions (such as nodes A and D in Figure 1). Unfortunately, we also showed there that the problem whether two nodes are equivalent is also NP-hard. Our pre-processing rules from [3] are based on contracting the graph describing the problem and it is not easy to implement them and to extend them to other problems. Moreover, this method is looking only for equivalent nodes and ignores other useful relations such as dependencies and exclusions.

The above problem is not the only problem combining Boolean and temporal variables. Fages [8] studies a constraint model for describing and solving min-cutset problems and log-based reconciliation problems. Again, there are Boolean validity variables, which can be connected by dependency constraints in case of log-based reconciliation problems, and ordering variables describing the order of the nodes in a linear sequence of nodes (to model acyclicity of the selected sub-graph). We believe

that there are many other real-life problems where Boolean variables are combined with numerical variables. Our learning method might be useful for such problems to discover implied constraints between the Boolean variables that also take in account the other constraints. Naturally, we can learn implied constraints in problems with Boolean variables only, such as SAT problems.

3 Preliminaries

A *constraint satisfaction problem* (CSP) P is a triple (X, D, C), where X is a finite set of decision variables, for each $x_i \in X$, $D_i \in D$ is a finite set of possible values for the variable x_i (the domain), and C is a finite set of constraints. A constraint is a relation over a subset of variables that restricts possible combinations of values to be assigned to the variables. Formally, a constraint is a subset of the Cartesian product of the domains of the constrained variables. We call the variable Boolean if its domain consists of two values {0, 1} (or similarly {*false*, *true*}). A *solution to a CSP* is a complete assignment of values to the variables such that the values are taken from respective domains and all the constraints are satisfied. We say that a constraint C is (generalized) *arc consistent* if for any value in the domain of any constrained variable, there exist values in the domains of the remaining constrained variables in such a way that the value tuple satisfies the constraint. This value tuple is called a support for the value. Note that the notion arc consistency is usually used for binary constraints only, while generalized arc consistency is used for n-ary constraints. For simplicity reasons we will use the term arc consistency independently of constraint's arity. The CSP is *arc consistent* (AC) if all the constraints are arc consistent and no domain is empty. To make the problem arc consistent, it is enough to remove values that have no support (in some constraint) until only values with a support (in each constraint) remain in the domains. If any domain becomes empty then the problem has no solution. We say that a value a in the domain of some variable x_i is *singleton arc consistent* if the problem $P|x_i=a$ can be made arc consistent, where $P|x_i=a$ is a CSP derived from P by reducing the domain of variable x_i to $\{a\}$. The CSP is *singleton arc consistent* (SAC) if all values in variables' domains are singleton arc consistent. Again, the problem can be made SAC by removing all SAC inconsistent values from the domains. Figure 2 shows an example of a CSP and its AC and SAC forms.

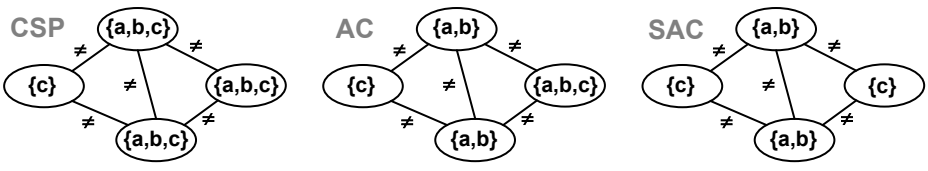

Fig. 2. A graph representation of a CSP, an arc consistent problem, and a singleton arc consistent problem (from left to right).

Assume now the constraint satisfaction problem with Boolean variables A, B, C, and D and with constraints A = B + C and D = B + C (like in Figure 1). This problem is both AC and SAC. Now assume that we assign value 1 to variable A. The problem remains AC but it is not SAC because value 0 cannot be assigned to variable D. This is an example of weak domain pruning in our temporal networks with alternatives. If we now include constraint A = D and make the extended problem AC then value 0 is removed from the domain of D by AC. Clearly, any assignment satisfying the original constraints also satisfies this added constraint. Hence we call this constraint *implied*, because the constraint is logically implied by the original constraints (sometimes, these constraints are also called *redundant*). Our goal is to find such implied constraints that contribute to stronger domain filtering.

4 Learning via SAC

As we already mentioned in the introduction and motivation, our original research goal was to easily identify some equivalent nodes in the temporal networks with alternatives. Recall, that finding all equivalent nodes is an NP-hard problem [3] so we focused only on finding equivalences similar to those presented in Figure 1 (nodes A and D). An easy way, how to identify such equivalences, is a trial-and-error method similar to shallow backtracking or SAC. Basically, we will try to assign values to pairs of variables and if we find that only identical values can be assigned to the variables then we deduce that the variables are equivalent (they must be assigned to the same value in any solution). As a side effect, we can also discover some dependencies between the variables (if 1 is assigned to B then 1 must be assigned to A) and exclusions between the variables (either B or C must be assigned to 0 or in other words it is not possible to assign 1 to both variables B and C).

We will now present the learning method for an arbitrary constraint satisfaction problem P. Recall, that we will only learn specific logical relations between the Boolean variables of P. We will gradually try to assign values to variables and each time we try the assignment, this assignment is propagated to other variables (the problem is made AC). If the assignment leads to a failure then we know that the other value in the domain must be assigned to the variable (recall that we are working with Boolean variables). The whole learning process consists of two stages.

First, we collect information about which variables are instantiated after assigning value 1 to some variable A. We distinguish between *directly instantiated variables*, that is, those variables that are instantiated by making the problem $P|_{A=1}$ arc consistent (one value in the variable domain is refuted by AC so the other value is used), and *indirectly instantiated variables*, that is, those variables where we found their value by refuting the other value in a SAC-like style (AC did not prune the domain, but when we try to assign a particular value to the variable it leads to a failure so the other value is used). Informally speaking, if we assign value 1 to variable A and make the problem arc consistent then all variables that are newly instantiated are directly instantiated variables. Indirectly instantiated variables are those variables B that are not instantiated by AC in $P|_{A=1}$ but for which only one value is compatible with A = 1 because if the other value is assigned to B, it leads to a failure after making the

problem AC (see procedure `Learn` below). More formally, let B be a non-instantiated (free) Boolean variable in $AC(P|_{A=1})$, where AC(P) is the arc consistent form of problem P (inconsistent values are removed from the domains of variables). If $P|_{A=1,B=0}$ is not arc consistent then value 0 cannot be assigned to B, hence value 1 must be used for B. Symmetrically, we can deduce that value 0 must be assigned to B if $P|_{A=1,B=1}$ is not arc consistent. Together, we can deduce which value must be used for B if value 1 is assigned to A. If both values for B are feasible then no information is deduced. If no value for B is feasible then value 1 cannot be used for A and hence A must be instantiated to 0. Note that information about indirectly instantiated variables is very important because it will help us to deduce implied constraints that improve propagation of the original constraint model. More formally, we are looking for implied constraints C such that $AC(P|_C) \subset AC(P)$, where $P|_C$ is a problem P with added constraint C and the subset relation means that all domains in $AC(P|_C)$ are subsets of relevant domains in AC(P) and at least one domain in $AC(P|_C)$ is a strict subset of the relevant domain in AC(P). In other words, constraint C helps in removing more inconsistencies from problem P.

The learning stage deduces three types of implied constraints. If B = 0 is indirectly deduced from the assignment A = 1 and A = 0 is indirectly deduced from the assignment B = 1 then the pair {A, B} forms an exclusion, which is an implied *exclusion constraint* ($A = 0 \vee B = 0$). Notice that this constraint really improves propagation because for example if 1 is assigned to A then the constraint immediately deduces B = 0, while the original set of constraints deduced no pruning for B. Similarly, if B = 1 is indirectly deduced from the assignment A = 1 then B depends on A, which is an implied *dependency constraint* ($A = 1 \Rightarrow B = 1$). Again, this constraint improves propagation. Note that we introduce this constraint only if variables A and B are not found to be equivalent. The equivalent variables are found using the following procedure. We construct a directed acyclic graph where the nodes correspond to the variables and the arcs correspond to the dependencies between the variables. These dependencies are found in the first stage, we assume both direct dependencies discovered by the AC propagation and indirect dependencies discovered by the SAC-like propagation. Strongly connected components of this graph form *equivalence classes of variables*. Note that if A and B are in a strongly connected component then ($A = 1 \Rightarrow^* B = 1$) and ($B = 1 \Rightarrow^* A = 1$), where $\Rightarrow^*$ is a transitive closure of relation $\Rightarrow$. All equivalent variables must be assigned to the same value in any solution so we can put equality constraint between these variables.

The following code of procedure `Learn` shows both the data collecting stage and the learning stage of our method. BoolVars(P) is a set of not-yet instantiated Boolean variables in P, doms(P) are domains of P, $D_X = \{V\}$ means that the domain of variable X consists of one element V, and AC(P) is the arc consistent form of problem P (AC(P) = fail if problem P cannot be made arc consistent).

The main advantage of the proposed method is simplicity and generality. Thanks to meta-nature of singleton consistency it can be implemented easily in any constraint solver and it works with any constraint satisfaction problem (even if global constraints and non-Boolean variables are included). The time complexity of the data collection stage is $O(n^2.|AC|)$, where n is the number of Boolean variables and |AC| is the complexity to make the problem arc consistent. Strongly connected components of the dependency graph can be found in time not greater than $O(n^2)$ and exclusions

and dependencies are generated in time $O(n^2)$. Clearly, majority of time to learn implied constraints by the above method is spent by collection information using the SAC-like method.

```
procedure Learn (P: CSP)
  for each A in BoolVars(P) do // data collecting stage
    Q ← AC(P|A=1)
    Direct(A) ← { X/V | DX = {V} in doms(Q)}
    for each B in BoolVars(Q) s.t. A ≠ B & Q ≠ fail do
      if AC(Q|B=0) = fail then
        Q ← AC(Q|B=1)
      else if AC(Q|B=1) = fail then
        Q ← AC(Q|B=0)
    end for
    Indirect(A) ← { X/V | DX = {V} in doms(Q)} - Direct(A)
    if Q = fail then
      P ← AC(P|A=0)
      if P = fail then stop with failure
  end for
  // learning stage
  G ← (BoolVars(P), {(A,B) | B/1 ∈ Direct(A) ∪ Indirect(A)})
  Equiv ← StronglyConnectedComponents(G)
  Excl ← { {A,B} | B/0 ∈ Indirect(A) & A/0 ∈ Indirect(B)}
  Deps ← { (A,B) | B/1 ∈ Indirect(A) & ¬ {A,B} ⊆ X ∈ Equiv}
  return (Equiv, Excl, Deps)
end Learn
```

5 Implementation and Experiments

To evaluate whether our learning technique is useful for problem solving we implemented the learning technique in SICStus Prolog 3.12.3 and tested it on 1.8 GHz Pentium 4 machine running under Windows XP. Note that we used a naïve (non-optimal) implementation of the SAC algorithm that is called SAC-1 [5]. This algorithm simply assigns a value to the variable and propagates this assignment via standard arc-consistency algorithm. The algorithm does not pass any data structures between several runs which makes it non-optimal. Nevertheless, its greatest advantage is that the implementation is very easy and can be realized in virtually any constraint solver. For the experiments we used existing benchmarks for min-cutset problems [11] and a dozen of benchmarks for SAT problems [9].

5.1 Learning for CSP

In our first experiment, we compared efficiency of the original constraint model for min-cutset problems from [8] with the same constraint model enhanced by the learned implied constraints. Note that these constraint models contain both Boolean variables

(validity) and integer variables (ordering of nodes). Recall that the min-cutset problem consists of finding the largest subset of nodes such that the sub-graph induced by these nodes does not contain a cycle. So it is an optimization problem. We used the data set from [11] with 50 activities and a variable number of precedence relations. Figure 3 shows the comparison of above models both in the runtime (milliseconds) and in the number of backtracks. It is important to say that the runtime for the enhanced model consists of the time to learn the implied constraints and the time to solve the problem to optimality (using the branch-and-bound method). The time to learn the implied constraints is negligible there (from 80 to 841 milliseconds) and hence we do not show that time separately in the graphs. We used the well-known Brélaz variable ordering heuristic also known as dom+deg heuristic (the variables with the smallest domain are instantiated first, ties broken by preferring the most constrained variables).

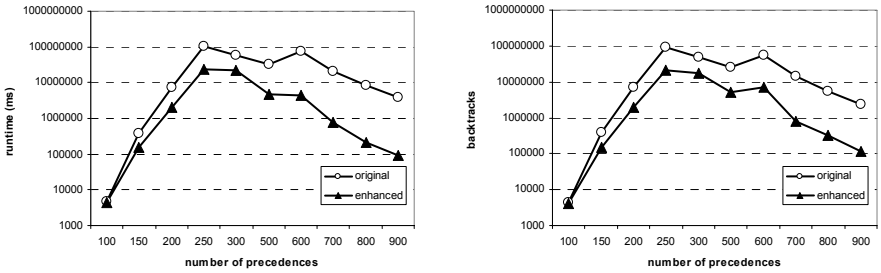

Fig. 3. Comparison of runtimes (milliseconds) and the number of backtracks for the original model of min-cutset problems and the model enhanced by the learned implied constraints with the Brélaz variable ordering heuristic.

The graphs in Figure 3 show a significant decrease of the runtime and of the number of backtracks, which is a promising result especially taking in account that the time to learn is included in the overall time. This decrease is mainly due to the learned exclusion constraints which capture cycles in the graph (one node in the exclusion must be invalid to make the graph acyclic). Clearly, the Brélaz heuristic is also influenced by adding constraints to the model so the implied constraints may change the ordering of variables during search and hence influence efficiency. As we want to see also the effect of implied constraints on pruning the search space, we need to use exactly the same search procedure for both models. The straightforward approach is to use a static variable ordering. Figure 4 shows the comparison of both models using the static variable ordering heuristic. Again, we used the branch-and-bound method to solve the problem to optimality. Due to time reasons, we used a cut-off limit 300 000 000 milliseconds (>83 hours) for a single run so the most complex problems (200 - 300, and 600 precedences) were not solved to optimality for the original model and hence information about the number of backtracks is missing.

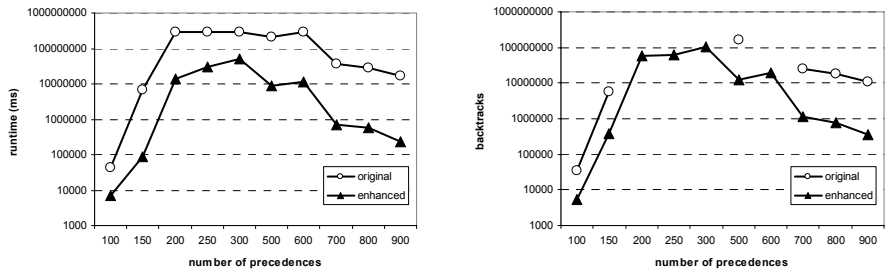

Fig. 4. Comparison of runtimes (milliseconds) and the number of backtracks for the original model of min-cutset problems and the model enhanced by the learned implied constraints with the static variable ordering heuristic (a logarithmic scale).

Again, the enhanced model beats the original model and shows a significant speedup. Moreover, by comparing both experiments, we can see that the learned constraints not only pruned more the search space by stronger domain filtering (which was our original goal) but in combination with the Brélaz heuristic they also make the search faster by focusing the search algorithm to critical (the most constrained) variables.

5.2 Learning for SAT Problems

Because our method works primarily with Boolean variables, the natural benchmark to test efficiency of the method was using SAT problems. We take several problem classes from [9], namely logistics problems from AI planning, all-interval problems, and quasigroup (Latin square) problems and encoded the problems in a straightforward way as CSPs. The choice of problem classes was driven by the idea that structured problems may lead to more and stronger implied constraints. It would be surely better to do more extensive tests with other problem classes, but a limited computation time forced us to select only few most promising classes. Again we used the Brélaz variable ordering heuristic in the search procedure which was backtracking search with maintaining arc consistency. Table 1 summarizes the results, it shows the problem size (the number of Boolean variables), the number of backtracks and the time to solve the problems (for the enhanced model the time includes both the time to learn as well as the time solve the problem), and the time for learning.

Table 1. Comparison of solving efficiency of the original and enhanced constraint models for selected SAT problems (the smallest #backtracks / runtime is in bold).

instance	size	original		enhanced		
		backtracks	runtime (ms)	backtracks	overall time (ms)	time to learn (ms)
logistics.a	828	>159827502	>60000000	**4**	**53677**	53657
logistics.b	843	>107546059	>60000000	**38494**	**91622**	65955
logistics.c	1141	>95990563	>60000000	**26195**	**165537**	150776
logistics.d	4713	>38809049	>60000000	**5738102**	**28866167**	16116604
ais6	61	16	**10**	**3**	400	390
ais8	113	**178**	**120**	523	3435	3155
ais10	181	3008	**2914**	**118**	14911	14811
ais12	265	66119	80386	**140**	**49091**	48921
qg1-07	343	26	**811**	0	146371	146291
qg1-08	512	**331474**	**12445947**	1791551	59608683	886605
qg2-07	343	34	**1061**	**0**	178987	178906
qg2-08	512	213992	8005862	213992	**7980054**	1053394
qg3-08	512	26	**170**	**22**	68018	67908
qg3-09	729	357521	2216758	**25246**	**343845**	233917
qg4-08	512	2956	**12839**	**367**	68088	66556
qg4-09	729	614	**3925**	**86**	225324	224934
qg5-09	729	1525	22573	**0**	***1933***	*1933*
qg5-10	1000	119894	2647697	**0**	***61318***	*61318*
qg5-11	1331	>1741008	>60000000	**0**	**855000**	854880
qg5-12	1728	>1195753	>60000000	**0**	***6467810***	*6467810*
qg5-13	2197	>802393	>60000000	**41641**	**23622817**	21695532
qg6-09	729	177	**2304**	**0**	51143	51113
qg6-10	1000	12234	238493	**0**	***63732***	*63732*
qg6-11	1331	1668478	34617658	**4545**	**3233200**	3153716
qg6-12	1728	>2512643	>60000000	**586472**	**22669216**	7159264
qg7-09	729	0	**40**	0	53337	53297
qg7-10	1000	348	**6930**	**0**	*46557*	*46557*
qg7-11	1331	27777	674701	**0**	***429658***	*429658*
qg7-12	1728	>2239230	>60000000	**148648**	**10344354**	6560683
qg7-13	2197	**261**	**14101**	525428	31893597	13893597

The experimental results show some interesting features of the method. First, the model enhanced by the learned implied constraints was frequently solved faster and using a smaller number of backtracks than the original model. The smaller number of backtracks is not that surprising, because the implied constraints contribute to pruning the search space. However, a shorter overall runtime for the enhanced model is a nice result, especially taking in account that the overall runtime includes the time to learn the implied constraints. The speed-up is especially interesting in the logistics problems, where the learning method deduced many exclusion constraints (probably thanks to the nature of the problem) which contributed a lot to decreasing the search space. The few examples when solving required more backtracks for the enhanced model (ais8, qg1-08, and qg7-13) can be explained by "confusing" the variable ordering heuristic by the implied constraints. Figure 3 and 4 showed that adding implied constraints influenced significantly the Brélaz variable ordering heuristic which is clear – the labeled variables have Boolean domains so the not-yet instantiated variables are ordered primarily by using the number of constraints in

which they are involved. It may happen that in some problems this may lead to a wrong decision as no heuristic is perfect for all problems. It will be interesting to study further how the added implied constraints influence structure-guided variable ordering heuristics.

A second interesting feature is that for several quasigroup problems which have no feasible solution, the learning method proved infeasibility (in italics in Table 1) so no subsequent search was necessary to solve the problem. Again in most problems it was still faster than using the original constraint model. Finally, though we almost always improved the solving time, the overhead added by the learning method (the additional time to learn) was not negligible and the total time to solve the problem was sometimes worse than using the original model. This is especially visible in simple problems, where we spent a lot of time by learning, while in the meantime the backtracking search found easily the solution in the original model. This leads to a straightforward conclusion that if the original constraint model is easy to solve, it is useless to spent time by improving the model, for example by adding the implied constraints. Of course, the open question is how to find if the model is easy to solve.

5.3 Reformulation for SAT Solvers

In the previous section, we used SAT problems to demonstrate how the proposed learning method improves the solving time for these problems. However, we modeled the SAT problems using constraints and we used constraint satisfaction techniques to solve such models (combination of backtrack search and constraint propagation), which is surely not the best way to solve SAT problems. In the era of very fast SAT solvers, it might be interesting to find out if the implied constraints, that we learned using a constraint model, can also improve efficiency of the SAT solvers. We used one of the winning solvers in the SAT-RACE 2006 competition, RSat [12], to validate our hypothesis, that the learned constraints may also improve efficiency of SAT solvers. Table 2 shows the comparison of the number of backtracks, the number of decision (choice) points, and runtime for the original SAT problem and for the SAT problem with the added implied constraints. Again, we used the problem classes from [9].

There is clear evidence that the implied constraints decrease significantly the number of choice points of the RSat solver (and in most cases also the number of backtracks). This is an interesting result, because the RSat solver is using different solving techniques than the CSP solvers, to which our learning algorithm is targeted. Nevertheless, regarding the runtime the situation is different. Thought the difference is not big, the model enhanced by the implied constraints is slower in most cases. This may be explained by the additional overhead for processing a larger number of clauses. Note that for some models, the percent of the implied constraints is 20-30% of the original number of constraints so if the solver is fast, this increase of the model size will surely influence the runtime. Still, it is interesting to see that the learned implied constraints are generally useful to prune the search space and perhaps, for more complicated problems, their detection may pay-off even if we assume time to learn these constraints (Table 1).

Table 2. Comparison of solving efficiency of the original model and the model with learned constraints solved by RSAT solver (the smallest #backtracks / #decisions / runtime is in bold).

instance	original			enhanced		
	backtracks	decisions	runtime (ms)	backtracks	decisions	runtime (ms)
logistics.a	137	1394	**40**	**31**	**176**	50
logistics.b	251	2019	**60**	**119**	**558**	90
logistics.c	238	2999	**75**	**126**	**617**	80
logistics.d	**33**	547	**130**	42	**377**	1022
ais6	14	46	5	**0**	**11**	**0**
ais8	20	74	10	**0**	**22**	10
ais10	1142	1877	90	**0**	**37**	**20**
ais12	19	152	**25**	**0**	**56**	30
qg1-07	105	134	140	**44**	**72**	**130**
qg1-08	**4732**	**5608**	**1542**	18288	20528	8142
qg2-07	**35**	54	130	37	**53**	130
qg2-08	**14017**	**16270**	**6228**	45678	52308	31320
qg3-08	122	175	**40**	122	**153**	50
qg3-09	57294	65736	26137	**38434**	**44384**	**19027**
qg4-08	638	737	**100**	**586**	**667**	110
qg4-09	8	30	60	**6**	**23**	60
qg5-11	44	78	**230**	**0**	**4**	370
qg5-13	38617	48396	**36111**	**32971**	**38733**	39046
qg6-09	0	15	**70**	0	**3**	130
qg6-12	12386	14426	**7731**	**11171**	**13230**	7761
qg7-09	1	7	**70**	**0**	**3**	130
qg7-12	4052	5042	**1862**	**3360**	**4104**	1912
qg7-13	2716	4139	1592	**1375**	**1935**	**1131**

5.4 Learning Efficiency

The critical feature of the proposed method is efficiency of learning, that is, how much time we need to learn the implied constraints. In our current implementation, this time is given be the repeated calls to the SAC algorithm so the time depends a lot on the number of involved Boolean variables and also on the complexity of propagation (the number of constraints). The following figure shows the time for learning as a function of the number of involved Boolean variables for experiments from the previous sections (plus some additional SAT problems).

Clearly, due to the complexity of SAC, the proposed method is not appropriate for problems with a large number of Boolean variables. Based on our experiments, as a rough guideline, we can say that the method is reasonably applicable to problems with less than a thousand of Boolean variables. This seems small for SAT problems, but we believe it is a reasonable number of Boolean variables in CSP problems where non-Boolean variables are also included.

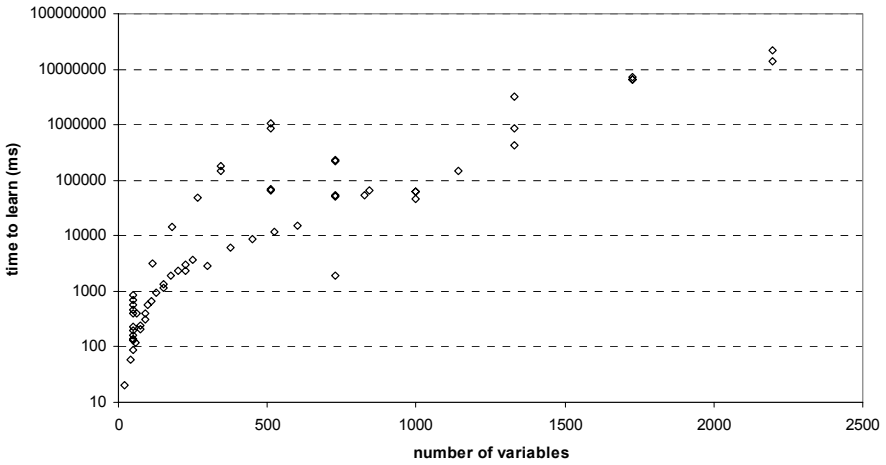

Fig. 5. Time to learn (in milliseconds) as a function of the number of involved Boolean variables (a logarithmic scale).

4 Conclusions

In the paper we proposed an easy to implement method for learning implied constraints over the Boolean variables in constraint satisfaction problems and we presented some preliminary experiments showing a surprisingly good behavior of this method. In the experiments we used naïve hand-crafted constraint models, that is, the models that a "standard" user would use to describe the problem as a CSP, so the nice speed-up is probably partly thanks to weak propagation in these models. Nevertheless, recall the holly grail of constraint processing – the user states the constraints and the solver provides a solution. For most users, it is natural to use the simplest constraint model to describe their problem and we showed that for such models, we can improve the speed of problem of solving.

To summarize the main advantages of the proposed method: it is easy to implement, it is independent of the input constraint model, and it contributes to speed-up of problem solving. The experiments also showed the significant drawback of the method – a long time to learn (an expected feature due to using SAC techniques). Clearly, the method is not appropriate for easy-to-solve problems where the time to learn is much larger than the time to solve the original constraint model. On the other hand, we did the majority of experiments with the SAT problems where all variables are Boolean, while the method is targeted to problem where only a fraction of variables is Boolean, such as the min-cutset problem. We believe that the method is appropriate to learn implied constraints for the base constraint model which is then extended by additional constraints to define a particular problem instance. So learning is done just once while solving is repeated many times. Then the time to learn is amortized by the repeated attempts to solve the problem. The time to learn can also be

decreased by identifying the pairs of variables that could be logically dependent. This may decrease the number of SAC checks. We did some preliminary experiments with the SAT problems, where we tried to check only the pairs of variables that are not "far each from other", but the results were disappointing – the system learned fewer implied constraints. Still, restricting the number of checked pairs of variables may be useful for some particular problems.

Note finally that the ideas presented in this paper for learning Boolean constraints using SAC can be extended to learning other type of constraints using other consistency techniques. However, as our experiments showed, it is necessary to find a trade-off between the time complexity and the benefit of learning.

Acknowledgments. The research is supported by the Czech Science Foundation under the contract no. 201/07/0205.

References

1. Barták, R.: A Flexible Constraint Model for Validating Plans with Durative Actions. In Planning, Scheduling and Constraint Satisfaction: From Theory to Practice. Frontiers in Artificial Intelligence and Applications, Vol. 117, IOS Press (2005), 39–48
2. Barták, R.; Čepek, O.: Temporal Networks with Alternatives: Complexity and Model. In Proceedings of the Twentieth International Florida AI Research Society Conference (FLAIRS 2007). AAAI Press (2007)
3. Barták, R.; Čepek, O.; Surynek, P.: Modelling Alternatives in Temporal Networks. In Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Scheduling (CI-Sched 2007), IEEE Press (2007), 129–136
4. Beck, J.Ch.; Fox, M.S.: Scheduling Alternative Activities. In Proceedings of the National Conference on Artificial Intelligence, AAAI Press (1999), 680–687
5. Debruyne, R.; Bessière; C.: Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann (1997), 412–417
6. Dechter, R.: Learning while searching in constraint satisfaction problems. In Proceedings of the Fifth National Conference on Artificial Intelligence. AAAI Press (1986), 178–183
7. Dechter, R.; Meiri, I. and Pearl, J.: Temporal Constraint Networks. Artificial Intelligence 49 (1991) 61–95
8. Fages, F.: CLP versus LS on log-based reconciliation problems for nomadic applications. In Proceedings of ERCIM/CompulogNet Workshop on Constraints, Praha (2001)
9. Hoos, H.H.; Stützle, T.: SATLIB: An Online Resource for Research on SAT. In SAT 2000, IOS Press (2000) 283–292. SATLIB is available online at *www.satlib.org*.
10. Mariot, K.; Stuckey, P.J.: Programming with Constraints: An Introduction. The MIT Press, (1998)
11. Pardalos, P.M.; Qian, T.; Resende, M.G.: A greedy randomized adaptive search procedure for the feedback vertex set problem. Journal of Combinatorial Optimization, 2 (1999) 399–412
12. Pipatsrisawat, T.; Darwiche, A.: RSat Solver, version 1.03. http://reasoning.cs.ucla.edu/rsat/, accesessed in March 2007.
13. Smith, B.: Modelling. A chapter in Handbook of Constraint Programming, Elsevier (2006) 377–406
14. Stallman, R.M.; Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence 9 (1997) 135–196