Journal of Artificial Intelligence Research 75 (2022) 637-675 Submitted 04/2022; published 10/2022

Communication-Aware Local Search for Distributed Constraint Optimization

Ben Rachmut RACHMUT@POST.BGU.AC.IL
Roie Zivan ZIVANR@BGU.AC.IL
Department of Industrial Engineering and Management
Ben-Gurion University of the Negev
Beer Sheva, Israel

William Yeoh WYEOH@WUSTL.EDU
Department of Computer Science and Engineering
Washington University in St. Louis
Saint Louis, United States

Abstract

Most studies investigating models and algorithms for distributed constraint optimization problems (DCOPs) assume that messages arrive instantaneously and are never lost. Specifically, distributed local search DCOP algorithms, have been designed as synchronous algorithms (i.e., they perform in synchronous iterations in which each agent exchanges messages with all its neighbors), despite running in asynchronous environments. 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 assumption of perfect communication is relaxed, the properties that were established for the state-of-the-art local search algorithms and the anytime mechanism may not necessarily apply.

In this work, we address this limitation by: (1) Proposing a Communication-Aware DCOP model (CA-DCOP) that can represent scenarios with different communication disturbances; (2) Investigating the performance of existing local search DCOP algorithms, specifically Distributed Stochastic Algorithm (DSA) and Maximum Gain Messages (MGM), in the presence of message latency and message loss; (3) Proposing a latency-aware monotonic distributed local search DCOP algorithm; and (4) Proposing an asynchronous anytime framework for reporting the best solution explored by non-monotonic asynchronous local search DCOP algorithms. Our empirical results demonstrate that imperfect communication has a positive effect on distributed local search algorithms due to increased exploration. Furthermore, the asynchronous anytime framework we proposed allows one to benefit from algorithms with inherent explorative heuristics.

1. Introduction

Recent advances in computation and communication have resulted in realistic distributed applications (e.g., IoT applications), in which humans and technology interact and aim to optimize mutual goals. A well known example for such an application is of a smart home, where smart devices must coordinate with each other in order to arrive at a good schedule that satisfies constraints and is guided by users' preferences (Fioretto, Yeoh, & Pontelli, 2017; Rust, Picard, & Ramparany, 2022). A promising multi-agent approach to solve these types of problems is to model them as *Distributed Constraint Optimization Problems* (DCOPs) (Modi, Shen, Tambe, & Yokoo, 2005; Petcu & Faltings, 2005; Fioretto, Pontelli,

& Yeoh, 2018), 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 (Yokoo, Ishida, Durfee, & Kuwabara, 1992) are rather simplistic and may need to be updated to reflect today's communication technologies: (1) Messages are never lost; (2) All messages have very small and bounded delays; and (3) All messages arrive in the order that they were sent. These assumptions do not reflect real-world characteristics, where messages may be disproportionately delayed, or dropped, due to congestion and bandwidth limitations.

Because DCOPs are NP-hard (Modi et al., 2005), considerable research effort has been devoted to developing incomplete algorithms for finding good solutions quickly (Yokoo & Hirayama, 1996; Maheswaran, Pearce, & Tambe, 2004; Zhang, Wang, Xing, & Wittenberg, 2005; Basharu, Arana, & Ahriz, 2005; Farinelli, Rogers, Petcu, & Jennings, 2008; Smith & Mailler, 2010; Hoang, Fioretto, Yeoh, Pontelli, & Zivan, 2018; Nguyen, Yeoh, Lau, & Zivan, 2019). Distributed local search algorithms (e.g., *Distributed Stochastic Algorithm* (DSA) (Zhang et al., 2005), and *Maximum Gain Messages* (MGM) (Maheswaran et al., 2004)) are simple incomplete algorithms with a naturally distributed structure.

The general design of most state-of-the-art local search algorithms for DCOPs (including DSA and MGM) is synchronous. However, the general setting in which agents are expected to perform is *asynchronous* (Lynch, 1997), since the environment in which they perform in is distributed and the agents do not hold a mutual clock. Therefore, the synchronization is achieved by the exchange of messages in each iteration of the algorithm. 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 (Zhang et al., 2005; Zivan, Okamoto, & Peled, 2014).

Although distributed local search algorithms commonly offer no (or little) quality guarantees, they were empirically found to produce high quality solutions (Yokoo & Hirayama, 1996; Maheswaran et al., 2004; Zhang et al., 2005). Some of these algorithms do guarantee other desirable properties, for example, MGM guarantees monotonicity and convergence to a 1-opt solution (i.e, a solution that cannot be improved by an action of a single agent (Maheswaran et al., 2004; Pearce & Tambe, 2007)). Another property that can be guaranteed by distributed local search algorithms is the *anytime* property (see Section 2.3). This property can be achieved for any synchronous distributed local search algorithm by using the anytime framework proposed by Zivan et al. (2014). Unfortunately, such distributed local search algorithms and the anytime framework take advantage of the common simplistic communication assumptions discussed above. As a result, the guarantees for achieving these properties (i.e., monotonicity and anytime properties) may no longer hold when communication is unreliable.

In fact, imperfect communication has a major effect on the performance of synchronous distributed local search algorithms. In the presence of message latency, every synchronous iteration is completed only after all the messages sent in the previous iteration arrive and, therefore, the advancement of the algorithm from one iteration to the next is dependent on the longest message delay in that iteration. When messages can be lost, an agent may expect to receive a message from its neighbor while the neighbor is not aware that the message it sent did not arrive. Thus, these agents are deadlocked, each waiting for the message from the other.

In this paper, we make the following contributions:[1]

1. We propose *Communication-Aware DCOP* (CA-DCOP),[2] an extension of the DCOP model in which patterns of communication disturbances (e.g., message latency and message loss) can be represented.

2. We demonstrate that existing distributed local search DCOP algorithms are not robust to imperfect communication. Thus, we analyze the performance and properties of standard local search algorithms after they are adjusted to perform asynchronously in scenarios that include message latency and message loss.

3. We propose a *latency-aware monotonic distributed local search* (LAMDLS) algorithm that is guaranteed to converge to a 1-opt solution (similar to the properties of the MGM algorithm (Maheswaran et al., 2004)).

4. We propose an asynchronous anytime mechanism that allows any local search algorithm running in an environment with imperfect communication to report the best solution it was able to generate during its run.

5. We show that the presence of imperfect communication can have a positive impact on exploitative asynchronous local search algorithms. Our empirical results reveal that solution quality may improve as the quality of communication degrades (both in terms of message latency and message loss).

Imperfect communication generates a discrepancy between the knowledge that agents hold to the actual state of the system. Thus, an agent may perform an action that it considers to be exploitative with respect to the information it holds, expecting to improve its own state and the global state as well. However, in reality, it's action degrades its own state and possibly the global state, and it unknowingly explored an unexpected part of the search space. Such explorative actions often exposes the agents to higher quality solutions, allowing them to converge to those better solutions.

2. Background

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

2.1 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 (Fioretto et al., 2018). Thus, we assume that all constraints define costs and not utilities. Our description

1. This work is an extension of our published AAMAS 2021 paper (Rachmut, Zivan, & Yeoh, 2021).
2. When referring to communication awareness, we do not mean that agents are aware of the communication pattern, but rather that the algorithms were designed such that they can overcome communication limitations.

of a DCOP is also consistent with the definitions in many DCOP studies (Modi et al., 2005; Petcu & Faltings, 2005; Gershman, Meisels, & Zivan, 2009).

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 constraints (relations), 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 $R_{ij} : D_i \times D_j \rightarrow \mathbb{R}^+ \cup \{0\}$.

A *binary DCOP* is a DCOP in which all constraints are binary. Agents are *neighbors* if they are involved in the same constraint. 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 partial assignment PA (i.e., $vars(PA) = \{X_i \mid \exists d \in D_i \wedge \langle X_i, d \rangle \in PA\}$). 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 variables ($vars(PA) = \mathcal{X}$). An *optimal solution* is a complete assignment with minimal cost.

For simplicity, we assume 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., (Petcu & Faltings, 2005; Yeoh, Felner, & Koenig, 2010)). We also assume that for every binary constraint, both agents involved in the constraint are aware of the content of the domains of both variables held by them, and that they incur the same cost from each combination of assignments to these two variables (i.e., that the problems are symmetric).

2.2 Distributed Local Search for DCOPs

Well-known algorithms that use this general framework are the *Distributed Stochastic Algorithm* (DSA) (Zhang et al., 2005), *DSA-Slope Dependent Probability* (DSA_SDP) (Zivan et al., 2014), and the *Maximum Gain Messages* (MGM) algorithm (Maheswaran et al., 2004). In the three algorithms, after an initial step in which agents select a value assignment for their variable (randomly according to Zhang et al. (2005)), 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. (2005), 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 its value using a stochastic strategy (see the work by Zhang et al. (2005) for details on the possible strategies and the differences in the resulting performance).

DSA_SDP is a variant of DSA, where agents use a more exploratory strategy. An agent may possibly change assignments, even if a non-deteriorating alternative assignment does not exist, to a random value after a given amount of iterations. The explorative strategy is monitored by correlating an agent's probability of assignment change with the potential for improvement caused by the replacement.

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' indices). 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) (Maheswaran et al., 2004).

2.3 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. (2014) 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. (2014) 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 h of the tree, 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 is better (i.e., its cost is smaller), 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. (2014)).

3. Related Work

Most of the studies that investigated the performance of distributed search algorithms in the presence of imperfect communication considered algorithms for solving distributed constraint satisfaction problems (DisCSPs) (Zivan & Meisels, 2006; Samadidana & Mailler, 2017). Zivan and Meisels (2006) proposed a distributed asynchronous simulator that allows to evaluate the performance of asynchronous complete search algorithms for solving DisCSPs. The simulator included a designated mailer agent that all messages that were sent from one agent to another in the system were sent through it. The mailer agent could simulate any desired communication pattern. We use the same approach in our experimental evaluation. The evaluation performed by Zivan and Meisels (2006) revealed a clear hierarchy in the favor of concurrent search algorithms (e.g., (Zivan & Meisels, 2006)) in comparison with synchronous and asynchronous backtracking algorithms (Yokoo & Hirayama, 2000).

In the work by Samadidana and Mailler (2017) and Samadidana (2021), the performance of distributed local search algorithms are evaluated when solving problems in scenarios including message latency and message loss. While the motivation for these studies is similar to ours in this paper, which is to evaluate the performance of distributed local search algorithms in the presence of imperfect communication, there are two major differences in the experimental setup. First, the experiments in the prior work are on DisCSPs (instead of DCOPs in this paper) with low constraint tightness (i.e., for every binary constraint, most of the assignments are allowed; in DCOP terms, they incur no cost). Second, the simulator in the prior work is synchronous and the agents performed in cycles (i.e., in each cycle of the algorithm, each of the agents in its turn performed computation and sent out messages, even if it did not receive any messages). In contrast, our evaluations are on an asynchronous simulator, in which agents act only when they receive a message.

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 message latency or loss in the context of DCOPs. Researchers have investigated the importance of message latency in evaluations of DCOP algorithms (Cruz, Gutierrez, & Meseguer, 2014; Tabakhi, Tourani, Natividad, Yeoh, & Misra, 2017; Tabakhi, Yeoh, Tourani, Natividad, & Misra, 2018). For example, Cruz et al. (2014) conducted experiments where agents are located physically apart in different machines connected by LAN. 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. (2017, 2018). In their work, they used realistic network simulators, such as ns-2 (McCanne & Floyd, 2011), 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 latency and losses while evaluations of Cruz et al. (2014) 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. (2014) and Tabakhi et al. (2017, 2018) 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 latency and message loss as well. For example, Zivan and Meisels (2006) 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.

The impact of communication delays on DisCSP algorithms was also studied by Fernàndez, Béjar, Krishnamachari, and Gomes (2002). 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 (2014) 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 and the computation effort that takes the message delay into account, which was measured by the average of the equivalent non-concurrent constraint checks (Chechetka & Sycara, 2006). Finally, Samadidana and Mailler (2017) and Samadidana (2021) studied the impact of both message latency and message loss on distributed local search algorithms. The main difference with our work is that all of the work discussed above focused on DisCSP algorithms while we focused on DCOP algorithms. Further, the simulator used by Samadidana and Mailler (2017) and Samadidana (2021) is synchronous and the agents performed in cycles (i.e., in each cycle of the algorithm, each of the agents in its turn performed computation and sent out messages, even if it did not receive any messages). In contrast, our evaluations are on an asynchronous simulator, in which agents act only when they receive a message. As a result, we were able to make the interesting observation that the quality of solutions found may improve as the quality of communication degrades, both in terms of message latency and message loss.

4. Local Search in the Presence of Imperfect Communication

The first relevant observation one must make when analyzing the effect of imperfect communication on distributed local search algorithms is that local search algorithms for DCOP 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 (Zhang et al., 2005; Maheswaran et al., 2004; Zivan et al., 2014). Therefore, the presence of imperfect communication may cause major limitations. Message latency can postpone the iterative advancement of the algorithm and message loss may cause a deadlock as a result of an agent perpetually waiting for a message that is lost and never resent.

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 (Arshad & Silaghi, 2004).[3]

However, in the presence of imperfect communication, the asynchronous approach may also suffer consequences on the quality of solutions the algorithm explores:

1. An agent may take into consideration obsolete assignments of its neighbors when selecting alternative value assignments since the information regarding their replacement was sent but not yet received.

2. When messages can be delayed, messages may be received in a different order than they were sent (i.e., a message sent at time t from an agent A_i to agent 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 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. (2014) 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 issue mentioned above in our empirical evaluation (Section 5), where we present the quality of the solution as a function of the magnitude of latency and the probability for a message to be lost.

The second issue above can be addressed using a timestamp mechanism in which 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. Thus, A_j could reconstruct the order in which the messages were sent and it can ignore outdated messages (Lamport, 1978).

In order to overcome the third issue, we present in this section, *Latency-Aware Monotonic Distributed Local Search* (LAMDLS), a monotonic algorithm robust to message latency. While this algorithm addresses scenarios in which messages are delayed and not lost, we discuss the conditions for it to apply to situations that include message losses. Finally, for the fourth issue, we additionally present an *Asynchronous Anytime Mechanism* (AAM) that agents can use in an asynchronous environment in which messages may be delayed or lost.

4.1 Communication-Aware DCOP

In order to extend the DCOP model to represent communication disturbances, we designed a Constrained Communication Graph (CCG), that represents the uncertainty in the communication between agents.

To represent the communication limitations, in a CCG, every link of communication between agents (the vertexes) is represented by an edge and the constraint on each edge

3. 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 and message loss is novel.

defines the latency and probability of a message loss with respect to the communication exchanged on the link of communication represented by the edge.

Formally, for every edge e in the CCG, td_e is the function that represents the delay on e (i.e., it calculates the time between when a message is sent and when it is received via edge e). In addition, $pl_e \in [0, 1)$ is the probability that a message sent through the communication link represented by e is lost. Having defined the CCG graph, we use it to extend existing multi agent optimization models such as DCOP. The result would be a *communication-aware* model, that is, a *Communication-Aware DCOP* (CA-DCOP) that includes all the elements of a DCOP: $\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R}$, and $\mathcal{G}$, which represents a CCG. An instance of a CCG defines the message patterns on the communication links between agents, as described above.

4.2 Latency-Aware Monotonic Distributed Local Search (LAMDLS)

We now describe the LAMDLS algorithm, which assumes that messages can be delayed but are never lost. We briefly discuss how one can extend LAMDLS to handle message losses in Section 4.2.1.

The basic idea behind the design of the LAMDLS algorithm is that neighboring agents will not be allowed to perform calculations and decide whether to replace their value assignments concurrently. Towards 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 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 (2014):[4] Every agent whose index[5] is smaller than the indices of all of its neighbors selects the color 1 and sends it to its neighbors. When an agent receives the color from one of its neighbor, it stores that information. When an agent receives the colors of all of its smaller indexed neighbors, it selects a color with the smallest number, which is not selected by one of its smaller index neighbors, and sends it to its neighbors. Eventually, every agent will have selected a color that is different from the color of its neighbors.

We demonstrate the execution of DOCS by considering the constraint graph depicted in Figure 1, where nodes correspond to agents, and the colors of the agents are shown under the nodes. Agents A_1 and A_2 have no neighbors with smaller indices. Thus, they select the color 1 (blue) and send this information in messages to their neighbors. Among these neighbors, agents A_3, A_4, and A_7 have no other neighbor with smaller indices. Thus, they select the color 2 (red) and send this information in messages to their neighbors. Finally, agents A_5 and A_6 select the colors 1 (blue) and 3 (green), respectively, before sending that information to their neighbors.

We now describe the theoretical properties of this algorithm and examine its properties in the presence of message latency.

4. We do not present this algorithm in a formal pseudocode because of its simplicity and since it is not a novel contribution.
5. We assume that each agent has a unique index.

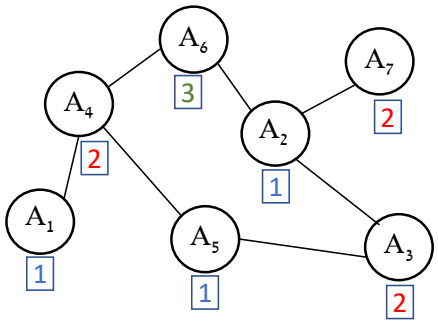

Figure 1: A numerical graph color partition.

Let n denote the number of agents in the graph; $color_i$ denote the color selected by agent A_i; 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 (computation step) time, which is $O(\Delta)$.

Proposition 1 *When DOCS converges, each agent $A_i \in \mathcal{A}$ has selected a color and stored the colors of all its neighbors.*

Proof: Assume that DOCS converged and there exists an agent A_i, where all of its neighbors with smaller indices have selected a color, and it did not. According to the algorithm, when it receives the color informing messages from all its smaller index neighbors, it will select a color. If the agent does not have smaller indexed neighbors, then it selects a color immediately, contradicting the assumption. If it does have smaller indexed neighbors, since messages are never lost, these messages will arrive, and agent A_i will select a color, which contradicts the assumption that the algorithm has converged. □

Proposition 2 *DOCS will converge after at most n iterations and, therefore, the time for convergence is at most $n(t + \delta)$.*

Proof: The worst case is when there is no concurrent selection of colors, resulting in the largest number of iterations needed by the algorithm. This happens, for example, in the case of a chain constraint graph, where agents are ordered according to their indices. Any other order that allows agents to select colors concurrently will speed up the process. In such a chain, there are $n - 1$ sequential messages, plus a message by the last agent in the chain informing its neighbors of its color. Thus, there are n sequential iterations, which each of them takes at most time t, and n sequential messages, where each of them takes at most time δ. □

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

Lemma 1 *When DOCS converges, the color selected by each agent $A_i \in \mathcal{A}$ is different from the colors selected by all its neighbors.*

Proof: Assume that DOCS converged and A_i chose an identical color to the one selected by its neighbor $A_j \in N(i)$ (i.e., $color_i = color_j$). Since indices are unique, assume

without loss of generality that $i < j$. Thus, A_j selects a color only after receiving the message that updates it with the selection of A_i, and the color it selects must be different than A_i. Since agents to not change their selection, this contradicts our assumption.□

Proposition 3 below was proven for a number of algorithms presented by Barenboim and Elkin (2014). Nevertheless, we provide a proof to ensure that our version preserves this property as well.

Proposition 3 *Upon the convergence of DOCS, the maximal number of colors selected by agents is* $\Delta + 1$.

Proof: We prove by induction on the number of agents n. When $n = 1$, $\Delta = 0$ and the number of colors is one. Assume inductively that the proposition holds for any number of agents $n = k$. We now provide the proof for $n = k + 1$: We remove the agent A_i with the largest index, and solve the problem using DOCS. According to our induction assumption, the number of colors selected by the agents is at most $\Delta + 1$. Further, the neighbors of agent A_i can hold no more than Δ colors since the agent has at most Δ neighbors and each agent can hold exactly one color. Thus, when agent A_i selects a color, it would select the one color among the $\Delta + 1$ existing colors that are not held by its neighbors. □

The ordered coloring division achieved by DOCS is used by LAMDLS as follows: 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 with its selection and they 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 of its neighbors with larger indices 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 $v^{N(i)}$ are initialized to zero and sc_i is initialized to one. After each computation step in which the agent considers to replace its value assignment, it increments sc_i by one. Furthermore, 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 pseudocode of LAMDLS, which can be executed after each agent selects its numerated color with DOCS. In LAMDLS, each agent A_i holds its set of neighbors $N(i)$, divided into two disjoint sets: One holding neighbors that have colors with smaller indices than its own $PC(i)$ and one holding the colors with larger indices $FC(i)$, 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. Each message from a neighbor A_j includes a value assignment and the neighbor's counter sc_j (line 6). Next, the agent updates its $local\_view$ and $v^{N(i)}$ with $value_j$ and sc_j (line 7). Then the agent checks if sc_i and $v^{N(i)}$ are consistent, if so, it increments sc_i by one, selects its value assignment, and sends a message including both to all of its neighbors (lines 8 – 11).

Algorithm 1: Latency-Aware Monotonic Distributed Local Search (LAMDLS)

```
   input : N(i),                                              // neighbors
           PC(i),                                // neighbors with larger color index
           FC(i)                                // neighbors with smaller color index
   output: value_i
 1 sc_i ← 1                                                   // local counter
 2 v^N(i) ← {0, 0, ..., 0}                          // initialize neighbors' counters
 3 value_i ← select_value                                     // assign its variable
 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)) then
 9 |   |   |   value_i ← select_value                           // assign its variable
10 |   |   |   sc_i ← sc_i + 1
11 |   |   |   send(value_i,sc_i) to N(i)
```

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 pseudocode, they are used for the consistency check. This is also true for the counters sc_i that are 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 of 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 4 *LAMDLS is weakly monotonic (i.e., each assignment replacement improves the global cost of the complete assignment held by the agents).*

Proof: Following Lemma 1, 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. Moreover, they can only replace assignments after receiving the current assignments of their neighbors. Thus, 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 5 *LAMDLS converges to a 1-opt solution.*

Proof: Each agent will get a chance to consider change its value after each message that it receives. Thus, if the algorithm has converged, it means that none of the agents decided to change their values, implying that none of the agents can improve their local costs. □

In the following paragraphs, we describe the start of a high-level trace for LAMDLS operating on the constraint graph presented in Figure 1. After the agents have selected

their colors, the algorithm is initialized. At this time, for each agent A_i, its sc_i equals 1 and the sc_j counters that it stores in $v^{N(i)}$ all equal 0. 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 computation steps. 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 (after receiving the messages of their neighbors the counters all equal 1). Agent A_7 can perform its computation step when it receives the message from A_1. At that time, its own counter will be 1, while the counter it holds for A_1 equals 2. 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 (i.e., wait until $v^{N(5)} = \{\langle A_3{:}2\rangle, \langle A_4{:}2\rangle\}$). 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.

LAMDLS is a synchronous incomplete search based algorithm (according to the taxonomy of Fioretto et al. (2018)). While it traverses the solution space, each agent maintains a solution candidate, and it is guaranteed to converge to a 1-opt solution as shown in Proposition 5. The agents replace their assignments in a specific order, which allows them to converge faster than the order used by MGM in the presence of message latency.

The algorithm's runtime complexity in a single computation step is $t = O(\Delta d)$, where d is the largest domain size and, as mentioned above, Δ is the maximal number of neighbors of an agent. Since selection of a value assignment aims to minimize the local constraint cost, an agent is required to check the cost of every value in its domain with the value assignments of all its neighbors (similar to DSA). In addition, a consistency check requires a runtime complexity of $O(\Delta)$, as an agent compares sc_i with each variable in $v^{N(i)}$. Thus, the runtime of the actions performed by an agent after each message it receives is $t = O(\Delta d + \Delta) = O(\Delta(1 + d)) = O(\Delta d)$.

The memory complexity required by LAMDLS is $O(\Delta)$ per agent since each agent needs to hold for each of its neighbors two constants. In term of communication characteristics, the message size is $O(1)$ and the number of messages is determined by the number of predefined communication steps.

4.2.1 Handling Message Loss

LAMDLS is monotonic and converges to a 1-opt solution. In contrast to asynchronous MGM, these properties are guaranteed by LAMDLS in the presence of message latency (Propositions 4 and 5). However, there may exist scenarios where messages may be lost and, as a result, LAMDLS may encounter a deadlock. Consider that a message sent by agent A_j to agent A_i does not reach its destination. In this case sc_i remains inconsistent with $v^{N(i)}$ and, thus, A_i continues to sit idle and the algorithm will not converge to a 1-opt solution.

In the general case, when no assumptions can be made on message arrivals, it is impossible to establish the properties of the algorithm. However, in many realistic cases, the probability that a number of consecutive messages sent from one agent to the other will not reach their destination tends to zero. Thus, in such situations, it is reasonable to assume that at least a single message sent by A_j will be delivered to A_i after k attempts.

A straightforward approach to avoid a deadlock in such a scenario is to send k messages on each edge in each step of the algorithm. Since one of these k messages is guaranteed to arrive at its destination, the necessary information required for the algorithm to proceed is guaranteed to be delivered. The down side of this approach is the increase in the use of communication. In future research, we plan to overcome this shortcoming by proposing methods in which the algorithm is guaranteed to converge to a 1-opt solution with a reduction in the use of the communication network (e.g., through the use of handshaking protocols similar to the TCP/IP protocol).

4.3 Asynchronous Anytime Mechanism (AAM)

The *Anytime Framework for Distributed Local Search Algorithms* (ALS) (Zivan et al., 2014) 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 (as mentioned in Section 2.3). However, its dependency on the synchronous structure is inherent and, therefore, it is not applicable for asynchronous algorithms. 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 operations that will improve the anytime global solution without the risk of reporting a low-quality solution as might happen if such exploration operations are used when the solution held by agents at the end of the run is reported.

We propose an *Asynchronous Anytime Mechanism* (AAM), which uses a spanning tree as used in the synchronous anytime framework proposed by Zivan et al. (2014). Algorithm 2 presents the pseudocode for the actions of agents within AAM. It is important to note that, unlike the pseudocode presented in Algorithm 1, which describes all actions of agents within a distributed search algorithm, here, we describe the actions related to the anytime mechanism, which are interleaved with any asynchronous local search algorithm that it is operating in conjunction with. To this end, the mechanism distinguishes between two types of messages: algorithm message (line 7) and AAM message (line 17).

Each agent A_i 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 subtree 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 generated either by actions of the algorithm (lines 8 – 9) or by the reception

Algorithm 2: Asynchronous Anytime Mechanism (AAM)

```
   input : P(i),                                   // A_i's parents in the tree
           C(i)                                    // A_i's children in the tree
   output: best_solution                           // context with smallest cost found
 1 is_leaf_i ← C(i) is empty                       // checks if A_i is a leaf
 2 CS(i) ← ∅                                       // context storage
 3 best_solution ← ∅
 4 best_cost ← ∞
 5 while algorithm's stop condition not met do
 6     when received message from A_j
 7         if receive algorithm message then
 8             update context_i in CS(i)
 9             if is_leaf_i AND cost(context_i) < best_cost then
10                 send (context_i, cost(context_i)) to P(i) // the agent is a leaf and
                       it reports a context with the best cost to its parents
11             else
12                 new_context = get_context(CS(i), context_i) // the agent is not
                       a leaf and it tries to generate a new context that is
                       consistent with the contexts in storage
13                 if new_context ≠ ∅ then
14                     if cost(new_context) < best_cost then
15                         send (new_context, cost(new_context)) to P(i) // if it
                               successfully generated a new consistent context
                               and the new context has the best cost, it reports
                               that context to its parents
16         if receive AAM message then
17             if receive context_j from A_j ∈ C_i then
18                 add context_j to CS(i)                // save context in storage
19                 new_context = get_context(CS(i),context_j)   // try to generate
                       a new context that is consistent with the contexts in
                       storage
20                 if new_context ≠ ∅ then
21                     if P(i) ≠ ∅ then
22                         send (new_context, cost(new_context)) to P(i)
23                     else if cost(new_context) < best_cost then
24                         best_solution ← new_context
25                         best_cost ← cost(new_context)
26                         send (best_solution,best_cost) to C(i)
27             if receive (best_solution_P(i), best_cost_P(i)) from P(i) then
28                 best_solution ← best_solution_P(i)  // if a context was received
                       from pseudo-parent, then it is the best solution found
29                 best_cost ← best_cost_P(i)
30                 remove contexts with cost ≥ best_cost
31                 send (best_solution,best_cost) to C(i)
```

of a context message from one of the children in the tree(line 18 – 19), and the anytime mechanism reacts to such changes.

After an update of $context_i$, if the the agent is a leaf in the spanning tree, it calculates its local cost ($cost(context_i)$), and examines whether the cost of the context generated does not exceed the best cost found for a solution. If the condition mentioned is met, the agent sends to its parent both its $context_i$ and $cost(context_i)$ (lines 10 – 11). Such a change in $context_i$ or in $cost(context_i)$ can be triggered either by an algorithmic message received from a neighbor or by a local decision an agent makes to replace its value assignment.

When a non-leaf agent A_i receives an AAM message from a child in the tree, it stores the context included in the message in its context storage $CS(i)$ (lines 18 – 19). Following any such message and after each context change, agent A_i seeks to generate a context that is consistent with all the contexts that it received from all of its children $C(i)$ (line 13 and line 20). If it is able to generate a consistent context, including assignments to all variables in the contexts sent to it by its children, and if the cost of this context is smaller than the best cost found so far, it sends the consistent context along with the corresponding cost to its parent (lines 14 – 16 and 21 – 23).

When the root agent generates a consistent context of all the variables in the problem, it checks whether its cost is the smallest found so far. If so, it stores it as the current best solution and sends it down the tree along with its cost (lines 24 – 27). Non-root agents, which receive a best solution message including the best cost, store them, filter out contexts with larger cost than the best cost, and pass the message on to their children in the tree (lines 28 and 32).

Proposition 6 *The solution reported by AAM will be a solution with the minimal cost among all the solutions 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 smaller cost than the best solution found thus far. Thus, a solution with the minimal cost must be considered as well. □

Proposition 7 *There can be a solution held by agents at some time during the algorithm that is not considered by AAM.*

Proof: Consider a problem with two neighboring agents A_1 and A_2, each holding a single variable with two possible values a and b in its domain. Initially, both agents assign a to their respective variables. At some point later, A_1 replaces its assignment to b and sends that information in 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 its value. 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 results that, indeed, an omniscient anytime mechanism with an access to a bird-eye view of the global cost finds better solutions than AAM. However,

this mechanism is implemented in a centralized manner. In a distributed environment with message latency, it is not feasible. In our empirical evaluation, we show that the results achieved with AAM are better than the results of the assignments that agents hold in the final iteration of the algorithm.

Proposition 8 *The maximal time*[6] *for an agent A_i to check if a new context can be generated is $(\theta_i \gamma_i)^{|C(i)|+1}$, where $\gamma_i = \max_{A_j \in C(i)}\{|context_i|, |context_j|\}$ with $|context_i|$ and $|context_j|$ denoting the number of agent-value assignments stored in the context and θ_i is the largest number of stored contexts received from a child agent or produced and stored by A_i.*

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

A corollary from Proposition 8 is that we prefer a tree with a small branching factor (e.g., a pseudo tree as we used in Section 5), in contrast to ALS, where a BFS tree with the smallest height is preferred.

While AAM guarantees to report the best solution explored by the asynchronous local search algorithm used, it requires the use of exponential memory for storing all contexts sent up the tree, and it requires exponential time for producing consistent contexts, while taking into consideration all contexts stored. This, of course, is not acceptable because the main reason for using a local search algorithm is to avoid exponential runtime and memory usage. Thus, in our experimental evaluation, we investigate the effect of limited space for storing contexts on the quality of the solution reported.

5. Experimental Evaluation

In this section, we first describe the design of our experiments before presenting the experimental results in Sections 5.2 to 5.5. Sections 5.2 and 5.3 describes the performance of asynchronous local search algorithms in the presence of message latency and loss, respectively. Section 5.4 presents the performance of LAMDLS in comparison to synchronous MGM. Finally, in Section 5.5, we evaluate AAM's ability to report high quality solutions.

5.1 Experimental Design

We investigate the performance of the CA-DCOP local search algorithms that were presented in the previous sections, using an asynchronous simulator, which was designed according to the guidelines by Zivan and Meisels (2006). Agents in our asynchronous simulator were implemented using Java threads. This simulator allows the implementation of any type of message delay pattern or any rate of message loss.[7] Other existing simulators for simulating message delays, such as ns-3 (Mayuga-Marcillo, Urquiza-Aguiar, & Paredes-Paredes, 2018; Amewuda, Katsriku, & Abdulai, 2018), offer a limited number of communication patterns from which one can select.

To evaluate the performance of distributed algorithms performing in a distributed environment, there is a need to establish which of the operations performed by agents could not

6. Throughout this paper, time is measured in *non-concurrent logic operations* (NCLOs) (Netzer, Grubshtein, & Meisels, 2012). We elaborate on this choice in Section 5.
7. The simulation's code is available at `https://github.com/benrachmut/CADCOP_2022_new`

have been performed concurrently and, thus, the runtime performance of the algorithm is the longest non-concurrent sequence of operations that the algorithm performed. In Zivan and Meisels (2006), DisCSP algorithms were evaluated, which their basic logic operations were constraint checks (CCs), thus, the performance was measured in terms of non-concurrent constraint checks (NCCCs). In Netzer et al. (2012), search based complete algorithms were compared with inference algorithms, thus, algorithms that perform different atomic logic operations (i.e., constraint checks and compatibility checks) were compared, and the results were reported in terms of non-concurrent logic operations (NCLOs). This approach is the one we adopt in this study, since we evaluate the quality of the solutions of the algorithms, as a function of the asynchronous advancement of the algorithm, when agents perform computation concurrently.

In each experiment, we randomly generated 100 different problem instances with 50 agents and we report the average over these instances. 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 100 NCLOs. We performed *t-tests* to evaluate the statistical significance of differences between all presented results.

In our simulator, message delays were simulated by passing all messages sent by agents to an abstract *mailing agent*. This abstract agent decides when to deliver the messages to their target agents, according to the selected pattern. The delay is selected in terms of the number of NCLO. Then, each agent has the opportunity to perform logic operations according to the algorithm instructions. In our implementation, each atomic logic operation was an evaluation of the cost of a combination of two value assignments (i.e., a constraint check).

In all the experiments we performed, we used three types of communication scenarios: (1) Perfect communication; (2) Message latency selected from a uniform distribution $td_e \sim U(0, UB)$ NCLOs, where UB is a parameter denoting the maximum latency; and (3) Message loss determined by $p \sim U(0, 1)$ such that a message is not delivered if $p < pl_e$, where pl_e is a parameter denoting the probability for message loss. Using the model described in Section 4.1, for each experimental scenario, an instance of a CCG was initiated, where td_e and pl_e remain similar for all edges in the graph. The magnitude of communication degradation was monitored through the parameters UB and pl_e.

We evaluated our algorithms on four different problem types. The first three problem types are commonly used in the DCOP literature and the last problem type is an *Overlapped Solar Systems* problem, which we describe later:

- **Uniform Random Problems :** These problems are random constraint graph topologies with density $p_1 = \{0.2, 0.7\}$. Each variable has 10 values in its domain. The constraint costs were selected uniformly between 1 and 100.

- **Graph Coloring Problems:** These problems are random constraint graph topologies in which each variable has three values (i.e., colors), and all constraints are "not-equal" cost functions, where an equal assignment of neighbors in the graph incurs a random cost between 10 and 100 and non-equal value assignments incur zero cost. We set the density $p_1 = 0.05$ similar to Zhang et al. (2005) and Farinelli et al. (2008).

- **Scale-free Network Problems:** These problems are generated using the Barabasi-Albert model (Barabási & Albert, 1999). An initial set of 10 agents was randomly

selected and connected. Additional agents were sequentially added and connected to 3 other agents with a probability proportional to the number of edges that the existing agents already had. The constraint costs were selected uniformly between 1 and 100. Each variable has 10 values in its domain. Similar problems were previously used to evaluate DCOP algorithms by Kiekintveld, Yin, Kumar, and Tambe (2010).

In addition to the problem types presented above, we also explored *Overlapped Solar Systems* problems. Overlapped solar systems is a realistic problem, inspired by the *Constant Speed Propagation Delay Model*, implemented in the ns-3 simulator (Mayuga-Marcillo et al., 2018; Amewuda et al., 2018). One characteristic that differentiates this problem in comparison to the three commonly used problems described above is an additional attribute of geographical locations of agents. The geographical locations allow us to model a dependency between the communication quality and the Euclidean distance between two agents. Therefore, instead of setting the td_e and pl_e to random values in the communication scenarios, we set them differently for this problem. Specifically, the td_e is drawn from a Poisson distribution $d \sim \text{Pois}(\Gamma \cdot distance_{ij})$, where Γ is a constant and $distance_{ij}$ determine the latency magnitude. This is also in contrast to the constant speed propagation delay model implemented in ns-3, where the delays that were calculated as a function of the distance between the geographic locations of the nodes were fixed and never changes (Mayuga-Marcillo et al., 2018; Amewuda et al., 2018). Regarding message loss, we define the probability pl_e that a message sent on edge e between agents A_i and A_j is delivered as follows: $pl_e = \frac{distance_{ij}}{maxDistance_{ij}}$, where $maxDistance_{ij}$ determines the distance of the furthest agents from A_i or A_j.

The formation of connections between the agents in overlapped solar systems problems is location dependent. Five agents are randomly selected to be the centers of the solar systems and they are connected. Each of these agents A_i^c is assigned two coordinates that are drawn from a continuous uniform distribution: $x_i^c \sim U(0,1)$ and $y_i^c \sim U(0,1)$. All other agents (i.e., stars in the solar systems) are randomly assigned to one of the solar systems. The index c represents the solar system in which the agent is assigned to, and it is equal to the index of the center agent of the solar system (i.e., if A_i^c is the center of a solar system, then $i = c$). The coordinates for an assigned agent (A_j^c where $j \neq c$) are drawn from a Normal distribution as follows: $x_j^c \sim N(\mu = x_i^c, \sigma = 0.05)$ and $y_j^c \sim N(\mu = y_i^c, \sigma = 0.05)$ based on the location of the center of the solar system that it was attached to. The probability that two arbitrary agents A_i and A_j will be neighbors is defined by $p_{ij} = (1 - \frac{distance_{ij}}{maxDistance_i})^\beta$, where $distance_{ij}$ is the Euclidean distance between agents A_i and A_j, $maxDistance_i$ is the Euclidean distance between agent A_i to the farthest agent, and β expresses the changes in the probability that both agents will be neighbors as a function of their distance (in our experiments we used $\beta = 3$). For each pair of agents, a random probability $p_r \in [0,1]$ was generated, and two agents are considered as neighbors if $p_r < p_{ij}$. The constraint costs were selected uniformly between 1 and 100.

5.2 Asynchronous Local Search in the Presence of Message Latency

Figure 2 presents the average quality of solutions produced by the synchronous (labeled SY) and asynchronous versions of DSA.[8] In the synchronous version, in each iteration, an

8. We used DSA-B with $p = 0.7$, to avoid assignment replacement in cases where an agent's local cost is zero.

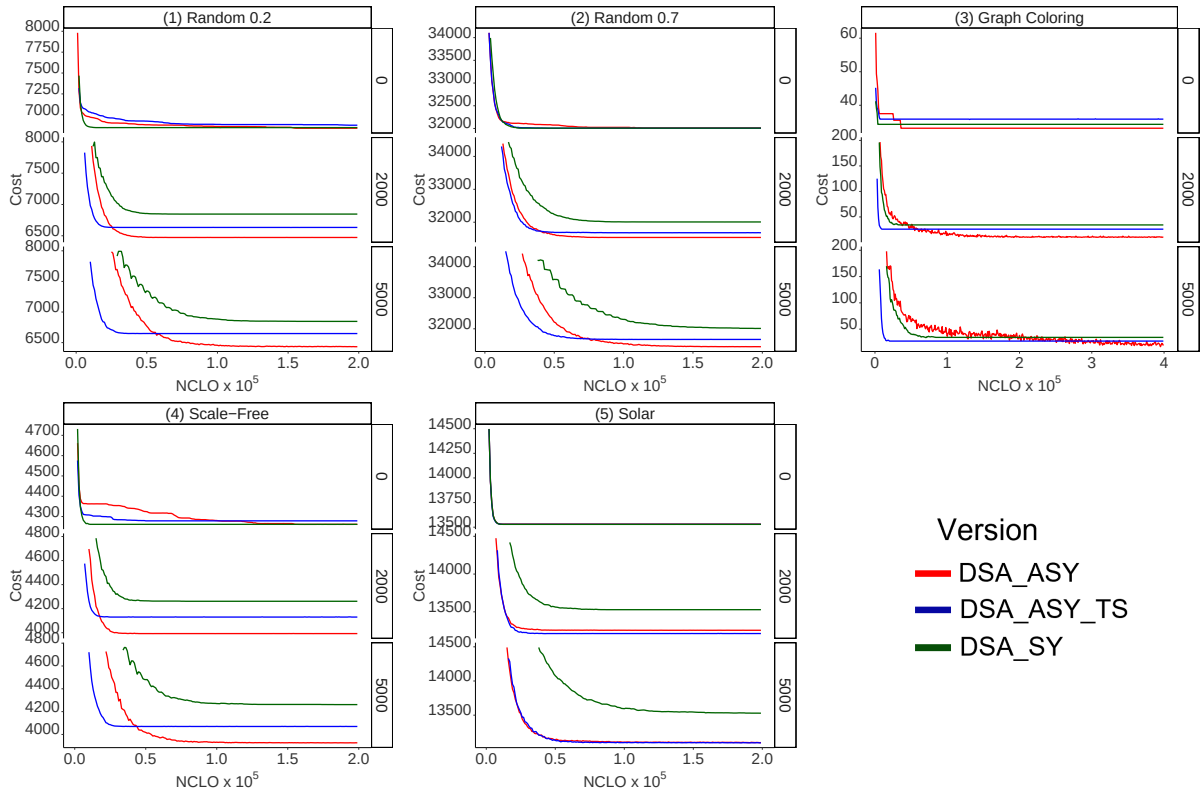

Figure 2: Costs of solutions for both synchronous and asynchronous DSA with different lengths of message delays sampled from a uniform distribution (Subgraphs 1-4) and Poisson distribution (Subgraph 5). Figure 14 in the appendix shows the corresponding color-blind friendly figure.

agent remains idle until the messages from all its neighbors are received. In contrast, in the asynchronous version, an agent initiates a computation step whenever it receives a message. We evaluated two asynchronous versions of the algorithm, one using timestamps (labeled ASY_TS) as suggested by Lamport (1978) (see Section 4 for more details) and one without timestamps (labeled ASY).

Each graph in the figure displays the results of the algorithm solving a different problem. The rows represent correspond to different magnitudes of latency – $UB = \{0, 2000, 5000\}$ for uniform random, graph coloring, and scale-free networks problems; $\Gamma = \{0, 2000, 5000\}$ for solar system problems. When UB or Γ equals 0, there is no latency. The results presented are the average global cost after each agent completed the number of NCLOs specified on the x-axis.

The different latency patterns affect the synchronous versions of DSA, where they converge slower for larger delays, but they converge to solutions of similar qualities. When the algorithms perform asynchronously, they generally converge faster and their differences with their synchronous counterparts increase with increasing magnitude of latency. In terms of solution quality, when there is no latency, both synchronous and asynchronous versions converge to similar solutions. However, when there is latency, the message delays have a

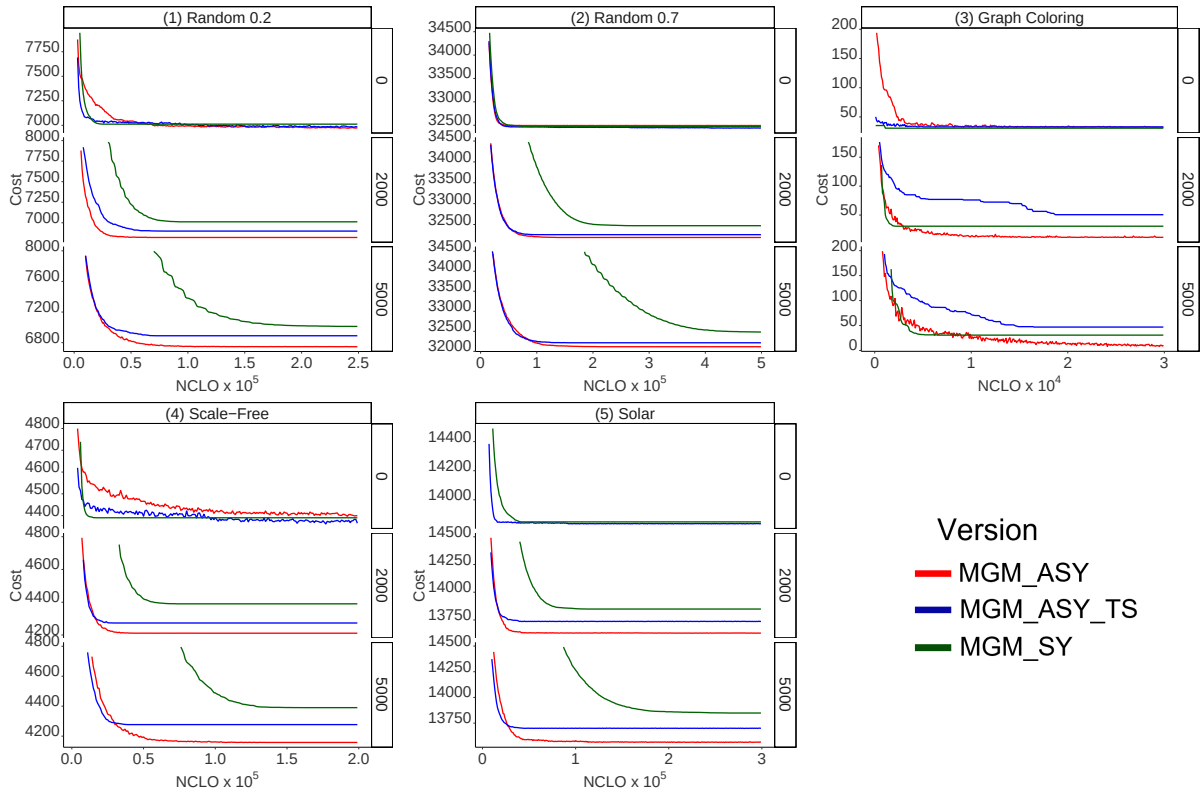

Figure 3: Costs of solutions for both synchronous and asynchronous MGM with different lengths of message delays sampled from a uniform distribution (Subgraphs 1-4) and Poisson distribution (Subgraph 5). Figure 15 in the appendix shows the corresponding color-blind friendly figure.

positive exploration effect on the asynchronous versions, allowing them to converge to better solutions.

Similar results are presented in Figure 3 for the experiments with MGM instead of DSA. In an asynchronous version of MGM, an agent perform computations each time it receives a new message. When it receives a value assignment message, it calculates its local reduction in cost and sends a cost reduction message to each of its neighbors. When the agent receives a cost reduction message, it checks whether it should change its value assignment by checking if its possible local reduction is the largest among the most recent local reduction messages that arrived from its neighbors. If it is the largest, then the agent changes its value assignment and sends value messages to all neighbors.

The results reveal that when messages are delayed, both asynchronous versions of MGM are not guaranteed to maintain their monotonic property. This is because agents can perform calculation and select value assignments while considering obsolete assignments of their neighbors (assignments that have been replaced). Nevertheless, losing monotonicity has a positive explorative effect in that they converge the better solutions than their synchronous counterpart.

One may expect the use of a timestamp to reduce inconsistent computation and improve the performance of the algorithms in the presence of message latency. However, the results show that the opposite happens – the use of timestamps reduces the explorative effect and the algorithm converges to worse solutions than without the timestamps. In contrast to the positive effect that the asynchronous versions have in the presence of message latency, in some of the cases, when messages are delivered instantly, the asynchronous algorithms have slower convergence rates. For MGM, it is most apparent that the version without timestamps converges slower, especially on graph coloring and scale-free networks. This is likely due to the two stage structure of MGM. In the asynchronous version, some of the agents may report inconsistent local reductions, before they were informed of their neighbors' assignments. Later, when there are fewer positive local reductions, this effect is reduced and the algorithm converges. For DSA, this phenomena is less apparent and is notable only when solving scale-free networks. A reasonable explanation for this is that in scale-free networks, there is a large difference between the computation of the hub agents and the computation performed by non-hub agents in each step of the algorithm. Overall, while the use of timestamps can improve the convergence rate, the difference in the convergence time is not significant compared to the improvement in solution quality when no timestamps are used.

To better evaluate the positive explorative effect due to message latency for asynchronous DSA, we ran an additional experiment where we sampled delays from a Poisson distribution. In Figure 4, we present the global cost of solutions found by asynchronous DSA, solving sparse uniform random problems, where each subfigure presents a different delay length (i.e., different values of λ, where $td_e \sim \text{Pois}(\lambda)$). The different curves present the different values of the parameter p used by the agents in DSA to determine whether to replace their value assignments. This parameter is used in DSA to determine the probability of simultaneous value assignment changes by neighbors and, thus, balance between exploration and exploitation (Zhang et al., 2005). In order to observe the differences between the performance of synchronous and asynchronous DSA, we also present the results of standard synchronous DSA with $p = 0.7$ (labeled 0.7_SY).

It is clear that the p value has a substantial effect on the performance of asynchronous DSA. When the delay is sampled from a Poisson distribution, for large delays, the performance improves with smaller values of p. The algorithm's convergence rate is faster with decreasing values of p in the presence of latency. The difference between the performance of the different versions is more apparent as the delay grows. For larger delays, the versions with the smaller p perform best.

Interestingly, when message delays are sampled from a Poisson distribution, the best results are achieved when asynchronous DSA uses low values of p (in contrast to the case where delays are sampled from a uniform distribution as depicted in Figure 2). This can be explained by the different level of exploration (in comparison to the case where delays are selected from a uniform distribution) that is correlated with the probability for a situation in which neighboring agents replace assignments concurrently. In standard synchronous DSA, an agent A_j sends a single message containing its value assignment to its neighbor A_i in each iteration. Thus, in cases where both agents have an improving alternative assignment, the probability that both agents replaced assignments concurrently is p^2. In asynchronous DSA, A_j may replace its value assignment following each message it receives. Thus, if it receives

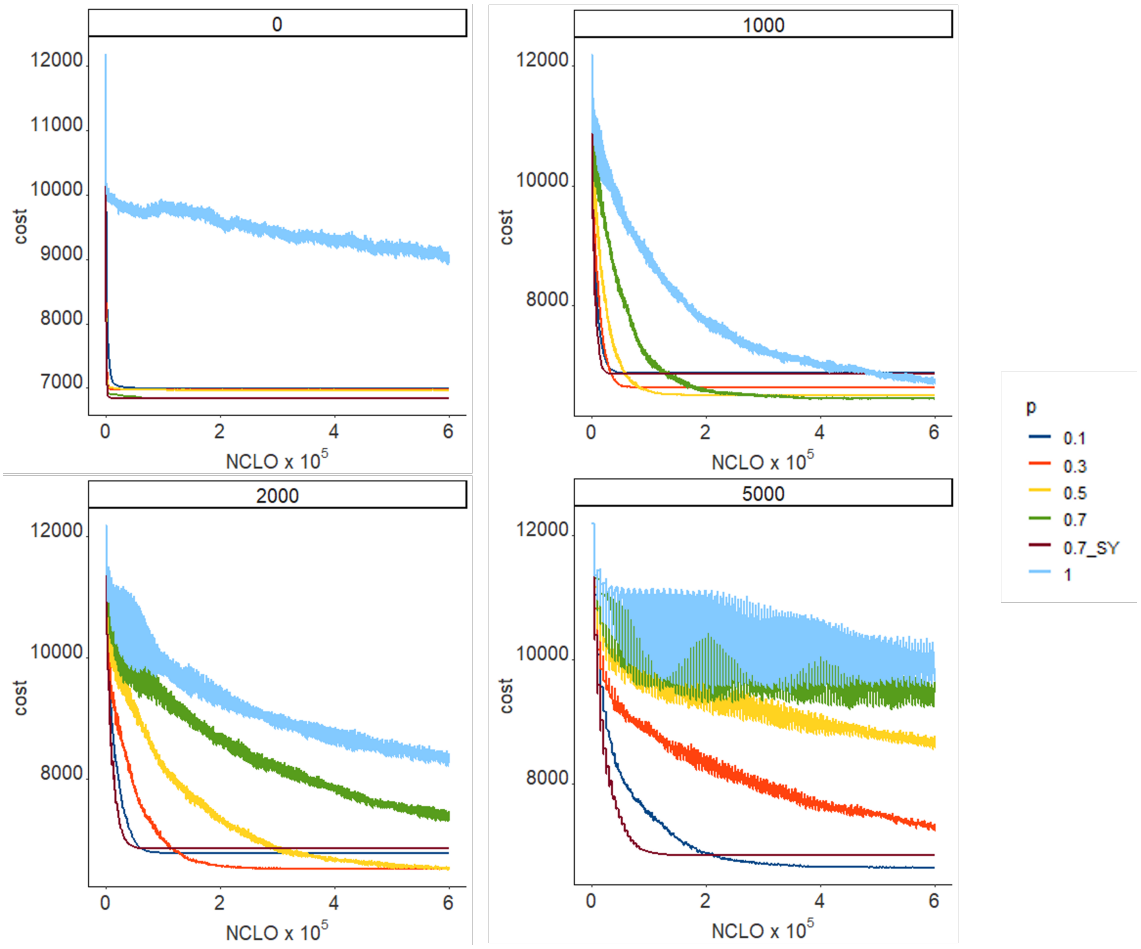

Figure 4: Solution cost of asynchronous DSA without timestamp, using different p values, when solving sparse uniform random problems, with delays sampled from a Poisson distribution. Figure 16 in the appendix shows the corresponding color-blind friendly figure.

a number of consecutive messages in a short time period, there is a much larger chance that it will perform the assignment replacement. In more detail, if neighboring agents A_j and A_i both have improving alternative value assignments, and they received k_j and k_i consecutive messages in a short time interval, respectively, then the probability that they both replaced their value assignment in this time interval is $(1-(1-p)^{k_j})(1-(1-p)^{k_i})$.

In order to highlight the relationship between message latency and exploration, we present in Figure 5 the local cost of a single agent during the run of asynchronous DSA when solving sparse uniform random problems. We depict both the agent's view, which takes into consideration the assignments included in the messages it received, and the actual cost considering the assignments of the neighboring agents at that time. The gap between the curves can be explained by the agents' asynchronous performance in the presence of imperfect communication, where an agent takes into consideration obsolete assignments of their neighbors when selecting alternative value assignments since the information regarding their replacement was sent but not yet received. Notice that in the beginning of the run (on

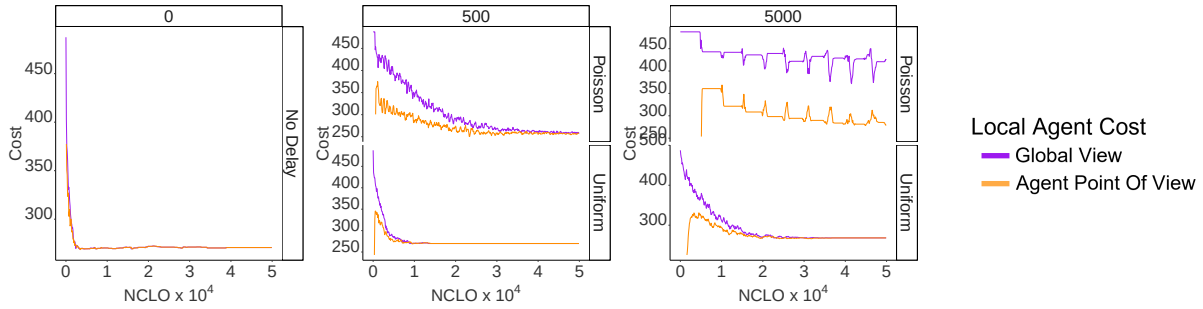

Figure 5: Local costs of a single agent performing asynchronous DSA, solving sparse uniform random problems. Figure 17 in the appendix shows the corresponding color-blind friendly figure.

the left of the subfigures), before the agents receive the assignments of their neighbors for the first time, they are not aware that they are violating constraints.

When the latency increases, the difference between the two disparate views increases and, thus, the agents perform actions that exploit obsolete information, which result in an increase in the cost. In essence, they are performing exploration despite thinking that they are performing exploitation.

5.3 Asynchronous Local Search Algorithms in the Presence of Message Loss

In the next set of experiments, we examined the effect of message loss on asynchronous versions of DSA and MGM without timestamps. Figure 6 presents the results of the two algorithms when solving the different benchmark problems for different probability pl_e values that a message will be lost. In the scenarios examined, messages that are not lost arrive at their destination instantly. Thus, the timestamp is insignificant. The mechanism avoids instances there messages are received in an order that is different than the one they were sent.

The effect of message loss on the algorithms when solving dense uniform random problems and solar systems problems was positive, similar to the effect of message latency. On the other hand, when solving sparse uniform random problems, the effect of message loss on the performance of the algorithms was negligible. For scale-free networks, asynchronous MGM finds better solutions when the messages have a high probability ($pl_e = 0.7$) to be lost. For graph coloring problems, which are less dense than other problems, the solution quality deteriorates as the probability of message loss increases.

5.4 Evaluation of Latency-Aware Monotonic Distributed Local Search (LAMDLS)

Figure 7 presents a comparison between the results of the proposed LAMDLS algorithm and the results of synchronous MGM (the only two monotonic and 1-opt algorithms among the ones we discussed) on uniform random problems. When there are no message delays, LAMDLS finds its first solution after MGM because LAMDLS does not search for solutions during the color selection phase. As expected, independent of message latency, each algorithm converges to the same result in terms of solution quality.

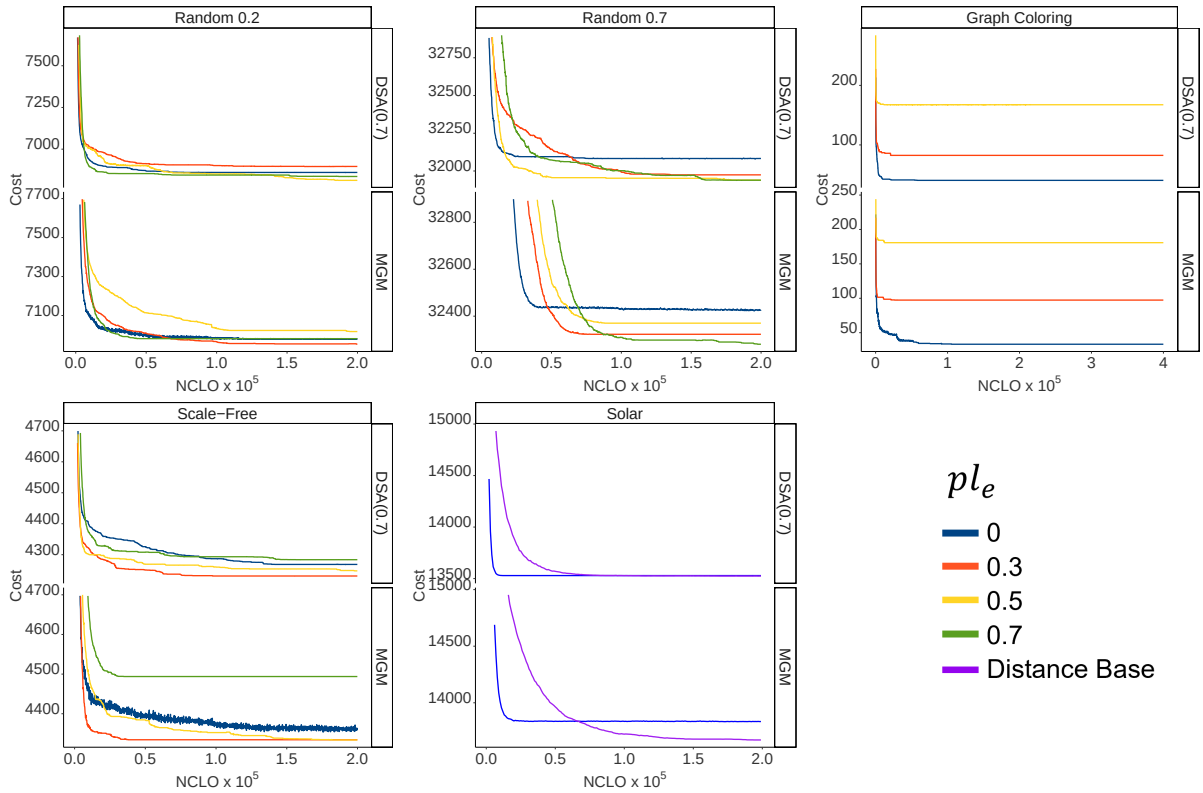

Figure 6: Solution quality of Asynchronous DSA and Asynchronous MGM, as a function of NCLOs. Experiments including various probabilities for message loss. Figure 18 in the appendix shows the corresponding color-blind friendly figure.

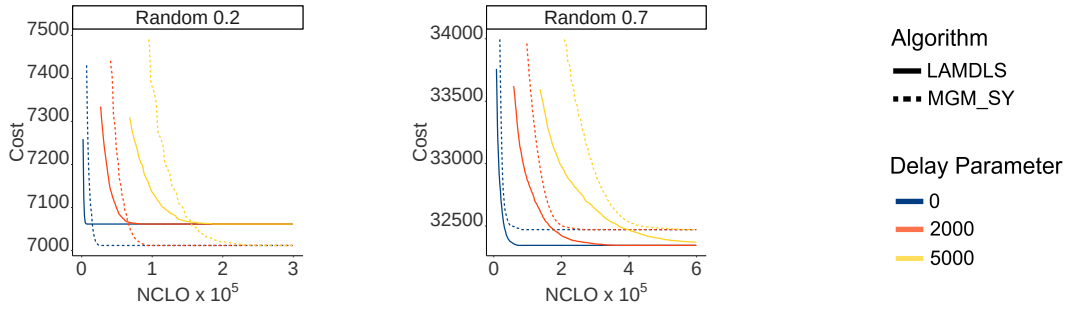

Figure 7: Solution quality of LAMDLS and synchronous MGM, as a function of NCLOs, solving sparse (left) and dense (right) uniform random problems. Figure 19 in the appendix shows the corresponding color-blind friendly figure.

Both algorithms converge faster on sparse problems than on dense problems. This is expected because both algorithms enforce that neighboring agents do not replace assignments concurrently and, thus, there is less concurrency in dense problems. Moreover, LAMDLS converged faster than MGM on sparse problems, but not on dense problems. The reason is that the ordered coloring method is less effective in generating concurrency in dense problems. Further, the density of problems affects the number of colors needed in the graph. The

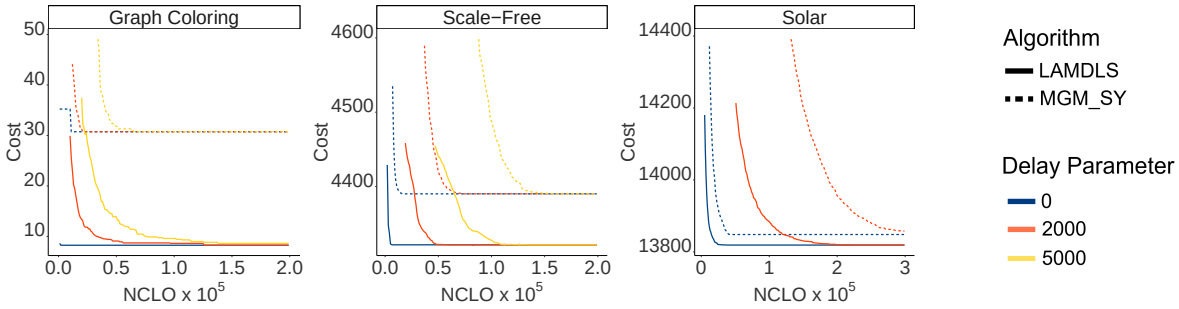

Figure 8: Solution quality of LAMDLS and synchronous MGM, as a function of NCLOs, solving Graph Coloring (left), Scale-Free Networks (middle) and Solar System (right) problems. Figure 20 in the appendix shows the corresponding color-blind friendly figure.

Graph Topology	**Paired t-test** p**-value**	**Shapiro-Wilk LAMDLS** p**-value**	**Shapiro-Wilk MGM_SY** p**-value**
Sparse Random	0.1300	0.4335	0.6558
Dense Random	0.0470	0.6534	0.2896
Graph Coloring	1.50×10^{-9}	2.20×10^{-16}	2.04×10^{-9}
Scale-Free Network	0.0059	0.5157	0.5002
Overlapped Solar System	0.4600	0.0031	0.0010

Table 1: Paired t-test p-value and Shapiro-Wilk normality test of solution quality for synchronous MGM and LAMDLS

average number of colors needed for sparse and dense problems was 7.04 and 17.83, respectively. While MGM had an advantage over LAMDLS on sparse problems, this advantage is not significant (results of the significance tests we performed are presented below).

Figure 8 presents the results of both algorithms solving structured problems, i.e., problems that either their topology (as in the case of scale-free nets) or their constraints (as in the case of graph coloring) have a predefined structure. On all these problems, LAMDLS converged faster than MGM. The graph coloring problems are sparser than the uniform random problems and, thus, these results are consistent with the ones presented above. The scale-free networks and the solar system problems are characterized by few hubs with many neighbors and most agents have a small number of neighbors. The color scheme structure of LAMDLS enables the non-hub agents to exchange value assignments concurrently. To support this claim, we checked the average number of colors that the algorithm used. The result was that the average number of colors was similar to the number of hubs – 10.3 and 12.85 for scale-free network and solar system problems, respectively.

In terms of solution quality, as expected, regardless of the magnitude of the message delays or type of problem, each algorithm converged to the same result. On all structured problems, LAMDLS find solutions with smaller costs than MGM. However, the significance of the results was problem dependent. The results of the statistical significance analysis (t-tests) of all the experiments presented in Figures 7 and 8 are presented in Table 1. Figure 9

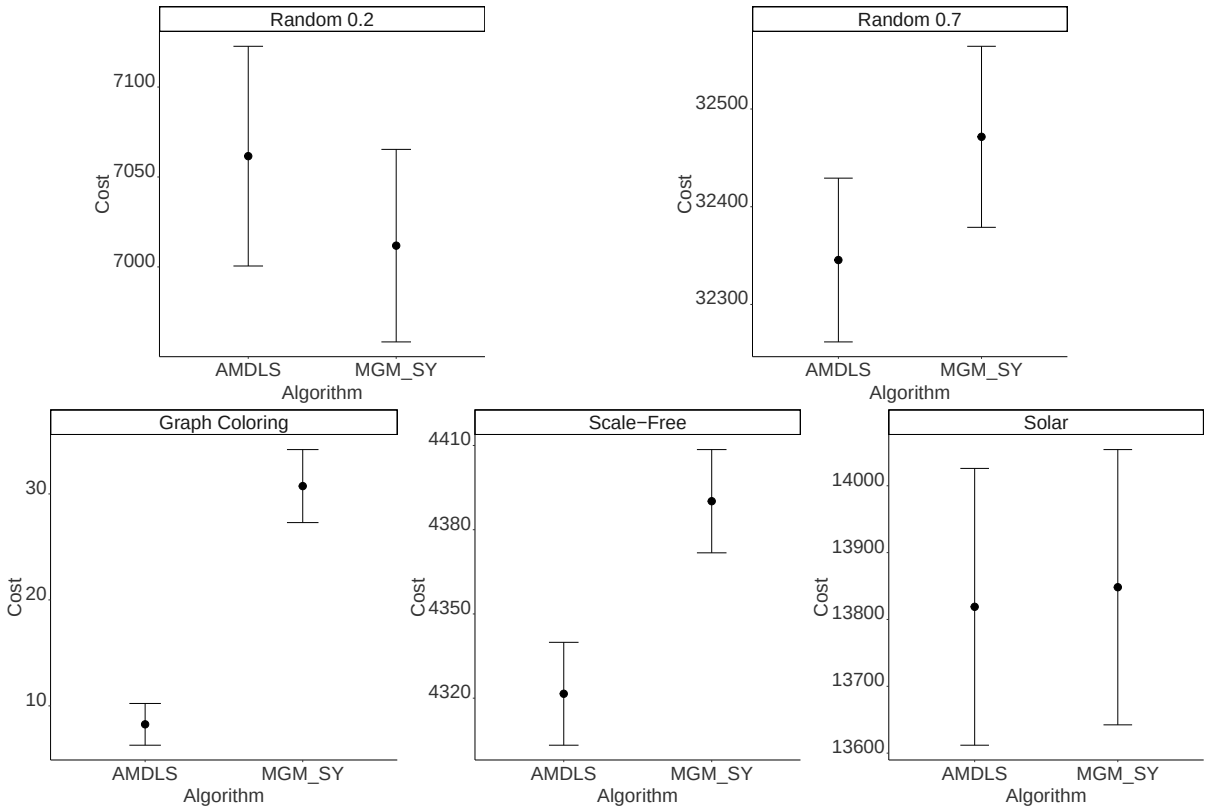

Figure 9: Average solution quality with standard error bars of LAMDLS and synchronous MGM at termination.

presents the average solution quality with error bars at termination.[9] They demonstrate that the significant differences between the two algorithms were observed on scale-free networks, and dense uniform random problems, in favor of LAMDLS.

5.5 Evaluation of Asynchronous Anytime Mechanism (AAM)

In order to demonstrate the effectiveness of AAM, we selected a highly explorative local search algorithm, DSA_SDP (Zivan et al., 2014), and implemented it in combination with AAM. Notice that anytime mechanisms do not contribute to monotonic algorithms such as synchronous MGM and LAMDLS and, therefore, DSA_SDP was introduced as an extremely explorative algorithm that ALS can use to improve the quality of solutions found (Zivan et al., 2014).

Figure 10 presents the results of an asynchronous version of DSA_SDP without memory limitations (left) and with memory limitations (right). We present the cost per iteration and the anytime cost, both from a global view, as well as the anytime cost of the solutions reported by AAM. As expected, following Proposition 8 and its corollary, the results reported by AAM are with larger costs than the global view anytime results, but they improve on

9. The experiments presented in figure 9 are with no message latency. Solution quality at termination is consistent with different message latencies due to the algorithms' 1-opt solution property. Thus, the algorithms' average costs in figure 9 are independent from the pattern of the message latency present.

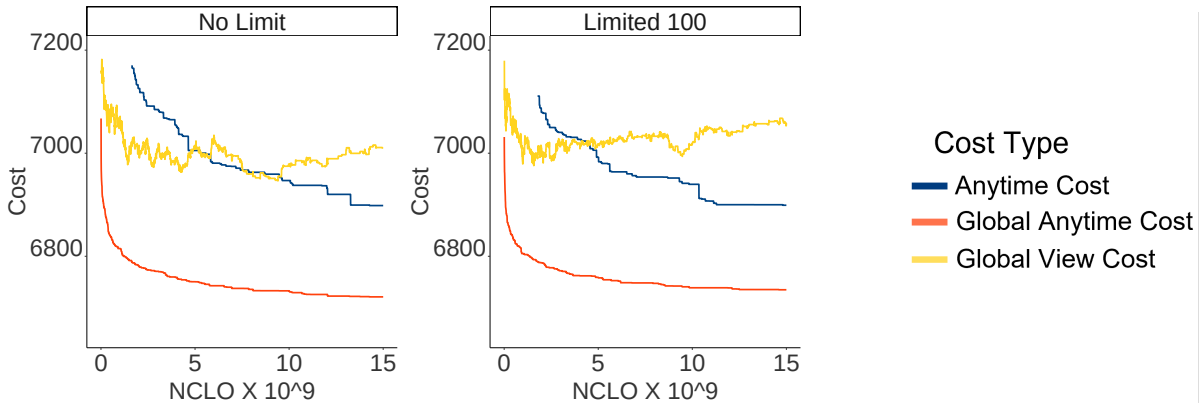

Figure 10: Global view solution cost, global view anytime cost and the anytime cost reported by AAM, of DSA_SDP solving sparse uniform random problems, without memory limitation (left) and with memory limitation to 100 contexts per agent (right). Messages were delivered with no delay. Figure 21 in the appendix shows the corresponding color-blind friendly figure.

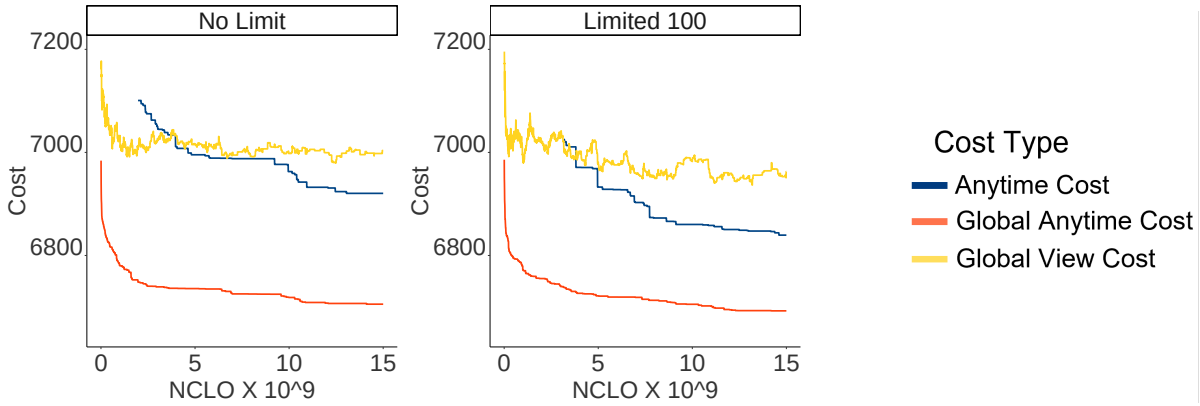

Figure 11: Global view solution cost, global view anytime cost and the anytime cost reported by AAM, of DSA_SDP solving sparse uniform random problems, without memory limitation (left) and with memory limitation to 100 contexts per agent (right). Message delays were sampled from a uniform distribution. Figure 22 in the appendix shows the corresponding color-blind friendly figure.

the results per iteration. Surprisingly, the differences between the results of the solutions found by the agents without memory limitations to the ones with memory limitations were not significant.

Figure 11 presents similar results in experiments in which message delays were selected from a uniform distribution with $UB = 1500$. In these experiments, AAM with memory limitations report solutions with smaller costs faster than AAM without memory limitations. The reason is that the context comparison computations are much faster when the number of contexts is limited.

Figures 12 and 13 present the results of similar experiments to the ones presented in Figures 10 and 11, except that this is on dense uniform random problems. The average costs reported by AAM are much closer to the global view anytime results. The reason is

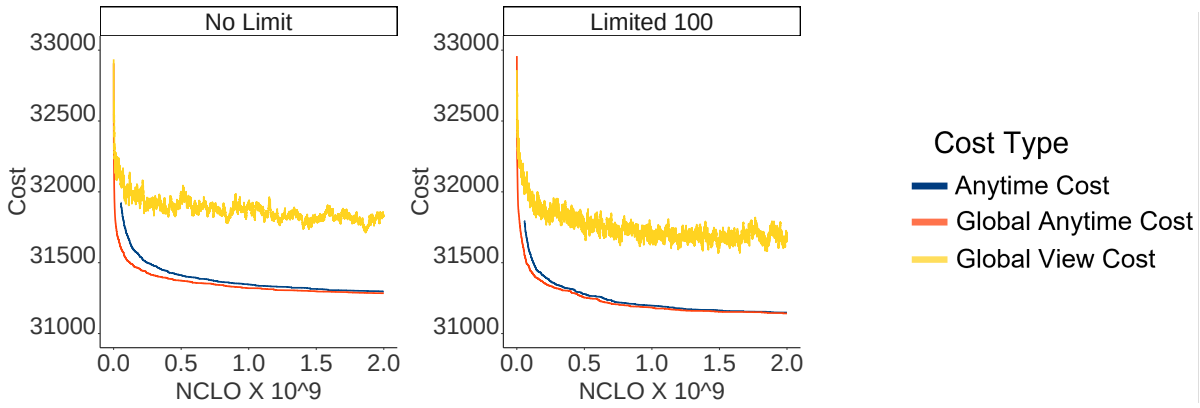

Figure 12: Global view solution cost, global view anytime cost and the anytime cost reported by AAM, of DSA_SDP solving dense uniform random problems, without memory limitation (left) and with memory limitation to 100 contexts per agent (right). Messages were delivered with no delay. Figure 23 in the appendix shows the corresponding color-blind friendly figure.

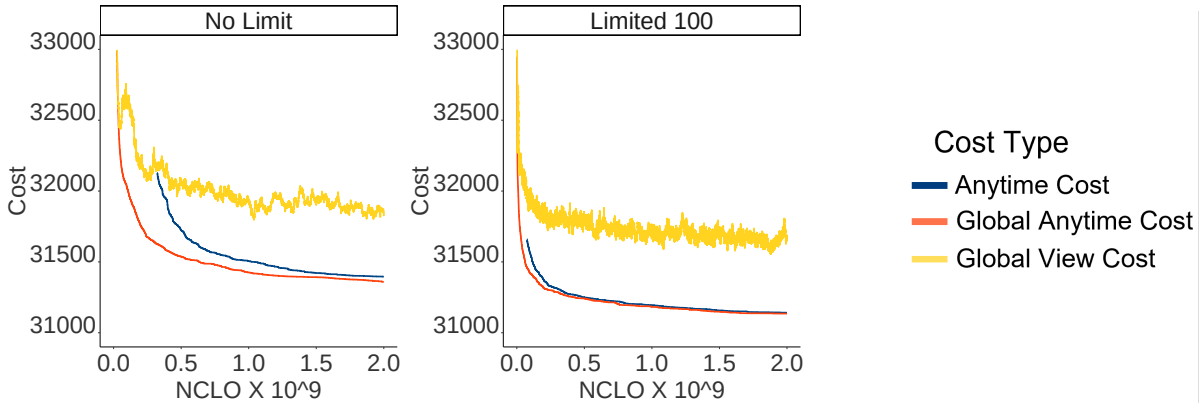

Figure 13: Global view solution cost, global view anytime cost and the anytime cost reported by AAM, of DSA_SDP solving dense uniform random problems, without memory limitation (left) and with memory limitation to 100 contexts per agent (right). Message delays were sampled from a Uniform distribution. Figure 24 in the appendix shows the corresponding color-blind friendly figure.

that the pseudo-tree used by the algorithm has fewer splits and therefore there are fewer context unifying operations. Again, the results with memory limitations are better than the results without memory limitations.

6. Conclusions

We investigated the implications of message latency and message loss on the performance of distributed local search algorithms for solving DCOPs. The synchronous structure of these algorithms imply that applying them in environments with message latency would result in a delay in execution, equal to the message with the largest delay sent in each iteration. The

implications of using these synchronous algorithms in the presence of message loss may be more severe, since agents might deadlock while waiting for a message that will not arrive.

To avoid exhibiting such a halt or delay of execution in environments with message latency or loss, we examined the performance of asynchronous versions of DSA and MGM, where agents perform a computation step and send messages following each message they receive. Surprisingly, this approach had a positive explorative effect on these algorithms. This phenomenon did not repeat itself when more explorative local search algorithms (e.g., DSA_SDP) were used to solve similar problems in environments with message latency and loss.

Some of the guaranteed theoretical properties of distributed local search algorithms are not preserved when using the asynchronous versions of the algorithms. The asynchronous version of MGM is not monotonic and the anytime mechanism proposed by Zivan et al. (2014) is not applicable to asynchronous distributed local search algorithms. Thus, we proposed a latency-aware monotonic distributed local search (LAMDLS) algorithm and an asynchronous anytime mechanism (AAM). As in the case of synchronous algorithms, LAMDLS was found to produce similar results to MGM (or better), only faster. AAM was found to report better results than the global view for an explorative distributed local search algorithm. These contributions thus fill an important gap in the literature on DCOPs operating in applications with imperfect communication.

While all the benchmarks used in our experimental study include binary DCOPs (and do not include problems with constraints of higher arity), we believe that this is appropriate since a message is sent from one agent to another and, therefore, the effect of a message delay or loss is binary.

In future work, we intend to investigate other types of constrained communication graphs and generate CA-DCOPs that are inspired by real-world problems, such as those with limited bandwidth and specific network topologies. For instance, in the smart home problem presented by Rust et al. (2022), algorithms aim to minimize the communication cost between agents. In such scenarios, an asynchronous approach may cause severe drawbacks. We also note that our solution for handling message loss is dependent on an assumption that may be considered oversimplistic. Therefore, we intend to propose methods for overcoming message loss in scenarios where the number of consecutive messages that are lost cannot be bounded. In addition, we intend to investigate the performance of other multi-agent optimization models and algorithms in the presence of imperfect communication, such as designated algorithms for solving distributed task allocation problems (Nelke & Zivan, 2017; Nelke, Okamoto, & Zivan, 2020).

Acknowledgments

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

Appendix A: Color-Blind Friendly Figures

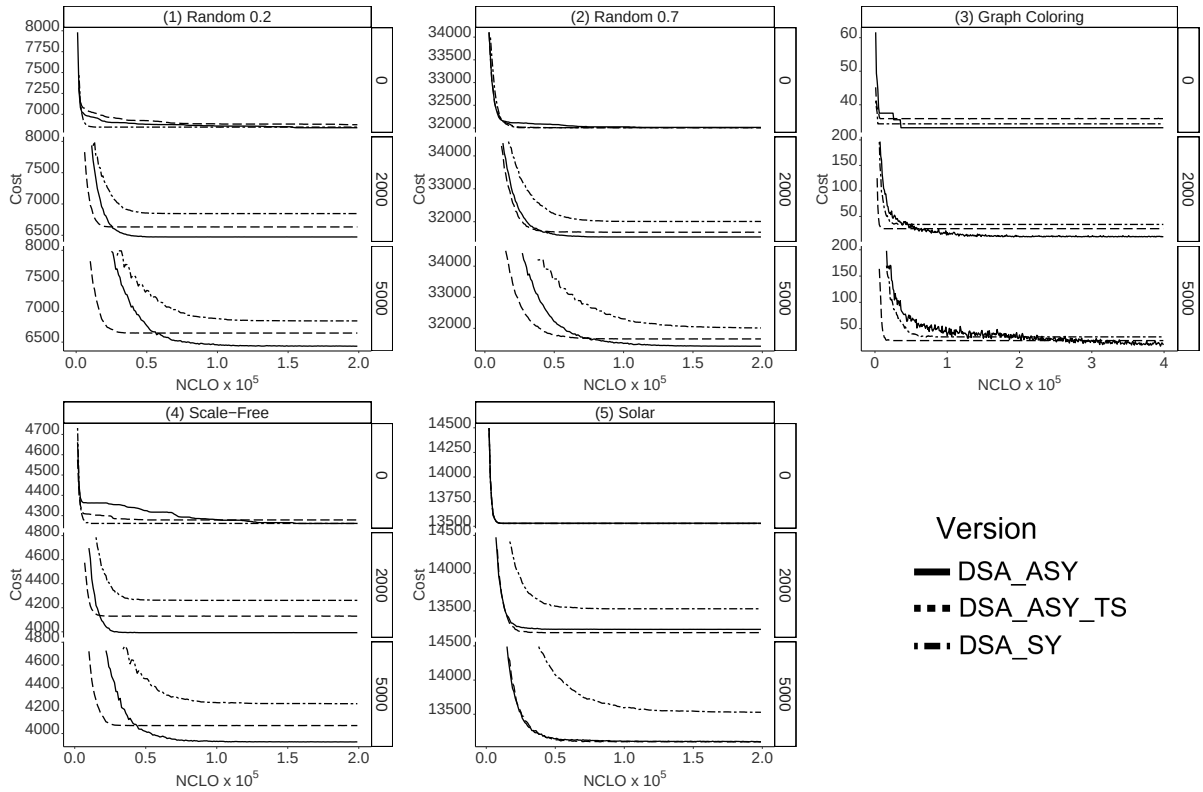

Figure 14: Costs of solutions for both synchronous and asynchronous DSA with different lengths of message delays sampled from a uniform distribution (Subgraphs 1-4) and Poisson distribution (Subgraph 5).

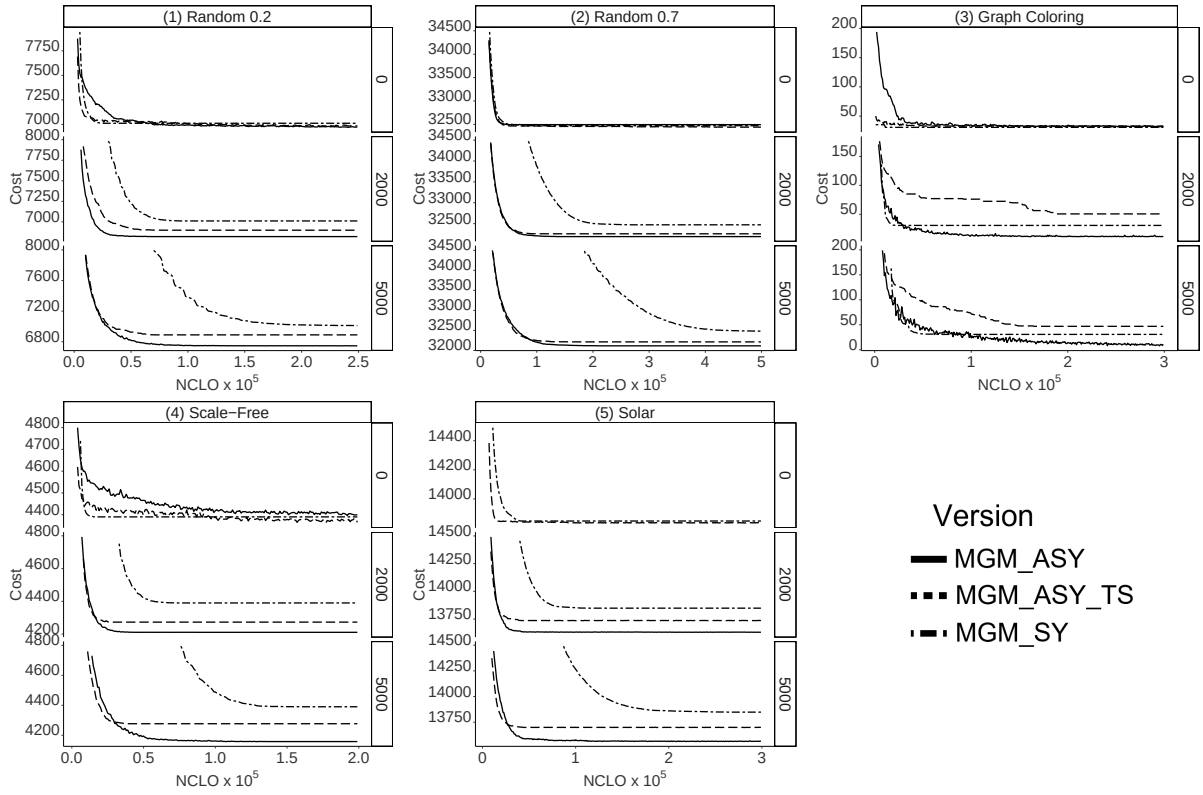

Figure 15: Costs of solutions for both synchronous and asynchronous MGM with different lengths of message delays sampled from a uniform distribution (Subgraphs 1-4) and Poisson distribution (Subgraph 5).

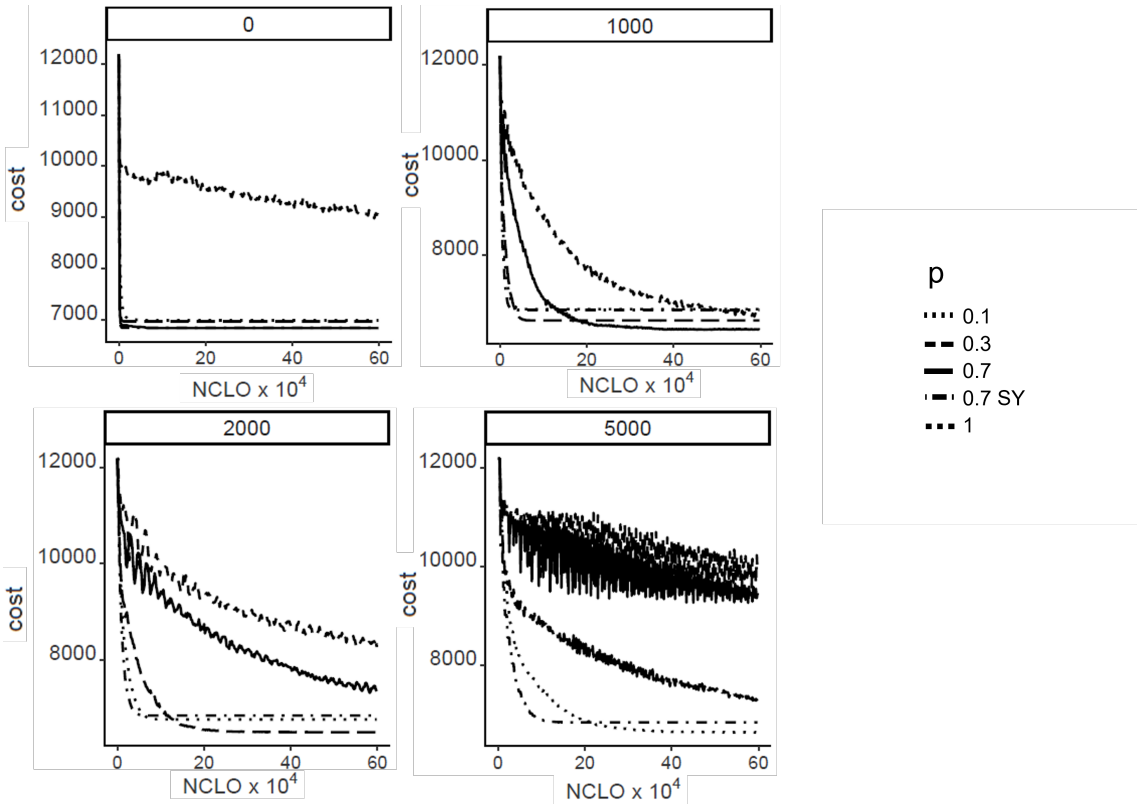

Figure 16: Solution cost of asynchronous DSA without timestamp, using different p values, when solving sparse uniform random problems, with delays sampled from a Poisson distribution.

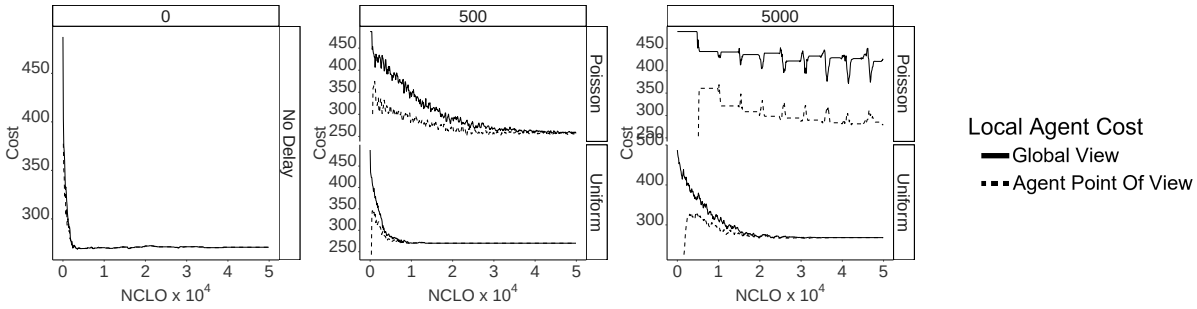

Figure 17: Local costs of a single agent performing asynchronous DSA, solving sparse uniform random problems.

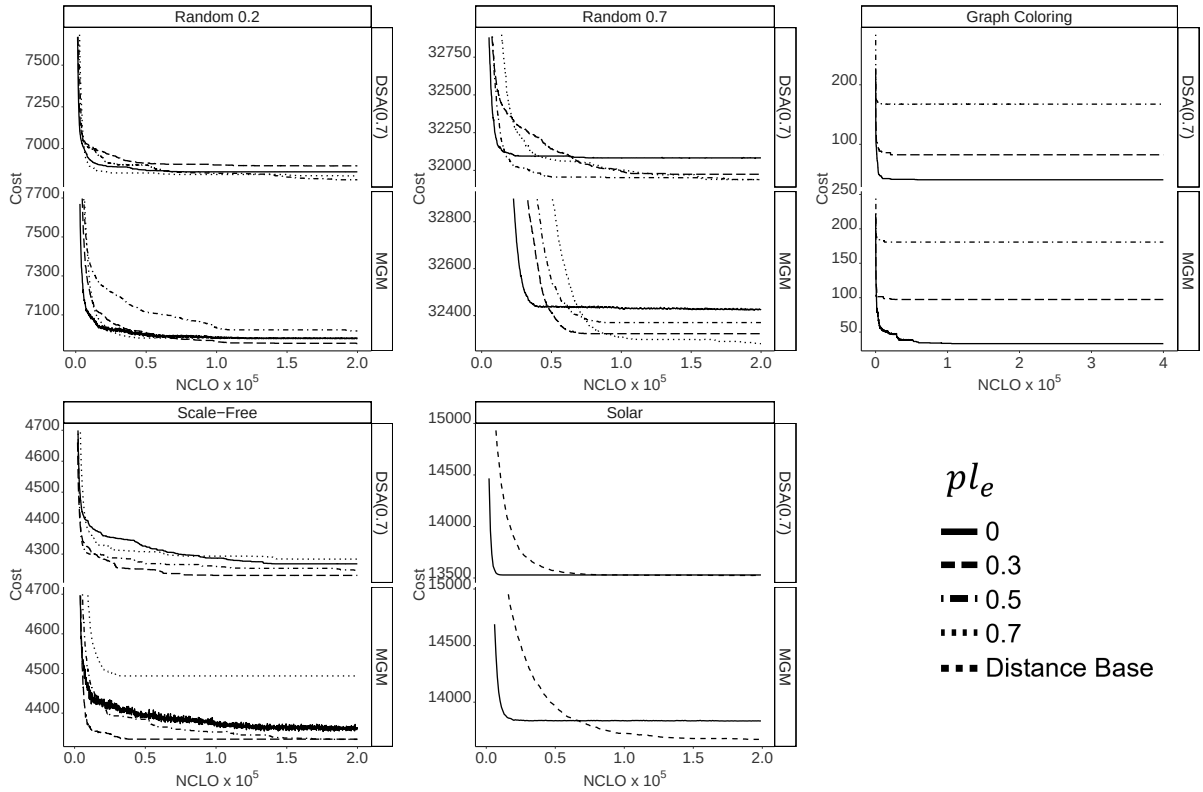

Figure 18: Solution quality of Asynchronous DSA and Asynchronous MGM, as a function of NCLOs. Experiments including various probabilities for message loss.

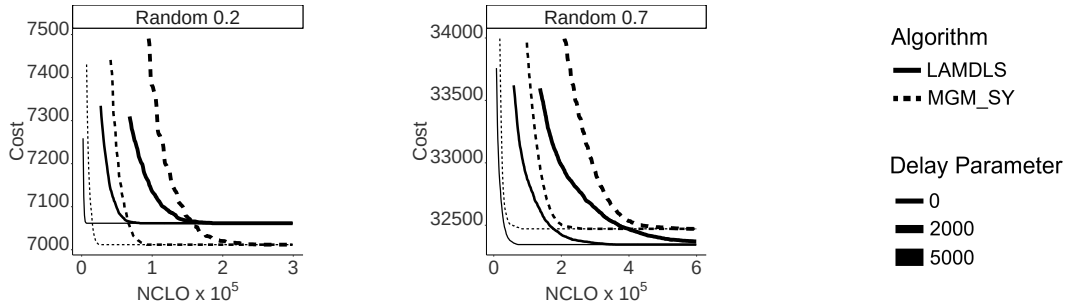

Figure 19: Solution quality of LAMDLS and synchronous MGM, as a function of NCLOs, solving sparse (left) and dense (right) uniform random problems.

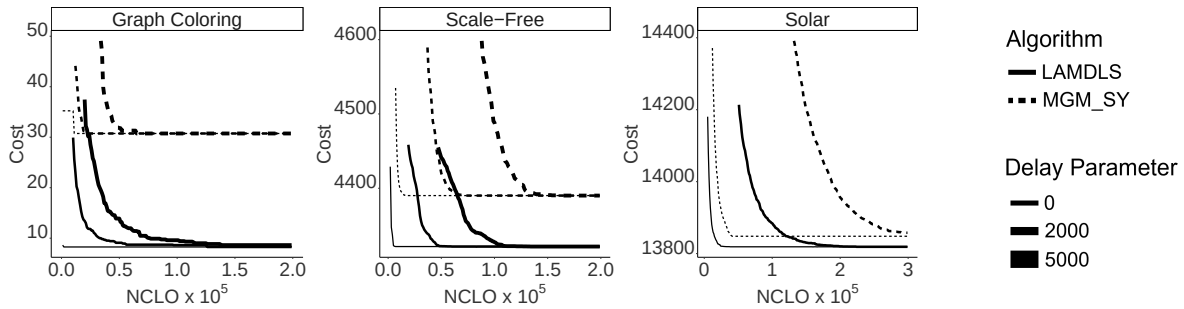

Figure 20: Solution quality of LAMDLS and synchronous MGM, as a function of NCLOs, solving Graph Coloring (left), Scale-Free Networks (middle) and Solar System (right) problems.

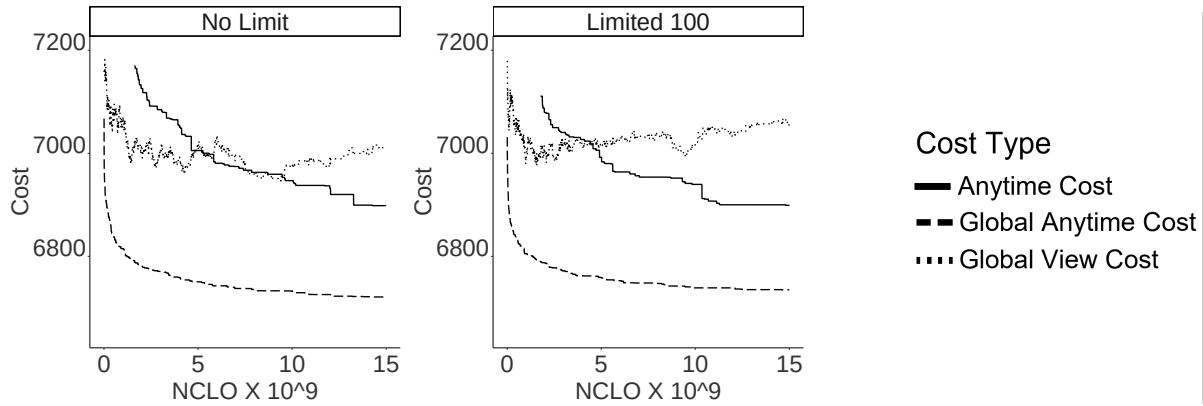

Figure 21: Global view solution cost, global view anytime cost and the anytime cost reported by AAM, of DSA_SDP solving sparse uniform random problems, without memory limitation (left) and with memory limitation to 100 contexts per agent (right). Messages were delivered with no delay.

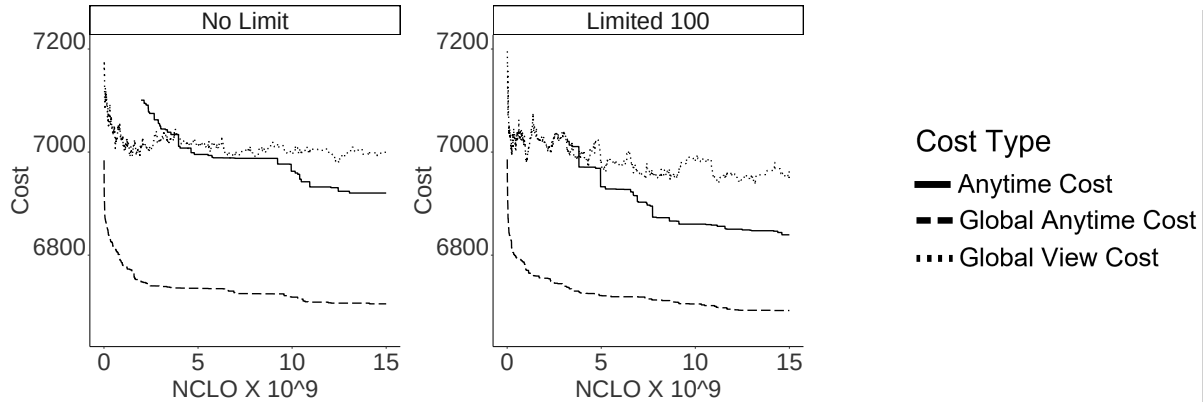

Figure 22: Global view solution cost, global view anytime cost and the anytime cost reported by AAM, of DSA_SDP solving sparse uniform random problems, without memory limitation (left) and with memory limitation to 100 contexts per agent (right). Message delays were sampled from a uniform distribution.

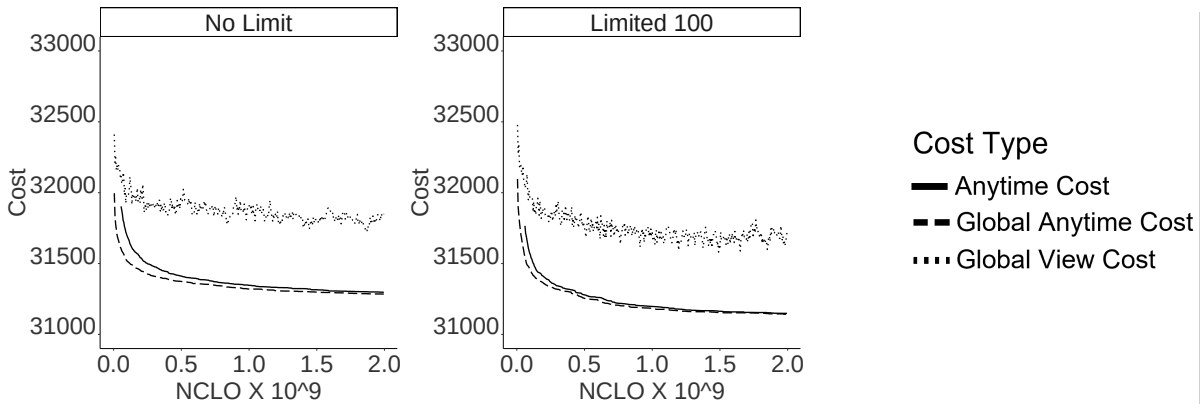

Figure 23: Global view solution cost, global view anytime cost and the anytime cost reported by AAM, of DSA_SDP solving dense uniform random problems, without memory limitation (left) and with memory limitation to 100 contexts per agent (right). Messages were delivered with no delay.

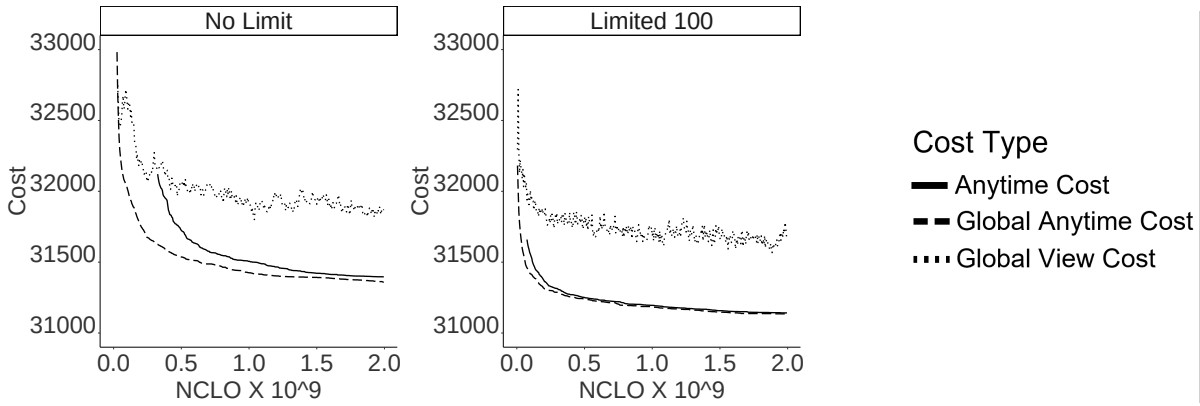

Figure 24: Global view solution cost, global view anytime cost and the anytime cost reported by AAM, of DSA_SDP solving dense uniform random problems, without memory limitation (left) and with memory limitation to 100 contexts per agent (right). Message delays were sampled from a Uniform distribution.

References

Amewuda, A. B., Katsriku, F. A., & Abdulai, J.-D. (2018). Implementation and evaluation of WLAN 802.11ac for residential networks in ns-3. *Journal of Computer Networks and Communications*, *2018*.

Arshad, M., & Silaghi, M. C. (2004). Distributed simulated annealing. *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, *112*.

Barabási, A.-L., & Albert, R. (1999). Emergence of scaling in random networks. *Science*, *286*(5439), 509–512.

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

Basharu, M., Arana, I., & Ahriz, H. (2005). Solving DisCSPs with penalty driven search. In *Proceedings of AAAI*, pp. 47–52.

Chechetka, A., & Sycara, K. (2006). No-commitment branch and bound search for distributed constraint optimization. In *Proceedings of AAMAS*, pp. 1427–1429.

Cruz, F., Gutierrez, P., & Meseguer, P. (2014). Simulation vs real execution in DCOP solving. In *Proceedings of the Distributed Constraint Reasoning Workshop*.

Farinelli, A., Rogers, A., Petcu, A., & Jennings, N. (2008). Decentralised coordination of low-power embedded devices using the Max-Sum algorithm. In *Proceedings of AAMAS*, pp. 639–646.

Fernàndez, C., Béjar, R., Krishnamachari, B., & Gomes, C. (2002). Communication and computation in distributed CSP algorithms. In *Proceedings of CP*, pp. 664–679.

Fioretto, F., Pontelli, E., & Yeoh, W. (2018). Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, *61*, 623–698.

Fioretto, F., Yeoh, W., & Pontelli, E. (2017). A multiagent system approach to scheduling devices in smart homes. In *Proceedings of AAMAS*, pp. 981–989.

Gershman, A., Meisels, A., & Zivan, R. (2009). Asynchronous forward bounding. *Journal of Artificial Intelligence Research*, *34*, 25–46.

Hoang, K. D., Fioretto, F., Yeoh, W., Pontelli, E., & Zivan, R. (2018). A large neighboring search schema for multi-agent optimization. In *Proceedings of CP*, pp. 688–706.

Kiekintveld, C., Yin, Z., Kumar, A., & Tambe, M. (2010). Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In *Proceedings of AAMAS*, pp. 133–140.

Lamport, L. (1978). Time, clocks, and the ordering of events in distributed system. *Communication of the ACM*, *2*, 95–114.

Lynch, N. A. (1997). *Distributed Algorithms*. Morgan Kaufmann Series.

Maheswaran, R., Pearce, J., & Tambe, M. (2004). Distributed algorithms for DCOP: A graphical game-based approach. In *Proceedings of PDCS*, pp. 432–439.

Mayuga-Marcillo, L., Urquiza-Aguiar, L., & Paredes-Paredes, M. (2018). Wireless channel 802.11 in ns-3. 10.20944/preprints201809.0367.v1.

McCanne, S., & Floyd, S. (2011). ns–network simulator.. http://nsnam.sourceforge.net/wiki/.

Modi, P. J., Shen, W., Tambe, M., & Yokoo, M. (2005). ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, *161*(1–2), 149–180.

Nelke, S. A., Okamoto, S., & Zivan, R. (2020). Market clearing-based dynamic multi-agent task allocation. *ACM Transactions of Intelligent Systems Technology*, *11*(1), 4:1–4:25.

Nelke, S. A., & Zivan, R. (2017). Incentivizing cooperation between heterogeneous agents in dynamic task allocation. In *Proceedings of AAMAS*, pp. 1082–1090.

Netzer, A., Grubshtein, A., & Meisels, A. (2012). Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence*, *193*, 186–216.

Nguyen, D. T., Yeoh, W., Lau, H. C., & Zivan, R. (2019). Distributed Gibbs: A linear-space sampling-based DCOP algorithm. *Journal of Artificial Intelligence Research*, *64*, 705–748.

Pearce, J. P., & Tambe, M. (2007). Quality guarantees on k-optimal solutions for distributed constraint optimization problems. In *Proceedings of IJCAI*, pp. 1446–1451.

Petcu, A., & Faltings, B. (2005). A scalable method for multiagent constraint optimization. In *Proceedings IJCAI*, pp. 1413–1420.

Rachmut, B., Zivan, R., & Yeoh, W. (2021). Latency-aware local search for distributed constraint optimization. In *Proceedings of AAMAS*, pp. 1019–1027.

Rust, P., Picard, G., & Ramparany, F. (2022). Resilient distributed constraint reasoning to autonomously configure and adapt IoT environments. *ACM Transactions on Internet Technology*.

Samadidana, S. (2021). Distributed constraint optimization problem solving in unstable environments. *American Journal of Science & Engineering*, *2*(1), 24–34.

Samadidana, S., & Mailler, R. (2017). Solving DCSP problems in highly degraded communication environments. In *Proceedings of WI*, pp. 348–355.

Smith, M., & Mailler, R. (2010). Getting what you pay for: Is exploration in distributed hill climbing really worth it?. In *Proceedings of IAT*, pp. 319–326.

Tabakhi, A. M., Tourani, R., Natividad, F., Yeoh, W., & Misra, S. (2017). Pseudo-tree construction heuristics for DCOPs and evaluations on the ns-2 network simulator. In *Proceedings of ICTAI*, pp. 1105–1112.

Tabakhi, A. M., Yeoh, W., Tourani, R., Natividad, F., & Misra, S. (2018). Communication-sensitive pseudo-tree heuristics for DCOP Algorithms. *International Journal on Artificial Intelligence Tools*, *7*(27), 1860008:1–1860008:24.

Wahbi, M., & Brown, K. N. (2014). The impact of wireless communication on distributed constraint satisfaction. In *Proceedings of CP*, pp. 738–754.

Yeoh, W., Felner, A., & Koenig, S. (2010). BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, *38*, 85–133.

Yokoo, M., & Hirayama, K. (1996). Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of AAMAS*.

Yokoo, M., & Hirayama, K. (2000). *Distributed Constraint Satisfaction Problems*. Springer Verlag.

Yokoo, M., Ishida, T., Durfee, E. H., & Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of ICDCS*, pp. 614–615.

Zhang, W., Wang, G., Xing, Z., & Wittenberg, L. (2005). Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, *161*(1–2), 55–87.

Zivan, R., & Meisels, A. (2006). Concurrent search for distributed csps. *Artificial Intelligence*, *170*, 440–461.

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

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