Connecting ABT with a SAT Solver

Jesús Giráldez-Crú [a], Guillermo Martín-Sánchez [b], and Pedro Meseguer [a]
[a] *IIIA - CSIC, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain*
[b] *Universidad Complutense de Madrid, Av. Séneca 2, 28040 Madrid, Spain*
guillemartinsan@gmail.com {jgiraldez,pedro}@iiia.csic.es

Abstract. Many real-world problems are nowadays encoded into SAT instances and efficiently solved by CDCL SAT solvers. However, some scenarios require distributed problem solving approaches. Privacy is often the main reason. This causes the distributed SAT problem. In this work, we analyze how this problem can be tacked in an efficient way, and present ABT_{SAT}, a new version of the ABT algorithm adapted to solve distributed SAT instances. It combines ABT execution with calls to CDCL SAT solvers and clause learning. ABT_{SAT} is sound and complete, properties inherited from ABT, and solves local problems efficiently by using CDCL SAT solvers.

Keywords. distributed SAT, clause learning, ABT

1. Introduction

The Boolean Satisfiability problem (SAT) is one of the most popular problems in Computer Science, from both theoretical and practical sides. As the first known NP-complete problem, it has been considered *intractable* during decades. However, recent advances in SAT solving have allowed that many SAT instances can be solved *efficiently* in practice. This is the case of many real-world applications, e.g., planning. They are translated into SAT instances and solved by efficient SAT solvers. Specifically, these solvers are known as CDCL SAT solvers, whose main component is the learning of new clauses derived from the conflicts found during the search. See [6] for a survey on CDCL SAT solvers.

The previous scenario is a *centralized* approach, i.e., the whole instance to solve is known by a single agent. However, real-world applications may require a *distributed* approach, where the problem instance is distributed among a set of agents but no agent knows the whole instance. *Privacy* is the main requirement for distributed problem solving: agents collaborate to find a solution but they do not want to reveal its sensitive information. Then, it is crucial to assure that the information exchanged during the solving process is just the strictly needed. This is the case of multi-agent planning (MAP) [5,2,3], where privacy matters. Hence, centralized approaches are not appropriate.

Distributed constraint satisfaction problems (distributed CSP) provide a suitable framework to model problems that require a distributed solving process. Distributed CSP can be translated into SAT, causing the distributed SAT problem. Asynchronous Backtracking (ABT) [7] was the pioneer algorithm to solve distributed CSP. Since SAT can be seen particular case of CSP, ABT can also solve distributed SAT. ABT is sound and complete, and it offers a reasonable level of privacy, so it seems to be a suitable candidate to participate in the resolution of MAP instances. On the other hand, CDCL SAT solvers appears to be a powerful tool to solve centralized problems. Notice that a MAP instance can be seen as composed of two coupled elements: (i) solving each local subproblem (i.e., the internal problem of each agent), where centralized methods have been shown as the most efficient choice, and (ii) the process of assuring coherence among the solutions of the local subproblems, that is mostly a communication issue between agents.

In this paper, we analyze how these two methods can be combined to solve distributed SAT. We present $\mathrm{ABT_{SAT}}$, a new version of ABT which is interfaced with a CDCL SAT solver, combining the key elements of both techniques. Each agent uses CDCL techniques to find partial solutions to its subproblem, or getting unsatisfiability cores when unsolvable. Communication between agents is performed following the ABT algorithm, which is sound and complete. In addition, ABT is enhanced with clause learning, and activity-based policies for its removal, improving its original performance.

2. Background

A CSP is defined by the tuple (X,D,C), where $x_i \in X$ is a variable that takes value in a finite and discrete $D_i \in D$, and C is a set of constraints over the variables X. The goal is to find an assignment of values to all variables satisfying all constraints, or prove that it does not exist. SAT can be seen as a particular case of CSP, in which X is a set of Boolean variables, i.e., their domains only have two possible values. A *literal* is either a Boolean variable x or its negation $\neg x$, and a *clause* is a disjunction of literals. A SAT instance ϕ is written in Conjunctive Normal Form (CNF) if it is a conjunction of clauses (it is well-known that every propositional formula can be expressed in CNF). To satisfy a formula ϕ in CNF, every clause must be satisfied. This requires that, at each clause, at least one literal must be assigned to $true$. Notice that, under this formulation, each clause can be seen as a different constraint. A *distributed CSP* is defined by the tuple (X,D,C,A,α), where (X,D,C) are as in CSP, A is a set of agents, and $\alpha : X \to A$ a mapping function of each variable to an agent. A *distributed SAT* instance is a tuple (ϕ,A,α), where ϕ is a propositional formula in CNF, and A and α are as in distributed CSP.

A *nogood* is an assignment (a conjunction of variable-value pairs) that cannot be extended consistently into a solution [4]; it has been found inconsistent either by the original constraints/clauses of the problem or by the search process. In the context of tree search, a nogood justifies the (temporary) removal of a value in the domain of a certain variable. This variable is the deepest variable in the search tree mentioned in the nogood. When all values of a variable are removed by nogoods, a new nogood can be produced by resolution [1,4]. For example, let us assume that $D_y = \{a,b\}$, and both values are forbidden by nogoods ng_1 and ng_2:

$$\left.\begin{array}{l} ng_1 : (x_1 = v_1) \wedge (y = a) \\ ng_2 : (x_2 = v_2) \wedge (y = b) \end{array}\right\} \Rightarrow ng_{new} : (x_1 = v_1) \wedge (x_2 = v2)$$

This is clear since the assignment $x_1 = v_1 \wedge x_2 = v2$ forbids all values of variable y, and hence no solution exists under this assignment. Therefore, the negation of this nogood must be satisfied by any solution. Often, this negation $\neg(x_1 = v_1 \wedge x_2 = v_2)$ is written as an implication $x_1 = v_1 \Rightarrow x_2 \neq v_2$, where left-hand side of $\Rightarrow$ is abbreviated as `lhs` and the right-hand side is abbreviated as `rhs`. Without loss of generality, hereinafter we only consider Boolean variables, and its notation using literals.

2.1. CDCL SAT solvers

CDCL SAT solvers are inspired in the DPLL procedure (i.e., a depth-first search), but they incorporate a number of new techniques. One of the most important is *clause learning*, which summarizes in new clauses the conflicts found in the past, in order to avoid them in the future. Empirically, it has been shown as a key element to solve real-world SAT instances, as planning instances are. During the search, a *conflict* occurs when all the literals of a certain clause took values, and the clause is not satisfied by any of them. The current (partial) assignment is, hence, a nogood. Applying conflict analysis techniques, this learnt clause is reduced, keeping only those literals that derived the conflict.

2.2. ABT

ABT is the pioneer asynchronous algorithm to solve distributed CSP. ABT 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. ABT is sound and complete, and finds a solution in finite time. ABT requires a total order among agents.

ABT can solve distributed SAT. In that case, each agent knows the total order among agents, the set of variables owned by it (*local variables*) and the set of clauses where its local variables appear. Initially, two agents are connected if their variables share a clause. More agents can be connected during ABT execution (see below). An agent must evaluate the subset of clauses where it is the lowest agent in the clause, according to the total order. So it must know the values of the variables in the clause owned by other agents. Each agent keeps an *agent view*, the set of values it believes are assigned to variables of higher agents connected with it, and a *nogood store*, a set of received nogoods during execution. Agents exchange assignments and nogoods. When an agent takes a value, informs of it to agents connected with it that appear lower in the order. When an agent cannot find a consistent assignnment for its variables, it generates a nogood and sends it to the closest agent in such nogood. If an agent receives a nogood including another agent not connected with it, it requests to add a link from that agent to itself. This is done until a solution is found, or until it is proven that no solution exists. ABT messages are:

- $OK?(ag, val)$: it informs agent ag that the sender has taken the value val.
- $NGD(ag, ng)$: it informs agent ag that the sender considers ng as a nogood.
- $ADL(ag)$: it asks agent ag to set a direct link to the sender.
- STP: the sender has found that there is no solution.

The ABT pseudocode for distributed SAT appears in Figure **??**, where $self$ is the agent that executes ABT, $\Gamma^-(self)$ and $\Gamma^+(self)$ are the set of higher and lower agents connected with $self$, and the rest of elements are self-explanatory.

3. The ABT$_{\text{SAT}}$ Algorithm

ABT can be directly applied to solve distributed SAT. However, original ABT has to handle directly the operations of finding compatible assignments and producing new nogoods, tasks which can be efficiently done by modern SAT solvers. We have connected the ABT algorithm with a CDCL SAT solver, producing the new version ABT$_{\text{SAT}}$, which maintains the correctness, completeness and termination of original ABT. In this section, we present ABT$_{\text{SAT}}$ and develops its properties.

```
procedure ABT()
  compute $myClausesToEval$;
  $myValue \leftarrow$ empty; $end \leftarrow$ false;
  CheckAgentView();
  while ($\neg end$) do
    $msg \leftarrow$ getMsg();
    switch($msg.type$)
      $OK?$   : ProcessInfo($msg$);
      $NGD$   : ResolveConflict($msg$);
      $ADL$   : SetLink($msg$);
      $STP$   : $end \leftarrow$ true;
procedure CheckAgentView(msg)
  if $\neg$consistent($myValue, myAgentView$) then
    $myValue \leftarrow$ chooseValue();
    if ($myValue$) then
      for each $child \in \Gamma^{+}(self)$ do
        sendMsg:$OK?(child, myValue)$;
    else Backtrack();
procedure ProcessInfo($msg$)
  Update($myAgentView, msg.Assig$);
  CheckAgentView();
procedure ResolveConflict($msg$)
  if Coherent($msg.Nogood, \Gamma^{-}(self) \cup \{self\}$) then
    CheckAddLink($msg$);
    add($msg.Nogood, myNogoodStore$);
    $myValue \leftarrow$ empty;
    CheckAgentView();
  else if Coherent($msg.Nogood, self$) then
    sendMsg:$OK?(msg.sender, myValue)$;
procedure Backtrack()
  $newNogood \leftarrow$ solve($myNogoodStore$);
  if ($newNogood =$ empty) then
    $end \leftarrow$ true; sendMsg:$STP(system)$;
  else
    sendMsg:$NGD(newNogood)$;
    Update($myAgentView$,rhs($newNogood$) $\leftarrow$ unknown);
    CheckAgentView();
procedure Update($myAgentView, newAssig$)
  add($newAssig, myAgentView$);
  for each $ng \in myNogoodStore$ do
    if $\neg$Coherent(lhs($ng$)$, myAgentView$) then remove($ng, myNogoodStore$);
function Coherent($nogood, agents$)
  for each $ag \in nogood \cup agents$ do
    if $nogood[ag] \neq myAgentView[ag]$ then return false;
  return true;
procedure CheckAddLink($msg$)
  for each ($var \in$ lhs($msg.Nogood$))
    if ($\alpha(var) \notin \Gamma^{-}(self)$) then
      sendMsg:$ADL(\alpha(var), self)$;
      add($\alpha(var), \Gamma^{-}(self)$);
      Update($myAgentView, var \leftarrow varValue$);
procedure SetLink($msg$)
  add($msg.sender, \Gamma^{+}(self)$);
  sendMsg:$OK?(msg.sender, myValue)$;
```

Figure 1. The ABT algorithm for distributed SAT.

3.1. Connection with a SAT solver. Soundness, Completeness and Termination.

ABT$_{\text{SAT}}$ builds propositional formulas ϕ and asks a SAT solver to get models for them or, if ϕ is unsatisfiable, get an unsatisfiability core of it, as depicted in Figure 2. The procedures that differ from the ones of ABT for distributed SAT appear in Figure 3, where the new or changed lines appear marked with $\rightarrow$.

The call to SAT in CheckAgentView returns a compatible value *myValue*, or if UNSAT, a core *myCore* of the original clauses that make the whole formula unsatisfiable. In that case, by resolution we can eliminate the local variables of *self* that appear in *myCore*. This is done by the function filter-core, that generates *newNogood* from *myCore*.

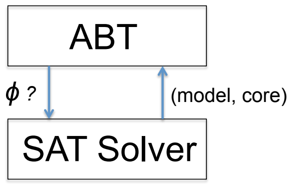

Figure 2. The structure of ABT$_{\text{SAT}}$.

```
procedure CheckAgentView(msg)
if ¬consistent($myValue, myAgentView$) then
→  ($myValue, myCore$) ← SAT($myAgentView \cup myClausesToEval \cup myNogoodStore$);
   if ($myValue$) then for each $child \in \Gamma^{+}(self)$ do sendMsg:$OK?(child, myValue)$;
   else Backtrack($myCore$);
procedure Backtrack($myCore$)
→  $newNogood$ ← filter-core($myCore$);
   if ($newNogood$ = empty) then
      $end$ ← true; sendMsg:$Stop(system)$;
   else
→     learn($newNogood$);
      sendMsg:$NGD(newNogood)$;
      Update($myAgentView$,rhs($newNogood$) ← unknown);
      CheckAgentView();
```

Figure 3. The ABT$_{\text{SAT}}$ algorithm; only changes wrt. original ABT are shown.

The ABT proofs of soundness and completeness [7] remain valid. If a solution exists, ABT$_{\text{SAT}}$ reaches a stable state where all agents satisfy their clauses and no more messages are exchanged. When no solution exists, the empty clause is produced by some agent, and the algorithm terminates. This argument also holds for completeness (notice that `filter-core` only eliminates variables that are local to the agent). Finally, these two cases (a solution is found, or the empty clause is produced) occur in finite time, since in ABT$_{\text{SAT}}$ there exist no infinite loops in the values of a certain agent: this is direct from ABT. See [7] for more details in this proof.

3.2. CDCL-based Subproblem Resolution

Each agent must satisfy all clauses for which it is the evaluator (the set *myClausesToEval*). In other words, it must find an assignment of its local variables satisfying these clauses. However, it must also take into account the literals assigned by higher priority agents, as well as the nogoods generated by lower priority agents. These tasks can be efficiently performed by a CDCL SAT solver. We build a SAT instance as follows.

First, this formula contains all the clauses to be evaluated by the agent. Notice that these clauses must be satisfied by this agent in *all* circumstances. Also, it must consider the values stored in the agent view. Each *OK*? message contains a set of literals, i.e., a set of unitary clauses, whose value is already fixed by other agents. Therefore, it is enough to add these unitary clauses to such formula. Finally, all nogoods that remain active, that is, are consistent with the agent view of *self*, must be also included in this formula[1].

Solving this SAT instance is equivalent to find an assignment that (i) satisfies all the clauses to be evaluated by the agent, and (ii) is consistent with the agent view (higher priority agents) and nogood store (lower priority agents). If the solver returns SAT, it also returns the assignment satisfying the formula, i.e., the assignment of the local variables of the agent. On the contrary, if it returns UNSAT, this indicates that no assignment can be found under the current agent view, and one can produce the conflict clause from the unsatisfiability core[2]. In particular, this clause contains all the variables (with its corresponding phase) that belong to the UNSAT core.

[1]Notice that any nogood that is no longer consistent will be trivially satisfied.

[2]The reduced set of clauses from which the empty clause can be derived.

3.3. Clause Learning and Removal Policies

In constrast to original ABT, we propose to keep some nogoods received during the execution in $\mathrm{ABT}_{\mathrm{SAT}}$, even if they are no longer consistent.[3] This may be beneficial because some may become consistent again, after changing the agent view. However, storing nogoods has drawbacks. Let us consider an agent handling n Boolean variables. In the worst case, this agent needs the reception of 2^n nogoods to produce a new nogood. If we keep all, the consumption of memory grows exponentially. This problem also exist in CDCL SAT solvers, where a new learnt clause is produced after each conflict.

In order to reduce this problem, we propose the use of clause removal policies. In CDCL SAT solvers, these policies remove the less *active* learnt clauses during the execution of the solver. The activity of a clause is measured by the number of conflicts that such clause produces. Similarly, we can measure the effectiveness of each nogood to produce conflicts, and remove those of them that are no effective any more. A direct way to do so is using the CDCL SAT solver. Notice that each nogood is encoded as a clause. They cannot be removed during the execution of the solver. However, their activity counters are updated. If we keep this information at each call to the solver, we can know which are the most active nogoods at a certain moment. Therefore, we can directly apply some clause removal policies in order to reduce their memory consumption.

4. Future Work and Conclusions

We presente $\mathrm{ABT}_{\mathrm{SAT}}$, a sound and complete algorithm to solve distributed SAT in finite time, which combines the benefits of CDCL SAT solving techniques to solve real-world SAT instances with the benefits of ABT to handle communications among agents in distributed problem solving scenarios. As future work, we plan an extense experimentation, in order to evaluate the benefits of this approach. In addition, we address the question of ordering: ABT requires a total order among the problem agents, but, how to select this order? It would be interesting to analyze if certain orders based on the particular features of each instance may be beneficial to improve the performance of $\mathrm{ABT}_{\mathrm{SAT}}$.

References

[1] A. Baker. The hazards of fancy backtracking. *Proc. AAAI'94*, pages 288–293, 1994.

[2] M. Benedetti and L. C. Aiello. SAT-based cooperative planning: a proposal. In *Mechanizing Mathematical Reasoning*, pages 494–513, 2005.

[3] Y. Dimopoulos, M. A. Hashmi, and P. Moraitis. μ-SATPLAN: Multi-agent planning as satisfiability. *Knowledge-Based Systems*, 29:54–62, 2012.

[4] G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. *Proc. CP'03*, pages 873–877, 2003.

[5] R. Nissim, R. Brafman, and C. Domshlak. A general, fully distributed multi-agent planning algorithm. *Proc. AAMAS'10*, pages 1323–1330, 2010.

[6] J. P. M. Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009.

[7] 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.

[3] In ABT, nogoods are removed as soon as they become inconsistent.