On the ratio of communications to computation in DCR efficiency metrics

Tracking Number: 673

ABSTRACT

We propose a way to define *the most expensive operation* to be used in evaluations of complexity and efficiency for simulated distributed constraint reasoning (DCR) algorithms. We also report experiments showing that the cost associated with a constraint check, even within the same algorithm, depends on the problem size. The DCR research has seen heated debate regarding the *correct* way to evaluate efficiency of simulated algorithms. DCR has to accommodate two established practices coming from very different fields: distributed computing and constraint reasoning. The efficiency of distributed algorithms is typically evaluated in terms of the network load and overall computation time, while many (synchronous) algorithms are evaluated in terms of the number of rounds that they require. Constraint reasoning research evaluates efficiency in terms of *constraint checks* and *visited search-tree nodes*.

We argue that an algorithm has to be evaluated from the point of view of specific *operating points*, namely of possible or targeted application scenarios. We then show how to report efficiency for a given operating point based on simulation. Additionally, new experiments we report here show the fact that the cost of a constraint check varies with the size of the problem, and we discuss the implications of this phenomenon.

1. INTRODUCTION

This article addresses the evaluation of distributed constraint reasoning algorithms. One of the major achievements of computer science consists of the development of the complexity theory for evaluating and comparing the efficiency of algorithms in a scalable way [8]. Complexity theory proposes to evaluate an algorithm in terms of the number of times it performs *the most expensive operation*. This number is seen as a function of the size of the problem. While such metrics do not reveal how much actual time is required for a certain instance, they allow for interpolating how the technique scales with larger problems. The assumption that computation speed will double each few years makes a polynomial factor in the cost irrelevant from a long perspective [8, 4].

Identifying the most expensive operation is not always trivial. For simple algorithms such as centralized sorting and graph traversal, this operation is typically identified as the processing inside *the most inner loop*. Constraint reasoning researchers have long used either the *constraint check* or the *visited search-tree node* as the most expensive operation in classical algorithms. In algorithms whose structure does not present a *most inner loop*, a scalable efficiency evaluation is usually based on the operation that seems to be the most often used and that is relatively expensive. However, the cost of operations can be distorted by specialized hardware where, for example, on some processors multiplication is as efficient as addition. This leads to an uncertainty about whether the obtained metric is relevant for evaluating the same algorithm on a new machine. Moreover, that *most expensive operation* can prove irrelevant for a competing algorithm who uses extensively another operation. In general, the relevance of a metric is therefore relative to the algorithm and the scenario where it will be used. The relativity of relevance of metrics is not considered important for some problem domains and algorithms that are similar, as is usually the case with constraint reasoning.

Evaluating distributed computing.

However, the relativity in the relevance of metrics is particularly acute for distributed computing. The two main reasons for it are:

- the relative ratio between cost (latency) of messages varies by 4-6 orders of magnitude between multiprocessors and remote Internet connections, and
- while the cost of a centralized computation can be expected to reduce over years, the cost (latency) of a message between two points is not expected to decrease significantly (in contrast with the other computation costs), since the limits of the current technology are already dictated by the time it takes light to traverse that distance over optical cable.

Indeed, the minimal time it can theoretically take a message to travel between two diametrically opposed points on the Earth is:

$$\frac{\pi * R_{Earth}}{speed_{light}} = \frac{3.14 * 6.378 * 10^6 m}{3 * 10^8 m/s} \approx 67ms.$$

Since the optical cables do not travel on a perfect circle around the Earth, it is reasonable to not expect significant improvements beyond the current some 150ms latency for such distances.

For a realistic understanding of the behavior of distributed algorithms, some experiments are performed using agents

Padgham, Parkes, Mueller and Parsons (eds.): Proc. of AAMAS 2008, Estoril, May, 12-16., 2008, Portugal, pp. XXX-XXX. **Cite as:** Title, Author(s), Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systens, pp. XXX-XXX, 2008.

placed on different computers on Internet, typically on a LAN [25, 9, 20, 12]. However, results obtained with LANs may not be valid for any other network topology, or for remote agents on Internet. Also, such results cannot be replicated and verified by other researchers, and therefore results using deterministic network simulators are also commonly requested.

In the following we provide the formal definition for the Distributed Constraint Optimization (DCOP) problem. Then we introduce a simple framework for unifying various versions of *logic time* systems. We show that the framework models well the different efficiency metrics and methodologies previously used to evaluate DCOP algorithms based on logic time. These previous methodologies are presented in the unifying framework to make the understanding of their properties easier. We then discuss the arguments and drawbacks for each such metric, and then argument our new procedure for evaluating experiments.

2. FRAMEWORK

Distributed Constraint Optimization (DCOP) is a formalism that can model naturally distributed problems. These are problems where agents try to find assignments to a set of variables that are subject to constraints. Several applications are addressed in the literature, such as multi-agent scheduling problems, oil distribution problems, auctions, or distributed control of red lights in a city [15, 22, 19].

DEFINITION 1 (DCOP). *A distributed constraint optimization problem (DCOP), is defined by a set A of agents* $A_1, A_2, ..., A_n$, *a set X of variables,* $x_1, x_2, ..., x_n$, *and a set of functions (aka constraints)* $f_1, f_2, ...f_i, ..., f_m$, $f_i : X_i \rightarrow \boldsymbol{R}_+$, $X_i \subseteq X$, *where only some agent* A_j *knows* f_i.

The problem is to find $argmin_x \sum_{i=1}^{m} f_i(x_{|X_i})$. *We assume that* x_i *can only take values from a domain* $D_i = \{1, ..., d\}$.

The DCOPs where the functions f_i are defined as $f_i : X_i \rightarrow \{0, \infty\}$, are called Distributed Constraint Satisfaction Problems (DisCSPs). Algorithms for the general DCOP framework can address any DisCSP and specialized algorithms for DisCSPs can often be extended to DCOPs.

3. EVALUATION FOR MIMD

Some of the early works on distributed constraint reasoning were driven by the need to speed up computations on multiprocessors, in particular (multiple instruction multiple data) MIMD architectures [25, 3, 10], sometimes even with a centralized command [3]. However, their authors pointed out that those techniques can be applied straightforwardly for applications where agents are distributed on Internet.

Among the earliest experimental research on DCR we mention [25] by Zhang and Mackworth. The metric proposed by them is based on Lamport's logic clocks described in the Definition 6.1 and in the Algorithm 18 in [25].

Logic clocks and logic time.

An event e_1 at agent A_1 is said to *causally precede* an event e_2 at agent A_2 if, had all agents attached all events that they knew to each existing message, A_2 would know about e_1 at the moment when e_2 takes place. Leslie Lamport proposes in [11] a way, called *logic clocks* to construct a tag, called *logic time* (LT), for each event and concurrent message in a distributed computation such that whenever an event e_1 causally precedes e_2 then the logic time of e_1 should be smaller than the logic time of e_2. If $LT(e)$ denotes the logic time of an event e, then we can write $LT(e_1) < LT(e_2)$. Otherwise, the logic time does not reflect the real time and some messages with smaller logic time may actually occur after concurrent messages with bigger logic time. Each process P_i maintains its own logic clock with logic time (LT_i) initially set to zero. Whenever P_i sends a message m, it attaches to m a tag, denoted $LT(m)$, set to the value of LT_i at that moment. The process P_i increments LT_i by the logic duration, t_e, of each local event (computation) e. Assume P_i receives a new message m_k from a process P_j. P_i has to make sure that the logic time LT_i of its future local events is higher than the LT_j of the past events at P_j. This is done by setting $LT_i = max(LT_i, LT(m_k) + L)$, where L is a logic time (duration) assigned to each message passing. We give in Algorithm 1 the procedures proposed in [11], tailored to unify the different metrics used for DCOPs. Certain authors use random values for the logic time of a message [7] and therefore we allow this in our framework by specifying a number series generator (NSG) R_L from which each message logic time (logic latency) is extracted with a function $next()$. A *logic time system* we will use here is therefore parametrized as $LT\langle R_L, E, T\rangle$ where E is a vector of types of local events and T a vector of costs, one for each type of event. For measurements assuming a constant latency of messages set to a value L, the R_L parameter used consists of that particular number, **L**, (written in bold face).

```
% R_L is the number series generator from which message
latencies are extracted using function next()
% E = {e_1, ..., e_k} is a vector of k local events
% T = {t_1, ..., t_k} is a vector of (logic) costs for events E
when event e_j happens do
    LT_i = LT_i + t_j;
end do.
when message m is sent do
    LT(m) = LT_i + next(R_L));
end do.
when message m is received do
    LT_i = max(LT_i, LT(m));
end do.
```

Algorithm 1: Lamport's logic time maintenance for participant P_i. Use of parameters $LT\langle R_L, E, T\rangle$ unifies previous versions for usage with DisCSPs found in (Zhang& Mackworth 1991; Yokoo, Durfee, Ishida& Kuwabara 1992; Meisels, Kaplansky, Razgon& Zivan 2002; Silaghi& Faltings 2004; Chechetka& Sycara 2006).

An experiment may simultaneously use several logic time systems, $LT^1\langle R_L^1, E^1, T^1\rangle, ..., LT^N\langle R_L^K, E^K, T^K\rangle$. Each process P_i maintains a separate logic clock, with times LT_i^u, for each $LT^u\langle R_L^u, E^u, T^u\rangle$. Also, to each message m one will attach a separate tag $LT^u(m)$ for each maintained logic time system $LT^u\langle R_L^u, E^u, T^u\rangle$. This is done in order to simultaneously evaluate a given algorithm and set of problems for several different scenarios (MIMD, LAN, remote Internet).

A common metric used to evaluate simulations of DCR algorithms is given by the *logic time to stability* of a computation. The logic time to stability is given by either:

- the highest logic time of an event occurring before *qui-*

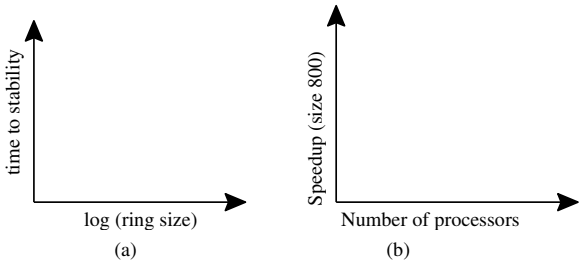

Figure 1: Graph axes depicting (a) logic time to stability vs problem size as (log scale) number of variables; and (b) logic time vs. number of processors at a given size of the DisCSP distributed to those processors.

escence is reached [25];

- the logic time tagging the message that makes the solution known to whoever is supposed to be informed about it [20].

Quiescence of an algorithm execution is the state where no agent performs any computation related to that algorithm and no message generated by the algorithm is traveling between agents.

Uses of logic time for multiprocessors.

The operation environment targeted by [25] consists of a network of transputers. The metric employed in [25] with simulations for a constraint networks with ring topology is based on the logic time system $LT\langle \mathbf{1}, \{semijoin\}, \{1\}\rangle$, where the number series generator $\mathbf{1}$ outputs the value 1 at each call to $next()$. Note that the single local event associated in [25] with a cost is the *semijoin*, which is due to the fact that the algorithms being tested there were not based on constraint checks but on semijoin operators (which consist of composing constraints and then projecting the result on a subset of the involved variables). Graph axes used in [25] depict *logic time to stability* vs *problem size as (log scale) number of variables*, and *logic time* vs. *number of processors (aka agents) at a given size of the DisCSP* distributed to those agents (see Figure 1).

A theoretical analysis of the time complexity of a DisCSP solver is presented in [3]. Logic time analysis is presented there under the name *parallel time*, targeting MIMD multiprocessors, where each value change (aka *visited search-tree node* in regular CSP solvers) has cost 1. Note that the obtained metric is $LT\langle \mathbf{0}, \{value\text{-}change\}, \{1\}\rangle$, where message passing is considered instantaneous. A sequential version of the same algorithm is also evaluated in [3] using the logic time $LT\langle \mathbf{0}, \{value\text{-}change, privilege\text{-}passing\}, \{1, 1\}\rangle$. The term coined in [10] for a similar theoretical analysis of the time complexity in parallel computations is *sequential time*.

4. EVALUATION FOR APPLICATIONS TARGETING INTERNET

Distributed constraint reasoning algorithms targeting the Internet had to account for the possibly high cost of message passing between agents on remote computers. The latency of message passing in this context is a function on the distance and available connections between the locations. As

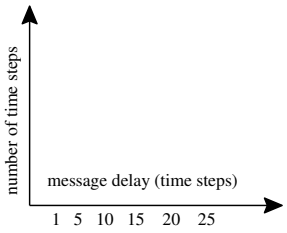

Figure 2: Performance graph for one problem instance used in (Yokoo.et.al 1992). The number of time steps is what some recent articles call *equivalent non-concurrent constraint checks* (NCCCs).

mentioned above, the theoretical lower bound on this latency can be 67ms, eight orders of magnitude larger than a basic operation on a computer (of the order of 1ns).

Network Simulators.

While some experiments use agents placed on distinct computers on a LAN, such experiments can somewhat shew the results due to the following.

- agents are geographically closer to each others than in Internet applications, and therefore the latency of messages can be 2-3 orders of magnitude smaller (1-2 ms instead of 100-200ms) [6].

- due to the shared medium used by the typical Ethernet implementation of LANs, the bandwidth is shared and communication between a pair of agents slows down communication between any other pair of agents.

These two issues act in different directions and it is not clear in which actual direction are the results skewed. This makes another argument toward evaluating performance on a simulated network. It is worth noting that early research, such as [25] perform experiments both with simulators and with actual execution on multiprocessors.

Metrics for Internet.

One of the first algorithms targeting Internet is the Asynchronous Backtracking solver in [23]. That work experimented with a set of different logic times, $LT^1, ..., LT^{25}$, where LT^i is defined by the parameters

$$LT^i\langle \mathbf{i}, \{constraint\text{-}check\}, \{1\}\rangle, \forall i \in [1, 25] \qquad (1)$$

[23] reports the importance of the message latency in deciding which algorithm is good for which task. Note that a curve in the obtained type of graph (see Figure 2) reports several metrics, but for a single problem size/type.

The *time steps* introduced in [23] correspond to the cost of a constraint check. A similar results graph is used in [21] having as axes checks vs checks/message, i.e., the logic time cost for one message latency when the unit is the duration of a constraint check (see Figure 3.a). This last graph also reports logic time for the latency $L = 0$

$$LT^0\langle \mathbf{0}, \{constraint\text{-}check\}, \{1\}\rangle, \qquad (2)$$

which corresponds to simulation of execution with agents placed on the processors of a MIMD with very efficient (instantaneous) message passing (similar to [3], but using the constraint check as the logic unit). This particular metric

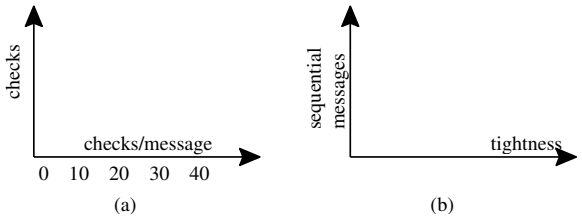

Figure 3: Performance graphs showing (a) (non-concurrent) checks versus various latency/check costs including 0 and (b) showing sequential messages versus constraint tightness.

is sometimes referred to as *the number of non-concurrent constraint checks*.

Cycles.

After the work in [23], most DCOP research focused on agents placed on remote computers with problem distribution motivated by privacy [24]. Due to the small ratio between the cost of a constraint check and the cost of one message latency in Internet, the standard evaluation model selected in many subsequent publications completely dropped the accounting of constraint checks. A common assumption adopted for evaluation is that local computations can be made arbitrarily fast (local problems are assumed small and an agent can make his computation on arbitrarily fast supercomputers). Instead, message latency between agents is a cost that cannot be circumvented in environments distributed due to privacy constraints. The metric in [24] is:

$$cycles = LT\langle \mathbf{1}, \emptyset, \emptyset \rangle$$

The original name for this metric is *cycles*, based on the next theorem (known among some researchers but not written down in this context).

THEOREM 1. *In a network system where all messages have the same constant latency L and local computations are instantaneous, all local processing is done synchronously only at time points kL (in all agents).*

PROOF. One assumes that all agents start the algorithm simultaneously at time L, being announced by a broadcast message, which reaches all agents at exactly time L (due to the constant time latency). Each agent performs computations only either at the beginning, or as a result of receiving a message.

Since each computation is instantaneous, any message generated by that computation is sent only at the exact time when the message triggering that computation was received. It can be noted that (**induction base**) any message sent as a result of the computation at the start will be received at time $2L$, since it takes messages L logic time units after the start to reach the target.

Induction step: All the messages that leave agents at time kL, will reach their destination at exactly time $(k+1)L$ (due to the constant latency L). Therefore the observation is proven by induction. □

As a consequence of this observation, any network simulation respecting these assumptions (that local computations are instantaneous and that message latencies are constant) can be performed employing a loop, where at each cycle each agent handles all the messages sent to it at the previous cycle. As such, $LT\langle \mathbf{1}, \emptyset, \emptyset \rangle$ is given by the total number of cycles of this simulator.

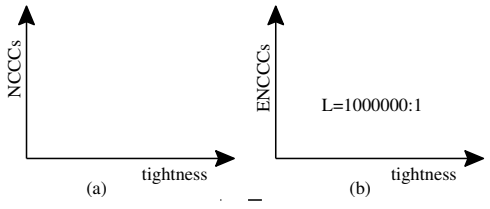

Figure 4: Performance graphs based on (a) NCCCs and (b) ENCCCs.

Sequential messages (SMs).

Some researchers voice concern that the name cycle is counter-intuitive and would suggest synchronous algorithms. A significant fraction of reviewers currently adhere particularly strongly to this opinion and are known to commonly reject articles that report results of this metric under the name *cycles*.

This is not our opinion. However, another name previously used for the same metric is *number of sequential messages (SMs)* [20] or length of the *longest causal chain of messages*. These names come from the common terminology of Lamport's logic clocks work [11] and directly suggest their meaning and applicability to general asynchronous algorithms. In particular *sequential messages* can be used for experimentation with random message latencies. The graphs in [20] depict results using as axes of coordinates *sequential messages* versus problem parameters (see Figure 3.b).

NCCCs and ENCCCs.

The practice of using *cycles* and *sequential messages* as time unit was intriguing for the CSP community which practices the use of constraint checks and search-tree nodes, and a debate at the Constraint Programming 2001 conference led to the subsequent re-introduction of logic time in the form of the metric in Equation 2. That work [13] proposes to build graphs with axes labeled *NCCCs (non-concurrent constraint checks)* versus problem type (Figure 4.a). This approach differs from earlier works by the fact that the cost of a message in such graphs is typically restricted to only 0, reporting solely constraint checks, (while the article admits that other values are possible). This metric (initially used for DCOPs targeting MIMD multiprocessors in [3]) has been earlier dropped for the evaluation of problems with privacy requirements on Internet. Its re-introduction (and in particular the NCCC acronym) has particular success, and a vocal community of reviewers currently still maintains and mandates its usage for evaluation of DCOP algorithms.

However, the importance of the latency of messages has been rediscovered recently and logic time cost for message latency is reintroduced in [2] under the name Equivalent *Non-Concurrent Constraint Checks (ENCCCs)*. ENCCCs are computed using the Equation 1. Current ENCCCs usage in graphs typically differs from earlier usage of the metric by being depicted versus *constraint tightness* or versus *density of constraint-graph* (with a label specifying the value of the logic latency L, i.e. the number of checks/message-latency).

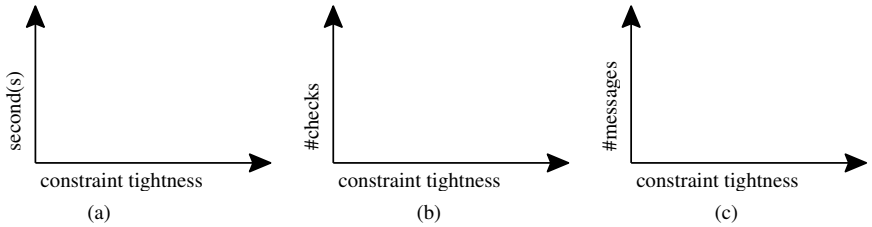

Figure 5: Performance graphs illustrating (a) seconds per constraint tightness of the problems; (b) total number of constraint checks per tightness, and (c) total number of exchanged messages per tightness.

Each graph depicts the behavior of several problem types for one message latency, rather than the behavior of one problem type for several message latencies.

Evaluations not related with the logic time.

Three other important metrics (not based on logic time) for evaluating DCOPs algorithm were introduced in [9] in conjunction with a DisCSP solver.

- the total running time in seconds for solving a DCOP with agents distributed on the computers of a LAN (with axes illustrated in Figure 5).a;
- the total number of constraint checks, illustrates the efficiency of the algorithms if used to solve a CSP, by converting the CSP into a DisCSP (DCOP) and solving it with a simulator(see Figure 5).b, and
- the total number of exchanged messages, which illustrates the total network load (with axes illustrated in Figure 5).c. It is worth noting that measuring the network load as the number of exchanged bytes rather than exchanged messages is a common practice in multi-party computations, and may be interesting in some DCOP solvers where the messages do not have constant size (e.g. DPOP [18]).

Note that these graphs introduced the use as axes of abscissas the *constraint tightness*, namely the fraction of allowed (i.e., 0-valued) tuples in each constraint.

Another evaluation methodology that is not fully based on logic time, but whose computation uses logic time, is called *cycle-based runtime*. It is equivalent with computing ENCCCs on a modified version of the algorithm, which adds synchronizations before sending a each message [5].

5. A NEW METHODOLOGY

Next we describe a new methodology for evaluating DCOP algorithms that we decided to employ recently [1], but which has not yet been introduced in sufficient detail.

Let us first mention the weaknesses in currently common methodologies, and which we want to fix with our new proposed approach:

- the weakness of the *cycles/sequential-messages* metric is that its assumptions do not apply to DCOP solvers with extensive local processing at each message (such as in the recent DPOP algorithm [18]). DPOP has very few messages and very expensive local computation at each message.
- NCCCs (in the version with message cost zero) do not take into account message latencies, which are an important cost for many typical DCOP algorithms. Moreover, (see the Experiments section) the cost of a constraint check grows linearly with the problem size (for the same algorithm), causing misleading curves.
- ENCCCs require depicting many graphs, one for each possible checks/latency ratio, and still does not offer a way to know which ratio is relevant to a given application. This is due to the fact that the cost that has to be associated with a constraint check depends on many factors, being a function of the algorithm, of the programming language, and (as we report here) even function of the problem size. Plots of different algorithms on the same ENCCCs graph are not comparable since their units often have different meaning and relevance (and may not even be bounded by a polynomial relation).
- time in seconds of experiments on a LAN, besides the fact that its measuring requires important hardware resources, it does not apply to remote Internet applications, or to other hardware, and cannot be replicated.

Our proposal is, given any well defined application scenario, to start by first computing the expected latency/checks ratio, following the next procedure.

Proposed Evaluation Method.

1. Retrieve the typical latency L_s in seconds for messages in the type of network of the targeted application. Such information is found in technical catalogs, encyclopedias, and technical articles. For example, some typical message latencies for remote machines on Internet are found in [17].
2. Compute the total execution time in seconds, t_p, for solving each complete test set of problems at size p using the simulator. Note that this is machine and programming language dependent, and therefore the used machine and programming language have to be specified.
3. Compute the total number of constraint checks, $\#CC_p$, at each problem size p [9] (see Figure 5.b).
4. Compute the cost in seconds that should be associated with a constraint check by computing the ratio $t_p/\#CC_p$. We note that for a given machine and programming language this cost depends on the problem size p, varying as much as an order of magnitude. For example, our C simulator for ADOPT on the problems

$$LT^i\langle \mathbf{i}, \{constraint\text{-}check, nogood\text{-}inference, nogood\text{-}validity, nogood\text{-}applicability\}, \{1,3,2,2\}\rangle, \forall i > 0 \quad (3)$$

in [16] uses between 3 to 28 microseconds per constraint check on a Linux PC at 700MHz. The smaller value was found at problems with 8 agents and 8 variables and the larger one at problems with 40 agents and variables. We discuss later our explanation for this phenomenon.

5. Compute the ratio message-latency/constraint-check for the given problem size p as $L_p = L_s * \#CC_p/t_p$. They can be displayed in a graph with axes shown in Figure 6.

6. Compute the graph in the Operating Point.

As it follows from the aforementioned weaknesses, the main problem with reporting ENCCCs is that we can find out neither where is a particular latency/check ratio relevant, nor which latency/check ratio is relevant for a given application. We propose a way to solve this problem by offering a little bit of additional information besides ENCCCs graphs. To compute the graph for the Operating Point based on ENCCCs (ENCCC-OPs) we identify the following alternatives:

- the ENCCCs graph with the logic time cost given by the targeted/average value of L_p as interpolated from values for the different problem types p in the graph in Figure 6, or

- the ENCCCs graph with the logic time cost given by the value of L that is closest to the targeted value of L_p, among the different values of L used as LT^i for the logic times schemes evaluated in the experiments (see Equation 1).

One can also draw graphs representing the Equivalent Message Latencies in the Operating Point (EML-OP) from the ENCCC-OP graph, where each ordinate is divided by the latency/check ratio L of the graph. The axis of ordinates shows the number of (equivalent) message delays. This graph has the advantage that the the ordinate has an easy to understand meaning, namely the message latency in the targeted destination, which is readily available. Such systems of coordinates are illustrated in Figure 7.b. EMLs can also be plotted against abscissae showing different ratio latency/checks, to illustrate better how algorithms behave in areas neighboring the operating point.

Yet an additional metric can be obtained measuring the *logic simulated seconds*, where each event is measured in the number of (micro)seconds it lasts (in average) as observed during experimentation. This has the advantage over actual seconds that they can be replicated and verified by other researchers.

The term *operating point* comes from graphs depicting behavior of transistors. The operating point is the area of these graphs that is of real interest for an application.

The advantage of our method is that it can be performed using only a simulator, its results are reproducible, and can be applied to difficult to evaluate experimentally settings, such as remote Internet connections.

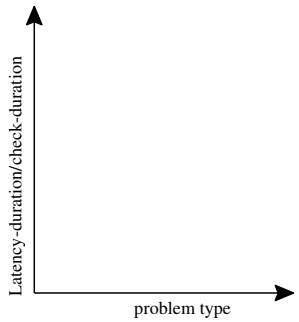

Figure 6: Axes for a graph displaying the expected logic latency L_p per problem type p (tightness or graph density).

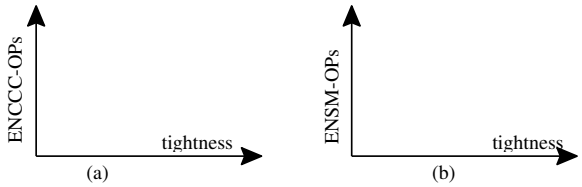

Figure 7: Performance graphs using (a) ENCCC-OPs and (b) EML-OPs.

Accounting for nogood validation.

Certain DCOP algorithms are not based on checking constraints repeatedly, but rather they compile information about constraints into new entities called *nogoods*. Afterward, these techniques work by performing inferences on such nogoods. Nogoods are a kind of constraints themselves. In such algorithms it makes sense to attribute costs to the different important operations on nogoods such as *nogood inference*, *nogood validity check*, and *nogood applicability check*. The new method for computing logic times at various message latencies is the Equation 3, where the coefficients of different nogood handling operations are selected based on a perceived complexity for those operations. The nogood inference operation is typically the most complex of these operations as it accesses two nogoods to create a third one (suggesting a logical cost of 3). Nogood-validity and nogood-applicability both typically involved the analysis of a nogood and of other data, local assignments and remote assignments, to be compared with the nogood (hence a logical cost of 2). These costs do not typically have an exact value since the sizes of nogoods vary within the same problem. A constraint check for binary constraints is cheaper than the verification of an average-sized nogood, and is given the logical cost of 1.

Why cost of checks varies with the problem size.

An interesting question raised by our experimental results is: Why do experiments reported here show that the cost associated with a constraint check varies with the size of the problem?

The cost associated with a constraint check (as measured

p (agents)	8	10	12	14	16	18	20	25	30	40
t_p (total seconds)	0.1404	0.1528	0.3012	0.5516	1.0068	2.5708	4.1176	47.7112	174.06	3767.38
$\#CC_p$ total checks	43887.8	38279.3	70279.4	116080	191501	381415	516835	4.1*10^6	10.9*10^6	132*10^6
$\frac{microseconds(t_p)}{check(\#CC_p)}$	3.199	3.992	4.286	4.752	5.257	6.74	7.967	11.47	15.98	28.4
L_p ($\frac{checks}{latency(200ms)}$)	62518.3	50103.8	46666.3	42088.5	38041.6	29672.9	25103.7	17437.3	12519	7041.1
ENCCC L=10^4 (10^3)	2228	1758	2910	4118	6008	11579	15001	95321	192750	1832356
ENCCC L=10^5 (10^3)	500	374	621	891	1310	2491	3195	21269	44598	439144
simulated time (s)	3.51	4.1	7.63	12.69	21.5	58.19	93.36	920.8	2711.8	48013

Table 1: Sample re-evaluation of ADOPT with our method. Columns represent problem size.

above) consist of an aggregation of the costs of all other operations executed by DCOP algorithm in preparation of the constraint check and in processing the results of the constraint check. Typically there are several data structures to maintain and certain information to validate, and these data structures may be larger with large problem sizes than with small problem sizes. The variation may also come from approximations in the way in which the cost of a constraint check is evaluated in comparison to operations for handling other data structure (such as nogoods [24]).

In certain situations, algorithms change their relative behavior in situations that are close to the operating point. Then precise measurements are important, and it makes sense to try to tune the logic time associated with each operation, in order to reduce the variation of the meaning of a unit of logic time with the problem size. One can approach this problem by trying many different combinations, or trying a hill climbing approach that tunes successively each of the parameters. One has to run complete sets of experiments for each of these possible costs (which is computationally expensive). A valuable future research direction consists in finding an efficient way of tuning these parameters.

However, a currently simpler alternative is to report efficiency in *simulated seconds* [1, 12], where each significant event is given a logic cost equal with the average time in microseconds as obtained from experiments.

Random message latencies.

Some researchers report that the introduction of random latencies has a strong impact on the efficiency for certain DCOP algorithms [7]. We have extensively experimented and we have found random latencies to have only around 5% impact (in both directions) on the number of sequential messages for the ADOPT [14] algorithm. In order to allow other researchers to replicate such experiments we propose to publish the seed with which we initialized the used random number generator (e.g., in our case the C library random number generator with seed 10000), as well as the equation used to distribute the latency uniformly in the range of expected latencies for the targeted application (in our case uniformly between 150ms and 250ms).

6. EXPERIMENTS

We will describe here how we conduct experiments with ADOPT [14], as an example of how our evaluation method can be applied to other algorithms. The illustration is based on a sample of Teamcore random graph coloring problems with 10 different sizes, ranging between 8 agents and 40 agents, with graph density 30%. The results are averaged over 25 problems of each size [14]. The targeted application scenario consists of remote computers on Internet.

Following the steps of our method we report the following:

1. The catalog message latency for our scenario is 200ms, varying between 150ms and 250ms (see [17]).
2. Simulated ADOPT with randomized latencies is implemented in C++ and runs on a the 700MHz node of a Beowulf (Linux Red Hat). The total time in seconds is given in the second row of Table 1.
3. The total number of constraint checks $\#CC_p$ for each problem size is given in the third row of Table 1.
4. The cost in (micro)seconds associated with each constraint check is computed as $t_p/\#CC_p$. It is given in the fourth row of Table 1.
5. The message-latency/constraint-check ratio (L_p) is computed by dividing the average latency found at Step 1 (200ms) by the items in the 4^{th} row. The results are given in the 5^th row of Table 1.
6. The operating point is defined by the fourth and fifth rows. The last step consists of reporting the results for this operating point (here we will use a Table rather than a graph, to make the processing more visible). We performed the experiments using several logic time systems, the available ones that are the closest to the obtained operating point are $L = 100,000$ and $L = 10,000$. In is now possible to re-run the experiments with all the L_p values found in our table. Here we will just report the results of the closest L, which is 10,000 for most problem sizes (one also can use $L = 100000$ for problems with 8 and 10 agents), see the $6^{th} and 7^{th}$ rows of Table 1. One can also interpolate the time between the predictions based on $L = 10,000$ and $L = 100,000$, function of the predicted L_p at each problem size.

   Next, for example, one can also report the simulated time (in simulated seconds) by multiplying each logic time (in ENCCC-OPs) with the corresponding cost per logic unit (here reported in the third row). We interpolate (liniarly) the time between the predictions based on $L = 10,000$ and $L = 100,000$, function of the predicted L_p at each problem size. We report the *simulated time* in the 8^th row of Table 1. This simulated time represents the average actual time (in seconds) that a problem of the corresponding size is expected to need in our *operating point*.

In Table 2 we show data for comparing between randomized versus constant latencies in simulation.

p (agents)	8	10	12	14	16	18	20	25	30	40
SMs (random)	793.2	631.48	1045.12	1480.36	2158	4172.56	5411.36	34284.2	69383.64	658731.36
SMs	736.16	602.76	1020.24	1438.64	2109.24	4125	5480.84	34067.2	68976.36	650307.08
total time (sec)	0.104	0.098	0.2776	0.4872	0.9636	2.55	4.1612	49.3452	181.2136	3813.5016
total checks	44637	40537	76936	126979	215303	428259	592419	4729413	12523233	151307539
μs/check	2.33	2.417	3.608	3.837	4.475	5.95	7.024	10.4	14.47	25.2
checks/latency	85841.9	82728.9	55429.8	52125.9	44687.2	33588.9	28473.5	19168.7	13821.5	7935.36

Table 2: ADOPT. First row is for randomized latency. Remaining rows are with constant message latency.

7. CONCLUSION

We have introduced a framework for enabling an unified representation of different logic clocks-based metrics used for efficiency evaluation of DCOPs. We analyzed all major metrics used in the past for evaluating algorithms for DCOPs by comparing them using this framework. We identify the meaning and the weaknesses of each currently common metric, and we propose a new methodology to analyze DCOPs, extending the one known as Equivalent Non-Concurrent Constraint Checks (ENCCCs). Our extension shows how to identify the ENCCCs graph that fits a given application scenario (named *operation point*). The obtained metric counts the equivalent non-concurrent constraint checks in the operation point (ENCCC-OPs) and its construction requires the evaluation of several other metrics, such as the total number of constraint checks and the total time to run the simulator as a centralized solver. We also show how to handle algorithm using nogoods.

Then we discuss remarkable experimental results on a simulator showing that cost associated with constraint checks can vary with the size of the problem even for the same implementation of teh same algorithm. We discuss the possible explanations, their implications, and how the issue can be handled (including open research directions).

8. REFERENCES

[1] Blinded.

[2] A. Chechetka and K. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *AAMAS*, 2006.

[3] Z. Collin, R. Dechter, and S. Katz. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*, 2000.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. Mc Graw Hill, 2003.

[5] J. Davin and P. J. Modi. Impact of problem centralization in distributed COPs. In *DCR*, 2005.

[6] M. Engineering. Synqnet. Technical report, Motion Engineering Inc, 2003. `www.motioneng.com/pdf/SynqNet_Tech_Whitepaper.pdf`.

[7] C. Fernàndez, R. Béjar, B. Krishnamachari, and C. Gomes. Communication and computation in distributed CSP algorithms. In *CP*, pages 664–679, 2002.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H.Freeman&Co, 1979.

[9] Y. Hamadi and C. Bessière. Backtracking in distributed constraint networks. In *ECAI'98*, pages 219–223, 1998.

[10] S. Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45(3):275–286, October 1990.

[11] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] R. N. Lass, E. A. Sultanik, P. J. Modi, and W. C. Regli. Evaluation of cbr on live networks. In *DCR Workshop at CP*, 2007.

[13] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *AAMAS02 DCR Workshop*, pages 86–93, 2002.

[14] P. Modi, M. Tambe, W.-M. Shen, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *AAMAS*, Melbourne, 2003.

[15] P. Modi and M. Veloso. Bumping strategies for the multiagent agreement problem. In *AAMAS*, 2005.

[16] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *AIJ*, 161, 2005.

[17] J. Neystadt and N. Har'El. Israeli internet guide (iguide). http://www.iguide.co.il/isp-sum.htm, 1997.

[18] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, 2005.

[19] A. Petcu, B. Faltings, and D. C. Parkes. M-dpop: Faithful distributed implementation of efficient social choice problems. *submitted to JAIR*, 2007.

[20] M.-C. Silaghi and B. Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence Journal*, 161(1-2):25–53, October 2004.

[21] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proc. of AAAI2000*, pages 917–922, Austin, August 2000.

[22] T. Walsh. Traffic light scheduling: a challenging distributed constraint optimization problem. In *DCR*, India, January 2007.

[23] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS*, pages 614–621, June 1992.

[24] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE TKDE*, 10(5):673–685, 1998.

[25] Y. Zhang and A. K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Third IEEE Symposium on Parallel and Distributed Processing*, pages 394–397, 1991.