Comparing performance of distributed constraints processing algorithms

*Amnon Meisels, Eliezer Kaplansky, Igor Razgon** *and Roie Zivan*

Dept. of Computer Science
Ben-Gurion University
Beer-Sheva 84105, Israel
{am,kaplan_e,irazgon,zivanr}@cs.bgu.ac.il

ABSTRACT

Search algorithms on distributed constraints satisfaction problems, DisCSPs, are composed of agents performing computations concurrently. The most common abstract performance measurement that has been universally adopted for centralized CSPs algorithms is the number of constraints checks performed. However, when it comes to distributed search, constraints checks are performed concurrently by all agents on the network and therefore a simple measurement of constraints checks is not adequate any more. In order to be able to compare the behavior of different algorithms, there is a need for a new distributed method to measure the search effort of a DisCSP algorithm.

A model for measuring concurrent constraints checks on distributed constraints satisfaction problems (DisCSPs) is presented. The advantages of the proposed model are clarified by a series of simple constraints network examples. An algorithm for computing the proposed measure, the *cumulative cost algorithm (CCA)*, is presented. A formal correctness proof of the CCA algorithm, according to the proposed model for concurrent constraints checks (CCCs) is presented.

1. INTRODUCTION

Distributed constraints satisfaction problems (DisCSPs) have been studied in the last decade and search algorithms for their solution have been proposed and discussed in the literature [2, 7, 9]. A DisCSP is composed of a set of agents, each with its own local constraints network. Agents cooperate by exchanging messages in order to find an assignment of values to all variables of all agents that is globally consistent (cf. [10]). Each agent owns exclusively a subset of variables. Variables are connected by constraints, either to local variables of the same agent or to variables of other agents.

Distributed search algorithms on distributed constraints networks (DisCSPs) are executed by a set of agents, where the sequence of computations of one agent depends in general on the intermediate results of other agents (cf. [10]). The intermediate results of agents' computations are distributed by exchanging messages between the agents. Usually, messages about assignments trigger the checking of constraints against received assignments by the receiving agent. If the check fails, the sending agent is requested to choose another assignment to its variables.

Distributed search algorithms on DisCSPs assume implicitly that all agents run the same algorithm, receiving messages about assignments of other agents and sending messages with their own assignments to other agents. Recent DisCSP algorithms are asynchronous and let all agents perform computations (and report about their resulting assignments) in a distributed manner [11, 1]. As a result, *part of the constraint checks of a distributed algorithm are performed concurrently*, and the fraction of this part (out of the total number of CCs) depends on the specific algorithm in question. The concurrency of a DisCSP algorithm transforms the performance measurement of such algorithms into a non trivial task.

Measurement of the computational efforts of an algorithm is mostly motivated by the need to estimate the time that will pass between the starting time of the algorithm and the time it returns with a satisfying solution. Standard measures of constraints processing algorithms (cf. [5, 6]) count the total number of constraints checks (CCs) that was executed. The number of CCs performed is generally accepted as a machine (and implementation) independent measure, because it counts the main computing operation of all backtracking algorithms (cf. [8]). It is also the generally accepted measure for the investigation of phase transitions in problem difficulty, for example (cf. [4, 13]). However, the simple counting of CCs is insufficient for DisCSPs, for two main reasons:

1. It does not take into account the time intervals in which agents run concurrently and the time intervals in which agents wait for other agents to finish their computations.

2. No account is taken of the time needed for an agent to send or receive a message, nor of the time for the message to be delivered.

The waiting periods were probably the trigger for the first discussion in the literature as to how one can measure and compare the computational efforts of DisCSP algorithms.

*contacting author: Tel +972-8-6477492, Fax: +972-8-6477650

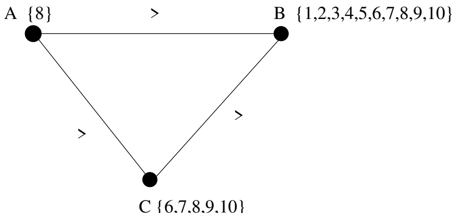

Figure 1: A Simple DisCSP.

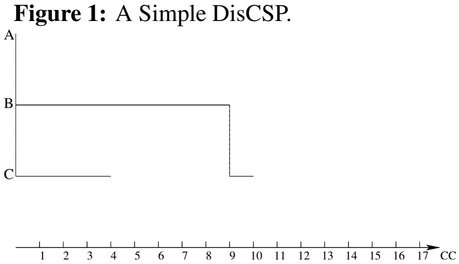

Figure 2: Time-plot of computations.

Yokoo [9, 10] suggests to divide the time that the algorithm consumes into *rounds*, where at the start of each round at least one agent has the necessary information to carry on its computation. The division of distributed computation into strict rounds is a simplification of the dependency between computations of different agents. An example in Section 1.3 demonstrates that this simplification generates an estimation of the concurrent search efforts that is not tight enough.

The present paper proposes a more general method of measurement, presents it in the form of a clear and general model and presents an algorithm for computing the measure. The proposed methods relies explicitly on the dependency among agents' computations and it is expected to generate a tighter (better) performance measure for DisCSP algorithm. In the following subsections the measurement problem is presented by analyzing a series of simple examples of DisCSPs.

1.1. A SIMPLE EXAMPLE WITH NAIVE METHODS

Consider the network in Figure 1 and assume agents to run a search algorithm that orders the agents lexicographically $< A, B, C >$. Assume the simplest algorithm - each agent waits for the agents preceding it to assign their variable, before it assigns its own variable. Upon receiving a message with former agent assignments, the agent attempts to assign values to its own variable, rejecting values that are inconsistent with former agents. Assume that agents scan their domain of values from left to right and after each assignment trial they perform constraints checks against former assignments of other agents. During the run of this search algorithm, agents perform the following series of actions:

1. Agent A assigns the value 8 to its variable and sends messages with this value to B and C.

2. Agent B receives a message from A with $A's$ assignment. It traverses the values in its domain from left to right, checking each assignment against the constraint with A. After performing 9 constraints checks, B assigns the value 9 to its variable and sends a message with this value to agent C.

3. Agent C gets a message from A with value 8 (with no loss of generality we assume that it receives this message almost simultaneously with B) and after 4 constraint checks assigns the value 9 to its variable. Afterwards it receives a message from agent B with the value 9 and after one additional constraint check assigns the value 10 to its variable.

4. The first solution is found.

A simple time plot of this series of operations is presented in Figure 2. Here the sequences of constraints checks (as a measurement unit for the search computations) that each of the three agents performs during the above sequence is plotted as a horizontal segment.

One can measure the search effort of this algorithm by summing all constraints checks of all agents. This naive measurement method will result in 14 constraints checks.

Another naive option is to take the maximal number of constraints checks performed by any of these three agents (assuming they all work concurrently) the result will be a total of 9 constraints checks.

A close inspection of the time plot of the computation segments of the agents in Figure 2 enables the following insight into this specific example: In the current algorithm, agent B performs 9 constraints checks. This computation triggers the last computation sequence of agent C. In our example, agent C can perform its first 4 constraints checks concurrently with agent B. A reasonable estimation for the total computation time of this algorithm (expressed in units of constraints checks), is the sum of 9 of agent B plus the final one of agent C, producing a result of 10 concurrent constraints checks altogether. In this simple example, none of the above naive measurement methods (i.e. sum of all CCs of all agents, or maximum of all CCs of all agents) can deliver this tight estimation.

1.2. DIVIDING CONCURRENT SEARCH INTO ROUNDS

Asynchronous search algorithms on DisCSPs can be seen as a loop that each agent is running: The agent receives a message, computes assignments based on these messages, and then sends one or more messages, if needed. If one adds an assumption that all messages are delivered instantly and in the order they were sent, then the concurrent computation can be divided into *rounds*. This is the model that was proposed by Yokoo [9, 10]. This division enables one to measure the computing efforts of the search algorithm as a sum of the concurrent constraints checks that are performed in each round. A natural measure of the concurrent constraints checks in each round is the *maximum over all the agents* for that round.

The constraints checks performed by each agent in each round *depend on assignments of a subset of the agents during the former round*. Dividing the computation of a distributed search algorithm for DisCSP into a series of rounds is somewhat of a simplification. Any sequence of constraints checks of agent may depend on assignments of different agents in many ways.

If we apply the rounds model of Yokoo to our simple example in Figure 1 and Figure 2 we can see that in the first round only agent A can perform its assignment. This round has zero constraints checks. In the second round both agents B and C perform computations based on the message received from agent A. Both agents search for an assignment that is consistent with the assignment of agent A. In this, second round, agent B and C perform 9 and 4 constraints checks correspondingly. Both of these computations are done concurrently and therefore we take the maximum of 9 CCs as the computational cost of this round.

The third round of computations involves only agent C, that has to find a consistent assignment to the new assignment of agent B. This needs only one constraint check, so the cost of this round is 1.

The total number of *concurrent constraints checks* (CCCs) is the sum of all CCCs of all rounds of the computation, producing, in our example, the final result of $0+9+1 = 10$ concurrent constraints checks. Note, that this result is identical to our intuitive calculation in the former subsection.

The general idea behind the model of computing concurrent constraints checks in *rounds* is a notion of *dependency*. When we say that an event b **depends on an event** a, we mean that **event** b **cannot start before** a **finishes**. *Dependency* relations define parts of the computation of agents that cannot be performed in parallel. For example, the search for an assignment of agent C (in Figure 1) that is consistent with the assignment of agent B, must be performed *sequentially* to that of agent B. It is easy to see that computing the search efforts by simply summing the CCs of all agents assumes **total dependency** of computations of all different agents. Where the other extreme approach that take the maximum number of CCs performed by all agents (over the whole search process), assumes a **total independence** of all agents. The model of *rounds* assumes that the dependent events are complete rounds. We now turn to a more complex example of a DisCSP problem to demonstrate that a refinement of the dependent events is needed.

1.3. A MORE COMPLEX EXAMPLE FOR COMPUTING CCCS

Assume that the agents of the DisCSP in Figure 3 are ordered by $< A, D, B, E, C >$ and run the same algorithm as in the previous example. The sequence of computations, for this example will be as follows:

1. Agent A assigns the value 0 to its variable and sends messages with this assignment to B and D.

2. Agent B gets the message from A and after 11 CCs assigns the value 0 to its variable. Then it sends a message with this value to agent C.

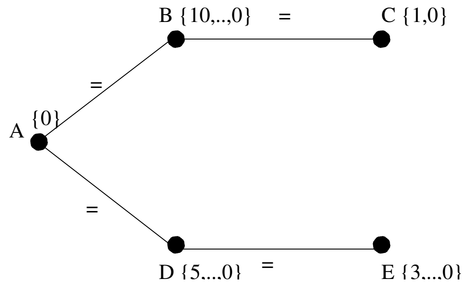

Figure 3: A more complex example.

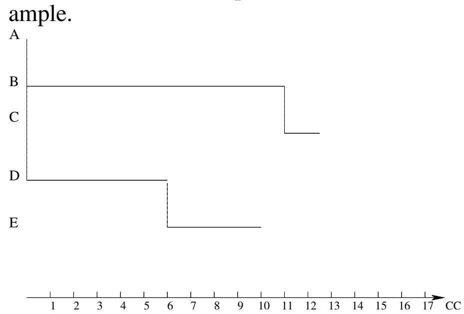

Figure 4: Time-plot of CCs.

3. Agent D gets the message from agent A and after 6 CCs it assigns the value 0 to its variable. Next, it sends a message with this assignment to agent E.

4. Agent C gets a message from agent B and after 2 CCs assigns the value 0 to its variable.

5. Agent E gets a message from agent D and after 4 CCs assigns the value 0 to its variable.

The time plot of computations of all agents for this example, is shown in Figure 4. Applying the rounds model of Yokoo, the computation is divided into 3 rounds:

1. *assignment of A:* The computation effort here is 0 CCs.

2. *assignment of B and D:* Here the concurrent computation effort is 11 CCCs
$max(B, D) = max(6, 11) = 11$.

3. *assignment of C and E:* Here 4 CCCs are executed concurrently
$max(C, E) = max(2, 4) = 4$.

The total computation effort according to the rounds model is, therefore, the sum of 0, 11 and 4, which amounts to 15 *concurrent constraints checks* (CCCs). However, in this example agents B and C act independently of agents D and E and vice versa. Inspecting the time plot in Figure 4, one can see two separate branches of computations.

One branch consists of agents A, B, C and the other branch of A, D, E. The first branch consume 13 CCCs and the second branch consumes 10. A good measure of the concurrent computation effort is the maximum of these two independent branches of computations. Therefore this more tight measurement results in 13 CCCs.

This example demonstrates that the division of computations by agents in a distributed search algorithm into rigid rounds, is too simplistic. The present paper proposes a more general dependency definition for distributed constraints processing algorithms and introduces an algorithm for computing the total number of concurrent constraints checks (CCCs). The CCCs-computing algorithm is proven to be a realization of the model definition.

The rest of the paper is organized as follows. Section 2 defines a general model for dependency of sequences of constraints checks computations among agents and Section 3 presents an algorithm for computing concurrent consistency checks, the *cumulative cost algorithm (CCA)*. A proof that the computation method of CCCs realizes our model is provided in section 4. The new model realization and its behavior are discussed in section 5.

2. A MODEL FOR CONCURRENT CONSTRAINTS CHECKS

The basic concept in our model is the *sequence of constraint checks* (SCC). A *SCC* is a set of constraint checks executed by a single agent, in one sequence, without sending or receiving any message during the execution of the sequence. Using the SCC object as a building block, one can assemble them together into a consistent model based on the **dependency** concept. The dependency between SCC objects is generated by the sending and the receiving of messages among agents.

In the proposed model, agent activities, while running a distributed search algorithm on a DisCSP, can be represented by two classes of objects:

- The set of messages sent and received by an agent.
- The set of all *sequences of constraint checks* (SCCs).

Definition 1 *An **event** of an agent A is either sending a message, receiving a message, or performing one SCC.*

Definition 2 *An event x of agent A is **Unary-Dependent** on event y of the same agent if the event x occurs* **after** *event y and no event z of agent A exists, such that z is after x and y is after z.*

Definition 3 *Event x of agent A is **Binary-Dependent** on event y of agent B if the event y is the sending of some message M from B to A and x is the event of receiving the message M, by agent A.*

Definition 4 *Event x is **Chain-Dependent** or **dependent** on event y if one of the following three conditions holds:*

- *x is Unary-Dependent on y.*
- *x is Binary-Dependent on y.*
- *There is a sequence of events $x_1, ...x_k$, where $x_1 = y$ and $x_k = x$, such that for all $i = 2 \dots k$, x_i is either Unary-Dependent or Binary-Dependent on x_{i-1}.*

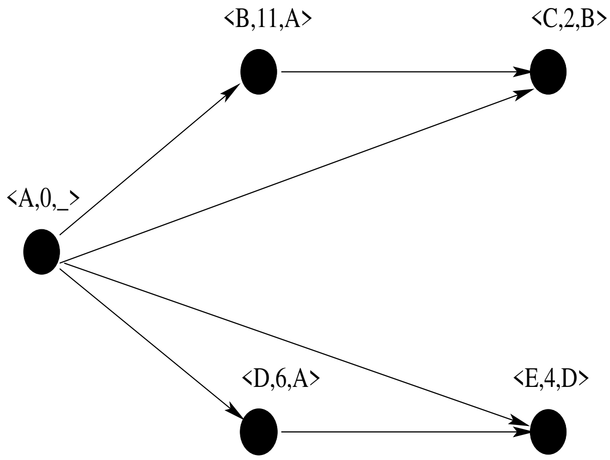

Figure 5: Dependencies Graph for the example in figure 3. Black circles represent SCC's and white circles represent messages.

Definition 5 *For each SCC object x define a set of SCCs, the **Dependency Set** $\Gamma^-(x) = \{y_1, y_2, \dots, y_n\}$, such that x is chain-dependent on any $y_i \in \Gamma^-(x)$ and $\Gamma^-(x)$ includes any SCC that x is chain-depended on.*

Next, we define the cost of computing, for a single SCC and for sets of SCCs:

Definition 6 *The **S_Cost** of SCC x is the number of constraints checks that the agent performs during this SCC.*

The **Cumulative Cost (C_Cost)** of a SCC x is defined recursively, to be the sum of costs of $\Gamma^-(x)$.

Definition 7 *If a SCC x is not Chain-Dependent on any other SCC then*

$$\textbf{\textit{C\_Cost(x)}} = S\_Cost(x) \tag{1}$$

otherwise

$$\textbf{\textit{C\_Cost(x)}} = max_{y \, \in \, \Gamma^-(x)}(C\_Cost(y) + M\_Cost) + S\_Cost(x) \tag{2}$$

The term M_Cost that appears in definition 7 is the cost of message transfer, in units of CCs. The *Total Number of Concurrent Constraints Checks (TCCC)* can be calculated recursively by using the cumulative costs of all SCCs. The *TCCC*, which represents the computing effort of a Distributed Search algorithm, is the maximal cumulative cost C_Cost over the set of all constraints checks sequences ($SCCset$). More formally:

Definition 8 *The total number of concurrent constraints checks that a distributed algorithm performs during search is:*

$$\textbf{\textit{TCCC}} = max_{x \in SCCset}(C\_Cost(x)) \tag{3}$$

To make the model clear, let us apply it to the scenario described in Figures 3, 4. The events of the application of the algorithm are described in the form of a *Dependencies Graph*, defined on the set of events where (a, b) is an edge of this graph if the event b is either Unary-Dependent or

Binary-Dependent on the event a. Each path in this graph is a *dependencies path*. The resulting graph is shown in Figure 5, Where black circles represent SCC's and white circles represent messages. Each message passing consists of two events: sending and receiving of the message, but for simplicity we show messages as atomic events. The SCC nodes of the dependencies graph in Figure 5 are labeled by triplets $< Node, S\_Cost, Dep\_Agent >$, where $Node$ is the ID of the agent, S_Cost is the cost (in CCs) of the SCC represented by this node, and Dep_Agent is the name of the agent that sent the message that triggered this SCC.

The first event in this scenario is the zero-cost SCC performed by agent A. It is easy to see that the cumulative costs of SCC's performed by agents B, C,D,E are 11, 13, 6, 10 respectively, therefore the TCCC is 13 which is equal to our intuitive estimation.

Note an important characteristic of the proposed model: Event b depends on event a *iff* the computation of event a must finish *before* the computation of b starts. This partial temporal ordering of computations at different agents serves as the basis for a realistic model of distributed computation. Each dependency path includes a complete ordering of events and enables a simple calculation of total computation effort along each path. This model bears a resemblance to time-ordering, as proposed by Lamport in [3]. In the next section we design an algorithm to compute the concurrent constraints checks (CCCs) measure on a DisCSP. The algorithm uses agents' CC counter values to propagate the measure of concurrent constraints checks. It uses CC counter values in analogy to Lamport's time-stamps technique [3].

3. THE CUMULATIVE COST ALGORITHM (CCA)

Consider the dependencies graph in Figure 5. In order to calculate the total number of concurrent constraints checks (TCCC) that represents the computation efforts of the algorithm in the example, all possible paths to the node with the final solution have to be followed. All edges in the dependencies graph in Figure 5 represent messages among agents. Each message triggers the computation that is represented by the cost of the SCC, to which the edge is leading.

The recursive definition of equation 2, for the cumulative cost, can be constructed on the fly during the computation of the DisCSP algorithm. To do that, each agent has to maintain a counter of CCs. The counter represents the agent view of the computation cost up to this point. This counter is updated at the end of each SCC computation. Messages include, in addition to the assignments of agents, also the value of the counter of the sending agent. Since a distributed search algorithm is typically asynchronous, many agents receive messages that have different counter values written on them, corresponding to several paths that lead to a node of a given agent on the dependencies graph.

The basic scheme of the proposed algorithm for calculating the total cumulative cost, is that each agent maintains a local counter of its view of the C_Cost. In the next section these counters will be shown to be equal to the C_Cost of the SCC node. We term these counters of the agents $AgentC\_Cost$. Each message that an agent sends includes the current value of the $AgenC\_Cost$ of the sending agent. We term the counter value on the message, $SenderC\_Cost$. Upon receiving a message and before performing a sequence of constraints checks, the receiving agent updates its counter by comparing it to the $SenderC\_Cost$ that is written on the received message. After the agent performs the next SCC that this message triggered, the agent updates its $AgentC\_Cost$ counter to include the constraints checks it just performed in this SCC.

One way to understand the idea of the CCA algorithm is to think of it in terms of achieving a total ordering of events, where events are the 3 SCCs and sending and receiving of messages. Our constraints checks counters behave exactly like Lamport's idealized clocks in [3]. They always increase in value for two events of the same agent. Following the algorithm for clock synchronization [3], the agents in the CCA algorithm send CC counter values that serve as timestamps. The result is a total ordering of events, as proven in [3], and therefore a total count of CCCs (as defined in section 2).

Algorithm 1 CUMULATIVE COST ALGORITHM (CCA)

Each agent initializes it's $AgentC\_Cost$ counter to 0.
When an agent sends a message it includes in the message the value of it's $AgentC\_Cost$ counter.
After an agent performs any SCC it updates it's $AgentC\_Cost$
$AgentC\_Cost = AgentC\_Cost + S\_Cost$
if An agent receives a message with a counter $SenderC\_Cost$ **then**
  if $SenderC\_Cost + M\_Cost > AgentC\_Cost$ **then**
    $AgentC\_Cost = SenderC\_Cost + M\_Cost$ $\{M\_Cost$ is the message transfer cost$\}$
  end if
end if
if Solution is found **then**
  Return the maximal $AgentC\_Cost$ over all agents.
end if

The target of the CCA algorithm is to return the computing cost of a given a DisCSP algorithm. In order to incorporate CCA into a DisCSP algorithm one needs to perform the following steps:

- The first line of the agent's code initializes $AgentC\_Cost$ to 0.
- After every constraints check performed by an agent, the $AgentC\_Cost$ counter is to be increased by 1.
- All messages contain an additional field - $SenderC\_Cost$.

- When an agent sends a message, it sets $SenderC\_Cost$ on the message to the value of its $AgentC\_Cost$.
- When an agent receives a message with a $SenderC\_Cost$ counter value plus transition cost larger than its own $AgentC\_Cost$ counter, it updates its $AgentC\_Cost$ counter to the value of the $SenderC\_Cost + M\_Cost$.

Let us trace the behavior of the CCA on the example in Figures 3, 4. Initially the counter of agent A is 0. Agent B receives a message from A with counter 0, after performing SCC it updates its counter to 11. Agent C receives a message from B with counter 11 and then updates it to 13 as the result of the execution of its SCC. In the same way, the counters of agents D and E will eventually be updated to 6 and 10 respectively. The maximal counter value is 13 and this value will be returned as the TCCC of the computation. The result of applying the CCA is equal to the result of application of our model.

In the next section we prove this claim formally. The proof is given explicitly because counting CCCs is computationally different than ordering events in a distributed set of agents. Moreover, the CCA algorithm can include in its cost calculation the cost of messages.

4. REALIZATION OF THE MODEL BY THE ALGORITHM

The goal of this section is to show that the inclusion of the CCA algorithm in any distributed search algorithm for DisCSP will compute the $TCCC$ as defined by our model in Equation 3. This goal can be stated formally by the following theorem:

Theorem 4.1 *Given that message transition cost M_Cost is zero for all messages, the CCA algorithm returns the total computing effort TCCC, as defined by Definition 8.*

A few lemmas are needed for the proof of this theorem. Let SV be the set of all the values of $AgentC\_Cost$ that was created during the run of the DisCSP algorithm that implements the CCA algorithm. Next, divide SV into equivalence classes by the use of the equality relation. Each value in the SV set is produced by one of the following two operations:

Update by message: An agent updates its old $AgentC\_Cost$ value by some larger $SenderC\_Cost$ value that it reads from a message.

SCC execution: The $AgentC\_Cost$ is updated *immediately after* the execution of a SCC sequence, by adding the S_Cost to the current value of the $AgentC\_Cost$ counter.

The value created by the *Update by message* method is not new. In other words, at least one element of SV with the same value already exists in SV. This is because every message shows an existing $AgentC\_Cost$ of some agent. This observation is the essence of the first Lemma.

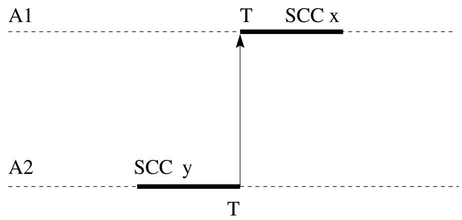

Figure 6: Dependency between two SCC.

Lemma 1 *In every equivalence class of SV there exists at least one element that was created by SCC execution of some agent.*

Let VT be the set of messages causing the $AgentC\_Cost$ value of some agent to be T. Let M be some message $\in VT$, passing from agent A to agent A'. This implies that the $AgentC\_Cost$ value of A just before sending M was T. This T value could be obtained in two ways: A got some message M' that set its $AgentC\_Cost$ value to T or performed some SCC.

In the first case, the message M' also belongs to VT. For the second case, observe an interesting fact: each message $\in VT$ is either Binary-Dependent on some other message that is also $\in VT$ or Chain-Dependent on some SCC causing the $AgentC\_Cost$ value of the sending agent to be T. Consider the graph of Binary-Dependencies induced by elements that are $\in VT$. This graph is a collection of DAG's. Therefore, there are messages of VT, that are independent of other messages of this set, and every message in VT is either "independent" or dependent on some "independent" message. But, by the fact observed above, every independent message in VT is dependent on some SCC causing the $AgentC\_Cost$ value of the agent performing it to be T. So, every message in VT is dependent on some such SCC. As can be see from Figure 6, this implies that if an agent starts some SCC and its $AgentC\_Cost$ value is equal to $T > 0$ then this T value is the result of the finishing value of another agent just **after** some SCC. More formally:

Lemma 2 *If the $AgentC\_Cost$ value of agent A_1, just before execution of some SCC x is equal to T, and $T > 0$ then there exists at least one SCC y performed by some agent A_2 such that the $AgentC\_Cost$ value just after the execution of y is equal to T and that SCC x is chain-dependent on SCC y.*

Clearly x depends on the event causing the $AgentC\_Cost$ value to be T. This event is either the SCC y, or a message dependent on the SCC y, by the discussion above. In both of these cases the lemma holds. □

Dependency of events implies order in time. This means for our case that the path of dependencies in the dependencies graph is a growing series of values of CCs. More formally:

Lemma 3 *If SCC x_1 of agent A_1 depends on SCC x_2 of agent A_2 then the $AgentC\_Cost$ value of A_1 just before execution of x_1 is not less then the $AgentC\_Cost$ value of A_2 just after the execution of x_2.*

The proof is straightforward. If x_2 chain-depends on x_1, then there is some sequence of events E, where each event in the sequence E is either Unary-Dependent or Binary-Dependent on its predecessor. The first event in E is the execution of the SCC x_1 and the last event in E is the execution of x_2.

Let A_i be the agent initiating the i-th event in the sequence. Clearly, its $AgentC\_Cost$ counter value will not be less then the counter value of A_{i-1} after occurrence of the $i-1$-th event. So, the counter of the agent initiating the last event in the sequence, **before the event** will not be less then the counter of the agent initiating the first event **after** its occurrence, by the transitivity of the $\leq$ relation $\Box$.

Every path in the dependencies graph is a growing series of values of CCs, therefore the CCs value in each node is the largest value on this path up to this node. Many dependencies paths may leads to the same node, and the above conclusion is true for all of these paths. The immediate conclusion is that the CCs value at a mode is the maximum value over all the paths that leads to that node. The next lemma states the above conclusion in exact terms and shows that if some $AgentC\_Cost$ value is created as the result of performing some SCC then this value is the cumulative cost C_Cost of this SCC.

Lemma 4 *If a value of the counter $AgentC\_Cost = T$ of some agent A is produced as the result of execution of some SCC x then $T = C\_Cost(x)$ (as defined in Definition 7).*

Proof: Let us assume that there are no SCC's of zero cost and reformulate the lemma in the following way: if some agent A with current $AgentC\_Cost$ value of T' performs a SCC x then its $AgentC\_Cost$ value T after performing x is equal to $C\_Cost(x)$.

Because there is a finite number of possible values in SV for T', let us apply induction on these values in increasing order. If $T' = 0$ then clearly x does not depend on any other SCC because the dependency of x on any other SCC y contradicts Lemma 3. The counter value of A after performing x is equal to $S\_Cost(x)$ which is the same as $C\_Cost(x)$ for this case.

Otherwise, for $T' > 0$, assuming that the claim of the lemma holds for all $T' < Z$ ($Z \in SV$), we will show it hold also for the case $T' = Z$. By Lemma 2 there is some SCC y performed by agent A', where x depends on y and the value of A' after performing y is T'. Because the counter value of A' just before execution of y is less than T', $T' = C\_Cost(y)$ by the induction assumption. From $T = T' + S\_Cost(x)$ we derive that $T = C\_Cost(y) + S\_Cost(x)$.

To show that $T = C\_Cost(x)$ we have to show that among all SCC's that x is dependent on, y is a SCC with the maximum cumulative cost. Assume that this is not true and there is some SCC z of agent A'' such that x depends on and $C\_Cost(z) > C\_Cost(y)$. Let T'' be the SCC z value of agent A'' after performing z. By Lemma 3 $T'' \leq T'$. Therefore, by the induction assumption $T'' = C\_Cost(z)$. Consequently $C\_Cost(z) \leq T'$ and $C\_Cost(z) \leq C\_Cost(y)$ which contradicts our assumption. Therefore $T = C\_Cost(x)$. $\Box$

The proof of Theorem 4.1 is now immediate. By Lemma 4 all $AgentC\_Cost$ values produced by SCC's are the C_Cost's of these SCCs. By Lemma 1 these values are representatives of SV. So, the set of all C_Cost's is equal to the set of all SV. Consequently, their maximal element is also equal.

Corollary 1 *The claim of Theorem 4.1 holds also for messages which have non-zero costs.*

Proof: To adapt the proof of Theorem 4.1 to non-zero cost messages, S_Costs may include the cost of the message received by the agent performing the SCC. In particular, for SCC x performed by agent A there may be two cases

- The predecessor of x with maximal C_Cost is performed by the same agent. In this case the $S\_Cost(x)$ remains as is.
- The predecessor of x with maximal C_Cost, is performed by another agent B. In this case, we add to $S\_Cost(x)$ the massage cost from B to A

The above rule ”hides” non-zero message costs within the cost of the consecutive SCC and the cost of message sending itself remains zero. This way, both the model and the proof of Theorem 4.1 remain valid. $\Box$

5. DISCUSSION

A new model for the measurement of concurrent search effort on DisCSP search algorithms was proposed. The model incorporates a dependencies graph among agents performing sequences of constraints checks and exchanging messages. The proposed model measures search effort by counting concurrent constraints checks (CCCs) on the dependencies graph. A simple algorithm for realizing the model within any distributed search algorithm, the cumulative cost algorithm (CCA), was defined and proven to realize the model. The implementation of the CCA within a DisCSP algorithm is shown.

The proposed CCA can be implemented easily within a distributed algorithm, without any need for strong assumptions on order of arriving messages and on instantaneous arrival of messages. In a realistic network of agents, where messages can arrive in any order and delay the distributed computation, the CCA is useful, it represents the actual computation.

By carrying counters within messages and updating counters by agents according to the longest delayed message counter, the CCA will measure the real performance of the tested algorithm. It does not need any artificial division of the run of a DisCSP algorithm into rounds. The actual computation paths (sequences) among agents are measured and the costliest path defined as the global cost (in CCCs) of the computation. The proposed model has three main advantages:

- The definition of dependency among agents computations is realistic. Two events are dependent in the model if and only if there is a real dependencies path between these two events.

- The implementation of the model, in terms of the CCA, is very simple. It does not need any simplification assumptions on the sequentiality of a simulator, nor additional data structures. This advantage is important, because the implementation of a distributed algorithm in a sequential environment is a non-trivial task.

- The CCA can be easily implemented within a real distributed system, with concurrently running agents. Thus we get a possibility to compare a real, not simulated, computational effort.

The CCA was implemented on the asynchronous backtracking (ABT) algorithm [11, 12] and run on randomly generated DisCSPs. One interesting test of the behavior of the proposed measure of CCCs is to compute it on runs of the ABT and compare it to other measurements of the computation. In our preliminary experiments we have performed the following analysis. Take the path of agents computations that produced the TCCC of the run of the ABT and assign each of the SCCs on this ("costliest") path to a round of computation. Compare the S_Cost of the node on the path to the maximal cost of all agents on the same round. Our preliminary experiments show that a partial set of nodes (agents) on the path that determines the overall TCCC have costs that are lower than the maximum of the round they belong to. This fact leads naturally to a tighter estimation of the concurrent search effort of the run of the ABT on the random DisCSPs we generated.

REFERENCES

[1] C. Bessiere, A. Maestre, and P. Messeguer. Distributed dynamic backtracking. In *Workshop on Distributed Constraint of IJCAI01*, 2001.

[2] M. Yokoo et. al. Distributed constraint satisfaction for formalizing distributed problem solving. In *IEEE Intern. Conf. Distrb. Comp. Sys.*, pages 614 – 621, 1992.

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

[4] J. Larrosa and P. Meseguer. Phase transition in max-csp. In *ECAI-96*, pages 190–194, Budapest, Hungary, 1996.

[5] P. Prosser. Binary constraint satisfaction problems: some are harder than others. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 95–99, Amsterdam, 1994.

[6] B. M. Smith. Phase transition and the mushy region in csp. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 100–104, Amsterdam, The Netherlands, 1994.

[7] G. Solotorevsky and E. Gudes. Algorithms for solving distributed constraint satisfaction problems (dcsps). In *Proceedings of AIPS96*, 1996.

[8] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[9] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proc. 1st Intrnat. Conf. on Const. Progr.*, pages 88 – 102, Cassis, France, 1995.

[10] M. Yokoo. *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems.* Berlin Springer, 2001.

[11] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Data and Kn. Eng.*, 10:673–685, 1998.

[12] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Inter. Conf. Multi Agent Systems*, pages 372–379, La Villette, Paris, July, 1998.

[13] M. Yokoo, K. Hirayama, and K. Sycara. The phase transition in distributed constraint satisfaction problems: First results. In *Proc. CP-2000*, pages 515–519, Singapore, 2000.