Latency-Aware Local Search for Distributed Constraint Optimization*

Ben Rachmut
Ben-Gurion University of the Negev
Beer Sheva, Israel
rachmut@post.bgu.ac.il

Roie Zivan
Ben-Gurion University of the Negev
Beer Sheva, Israel
zivanr@bgu.ac.il

William Yeoh
Washington University in St. Louis
St. Louis, MO, USA
wyeoh@wustl.edu

ABSTRACT

Most studies investigating models and algorithms for distributed constraint optimization problems (DCOPs) assume messages arrive instantaneously or within a (short) bounded delay. Specifically, distributed local search DCOP algorithms have been designed as synchronous algorithms, performing in an asynchronous environment, i.e., algorithms that perform in synchronous iterations in which each agent exchanges messages with all its neighbors. This is true also for an anytime mechanism that reports the best solution explored during the run of synchronous distributed local search algorithms. Thus, when the assumptions on instantaneous message arrival are relaxed, the state of the art local search algorithms and mechanism do not apply.

In this work, we address this limitation by: (1) Investigating the performance of existing local search DCOP algorithms in the presence of message latency; (2) Proposing an asynchronous monotonic distributed local search DCOP algorithm; and (3) Proposing an asynchronous anytime framework for reporting the best solution explored by non-monotonic asynchronous local search DCOP algorithms. Our empirical results demonstrate that, up to some extent, message delays have a positive effect on distributed local search algorithms due to increased exploration. The asynchronous anytime framework proposed, allows a maximal benefit from such latency based explorative heuristics.

KEYWORDS

Distributed Constraint Optimization; Distributed Local Search; Message Latency

ACM Reference Format:
Ben Rachmut, Roie Zivan, and William Yeoh. 2021. Latency-Aware Local Search for Distributed Constraint Optimization. In *Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), Online, May 3–7, 2021*, IFAAMAS, 9 pages.

1 INTRODUCTION

Recent advances in computation and communication have resulted in realistic distributed applications, in which humans and technology interact and aim to optimize mutual goals (e.g., IoT applications). A promising multi-agent approach to solve these types of problems is to model them as *Distributed Constraint Optimization Problems* (DCOPs) [7, 12, 15], where decision makers are modeled as *agents* that assign *values* to their *variables*. The goal in a DCOP is to optimize a global objective in a decentralized manner. Unfortunately, the communication assumptions of the DCOP model are overly simplistic and often unrealistic: (1) All messages arrive *instantaneously* or have *very small and bounded delays*; and (2) Messages sent *arrive in the order that they were sent*. These assumptions do not reflect real-world characteristics, where messages may be disproportionally delayed due to different bandwidths in different communication channels.

Because solving DCOPs optimally is NP-hard [12], considerable research effort has been devoted to developing incomplete algorithms for finding good solutions quickly [2, 5, 9, 10, 13, 16, 21, 22]. Distributed local search algorithms (e.g., *Distributed Stochastic Algorithm* (DSA) [22] and *Maximum Gain Messages* (MGM) [10]) are simple incomplete algorithms with a naturally distributed structure. Although they commonly offer no (or little) quality guarantees, they were empirically found to produce high quality solutions [10, 21, 22]. An anytime mechanism allows one to report the best solution explored by such algorithms, even when they perform rapid exploration [14, 24].

Unfortunately, these simple distributed local search algorithms and the anytime framework take advantage of the common simplistic communication assumptions discussed above. To make matters worse, they even depend on them for achieving some desired properties (e.g., monotonicity, convergence, etc.). As a result, the guarantees for achieving these properties may no longer hold when communication is unreliable.

In this paper, we make the following contributions:

(1) We analyze the performance and properties of standard local search algorithms, after they are adjusted to perform in scenarios including message latency.
(2) We propose an asynchronous local search algorithm that is monotonic and its convergence is guaranteed to a 1-opt solution (similar to the properties of the MGM algorithm [10]). We demonstrate empirically that the proposed algorithm converges faster than synchronous MGM in the presence of message latency.
(3) We propose an asynchronous anytime mechanism that allows any local search algorithm performing in an environment with imperfect communication, to report the best solution it has explored during its run.
(4) We show that, up to some extent, message latency can have a positive effect on local search algorithms. The reason is that it increases the amount of exploration, albeit unintended, and, consequently, local search algorithms can traverse better solutions, which will be reported via the anytime mechanism we proposed.

*This research is partially supported by US-Israel Binational Science Foundation (BSF) grant #2018081 and US National Science Foundation (NSF) grant #1838364.

Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.), May 3–7, 2021, Online.

2 RELATED WORK

We present in this section existing work on message latency in distributed constraint reasoning studies. There is a very limited amount of work on the study of communication times in the context of DCOPs. Researchers have investigated the importance of communication times in evaluations of DCOP algorithms [4, 17, 18]. For example, Cruz et al. conducted experiments where agents are located physically apart in different machines connected by LAN [4]. They observed that communication times are orders of magnitude larger than what is typically assumed in the DCOP community, thereby issuing a call of action to better investigate this area. Our research, in a large part, is in response to this call. Another thread of relevant research is the work done by Tabakhi et al. [17, 18]. In their work, they used realistic network simulators, such as ns-2 [11], to simulate the wireless communication times between agents. There are three key differences between these related works and ours: (1) Our empirical evaluations systematically varied the different degrees of message delays while evaluations of Cruz et al. were for a fixed setting only; the reason is that their evaluations are on actual physical hardware while ours are on simulations; (2) We focus on *incomplete* local search DCOP search algorithms in this paper while both Cruz et al. and Tabakhi et al. focused on *complete* DCOP search algorithms; (3) Finally, we proposed new variants of DCOP algorithms that are more robust to message delays in this paper while they focused only on evaluating existing algorithms.

In the neighboring area of *Distributed Constraint Satisfaction Problems* (DisCSPs), researchers have investigated the impact of message delays as well. For example, Zivan and Meisels [23] proposed a method for simulating any type of message delays in which all messages sent by agents are first delivered to an additional mailing agent. This agent decides on the delay of each message and delivers the messages only after the selected delay time has passed. Time was counted in terms of logical steps of the algorithm (e.g., constraint checks or iterations of the algorithm). Our simulator for DCOPs can be seen as an extension of their simulator for DisCSPs (see Section 5).

The impact of communication delays on DisCSP algorithms was also studied by Fernàndez et al. [6]. Similar to our findings, they found that random delays can actually improve the performance and robustness of AWC (an asynchronous complete algorithm for DisCSPs). Wahbi and Brown [19] decoupled the communication graph from the underlying constraint graph of the problem and studied the effect of different communication graph topologies on ABT and AFC-ng. In this work, the communication load was measured by the number of transmission messages during the algorithm execution *(#transmission)* and the computation effort that takes the message delay into account, which was measured by the average of the equivalent non-concurrent constraint checks *(#ncccs)* [3]. The main difference with our work is that they focused on complete DisCSP algorithms while we focused on incomplete DCOP algorithms in this paper. Nevertheless, we are encouraged by the fact that in different scenarios and for different algorithms, it has been shown that message latency can improve a distributed search algorithm's performance.

3 BACKGROUND

In this section, we present background on DCOPs, distributed local search algorithms for DCOPs, and the anytime mechanism that can be used in conjunction with distributed local search algorithms to keep track of the best solution found.

3.1 Distributed Constraint Optimization Problems (DCOPs)

Without loss of generality, in the rest of this paper, we will assume that all problems are minimization problems as is commonly assumed in the DCOP literature [7]. Thus, we assume that all constraints define costs and not utilities. Our description of a DCOP is also consistent with the definitions in many DCOP studies [8, 12, 15].

A DCOP is a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$, where $\mathcal{A}$ is a finite set of agents $\{A_1, A_2, \ldots, A_n\}$; $\mathcal{X}$ is a finite set of variables $\{X_1, X_2, \ldots, X_m\}$, where each variable is held by a single agent (an agent may hold more than one variable); $\mathcal{D}$ is a set of domains $\{D_1, D_2, \ldots, D_m\}$, where each domain D_i contains the finite set of values that can be assigned to variable X_i and we denote an assignment of value $d \in D_i$ to X_i by an ordered pair $\langle X_i, d \rangle$; and $\mathcal{R}$ is a set of relations (constraints), where each constraint $R_j \in \mathcal{R}$ defines a non-negative *cost* for every possible value combination of a set of variables and is of the form $R_j : D_{j_1} \times D_{j_2} \times \ldots \times D_{j_k} \rightarrow \mathbb{R}^+ \cup \{0\}$. A *binary constraint* refers to exactly two variables and is of the form $R_{ij} : D_i \times D_j \rightarrow \mathbb{R}^+ \cup \{0\}$.[1] A *binary DCOP* is a DCOP in which all constraints are binary. Agents are *neighbors* if there is a constraint that they are both involved in. A *partial assignment* (PA) is a set of value assignments to variables, in which each variable appears at most once. *vars(PA)* is the set of all variables that appear in PA, $vars(PA) = \{X_i \mid \exists d \in D_i \wedge \langle X_i, d \rangle \in PA\}$. A constraint $R_j \in \mathcal{R}$ of the form $R_j : D_{j_1} \times D_{j_2} \times \ldots \times D_{j_k} \rightarrow \mathbb{R}^+ \cup \{0\}$ is *applicable* to PA if each of the variables $X_{j_1}, X_{j_2}, \ldots, X_{j_k}$ is included in $vars(PA)$. The *cost of a partial assignment* PA is the sum of all applicable constraints to PA over the value assignments in PA. A *complete assignment* (or a *solution*) is a partial assignment that includes all the DCOP's variables ($vars(PA) = \mathcal{X}$). An *optimal solution* is a complete assignment with minimal cost.

For simplicity, we make the common assumption that each agent holds exactly one variable (i.e., $n = m$) and we focus on binary DCOPs. These assumptions are common in DCOP literature (e.g., [15, 20]). We also assume that all agents are aware of the costs incurred by the constraints they are involved in (i.e., that the problems are symmetric).

3.2 Distributed Local Search for DCOPs

The general design of most state-of-the-art local search algorithms for DCOPs is synchronous. Since the environment in which they perform in is distributed and asynchronous (no mutual clock), the synchronization is achieved by the exchange of messages in each iteration of the algorithm. Thus, in each iteration, an agent receives the messages sent to it from its neighbors in the previous iteration, performs computation and sends messages to all its neighbors [22, 24]. All local search algorithms include in each step of the

[1] We say that a variable is *involved* in a constraint if it is one of the variables the constraint refers to.

algorithm an iteration in which agents share with their neighbors their value assignments. In some algorithms, e.g., MGM, a step of the algorithm includes more than one iteration [10, 14].

Two of the most well known algorithms that use this general framework are the *Distributed Stochastic Algorithm* (DSA) [22] and the *Maximum Gain Messages* (MGM) algorithm [10]. In both algorithms, after an initial step in which agents select a value assignment for their variable (randomly according to Zhang et al. [22]), agents perform a sequence of steps until some termination condition is met. In each step, an agent sends its value assignment to its neighbors in the constraint graph and receives the value assignments of its neighbors. The algorithms differ in the way that the agents decide on whether to replace their value assignments. In DSA, this decision is made stochastically and has a large effect on the performance of the algorithm. According to Zhang et al. [22], if an agent in DSA cannot improve its current state by replacing its current value assignment, it does not replace it. If it can improve (or keep the same cost, depending on the version used), it decides whether to replace the value assignment using a stochastic strategy (see the work by Zhang et al. [22] for details on the possible strategies and the differences in the resulting performance). In MGM, a second iteration is performed in which agents share with their neighbors the maximal possible gain they can achieve by replacing their value assignment. An agent replaces its assignment, only if its gain is larger than all its neighbors (ties are broken deterministically using agents' indexes). As a result, in MGM, neighboring agents cannot replace assignments concurrently and, thus, its improvement of the general cost is (weakly) monotonic when applied to symmetric DCOPs. This is in contrast to DSA where, when neighboring agents change assignments concurrently, the result may be an increase in the overall cost of the current solution. Moreover, when MGM converges, each agent has a chance to replace its assignment, but cannot find an alternative assignment that reduces its local cost (and with it the overall cost). Thus, it converges to a 1-opt solution (a solution that cannot be improved by the actions of a single agent) [10].

3.3 Anytime Distributed Local Search (ALS)

During the execution of a distributed local search algorithm, each agent is aware of the cost of its current assignment, but no agent is aware of the global cost of the current solution. Thus, in order to report the best solution that was explored by the algorithm, Zivan et al. [24] proposed a mechanism (or framework) that can be used with any local search algorithm, which guarantees that it will report the best solution found by the local search algorithm. The framework includes a *Breadth-First Search* (BFS) spanning tree of the constraint graph, which the agents use in order to aggregate the costs they incur in each iteration, such that a single agent (the root of the BFS tree) can decide which solution was best. Zivan et al. [24] proved that the overhead in time, memory, communication and privacy is relatively small.

In more detail, following every iteration k, a leaf agent in the spanning tree includes in the message it sends to its parent in the tree its local cost. In the following iteration $k+1$ the parent will sum the costs received from its children, add its own cost and send the resulting sum of costs to its parent. Thus, after a number of iterations equal to the height of the tree (h), the root agent will be able to compare the cost of the solution that agents held in iteration k, with the cost of the best solution found so far. If indeed the solution in iteration k was better (i.e., its cost was lower), it sends this information down the tree. Hence, following iteration $k + 2h$, all agents are aware that the current best solution is the one they held in iteration k. Each agent must use a memory of size at most $2h$, to store the relevant costs and assignments that will allow this process. In terms of runtime, in order to report the best among m iterations, the mechanism must run for $m + 4h$ iterations, including the generation of the BFS tree. The communication overhead is negligible, since all that is required is a constant addition of information to messages, which are sent by the algorithm on tree edges. The height h is expected to be small, since the mechanism uses a BFS tree (see analysis by Zivan et al. [24]).

4 LOCAL SEARCH IN THE PRESENCE OF MESSAGE LATENCY

The first relevant observation one must make when analyzing the effect of latency on distributed local search algorithms is the following: The general setting that agents are expected to perform in is *asynchronous* (no mutual clock between the agents). However, the algorithms are by design *synchronous* since each agent sends messages to all its neighbors and *waits* to receive messages from all of them in each iteration [10, 22, 24]. Thus, message latency has a major effect on the performance of such algorithms since every synchronous iteration is completed only after all the messages sent in the previous iteration arrive.

In order to overcome this limitation, agents can perform asynchronously (i.e., avoid waiting for all the messages from neighboring agents to arrive before performing the computation phase of the iteration). Instead, in the asynchronous versions of the algorithms, agents perform computation whenever they receive a message. Then, following each computation, they read the messages that were received during the computation and compute again. If an agent does not receive any message during computation, it returns to a waiting mode. In any computation phase, agents consider the information received in the message that was last received from each of their neighbors.[2]

However, when messages can be delayed, this approach is expected to have consequences on the quality of solutions the algorithm explores:

(1) An agent may take into consideration, when selecting alternative value assignments, obsolete assignments of its neighbors, since the information regarding their replacement was sent but not yet received.
(2) The last message received from a neighbor may not have been the one that was last sent among the messages that were received from this neighbor (i.e., a message sent at time t from an agent A_i to A_j can arrive before a message sent at time $t' > t$ from A_i to A_j).
(3) Algorithms such as MGM, which are weakly monotonic when performing synchronously, will not maintain this

[2]The use of the asynchronous version of DSA and MGM is not a novel contribution of this paper; however, the analysis of their performance in the presence of message latency is novel.

property. An agent may perform actions that indeed deteriorate the overall solution because it is not aware of the current assignment of its neighbors.

(4) The anytime mechanism proposed by Zivan et al. [24] is not applicable in such settings since it is dependent on the ability of agents to evaluate the cost of their state (their value assignments and the value assignments of their neighbors) in each iteration.

We address the first weakness in our empirical evaluation, where we present the quality of the solution as a function of the latency magnitude. The second weakness among the ones listed above can be handled if every agent A_i sending a message to its neighbor A_j, would add the number of previous messages it sent to A_j to the message (a timestamp). Thus, A_j could ignore messages, which arrive later than messages sent after them.

In order to overcome the third and fourth weaknesses, we present, in the rest of this section, an *Asynchronous Monotonic Distributed Local Search* (AMDLS) algorithm and an *Asynchronous Anytime Mechanism* (AAM) that agents can use in an asynchronous environment in which messages may be delayed and can be received not order in which they were sent.

4.1 Asynchronous Monotonic Distributed Local Search (AMDLS)

The basic idea behind the design of the AMDLS algorithm is that neighboring agents will not be allowed to concurrently perform calculations and decide whether to replace their value assignments. To this end, we use an ordered graph coloring structure, in which agents are divided into subsets. Agents that belong to the same subset hold the same color while agents from different subsets hold different colors. The set of subsets is ordered (i.e., there is a mapping from colors to the smallest natural numbers – from 1 to NC where NC is the number of colors). The neighbors of each agent must hold a different color than its own, and the agent must know which of them are ordered before it and which after. In order to generate this structure, the agents perform a *Distributed Ordered Color Selection* (DOCS) algorithm, a simple distributed algorithm, inspired by Barenboim and Elkin [1]:[3]

(1) Every agent that its index[4] is smaller than all its neighbors, selects the color 1 and sends it to its neighbors.

(2) An agent that received the colors of all its lower indexed neighbors, selects a color with the smallest number, which was not selected by one of its smaller index neighbors, and sends it to all its neighbors.

(3) An agent that received a color from its neighbor, stores it.

We demonstrate the performance of DOCS by considering the constraint graph depicted in Figure 1 (nodes represent agents and depicted beneath each node, is the color it selected). Agents A_1 and A_2 have no neighbors with smaller indexes, therefore they select the color 1 (blue) and send messages to their neighbors. Among these neighbors, agents A_3, A_4 and A_7 have no other neighbor with smaller index, thus, they select the color 2 (red). Finally, agent A_5 selects 1 and agent A_6 selects the color 3 (green).

[3]We do not present this simple algorithm in algorithm format because of its simplicity and since it is not a novel contribution.

[4]We assume that each agent has a unique index.

Algorithm 1 AMDLS

input: $N(i), PC(i), FC(i)$

1. $sc_i \leftarrow 1$;
2. $v^{N(i)} \leftarrow \{0, 0, \ldots, 0\}$
3. $value_i \leftarrow select\_value$;
4. send($value_i$, sc_i) to $N(i)$;
5. **while** stop condition not met **do**
6. **when** received ($value_j$,sc_j) from A_j
7. update $local\_view$ and $v^{N(i)}$
8. **if** consistent($v^{N(i)}, sc_i, PC(i), FC(i)$)
9. $value_i \leftarrow select\_value$;
10. $sc_i \leftarrow sc_i + 1$;
11. send($value$,sc_i) to $N(i)$;

The properties of this algorithm are as follows: Let Δ denote the maximal number of neighbors an agent can have, δ denote the maximal time a message is delayed, and t denote the maximal iteration time, which is $o(\Delta)$.

Lemma 1. *When DOCS converges, each agent has selected a color and stored the colors of all its neighbors.*

Lemma 2. *DOCS will converge after at most n iterations. Thus, the time for convergence will be at most* $nt + \delta(n-1)$.

It is important to mention that in practice, as we demonstrate empirically, the convergence is much faster.

Lemma 3. *The maximal number of colors selected after the convergence of DOCS is* $\Delta + 1$.

The ordered coloring division achieved by this DOCS is used as follows by AMDLS: Each agent holds a designated data structure in which it counts the number of computation steps performed by its neighbors. After each computation step, in which the agent considers to replace its value assignment, it informs its neighbors, which update their local data structure. An agent performs the k-th iteration, when the number of iterations performed by each of its neighbors with a smaller color index than its own, is equal to k and all its neighbors with larger indexes than its own have performed $k-1$ iterations.

In more detail, each agent A_i holds a vector of natural numbers $v^{N(i)}$, one number for each of its neighbors. In addition, it holds a step counter sc_i for counting the steps of computation it performs. At the beginning, all numbers in $v^{N(i)}$ and sc_i are initialized to zero. After each computation step in which the agent considers to replace its value assignment, it increments sc_i by one. Further, the agent includes its counter Sc_i in each message that it sends. When an agent receives a message from a neighbor A_j, it updates the relevant entry in $v^{N(i)}$.

Algorithm 1 presents the pseudo-code of AMDLS, which it executes after each agent selected its numerated color. In AMDLS, each agent A_i holds its set of neighbors $N(i)$, divided into two disjoint sets, one holding its the colors with smaller indexes than its own $PC(i)$ and one holding the colors with larger indexes $FC(i)$, such that $PC(i) \cup FC(i) = N(i)$ and $PC(i) \cap FC(i) = \emptyset$. After initiating the local counter sc_i and the vector of numbers $v^{N(i)}$ (lines 1 – 2), the agent selects a value for its variable and sends it along with sc_i to all its neighbors (lines 3 – 4). Then, as long as the algorithm has not terminated, the agent reacts to messages it receives.

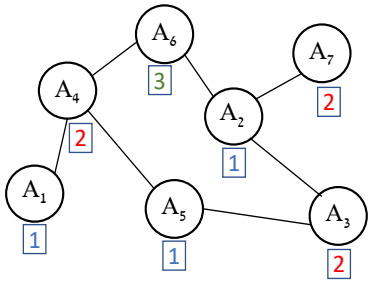

Figure 1: A numerical graph color partition.

Each message from a neighbor A_j, includes a value assignment and the neighbor's counter SC_j (line 5). After storing both, if sc_i and $v^{N(i)}$ are consistent, the agent increments sc_i by one, selects its value assignment, and sends a message including both to all of its neighbors (lines 6 – 11).

The value assignment selected is always the one minimizing the local constraint costs. In addition, sc_i is consistent with $v^{N(i)}$ if and only if, for each agent $A_j \in N(i)$, if $A_j \in PC(i)$, then $sc_j = sc_i + 1$ and, if $A_j \in FC(i)$, then $sc_j = sc_i$. Notice that while $PC(i)$ and $FC(i)$ are not used in the pseudo-code, they are used for the consistency check. This is also true for the counters sc_i, exchanged by the agents. Intuitively, this consistency check ensures that (1) neighboring agents do not replace assignments concurrently and (2) following each computation step of an agent, all its neighbors will have an opportunity to perform a computation step before it performs its next computation step. Formally, we prove the following two propositions:

Proposition 1. *AMDLS is weakly monotonic (i.e., each assignment replacement improves the global cost of the complete assignment held by the agents).*

Proof: For each pair of neighboring agents, the order on which they can replace their assignments is well defined. Thus, neighboring agents cannot replace assignments concurrently and, since the problem is symmetric, the overall sum of constraints must not increase when the local cost following an assignment replacement does not increase. □

Proposition 2. *AMDLS converges to a 1-opt solution.*

Proof: Upon convergence, each agent will get a chance to consider a replacement for its assignment. Thus, convergence implies that even after all messages arrive, no agent can improve its local state by replacing its value assignment. □

We now describe the start of a high-level trace for AMDLS operating on the constraint graph presented in Figure 1. After the agents have selected their colors, the algorithm is initialized. At this time, the counters of all agents are equal to zero and this is also true for the content of their vector holding the counters of their neighbors. For example, A_1 has one neighbor, hence $v^{[N(1)]} = \{\langle A_4{:}0\rangle\}$. On the other hand, agent A_4 has three neighbors and $v^{[N(4)]} = \{\langle A_1{:}0\rangle, \langle A_5{:}0\rangle, \langle A_6{:}0\rangle\}$. Moreover, $PC(1) = \emptyset$ and $FC(1) = \{A_4\}$. Similarly, $PC(4) = \{A_1, A_5\}$ and $FC(4) = \{A_6\}$.

After exchanging random assignments (that can be included in the colors selection messages) the agents wait for their state to indicate that it is their turn to perform steps of computation. Agents A_1, A_2, and A_5 can perform their first computation step concurrently, since their PC set is empty and the counters of the neighbors in their FC sets are equal to their own. Agent A_7 can perform its step of computation when it receives the message from A_1. At that time, its own counter will be zero, while the counter it holds for A_1 is equal to 1. Agent A_5 on the other hand must wait for a message from both agents A_3 and A_4 before it can perform its second computation step. At that time, $v^{[N(5)]} = \{\langle A_3{:}1\rangle, \langle A_4{:}1\rangle\}$. In order to perform the second computation step, A_2 must wait for messages indicating that agents A_3, A_6, and A_7 performed their first computation step. Agent A_1, on the other hand, has to wait only for the message from A_4 in order to perform its second computation step.

4.2 Asynchronous Anytime Mechanism (AAM)

The anytime framework for distributed local search algorithms (ALS) [24] enables the agents to report the best solution traversed by any synchronous algorithm (i.e., an algorithm where, at each iteration, agents receive messages sent to them by all their neighbors in the previous iteration, and send messages to all their neighbors, which will be received in the next iteration). The framework proposed requires a very low overhead in terms of runtime, memory, communication, and privacy. However, its dependency on the synchronous structure is inherent and, therefore, it is not applicable for asynchronous algorithms performing in an asynchronous environment with message latency. In more detail, each anytime message in ALS carries the iteration number, used to identify the costs related to the same state of the algorithm, to generate solution costs and to notify agents of the best solution found. These are not available when performing asynchronous algorithms.

The standard motivation for enhancing local search algorithms with an anytime framework is the ability to perform exploration heuristics that will improve the anytime global solution without the risk of reporting a low-quality solution as might happen if such explorative heuristics are used when the solution held by agents at the end of the run is reported. However, in the presence of message latency, as we demonstrate in our experimental study (see Section 5), the latency itself generates exploration that improves the global anytime solution. Thus, there is extra motivation to capture and report the best anytime solution.

We propose an *Asynchronous Anytime Mechanism* (AAM), which uses a spanning tree as used in the synchronous anytime framework proposed by Zivan et al. [24]. Algorithm 2 presents the pseudo-code for the actions of agents within AAM. It is important to notice that, unlike the pseudo-code presented in Algorithm 1, which describes all actions of agents within an asynchronous distributed algorithm, here we are only describing the actions related to the anytime mechanism, which are interleaved with any asynchronous local search algorithm. Each agent maintains a data structure, which we call $context_i$ in which it stores its own value assignment, the value assignments received from its neighbors, and the value assignments of all the agents in the sub-tree that it is the root of. For a leaf agent, the context includes only its own assignment and the assignment of its neighbors. A change in $context_i$ can be

Algorithm 2 Asynchronous Anytime Mechanism (AAM)

input: $P(i), C(i)$.

1. $is\_leaf_i \leftarrow C(i)$ is empty;
2. $CS(i) \leftarrow \emptyset$;
3. $best\_solution \leftarrow \emptyset$;
6. $best\_cost \leftarrow \infty$;
7. **when** is updated $(cotnext_i)$
8. **if** is_leaf_i & cost($new\_context$) < $best\_cost$
9. send ($context_i$, cost($context_i$)) to $P(i)$
10. **else**
11. $new\_context$ = get_context($CS(i)$,$context_i$);
12. **if** $new\_context$ is not empty & cost($new\_context$) < $best\_cost$
13. send ($new\_context$, cost($new\_context$)) to $P(i)$
14. **when** receive $context\_j$ from $A_j \in C_i$
15. add $context_j$ to $CS(i)$;
16. $new\_context$ = get_context($CS(i)$,$context_j$);
17. **if** $new\_context$ is not empty
18. **if** $P(i)$ is not empty
19. send ($new\_context$, cost($new\_context$)) to $P(i)$
20. **else**
21. if cost($new\_context$) < $best\_cost$
22. $best\_solution \leftarrow new\_context$
23. $best\_cost \leftarrow$ cost($new\_context$)
24. send ($best\_solution$,$best\_cost$) to $C(i)$;
25. **when** receive $best\_solution_{P(i)}$,$best\_cost_{P(i)}$ from $P(i)$
26. $best\_solution \leftarrow best\_solution_{P(i)}$;
27. $best\_cost \leftarrow best\_cost_{P(i)}$;
28. remove contexts with cost $\geq best\_cost$;
29. send ($best\_solution$,$best\_cost$) to $C(i)$;

generated either by actions of the algorithm or by the reception of a context message from one of the children in the tree, and the anytime mechanism reacts to such changes.

A leaf in the tree sends to its parent $context_i$ and its local cost for this context, c_i with each change of $context_i$ under the condition that the cost of the context generated does not exceed the best cost found for a solution so far (line 7 – 9 of Algorithm 2). When a non-leaf agent receives a message from a child in the tree, it stores the context included in the message in its context storage $CS(i)$ (line 15). Following any such message and after each context change, the non-leaf agent seeks to generate a consistent context with all contexts it received from all members of $C(i)$ (its children in the tree). If it is able to generate a consistent context, including assignments to all variables in the contexts sent to it by its children, if its cost is lower than the best cost found for a solution, it sends it along with the corresponding cost to its parent (lines 11 – 13 and 16 – 19). When the root agent generates a consistent context of all the variables in the problem, it checks whether its cost is the lowest found so far. If so, it stores it as the current best solution and sends it down the tree along with its cost (lines 21 – 24). Non-root agents, which receive a best solution message including the best cost, store them, filter out contexts with higher cost than the best cost, and pass the message on to their children in the tree (lines 25 and 29).

Proposition 3. *The solution reported by AAM will be a solution with the minimal cost that can be composed of all contexts that were sent to parents in the tree during the algorithm execution.*

Proof: Since all contexts received by agents from their children in the spanning tree are stored and all the consistent combinations (which do not exceed the cost of the best solution found) and their costs are sent up the tree, the root agent will consider all possible solutions that have a chance to have a lower cost than the current solution it is holding. Thus, a solution with the minimal cost must be considered as well. □

Proposition 4. *There can be a solution held by agents at some time during the algorithm that is not considered by AAM.*

Proof: Consider a problem including two neighboring agents, A_1 and A_2, each holding a single variable with two values in its domain a and b. In the beginning, both agents assign a to their respective variables. At some iteration during the algorithm, A_1 replaces its assignment to b and sends a message to A_2. Before A_2 receives the message from A_1, indicating that it replaced its assignment, it also replaces its assignment to b. Thus, one of the contexts that A_1 held included an assignment $\langle A_1, b\rangle$, $\langle A_2, a\rangle$, which indeed was the state after A_1 replaced its assignment and before A_2 replaced his. Nevertheless, A_2 did not hold such a context, and therefore it will not report a cost for it to its parent. □

We demonstrate in our empirical results that, indeed, a global view anytime mechanism reports better solutions than AAM. However, AAM's results are better than the results of the assignments agents hold in the final iteration of the algorithm.

Let γ_i denote the maximum between Δ_i and the size of each context received from $C(i)$ and θ_i denote the largest number of stored contexts received from a child in C_i or produced and stored by A_i.

Proposition 5. *The maximal time (number of NCLOs) for an agent A_i to check if a new context can be generated is $(\theta_i\gamma_i)^{|C(i)|+1}$.*

Proof: This is the exhaustive result of checking the compatibility of each context received from a child with each other and the contexts produced and stored by the agent. □

A corollary from Proposition 5 is that we prefer a tree with a small branching factor, in contrast to ALS, where a BFS tree with the smallest height is preferred.

5 EXPERIMENTAL EVALUATION

In order to evaluate the success of the proposed adjustments of local search distributed algorithms and the asynchronous anytime mechanism for environments including message latency, we used an asynchronous simulator, in which agents are implemented by Java threads. It included a *mailing agent*, which simulated the delays of messages, as suggested by Zivan and Meisels [23]. For each message, a delay in terms of the number of *Non-Concurrent Logic Operations* (NCLO) was selected (the atomic logic operations in these algorithms is the evaluation of the cost of a combination of two assignments, i.e., a constraint check), and the message was delivered to the agent it was sent to, after that agent had the opportunity to perform this number of logic operations.

We evaluated the algorithms on problems including 50 agents. These included random uniform minimization DCOPs with density $p_1 = \{0.2, 0.7\}$ and on structured problems (i.e., graph coloring problems and scale-free networks). In each experiment, we

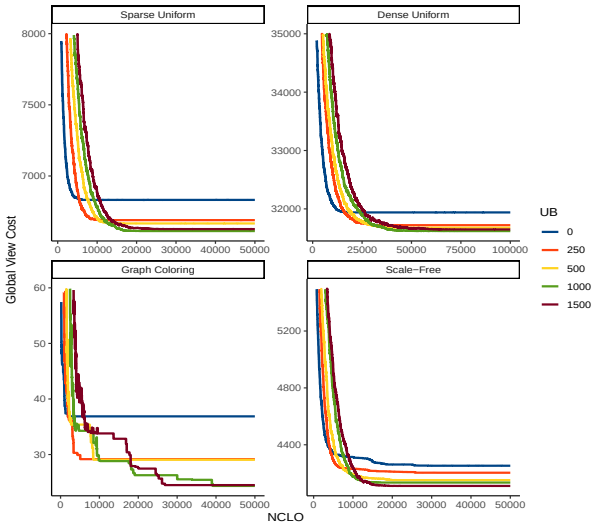

Figure 2: Costs of solutions of asynchronous DSA with different lengths of message delays.

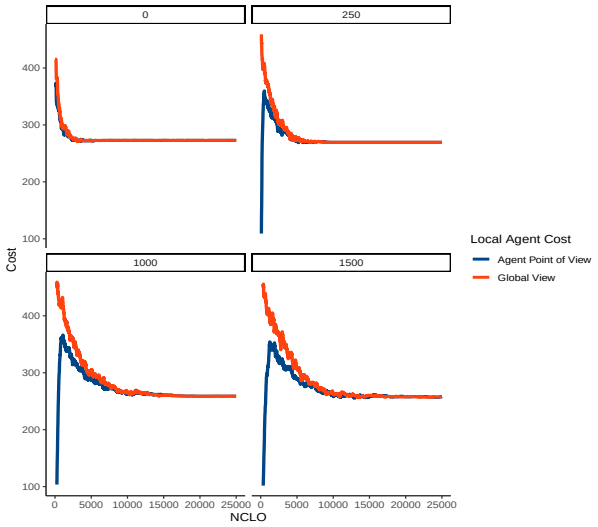

Figure 3: Local costs of a single agent, solving sparse problems, from a local and global view, with message latency.

randomly generated 100 different problem instances. The results presented in the graphs are an average of those 100 runs. In order to demonstrate the convergence of the algorithms, we present the sum of costs of the constraints involved in the assignment that would have been selected by each algorithm every 10 NCLOs. We performed *t-tests* to evaluate the significance of differences between all presented results.

Figure 2 includes the results of an asynchronous version of DSA[5] in which an agent initiates a computation step whenever it receives a message. Each message sent was delayed for a number of NCLOs that is selected uniformly between 0 and UB. The results presented are the average global cost after each agent completed the number of NCLOs specified on the x-axis. The version with $UB = 0$ is actually standard DSA since messages arrive instantaneously. It is clear from the results that, in most cases, message delays have a positive exploration effect on the performance of DSA. As expected, exploration and exploitation need to be balanced and, in some cases, long delays ($UB = 1500$) cause a deterioration in the global cost.

In order to explain the relationship between message latency and exploration, we present in Figure 3 the local cost of a single agent during the run of the asynchronous DSA algorithm. We depict both the agents view, which takes into consideration the assignments included in the messages it received, and a global view, considering the actual assignments of the neighboring agents at that time. It is apparent that the larger the latency, the larger the difference between the two different views and, thus, the agents perform actions that exploit obsolete information, which results in an increase of the cost (i.e., exploration). This temporary increase in utility is apparent for $UB > 0$. Notice that in the beginning of the run (on the left), before the agents receive the assignments of their neighbors for the first time, they think that they are not violating any constraints. Their views change dramatically with every message received from a neighbor.

[5]DSA-B with $p = 0.7$.

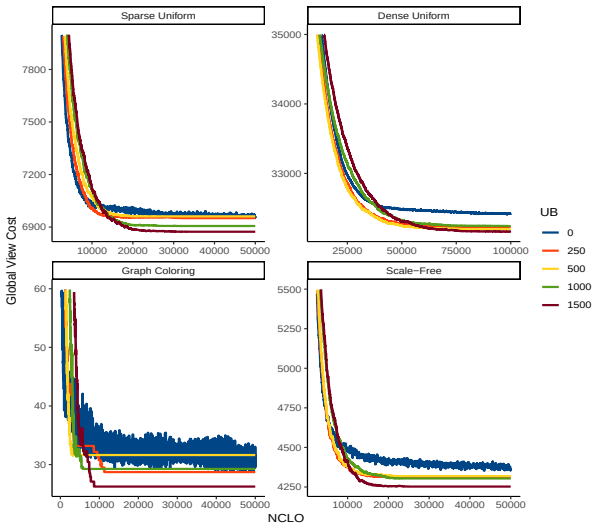

Figure 4: Costs of solutions of asynchronous MGM with different lengths of message delays.

Similar results, to the results presented in Figure 2 for asynchronous DSA, are presented in Figure 4 for the asynchronous MGM algorithm. The results reveal that when messages are delayed, the algorithm does not maintain its monotonic property. This is because agents can perform calculation and select value assignments while considering assignments of their neighbors that were replaced. Nevertheless, losing monotonicity has a positive explorative effect (i.e., the algorithms converge to solutions with higher quality, that is, to solutions with lower cost). Once again, the length of delays that cause the best explorative effect is problem dependent.

Figure 5 presents the results of the proposed AMDLS algorithm compared to synchronous MGM, the only two monotonic algorithms presented, solving the four types of problems with different message latency. The lines in AMDLS start later since they are

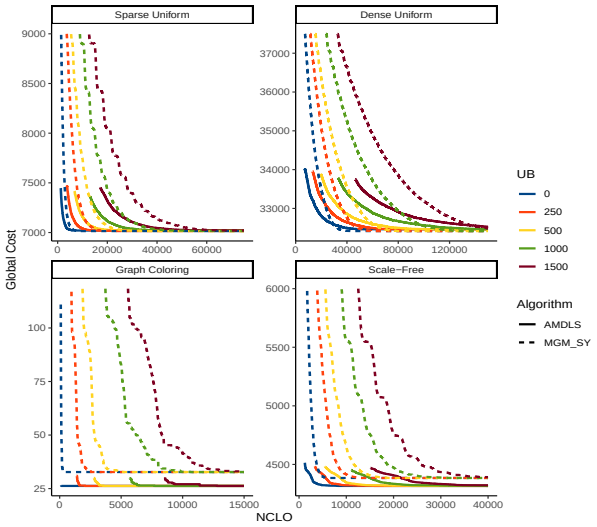

Figure 5: Costs of solutions of AMDLS and MGM with different lengths of message delays.

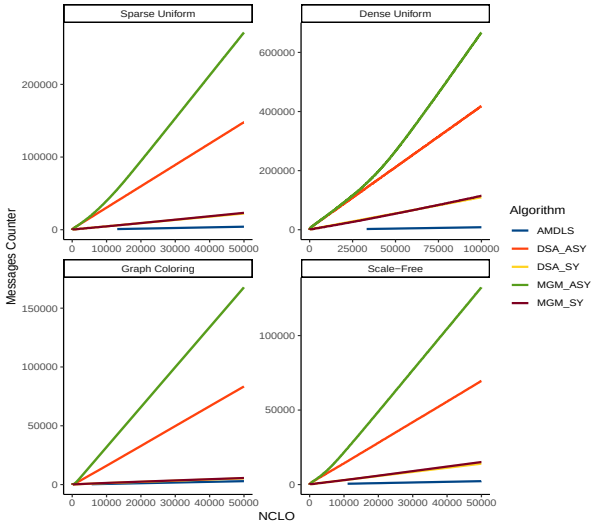

Figure 6: Number of messages sent by the algorithms, performing in environments with $UB = 1000$.

depicted only after the colors are selected by the agents and they all select the first assignment. As expected, regardless of the message delay, each algorithm converges to the same result in terms of solution quality for each problem. The differences in solution quality between the two algorithms were not significant, except in scale-free networks. However, the significant difference between the algorithms was in the time for convergence. In environments with message latency, AMDLS converged faster and the differences grew relatively to the length of the delays.

Figure 6 presents the number of messages sent by the different algorithms in scenarios with $UB = 1000$. It is clear that in terms of communication, the asynchronous versions of DSA and MGM are most wasteful and, thus, their advantage in solution quality comes with a cost in communication. On the other hand, AMDLS is most

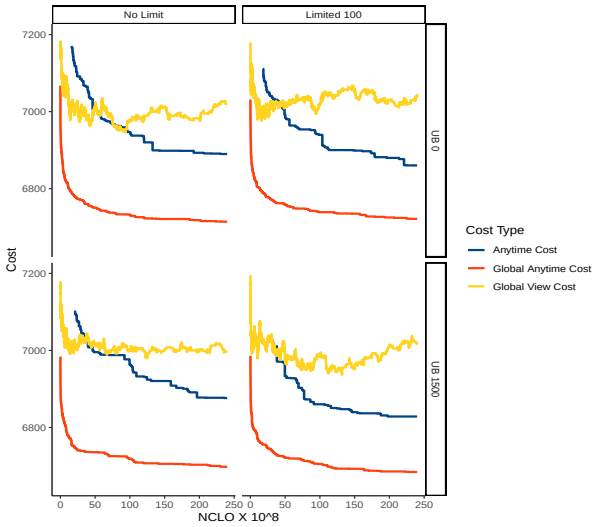

Figure 7: Comparison between DSA_SDP when combined with AAM solving sparse random problems ($p_1 = 0.2$).

economic in its use of the communication network than the other algorithms.

Figure 7 presents a comparison between the results reported for an asynchronous version of DSA_SDP, which is an explorative version of DSA that was reported to be successful when combined with the synchronous anytime mechanism (ALS) [24]. In each graph, we present the cost per iteration from a global view, the anytime cost from a global view, and the cost of the solutions reported by AAM. In the two upper graphs, there was no message latency, while in the two bottom graphs, $UB = 1500$. In the experiments where their results are depicted on the two left graphs, agents performing AAM had no memory limit. In the experiments that their results are depicted in the right graphs, each agent was limited in memory to a size that is sufficient to store only 100 contexts. The agents discarded the contexts that were least similar to their current context (last generated).

As expected, following Proposition 5 and its corollary, the results reported by AAM are with higher costs than the global view anytime results, but they improve on the results per iteration. Surprisingly, the results there was not much different in the quality of the results between the unlimited- and limited-memory versions.

6 CONCLUSIONS

We investigated the implications of message latency on the performance of distributed local search algorithms for solving DCOPs. Our analysis identified major limitations to the empirical and theoretical properties of local search DCOP algorithms in the presence of message latency. We addressed those limitations through approaches that are robust to latency. Therefore, these findings increase the applicability of DCOP algorithms in the real world where latency are always present.

In future work, we intend to investigate the effect of latency on incomplete inference algorithms such as Max-sum [5].

REFERENCES

[1] L. Barenboim and M. Elkin. 2014. Combinatorial algorithms for distributed graph coloring. *Distributed Computing* 27, 2 (2014), 79–93.

[2] M. Basharu, I. Arana, and H. Ahriz. 2005. Solving DisCSPs with penalty driven search. In *Proceedings of AAAI*. 47–52.

[3] A. Chechetka and K. Sycara. 2006. No-Commitment Branch and Bound Search for Distributed Constraint Optimization. In *Proceedings of AAMAS*. 1427–1429.

[4] F. Cruz, P. Gutierrez, and P. Meseguer. 2014. Simulation vs Real Execution in DCOP Solving. In *Proceedings of the Distributed Constraint Reasoning Workshop*.

[5] A. Farinelli, A. Rogers, A. Petcu, and N. Jennings. 2008. Decentralised Coordination of Low-Power Embedded Devices Using the Max-Sum Algorithm. In *Proceedings of AAMAS*. 639–646.

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

[7] F. Fioretto, E. Pontelli, and W. Yeoh. 2018. Distributed Constraint Optimization Problems and Applications: A Survey. *Journal of Artificial Intelligence Research* 61 (2018), 623–698.

[8] A. Gershman, A. Meisels, and R. Zivan. 2009. Asynchronous Forward Bounding. *Journal of Artificial Intelligence Research* 34 (2009), 25–46.

[9] K. D. Hoang, F. Fioretto, W. Yeoh, E. Pontelli, and R. Zivan. 2018. A Large Neighboring Search Schema for Multi-agent Optimization. In *Proceedings of CP*. 688–706.

[10] R. Maheswaran, J. Pearce, and M. Tambe. 2004. Distributed Algorithms for DCOP: A Graphical Game-Based Approach. In *Proceedings of PDCS*. 432–439.

[11] S. McCanne and S. Floyd. 2011. ns–Network Simulator. http://nsnam.sourceforge.net/wiki/.

[12] P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. 2005. ADOPT: Asynchronous Distributed Constraint Optimization with Quality Guarantees. *Artificial Intelligence* 161, 1–2 (2005), 149–180.

[13] D. T. Nguyen, W. Yeoh, H. C. Lau, and R. Zivan. 2019. Distributed Gibbs: A Linear-Space Sampling-Based DCOP Algorithm. *Journal of Artificial Intelligence Research* 64 (2019), 705–748.

[14] S. Okamoto, R. Zivan, and A. Nahon. 2016. Distributed Breakout: Beyond Satisfaction. In *Proceedings of IJCAI*. 447–453.

[15] A. Petcu and B. Faltings. 2005. A Scalable Method for Multiagent Constraint Optimization. In *Proceedings IJCAI*. 1413–1420.

[16] M. Smith and R. Mailler. 2010. Getting What You Pay For: Is Exploration in Distributed Hill Climbing Really Worth It?. In *Proceedings of IAT*. 319–326.

[17] A. M. Tabakhi, R. Tourani, F. Natividad, W. Yeoh, and S. Misra. 2017. Pseudo-tree Construction Heuristics for DCOPs and Evaluations on the ns-2 Network Simulator. In *Proceedings of ICTAI*. 1105–1112.

[18] A. M. Tabakhi, W. Yeoh, R. Tourani, F. Natividad, and S. Misra. 2018. Communication-Sensitive Pseudo-Tree Heuristics for DCOP Algorithms. *International Journal on Artificial Intelligence Tools* 7, 27 (2018), 1860008:1–1860008:24.

[19] M. Wahbi and K. N. Brown. 2014. The impact of wireless communication on distributed constraint satisfaction. In *Proceedings of CP*. 738–754.

[20] W. Yeoh, A. Felner, and S. Koenig. 2010. BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. *Journal of Artificial Intelligence Research* 38 (2010), 85–133.

[21] M. Yokoo and K. Hirayama. 1996. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of AAMAS*.

[22] W. Zhang, G. Wang, Z. Xing, and L. Wittenberg. 2005. Distributed Stochastic Search and Distributed Breakout: Properties, Comparison and Applications to Constraint Optimization Problems in Sensor Networks. *Artificial Intelligence* 161, 1–2 (2005), 55–87.

[23] R. Zivan and A. Meisels. 2006. Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence* 46 (2006), 415–439.

[24] R. Zivan, S. Okamoto, and H. Peled. 2014. Explorative anytime local search for distributed constraint optimization. *Artificial Intelligence* 211 (2014).