On the Evaluation of DisCSP Algorithms*

Ismel Brito, Fernando Herrero, and Pedro Meseguer

Institut d'Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain.
{ismel|fhcarron|pedro}@iiia.csic.es

Abstract. Not every research paper in DisCSP evaluates algorithms in the same way. Motivated by this fact, we revise some elements of the area of distributed algorithms as well as distributed constraints, which can help to develop a well-founded methodology for evaluation of DisCSP algorithms. Although preliminary, we suggest a number of points which should be considered in such methodology.

1 Introduction

In this paper we aim at collecting a number of thoughts and ideas about the task of how evaluate algorithms for solving DisCSP. As researchers on distributed constraint satisfaction, we often develop new versions of existing procedures, we devise new heuristics and we produce new solving algorithms. To assess the practical importance of these new developments, their evaluation is a crucial point. Facing this issue, we often consider questions like,

- what is the most adequate environment to test our algorithms?
- on which benchmarks should they been evaluated?
- which are the most adequate parameters to measure algorithmic efficiency?

Often, different research groups have different answers to these questions. Our goal is to achieve a consensus in the community of distributed constraint satisfaction, in order to establish a common accepted *methodology* on the way algorithms should be evaluated. Obviously, this methodology should follow standard methods in the area of *distributed algorithms* (see [4] for a comprehensive review of this area). In addition, since constraint solving is NP-complete, many solving algorithms have the same worst-case complexity. To really evaluate these algorithms in practice, we have to identify some parameters whose measure could give an idea of the amount of resources used in the algorithm execution. The methodology has to answer two types of questions. First, to define *what* parameters should be measured (total CPU time, number of cycles, concurrent constraint checks, number of messages exchanged, etc.). Second, to define *how* this can be measured, in a double sense: *on which environment* evaluation is performed (reality vs. simulation, several computers vs. one computer), and *on which benchmarks*

* This research is supported by the REPLI project TIC-2002-04470-C03-03.

(distributed random, distributed versions of existing CSP, specific DisCSP appplications, etc.). As a consequence, we expect that comparison among different approaches would be facilitated, and the value of scientific communication would be promoted.

In the following, we discuss some of these issues (the question of benchmarks is not considered) based on our experience. We strongly believe that other research groups can provide valuable ideas and suggestions, and we urge them to do so.

2 Preliminaries

There are several definitions of distributed constraint satisfaction problems. Without trying to be exhaustive, we think that all of them share the following idea. A *distributed constraint satisfaction problem* (DisCSP) is a CSP which is distributed among several agents. Each agent contains a part of the problem, but no agent contains the whole problem. Some overlapping may exist among agents, although no two of them can contain exactly the same initial information. Because some reasons (privacy, size, etc.), the information of each agent cannot be transferred into a central server, where the whole problem could be solved by classical, centralized CSP solving methods. In the distributed setting, the task is to find a solution of the problem (an assignment of all the variables satisfying all constraints), by exchanging messages among agents.

Depending on the model that we assume about the timing of events in the distributed system, we obtain different types of algorithms. In [4], three timing models are considered, which are informally described as follows:

1. *The synchronous model*. "This is the simplest model to describe, to program and to reason about. We assume that components (agents) take steps simultaneously, that is, that execution proceeds in synchronous rounds."
2. *The asynchronous model*. "We assume that separate components (agents) take steps in arbitrary order, at arbitrary relative speeds."
3. *The partially synchronous model*. "We assume some restrictions on the relative timing of events, but execution is not completely lock-step as it is in the synchronous model."

These three timing models generate three types of algorithms for DisCSP solving. Broadly speaking, a synchronous algorithm is based on the notion of *privilege*, a token that is passed among agents. Only one agent is active at any time, the one having the privilege, while the rest of agents are waiting [1]. When the process in the active agent terminates, it passes the privilege to another agent, which now becomes the active one. In an asynchronous algorithm every agent is active at any time, and it does not have to wait for any event. A partially synchronous algorithm in in between of these two types. An agent running a partially syncronous algorithm may require to wait for some special event, but not for every event.

To solve a DisCSP instance, the three types of algorithms differ in their functionality and efficiency. Considering functionality, asynchronous algorithms are the most general

[1] Except for special topological arrangements of the constraint graph. See [2] for a synchronous algorithm where several agents are active concurrently.

and portable, because they impose no assumptions on the timing of computation steps. Usually, they are more robust and offer more privacy than the other two types. Regarding efficiency, as the amount of resources required to compute a solution, there is some debate on which type of algorithm is more efficient. We come back on this issue in the Section 5.

3 Evaluation

Two complexity measures, on time and on communication, are proposed in [4] for distributed algorithms that exchange messages. *Time complexity* aims at bounding the time required to compute a global solution by the whole system. *Communication complexity* considers the amoung of network resources needed to achieve a solution.

3.1 Time Complexity

For synchronous algorithms, [4] proposes using the number of rounds required to find a solution as the time complexity measure. For asynchronous algorithms, [4] requires to have an upper bound on the time between succesive chances of a task to perform a step. This is called a timed execution. The time of the event is the supremum of the times that can be assigned to such an event in all timed executions. Since CSP solving is NP-complete, this worst-case expression is exponential and does not help in clarifying the relative efficiency of different algorithms.

Alternatively, [3] proposes a new measure of time complexity as counting the number of constraint checks that cannot be performed concurrently when solving a DisCSP. A constraint check occurs when a value tuple is checked against a constraint. In classical CSP it is considered an atomic operation, which has to be performed for (almost) all constraint algorithms, so the number of constraint checks is a good estimation of the search effort. Inspired in the logical clocks of Lamport [5], in [3] the number of concurrent constraint ckecks is computed as follows. Each agent keeps a counter of its own performed constraint checks, and every message that it sends contains the value of that counter (when it was sent). When the receiver gets that message, it updates its own counter to the maximum between its counter and the counter contained in the message. When the algorithm stops, the maximum of the counters is the total concurrent constraint checks, and approximates the size of the longest sequence of checks that cannot be done concurrently.

At the end of the search, the number of concurrent constraint checks performed approximates the runtime of the algorithm if it is assumed that the elapsed time between two constraint checks not performed concurrently is approximately the same. However, this assumption does not hold in presence of random delays or for partially synchronous algorithms with unbounded waiting episodes. In this last case, waiting episodes can be counted at agent level. Following a similar approach to concurrent constraint checks, we can assess the longest sequence of waiting episodes which cannot be performed concurrently.

Other measures can provide complementary information. For instance, the distribution of constraint checks really performed by agents in the network gives some idea of how balanced is search effort among agents.

3.2 Communication Complexity

For the three timing models considered, [4] considers the total number of messages exchanged as the measure of communication complexity. How messages are counted depends on the communication model used, described in Section 3.3. This is also the common position of the distributed constraints community.

The size of messages can also be taken into account as secondary measure, following [4]. The cost of sending a message is the cost of setting the communication link plus the cost of properly sending the message. The cost of setting the communication link is paid when the first message is sent through that link. The cost of properly sending a message depends on its length (the message size plus the header added by the communication software). So message size has to be considered, especially when comparing algorithms exchanging messages whose sizes differ in more than a constant.

Assuming the Unicast communication model (see Section 3.3), the idea of concurrent constraint checks can be applied to messages. Each agent keeps a counter of the sent messages, and every message contains the value of that counter when it was sent. When the receiver gets that message, it updates its own counter to the maximum between its counter and the counter contained in the message. We call this value concurrent messages, and gives an idea of the length of the longest sequence of messages that cannot be done concurrently.

Other measures can provide complementary information. For instance, the distribution of the number of messages sent/received by agents in the network gives some idea of how balanced is the communication among agents.

3.3 Communication Model

It is often the case that algorithm description and analysis do not consider the underlying communication model. However, a real-case study should take this into account, as the communication costs may vary depending on which model is used. We analyze two communication models:

1. *Unicast* (also called *send/receive* or *point-to-point* communication). On a unicast network, messages are sent one by one to each of the recepients, thus requiring linear resources on the number of agents. This is the common model used in experiments and simulations.
2. *Multicast*. On the other hand, advantage could be taken from multicast networks, such as IP networks, on which agents can subscribe to a group and messages sent to that group do not imply any additional cost per agent. This model provides constant time and resources, irrespective to the number of recepients.

Since this is an implementation issue, it makes sense to reflect which of these models was actually used when presenting experimental results. It is not uncommon to consider “broadcast” communication as a single process, when in fact the implementation means sending one message to each receiver.

4 Simulator

Ideally, to evaluate a new algorithm one should have n dedicated processors connected to a common network on which tests would be done. However, this setting is often not available in most of our labs. Even if there is a number of computers available, the workload of each computer and the load of the communication network are out of the control of the experimenter, and these aspects have a significant impact on the efficiency of the algorithms. Because of that, we consider that simulation into a single computer is a suitable alternative to make the tunning and most of the experimentation for DisCSP algorithms. After that, some algorithms can be tested on a real setting, assuming the resources needed to perform a field test. In the following, we consider the different options for DisCSP algorithms when are evaluated by simulation on a single computer.

Usually, DisCSP algorithms are described in terms of agents. An agent is an autonomous entity that contains a part of the problem, it is able to perform its own reasoning process and to communicate with other agents. In a multi-task computer (for instance, a desktop with Linux operating system (OS)), a direct option is to implement each agent as a different task, all having the same priority. The OS scheduler is in charge of activating / desactivating the agents, that take control of the CPU as any other task in the system. Communication among agents is performed using standard task communication facilities (usually implemented using disk storage). This approach is relatively simple to implement but present some drawbacks. First, it depends on the OS, so results obtained in computers with different OS could not be directly comparable. Second, even using the same computer and the same implementation, it is difficult to reproduce exactly the same results when repeating the same experiments. There are some sublte factors (such as the mail server, the network load, the disk storage) which change between executions and are out of the control of the experimenter. Because of that, exact reproduction of previous results is almost impossible with this approach.

To overcome this fact, an alternative is to use a simulator that offers the same facilities as the OS, but allows one complete control. This simulator allows agents to execute, performs the scheduling among agents and provides communication facilities. With this approach, results are reproducible, the same experiment generates the same results (providing random elements are initialized with the same seed).

The first simulator of this kind appears in the seminal work of Yokoo [6, 7]. Each agent keeps its own clock, which is incremented at each cycle of computation. One cycle for an agent consists of reading all its incoming messages, processing them and writing all messages generated as answers. It is assumed that a message sent at time t is available to the receiver at time $t + 1$. This means a kind of synchronicity in the activation of agents, which is somehow contradictory with the evaluation of asynchronous procedures. We come back on this point in Section 5.

Another scheduling policy is to activate agents randomly: a random number between 1 and n determines the identifier of the agent to activate. When this agent terminates, the same process selects the next agent to activate. This approach seems to be more adequate to evaluate asynchronous procedures. Other scheduling policies could offer some interesting alternatives.

5 Discussion

In this Section we contrast some of the criteria presented above with current practices in the evaluation of DisCSP algorithms. With this exercise we identify some aspects which could be improved in distributed algorithm evaluation.

5.1 Evaluation Parameters

Time and communication. Often we see DisCSP algorithms which are evaluated considering time or communication, but not both aspects. In general, we think that this approach provides incomplete information and does not allow one to assess globally the amount of resources needed for an algorithm. Following standard practice in distributed algorithms, we propose to use these two measures when evaluating DisCSP algorithms. Some researchers have suggested to aggregate both measures in one (or translate one measure into another). When possible, this approach is attractive because it allows us to deal with a single number. However, in many cases it cannot be done without making arbitrary assumptions, difficult to justify. In such cases, we suggest to keep both measures separated.

Timing model. Evaluating an algorithm should follow methods which are adequate for the timing model assumed by the algorithm. Synchronous algorithms can be evaluated using the number of rounds as time complexity measure. However, asynchronous algorithms should not be evaluated using methods that assume a synchronous model (such as the number of rounds).

An interesting question is the evaluation of partially synchronous algorithms, especially on those parts which require waiting for some event caused by other agents. During a waiting episode, an agent may not use its own resources but it is causing some delay to agents which require its input. Waiting episodes can be counted at agent level. In addition, following a similar approach to concurrent constraint checks, we can assess the longest sequence of waiting episodes which cannot be concurrent.

Communication model. Most of DisCSP papers does not deal explicitely with the communication model. It is usually assumed that when an agent sends a message to p other agents, this causes p physical messages in the network. In other words, the unicast model is implicitely assumed. This is fine, the only concern here is that the communication model should be made explicit, so algorithms could be evaluated using different models. This will bring closer the DisCSP paradigm to real communication networks, which finally could promote the use of DisCSP algorithms for practical applications.

Message size. When messages of different sizes are present in DisCSP algorithms, usually size differences are neglected and the number of messages is the only evaluation parameter considered. We believe that this is not a fair approach and the message size cannot be ignored, especially when message sizes differ in more than a constant (for instance, in a function that depends on problem dimensions). We suggest to take message size differences into account, as suggested in the area of distributed algorithms [4].

5.2 Processing Messages: One by One vs Packets

Asynchronous DisCSP algorithms are often described assuming that agents react inmediately after receiving a message: they process messages one by one. However, some algorithms are evaluated processing messages by packets: an agent reads all messages that are waiting in the input buffer and processes them as a whole. It is worth noting that these two strategies may produce quite different results considering the evaluation parameters described above.

The motivation of asynchronous algorithms for processing messages by packets, instead of one by one, is to prevent useless work. A simple example occurs when two consecutive messages arrive from the same agent, informing that it has taken two different values. Obviously, the first message becomes obsolete as soon as the second arrives. All the work generated by processing the first message and extra messages that this processing might be caused, could be saved if the agent would have known the second message. Somehow, this idea was mentioned in [7] and [8]. Recently, in [1] a formal protocol for processing messages by packets is proposed.

Informally, when any agent processes messages by packets, it first reads all messages that are in its input buffer. Then, it processes all read messages as a whole, ignoring those messages that become obsolete by the presence of another message. The agent looks for any consistent value after its agent view and its nogood store are updated with these incoming messages.

Thus, every outgoing message that an agent will send is consequence of the previous incoming messages because all of them update the agent view before agent checks consistency. Therefore, before agent looks for a consistent value, the agent's concurrent counter has to be updated to the maximum value between its own counter before starting to process the packet and the maximum of all concurrent counter of all messages contained in the processed packet.

Empirically, we have tested both types of message processing on distributed binary random problems using two algorithms: one asynchronous and one partial synchronous. The former is the well-known *ABT* algorithm [6, 7]. The latter is *ABT-Hyb*[1], an novel *ABT*-like algorithm which introduces some synchronization points to avoid sending redundant messages. It can be seen as a partially synchronous algorithm.

In our experiments, we have 16 variables/agents ($n = 16$) and 8 values per variable ($m = 8$). The connectivity of the network is set to 0.5 (p_1=0.5). On Table 1 and Table2 we report results averaged over 100 executions in terms of the following parameters:

- the sum of all constraint checks performed by all agents (cc)
- the number of concurrent constraint checks (ccc)
- the total number of messages exchanged ($mess$)
- the number of concurrent messages, computed in the same way as ccc ($cmess$)
- the total number of *Info* messages exchanged ($info$)
- the total number of *Back* messages exchanged ($back$)
- the total number of *Add-Link* messages exchanged ($link$)
- the number of *Back* messages that are obsolete when are received ($obso$)

Regarding the communication cost, the number of messages exchanged in both algorithms processing messages by packets is lower than processing messages one by

messages processing	cc	ccc	mess	cmess	info	back	link	obso
one by one	92,860	23,148	33,184	3,635	25,413	7,733	38	4,824
by packets	77,550	35,408	31,986	5,558	24,877	7,770	39	2,339

Table 1. Results in the pick of difficulty for *ABT* with both types of messages processing

messages processing	cc	ccc	mess	cmess	info	back	link	obso
one by one	57,364	22,720	24,107	4,250	19,720	4,437	37	1,567
by packets	56,680	22,603	23,963	4,229	19,660	4,303	67	1,525

Table 2. Results in the pick of difficulty for *ABT-Hyb* with both types of messages processing

one. Considering the number of concurrent constraint checks, processing messages by packets increases the number of concurrent constraint checks with respect to processing messages one by one. However, the number of obsolete messages decreases when agents process messages by packets. This phenomenon can be seen better if we compute the following ratios:

– ratio of concurrency of constraint checks,

$$rccc = 1 - \frac{ccc}{cc} \tag{1}$$

– ratio of concurrency of messages,

$$rcm = 1 - \frac{cmess}{mess} \tag{2}$$

– ratio of information quality of $Back$ messages,

$$riq = 1 - \frac{obso}{back} \tag{3}$$

The ratio $rccc$ can give us an idea of how concurrent is our algorithm. On contrast, ratio $rccc$ and ratio riq can help us to measure the use of the resources of the network. These parameters are easily extended to synchronous algorithms. In them, $rccc = 0$, $rcmess = 0$ and $riq = 1$ [2].

On Table 3 and Table 4 we show the results of computing these ratios to the experimental results reported on Table 1 and Table 2. Regarding *ABT*, we can see that it becomes less concurrent when messages are processed by packets, although the quality of the information is higher. Regarding *ABT-Hyb* when messages are processing by packets, the concurrency of the algorithm and the quality of the information remains approximately the same as processing messages one by one. This happens because an *ABT-Hyb* agent can be in a *waiting state* without sending any outgoing message. In that state, the agent receives all $Info$ messages updating its agent view accordingly.

[2] Except for special arrangements of the constraint graph, as described in [2]

messages processing	rccc	rcmess	riq
one by one	0.7507	0.8905	0.3757
by packets	0.5434	0.8262	0.6692

Table 3. Ratios for *ABT* algorithm with both types of messages processing.

messages processing	rccc	rcmess	riq
one by one	0.6039	0.8237	0.6395
by packets	0.6013	0.8235	0.6456

Table 4. Ratios for *ABT-Hyb* algorithm with both types of messages processing.

Then, when an agent leaves the *waiting state* it will have a better idea of the current assignments of the other agents.

Finally, it is worth noting that although concurrency decreases when processing messages by packets, this does not necessarity means that the process is less efficient. In fact, it saves some useless work. This is reflected in the increment of riq (ratio of information quality) of the *Back* messages and in the decrement of $rccc$ (ratio of concurrent constraint checks) and $rcmsg$ (concurrent messages).

6 Summary

We believe that the evaluation of current DisCSP algorithms is not completely established, and a common methodology is badly needed. Such methodology should follow standard evaluation methods in distributed algorithms. We have reviewed some basic elements of this area, such as the timing model, the communication model, time and communication complexities. We have also considered evaluation procedures suggested from the distributed constraint community. We have tried to apply them to the evaluation of DisCSP algorithms. Doing this exercise, we have identified some points which should be followed in the evaluation of DisCSP algorithms. These results can be seen as preliminary. More work is needed to achieve a global and coherent methodology for the evaluation of DisCSP algorithms.

References

1. Brito I., Meseguer P. Synchronous, asynchronnous and hybrids algorithms for DisCSP. Submitted to *CP-04 Workshop on Distributed Constraint Reasoning*.
2. Collin Z., Dechter R., Shmuel K. On the Feasibility of Distributed Constraint Satisfaction. *In Proc. of the 12th International Joint Conference on Artificial Intelligence, IJCAI-91*, 318–324, 1991.
3. Meisels A., Kaplansky E., Razgon I., Zivan R. Comparing Performance of Distributed Constraint Processing Algorithms. *AAMAS-02 Workshop on Distributed Constraint Reasoning*, 86–93, Bologna, Italy, 2002.
4. Lynch N. *Distributed Algorithms*, Morgan–Kaufmann, 1996.

5. Lamport L. Time, Clock, and the Ordering of Evens in a Distributed System. *Communications of the ACM*, **21(7)**, 558–565, 1978.
6. Yokoo M., Durfee E.H., Ishida T., Kuwabara K. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. *In Proc. of the 12th International Conference on Distributed Computing System*, 614–621, 1992.
7. Yokoo M., Durfee E.H., Ishida T., Kuwabara K. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Knowledge and Data Engineering* **10**, 673–685, 1998.
8. Zivan, R. and Meisels, A. *Synchronous and Asynchronous Search on DisCSPs*. In Proc. of EUMAS-2003, Oxford, UK, 2003