Appears in: *Recent Advances in Constraints, Springer-Verlag LNAI-3419, p71-85, 2005*

Dynamic Distributed BackJumping

Viet Nguyen[1], Djamila Sam-Haroud[2], and Boi Faltings[2]

[1] Laboratory of Autonomous Systems
[2] Laboratory of Artificial Intelligence
Ecole Polytechnique Federale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
{viet.nguyen,jamila.sam,boi.faltings}@epfl.ch

Abstract. We consider Distributed Constraint Satisfaction Problems (DisCSP) when control of variables and constraints is distributed among a set of agents. This paper presents a distributed version of the centralized BackJumping algorithm, called the *Dynamic Distributed BackJumping* - *DDBJ* algorithm. The advantage is twofold: *DDBJ* inherits the strength of synchronous algorithms that enables it to easily combine with a powerful dynamic ordering of variables and values, and still it maintains some level of autonomy for the agents. Experimental results show that *DDBJ* outperforms the *DiDB* and *AFC* algorithms by a factor of *one to two* orders of magnitude on hard instances of randomly generated DisCSPs.

Keywords: Search, Constraint Satisfaction, Distributed Systems, Multi-Agent Systems.

1 Introduction

Constraint Satisfaction has been used as a powerful paradigm for general problem solving. It consists of finding values for problem variables in some particular domains subject to constraints that specify possible consistent combinations. Solving a CSP is to find a set of variable assignments that satisfies all the constraints.

A distributed CSP (DisCSP) is a CSP where variables and constraints are distributed among a network of automated agents. Each agent may hold one or more variables which are connected by local constraints, and also connected by inter-constraints to variables of other agents. Many application problems in Multi-Agent Systems (MAS) can be formulated and solved using a DisCSP framework ([1]), such as distributed resource allocation problems, distributed scheduling problems or multi-agent truth maintenance tasks.

In solving DisCSPs, agents exchange messages about the variable assignments and conflicts of constraints. Several distributed search algorithms have been proposed for solving DisCSPs. They can be divided into two main groups: asynchronous and synchronous algorithms. The former are algorithms in which the process of assigning variable values and exchanging messages is performed asynchronously between the agents, whereas in the latter group, agents assign values to variables in a synchronous, sequential way. Each group has different strengths and drawbacks.

The main contribution of this paper is to introduce the first distributed version of the centralized algorithm *BackJumping* ([2]), called *Dynamic Distributed BackJumping*. The advantage is twofold: *DDBJ* inherits the strength of synchronous algorithms that enables it to easily combine with a powerful dynamic ordering of variables and values, and still it maintains some level of autonomy for the agents. Experimental results show that *DDBJ* outperforms some existing algorithms.

2 Related Work

One of the pioneer algorithms is the *Asynchronous BackTracking - ABT* algorithm ([3, 4]). It is a distributed, asynchronous version of a generic backtracking algorithm. Agents communicate by two types of messages: `OK?` messages to distribute the current value, and `Nogood` messages to declare new constraints. The simplicity and computational concurrency are its strengths. *ABT* needs polynomial space for storing nogoods to be complete ([3]). The algorithm requires the assumption that messages are received in the order in which they were sent for completeness, otherwise all nogoods have to be stored and it would suffer from exponential space complexity. One way to work around is to attach a sequence number for each message, so the order of messages can be determined at the receiving end.

A later version of *ABT* which makes use of dynamic ordering of agents, called the *Asynchronous Weak-Commitment Search - AWC*, is given in [4]. This algorithm is shown to be faster than *ABT*, but the main drawback is that it requires exponential space for completeness.

The *Distributed Dynamic Backtracking - DiDB* algorithm is another distributed, asynchronous algorithm which is inspired by its centralized version *Dynamic Backtracking* ([5]), presented in [6, 7]. The algorithm transforms the constraint network into a directed acyclic graph and performs dynamic jumps over the set of conflicting agents. This algorithm requires the assumption that messages are received in the order in which they were sent and polynomial space for nogood stores. However, the main weakness is the problem of message duplication. Due to asynchrony, an agent may keep asking values of its parents, and the parents keep sending reply messages. This process propagates down the whole graph, creates many duplicated messages. Experimental results show that the number of messages increases dramatically.

Another distributed asynchronous algorithm is given lately in [8], the *Asynchronous Aggregation Search - AAS*. This algorithm works in a similar way as *ABT*, except that consistent values of the partial solution are also included in `OK` messages. This mechanism helps in reducing number of backtracks. For problems with large variable domains, including consistent values produces long messages. Thus, *AAS* is more practical for problems with small variable domains.

A recently proposed algorithm, called the *Asynchronous Forward Checking - AFC* ([9]), belongs to the group of distributed synchronous algorithms. It is a generic backtracking algorithm combined with a look ahead heuristic by means of asynchronous forward checking messages. Agents assign their values for variables sequentially by having one current partial assignment shared among all agents. When a dead end is detected, the algorithm backtracks sequentially following the reverse ordering. A strength of this algorithm is in its algorithmic simplicity and good computational efficiency, inherited from centralized algorithms. It has been shown to provide better performance, in terms of number of messages and constraint checks, than asynchronous algorithms *ABT* and *DiDB* ([9]). The main drawback of *AFC* is that it does not exploit concurrency: at any time, there is only either one `AFC` or one `BT` message that is exchanged between the agents, results in long running time compared to asynchronous algorithms.

3 Preliminaries

Classically, Constraint Satisfaction Problems (CSP) have been defined for problems in centralized architectures. A finite CSP is defined by a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where

- $\mathcal{X} = \{x_1, ..., x_n\}$ is the set of n variables.
- $\mathcal{D} = \{D_1, ..., D_n\}$ is the set of n finite, discrete domains of variables $x_1, ..., x_n$, respectively.
- $\mathcal{C} = \{C_1, ..., C_k\}$ is the set of k constraints on the variables. These constraints give the allowed values that the variables can simultaneously take. $\mathsf{var}(C_i)$ is the set of variables that are constrained by C_i.

A *solution* to a CSP is an assignment of values taken from the domains to all variables such that all the constraints are satisfied. Constraint satisfaction is NP-complete in general, and it is typically solved by a tree-search procedure with backtracking.

A distributed CSP (DisCSP) is a CSP in which the variables and constraints are distributed among a network of automated agents. Formally, a finite DisCSP is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are the same as in centralized CSP, and

- $\mathcal{A} = \{A_1, ..., A_p\}$ is the set of p agents
- $\phi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps variables to agents

Solving a DisCSP is to find an assignment of values to variables by the collective and coordinated action of automated agents. A *solution* to a DisCSP is a compound assignment of values to all variables such that all constraints are satisfied.

In DisCSP, agents communicate with each other by sending messages. We make the following assumptions for the communication model similar to those proposed in [4]:

1. An agent can send messages to other agents iff the agent knows the addresses of the agents.
2. The delay in delivering a message is finite but random; there is no message lost.

The second assumption has been partially relaxed from the original one in [4] that also assumes that messages are received in the order in which they were sent. Some algorithms (*ABT*, *DiDB*) require this assumption to be complete. Furthermore, for simplicity and without loss of generality, we assume that:

1. ϕ is a one-to-one function; it means that each agent holds only one variable; and there are no intra-agent constraints. (In DisCSP, it is assumed that intra-agent variables/constraints can be solved efficiently by some centralized algorithm. Distributed algorithms are to focus on the *cooperative* solving techniques between *distributed* solvers (e.g. agents).)

2. $\mathcal{C}$ are binary constraints so that $\mathsf{var}(C_i) = 2$, and every constraint is known by both agents involved in the constraint.

By these assumptions, the constraint network is simplified to a constraint graph where agents represent graph nodes and constraints represent graph edges.

4 The Algorithm DDBJ

The *Dynamic Distributed BackJumping - DDBJ*, is a complete, distributed, semi-asynchronous version of a graph-based backjumping algorithm which was previously introduced in centralized CSP ([2]). The algorithm combines the concurrency of an asynchronous dynamic backjumping algorithm and the computational efficiency of the synchronous *AFC* algorithm ([9]), coupled with the heuristics of dynamic value and variable ordering.

The Distributed BackJumping procedure

Agents perform value assignments in two phases:

- *Advancing forward* phase: which occurs when a new assignment tuple is added to the current partial solution.
- *Backjumping (backward)* phase: which occurs when an agent encounters a conflict. The process is "jumped back" to the culprit agent.

An agent is either in a *forward* phase or a *backward* phase. Algorithmically, the *forward* phase is performed sequentially: the assigning agent sends an `OK` to the next agent and `FC` messages to unassigned connected agents (similarly to *AFC* algorithm). If an agent detects a conflict when receiving some `OK`/`FC` message, it performs the *backward* phase asynchronously to backjump to the culprit agent, and also sends `NG` messages to unassigned agents. At any time, there can be several culprit agents detected and thus several backjumps are performed simultaneously. The culprit agents will change their values, hence the current partial solution (CPS), and perform the *forward* phase, without synchronizing with other agents nor waiting for other agents to switch phases. Consequently, at any time, agents are performing the *forward* and *backward* phases simultaneously in parallel without any synchronous control.

An example of algorithm execution is illustrated in Fig.1. At time *t1*, agent `A3` sends one `OK` message to `A4` (solid lines) and `FC` messages to connected agents (dotted lines). At a later time *t2*, `A11` finds a conflict and backjumps to `A3` by a `BT` message (dashed lines) and sends `NG` messages to others (not shown). At the same time, `A3`'s assignment has already propagated down to `A6` and `A7`, and get backjumped at `A6` to `A4` and backtracked at `A7` to `A5`. However, the asynchronous executions at `A6` and `A7` and the consequent ones will soon be overwritten by the new assignment at `A3`. These execution flows are carried out simultaneously.

In *AFC*, backtracking is performed sequentially (or synchronously) from the detecting agent to the culprit. At any time, there is only either one `OK` or one `BT` message being sent. In *DDBJ*, any agent who receives an `OK` or `FC` message can initiate a backjump.

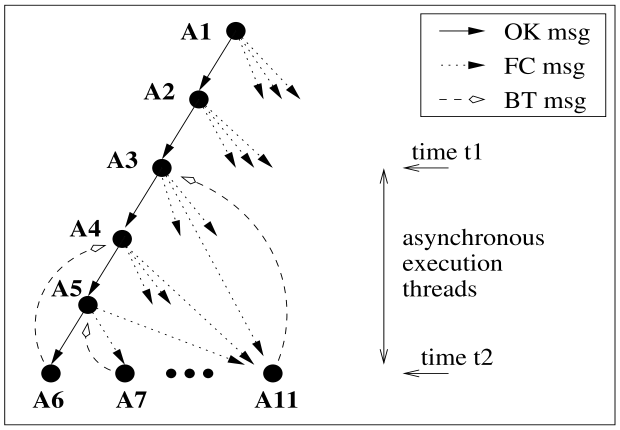

Fig. 1. An example of the *DDBJ* algorithm execution

Thus, there can be several `OK` and `BT` messages exchanged simultaneously, generating multiple asynchronous execution threads. However, there is only one `OK` message which may potentially lead to a solution (the most updated one or equivalently the one on the highest level of the search tree). The other `OK` messages will continue to propagate and create the assignment chains down the search tree, until only when the `NG` or newer messages arrive. Usually, it takes some cycles to stop these obsolete processes, depending on the size of the network, the connectivity density, the message delivering delay, etc.

The *DDBJ* algorithm is executed on every agent. Each maintains current value assignments of other agents in an $AgentView$ ([3]). We also adopt the $AgentView.consistent$ from [6] to represent whether the CPS it holds is consistent. To determine which `OK` message is the most updated one and to discard obsolete messages, we introduce for each agent a time flag called $TimeStamp$ which is incremented by 1 when the agent changes its value. When sending `OK`/`FC` messages, an agent includes its $TimeStamp$ with its assignment. The receiving agent checks the attached $TimeStamp$s and updates its context only if the message is valid. In the example above, by the $TimeStamps$, `A4`'s new assignment (due to `A6`'s backjump) will overwrite executions from `A5` (due to `A7`'s backtrack); however the new `A3`'s assignment (due to `A11`'s backjump) will eventually overwrite all executions below it.

The Dynamic Value and Variable Ordering Heuristics

The *DDBJ* algorithm uses dynamic value and variable ordering heuristics. Each agent keeps a potential conflict counter list of its domain values, and a potential conflict counter list of other agents (variables). An agent chooses the value which has the lowest counter value to assigns its variable, and sends the `OK` message (which contains the partial solution) to the agent (variable) which has the highest counter value (and `FC` messages to other linked agents). If there is a tie, the agent can use the chronological order. At start, all the counter values are equally zeros.

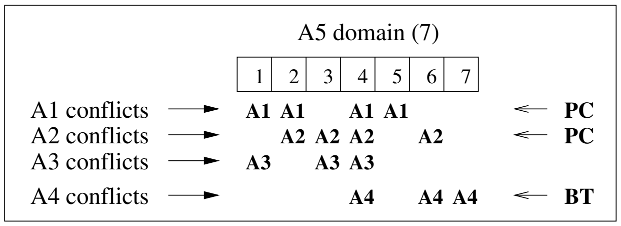

Fig. 2. An example of the heuristics: Agent A5 comes to a dead end, sends a BT message to culprit agent A4, sends "potential conflict" - PC messages to A1, A2

When a dead end is detected by an agent, the dead end discovering (DED) agent performs updating its priority lists in two steps. In the first step, it decreases the counter of the culprit agent (the agent whose value causes the dead end), then it sends the BT message to the culprit agent. The culprit agent, upon receiving the BT message, increases the counter of the sender (the DED agent) and increases the counter of its value that causes the backtrack, then it follows the backjumping procedure. In second step, the DED agent determines its "potential conflicting agents" (PC agents). A PC agent is the first agent whose value conflicts with a value in the domain of the DED agent. The DED agent increases the counters of the PC agents, sends a "potential conflict" - PC message to the PC agents. The PC agents, after receiving the PC message, increase the counters of their values (that cause the dead end), increase the counter of the DED agent. The idea here is to give more priority to the agents at higher top level of the search tree to change their values. The heuristics of dynamic ordering of value and variable would intuitively help to avoid thrashing on values selected by the very first agents and improve the ordering of agents.

An example is shown in Fig.2 to illustrate how the heuristics work. Agent A5 has 7 values in its domain. The value of agent A1 conflicts with the values (value id) 1, 2, 4, 5 of agent A5, thus these values are removed from the available values of agent A5. The value of agent A2 conflicts with the values 2, 3, 4, 6. The value of agent A3 conflicts with the values 1, 3, 4. The value of agent A4 conflicts with the values 4, 6, 7 where the value 7 is the last available value in the domain of agent A5. Thus A4 is the culprit agent with respect to agent A5. Following the first step, agent A5 increases the counter of agent A4, sends a BT to agent A4. Agent A4, upon receiving the BT, increases the counter of agent A5 and increases the counter of its corresponding value.

In the second step, agent A5 determines that A1 and A2 are the PC agents, as they are first agents who remove the values 1, 2, 3, 4, 5, 6 from its domain. Agent A3 is not a PC agent, since its value conflicts with the values 1, 3, 4 of A5 that have been removed previously by A1, A2. Thus, agent A5 increases the counters of A1 and A2, sends PC messages to A1 and A2. A1 and A2, when receive the PC message, increase the counter of A5 and also the counter of their corresponding value.

Detailed Algorithm Description

The *DDBJ* algorithm uses 8 types of messages as follows:

1. SUCCESS: a *termination* message which is broadcasted to all agents, by the last assigned agent, when a solution has been found.
2. FAILURE: a *termination* message which is broadcasted to all agents, by the first agent, when it has determined the problem has no solution.
3. ERROR: a *termination* message which is broadcasted to all agents when the algorithm encounters error (e.g. exceeded limit of time/resources).
4. OK: a message which contains the current partial solution (CPS) composed of a list of $(variable, value)$ tuples and their associate $TimeStamp$'s. This message is sent to the next agent according to the sending agent's decision of ordering.
5. FC: a message which contains a copy of OK message. This message is sent by the assigning agent to the linked agents that have not been assigned, according to its $AgentView$.
6. NG: a message which contains a nogood partial solution. It is sent to the linked agents that have not been assigned, according to its $AgentView$.
7. BT: a message which contains a nogood partial solution. It is sent back to the culprit agent (the last agent in the nogood partial solution).
8. PC: a message which contains a nogood partial solution. It is sent to potential conflicting agents determined by the agent when a conflict occurs.

The *DDBJ* algorithm is executed simultaneously on all agents in parallel. An appropriate function is called depending on the type of the received message. At start, an empty OK message is sent to the first agent for initialization.

Upon receiving an OK message, function receiveOK() is executed. It first checks if the message is valid (line 1); otherwise, it is older than, or equally timely to, the stored $TimeStamp$s [3] and discarded. Next, $TimeStamp$s get updated (line 2). It then checks whether the message's partial solution (MPS) contains *the previously determined nogood* (meaning current $AgentView.consistent$ = $false$ and the MPS contains $AgentView$). If it is the case, the agent simply does nothing and returns (line 3,4). Otherwise, it updates its context by the MPS (line 6). If the update succeeds, meaning its consistent domain of values is not empty, the agent assigns the value (line 8). Otherwise, it backtracks to the last assigned agent (line 10).

Function receiveFC() is called when an FC message is received. The agent checks and discards obsolete message (line 1), otherwise updates its $TimeStamp$s (line 2). It then checks whether the message does not contain the previously determined nogood. If it is the case, it resets the consistency state to $true$ (line 3,4). Whenever the consistency state is $true$ (line 5), the agent updates its context (line 6). If the update does not succeed, it does the following: sending NG messages to linked agents that are not assigned, sending PC messages to the determined PCAs, updating its memory of PCAs and backjumping to the culprit agent.

When receiving an NG message, the function receiveNG() checks to see if $AgentView$ contains the MPS. If it is the case, it removes last one or more tuples in its $AgentView$ to be the same as the received nogood, restores the values accordingly (which are associate with those tuples) (line 2) and resets the consistency state (line 3). Otherwise, if the message is newer than its $AgentView$, the agent updates

[3] the latter happens when the agent has already received an NG message which contains the same time flag

```
procedure receiveOK()
 1: if $Msg$ is newer than $AgentView$ then
 2:     update $TimeStamp$s
 3:     if previously determined nogood then
 4:         return
 5:     set $AgentView.consistent = true$
 6:     updateDomain(MPS)
 7:     if success then
 8:         assignVal()
 9:     else
10:         backJump($previous$)
end
procedure receiveFC()
 1: if $Msg$ is newer than $AgentView$ then
 2:     update $TimeStamp$s
 3:     if not previously determined nogood then
 4:         set $AgentView.consistent = true$
 5:     if $AgentView.consistent$ then
 6:         updateDomain(MPS)
 7:         if not success then
 8:             update PCA
 9:             send NG to unassigned agents; PC to agents in PCA
10:             backJump($culprit$)
end
procedure receiveNG()
 1: if $AgentView$ orderly contains $Msg$ then
 2:     restoreDom()
 3:     set $AgentView.consistent = false$
 4: else if $Msg$ is newer than $AgentView$ then
 5:     set $AgentView.consistent = false$
 6:     update $TimeStamp$s
 7:     updateDomain(MPS-$last$)
 8:     if not success then
 9:         update PCA
10:         send NG to unassigned agents; PC to agents in PCA
11:         backJump($culprit$)
12: if $self$ is assigned then
13:     reset to unassigned
end
```

its context (line 5,6,7). If the update does not succeed, it functions similarly to function receiveFC(). In both cases, if the agent is an assigned agent, it has to reset itself unassigned (line 11,12).

Function receivePC() simply updates the agent's memory of PCAs and value priority. Function receiveBT(), when a BT message is received, first updates the memory of PCAs and value priority (line 1,2). It then finds the next available value, by calling function assignVal(). Note that it has to check if the message is still valid (meaning that

```
procedure receivePC()
 1: update value priority / PCA
end
procedure receiveBT()
 1: update value priority / PCA
 2: if $self$ is assigned then
 3:     if my $AgentView$ is NOT newer $Msg$ then
 4:         assignVal()
end
procedure assignVal()
 1: findNextVal()
 2: if found a consistent value then
 3:     Increase $TimeStamp$
 4:     if $self$ is last agent then
 5:         broadcast SUCCESS to all agents
 6:     else
 7:         send OK to next agent; FC to connected agents
 8: else
 9:     backJump($previous$)
end
procedure backJump($AgentIndex$)
 1: if $self$ is first agent then
 2:     broadcast FAILURE to all agents
 3: else
 4:     set $AgentView.consistent$ = $false$
 5:     reset to unassigned
 6:     send BT to agent $AgentIndex$
 7:     update PCA
end
```

its variable is assigned and the message is not too old), (line 3,4,5), since several BT messages can be sent simultaneously to the agent, and some have already arrived and been processed.

Function assignVal() tries to find a next consistent value (line 1), forwards the CPS to the next agent (line 7), otherwise it backtracks (line 9). Function backJump($AgentIndex$) performs the backjumping by resetting the agent context and sending BT message to agent $AgentIndex$. Function updateDomain(MPS) simply updates its value domain, $AgentView$ with the input MPS. As soon as it finds the domain empty, the function returns the detected nogood.

5 Soundness, Completeness and Termination

The argument for soundness is close to the one given in [9]. The fact that agents only forward consistent assignments in OK messages at only one place in function assignVal(), line 7, implies that the receiving agents receive only consistent assignments. A solution is reported by the last agent only in function assignVal() at line 5. At this point, all the agents have assigned their variables, and the assignments are consistent. Thus the algorithm is sound.

For completeness, we need to show that *DDBJ* is able to produce all solutions and terminate. The algorithm only backtracks, by sending `BT` messages, in function backJump(), which implements the graph-based backjumping. It has been shown in [10] that graph-based backjumping only makes *safe jumps*. In other words, the algorithm backjumps to the culprit variable, and this jump does not lead to missing any solution. Similarly in *DDBJ*, multiple *safe jumps* may be performed at the same time simultaneously which are caused by different culprits detected by different agents. The re-assignments of the culprit agents then happen simultaneously. However, the one with the highest level in the search hierarchy tree will eventually replace all others. Thus the algorithm performs an exhaustive search and is able to produce all solutions. Hence, it is complete.

In each backtrack step, there is at least one value of a variable that is removed (line 5 in backJump()). The fact that the domains of variables are finite implies finite number of backtracks, or `BT` messages, until `FAILURE` messages are broadcasted (line 2 in backJump()). Similarly, each `OK` message (only sent in assignVal(), line 7) increases the number of assigned variables by 1, until the last variable where `SUCCESS` messages are broadcasted. Therefore, the algorithm terminates.

In *DDBJ*, agents do not have to store nogoods. An agent has to keep only the current $AgentView$ and the associated $TimeStamp$'s, which have at most n elements. In addition, an agent also needs to maintain two priority lists of its value domain and other agents. Thus, the algorithm's spatial complexity is linear.

6 Experimental Results

This section gives an experimental evaluation of our algorithm *DDBJ* in comparison with two other well known algorithms, the distributed asynchronous algorithm - *DiDB* ([7]) and the distributed synchronous algorithm - *AFC* ([9]). *DDBJ* is tested in 2 versions: one version is without the dynamic ordering heuristics, called *DBJ*, to measure the performance of the semi-asynchronous backjumping procedure itself, and the other version is the full *DDBJ* algorithm.

The algorithms are tested on distributed binary CSPs which are randomly generated using the problem generator *JavaCSP* ([11]). The problems are generated based on 4 setting parameters:

- v - The number of variables (or number of agents),
- d - The number of values in the domain of each variable (domain size),
- c - The constraint density (which reflects the number of constraints), and
- t - The constraint tightness (which refers to the number of value pairs which are disallowed by the constraint).

These settings are commonly used in experimental evaluation of CSP algorithms ([12, 13, 9]). The problem generator has the ability to generate only feasible problem instances (having solutions). Thus, it is advantage to generate only feasible problem instances for problems in transition phase which are most hardest to solve and so it is easy to highlight differences in algorithm performance ([4]). Note that the problem instances are generated with the setting parameters applied globally, not by interleaving of independent subproblems.

We recall the distinction between Distributed Systems and Distributed Computing ([4]). The latter is belong to the research field of High Performance Computing, where the problem is to divide/distribute, in a efficient way, some computation load onto several connected (or distributed) computing machines. The efficiency is then defined as $speedup/N$ where N is the number of distributed machines ([14]).

In this work, we are concerning the former case, Distributed Systems, where the problems in question have their distributed characteristics in nature: they are spread over a number of distributed agents. As in [4, 7, 9, 6], we use the following measures as the criteria for evaluation:

- *Number of cycles* (or running time): to estimate the algorithm concurrency / asynchrony, as used in [3].
- *Number of messages*: to estimate the overhead of the algorithm affecting on the distributed environment, where the cost of sending messages is usually considered being more expensive than local computation of agents ([9]).
- *Number of constraint checks*: to evaluate computational efforts done locally.
- *Number of value assignments*: to represent the cost of value changes committed that may be high in some applications.

The first two measures are the most important factors in measuring the efficiency of distributed algorithms. The number of cycles indicates the running time of an algorithm. More importantly, it shows how much parallelism is exploited in asynchronous algorithms compared to synchronous ones. The notion of "concurrent checks" is discussed in [15]. In this work, we make an assumption that the constraints are simple so that an agent is able to process incoming messages, perform necessary constraint checks and send out messages in one clock cycle ([3]). Thus, the ratio "N.Constraint checks/N.Cycles" gives a good estimate of the average number of concurrent constraint checks. As argued in [16], synchronous distributed algorithms usually have better efficiency than asynchronous ones (in terms of overheads, redundant efforts, etc.), but asynchronous algorithms can exploit concurrency, thus resulting in better running time (or less number of running cycles).

The messages are set up to be delivered to destination *not necessarily in the order in which they were sent*, except for the algorithm *DiDB* where it requires the messages are delivered in order. The number of messages is an important measure for DisCSP algorithms, since in distributed environment, sending messages to other distributed agents is considered expensive ([4]).

To simulate a distributed environment and asynchronous execution, we use a discrete event simulator. We have a global discrete clock counting in cycles to simulate a real time clock. At each cycle, all agents read the incoming messages, process the computation and send out messages to other agents. If there is no incoming message, an agent simply sits idle. We recall the assumption that an agent is able to process incoming messages, perform necessary constraint checks and send out messages in one clock cycle. The algorithm is executed simultaneously in parallel on all agents. All agents terminate when an termination message is broadcasted and the algorithm finishes. The

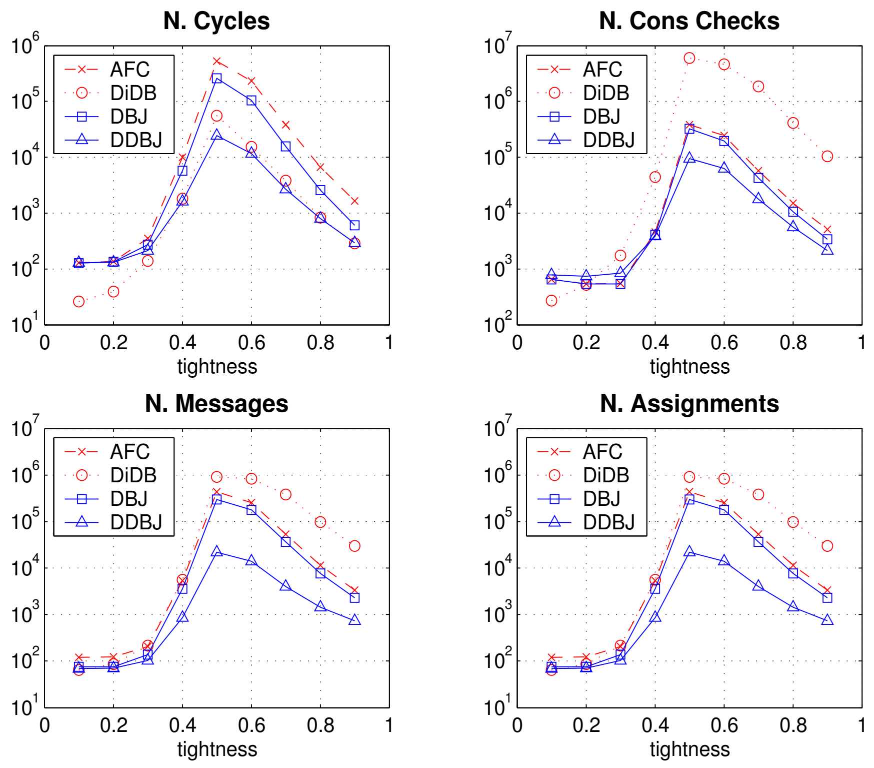

Fig. 3. Results (in *log10* scale) for N.vars v=15, domain d=15, density c=0.5. At transition phase when tightness $t = 0.5 - 0.7$, *DiDB* solved $50\% - 80\%$, *AFC*, *DBJ* and *DDBJ* solved 100% of 100 generated instances

algorithm's running time is counted as the number of global clock cycles. Furthermore, to simulate the real distributed environment as close as possible, we set up the link channels between agents such that the delivery time is randomly generated between 1 and the total number of agents, which best reflects the effect of the size of the constraint network. Because the concurrency of computation of asynchronous algorithms is difficult to see from other measurements (number of constraint checks, number of messages), this setting helps to differentiate asynchronous and synchronous (or sequential) execution schema. The same argument for comparing algorithms is also pointed out in [15].

Because of limited space, the results of 2 test sets are presented. The first test set includes problems with the number of variables $n = 15$, the variable domain $d = 15$, the constraint density probability $c = 0.5$ and the constraint tightness varying from 0.1 to 0.9 in 0.1 steps. The results in *log10* scale are shown in Figure 3. Each plot point is the average of results taken from 100 randomly generated instances. An algorithm is stopped when the number of running cycle reaches a limit of $10,000,000$ cycles or the number of messages sent *in one cycle* exceeds $100,000$.

In term of running time, *DBJ* is about 2-4 times faster than *AFC* at transition phase. The difference indicates the concurrency effect of the asynchronous backward phase of *DBJ*. *DiDB*, because of its fully asynchronous nature, is better than *DBJ* and *AFC*. However, when combined with the dynamic ordering heuristics, *DDBJ* is the best algorithm among the four for most cases.

On number of messages, *DDBJ* is better than the other three algorithms by a factor of one order approximately. The only drawback is that the message `OK` of *DDBJ* (and *AFC*, *DBJ*) is longer than that of *DiDB*. However, since the number of elements in a message is at most equal to the number of variables n and each element contains agent id, value id and its associate $Timestamp$, that all can be represented by 3 integer numbers, the size of a message is not more than $3n$ integer numbers.

In term of computational performance, *DDBJ* outperforms both algorithms *DiDB* and *AFC* by a factor of 5 to 100 on hard instances, where *DBJ* comes next. This can be explained by the fact that by combining good value/variable ordering heuristics and exploiting concurrency, it also helps to increase the algorithm's computational efficiency and reduces the number of messages. Note that the synchronous algorithm *AFC* always performs better than the fully asynchronous algorithm *DiDB*, that it agrees with the result obtained in [9].

In more details, at transition phase where problems are hardest to solve (constraint tightness is between 0.5 and 0.7), *DiDB* is only able to solve $50\% - 80\%$ of the generated problem instances: we stop the algorithm when the number of messages sent in one cycle exceeds the limit of $100,000$ messages, since most of the time and memory resources are consumed by processing duplicated messages. This message duplication problem arises significantly when the messages are delivered with some random delay. The other three algorithms are able to solve all the problems within the limits of running cycles and messages.

In the second test set, we evaluate the algorithms by 4 feasible, high dimension problems, with the number of variables equals 20, 30, 30 and 40, respectively. The constraint tightness is set to a value close to 0.5 so that the problems are in the transition phase. The limit of number of cycles is now set to $100,000,000$. We exclude *DiDB* because of its limited capacity of solving high dimension problems: the number of messages explodes exponentially so that after a few hundred running cycles, the number of messages soon exceeds the limit of available resource. The results in *log10* scale are shown in Figure 4. The percentages show the numbers of problems solved by the algorithms. Each subgraph shows the median value of the results of 50 generated instances. The reason of taking the median value instead of the mean value is that in the transition phase, the variance of the results is too high, thus the median value indicates better the result average.

It is clear that the semi-asynchronous algorithm *DBJ* always performs better than *AFC* by a factor of 2 or more. It shows the effect of the asynchronous backjumping phase on the algorithm efficiency. *DDBJ* outperforms both the others by a factor of one to two orders for all measures. On the number of problems solved, *DDBJ* is able to solve all the problem instances for the 4 cases within the time limit, where the other two algorithms can not. This measure again confirms the high efficiency of the heuristics used in *DDBJ*. For the last two problems where the numbers of variables are 30 and 40,

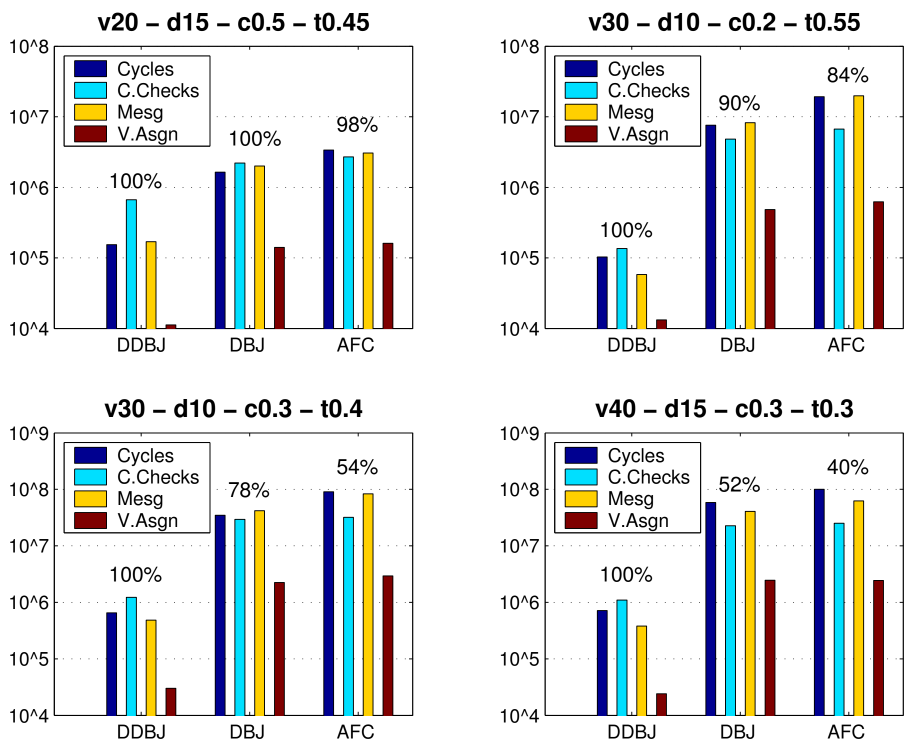

Fig. 4. Results (in *log10* scale) of feasible, high dimension problems. The percentages represent the number of problems solved within a time limit.
a) N.vars v=20, domain d=15, density c=0.5, tightness t=0.45
b) N.vars v=30, domain d=10, density c=0.2, tightness t=0.55
c) N.vars v=30, domain d=10, density c=0.3, tightness t=0.4
d) N.vars v=40, domain d=15, density c=0.2, tightness t=0.4

AFC is able to solve only 54% and 40% of the instances. The performance measures of *AFC* are at least one order behind those of *DDBJ*. These factors will be larger if we increase the running time limit for *AFC* to solve more instances.

One can also notice that as the number of variables increases, the performance difference between *DDBJ* and the other algorithms increases. When v=15, *DDBJ* is faster by about one order of magnitude, when v=30,40, *DDBJ* outperforms the others by about two orders of magnitude on number of running cycles and number of messages.

7 Conclusion

A new complete, distributed, semi-asynchronous algorithm, *DDBJ*, is presented. The algorithm adopts a sequentially assigning procedure, an asynchronous forward checking scheme in its *advancing phase* and an asynchronous graph-based safe-backjumping scheme in its *backjumping phase*. The sequentiality of variable assignment enables *DDBJ* to integrate the powerful heuristics of dynamic value and variable ordering and still easily to control the algorithm completeness. Experimental results show that the *DDBJ* algorithm outperforms the *DiDB* and the *AFC* algorithms by a factor of *one to two* orders of magnitude on hard instances of randomly generated DisCSPs, both on concurrent running time, number of messages and on other measures of number of constraint checks, number of variable assignments.

Acknowledgments

We would like to thank Prof. Amnon Meisels for his visiting presentation on the *AFC* algorithm. We also thank Arnold Maestre, Dr. Christian Bessière for their helpful explication of the *DiDB* algorithm. Many thanks to Dr. Bart Craenen for his problem generator *JavaCSP*. This work was performed at the Artificial Intelligence Laboratory, Ecole Polytechnique Fédérale de Lausanne and was sponsored by project `COCONUT` under contract number IST-2000-26063.

References

1. Yokoo, M., Hirayama, K.: Algorithms for Distributed Constraint Satisfaction: A Review. In: Proceedings of Autonomous Agents and Multi-Agent Systems. (2000)
2. Dechter, R.: Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. Artificial Intelligence **41(3)** (1990) 273–312
3. Yokoo, M., Durfee, E., Ishida, T.: Distributed constraint satisfaction for formalizing distributed problem solving. In: Proceedings DCS. (1992)
4. Yokoo, M.: Distributed Constraint Satisfaction. Springer-Verlag (2001)
5. Ginsberg, M.: Dynamic Backtracking. Journal of Artificial Intelligence Research **1** (1993) 25–46
6. Hamadi, Y.: Interleaved backtracking in distributed constraint networks. International Journal on Artificial Intelligence Tools **11** (2002) 167–188
7. Bessière, C., Maestre, A., Meseguer, P.: Distributed Dynamic Backtracking. In: Proceedings of the IJCAI'01 workshop on Distributed Constraint Reasoning. (2001)
8. Silaghi, M., Sam-Haroud, D., Faltings, B.: Asynchronous Search with Aggregations. In: Proceedings AAAI'00. (2000)
9. Meisels, A., Zivan, R.: Asynchronous Forward-checking on DisCSPs. In: Proceedings of the Workshop on Distributed Constraints (DCR-03), Acapulco, August 2003. (2003)
10. Dechter, R., Frost, D.: Backtracking algorithms for constraint satisfaction problems - a tutorial survey. Technical report, University of California, Irvine (1998)
11. Craenen, B.: JavaCsp package. http://www.xs4all.nl/˜bcraenen/JavaCsp/ (2003)
12. Prosser, P.: Binary constraint satisfaction problems: some are harder than others. In: Proceedings of the 11th European Conference on Artificial Intelligence - ECAI'94. (1994)
13. Bessiere, C.: Random Uniform CSP Generators. http://www.xs4all.nl/˜bessiere/generator.html (1996)
14. Dowd, K., Severance, C.: High Performance Computing. Second edn. O'Reilly & Associates (1998)
15. Meisels, A., Kaplansky, E., Razgon, I., Zivan, R.: Comparing performance of distributed constraints processing algorithms. In: Proceedings of the Workshop on Distributed Constraint Reasoning, in AAMAS-2002. (2002)
16. Barbosa, V.C.: An Introduction to Distributed Algorithms. The MIT Press (1996)