Improving Asynchronous Backtracking for Dealing with Complex Local Problems

Arnold Maestre[1] **and Christian Bessiere**[1]

Abstract. Distributed constraint satisfaction, in its most general acceptation, involves a collection of agents solving local constraint satisfaction subproblems, and a communication protocol between agents, in order to allow the distributed system converge to a global solution. The literature, however, often concentrates on the reduction where each agent owns exactly one variable, under the rationale that the corresponding algorithms are easily extended to the most general case. While this is mostly true, the specificities of agents handling local CSPs give way to numerous improvements, since a trade-off becomes possible between local and distributed search effort. In this paper, we seek to improve nogood learning and solver cooperation in multi-variables distributed constraint satisfaction problems. We propose incremental improvements to be implemented on top of an ABT-like algorithm, and make experimental evaluations of the performance improvement they bring.

1 INTRODUCTION

Constraint satisfaction is a powerful paradigm for solving combinatorial problems. It has been widely used for modeling problems in artificial intelligence and solve real world problems. The increasing interest in distributed computing has led to distributed approaches of the constraint satisfaction problem (DisCSP), where the global problem is naturally distributed among a set of agents, which then have to communicate in order to match their local solutions.

Those approaches can be classified following a set of characteristics: search control (centralized or distributed), communication protocol (shared memory or message passing), memory requirements (polynomial or not), completeness and agent or variable ordering (static or dynamic). Additionally, DisCSP solvers are said to be variable-based when each agents cares for a given subset of the variables, or constraint-based if the focus is put on constraints being distributed among agents.

In the following, we will focus on complete, statically ordered, variable-based, message passing distributed CSP algorithms with decentralized control. This field is dominated by the algorithms of Yokoo and colleagues, asynchronous backtracking (ABT) [20], asynchronous weak commitment (AWC) [17] and distributed break-out (DB) [18]. A pioneer algorithm in the field, ABT is a statically ordered, asynchronous protocol in which high priority agents propose values to lower priority ones, that can refute them with a nogood in case of inconsistency. It has been shown to be complete under polynomial space constraints.

In AWC, agents communicate with all their neighbors and reorder themselves dynamically in order to soften the influence of a bad choice from a high priority agent. Although AWC is incomplete unless agents can store a potentially exponential number of nogoods, Silaghi et al have shown in [14] that it is possible to build a polynomial space algorithm with dynamic reordering as soon as one accepts to bound the maximum time from start during which an agent can be reordered.

Finally, DB works in synchronous phases, where agents in a neighborhood agree on the local move bringing the best improvement to the current state of the system and increase the weight of violated constraints in order to escape local minima.

All these protocols are applied to variable-based distributed problems. Although the generic framework for such problems makes it possible for one agent to own several variables, few works explore the specificities of complex local subproblems, namely the complexity of agent instantiation, which can no longer be seen as a quasi-instantaneous operation, the increasing necessity of agent collaboration, and the generation of meaningful nogoods. Instead, most of these works focus on the ordering of agents and/or variables, which is typically even more important when the granularity of local agents increases, and is indeed a good way to improve performance. In [1] the authors use different heuristics to statically order or dynamically re-order complex agents running a specialized hybridation of ABT and AWC. In [19] this concept is pushed further with the multi-AWC algorithm, based on AWC, and individual variables, rather than agents, are re-ordered to improve search efficiency. Likewise in [10], multi-DB, a multi-variable variant of the distributed break-out algorithm is proposed, which performs well compared to multi-AWC on 3-SAT benchmarks.

In [11], the authors evaluate the computational effort and network load needed to solve distributed 3-coloring problems with varying numbers of intra-agent and inter-agent constraints. The algorithm used is a clever extension of ABT, which minimizes the number of messages sent by avoiding those that are trivially unnecessary, but doesn't take into account the issues inherently linked to the complexity of local problems. Silaghi addresses some of these issues in [15], where he proposes various solutions: interruptible backtracking, message queues processed by compactors in order to reduce the number of messages effectively handed to the core agent, and communication policies in order to cool down message bursts.

None of these works, however, insist on producing interesting nogoods from conflicting multi-variable agents, nor do they explore the collaboration between complex agents when a new instantiation is to be chosen. Actually, each new instantiation, if different from the previous one, produces a new batch of messages to be sent to lower priority agents, which in turn may have to change their instantiation. In the multi-variable case, the problem is a bit different, since a carefully chosen instantiation may allow the agent to inform only

[1] LIRMM-CNRS 161, rue Ada, 34392 Montpellier Cedex 5, France. Email: maestre|bessiere@lirmm.fr

a subset of its lower priority neighbors. The cost of message passing being (in the most general case) prohibitively higher than the cost of local computation, it seems interesting to make a bit more local computation to save some messages (aka *stay and play*) rather than going with the first feasible local solution (aka *scoop and run*). In particular, if a solution exists which allows the agent to satisfy all constraints while minimizing the number of changes to variables linked to lower priority agents, it should be preferred over a more disturbing one.

In the following, we will explore those issues, and provide ways to produce high quality nogoods and cooperative instantiations in a distributed constraint satisfaction problem with multiple variables per agent. We will then present an experimental framework, as well as an empirical evaluation of our work.

2 PRELIMINARIES

In a centralized setting, a constraint network is defined by a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of n variables, $\mathcal{D}$ is the set of their respective finite domains whose size is at most d, and $\mathcal{C}$ is a set of constraints. A constraint is a relation over a subset of variables, which defines the combinations of values deemed acceptable for those variables. In the following, we will restrict our attention to constraints involving two variables, namely *binary* constraints. A constraint among the variables x_i and x_j will be denoted by c_{ij}. A *solution* is an assignment of values to variables which satisfies every constraint. The constraint satisfaction problem (CSP) is generally formulated as finding a solution to the constraint network. This is usually done by a combination of tree-search with backtracking, look-ahead and backjumping techniques.

A distributed constraint satisfaction problem (DisCSP) is a CSP where the variables, domains and constraints of the network are distributed over a set of autonomous agents. Formally, a variable-based distributed constraint network is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \varphi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are as before. $\mathcal{A}$ is a set of p agents, and $\varphi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps each variable to its agent. Each variable belongs to one agent. The distribution of variables divides $\mathcal{C}$ in two disjoint subsets, $\mathcal{C}_{intra} = \{c_{ij} | \varphi(x_i) = \varphi(x_j)\}$, and $\mathcal{C}_{inter} = \{c_{ij} | \varphi(x_i) \neq \varphi(x_j)\}$, called intra-agent and inter-agent constraint sets, respectively. An intra-agent constraint c_{ij} is known by the agent owner of x_i and x_j, and it is unknown by other agents. Usually, it is considered that an inter-agent constraint c_{ij} is known by the agents $\varphi(x_i)$ and $\varphi(x_j)$ [21, 8]. As in the centralized case, a solution of a DisCSP is an assignment of values to variables satisfying every constraint. The agents are ordered, as it is the case in ABT. An agent A_k before A_l in the ordering is said to have higher priority than A_l. We suppose the variable ordering is consistent with the agent ordering: if x_i belongs to agent A_k and x_j belongs to A_l, x_i is before x_j in the ordering if A_k is before A_l. $\Gamma^-(k)$ (resp $\gamma^-(k)$) is the set of agents (resp. variables) constrained with variables belonging to agent A_k and appearing before it in the agent ordering. Conversely, $\Gamma^+(k)$ (resp. $\gamma^+(k)$) is the set of agents (resp. variables) constrained with variables in A_k and appearing after it in the agent ordering. In the case of non-trivial, ordered agents, the subproblem Π_k to be solved by A_k is composed of all variables belonging to A_k, its intra-agent constraints as well as inter-agent constraints linking it to higher priority agents.

DisCSPs are solved by the collective action of agents in $\mathcal{A}$, which asynchronously run a process of distributed constraint satisfaction, and send messages to other agents in the system in order to have all local solutions converge to a global one. Considering message passing, it is currently assumed that the delay in delivering a message is finite but random. For a given pair of agents, messages are delivered in the order they were sent.

Let us denote $S(k)$ the set of variables belonging to agent A_k. $S(k)$ can be partitioned in two disjoint subsets, $S_i(k) = \{x_j \in A_k | \forall c_{jm}, x_m \in A_k\}$ and $S_o(k) = \{x_i \in A_k | \exists c_{ij}, x_j \notin A_k\}$. In a nutshell, $S_i(k)$ is a set of private variables, which only share constraints with variables inside A_k. Conversely, $S_o(k)$ is a set of variables linked to the outside world and sometimes referred to as *negotiation variables*. Moreover $S_o(k)$ can be seen as the union of two (not necessarily disjoint) sets, $S_o^+(k) = \{x_i \in A_k | \exists c_{ij}, x_j \in A_l, l > k\}$ and $S_o^-(k) = \{x_i \in A_k | \exists c_{ij}, x_j \in A_l, l < k\}$. $S_o^+(k)$ is the set of variables constrained with variables belonging to agents lower than A_k in the ordering, similarly $S_o^-(k)$ is the set of variables constrained with variables belonging to higher priority agents. Finally, we will denote $S_o^j(k)$ the set of variables in A_k constrained with variables in A_j.

The instantiation of variables in $S(k)$ will be denoted $I(k)$, and the instantiation of a subset u thereof, $I(k)_{|u}$.

Some distributed constraint satisfaction algorithms make heavy use of nogoods to exchange data about unfeasible partial assignments. A *directed nogood* for a value c of variable x_k is an expression of the form, $(x_i = a) \wedge (x_j = b) \wedge \ldots \Rightarrow (x_k \neq c)$ meaning that the assignment of c to x_k is inconsistent with the assignments of $a, b, \ldots$ to $x_i, x_j, \ldots$. This nogood explains the deletion of value c, and is relevant as long as values $a, b, \ldots$ are assigned to variables $x_i, x_j, \ldots$, and is equivalent to its non-directed counterpart, $\neg((x_i = a) \wedge (x_j = b) \wedge \ldots \wedge (x_k = c))$. The left-hand and right-hand sides (abbreviated as *lhs* and *rhs* respectively) of the directed nogood are defined from the position of the "$\Rightarrow$" symbol.

When together several nogoods yield an inconsistency, they are resolved into a new nogood, following the method exposed in [7]. Let $(x_j = b)$ be the lowest priority assignment in the left-hand sides. The *lhs* of the resolvent will have the conjunction of the left-hand sides of all inconsistent nogoods, barring references to x_j. The right-hand side is then $(x_j \neq b)$. The new nogood should be stored by the process owning x_j to explain the deletion of (x_j, b), while all nogoods referring to $(x_j = b)$ become irrelevant and can be deleted.

3 BASE ALGORITHM

3.1 Asynchronous Backtracking

ABT is one of the first complete algorithms proposed to solve DisCSPs [20]. Although early publications focus on agents owning exactly one variable, [11] extends its purpose to complex local agents. We will use this algorithm as a base for our work.

In this protocol, agents are ordered statically and inter-agent constraints are directed from high priority agents to low priority ones to build an acyclic graph. Each agent runs a similar process, and stores information about the outside world, in the form of an *agent view* and a *nogood store*. The agent view of an agent A_j is the set of values that it believes to be assigned to variables belonging to higher priority agents and constrained with a variable in A_j. The nogood store is a list of nogoods which are either generated locally to reflect the impact of some higher priority agent's decision on the local problem, or received from lower priority agents to forbid some unfeasible value combination. A choice has been made to store at most one nogood per value, in order to keep the storage space requirements reasonable.

Each agent A_j solves its own subproblem and communicates asynchronously with neighboring agents to build a global solution by matching local ones. Agents generally exchange assignments from

top to bottom using *info* messages and nogoods (from the bottom up) using *back* messages. When receiving an assignment, an agent has to check its local solution for consistency and, if it is not consistent, it must find a new feasible assignment. When receiving a relevant nogood (that is, one that is not obsolete considering the local agent view), A_j has to remove the corresponding value, store the nogood as a justification of this deletion and again find a new assignment. If the *lhs* of the nogood received includes a variable x_k yet unconstrained with A_j, a request is sent via a third type of message, *addlink*, to the higher priority agent A_i owning x_k. A_i must then add a communication link between A_i and A_j regarding x_k and inform A_j if the value of x_k changes in the future.

Finding a solution to the local subproblem is done the same way as in the centralized case, with a tree search. Whenever an agent A_i cannot find a consistent assignment to one of its variables x_k, either because of the original constraints or because of the stored nogoods, a new nogood is generated from those emptying the domain for x_k. Let x_j be the lowest priority variable in the nogood. If x_j belongs to A_i, it is a local back-jump, otherwise the nogood is sent to the agent owning x_j. If an empty nogood is produced, the problem is unsolvable. In fact, no superset of a nogood can be part of solution. The agent exits after sending a *stop* message to all its neighbors.

Yokoo, Sycara and Hirayama propose a number of enhancements for their algorithm. First, the agent and variable orderings follow a max-degree heuristic. Second, as explained above, they use nogood learning inside the agent in order to be able to backjump instead of simply backtracking on the local subproblem. Additionally, they try as much as possible to keep the internal variables to their previous values. Finally, an agent keeps all external nogoods generated during its instantiation phase in a stack, and sends them all at once when done. It also sends the values of changed variables to its neighbors when the instantiation is complete.

3.2 Discussion

In order to keep internal and external knowledge separated, and free ourselves from considering the algorithm to be used for solving internal subproblems, we will not derive nogoods inside agents. Thus, the nogood store for a given agent A_k will contain nogoods derived from the values of variables in $\gamma^-(k)$, concerning variables in $S_o^-(k)$, and nogoods received from agents in $\Gamma^+(k)$, concerning variables in $S_o^+(k)$. This way, the local solver can be completely independent from the communication protocol, since it will be called after the necessary domain reductions have been operated. This allows the use of *ad hoc* solvers, including local propagation and consistency maintenance.

Moreover, when an agent A_i is successfully instantiated, it is not necessary to send all changed values to all neighbors, since only agents of $\Gamma^+(i)$ need to be informed, and the only noteworthy changes are those occurring in $S_o^+(i)$. Let $I_p(i)$ be the previous instantiation, and $I_c(i)$ the current one. Then, for each agent A_k in $\Gamma^+(i)$, if $\{I_c(i)_{|S_o^k(i)} - I_p(i)_{|S_o^k(i)}\} \neq \emptyset$, this set difference should be sent to A_k. This allows A_i to avoid sending useless messages to agents on which its new instantiation has no impact, while keeping the actual messages as terse as possible.

4 IMPROVED NOGOOD LEARNING

4.1 Nogood Selection

In a perfect world, each agent would be able to store an arbitrary number of nogoods, and select the best resolvent when failing to find a consistent set of assignments for its variables. Unfortunately, in the most general case, storage space is in limited quantity, and selecting the most suitable nogood with respect to one particular criterion (or set thereof) means generating all possible candidates in order to extract the best one, which could be prohibitively costly.

The nogood store keeps at most one nogood per value, that is, even if there are several relevant justifications to the deletion of one particular value in S_o, only one is stored. This behavior defaults to storing the first justification met during the search, and discarding all subsequent justifications, since only one is stored. Hence, only one nogood can be generated upon failure, and it mostly depends on the timing of events during search (different timings yielding different resolvents).

However, if several justifications are valid for some values, actively selecting the one to store could allow agents to see past the randomness of message delivery. When comparing two nogoods eliminating the same value, a heuristic criterion could be to select the nogood with *the highest possible lowest variable involved*. This way, the resolvent nogood would be sent as high as possible in the agent ordering, thus saving unnecessary search effort [3].

As seen in section 2, agents store two types of nogoods: those generated locally, which are the consequences of $info$ messages on the domains of variables belonging to S_o^-, and those received through $back$ messages to forbid values on variables in S_o^+.

ABT accepts a $back$ message if the incoming nogood is consistent with its whole agent view, including its own assignment. Once this nogood is stored, the local value it refers to is eliminated, which makes the nogood consistent with the whole agent view except for the local assignment. Following the definitions in [2, 5] the nogood is 1-relevant. If this agent receives a 1-relevant nogood for the same variable in a $back$ message, where the only discrepancy with this variable's assignment, this nogood deserves to be considered because it brings interesting information: it gives a valid justification to discard a value, even though that value may already have been discarded. As a matter of fact, given a different timing, this very nogood could be the one stored, and the other one would be in the incoming message list. Thus, both have to be compared from a heuristic point of view, and the best one should be selected.

Considering $info$ messages, the situation is a bit different, since the information they are derived from is stored locally. Actually, each agent is supposed to know the inter-agent constraints relevant to its subproblem, as well as the values of variables in γ^-, which are stored in its agent view. So there is no need to aggressively challenge stored nogoods each time the agent view is updated with a new $info$ message. Instead, the agent can wait until it really needs an optimal nogood store – that is, when a failure is detected and a $back$ message must be sent – and then only, check each value in S_o^- against each variable in γ^-, replacing existing nogoods with potentially better ones in the process. As a result, the resolvent nogood should be more efficient.

4.2 Nogood minimization

It has been hinted in [9], following [12], that it is possible to minimize the nogoods resolved upon failure, in order to improve their efficiency, this process being in the most general case a very time-consuming one. This issue is even more dramatic in the case of complex local problems. Indeed, inconsistency becomes harder to prove as local problems grow bigger, and in the same time, it becomes harder to sort out which of the stored nogoods really cause said inconsistency. Checking whether a subset of the resolved nogood is a

conflict set or not was a linear test in the single variable setting. With multiple variables per agent, it becomes full-fledged coNP-complete. If we want to be able to minimize our resolvent-based nogoods, we should at least find a way to optimize the subset exploration.

To this end, when the failure is detected, we will resolve a nogood Ng as shown above. Then, we will try and determine which external assignments, if any, are redundant in Ng, by repeatedly solving the local problem while activating only a subset of stored nogoods, corresponding to a subset of assignments in Ng.

Let $\mathcal{N}$ be the set of nogoods selected by A_i upon failure, and $\mathcal{K}$ the resolvent of $\mathcal{N}$ (the set of assignments appearing in the left-hand sides). Nogoods, as well as assignments, can be seen as additional constraints. Let then $\Pi_{\mathcal{K},\mathcal{N}}(i)$ be $\Pi(k) + \mathcal{N} + \mathcal{K}$. Since A_k failed to instantiate, we know that $\Pi_{\mathcal{K},\mathcal{N}}(i)$ is inconsistent. Hence, $Solve(\Pi_{\mathcal{K},\mathcal{N}}(i))$ returns false. We are now looking for a smaller set $\kappa \subset \mathcal{K}$, such that $\Pi_{\kappa,\mathcal{N}}(i)$ is still inconsistent. We can easily build such a set by ordering the assignments in $\mathcal{K}$. Starting from $\kappa = \emptyset$, we just have to add assignments from $\mathcal{K}$ into κ until $\Pi_{\kappa,\mathcal{N}}(i)$ returns false. This function is called $ShortenResolvent$.

De Siqueira and Puget propose a polynomial algorithm to reduce a conflict-set (in the sense of arc-inconsistent subset of constraints) until it is minimal wrt inclusion in [4]. Although we won't be able to claim polynomial performance in the case of resolvent minimization, we can propose an adaptation of their algorithm.

The first step involves generating a subset of $\mathcal{K}$ that renders the subproblem inconsistent, if such a subset exists. The $ShortenResolvent$ algorithm is then called recursively with the previous list of assignment in which the last assignment becomes the first. This process is repeated until the last element of the new list is equal to the last element of the initial list. The list now contains the assignments of a minimal resolvent nogood.

5 AGENT COOPERATION

In [13], Petcu and Faltings evaluate the usefulness of local contention techniques to avoid spreading conflicts among agents running the distributed breakout algorithm. Using interchangeability techniques, they manage to repair conflicts with a number of neighboring agents without involving other, still non-conflicting, nodes.

5.1 Solution stability

A similar idea can be applied in ABT-like algorithms, by avoiding unnecessary *flips*, or value changes, during the instantiation phase. Since a satisfied agent only propagates its *flips*, keeping the new solution to the subproblem as close as possible to the previous one looks like a valid heuristic.

Each local subproblem can be seen as a dynamic CSP, with each incoming message adding or removing constraints dynamically. If we consider it so, we can benefit from the conclusions of [16]. In this paper, Verfaillie and Schiex analyze the respective benefits of various methods for maintaining an existing solution in a CSP subject to dynamic changes. To this end, they measure the number of constraints checks needed by each type of algorithm in function of the size of the change, for various types of problems (under-constrained, critically constrained and over-constrained). Among these methods are a nogood recording algorithm very similar to the one used in [11], and a simple backtracking, restarting from scratch with a simple value ordering heuristic: use the *latest successful assignment* first if available. Noticeably, they show that there is no significant difference between nogood recording and backtracking on under-constrained and critically constrained problems, neither in terms of performance nor in terms of solution stability.

This is interesting, because in the case of distributed CSPs, local subproblems are likely to be under-constrained. An over-constrained subproblem is indeed fatal to the system, while a critically constrained one probably means that the global problem is unsolvable, since it can only add constraints to this already critical instance.

When trying to preserve the internal solution across search, it is important to notice that not all flips deserve the same attention, because not all of them will be propagated. In fact, only value changes occurring in S_o^+ need to be disclosed to lower priority agents. So instead of trying to preserve the former value of every single internal node, it should be more rewarding to concentrate on the variables constrained with lower priority agents. To this end, and keeping in mind the idea of a black-box, independent internal solver, we decided to prioritize variables of S_o^+ in the variable ordering.

5.2 Value selection heuristics

It has been noted in [17] that one of the main differences between ABT and AWC is that AWC uses the *min-conflict* metric as a value ordering heuristic. That is, when selecting a value, if there exist multiple values satisfying all constraints with higher priority variables, the one that minimizes the number of constraint violations with lower priority ones should be preferred. Contrary to the claim that this feature is easily introduced into ABT, this is only possible because AWC agents send their values to all neighbors (not just lower priority ones). In ABT, agents are not aware of lower priority agents' values, so *min-conflict* is not an option. To improve inter-agent cooperation, we chose to use a variant of Geelen's *Promise* heuristic [6]. This heuristic selects a value that least reduces the possible assignments for the remaining uninstantiated variables, by preferring the value v that maximizes the product of the number of supports for v in all neighboring variables. Our value ordering will be a bit different, in that we only want to maximize the choice for *extra-agent*, lower priority neighboring variables, so we'll only have to use the heuristic on variables in S_o^+, whenever those variables cannot keep their latest successful assignment, without considering *intra-agent* neighboring variables. Moreover, the effects of the heuristic will be empowered when used in conjunction with our variable ordering heuristic, which selects variables in S_o^+ first.

6 EXPERIMENTS

6.1 Experimental Setting

Experiments in the distributed constraint satisfaction framework differ mainly on two parameters: the way in which the distribution is achieved or simulated, and the problems to solve. We chose to have each agent be an independent process, all agents running on a single machine and communicating through a loop-back interface. Thus, agent activity is asynchronous, and the scheduling process is handled by the OS kernel, which is supposed to be fair and efficient.

We evaluated the different algorithms detailed above on binary random CSPs. A binary random DisCSP class is characterized by $\langle \#A, n, d, C, T, iC, iT \rangle$ where $\#A$ is the number of agents, n is the number of variables, d the number of values per variable, C (resp. iC) the network *connectivity* defined as the number of inter-agent constraints (resp. the number of intra-agent constraints on each agent), and T (resp. iT) the constraint *tightness* defined as the number of forbidden value pairs on inter-agent or intra-agent constraints,

respectively. The constrained variables and the forbidden value pairs are chosen at random. Each agent is assigned all its variables, and the constraints binding them to the neighboring agents.

The problems generated all had 60 variables with a domain of size 10 and 180 constraints of tightness 50. In the centralized case, this problem is situated at the phase transition, with an equivalent number of solvable and unsolvable instances. We then proceeded to generate random binary DisCSPs by choosing a number of agents among which to distribute the problem. When varying the number of agents, we split variables evenly between agents (i.e., $n/\#A$ variables per agent), and we modified the relevant parameters in order to keep an equal number of inter-agent and intra-agent constraints (i.e., $\#A \cdot iC = C/2$) whenever it was possible. For example, with 15 agents, we used the class $\langle 15, 60, 10, 90, 50, 6, 51 \rangle$. Notice we adjusted iT in order to keep the problem at the complexity peak. Using this scheme, the more agents we generate, the more uniform the problem structure becomes. With 60 agents, it is a simple mapping of the uniform random binary CSP to the agents.

6.2 Results

Considering performance evaluation, we report the number of sequential messages (#msgs), and sequential constraint checks (#ccks). The former represents the length of the longest chain of sequential messages needed to solve the problem, and is a good evaluation of the distributed performance of the algorithm, while the latter can be seen as a measure of computational requirement across search.

We compared 4 different protocols. Results are summarized in Table 1. Algorithms in the right-hand side of the table use improved nogood learning, while algorithms at the bottom of the table try and cooperate with lower priority agents. Hence, ABT is our implementation of Yokoo's ABT, shown as a base for comparison. On top of ABT, ABT_{nl} puts our nogood selection and minimization strategies to good use, ABT_{ac} implements our agent cooperation schemes and ABT_m implements all of this.

ABT	#ccks	#msgs	ABT_{nl}	#ccks	#msgs
#A=5	39,901	1,697	#A=5	91,244	1,512
#A=10	51,954	2,074	#A=10	106,517	2,050
#A=15	57,931	5,637	#A=15	118,742	4,587
#A=20	88,841	8,517	#A=20	179,238	7,351
#A=30	135,614	12,204	#A=30	257,411	10,621
ABT_{ac}	#ccks	#msgs	ABT_m	#ccks	#msgs
#A=5	42,217	1,544	#A=5	107,807	1,442
#A=10	53,954	2,051	#A=10	114,240	1,979
#A=15	60,663	4,687	#A=15	127,621	4,509
#A=20	85,006	7,423	#A=20	162,642	7,123
#A=30	119,754	10,948	#A=30	242,981	10,143

Table 1. Results on problems with various agent granularity.

The nogood minimization scheme (ABT_{nl}) is a very effective way to decrease the number of sequential messages, but it incurs a significant overhead in terms of constraint checks. Happily, local subproblems are less constrained than the global one, thus they fall in the under-constrained region for problems at the peak, otherwise, repeatedly solving the problem in order to minimize a nogood could prove counter-productive; in that case, using an anytime approach should help balance the penalty. Strikingly, the agent cooperation improvement (ABT_{ac}) offers comparable savings in #msgs at a fraction of the #ccks cost. Of course, the penalty for the support counting heuristic is clear on big agents (small number of agents), as well as the burden of using a locally less than optimal variable ordering. Still there is a significant payoff, and ABT_{ac} ends up making less constraint checks than plain ABT on problems with lots of small agents.

Unsurprisingly, ABT_m improves over all other methods in terms of message passing, but needs a powerful CPU in order to do so. Like its brethren ABT_{nl}, it would be best used if message passing is very slow and/or very costly when compared to CPU cycles (which we believe is often the case).

7 CONCLUSION

In this paper, we presented incremental enhancements for solving distributed CSPs with complex local problems using ABT-like procedures. Those enhancements are in the fields of nogood learning, through careful selection of stored nogoods and minimization of locally generated resolvents, and agent cooperation, through helpful attitude towards lower priority agents and increased stability of local solutions. The experimental results show consistent performance improvements in terms of sequential message passing and global network load, with various agent granularity, although this comes at the cost of local computation.

REFERENCES

[1] A. Armstrong and E. Durfee, 'Dynamic priorization of complex agents in distributed constraint satisfaction problems', in *Proc. IJCAI 1997*.

[2] R.J. Bayardo and D.P. Miranker, 'A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem', in *Proc. AAAI 1996*.

[3] C. Bessiere, I. Brito, A. Maestre and P. Meseguer, 'The ABT Family', Technical Report LIRMM-CNRS, 2003.

[4] N. de Siqueira and J. F. Puget, 'Explanation-based generalisation of failures', in *Proc. ECAI 1988*.

[5] R. Dechter, 'Constraint networks', in *Encyclopedia of Artificial Intelligence*, Wiley and Sons, (1992).

[6] P.A. Geelen, 'Dual viewpoint heuristics for binary constraint satisfaction problems.', in *Proc. ECAI 1992*.

[7] M.L. Ginsberg, 'Dynamic backtracking', *Journal of Artificial Intelligence Research*, **1**, (1993).

[8] Y. Hamadi, C. Bessière, and J. Quinqueton, 'Backtracking in distributed constraint networks', in *Proc. ECAI 1998*.

[9] K. Hirayama and M. Yokoo, 'The effect of nogood learning in distributed constraint satisfaction', in *Proc. ICDCS 2000*.

[10] K. Hirayama and M. Yokoo, 'Local search for distributed sat with complex local problems', in *Proc. AAMAS 2002*.

[11] K. Hirayama, M. Yokoo, and K.P. Sycara, 'The phase transition in distributed constraint satisfaction problems: First results', in *Proc. CP 2000*.

[12] D. Mammen and V. Lesser, 'Problem structure and subproblem sharing in multi-agent systems', in *Proc. ICMAS 1998*.

[13] A. Petcu and B. Faltings, 'Applying interchangeability techniques to the distributed breakout algorithm', in *Proc. IJCAI 2003*.

[14] M.C. Silaghi, 'Abt with asynchronous reordering', in *Proc. IAT 2001*.

[15] M.C. Silaghi, *Asynchronously solving distributed problems with privacy requirements*, Ph.D. dissertation, EPFL, Lausanne, 2002.

[16] G. Verfaillie and T. Schiex, 'Maintien de solution dans les problèmes dynamiques de satisfaction de contraintes: bilan de quelques approches', *Revue d'Intelligence Artificielle*, **9**(3), 1995.

[17] M. Yokoo, 'Asynchronous weak-commitment search for solving distributed constraint satisfaction problems', in *Proc. CP 1995*.

[18] M. Yokoo, K. Hirayama, 'Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems', in *Proc. ICMAS 1996*.

[19] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara, 'Distributed constraint satisfaction algorithm for complex local problems', in *Proc. ICMAS 1998*.

[20] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara, 'Distributed constraint satisfaction for formalizing distributed problem solving', in *Proc. DCS 1992*.

[21] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara, 'The distributed constraint satisfaction problem: Formalization and algorithms', *IEEE Trans. Knowledge and Data Engineering*, (1998).