Generic run-time measurement for DisCSPs search algorithms[1]

Roie Zivan and Amnon Meisels [2]

Abstract.
The efficiency of Distributed Constraints satisfaction algorithms must be measured by concurrent performance measures. In Previous studies we have proposed the count of non concurrent logical steps as an asynchronous measure. This measure is independent of the specific implementation and is not affected by the level of concurrency of the performing algorithm. The method of counting non-concurrent constraints-checks (NCCCs) was later extended to the case of systems with message delays. The method inherently assumes that each message received by an agent triggers some computation.

In some DisCSP algorithms, such as AWC and ABT_DO, agents receive assignment messages from lower priority agents which do not cause a computation. The present paper proposes a generalization of the method of counting non-concurrent logical steps as run-time measure for distributed search algorithms. Logical time counters carried by messages do not immediately update the receiving agent's logical counter. Instead, the agent stores the data carried by the message and tags it with the logical time carried by the message. Only when the agent uses the stored information for the first time, the logical counter of the delivering message is considered.

The proposed method ensures that the reported solution does not include logical computation steps that could have been performed concurrently. The proposed general method is presented in detail, demonstrated on the relevant DisCSP algorithms and its correctness is proven.

Key words: Distributed Constraint Satisfaction, Search, Distributed AI.

1 Introduction

Distributed constraints satisfaction problems (*DisCSPs*) are composed of agents, each holding its local constraints network, that are connected by constraints among variables of different agents. Agents assign values to variables, attempting to generate a locally consistent assignment that is also consistent with all constraints between agents (cf. [9, 8]). Agents check the value assignments to their variables for local consistency and exchange messages among them, to check consistency of their proposed assignments against constraints with variables that belong to different agents [9, 1].

Search algorithms on DisCSPs are run concurrently by all agents and their performance must be measured in terms of distributed computation. Two measures are commonly used to evaluate distributed algorithms - run time, which measures the computational effort and network load [4]. The run time performance of search algorithms on DisCSPs has traditionally been measured by the number of computation cycles or steps (cf. [9]). In order to take into account the effort an agent makes during its local assignment the computational effort can be measured by the number of constraints checks that agents perform. However, care must be taken to measure the *non-concurrent* constraints checks. In other words, count computational effort of concurrently running agents *only once* during each concurrent running instance ([5, 7, 14]). Measuring the network load poses a much simpler problem. Network load is generally measured by counting the total number of messages sent during search [4].

The first attempts to compare run times of distributed search algorithms on DisCSPs used a synchronous simulator and instantaneous message arrival. During one step of computation (cycle) of the simulator, all messages of all agents are delivered and all resulting computations by the receiving agents are completed [9]. The number of these synchronous steps of computation in a standard simulator served to measure the non-concurrent run-time of a DisCSP algorithm [9]. It is clear that the comparison of asynchronous search algorithms by synchronizing them to run on a simulator is not satisfactory. In fact, comparing concurrent run-times of distributed computations must involve some type of asynchronous time considerations [3, 5, 14].

In an asynchronous environment, the task of measuring the run-time of an algorithm becomes more complicated. In order to determine the length of the sequence of consecutive constraints checks which were performed, the measurement method must determine which of the logical steps *could not have been performed concurrently*. A method for measuring non-concurrent logical computation steps was proposed in [5], for the case of optimal communication (i.e. no message delays) between agents. This measure was generalized in two later studies to the case of imperfect communication networks [14, 13] in order to study its effect on different distributed search algorithms.

In both of the methods proposed above, which are based on Lamport's clock synchronizing algorithm [3], each message carries the logical of its sending. The receiver of each message advances its logical clock to the clock carried by the message if it is larger than its own current clock. This method ensures that concurrently performed logical steps are not counted more than once [3, 5]. However, both of the methods assume that *every message must trigger a non-empty sequence of logic steps*. This assumption is compatible with most $DisCSP$ search algorithms (cf. [9, 10, 6]). In the general algorithmic case, agents are allowed to send messages that do not necessarily trigger computation. In such a case, the methods presented in [5] and [11] are not adequate.

The present paper proposes a generalized method for counting

[1] Supported by the Lynn and William Frankel center for Computer Sciences.
[2] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84-105, Israel email:{zivanr,am}@cs.bgu.ac.il

non-concurrent logical steps for measuring run-time of distributed search algorithms. In order to ensure that the reported count does not include logical computation steps that could have been performed concurrently, the logical time counters carried by agents are not updated indiscriminately by the receiving agent. Instead, the agent stores the data carried by the message and tags it with the logical time carried by the message. Only when the agent *uses* the stored information for the first time, the logical counter carried by the delivering message is considered and in case its value is larger than the agent's counter, the agent increments its counter.

The plan of the paper is as follows. Distributed constraint satisfaction problems ($DisCSPs$) are presented in Section 2. The existing methods for measuring non-concurrent computation, including the method for perfect communication and the asynchronous message delay simulator ($AMDS$), are presented in Section 3. Section 4 presents the general method for counting non-concurrent logical steps. A demonstration of the cases in which the existing methods fail to report non-concurrent time is given, followed by a detailed description of the method and its compatibility to a variety of cases. Section 5 presents a correctness proof for the proposed general method. Our conclusions are in section 6.

2 Distributed Constraint Satisfaction

A distributed constraints network (or a distributed constraints satisfaction problem - *DisCSP*) is composed of a set of k agents $A_1, A_2, ..., A_k$. Each agent A_i contains a set of constrained variables $X_{i_1}, X_{i_2}, ..., X_{i_{n_i}}$. Constraints or **relations** R are subsets of the Cartesian product of the domains of the constrained variables [2]. A **binary constraint** R_{ij} between any two variables X_j and X_i is defined as: $R_{ij} \subseteq D_j \times D_i$. In a distributed constraint satisfaction problem *DisCSP*, the agents are connected by constraints between variables that belong to different agents (cf. [9, 8]). In addition, each agent has a set of constrained variables, i.e. a *local constraint network*.

An assignment (or a label) is a pair $< var, val >$, where var is a variable of some agent and val is a value from var's domain that is assigned to it. A *partial assignment* (or a compound label) is a set of assignments of values to a set of variables. A **solution** to a *DisCSP* is an assignment that includes all variables of all agents, that is consistent with all constraints. Following all former work on *DisCSP*s, agents check assignments of values against non-local constraints by communicating with other agents through sending and receiving messages. An agent can send messages to any one of the other agents [9].

The delay in delivering a message is assumed to be finite [9]. One simple protocol for checking constraints, that appears in many distributed search algorithms, is to send a proposed assignment $< var, val >$, of one agent to another agent. The receiving agent checks the compatibility of the proposed assignment with its own assignments and with the domains of its variables and returns a message that either acknowledges or rejects the proposed assignment. The following assumptions are routinely made in studies of $DisCSP$s and are assumed to hold in the present study [9, 1].

1. All agents hold exactly one variable.
2. The amount of time that passes between the sending of a message to its reception is finite.
3. Messages sent by agent A_i to agent A_j are received by A_j in the order they were sent.
4. Every agent can access the constraints in which it is involved and check consistency against assignments of other agents.

```
• upon receiving message $msg$:
  1. LTC ← max(LTC, msg.LTC)
  2. delay ← $choose\_delay$
  3. msg.$delivery\_time$ ← msg.LTC + delay
  4. $outgoing\_queue$.add(msg)
  5. $deliver\_messages$
• when there are no incoming messages and all agents are idle
  1. LTC ← $outgoing\_queue$.first_msg.LTC
  2. $deliver\_messages$
• deliver_messages
  1. foreach (message m in outgoing queue)
  2.   if ($m.delivery\_time \leq$ LTC)
  3.     deliver(m)
```

Figure 1. The Mailer algorithm

3 Measuring performance of DisCSP search algorithms

The standard model of Distributed Constraints Satisfaction Problems has agents that are autonomous asynchronous entities. The actions of agents are triggered by messages that are passed among them. In real world systems, messages do not arrive instantaneously but are delayed due to networks properties. Delays can vary from network to network (or with time, in a single network) due to networks topologies, different hardware and different protocols used.

Non-concurrent steps of computation, in systems with no message delay, can be counted by a method similar to that of [3, 5]. Every agent holds a counter of computation steps which it increments each time it performs a step of computation. Every message carries the value of the sending agent's counter. When an agent receives a message it updates its counter to the largest value between its own counter and the counter value carried by the message. By reporting the cost of the search as the largest counter held by some agent at the end of the search, a non-concurrent measure of search effort is achieved (see [3]).

On systems with message delays, the situation is different. To introduce the problems of counting in the presence of message delays, let us start with the simplest possible algorithm. Synchronous backtracking (SBT) performs assignments sequentially, one by one and no two assignments are performed concurrently. Consequently, the effect of message delay is very clear. The number of computation steps is not affected by message delay and the delay in every step of computation is the delay on the message that triggered it. Therefore, the total time of the algorithm run can be calculated as the total computation time, plus the total delay time of messages. In the presence of concurrent computation, the time of message delays must be added to the run-time of the algorithm *only if no computation was performed concurrently*. To achieve this goal, the simulator counts message delays in terms of computation steps and adds them to the accumulated run-time. Such additions are performed only for instances when no computation is performed. In other words, when the delay of a message causes all agents to wait, performing no computation.

In order to simulate message delays, all messages are passed by a dedicated $Mailer$ thread. The mailer holds a counter of non-concurrent computation steps performed by agents in the system. This counter represents the logical time of the system and we refer to it as the *Logical Time Counter* (LTC). Every message delivered by the mailer to an agent, carries the LTC value of its delivery to the receiving agent. An agent that receives a message updates its counter to the maximum value between the received LTC and its own value. Next, it performs the computation step, and sends its outgoing messages with the value of its counter, incremented by 1. The same mechanism can be used for computing computational effort, by counting non-concurrent constraints checks. Agents add to the counter values in outgoing messages the number of constraints checks performed in the current step of computation.

The mailer simulates message delays in terms of non-concurrent computation steps. To do so it uses the LTC, according to the algorithm presented in figure 1. Let us go over the details of the $Mailer$ algorithm, in order to understand the measurements performed by the simulator during run time.

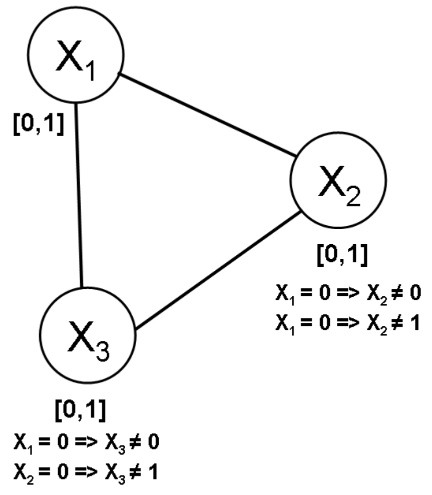
When the mailer receives a message, it first checks if the LTC value that is carried by the message is larger than its own value. If so, it increments the value of the LTC (line 1). In line 2 a delay for the message (in number of steps) is selected. Here, different types of selection mechanisms can be used, from fixed delays, through random delays, to delays that depend on the actual load of the communication network. To achieve delays that simulate dependency on network load, for example, one can assign message delays that are proportional to the size of the outgoing message queue.

Each message is assigned a $delivery\_time$ which is the sum of the value of the message's LTC and the selected delay (in steps), and placed in the $outgoing\_queue$ (lines 3,4). The $outgoing\_queue$ is a priority queue in which the messages are sorted by $delivery\_time$, so that the first message is the message with the lowest $delivery\_time$. In order to preserve the third assumption from section 2, messages from agent A_i to agent A_j cannot be placed in the outgoing queue before messages which are already in the outgoing queue which were also sent from A_i to A_j. This property is essential to asynchronous backtracking which is not correct without it (cf. [1]). The last line of the $Mailer$'s code calls method $deliver\_messages$, which delivers all messages with $delivery\_time$ less or equal to the mailer's current LTC value, to their destination agents.

When there are no incoming messages, and all agents are idle, if the $outgoing\_queue$ is not empty (otherwise the system is idle and a solution has been found) the mailer increases the value of the LTC to the value of the $delivery\_time$ of the first message in the outgoing queue and calls $deliver\_messages$. This is a crucial step of the simulation algorithm. Consider the run of a synchronous search algorithm. For *Synchronous Backtracking* (SBT) [9], every delay needs the mechanism of updating the Mailer's LTC (line 1 of the second function of the code in figure 1). This is because only one agent is computing at any given instance, in synchronous backtrack search.

The non-concurrent run time reported by the algorithm, is the largest LTC value that is held by any agent at the end of the algorithm's run. By incrementing the LTC only when messages carry LTCs with values larger than the mailer's LTC value, steps that were performed concurrently are not counted twice. This is an extension of Lamport's logical clocks algorithm [3], as proposed for DisCSPs by [5], and extended for message delays in [14].

A similar description holds for evaluating the algorithm run in non-concurrent constraints checks. In this case the agents need to extend the value of their LTCs by the number of constraints checks they actually performed in each step. This enables a concurrent performance measure that incorporates the computational cost of the local step, which might be different in different algorithms. It also enables to evaluate algorithms in which agents perform computation which is not triggered or followed by a message.



X_1
[0,1]
X_2
[0,1]
$X_1 = 0 \Rightarrow X_2 \neq 0$
$X_1 = 0 \Rightarrow X_2 \neq 1$
X_3
[0,1]
$X_1 = 0 \Rightarrow X_3 \neq 0$
$X_2 = 0 \Rightarrow X_3 \neq 1$

Figure 2. An example of the failure of a non generic method

4 A generic method for counting non-concurrent logical steps

In order to understand why the methods presented in the previous section are not sufficient for every search algorithm, we first give an example of how additional messages may cause logical steps which could have been performed concurrently to be counted as non-concurrent logical steps.

4.1 Demonstration

Consider the following example. Each of the agents in the DisCSP presented in figure 2 holds exactly one variable. The domain of each of the variables includes two values - 0 and 1. The agents are ordered according to their indices. Agent X_2 does not have a consistent assignment when agent X_1 assigns 0. Agent X_3 does not have a consistent assignment when agents X_1 and X_2 both assign 0 to their variables.

The algorithm performed by the agents is standard ABT [9, 1] with a small change. Nogood messages are sent to all the agents whose assignments are included in the $Nogood$ (as in AWC [9]) and not just to the agent with the lowest priority whose assignment is included in the $Nogood$. The algorithm runs as follows:

1. X_1 assigns its variable the value 0, and sends to X_2 and X_3 a corresponding **ok?** message. The LTC on both messages is 0 since it has not performed any CCs.
2. X_2 assigns its variable the value 0, and sends to X_3 a corresponding **ok?** message. $LTC = 0$.
3. X_3 receives both **ok?** messages from X_1 and X_2.
4. After trying to assign both values, X_3 sends to X_1 and X_2 the **Nogood** $X_1 = 0 \Rightarrow X_2 \neq 0$. $LTC = 4$. (In standard ABT this nogood would have been sent only to X_2).

5. After trying to assign both values, X_2 sends to X_1 the **Nogood** $X_1 \neq 0$. $LTC = 2$.
6. X_1 receives both Nogoods from X_2 and X_3, increments its LTC to 4, replaces its assignment with 1 and sends to X_2 and X_3 a corresponding **ok?** message.
7. X_2 receives the **ok?** message from X_1. performs two CCs ,replaces its assignment to 1 and sends an **ok?** message to X_3 with $LTC = 6$
8. X_3 receives both **ok?** messages from X_1 and X_2. It updates its LTC to be 6. Then it performs two CCs and assigns 0. Its LTC value is 8.
9. X_2 receives the **nogood** from X_3 which is discarded since it is obsolete.

At the end of the run of this example the $LTCs$ of the agents are $4, 6$ and 8 respectively. This means that the number of non-concurrent constraints checks reported is 8. However, a closer look at the steps of the algorithm run would reveal that the **nogood** sent by X_3 to X_1 at step 4 and received at step 6 did not trigger computation and its data was not used by X_1. It did however cause X_1 to increment its LTC to 4. If $X_!$ would have ignored this **nogood** message, the cost of the algorithm run was 6.

The error in the counting method presented above was caused by a message which did not trigger computation and carried an LTC which was higher than the LTC of the receiving agent, and incremented its counter. This message, which the algorithm could have done without, did not affect the amount of computation steps of the algorithm but did affect their counting. In standard ABT, such a message would not have been sent and therefore the run-time results reported by the non-concurrent method of [5] would have been correct. The above example may seam not relevant in the case of standard ABT but it becomes very relevant when dynamic agent ordering is used in asynchronous backtracking algorithms. In asynchronous backtracking with dynamic agent ordering [9, 12] the agents hold in their *Agent Views* assignments of both higher and lower priority agents. The agents check their current assignment only against assignments of agents with higher priority according to the current order. However, since the priority order is dynamic, an assignment of a lower priority agent which is currently irrelevant, may become relevant as a result of a change in the order of priorities, thus it is not discarded from the agent's *Agent View*. When running ABT with dynamic ordering (ABT_DO [12]), agents send their assignments to all their neighbors (and not only to their current lower priority neighbors) for the same reason [9, 12].

Messages which carry the assignments of lower priority agents to higher priority agents do not trigger immediate computation since the received assignment cannot rule out the local assignment even if they are in conflict.

4.2 A general Distributed Measurement Method

In order to construct a generic method for counting non-concurrent logic steps the existing methods of [5, 14] should be adjusted to deal with messages which do not trigger immediate computation and their data is stored for later use. In order to preserve the concept of *non-concurrent* logic steps, for every message received, before updating the local *logic time counter* (LTC) the agent must make sure that the computation performed in order to produce the data carried by the message *could not have been performed* concurrently with the steps of computation it is about to perform.

In the alternative general method we suggest in this paper, like in the previous suggested methods of [5, 14], each agent holds a *Logic Time Counter* (LTC) of logic steps (steps of computation, constraints checks,...). The counter is incremented whenever an agent performs the logic step. Each message caries the LTC of the sending agent. The receiver of a message, instead of comparing the LTC carried by the message with its own and updating its local LTC to the largest among the two (as done in [5, 14]), stores the received data together with the received LTC. When the agent uses the stored data for the first time, it performs the comparison and updating of the local LTC compared with the stored LTC attached to the stored data.

In standard DisCSP algorithms, the suggested method would give exactly the same results as the methods of [5, 11]. That is because each message in standard DisCSP algorithms triggers computation which evaluates the data received. Therefore using the suggested general method, the agents would immediately use the stored data and update their local LTC with the largest between the received LTC and their own. In algorithms such as AWC [9], and ABT_DO [12] messages which carry data which is not used in the following computation performed by the receiving agent will not effect the counting method until it is used.

5 Correctness of the General Measure method

In order to prove the validity of the proposed general measurement method we first prove that in the case that all messages trigger immediate computation the method reports the same measures as the methods presented in [5, 14] and therefore the correctness proofs presented there hold. This fact is immediate.

Second, we prove that for a message carrying data which is not used by the receiving agent, the general method is correct. Since the data is not used, the results should not be affected by the message sent. In fact, the desired measure in this case is the one that we would get if the algorithm would run without this message being sent. Since in the proposed method counters are only taken into consideration when the data carried by the message is used, that is precisely the resulting measure.

Last, we consider the case of a message m which was received by an agent when its LTC was equal to t_1 and the data carried by m was first used when the agent's LTC was equal to t_2. Since the data carried by m was only used at t_2 this means that although m was received at local logical time t_1, the same computation would have been performed if the message would have arrived at any time between t_1 and t_2. In other words, the logical steps performed by the sender of m in order to produce the data carried by m, could have been performed concurrently with the logical steps performed by the receiving agent between t_1 and t_2. This contradicts the possibility of counting these steps as non-concurrent $\square$.

6 Discussion

When measuring the run-time performance of distributed algorithms the concurrency of agents must be taken into consideration. A general method for measuring distributed performance must use an asynchronous simulator, to simulate a real distributed system. Since the logical steps in different simulating systems can be performed in many different scenarios, one must measure the longest sequence of logical steps which could not have been performed concurrently during the run of the algorithm. The proposed general method in this paper actually reports the longest among all of the non-concurrent (shortest) sequences of logical steps performed by the algorithm. The method was demonstrated and proven to be correct for any distributed search algorithm and for two nonconcurrent measures - number of steps of computation and number of $NCCCs$.

REFERENCES

[1] C. Bessiere, A. Maestre, I. Brito, and P. Meseguer. Asynchronous backtracking without adding links: a new member in the abt family. *Artificial Intelligence*, 161:1-2:7–24, January 2005.

[2] Rina Dechter. *Constraints Processing*. Morgan Kaufman, 2003.

[3] L. Lamport. Time, clocks, and the ordering of events in distributed system. *Communication of the ACM*, 2:95–114, April 1978.

[4] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series, 1997.

[5] A. Meisels, I. Razgon, E. Kaplansky, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 86–93, Bologna, July 2002.

[6] A. Meisels and R. Zivan. Asynchronous forward-checking for distributed csps. In W. Zhang, editor, *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.

[7] M. C. Silaghi. *Asynchronously Solving Problems with Privacy Requirements*. PhD thesis, Swiss Federal Institute of Technology (EPFL), 2002.

[8] G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (dcsps). In *Constraint Processing-96*, pages 561–2, New Hamphshire, October 1996.

[9] M. Yokoo. Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents & Multi-Agent Sys.*, 3:198–212, 2000.

[10] R. Zivan and A. Meisels. Concurrent backtrack search for discsps. In *Proc. FLAIRS-04*, pages 776–81, Maiami Florida, May 2004.

[11] R. Zivan and A. Meisels. Message delay and discsp search algorithms. In *Proc. 5th workshop on Distributed Constraints Reasoning, DCR-04*, Toronto, 2004.

[12] R. Zivan and A. Meisels. Dynamic ordering for asynchronous backtracking on discsps. In *CP-2005*, pages 32–46, Sigtes (Barcelona), Spain, 2005.

[13] R. Zivan and A. Meisels. Message delay and asynchronous discsp search. *Archives of Control*, 2006.

[14] R. Zivan and A. Meisels. Message delay and discsp search algorithms. *Annals of Mathmatics and Artificial Inteligence (AMAII)*, 2006.