ABT with Clause Learning for Distributed SAT *

Jesús Giráldez-Cru, Pedro Meseguer

IIIA - CSIC, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain
{jgiraldez,pedro}@iiia.csic.es

Abstract. Transforming a planning instance into a propositional formula ϕ to be solved by a SAT solver is a common approach in AI planning. In the context of multiagent planning, this approach causes the *distributed* SAT problem: given ϕ distributed among agents –each agent knows a part of ϕ but no agent knows the whole ϕ–, check if ϕ is SAT or UNSAT by message passing. On the other hand, Asynchronous Backtracking (ABT) is a complete distributed constraint satisfaction algorithm, so it can be directly used to solve distributed SAT. Clause learning is a technique, commonly used in centralized SAT solvers, that can be applied to enhance ABT efficiency when used for distributed SAT. We prove that ABT with clause learning remains correct and complete. Experiments on several planning benchmarks show very substantial benefits for ABT with clause learning.

1 Introduction

Problem solving is often assumed as a *centralized* activity: the instance to solve is contained into a single agent, that has direct access to every detail of the instance to perform the solving process. However, in *distributed* problem solving the instance is distributed among several agents; each agent knows a part of the instance but no agent knows the whole instance. *Privacy* is a main motivation for distributed problem solving. When several agents collaborate for solving a problem, it may occur that some could see others as potential competitors. In this case, it is of the greatest importance to assure that the solving process is done without revealing more information than the strictly needed. This is essential for real-world applications, where companies by no means want to disclose sensitive information, of great interest for their business purposes.[1]

In the planning context, a common approach for classical planners when solving an instance is (i) translating the instance into a propositional formula which is SAT (satisfiable) iff the instance has a solution, (ii) solving the formula by an "off-the-shelf" SAT solver, and (iii) retranslating the solution into planning terms. In multiagent planning (MAP) [14, 3, 11], where privacy matters, this approach generates the *distributed* SAT problem: a propositional formula is distributed among several agents, each contains a part of the formula but none knows the whole formula.[2] Intense communication allows to synthesize a solution. This problem has been considered before [12, 15].

* Partially funded by TIN2013-45732-C4-4-P and TIN2015-71799-C2-1-P.

[1] Privacy is not required in all distributed scenarios. But when present, it causes a major concern.

[2] This approach differs from an existing meaning in the SAT community, where "distributed" usually means "parallel", and the main goal is finding efficiency gains with respect to centralized SAT.

ABT [18] –that stands for *asynchronous backtracking*– is a distributed algorithm that originally was presented for distributed CSP. Since SAT is a special case of CSP, the ABT algorithm can be used to solve distributed SAT. It is a correct and complete algorithm and offers a reasonable level of privacy, so ABT appears as a suitable candidate to solve MAP instances. We acknowledge the combination of distributed CSP algorithms with other solving techniques [14, 8, 19] in the MAP context. Using exclusively distributed constraint satisfaction algorithms has been explored [7]. In this paper, we use ABT as the only algorithm for MAP, assuming that each agent handles a single variable. The generalization to multiple variables per agent is later discussed.

The main contribution of this paper is to import clause learning –a very successful technique to solve industrial SAT instances in the centralized case– into ABT to solve distributed SAT. This is not trivial: one has to decide which clause to learn and who is the learning agent, for each learning episode. Our approach considers that each time an agent receives a backtracking it learns a new clause. This does not cause new messages with respect to the original algorithm. We prove that this new version of ABT remains correct and complete. In practice, it shows a much better performance than original ABT on several planning benchmarks. An instance with more than eight hundred variables have been solved, which is a novelty in the performance of distributed algorithms, often solving instances of more modest size (original ABT could not solve that instance in a timeout of 10 hours).

2 Background

2.1 Definitions and Notation

A *centralized* CSP is defined by a tuple (X, D, C), where $X = \{x_1, x_2, ..., x_n\}$ is a set of n variables taking values in a collection of finite and discrete domains $D = \{D_1, D_2, ..., D_n\}$, such that x_i takes value in D_i, under a set of constraints C. A constraint indicates the combinations of permitted values in a subset of variables. A solution is an assignment of values to variables that satisfies all constraints. The goal is to find a solution or to prove that it does not exist. On the SAT problem, we recall the following concepts from propositional logic: a *literal* l is a variable x or its negation $\neg x$; a *clause* is a disjunction of literals; a *formula* ϕ in conjunctive normal form (CNF) [3] is a conjuntion of clauses. A *centralized* SAT instance is defined by a formula ϕ in CNF, where variables may take the values *true* or *false*. The goal is determining if there exists an assignment that evaluates ϕ as *true*. Notice that to satisfy ϕ, each clause must be satisfied, so at least one literal in each clause must be *true*. The *resolution* between clauses $A \vee x$ and $B \vee \neg x$ results into the clause $A \vee B$, where A and B are disjunctions of literals.

A *distributed* CSP is defined by (X, D, C, A, α) where (X, D, C) are as in the centralized case, A is a set of agents and α is a mapping that associates each variable with an agent. For simplicity, we assume that no agent controls more than one variable. Each agent knows all constraints in which its variable is involved. It is not possible to join all the information into a single agent. The solution is found by message passing. A *distributed* SAT instance is defined by a tuple (ϕ, A, α), where ϕ is as in centralized

[3] Any propositional formula can be translated into CNF in linear time.

SAT, and A and α as in distributed CSP. Each agent knows the clauses where its variable appears.

Defined in a centralized context, a *nogood* is an assignment (a conjunction of variable-value pairs) that cannot be extended consistently into a solution [13]. A nogood ng is a justification to remove the value of the deepest variable in the search tree mentioned in ng. When all the values of such variable are removed by nogoods, one can perform *resolution* among them to produce a *new nogood* [2, 13]. For example, let us assume that $D_y = \{a, b\}$ and both values are removed by the following nogoods,

$$(x_1 = v_1) \wedge (x_2 = v_2) \wedge (y = a)$$
$$(x_2 = v_2) \wedge (x_3 = v_3) \wedge (y = b)$$

If y is deeper in the search tree than x_1, x_2 and x_3, one can resolve them to obtain the new nogood,

$$(x_1 = v_1) \wedge (x_2 = v_2) \wedge (x_3 = v_3)$$

2.2 Centralized SAT Solving

Most of the modern (complete) SAT solvers are based on the DPLL procedure [10]. It is a depth-first search algorithm; its core idea is branching in each variable (*decisions*), assigning them a value, until all clauses are satisfied (then the formula is SAT), or until a conflict is found and it backtracks to a new assignment. A formula is UNSAT if a conflict is found for all assignments. It also includes the Unit Propagation (UP) rule. This rule is triggered when a certain clause has all its literals but one assigned and the clause is not satisfied. It forces this unassigned literal the value that satisfies such clause. This may occur after every assignment (by decisions or by other propagations).

The Conflict-Driven Clause-Learning (CDCL) SAT solvers are inspired in the DPLL algorithm, but they also include a wide variety of techniques [4]. One of them are the clause learning mechanisms [16], that summarize in new clauses the conflicts that were found in the past, in order to avoid them in the future. Empirically, it has been shown as a key technique to solve real-world SAT instances.

A *conflict* occurs when all literals of a clause are assigned but the clause is still unsatisfied. Hence, that (partial) assignment cannot satisfy the formula, and a new clause can be learnt to avoid the same conflict in the future. The new clause is found by analyzing the implication graph, i.e., the graph that represents the decisions and propagations that provoked the conflict. See an example in Figure 1. A cut in the implication graph can be seen as a conjunction of the links it cuts. Any cut in the implication graph leaving the conflict in one side and all the decisions in the other side is an inconsistent assignment; its negation produces the new clause to learn. From a conflict, many clauses can be learnt. Experimentally, good performance has been found learning the 1-UIP clause [4] (see Section 3.2).

As a toy example, let us consider this formula with 3 clauses (c_i stands for the i-th clause): $\phi = (\neg x_2) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$. Starting with the value $false$, $x_2 = false$ (by UP); decision $x_1 = false$ causes propagations $x_3 = true$ (by c_2), and $x_3 = false$ (by c_3). So there is a conflict; the implication graph finds that $x_1 = false$ is inconsistent, the clause to learn is simply (x_1). At this point, $x_1 = true$ and $x_2 = false$ (both by UP) satisfy the set of clauses, for any value of x_3. This is a solution for ϕ.

2.3 ABT

Asynchronous Backtracking (ABT) [18] was the pioneer asynchronous algorithm to solve distributed CSP. ABT is a distributed algorithm that is executed autonomously in each agent, which takes its own decisions and informs of them to other agents; no agent has to wait for decisions of others. When solving a problem instance, there are as many ABT executions as agents. A telegraphic description follows (for details, consult [18]).

ABT computes a global consistent solution or detects that no solution exists in finite time; it is correct and complete. ABT requires agents to be totally ordered. A binary constraint is translated into a directed link, from the higher to the lower agent that it connects.[4] An ABT agent keeps its agent view and nogood store. The agent view is the set of values it believes are assigned to higher agents connected with it. The nogood store keeps the nogoods received as justifications of inconsistent values. Agents exchange individual assignments and nogoods. When an agent cannot find any value consistent with its agent view, because of the original constraints or because of the received nogoods, a new nogood is generated from its agent view, and it is sent to the closest agent in the new nogood, causing backtracking. If an agent receives a nogood including another agent not connected with it, the receiver requires to add a link from that agent to itself. From this point on, a link from the other agent to itself will exist, receiving the values taken by that agent. ABT uses these messages:

1. $OK?(agent, value)$. It informs *agent* that the sender has taken *value* as value.
2. $NGD(agent, ng)$. It informs *agent* that the sender considers *ng* as a nogood.
3. $ADL(agent)$. It asks *agent* to set a direct link to the sender.

3 ABT Enhanced with Clause Learning

ABT can solve distributed SAT. Clause learning, developed for centralized SAT, can also be applied to ABT for solving distributed SAT, causing very substantial gains.

3.1 Clause Learning

When ABT solves a distributed SAT instance, let us consider a Boolean variable x of ϕ with a conflict, as depicted in Figure 1. A conflict appears when both possible values of x are forbidden, as effect of the assignments of previous variables in the total order: *false* is forbidden by $assig_1$, *true* is forbidden by $assig_2$. We say that,

$$assig_1 \wedge (x = false)$$
$$assig_2 \wedge (x = true)$$

are nogoods removing the values of x. In SAT terms $assig_1$ corresponds to the links pointing to $x = true$ in the implication graph, while $assig_2$ corresponds to the links pointing to $x = false$. Both assignments cause the conflict; their conjunction $assig_1 \wedge assig_2$ is the new nogood found by x. It corresponds to the cut that has in its right

[4] ABT can also deal with non-binary constraints; it is described in [5].

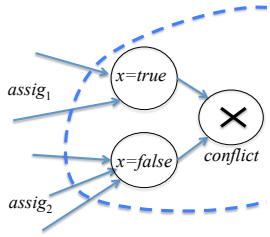

Fig. 1. The dotted line does the cut in the conflict graph.

side the conflict variable only. It is easy to see that the negation of this conjunction $\neg(assig_1 \wedge assig_2)$ is a logical consequence of the original formula[5], so we can add the negation of this conjunction –that from now on we call *new learnt clause*– to the formula ϕ without altering its satisfiability.

Which agent has to learn this new clause? Let y be the agent to which x backtracks to, after discovering the conflict. We know that y is the closest agent to x in the new clause. In addition, y is connected (or it will be connected, after the reception of the *NGD* message by the use of *ADL* messages) with all the other agents in the new clause. So y is the right agent to store and evaluate the new clause: it is the last in the ordering among the new clause agents and it has direct connections with all of them. It is enough to store the new clause in a single agent: the ABT termination condition [18] cannot be achieved if there is at least one unsatisfied clause in an agent.

There is a drawback: if a new clause is added after each backtracking, memory may grow exponentially. This drawback also exists in centralized SAT solving. To avoid the extra overhead caused by keeping an increasing number of clauses in large formulas, some clauses deletion policies have been proposed [4, 16, 1]. However, in our experimentation we detected no memory overhead (each SAT instance was solved using a maximum of 4GB of RAM). For this reason, we did not implement any clause deletion policy in our algorithm. The applicability of these policies to the proposed solution remains for future work.

In CDCL SAT solvers, clause learning is usually used jointly with non-chronological backtracking (the backjump destination and the learnt clause are related). In the distributed case, things are different because the agent that finds the conflict does not see the whole formula, only a subset of clauses. Then, it cannot do a complete conflict analysis to determine where to backjump. Since ABT agents have a limited view of

[5] If the original formula is satisfiable, variable x in the satisfying assignment will take some value, either *true* or *false*. But that assignment necessarily has to satisfy $\neg assig_1 \vee \neg assig_2$, otherwise x will have no value. So $\neg assig_1 \vee \neg assig_2 = \neg(assig_1 \wedge assig_2)$ can be legally added to the formula without changing its satisfiability. If the original formula is unsatisfiable, any other clause can be added to it, because the resulting formula will remain unsatisfiable.

the whole problem, the agent that finds a conflict backtracks to the closest agent in the nogood obtained from that conflict, following the backtracking policy of original ABT.[6]

Adding clause learning to ABT maintains its correctness and completeness, as we prove in the following theorem.

Theorem 1. *ABT enhanced with clause learning remains correct and complete.*

Proof. ABT is correct and complete [18]. ABT with clause learning on ϕ finds:

1. A satisfying assignment, which is a correct solution for ϕ' (the set of clauses in memory when the solution was found). This is also a solution for ϕ since $\phi \subseteq \phi'$;
2. There is no satisfying assignment (the two values of a variable have been unconditionally removed). Since all added clauses are logical consequences of ϕ, ABT on ϕ' would not remove any value that would not been removed by ABT on ϕ.

On completeness, the same argument (2) applies: the added clauses are logical consequences of ϕ, so they will never remove any value that would not have been finally removed by ϕ. So ABT with clause learning is correct and complete. □

In summary, we propose a new version of ABT that performs clause learning. Each time a *NGD* message reaches an agent, it learns the clause that is the negation of the nogood contained in that message. These learnt clauses can be seen as new constraints that summarize the conflicts found during the search. Each conflict is found after several *wrong* decisions, resulting in an inconsistency. Therefore, learning the reasons of a conflict allows us to detect it in the future in earlier stages, i.e., reducing the number of wrong decisions that lead to the same conflict. It is worth noting that this does not increase the number of messages used by normal ABT. To the best of our knowledge, it is first time that clause learning occurs in the distributed context. This novel approach keeps correctness and completeness of original ABT.

3.2 Learning 1-UIP Clause

Which is the right clause to learn? In centralized SAT solving, a 1-UIP clause seems to be the best practical choice. This clause is related to the decision level of each variable involved in the implication graph. A decision level contains a decision variable and all variables propagated by it (forced by UP), and it is increased in each new decision. Formally, a *1-UIP clause* is the first cut in the implication graph (from the conflict to the decision variables) that only contains one literal of the last decision level (i.e., the decision level of the conflict). Notice that the implication graph may contain several 1-UIP clauses. In ABT, the first learnt clause is not necessarily the 1-UIP. However, we show that in each conflict a 1-UIP clause is learnt by some agent.

Theorem 2. *For a single conflict, ABT with clause learning for distributed SAT learns exactly all possible clauses that can be derived from the implication graph of that conflict, if the total order of the variables in ABT is the same as in the implication graph.*

[6] In the toy example of 2.2 with lexicographic variable ordering, ABT executed on x_3 detects the conflict but it knows c_2 and c_3 only (the clauses where x_3 appears). It finds the nogood $\neg x_1 \wedge \neg x_2$. Then, x_3 backtracks to the deepest variable in the nogood, that is x_2.

Proof. Each time a CDCL SAT solver finds a conflict, there exists in the last decision level at least one variable whose value was forced by UP, and (at least) an unsatisfied clause with no unassigned literals. This is the conflict clause. Notice that the implication graph of this conflict defines a total order among the variables involved. Applying resolution between this conflict clause and the clause that forced (by UP) the last variable in the ordering, we obtain a new clause (which is a logical consequence from the formula, and thus it can be added to the formula without altering its satisfiability). Using this resulting clause, this step can be repeated as many times as variables were assigned by UP, obtaining a new learnt clause at each step. The last possible learnt clause contains the decision variable of the last decision level.

Let us assume now an ABT algorithm whose agents order is the same as the one in the implication graph of a certain conflict. When an agent (variable) finds a conflict, there exists a pair of clauses that cannot be satisfied under its current agent view. The generated nogood ng is the resolvent between these two clauses, which is exactly the first possible learnt clause in the implication graph, and it is learnt by the highest priority agent in ng. If this agent has one of its values forbidden by another clause ω (it corresponds to a variable assigned by UP in a CDCL), it will apply resolution between ng and ω, and will send the resolvent to another agent, which will learnt this new clause. Hence, these clauses are exactly the cuts in the implication graph. This process will be repeated till the agent which receives the nogood has no forbidden values (it corresponds to the decision variable of the last decision level in a CDCL), and this is exactly the last clause that can be learnt by a CDCL. □

Assuming that the total order used by ABT is the same the the total order in the implication graph is a strong assumption, and reduces the effect of UP in ABT with respect to CDCL SAT solvers. However, this restriction is imposed in the original ABT.

Corollary 1. *ABT for distributed SAT with clause learning learns a 1-UIP clause.*

Proof. One of the derived clauses from a conflict is a 1-UIP clause. As ABT learns all possible clauses from a conflict (Th.2), one of them is precisely a 1-UIP clause. □

Therefore, after a conflict this approach assures that some agent has learnt a 1-UIP clause, although we do not know which agent has done it.

3.3 Example

Let us consider the following SAT formula ϕ:

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_4) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_3 \vee x_4 \vee \neg x_5)$$

Let us summarize the performance of ABT with clause learning in ϕ, considering an initial lexicographical ordering of its variables; c_i stands for the i-th clause of ϕ. We also consider that it first assigns the value $false$ to a variable if there is no nogood forbiding that value. We can deduce the following facts: [7]

[7] For simplicity, we do not give the trace of ABT, which is quite long.

1. Decision $x_1 = \mathit{false}$ causes the propagations $x_2 = \mathit{false}$ (by c_1), which in turn causes the propagations $x_3 = \mathit{false}$ (by c_2) and $x_4 = \mathit{false}$ (by c_3). There is a conflict in x_5, which triggers a cascade of backtrackings (from x_5 to x_4, from x_4 to x_3, from x_3 to x_2, from x_2 to x_1). In these backtrackings, the algorithm learns the following clauses: $c_{l_1} = (x_3 \vee x_4)$, $c_{l_2} = (x_2 \vee x_3)$, $c_{l_3} = (x_2)$, $c_{l_4} = (x_1)$.
2. Clause c_{l_4} is unit so $x_1 = \mathit{true}$ (by UP); the same occurs with $x_2 = \mathit{true}$. Decision $x_3 = \mathit{false}$ causes the propagation $x_4 = \mathit{true}$ (by c_{l_1}). This satisfies clauses c_4 and c_5, with the decision $x_5 = \mathit{false}$. The original clauses are satisfied, this assignment is a solution for the formula.

Observe that learnt clauses help to prune the search tree, avoiding traversing zones that do not contain any solution. Clause $c_{l_3} = (x_2)$ avoids exploring $x_2 = \mathit{false}$ which does not drive to any solution. After the propagation $x_2 = \mathit{true}$ and the decision $x_3 = \mathit{false}$, clause $c_{l_1} = (x_3 \vee x_4)$ forces $x_4 = \mathit{true}$, avoiding $x_4 = \mathit{false}$ which does not drive to any solution. Clauses c_{l_1} and c_{l_3} have been learnt under $x_1 = \mathit{false}$, and they are used when exploring $x_1 = \mathit{true}$. Without clause learning, ABT would have to traverse a larger search tree. In addition to visiting more nodes, more messages were exchanged among the agents, messages that do not lead to any solution. It is worth noting that the nogood $(\neg x_3 \wedge \neg x_4)$ (that corresponds to the learnt clause c_{l_1}) was recorded as a justification of the removal of value false for x_4. Original ABT would have removed that nogood after backtracking to x_3.

4 Experimental Results

We evaluate the performance of ABT with clause learning (=ABT_{CL}) against plain ABT, in terms of communication cost (total number of messages exchanged among agents) and computation effort (equivalent non-concurrent constraint checks, ENCCCs). Upon receipt of a message msg from another agent, the receiving agent ag updates its $ENCCC$ counter as: $ENCCC_{ag} = max\{ENCCC_{ag}, ENCCC_{msg} + 1000\}$.[8] In a distributed scenario, exchanging messages among agents has a much higher cost than any other operation performed by an agent without communication.

We have used some planning benchmarks from the SAT Competition 2005 and from the SATLIB repository.[9] Moreover, we have generated a set of benchmark composed of 100 classical random 3-CNF formulas with 50 variables and 200 clauses. In Table 1, we present the results, on average, for each benchmark with lexicographic variable ordering. Each version of ABT was run in our simulator with a timeout of 10 hours (to allow plenty of time for the execution of original ABT) with a limit of 4GB of RAM memory per instance. Remark that we are only reporting the results for the instances that were solved by both ABT versions (original ABT and ABT with clause learning) discarding those that could not be solved by any of them in the established timeout of 10 hours.[10] On all the planning instances, the benefits of ABT with clause learning

[8] Exchanging a message has a cost of 1000 *ENCCC*. We choose such *arbitrary* value to emphasize that sending a message is much more costly than performing internal CPU operations.

[9] `http://www.satcompetition.org/` and `http://www.satlib.org/`.

[10] Except in the Ferry benchmark, where we report one instance unsolved by ABT in the timeout.

		#solved		#messages		*ENCCC*	
Benchmark	#inst	ABT	ABT_{CL}	ABT	ABT_{CL}	ABT	ABT_{CL}
Depots	8	**5**	**5**	120101314.60	**12953075.60**	98746867.40	**1578372.60**
DriverLog	20	**11**	**11**	56698280.45	**19969830.64**	49797937.36	**3761396.55**
Ferry	18	0	**1**	-	**625177269.00**	-	**110158150.00**
Rovers	11	**9**	**9**	21674815.78	**4720008.11**	33090165.44	**1724779.33**
Satellite	10	**5**	**5**	329448853.20	**103446296.00**	580921823.00	**10399030.00**
Blocksworld	7	**5**	**5**	24245647.40	**16771146.20**	16041447.80	**2328836.80**
Logistic	4	1	**2**	236392659.00	**7370661.00**	670032346.00	**5043248.00**
random	100	**100**	**100**	487734.01	**335262.44**	3692305.51	**1022613.69**

Table 1. Results as the number solved instances, number of messages exchanged and ENCCCs, on average per each benchmark, solved by both algorithms in the timeout.

are clear, in both #messages and ENCCCs (arriving to savings of orders of magnitude in some cases). It is worth mentioning that ABT with clause learning has solved an instance with more than one eight hundred variables (843 variables and 7301 clauses, found as `logistics/logistics.b.cnf`), while original ABT could not solve it in the timeout (because of that, it is not recorded in Table 1). To the best of our knowledge, it is first time that a distributed algorithm solves an instance of such size. Since all planning instances reported in Table 1 are satisfiable, we also experimented with unsatisfiable formulas, using classical random 3-SAT instances. We experimented with 100 random instances (51 satisfiable, 49 unsatisfiable) of 50 variables and 200 clauses. Results for this class also indicate that ABT enhanced with clause learning performs clearly better than original ABT.

5 Discussion and Conclusions

We have focused on ABT, while other efficient algorithms exist for distributed constraint solving. Why? We consider that clause learning is rather independent to the techniques used by existing algorithms, so it is expectable that, in the case that these algorithms were combined with clause learning, they would also increase their efficiency. Here we are using ABT as baseline, in order to show the benefits that clause learning may cause when included in a distributed constraint algorithm, although we believe that results of the same kind could be observed when clause learning is combined with other algorithms. A similar reasoning applies to heuristics.

We assumed the simplifying assumption of one variable per agent. Under this assumption we have shown how clause learning produces an important improvement in the communication cost among ABT agents. We are aware that the natural translation of a multiagent planning instance into a distributed propositional formula may assign several Boolean variables to the same agent. There are two classical reformulations, compilation and decomposition, that allows to comply with this assumption. We skip details because space limitations, the interested reader is addressed to [17, 6, 9]. However, these reformulations imply some drawbacks. As future work, we plan to extend this approach for agents with several variables without using any reformulation.

To conclude, we have presented ABT enhanced with clause learning, a new version of ABT for solving distributed SAT. We stress the inclusion of the powerful technique of clause learning. To the best of our knowledge, it is first time that clause learning is combined with a distributed algorithm. Interestingly, ABT with clause learning maintains the correctness and completeness of the original ABT. We have proved that a 1-UIP clause, the one most preferred in the centralized SAT, is learnt by some agent after a conflict. Experimentally, we observe that clause learning causes a substantial improvement in performance, with respect to the original algorithm when tested on planning benchmarks. ABT with clause learning can be useful for multiagent planning, and for other domains (as scheduling) where problems have to be solved distributedly.

References

1. G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proc. IJCAI'09*, pages 399–404, 2009.
2. A. Baker. The hazards of fancy backtracking. *Proc. AAAI'94*. 288–293, 1994.
3. M. Benedetti and L. C. Aiello. SAT-based cooperative planning: a proposal. In *Mechanizing Mathematical Reasoning*. 2005.
4. A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.
5. I. Brito and P. Meseguer. Asynchronous backtracking for non-binary disCSP. In *ECAI-06 Workshop on Distributed Constraint Satisfaction*, 2006.
6. D. A. Burke and K. N. Brown. Efficient handling of complex local problems in distributed constraint optimization. *Proc. ECAI'06*. 701–702, 2006.
7. P. Castejon, P. Meseguer, and E. Onaindia. Multi-agent planning by distributed constraint satisfaction. *Proc. CAEPIA'15*. 41–50, 2015.
8. K. Dakota and A. Komenda. Deterministic multi agent planning techniques: experimental comparison. *Proc DMAP (ICAPS workshop)*. 43–47, 2013.
9. J. Davin and P. J. Modi. Hierarchical variable ordering for multiagent agreement problems. *Proc. AAMAS'06*. 1433–1435, 2006.
10. M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
11. Y. Dimopoulos, M. A. Hashmi, and P. Moraitis. μ-SATPLAN: Multi-agent planning as satisfiability. *Knowledge-Based Systems*, 29:54–62, 2012.
12. K. Hirayama and M. Yokoo. Local search for distributed SAT with complex local problems. *Proc. AAMAS'02*. 1199–1206, 2002.
13. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. *Proc. CP'03*. 873–877, 2003.
14. R. Nissim, R. Brafman, and C. Domshlak. A general, fully distributed multi-agent planning algorithm. *Proc. AAMAS'10*. 1323–1330, 2010.
15. E. Ruiz. Distributed SAT. *Artificial Intelligence Review*, 35:265–285, 2011.
16. J. M. Silva and K. Sakallah. GRASP - a new satisfiability algorithm. *Proc. ICCAD*. 220-227, 1996.
17. M. Yokoo. *Distributed constraint satisfaction*. Springer–Verlag, 2001.
18. M. Yokoo, E. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. Know. and Data Engin.*, pages 673–685, 1998.
19. Y. Zhang and S. Kambhampati. A formal analysis of required cooperation in multi-agent planning. *Proc. DMAP'14 (ICAPS workshop)*. 30–37, 2014.