PC-SyncBB: A Privacy Preserving Collusion Secure DCOP Algorithm

Tamir Tassa[1], Tal Grinshpoun[2], Avishay Yanai[3]

[1]*Department of Mathematics and Computer Science, The Open University, Ra'anana, Israel*

[2]*Department of Industrial Engineering and Management and Ariel Cyber Innovation Center, Ariel University, Ariel, Israel*

[3]*VMware Research, Israel*

Abstract

In recent years, several studies proposed privacy-preserving algorithms for solving Distributed Constraint Optimization Problems (DCOPs). Those studies were based on existing DCOP solving algorithms, which they strengthened by implementing cryptographic weaponry that enabled performing the very same computation while protecting sensitive private data. All of those studies assumed that agents do not collude. In this study we propose the first privacy-preserving DCOP algorithm that is immune to coalitions. Our basic algorithm is secure against any coalition under the assumption of an honest majority (namely, the number of colluding agents is $< n/2$, where n is the overall number of agents). We then proceed to describe two variants of that basic algorithm: a more efficient variant that is secure against coalitions of size $\leq c$, for some constant $c < (n-1)/2$; and another variant that is immune to agent coalitions of any size, but relies on an external committee of mediators with an honest majority. Our algorithm – PC-SyncBB – is based on the classical Branch and Bound DCOP algorithm. It offers constraint, topology and decision privacy. We evaluate its performance on different benchmarks, problem sizes, and constraint densities. We show that achieving security against coalitions is feasible. Our experiments indicate that PC-SyncBB can run in reasonable time on problems involving up to 19 agents. As all existing privacy-preserving DCOP algorithms

base their security on assuming solitary conduct of the agents, we view this study as an essential first step towards lifting this potentially harmful assumption in all those algorithms.

Keywords: DCOP, Branch and Bound, Privacy, Multiparty Computation, Collusion-Secure

1. Introduction

Constraint optimization [1] is a powerful framework for describing optimization problems in terms of constraints. In many practical artificial intelligence applications, such as Meeting Scheduling [2], Mobile Sensor nets [3], and the Internet of Things [4], the constraints are enforced by distinct participants (agents). Hirayama and Yokoo [5] termed such problems as Distributed Constraint Optimization Problems (DCOPs). Various algorithms for solving DCOPs have been proposed, some of which are complete [5, 6, 7, 8, 9, 10], and some are incomplete [3, 11, 12, 13]. The complete algorithms ensure finding the optimal solution, and they compete in terms of efficiency, i.e., reducing the runtime and/or communication overhead. Incomplete algorithms, on the other hand, do not guarantee optimality, and the competition among them is usually with regard to solution quality.

The main motivation for DCOP research stems from the inherent distributed structure of many real-world problems, and the *privacy* concerns that are associated with this distribution. Léauté and Faltings [14] offered the basic definitions of privacy in this framework. The four notions of privacy that they describe are: agent privacy, topology privacy, constraint privacy and decision privacy. (We elaborate on those four types of privacy after providing the formal DCOP definitions in Section 2.) Several studies considered a solution of DCOPs in a manner that preserves (some of) those privacy types.

This line of research began with the work of Silaghi and Mitra [15]. They proposed a privacy-preserving solution to Distributed Weighted Constraint Satisfaction Problems (DisWCSPs); those are distributed problems that are similar to DCOPs, but differ from them in the distribution model and, consequently, in the related privacy targets. Their solution is strictly limited to small scale problems since it depends on an exhaustive search over all possible assignments. As their solution is based on the BGW protocol [16], it relies on the assumption of honest majority. To the best of our knowledge, the DisWCSP model [15] did not receive much focus in the past 15 years.

All of the subsequent studies considered DCOPs. The main motif in those studies was to develop privacy-preserving versions of existing algorithms. Greenstadt et al. [17] devised a version of the DPOP algorithm [9], called SSDPOP; that algorithm adds a secret sharing phase to the basic DPOP algorithm and, consequently, reduces the privacy loss of DPOP. Léauté and Faltings [14] proposed three privacy-preserving versions of DPOP – P-DPOP$^{(+)}$, P$^{3/2}$-DPOP$^{(+)}$ and P^{2}-DPOP$^{(+)}$ – that differ in their privacy guarantees and in their runtime performance. Grinshpoun and Tassa [18] developed P-SyncBB, a privacy-preserving version of the complete search algorithm SyncBB [5], which provides topology, constraint and decision privacy. Tassa et al. [19] presented P-Max-Sum, a privacy-preserving version of the incomplete inference-based Max-Sum algorithm [3], which respects topology, constraint and assignment/decision privacy. Lastly, Grinshpoun et al. [20] devised P-RODA, a secure implementation of region-optimal algorithms. First, the various existing algorithms in the region-optimality family (e.g., KOPT [11], DALO [21]) were encompassed by a framework called RODA (Region Optimal DCOP Algorithm). Next, a secure implementation of RODA, called P-RODA, was devised. P-RODA preserves constraint privacy and partial decision privacy.

The above described works on DCOP algorithms cover a variety of solution techniques, such as complete search (P-SyncBB), complete inference (the P-DPOP$^{(+)}$ family), incomplete inference (P-Max-Sum), and region optimality (P-RODA). However, all of those works based their security on assuming *solitary* conduct of the agents. Alas, subsets of agents may try to collude and combine the information which they have (consisting of their private inputs and messages received in the course of the DCOP solving algorithm) in order to infer information on other agents. For example, P-SyncBB [18] is no longer secure if the first agent A_1 colludes with some agent A_k, $k > 1$, since together they can find out the cost of a CPA that involves the variables $X_1, \ldots, X_k$ and, consequently, may learn information on private constraints between $X_2, \ldots, X_{k-1}$ (see [18, Section 4.5]).

In this paper we introduce the first privacy-preserving DCOP algorithm that is immune against such coalitions. We depart from the classical SyncBB algorithm [5] and devise PC-SyncBB, a Privacy-preserving and Collusion-secure Synchronous Branch and Bound algorithm, that completely simulates the operation of SyncBB and provides topology, constraint, and decision privacy, even in the presence of coalitions of agents. Our basic algorithm is secure under the assumption of an honest majority; namely, its privacy guarantees hold against any coalition of size smaller than $n/2$, where n is the number of agents. We then proceed to devise two variants of that algorithm. In one of them, we assume a stricter upper bound on the coalition size, i.e., that its size is $\leq c$ for some constant $c < (n-1)/2$. Such a limitation on the coalition size translates into higher efficiency in terms of runtime and communication costs. In another variant, we delegate some of the computations from the agents to an external committee of mediators. The model of using external mediators that assist in performing computations in a multi-agent environment and are trusted to do so honestly, but at the same time are not allowed access to private inputs of the agents, is known in cryptography as "the mediated model", see e.g. [22, 23, 24]. We show that if there exists an honest majority among the mediators, then that variant of PC-SyncBB is secure against any coalition among the agents and, in addition, it is more efficient than our basic variant (that is immune only against coalitions of size smaller than $n/2$).

The paper is outlined as follows. In Section 2 we provide the standard DCOP definitions. In Section 3 we describe the basic variant of PC-SyncBB and analyze its properties. Then, in Section 4, we describe the two additional variants of PC-SyncBB. In Section 5 we provide experimental results regarding the run-time performance and communication complexity of PC-SyncBB on different benchmarks, problem sizes, constraint densities, and coalition sizes. We conclude in Section 6.

A preliminary version of this paper was published at IJCAI 2019 conference [25]. The present journal version extends the preliminary version by including two additional variants of the algorithm, as described above (Section 4). Additionally, this journal version introduces an alternative computation for one of the algorithm's core sub-protocols (Section 3.4), which significantly improves the overall performance of the algorithm. Finally, the current version includes complete proofs (Sections 3.3 and 3.5), an extended and comprehensive experimental evaluation (Section 5), and a detailed example (Appendix A).

2. DCOP definitions

A Distributed Constraint Optimization Problem (DCOP, [5]) is a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ where $\mathcal{A}$ is a set of agents $A_1, A_2, \ldots, A_n$, $\mathcal{X}$ is a set of variables $X_1, X_2, \ldots, X_m$, $\mathcal{D}$ is a set of finite domains $D_1, D_2, \ldots, D_m$, and $\mathcal{R}$ is a set of relations (constraints). Each variable X_i takes values in the domain D_i, and it is held by a single agent. Each constraint $C \in \mathcal{R}$ defines a non-negative cost for every possible value combination of a set of variables, and is of the form $C : D_{i_1} \times \cdots \times D_{i_k} \to [0, q]$, for some $1 \le i_1 < \cdots < i_k \le m$, and a publicly known maximal constraint cost q.[1]

An *assignment* is a pair including a variable, and a value from that variable's domain. We denote by a_i the value assigned to the variable X_i. A *partial assignment* (PA) is a set of assignments in which each variable appears at most once. A constraint $C \in \mathcal{R}$ is *applicable* to a PA if all variables that are constrained by C are included in the PA. The cost of a PA is the sum of all applicable constraints to the PA. A *full assignment* is a partial assignment that includes all of the variables. The goal in Constraint Optimization Problems is to find a full assignment of minimal cost.

[1] Our framework can include also the case of *hard* constraints, i.e., combinations of assignments that are strictly forbidden, see [18].

For simplicity, we assume that each agent holds exactly one variable, i.e., $n = m$. We let n denote hereinafter the number of agents and the number of variables. We consider a binary version of DCOPs, in which every $C \in \mathcal{R}$ constraints exactly two variables and takes the form $C_{i,j} : D_i \times D_j \to [0, q]$. These assumptions are customary in DCOP literature, see e.g. [8, 9].

Léauté and Faltings [14] have distinguished between four notions of privacy.

- *Agent privacy* – hiding from each agent the identity or even the existence of other agents with whom he is not constrained.
- *Topology privacy* – hiding from each agent the topological structures in the constraint graph (namely, the graph over the set of variables where an edge connects two variables iff there is a constraint that relates them) beyond his[2] own direct neighborhood in the graph.
- *Constraint privacy* – hiding from each agent the constraints in which he is not involved. Namely, agent A_k should not know anything about $C_{i,j}(\cdot, \cdot)$ if $k \notin \{i, j\}$.
- *Assignment/Decision privacy* – hiding from each agent the intermediate/final assignments to other variables.

The notions of privacy which our proposed algorithm respects are topology, constraint and assignment/decision privacy.

3. A Secure Synchronous Branch and Bound

Synchronous Branch-and-Bound (SyncBB) [5] was the first complete algorithm for solving DCOPs. SyncBB operates in a completely sequential manner, a fact that inherently renders its synchronous behavior. It is also the most basic *search* algorithm for solving DCOPs, and other more sophisticated DCOP search algorithms, such as NCBB [26], AFB [6] and BnB-ADOPT [10], use the

[2] We use the masculine form for simplicity.

Branch and Bound structure as a core ingredient. The SyncBB algorithm assumes a static public ordering of the agents, $A_1, \ldots, A_n$. The search space of the problem is traversed by each agent assigning a value to his variable and passing the *current partial assignment* (CPA) to the next agent in the order, along with the current cost of the CPA. After an agent completes assigning all values in the domain to his variable, he *backtracks*, i.e., he sends the CPA back to the preceding agent. To prevent exhaustive traversal of the entire search space, the agents maintain an *upper bound*, which is the cost of the best solution that was found thus far. The algorithm keeps comparing the costs of CPAs and the current upper bound, in order to *prune* the search space.

Herein we devise a secure implementation of SyncBB, called PC-SyncBB (Privacy-preserving and Collusion-resistant SyncBB). Another secure implementation of SyncBB, called P-SyncBB, was previously introduced by Grinshpoun and Tassa [18, 27]. The two algorithms are fundamentally different. While in P-SyncBB agents are exposed to sensitive information such as assignments of other agents, costs of CPAs, and the value of the upper bound, PC-SyncBB avoids such information disclosure, as indicated above. Hence, the outline of PC-SyncBB is simpler than that of P-SyncBB, because there is no need to implement mechanisms for preventing illegal inferences that can be deduced from such information. For example, as in P-SyncBB agents are informed of the CPA, they can infer the final decision of other agents. To prevent that (in order to achieve decision privacy), P-SyncBB implements a delicate cryptographic mechanism; such a mechanism is not needed in PC-SyncBB, since it keeps assignment information secret and it performs computations on secret data. On the other hand, as in PC-SyncBB much less information is revealed, and as PC-SyncBB is designed to be resistant to coalitions (while P-SyncBB is not), the secure multiparty computational tasks in PC-SyncBB are harder. Hence, the cryptographic approach taken in PC-SyncBB is completely different and it is much more involved than the corresponding one in P-SyncBB. The most prominent example is the problem of verifying inequalities between values that are held by more than one agent, as happens each time the cost of the CPA is compared to the upper bound; the secure multiparty computation that PC-SyncBB has to invoke to solve such problems is much more intricate than the one that P-SyncBB invokes, since in PC-SyncBB such inequality verifications need to be performed over data which is distributed among *all* agents, and it is needed to do so in a manner that is resistant to coalitions.

This section is organized as follows. In Section 3.1 we discuss the setting in which PC-SyncBB operates, introduce notations that we will use, and describe the main internal variables that each agent holds. The algorithm is given in Section 3.2. The secure multiparty sub-protocols that the algorithm invokes are described in Sections 3.3 and 3.4. We discuss the properties of PC-SyncBB in Section 3.5. A detailed example illustrating the operation of PC-SyncBB is presented in Appendix A.

3.1. Preliminaries

General assumptions and notations. The design of PC-SyncBB is based on several general assumptions. The first two assumptions are inherent in the SyncBB algorithm [5] and its derivatives, e.g. [6, 18], whereas the other assumptions are specific for PC-SyncBB:

(1) All agents can directly communicate with each other, even if they are not constrained. In particular, broadcast is allowed.

(2) There is a static public ordering of the agents, $A_1, \ldots, A_n$.

(3) The upper bound on the cost of any possible solution is $q_\infty := \binom{n}{2}q + 1$, and it is known to all agents.

(4) Cryptographical computations always take place over a finite algebraic domain (typically a group or a field). The cryptographical selections that we made in the design of PC-SyncBB, require the algorithm to operate over a finite prime-ordered field. We let S denote the size of that field. Hence, all agents agree upfront on a prime S that is greater than q_∞. The latter restriction is essential since it ensures that all computed values (which relate to costs of CPAs) can be uniquely represented in that field.

(5) For every pair of indices $1 \leq t < k \leq n$, $\Gamma(t,k)$ is a Boolean predicate that equals **true** iff X_t and X_k are constrained. Then, $I_k^- := \{t : 1 \leq t < k \text{ and } \Gamma(t,k)\}$ and $I_k^+ := \{t : k < t \leq n \text{ and } \Gamma(k,t)\}$ are sets containing the indices of all agents that precede/follow A_k in the order and whose variable is constrained with X_k. We also let $I_k := I_k^- \cup I_k^+$.

Value ordering. Each agent A_k maintains two value orderings over his domain D_k. Each of those orderings can be described by a vector of length $|D_k|$ that contains all values in D_k in the corresponding order. The first ordering, denoted $\mathbf{u}_k$, is fixed and known to all agents A_t such that $t \in I_k$. Then if A_t and A_k are constrained, they can describe their constraint $C_{t,k}$ as a matrix $M_{t,k}$ of $|D_t|$ rows and $|D_k|$ columns, where the value in the r-th row and s-th column is

$$M_{t,k}(r,s) = C_{t,k}(\mathbf{u}_t(r), \mathbf{u}_k(s)) \,. \tag{1}$$

The second ordering, denoted $\mathbf{w}_k$, is generated at random by A_k whenever he begins a new traversal over his domain. That ordering determines the order in which A_k scans the values in his domain during that stage of the search. Agent A_k generates such an ordering each time a CPA is passed to him from the preceding agent A_{k-1}. After $\mathbf{w}_k$ is generated, A_k traverses his domain, during that stage in the search loop, in the order which $\mathbf{w}_k$ spells out: he will first check the assignment $X_k \leftarrow \mathbf{w}_k(1)$, then $X_k \leftarrow \mathbf{w}_k(2)$, and so forth until the last assignment $X_k \leftarrow \mathbf{w}_k(|D_k|)$ is checked. That ordering is kept secret from all other agents, in order to prevent agents from inferring sensitive information on the current assignments of other agents (see Theorem 3 in Section 3.5).

Internal variables. Every agent A_k maintains the following variables:

(1) sCPA_k is an array of length n that holds additive shares in the cost of the CPA. Assume that agents A_t and A_k are constrained and that $C_{t,k}$ is applicable to the CPA. Then the cost of the CPA includes, as one of its addends, the value $C_{t,k}(X_t, X_k)$. In such a case $\text{sCPA}_k(t)$ and $\text{sCPA}_t(k)$ will both store random

values in $\mathbb{Z}_S$ so that

$$C_{t,k}(X_t, X_k) = (\mathrm{sCPA}_t(k) + \mathrm{sCPA}_k(t)) \mod S\,.^3 \tag{2}$$

If, on the other hand, $C_{t,k}$ is *not* applicable to the CPA (i.e. the CPA does not include X_k or X_t or both), then $\mathrm{sCPA}_k(t) = \mathrm{sCPA}_t(k) = 0$. In view of the above, the overall cost of the CPA, at any stage of the algorithm's run, equals

$$Cost(CPA) = \sum_{k=1}^{n} \sum_{t \in I_k} \mathrm{sCPA}_k(t) \bmod S\,. \tag{3}$$

(Note that the internal vectors sCPA_k, for any $1 \le k \le n$, could actually be of length $|I_k|$, rather than n. But in order to avoid cumbersome notations we assume herein that all those vectors are of length n.)

(2) sUB_k holds an additive share in the current upper bound (the cost of the best full assignment that was discovered thus far). Each such share is random and uniformly distributed over $\mathbb{Z}_S$. At any stage of the algorithm's run,

$$UpperBound = \sum_{k=1}^{n} \mathrm{sUB}_k \bmod S\,. \tag{4}$$

(3) p_k is a pointer to a value in the ordering $\mathbf{w}_k$. The current assignment to X_k is given by $\mathbf{w}_k(p_k)$.

(4) $OptimalSetting_k$ stores the assignment to X_k in the currently best full assignment that was found thus far.

3.2. The PC-SyncBB algorithm

The PC-SyncBB algorithm is given in Algorithm 1, which we proceed to describe.

[3] Hereinafter, when we write $a = b \bmod S$ we mean that a is the residue of b modulo S.

Algorithm 1 – PC-SyncBB (executed by agent A_k) – first part

procedure init

sCPA$_k(t) \leftarrow 0$ for all $1 \leq t \leq n$
$p_k \leftarrow 0$
if $k > 1$ **do**
    sUB$_k \leftarrow 0$
else
    sUB$_k \leftarrow q_\infty$
    assign_CPA()

procedure assign_CPA

if $p_k = 0$ **do**
    Generate a new random ordering of D_k into $\mathbf{w}_k$
$p_k \leftarrow p_k + 1$
if $p_k > |D_k|$ **do**
    backtrack()
else
    $X_k \leftarrow v := \mathbf{w}_k(p_k)$
    update_shares_in_CPA(k, v)
    if $k = n$ **do**
        if compare_CPA_cost_to_upper_bound() = **true do**
            broadcast(**NEW_OPTIMUM_FOUND**)
        assign_CPA()
    else
        if compare_CPA_cost_to_upper_bound() = **false do**
            assign_CPA()
        else
            send(**CPA_MSG**) to A_{k+1}

procedure backtrack

if $k > 1$ **do**
    sCPA$_k(t) \leftarrow 0$ for all $t \in I_k^-$
    send(**ZERO_SHARE_MSG**,k) to A_t for all $t \in I_k^-$
    send(**BACKTRACK_MSG**) to A_{k-1}
else
    broadcast(**COMPLETE**)

Algorithm 1 – PC-SyncBB (executed by agent A_k) – second part

when received (**NEW_OPTIMUM_FOUND**) **do**

31: $\text{sUB}_k \leftarrow \sum_{t \in I_k} \text{sCPA}_k(t)$

32: $OptimalSetting_k \leftarrow X_k$

when received (**CPA_MSG**) **do**

33: $p_k \leftarrow 0$

34: assign_CPA()

when received (**ZERO_SHARE_MSG**,k') **do**

35: $\text{sCPA}_k(k') \leftarrow 0$

when received (**BACKTRACK_MSG**) **do**

36: assign_CPA()

when received (**COMPLETE**) **do**

37: $X_k \leftarrow OptimalSetting_k$

38: Terminate

The procedure init. Every agent A_k initializes all entries in his vector sCPA_k as well as p_k to zero (Lines 1-2). Then, every agent A_k, $k > 1$, initializes sUB_k to zero, while A_1 initializes it to q_∞ (Lines 3-6). Such settings imply that $q_\infty = \sum_{k=1}^{n} \text{sUB}_k \bmod S$, in agreement with Eq. (4) (since the initial upper bound is set to q_∞). Finally, the procedure init triggers the search by having A_1 call the procedure assign_CPA (Line 7).

The procedure assign_CPA. If this procedure is called when $p_k = 0$, it means that A_k now begins a new traversal over his domain. Hence, in such a case he generates a new random ordering, $\mathbf{w}_k$, of D_k (Lines 8-9). In order to move to the next value in $\mathbf{w}_k$, A_k increments the pointer p_k (Line 10). If p_k becomes greater than $|D_k|$ it means that the domain D_k was already fully scanned, so A_k performs the procedure backtrack (discussed below) in order to return the search torch back to the preceding agent A_{k-1} (Lines 11-12). Otherwise, A_k assigns $v := \mathbf{w}_k(p_k)$ to X_k (Line 14). Consequently, as X_k has a new value, the CPA's cost is changed, so new random shares of that cost must be computed. This is done by calling the sub-protocol update_shares_in_CPA(k, v) (Line 15), which recomputes $\mathrm{sCPA}_k(t)$ and $\mathrm{sCPA}_t(k)$, for all $t \in I_k^-$, so that the right-hand side of Eq. (3) equals the new CPA's cost. (We discuss that sub-protocol in Section 3.3.)

We now separate the discussion according to the index k of the operating agent. If $k = n$, then a new full assignment is reached. It is needed to compare its cost, which equals $\sum_{k=1}^{n} \sum_{t \in I_k} \mathrm{sCPA}_k(t) \bmod S$, Eq. (3), to the current upper bound, $\sum_{k=1}^{n} \mathrm{sUB}_k \bmod S$, (Eq. (4)). This comparison must be done in a secure manner. To that end, A_n invokes compare_CPA_cost_to_upper_bound (Line 17), a secure multiparty sub-protocol that we discuss in Section 3.4. It returns **true** if the cost of the current full assignment is lower than the upper bound, namely, if

$$\sum_{k=1}^{n} \sum_{t \in I_k} \mathrm{sCPA}_k(t) \bmod S < \sum_{k=1}^{n} \mathrm{sUB}_k \bmod S\,, \tag{5}$$

and **false** otherwise. If the current full assignment does improve the upper bound, then A_n broadcasts the message **NEW_OPTIMUM_FOUND** (Line 18). Upon receiving such a message, every agent A_k stores the sum of his current shares, $\sum_{t \in I_k} \mathrm{sCPA}_k(t)$, in sUB_k and he also stores the current assignment of X_k in $\mathit{OptimalSetting}_k$ (Lines 31-32). Finally, whether the current full assignment is a new optimum or not, A_n calls the procedure assign_CPA again in order to test the next value in his domain (Line 19).

If $k < n$, the agents examine the possibility to prune the search space: they first check whether the CPA's cost is already greater than or equal to the upper bound, by invoking compare_CPA_cost_to_upper_bound (Line 21). If it returns **false** then Eq. (5) does not hold, i.e., the cost of the CPA is already greater than or equal to the upper bound. In such a case there is no point in pursuing the current path in the search space, so A_k calls the procedure assign_CPA again in order to test the next value in his domain (Line 22). Otherwise, A_k passes the torch onward to A_{k+1} (by sending him the message **CPA_MSG** in Line 24) in order to continue the search over CPAs with the currently selected assignments to $X_1, \ldots, X_k$. When A_{k+1} receives the message **CPA_MSG**, he zeroes the pointer p_{k+1} to his domain D_{k+1}, in order to start traversing all values in D_{k+1}

as possible extensions to the current k-CPA, and then he calls the procedure assign_CPA (Lines 33-34).

The procedure backtrack. When agent A_k, $k > 1$, executes the procedure backtrack, he does two things. First, he zeroes $\mathrm{sCPA}_k(t)$ for all $t \in I_k^-$ (Line 26) and sends a **ZERO_SHARE_MSG** message, with his index k, to all agents that precede him and are constrained with him (Line 27). Any such agent, upon receiving the **ZERO_SHARE_MSG** message, zeroes the relevant share in his own array (Line 35). As a result of the above two actions, Eq. (3) still holds for the reduced CPA that is obtained after this backtracking. Afterwards, A_k sends a **BACKTRACK_MSG** message to A_{k-1} (Line 28). When the latter receives that message, he calls assign_CPA in order to change the assignment of his variable to the next value in his domain and proceed the search with the new modified CPA (Line 36).

When A_1 performs backtrack, it means that he completed a traversal of D_1, and, consequently, the entire search space $(D_1 \times \cdots \times D_n)$ was scanned. Therefore, the algorithm terminates with the last optimum found being the global optimum. In such a case A_1 broadcasts the message **COMPLETE** (Line 30). When receiving such a message, every agent A_k assigns to his variable X_k the value $OptimalSetting_k$ (which was his assignment in the last optimal solution that was found) and then he terminates (Lines 37-38).

3.3. The sub-protocol update_shares_in_CPA

This subsection is organized as follows. In Section 3.3.1 we introduce the Paillier cipher, which is the main cryptographic tool that we use in the sub-protocol update_shares_in_CPA. In Section 3.3.2 we describe the initial computations that each agent has to perform in the outset of the protocol, in preparation for the sub-protocol update_shares_in_CPA, which is described in Section 3.3.3.

3.3.1. On probabilistic homomorphic encryption

A cipher is called public-key (or asymmetric) if its encryption function $\mathcal{E}(\cdot)$ of a plaintext depends on one key, K_e, which is publicly known, while the corresponding decryption function $\mathcal{E}^{-1}(\cdot)$ of a ciphertext depends on a private key, K_d, that is known only to the owner of the cipher, and K_d's derivation from K_e is computationally hard.

A cipher is called (additively) *homomorphic* if for every two plaintexts, m_1 and m_2, $\mathcal{E}(m_1 + m_2) = \mathcal{E}(m_1) \cdot \mathcal{E}(m_2)$. When the encryption function is randomized (in the sense that $\mathcal{E}(m)$ depends on m as well as on a random string), $\mathcal{E}$ is called *probabilistic*. Hence, a probabilistic encryption function is a one-to-many mapping (every plaintext m has many encryptions $m' = \mathcal{E}(m)$), while the corresponding decryption function is a many-to-one mapping (all possible encryptions m' of the same plaintext m are mapped by $\mathcal{E}^{-1}(\cdot)$ to the same m).

Using probabilistic encryption is essential when the underlying domain of plaintexts is sufficiently small to allow exhaustive search. For example, in our implementation, as we describe in Sections 3.3.2 and 3.3.3 below, the plaintexts are either 0 or 1. Hence, in order to securely hide them, a probabilistic encryption is in order.

The semantically secure Paillier cipher [28] is a public-key cipher that is both homomorphic and probabilistic. Its plaintext domain is $\mathbb{Z}_\nu$, for a modulus ν which is the product of two large primes. Its ciphertext domain is $\mathbb{Z}^*_{\nu^2}$. The reader is referred to [28] for a full description of this cipher.

3.3.2. Initial computations

Before starting PC-SyncBB, each of the agents A_k, $k < n$, creates a key pair in a Paillier cipher. Specifically, A_k generates a Paillier modulus $\nu_k > S$ and the proceeds to generate a key pair. Letting $\mathcal{E}_k$ denote the encryption function in A_k's cipher, then $\mathcal{E}_k$ is a function from $\mathbb{Z}_{\nu_k}$ to $\mathbb{Z}^*_{\nu_k^2}$. It is additively homomorphic, in the sense that for every two plaintexts x and y, $\mathcal{E}_k(x + y) = \mathcal{E}_k(x) \cdot \mathcal{E}_k(y)$, where addition is modulo ν_k and multiplication is modulo ν_k^2.

After doing so, A_k sends the corresponding modulus ν_k and public encryption function $\mathcal{E}_k$ to A_t for all $t \in I_k^+$.

After creating $\mathcal{E}_k$, A_k computes a vector $\mathbf{z}_k^1$ of length $|D_k|$ where $\mathbf{z}_k^1(1) = \mathcal{E}_k(1)$ and $\mathbf{z}_k^1(i) = \mathcal{E}_k(0)$ for all $2 \le i \le |D_k|$. It is important to compute the latter $|D_k| - 1$ encryptions with $|D_k| - 1$ independently selected random strings. Then, A_k defines the vectors $\mathbf{z}_k^i = CRS\left(\mathbf{z}_k^{i-1}\right)$, for $2 \le i \le |D_k|$, where $CRS(\cdot)$ is a circular right-shift by one position of the vector entries. Hence, $\mathbf{z}_k^i$ encrypts the vector $(0,\ldots,0,1,0,\ldots,0)$ where the 1 appears in the i-th entry, $1 \le i \le |D_k|$. Given the manner in which those vectors were computed and the probabilistic and semantic security properties of the Paillier cipher, a polynomially-bounded adversary who gets any random sequence of those vectors (i.e. $\mathbf{z}_k^{i_1}, \mathbf{z}_k^{i_2}, \ldots$) will not be able to distinguish between the $\mathcal{E}_k(1)$ and the $\mathcal{E}_k(0)$ entries in them (with a non-negligible probability of success).

3.3.3. The sub-protocol

We are now ready to describe the sub-protocol update_shares_in_CPA (Algorithm 2). It is triggered by A_k whenever he assigns a new value v to his variable, X_k. When that happens, it is needed to update the shares of all agents $A_1, \ldots, A_k$ so that the validity of Eq. (3) is maintained. The shares that should be modified in wake of such an assignment are $\text{sCPA}_k(t)$ and $\text{sCPA}_t(k)$ for all $t \in I_k^-$. Those shares will be modified so that, in view of Eq. (2), the sum of $\text{sCPA}_k(t)$ and $\text{sCPA}_t(k)$, for any fixed $t \in I_k^-$, will equal $C_{t,k}(X_t, X_k)$ for the current assignments of X_t and X_k (X_k's assignment equals v, and it is passed to the sub-protocol as an input).

Assume that $t \in I_k^-$. Then the contribution of the pair X_t and X_k to the CPA is $M_{t,k}(r,s)$, where $\mathbf{u}_t(r) = X_t$ and $\mathbf{u}_k(s) = X_k$ (see Eq. (1)). Recall that A_t does not know s while A_k does not know r. In order to compute the new respective shares, $\text{sCPA}_k(t)$ and $\text{sCPA}_t(k)$, so that Eq. (2) holds, these two agents perform the following computation.

When A_t performed last time the procedure assign_CPA and set there the current assignment to X_t, he called update_shares_in_CPA (Algorithm 2), see Line 15 in PC-SyncBB. In Line 8 of Algorithm 2 he sent to all agents in I_t^+ the vector $\mathbf{z}_t^j$ which encodes his assignment at that point in time. Going back to the present, when A_k executes update_shares_in_CPA he holds a vector $\mathbf{z}_t$ that he received from A_t, for every $t \in I_k^-$. That vector equals $\mathbf{z}_t^r$, where r is the index in $\mathbf{u}_t$ in which the current assignment to X_t is stored. Even though A_k cannot infer from $\mathbf{z}_t$ the current value of X_t, he can still correctly update his shares vis-a-vis A_t. To that end, A_k computes

$$y_t := \prod_{i=1}^{|D_t|} \mathbf{z}_t(i)^{[(M_{t,k}(i,s)-\rho) \bmod S]}, \tag{6}$$

where s is the index of the entry in $\mathbf{u}_k$ that holds v – the current assignment to X_k, and ρ is a value that A_k selected uniformly at random (independently for each A_t) from $\mathbb{Z}_S$ (Lines 2-3 of Algorithm 2). The key observation here is the equality in the following lemma.

Lemma 1. *The homomorphism of $\mathcal{E}_t$ implies that $y_t = \mathcal{E}_t\left([(M_{t,k}(r,s) - \rho) \bmod S]\right)$.*

Proof. Recall that the vector $\mathbf{z}_t$ is an $\mathcal{E}_k$-encryption of the vector $\mathbf{e}_r = (0, \ldots, 0, 1, 0, \ldots, 0)$, where the value 1 is stored in the r-th component, and r is the index such that $\mathbf{u}_t(r)$ is the current assignment to X_t. Hence, by Eq. (6),

$$y_t = \prod_{i=1}^{|D_t|} \mathcal{E}_k(\mathbf{e}_r(i))^{[(M_{t,k}(i,s)-\rho) \bmod S]}.$$

Since $\mathcal{E}_k$ is homomorphic, it follows that for any $x \in \mathbb{Z}_{\nu_k}$ and integer m, $\mathcal{E}_k(x)^m = \mathcal{E}_k(x \cdot m)$. Hence,

$$y_t = \prod_{i=1}^{|D_t|} \mathcal{E}_k(\mathbf{e}_r(i) \cdot [(M_{t,k}(i,s) - \rho) \bmod S]).$$

Using the homomorphism once again, we infer that

$$y_t = \mathcal{E}_k\left(\sum_{i=1}^{|D_t|} \mathbf{e}_r(i) \cdot [(M_{t,k}(i,s) - \rho) \bmod S]\right).$$

Finally, since $\mathbf{e}_r(i) = 1$ when $i = r$ and $\mathbf{e}_r(i) = 0$ otherwise, we conclude that

$$y_t = \mathcal{E}_k([(M_{t,k}(r,s) - \rho) \bmod S]). \quad \square$$

Next (Algorithm 2, Line 4), A_k sends y_t to A_t who decrypts it and stores it in $\mathrm{sCPA}_t(k)$. In view of Lemma 1, A_t obtains $\mathrm{sCPA}_t(k) = (M_{t,k}(r,s) - \rho) \bmod S$ whereas A_k sets $\mathrm{sCPA}_k(t) = \rho$ (Algorithm 2, Lines 5-6). Those two uniformly random shares satisfy $M_{t,k}(r,s) = (\mathrm{sCPA}_t(k) + \mathrm{sCPA}_k(t)) \bmod S$, which fulfils the required equality in Eq. (2).

The above described updates are carried out by A_k and A_t for all $t \in I_k^-$. After completing all those updates, the updated shares satisfy Eq. (3).

Algorithm 2 – The sub-protocol update_shares_in_CPA

---

when received k, the index of the agent A_k that invokes the procedure, and v, A_k's current assignment

1: **for all** $t \in I_k^-$ **do**
2: A_k selects uniformly at random $\rho \in \mathbb{Z}_S$
3: A_k computes y_t as given in Eq. (6), where $\mathbf{z}_t$ is the vector that A_k received from A_t in the last time
4: A_k sends the computed y_t to A_t
5: A_t sets $\mathrm{sCPA}_t(k) \leftarrow \mathcal{E}_t^{-1}(y_t)$
6: A_k sets $\mathrm{sCPA}_k(t) \leftarrow \rho$
7: **if** $k < n$ **do**
8: A_k sends to all A_t where $t \in I_k^+$ the vector $\mathbf{z}_k^j$ where j is the index for which $\mathbf{u}_k(j) = v$

Example. Suppose that the torch is passed to agent A_k, and let $t \in I_k^-$. Suppose that A_k's *ordered* domain is $D_k = (10, 20, 30)$ while A_t's is $D_t = (40, 50, 60, 70)$. Assume that A_t's current assignment is $X_t = 50$, namely, the value in D_t that is identified by the assignment index $r = 2$, while A_k's is $X_k = 30$, i.e., its assignment index is $s = 3$. At this point, A_k already holds $\mathbf{z}_t = (\mathcal{E}_t(0, \mathsf{rnd}_1), \mathcal{E}_t(1, \mathsf{rnd}_2), \mathcal{E}_t(0, \mathsf{rnd}_3), \mathcal{E}_t(0, \mathsf{rnd}_4))$ — the vector that he received from A_t after the latter had set his current assignment. (Recall that the encryption function $\mathcal{E}_t$ is a probabilistic one, in the sense that it depends not only on the plaintext, being 0 or 1 in our case, but also on random independent values that we mark here by rnd_i.) Now, A_k chooses a random value $\rho \in \mathbb{Z}_S$ and locally computes

$$y_t := \prod_{i=1}^{4} \mathbf{z}_t(i)^{[(M_{t,k}(i,3)-\rho) \bmod S]} = \mathcal{E}_t(0, \mathsf{rnd}_1)^{[(M_{t,k}(1,3)-\rho) \bmod S]} \cdot \mathcal{E}_t(1, \mathsf{rnd}_2)^{[(M_{t,k}(2,3)-\rho) \bmod S]} \cdot \mathcal{E}_t(0, \mathsf{rnd}_3)^{[(M_{t,k}(3,3)-\rho) \bmod S]} \cdot \mathcal{E}_t(0, \mathsf{rnd}_4)^{[(M_{t,k}(4,3)-\rho) \bmod S]} .$$

A_k sends that value to A_t, who proceeds to apply on it the decryption function $\mathcal{E}_t^{-1}$. By Lemma 1, the value that A_t obtains after decryption is $(M_{t,k}(2,3) - \rho) \bmod S$, which he sets as his new share $\mathrm{sCPA}_t(k)$. Note that as that value incorporates the random and secret addend ρ, then A_t can learn from it no information on $M_{t,k}(2,3)$, and hence remains totally oblivious to A_k's current assignment index value $s = 3$. A_k, on the other hand, sets $sCPA_k(t) = \rho$. A_k too remains oblivious of A_t's current assignment since that assignment was conveyed to him only through $\mathcal{E}_t$-encrypted values. Mission is thus complete: A_t and A_k now hold two new random shares whose sum equals the cost relating to their current assignments, i.e., $sCPA_t(k) + sCPA_k(t) = M_{t,k}(2,3) \bmod S$.

3.4. The sub-protocol compare_CPA_cost_to_upper_bound

The sub-protocol compare_CPA_cost_to_upper_bound verifies the inequality in Eq. (5). Agent A_k, $1 \le k \le n$, holds two integers modulo S: $a_k := \sum_{t \in I_k} \mathrm{sCPA}_k(t) \bmod S$ and $b_k := \mathrm{sUB}_k \bmod S$. The goal is to determine whether the integer $\tilde{a} := \sum_{k=1}^{n} a_k \bmod S$ is smaller than the integer $\tilde{b} := \sum_{k=1}^{n} b_k \bmod S$ or not.

That verification is carried out by a secure multiparty computation (MPC hereinafter). In Section 3.4.1 we provide a prelude to the topic of MPC. Then, in Sections 3.4.2 and 3.4.3 we describe two possible MPC solutions for the verification of the inequality in Eq. (5).

3.4.1. A prelude to secure multiparty computation

The MPC protocols that we will use to privately verify the inequality in Eq. (5) are secure under two assumptions: all agents are semi-honest, and there exists among them an honest majority. We proceed to explain those assumptions.

Semi-honest and malicious agents. Like in all prior art on privacy-preserving DCOP algorithms (which we review in the Introduction), we assume that the agents are semi-honest; namely, they follow the prescribed protocol but try to glean more information than allowed from the protocol transcript.

Another type of parties[4] that is considered in the MPC literature is the malicious type. Malicious parties may deviate from the prescribed protocol and may also provide wrong inputs, in attempt to sabotage the computation and, possibly, use the resulting messages that they receive from other parties in order to infer sensitive information on other parties' inputs. MPC protocols that are designed to be immune to malicious parties are usually significantly costlier than the corresponding MPC protocols for semi-honest parties. In addition, the presence of a malicious party introduces a new severe problem, known as *the input consistency problem*; namely, the need to verify that each party uses all the time the same input, and does not try to inject into different stages of the computation different and incorrect inputs. General solutions for the input consistency problem are quite expensive and, hence, a tailored input consistency mechanism should be devised in our context. In view of all of the above, the case of malicious parties/agents is one that introduces new and significant challenges, and, consequently, we defer the study of such a case to a future work.

Honest majority. In contrast to prior art, we assume that some of the agents may collude in order to combine their inputs and messages received during the execution of the protocol, for the purpose of extracting private information on other agents. However, we assume that the number of colluding agents is less than half of the agents. (Such an assumption is referred to in the MPC literature as the *honest majority* assumption).

[4]We note that in the cryptographic literature on MPC it is customary to speak of *parties*; in the context of DCOPs, one speaks of *agents*. We shall use those terms here interchangeably.

An MPC protocol allows the agents $A_1, \ldots, A_n$ to compute any function f over private inputs that they hold, $x_1, \ldots, x_n$, so that at the end of the protocol everyone learns $f(x_1, \ldots, x_n)$ but nothing else beyond what every agent may naturally infer from the final output and his own input.[5] In our context, the private input of agent A_k is $x_k = (a_k, b_k)$, where $a_k = \sum_{t \in I_k} \mathrm{sCPA}_k(t)$ and $b_k = \mathrm{sUB}_k$. Hereinafter, all values and all additions are modulo S (we omit the mod S notation for convenience). The function f that needs to be securely evaluated is

$$f\left((a_1, b_1), \ldots, (a_n, b_n)\right) = \left\{ \tilde{a} := \sum_{k=1}^{n} a_k \overset{?}{<} \tilde{b} := \sum_{k=1}^{n} b_k \right\}, \tag{7}$$

where, hereinafter, $x \overset{?}{<} y$ denotes a bit that equals 1 if $x < y$ and 0 otherwise.

MPC protocols require the function f to be represented by a circuit C such that for every set of inputs, $x_1, \ldots, x_n$, the output of the circuit, $C(x_1, \ldots, x_n)$, equals $f(x_1, \ldots, x_n)$. A circuit representation of a function f is essentially a directed acyclic graph (DAG), $G = (V, E)$, with the following properties. The graph has a leaf node (i.e., a node with indegree zero) for every input of f, and a root node (i.e., a node with outdegree zero) for every output of f. The former nodes are called *input gates*, while the latter ones are called *output gates*. (In our case, the function f in Eq. (7) has a single output.) In addition, the graph may have multiple internal nodes (ones with positive indegrees and outdegrees) that are called *operation gates*, or simply, gates.

For a gate g, we denote by Succ_g the set $\{g' \mid (g, g') \in E\}$, i.e., all gates g' such that there exists a directed edge from g to g'. Similarly, we denote by Pred_g the set $\{g' \mid (g', g) \in E\}$, i.e., all gates g' such that there exists a directed edge from g' to g. We restrict our attention to circuits in which for each gate g, $|\mathsf{Pred}_g| = 2$ while $|\mathsf{Succ}_g|$ is unbounded. Namely, each gate has exactly two predecessor gates, $\mathsf{Pred}_g := \{g_\ell, g_r\}$, to which we refer as the left/right predecessor gates of g. Letting α_ℓ and α_r denote the output values of

[5] For example, if f outputs the median among $x_1, \ldots, x_n$, then every agent may learn that there are at least $\frac{n}{2}$ values greater (or smaller) than his own.

g_ℓ and g_r, respectively, then the output of gate g is a simple function of those two values, $g(\alpha_\ell, \alpha_r)$. (We slightly abuse notation and use g for both a gate and its function.)

The private values $x_1, \ldots, x_n$ determine the input values to all of the circuit's input gates. Then, the following process is performed repeatedly: for each operation gate g, once both g_ℓ and g_r are assigned values, say α_ℓ and α_r, respectively, the gate g is assigned the value $g(\alpha_\ell, \alpha_r)$. This process is repeated until all output gates are assigned. The output of $C(x_1, \ldots, x_n)$ is defined to be the values assigned to all output gates of C at the end of an evaluation process.

Two main types of circuits are discussed in the MPC literature: an *arithmetic* circuit, meaning that the values assigned to gates are from an arbitrary finite field $\mathbb{F}$, and the operation gates are either the addition or the multiplication functions (over two operands); and a *Boolean* circuit, meaning that the values assigned to each gate are from $\{0, 1\}$, and the operation gates are either the logical `XOR` or `AND` functions. It is well known that both types of circuits can express any function (i.e., they are Turing-complete). However, some functions are 'better' represented by an arithmetic circuit, while others are 'better' represented by a Boolean circuit. In the context of MPC, the suitability of a circuit to the relevant function is determined by the complexity of the circuit, denoted $|C|$, which is commensurate with the number of multiplication or `AND` gates in the circuit, and the circuit's depth (the length of the longest path from an input gate to an output gate), denoted $d(C)$.

Protocols for secure computation of arithmetic circuits substantially differ from protocols for secure computation of Boolean circuits. While the former protocols heavily rely on secret sharing [29], the latter ones are based on a cryptographic primitive called 'garbled circuits' [30]. While secret-sharing-based protocols require parties to perform fast efficient arithmetic operations, garbled-circuit-based protocols require costly cryptographic operations like computing pseudorandom functions. Additionally, the communication complexity of each party in secret-sharing-based protocols (e.g., [31, 32]) is $O(|C| \cdot \ell)$ where $\ell = \lceil \log S \rceil$ (i.e., the length of the binary representation of each value); in particular, it is independent of the number of parties n. However, the communication complexity of each party in garbled-circuit-based protocols is $O(n^2 \cdot |C| \cdot \kappa)$, where κ is a protocol's security parameter ($\kappa = 128$ is standard); in particular, that complexity *does* depend on n. On the other hand, the advantage of garbled-circuit-based protocols is that they are *constant-round*, i.e. the parties have to sequentially interact with each other only a constant number of times; in particular, this constant does not depend on the circuit's structure. In contrast, in secret-sharing-based protocols the number of rounds is proportional to the depth of the circuit; hence, a secure evaluation of deeper circuits takes more time.

Jumping ahead, we found out that secure protocols for arithmetic circuits are very efficient in our context. Thus, we use the generic secret-sharing-based protocol of Damgård and Nielsen [31], enhanced by a recent work by Chida et al. [32] that demonstrates some performance optimizations. We plug into that protocol a circuit representation of the function f (Eq. (7)). Since the values $\tilde{a}$ and $\tilde{b}$ can be computed by addition gates only, the dominant part of the function f (i.e., the part that involves multiplication gates) is the comparison $\tilde{a} \stackrel{?}{<} \tilde{b}$. For that purpose, we use the comparison circuit representation by Nishide and Ohta [33]. A more detailed discussion is given later on.

We describe herein two MPC methods for computing Eq. (7). We begin with the method that we applied in the preliminary version of this study [25], that uses a Boolean circuit (Section 3.4.2). Then, we describe an alternative method that uses an arithmetic circuit (Section 3.4.3). We compare the two methods, theoretically and experimentally, in Section 5.

3.4.2. Secure computation of Eq. (7) using a garbled-circuit-based protocol

In [25] we used a Boolean circuit. In order to enable a smaller representation of the Boolean circuit, we took there S to be the smallest power of 2 greater than $2q_\infty$. With such a setting of S, the computation of Eq. (7) is carried out as follows. Instead of holding two inputs, a_k and b_k, each A_k needs only to hold one input, $d_k := (b_k - a_k) \bmod S$. Then, the Boolean circuit that was used in

[25] evaluates the function

$$f'(d_1, \ldots, d_n) := \mathsf{msb}\left(\sum_{k=1}^{n} d_k \bmod S\right) . \tag{8}$$

The key observation in [25] is that if S is selected so that $S > 2q_\infty$, then $f'(d_1, \ldots, d_n)$ in Eq. (8) equals $f((a_1, b_1), \ldots, (a_n, b_n))$ in Eq. (7). Thus, the circuit only sums up n values modulo S and then outputs the most significant bit in the sum. Since f' outputs the most significant bit of $\sum_{k=1}^{n} d_k$, a Boolean circuit is the more fitting choice for evaluating f'.

Hence, we used in [25] a garbled-circuit-based protocol (specifically, the Ben-Efraim-Omri protocol [34]). We note that garbled-circuit-based protocols typically consist of two phases: a garbling (or offline) phase in which the interacting agents jointly generate an encrypted version of the circuit C, and an evaluation (or online) phase in which the circuit's output on the given inputs is computed. In the course of Algorithm 1, the agents would need to compute f' (Eq. (8)) several times, each time on a different set of inputs. While the Boolean circuit C that computes f' is fixed, in each such computation the agents need to use an independent garbled version of C. They could produce upfront many such garbled versions of C (by running the offline phase of the Ben-Efraim-Omri protocol). But in order to compute its output on each set of inputs, they would need to run the online phase of the Ben-Efraim-Omri protocol only when those inputs are known. (We refer the reader to [25] for a more detailed description of the Ben-Efraim-Omri protocol for computing our comparison function f', Eq. (8).)

Garbling scheme. In order to provide the reader a taste of what a garbled circuit looks like, we consider the simpler case of only two parties: a *garbler*, who is given a description of a Boolean circuit C and produces the garbled version of C, called *garbled circuit* and denoted $\tilde{C}$; and an *evaluator*, who is given the garbled circuit $\tilde{C}$ and a single key for each input wire (a wire is either an edge that connects two operation gates or an edge connected to an input/output gate only), then, using the evaluation procedure of the garbling scheme he can obtain the corresponding key of the output wire. The keys obtained in the evaluation process hide the actual values that are carried through the wires. To enable the evaluator to learn the actual output of C (and not only the key that hides it), the garbler supplies a decoding map from a key to an actual bit. A description of a simple garbled-circuit-based protocol follows:

1. The garbler and evaluator agree on a Boolean circuit, C, that they wish to compute securely.
2. *Garbler.*
   (a) For each wire, w, in the circuit, choose two random keys $k_{w,0}, k_{w,1} \leftarrow \{0,1\}^{\kappa}$, where κ is the security parameter of the scheme.
   (b) For each gate, g, in the circuit, denote the function of that gate by $g : \{0,1\}^2 \to \{0,1\}$.
   (c) For each gate, g, produce the garbled-gate, $\tilde{g}$, as follows: Let α, β, and γ be the left/right input wires and output wire of g, respectively. The garbled gate $\tilde{g}$ is a quadruple $(\tilde{g}_{00}, \tilde{g}_{01}, \tilde{g}_{10}, \tilde{g}_{11})$ where

$$\begin{aligned}\tilde{g}_{00} &= E_{k_{\alpha,0},k_{\beta,0}}(k_{\gamma,g(0,0)})\\ \tilde{g}_{01} &= E_{k_{\alpha,0},k_{\beta,1}}(k_{\gamma,g(0,1)})\\ \tilde{g}_{10} &= E_{k_{\alpha,1},k_{\beta,0}}(k_{\gamma,g(1,0)})\\ \tilde{g}_{11} &= E_{k_{\alpha,1},k_{\beta,1}}(k_{\gamma,g(1,1)})\end{aligned}$$

   such that E is a double encryption scheme (i.e., a scheme that applies two consecutive encryptions, with two independent keys) with a property that allows one to know, for a given ciphertext c and two keys k_1, k_2, whether c is the output of $E_{k_1,k_2}(m)$ for some m or not.
   (d) For each gate, g, randomly permute the quadruple $\tilde{g}$.
   (e) The garbled circuit $\tilde{C}$ is the collection of all garbled gates. That is, $\tilde{C} = \{\tilde{g} \mid g \in C\}$.
   (f) For each output wire, w, with keys $k_{w,0}, k_{w,1}$, set the decoding map to $\mathsf{decode}_w := \{k_{w,0} \to 0, k_{w,1} \to 1\}$.
   (g) Send $\tilde{C}$ and decode_w for all circuit output wires, w, to the evaluator.

3. *Obtaining inputs.* The evaluator needs to obtain a single key (out of the two possible) for each input wire of C. There are wires that are associated with the garbler's input bits and wires that are associated with the evaluator's inputs bits:
   (a) For each input wire, w, that is associated with the garbler's input bit, x, the garbler sends the key $k_{w,x}$ to the evaluator.
   (b) For each input wire, w, that is associated with the evaluator's input bit, x, the garbler and the evaluator execute an *oblivious transfer* (OT) protocol. OT allows the evaluator to obtain $k_{w,x} \in \{k_{w,0}, k_{w,1}\}$, so that the garbler remains oblivious of the value of the selection bit x and the evaluator remains oblivious of the non-obtained key $k_{w,1-x}$.
4. *Evaluator.*
   (a) Given the garbled circuit $\tilde{C} = \{\tilde{g} \mid g \in C\}$ and a single key for each input wire, the evaluator proceeds as follows: Traverse the circuit in a topological order from the input wire to the output wires; then, for each gate g on the way, let α and β be g's input wires and γ be its output wire. The evaluator has the keys k_α and k_β. He computes $k'_{00} = E^{-1}(\tilde{g}_{00})$, $k'_{01} = E^{-1}(\tilde{g}_{01})$, $k'_{10} = E^{-1}(\tilde{g}_{10})$, and $k'_{11} = E^{-1}(\tilde{g}_{11})$. The special property of the encryption scheme implies that only one of $k'_{00}, k'_{01}, k'_{10}, k'_{11}$ is valid, so the evaluator concludes that this is the key for the output wire γ, namely, this is k_γ.
   (b) For each circuit output wire, w, compute $b_w = \mathsf{decode}_w(k_w)$, where k_w is the key obtained by the evaluation process above. The evaluator outputs b_w.

The security of the garbling scheme follows from the fact that it is not possible for the evaluator to obtain two keys for the same wire, as guaranteed by the OT sub-protocol. In addition, when decrypting (i.e., when computing E^{-1}), the entry for which decryption succeeds tells nothing about the actual value carried through the wire, because the quadruple $\tilde{g}$ is arranged in a random order.

As mentioned above, only Steps 3-4 above require the parties' inputs; Step 2 can be performed in an offline (pre-processing) phase.

We note that the above example assumes only two parties. In our context, though, we use the multiparty version of Ben-Efraim and Omri. In that version, all parties take both roles of garblers and evaluators.

The structure of the Boolean circuit. Let d_k denote the ℓ-bit input of agent A_k. Let d_k^i, $0 \leq i \leq \ell - 1$, denote the bits in the binary representation of d_k. Then the circuit takes as input ℓn bits: d_k^i, $1 \leq k \leq n$, $0 \leq i \leq \ell - 1$. The circuit needs to compute $\sum_{k=1}^{n} d_k \bmod S$. This can be done by implementing $n-1$ sub-circuits, each of which adds two ℓ-bit integers. Note that since $S = 2^\ell$ and we are interested in the sum modulo S, we may ignore "spill-over" bits (corresponding to 2^ℓ) that may occur when adding the two ℓ-bit addends.

Let the two ℓ-bit addends be $x = \sum_{i=0}^{\ell-1} x_i 2^i$ and $y = \sum_{i=0}^{\ell-1} y_i 2^i$, and let $z = \sum_{i=0}^{\ell-1} z_i 2^i$ be their sum modulo S. Let $t^1, \ldots, t^{\ell-1}$ be $\ell - 1$ temporary Boolean variables. We have $z^0 = x^0 \oplus y^0$ and $t^1 = x^0 \wedge y^0$. Then, for $i = 1, \ldots, \ell - 2$ we have $z^i = x^i \oplus y^i \oplus t^i$ and $t^{i+1} = (x^i \wedge y^i) \vee (x^i \wedge t^i) \vee (y^i \wedge t^i)$. Finally, we compute $z^{\ell-1} = x^{\ell-1} \oplus y^{\ell-1} \oplus t^{\ell-1}$. In view of the above, the complexity of the sub-circuit that computes z from x and y is $1 + (\ell - 2) \cdot 5$ `AND` gates (since an `OR` gate can be implemented using a single `AND` gate). Overall, the entire circuit that adds the n agents' ℓ-bit inputs consists of $(n-1)\big(1 + (\ell - 2) \cdot 5\big)$ `AND` gates and it has a depth of $(\ell - 1) \log n$.

3.4.3. Secure computation of Eq. (7) using a secret-sharing-based protocol

In the present study we were pleasantly surprised to find out that even though our desired function, Eq. (7), can be represented by Eq. (8), which strongly suggests using a Boolean circuit, an arithmetic circuit representation turns out to be much more efficient. This counter-intuitive finding is due to the following reasons:

(1) As described at the end of Section 3.4.2, a Boolean representation of the function f' in Eq. (8) has to include $n - 1$ sub-circuits of ℓ-bits full-adder ($\ell = \log S$), where a full-adder sub-circuit has roughly 5ℓ `AND` gates. In contrast,

an arithmetic circuit that computes f directly (Eq. (7)) first adds up all a_k's and all b_k's in order to obtain $\tilde{a}$ and $\tilde{b}$, and then performs the comparison. As mentioned above, the complexity of an arithmetic circuit is determined only by the number of *multiplication* gates. Since obtaining $\tilde{a}$ and $\tilde{b}$ requires only *addition* gates, which are essentially 'for free', the complexity of the circuit C equals the complexity of a circuit for a *single comparison*. Moreover, $|C|$ no longer depends on the number of parties n.

(2) The main benefit of using a Boolean circuit representation and hence a garbled-circuit-based protocol, is that it is constant-round. However, we observe that this fact is not relevant in our case, since it is possible to represent the comparison function by an arithmetic circuit with a *constant depth*. Such a circuit representation was found by Nishide and Ohta [33]. Thus, the secret-sharing-based protocol for computing that circuit is *constant-round* as well, just like a garbled circuit.

In view of the above arguments, we introduce in this paper a secure protocol for computing Eq. (7) which is based on an arithmetic circuit. To that end, we used the secret-sharing-based protocol of Damgård and Nielsen [31]. That protocol relies on the same two assumptions as ours (see the beginning of Section 3.4.1): semi-honesty of all agents, and an honest majority. Their protocol builds on Shamir's threshold secret sharing scheme [29]. Such a scheme allows a 'dealer' to 'split' a secret s that he holds into n 'shares', $s_1, \ldots, s_n$, so that any subset of up to t shares, where $t < n$ is some predetermined threshold, reveals nothing about s, whereas any subset of $t + 1$ shares enables the full reconstruction of the secret s.

Shamir threshold secret sharing. The Shamir threshold secret sharing scheme has two procedures: Share and Reconstruct, which we proceed to describe:

- $\mathsf{Share}_{t,n}(s)$. Given a secret $s \in \mathbb{F}$, the procedure samples a uniformly random polynomial $p(\cdot)$ over $\mathbb{F}$, of degree t, where the free coefficient is s. That is, $p(x) = s + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_t x^t$, where α_j, $1 \leq j \leq t$, are selected uniformly at random from $\mathbb{F}$. The procedure outputs n values – $p(1), \ldots, p(n)$ – where $s_i = p(i)$ is the share given to agent A_i, $1 \leq i \leq n$. Together, the tuple $\langle s_1, \ldots, s_n \rangle$ is called a t-sharing of s, and is denoted by $[s]_t$. It is easy to see that any selection of t shares out of $s_1, \ldots, s_n$ reveals nothing about the secret s, whereas any subset of $t+1$ or more shares fully determines s, by means of polynomial interpolation.

- $\mathsf{Reconstruct}_t(s_1, \ldots, s_n)$. The procedure is given any selection of $t+1$ shares out of $\langle s_1, \ldots, s_n \rangle$, and it then interpolates a polynomial $p(\cdot)$ of degree at most t using the given points $\{(i, s_i)\}_i$, and outputs $s = p(0)$.

MPC based on Shamir sharing. Here we describe the general methodology in Shamir-sharing-based MPC. To do this, we show how to securely compute two types of arithmetic gates: addition and multiplication.

Let t be an integer smaller than $n/2$. Herein we use the setting $t = \lfloor (n-1)/2 \rfloor$. Then the construction that we proceed to describe below will be based on Shamir's t-out-of-n secret sharing scheme.

First, we show how the parties can obtain a sharing of a random value that is unknown to anyone: Each party A_i picks a random value s^i and calls $\mathsf{Share}_{t,n}(s^i)$. By that, every party A_j obtains n shares: $s_j^1, \ldots, s_j^n$. Then, A_j sums up all shares that he received in order to obtain $s_j = \sum_i s_j^i$. It is easy to verify that s_j is a valid share of the value $s = s^1 + \ldots + s^n$, which is random (since it is a sum of random values) and unknown to anyone (since every party contributes to that sum his own random share that is known only to him). Similar to that procedure, the parties may also generate a $2t$-sharing of the same secret s, by replacing the call to $\mathsf{Share}_{t,n}(s^i)$ with a call to $\mathsf{Share}_{2t,n}(s^i)$. This becomes handy in the procedure for multiplying two secrets, as described below.

We are now ready to describe the addition and multiplication procedures:

- *Addition.* Given the sharing $[a]_t$ and $[b]_t$ (i.e., each party A_i holds two shares, a_i and b_i, in a and b, respectively), the parties wish to obtain a

t-sharing of $a + b$ without revealing anything about a or b. This can be done easily, without any interaction between the parties, since $c_i := a_i + b_i$ is a valid share for A_i in $c = a+b$. Note that addition with a constant also works in a similar manner. That is, given a t-sharing $[a]_t$ and a constant c, the set $a_1 + c, \ldots, a_n + c$ is a valid t-sharing of $a + c$, i.e., it is equivalent to $[a + c]_t$.

- Multiplication. In contrast to addition, the multiplication procedure for computing $c = a \cdot b$ requires interaction. Given the sharing $[a]_t$ and $[b]_t$ (i.e., each party A_i holds two shares, a_i and b_i, in a and b, respectively), the parties wish to obtain a t-sharing of $c = a \cdot b$ without revealing anything about a or b.

  As before, we first have each party A_i multiply his own shares to obtain $c_i = a_i \cdot b_i$. But now, notice that $c_1, \ldots, c_n$ is not a t-sharing anymore, but a $2t$-sharing; indeed, such a multiplication is equivalent to multiplying two degree-t polynomials $p(\cdot)$ and $q(\cdot)$, hiding a and b, respectively, and such a multiplication yields a new polynomial of degree $2t$. The task is, then, to reduce the degree of the secret sharing polynomial back to t. To this end, the parties produce t- and $2t$-sharings of the same random value. We shall denote this random value by r (in the context of the t-sharing) as well as by $R = r$ (in the context of the $2t$-sharing). Let us denote those sharings by $[r]_t$ and $[R]_{2t}$, respectively (where $r = R$). Now, the parties obtain a $2t$-sharing of $\tilde{c} := c + R$, by each party locally computing $\tilde{c}_i = c_i + R_i$. Then, each party A_i sends $\tilde{c}_i$ to A_1, who runs $\tilde{c} \leftarrow \mathsf{Reconstruct}_{2t}(\tilde{c}_1, \ldots, \tilde{c}_n)$ and broadcasts it to everyone. Finally, each party A_i locally computes $\hat{c}_i = \tilde{c} - r_i$. Note that the set $\hat{c}_1, \ldots, \hat{c}_n$ is a valid t-sharing of $c = a \cdot b$, since $[r]_t$ is a t-sharing of r, $\tilde{c}$ is a constant, and $\tilde{c} - r = a \cdot b + R - r = a \cdot b$.

Securely comparing $cost(CPA)$ and UB. In the context of our protocol, each agent A_k has private inputs $a_k := \sum_t \mathrm{sCPA}_k(t) \mod S$ and $b_k := \mathrm{sUB}_k$. To input those values to the computation, A_k calls $\mathsf{Share}_{t,n}(a_k)$ and $\mathsf{Share}_{t,n}(b_k)$, which results in the sharings $[a_k]_t$ and $[b_k]_t$. Then, the parties use the addition procedure described above to obtain sharings of the sum of a_k's and b_k's. That is, $[\tilde{a}]_t = [\sum_k a_k]_t$ and $[\tilde{b}]_t = [\sum_k b_k]_t$. Finally, the parties run a protocol for securely comparing two secret values $\tilde{a}$ and $\tilde{b}$. Such task is, on its own, a subject for a line of research in the MPC literature; therefore, we only describe the high level idea of the state of the art, leaving the details out of the scope of this paper.

Nishide and Ohta [33] proposed a circuit representation for comparing two shared values, $\tilde{a}$ and $\tilde{b}$, in an indirect fashion. Namely, instead of a circuit that verifies whether $\tilde{a} < \tilde{b}$ over the two secrets $\tilde{a}$ and $\tilde{b}$ directly, they designed a circuit that obtains the same result, indirectly. Their circuit first computes three comparisons between some secret and a *public value*; such circuits are much lighter (in comparison to circuits that compare two values that are both secret). Then, the circuit combines the results from these three comparisons in order to obtain the result of the desired direct comparison. Specifically, instead of the comparison $u \stackrel{?}{<} v$, with u and v being the two secrets, one can compute $u \stackrel{?}{<} v$ from w, x, y where $w := u \stackrel{?}{<} S/2$, $x := v \stackrel{?}{<} S/2$ and $y := (u - v) \bmod S \stackrel{?}{<} S/2$ by

$$u \stackrel{?}{<} v = w\bar{x} \vee \bar{w}\bar{x}\bar{y} \vee wx\bar{y}\,.$$

The equality above can be readily verified by its truth table. We stress that the intermediate values w, x, y remain secret from the agents, while only the final comparison result is revealed.

The key insight from the circuit design of Nishide and Ohta is that a circuit that directly computes a comparison of two secrets has a much higher complexity than a circuit that breaks that comparison into three comparisons between a secret and a public value, and then combines those three results (which remain hidden) in order to get the final result.

The number of multiplication gates in the arithmetic circuit C_f is given by $279 \cdot \log p + 5$ in a circuit of depth 15, when $\tilde{a}, \tilde{b} \in \mathbb{Z}_p$. Note that in the secret-sharing-based approach the communication complexity does not depend on the number of parties, since all preliminary addition operations are done locally and there is only a single invocation of a comparison between two values.

3.5. Properties of PC-SyncBB

The main properties of this algorithm are stated below.

Theorem 2. *PC-SyncBB is complete and sound.*

Proof. The completeness of PC-SyncBB follows from the exhaustive search structure. Only partial assignments whose cost reach the upper bound are not extended and therefore it is guaranteed that the algorithm finds an optimal solution. Termination also follows from the exhaustive structure of the Branch-and-Bound algorithm in which no partial assignment can be explored twice.

PC-SyncBB is sound, in the sense that it outputs a correct solution, as implied by the correctness of update_shares_in_CPA (which guarantees that Eqs. (3) and (4) are always correct) and compare_CPA_cost_to_upper_bound (which guarantees the correctness of validating Eq. (5)). □

Theorem 3. *PC-SyncBB provides constraint-, topology-, and assignment/decision-privacy. Even if any subset $\mathcal{B} \subsetneq \mathcal{A}$ of agents collude, where $|\mathcal{B}| < n/2$, they would not be able to infer information on (values or existence of) constraints between two agents outside the coalition, or on value assignments or final decisions of such agents.*

Proof. The only way in which privacy can be breached is through the data which is transmitted between agents. In the main body of PC-SyncBB (Algorithm 1) the only data which the agents transmit between themselves are command messages. Those messages convey information only with regard to the sizes of the variable domains, $|D_k|$, $1 \leq k \leq n$, but those domains are assumed to be publicly known anyway. Since the order in which each agent traverses his domain during the search is random and kept secret from all other agents (as discussed in Section 3.1), such messages do not include any information regarding the assignments, the final decisions, the constraints, or the constraint graph topology.

In addition to those command messages, information is exchanged also in the two sub-protocols. In update_shares_in_CPA, the agent A_k receives from every A_t, where $t \in I_k^-$, his vector $\mathbf{z}_t$. That vector is computed by A_t whenever he assigns a new value from his domain to X_t. As each of those computations is made independently of previous computations, and as the Paillier cipher is semantically secure, A_k cannot infer from $\mathbf{z}_t$ any information on the current assignment of A_t. Moreover, as A_t sends the same vector $\mathbf{z}_t$ to all agents A_k, $k \in I_t^+$, upon their request, no coalition, of any size, can gain additional knowledge on A_t's assignments. Another place in update_shares_in_CPA in which data is exchanged is in Line 4. There, agent A_k sends to A_t the value y_t, which includes the $\mathcal{E}_t$-encryption of $[(M_{t,k}(r,s) - \rho) \bmod S]$. Since ρ is selected uniformly at random from $\mathbb{Z}_S$, this value contains no information at all. Moreover, since A_k selects in Line 3 an independent random ρ for each A_t, also here there is no point in performing coalitions.

As for the compare_CPA_cost_to_upper_bound sub-protocol, it is secure, under the assumption of honest majority, since it implements either the Ben-Efraim-Omri protocol or the Damgård-Nielsen protocol, which were both shown to be secure under that assumption [31, 34]. □

3.5.1. On potential information leakages of the protocol

Like *all* preceding papers on privacy-preserving solution of DCOPs, our algorithm does not guarantee *perfect* privacy, as it may leak some very benign information on the constraint graph topology. While achieving perfect privacy is possible, in theory, in any multiparty computation, it is very hard to do so while maintaining practicality. Hence, in almost all studies that deal with privacy-preserving solutions of practical problems, one accepts benign information leakages. We proceed to elaborate on that matter below.

In the context of MPC, quantifying the amount of information leakage and identifying all possible scenarios in which information may leak is, in general, an exhausting task. Hence, in order to analyze the privacy guarantees of an MPC protocol that is designed to compute some functionality, the following approach is common in the MPC literature. One considers a theoretical scenario in which the parties (agents) compute the same desired functionality by delegating their private inputs to an imaginary third party T; then T performs the computation of the functionality by itself and provides the computed output to the designated party or parties. In that theoretical scenario, T is trusted by all parties to be perfectly honest and use the secret inputs that were revealed to him only for the sake of the computation. In particular, T is assumed not to reveal the secret information to any of the real parties.

The goal in designing MPC protocols is to render them secure against corrupted parties; namely, parties that attempt to use the information that they receive during the execution of the protocol in order to infer sensitive information on other parties. An MPC protocol is considered perfectly secure if any information that the corrupted parties may infer about other parties' inputs during the real protocol, is an information that they could have also inferred in the theoretical protocol that involves the imaginary trusted party. If during the real protocol the parties may infer information that would have not been revealed to them during the theoretical protocol with T, then such information is considered to be an information leakage.

Let us illustrate those concepts with a toy example. Let us assume a case where two parties, P_1 and P_2, wish to compute the average of their numbers x_1 and x_2, respectively. In a protocol involving an imaginary party, T, the two parties, P_1 and P_2, send x_1 and x_2 to T. Then, T proceeds to compute the average $m = (x_1 + x_2)/2$ and he then sends the computed m back to them. In this example, if P_1 is corrupted, then he may learn P_2's input x_2 even when T is involved, since $x_2 = 2m - x_1$. Therefore, a real cryptographic protocol that reveals x_2 to P_1 would still be considered perfectly secure.

Now, consider an extension of the above example to the case of three parties, P_1, P_2, P_3, with inputs x_1, x_2, x_3, respectively. Now, if P_1 is corrupted, it no longer learns x_2 (nor x_3) when T is involved. He only learns some relation between x_2 and x_3; specifically, he learns that $x_2 + x_3 = 3m - x_1$, where m is the computed average of the three inputs. In such a case, a real protocol that may reveal to P_1 the value of x_2, or any linear combination of x_2 and x_3 other than $x_2 + x_3$, would be considered as not perfectly secure, and such an excess information would be considered an information leakage of the protocol.

In our context, we consider an imaginary trusted party T that replaces all invocations of the sub-protocols update_shares_in_CPA and compare_CPA_cost_to_upper_bound. In the theoretical protocol that simulates update_shares_in_CPA, T waits for two parties A_t and A_k to input their current assignments to X_t and X_k, and then he returns random shares, $\text{sCPA}_t(k)$ to A_t and $\text{sCPA}_k(t)$ to A_k, so that their sum equals $C_{t,k}(X_t, X_k)$. Obviously, if only A_t is corrupted, he learns nothing about X_k (and vice versa) since each of the two shares are truly random. In the theoretical protocol that simulates compare_CPA_cost_to_upper_bound, T waits for each party A_k to input two shares: the share for the cost of the current variable assignments, that is, $a_k = \sum_{t \in I_k} \text{sCPA}_k(t)$, and the share of the minimal cost that was found so far, $b_k = \text{sUB}_k$. Note that the actual cost of the current assignment is $\sum_k a_k$ and the actual minimal cost found so far is $\sum_k b_k$. Thus, T computes $\tilde{a} = \sum_k a_k$ and $\tilde{b} = \sum_k b_k$ and outputs **true** if $\tilde{a} < \tilde{b}$ and **false** otherwise. In this case, it is guaranteed that no collusion of corrupted parties can learn the actual values $\tilde{a}$ or $\tilde{b}$, but only their order, namely, whether $\tilde{a} < \tilde{b}$, since this is what T outputs.

The crucial point to notice is that while each separate invocation of our MPC protocols update_shares_in_CPA and compare_CPA_cost_to_upper_bound is perfectly secure, the "bigger protocol" may reveal excess information, even when T is involved! This is due to the reason that the bigger protocol, PC-SyncBB, invokes the two sub-protocols update_shares_in_CPA and compare_CPA_cost_to_upper_bound many times and on related inputs, and that may allow some privacy loss.

To demonstrate such a potential privacy loss, consider the case of $n = 3$ agents, and let us assume that A_1 is corrupted. Assume further that X_1 is not constrained with either X_2 or X_3, and that $|D_1| = |D_2| = |D_3| = d$. Suppose that A_1 has an auxiliary information by which either (a) X_2 and X_3 are not constrained or, (b) X_2 and X_3 are constrained and all entries in the cost matrix $M_{2,3}$ are distinct. A_1 may learn which of the two possible cases,

(a) or (b), holds (with high probability) by observing the number of times he receives the message **NEW_OPTIMUM_FOUND** during the execution of the protocol. The maximum possible number of times is d^2. However, if X_2 and X_3 are not constrained, then the upper bound will drop from q_∞ to zero immediately, while if they are constrained, the lower bound will be updated $d^2/2$ times, on average. Hence, if case (a) holds, A_1 will observe exactly one **NEW_OPTIMUM_FOUND** message, while if case (b) holds he will observe that message on average $d^2/2$ times. So the larger d is (the domain size), the easier it is for A_1 to distinguish between the two possible cases.

Clearly, the above scenario is highly contrived, but its sole purpose is to illustrate the manners in which corrupted parties may, in theory, extract undesired excess information from their view during the execution of the PC-SyncBB protocol. It seems that in more realistic scenarios (where each agent is constrained with at least one other agent, n is larger, the domains are of different sizes, and the auxiliary information available to the corrupted agent and the desired inference task are of a more realistic nature), corrupted agents will be unable to extract meaningful information from their view during the protocol.

Can we achieve perfect privacy?

The answer is yes, but the implication is that no pruning of the search tree would be possible, since any such branch not visited leaks some information about the relation of the other agents' private assignments and their constraints. Alas, such a protocol, that would be indeed perfectly secure, would no longer be a secure implementation of the SyncBB algorithm; it would be a secure implementation of an exhaustive search, which is clearly an impractical approach for solving an NP-hard problem.

4. Two variants of the basic PC-SyncBB

In the previous section we described the basic version of PC-SyncBB. That version is immune against coalitions of size smaller than $n/2$. The main bottleneck of that algorithm is the MPC protocol (either Ben-Efraim-Omri [34] or

Damgård-Nielsen [31], depending on the selected implementation) that is invoked by compare_CPA_cost_to_upper_bound (see the experimental evaluation in Section 5). In the basic variant of PC-SyncBB, as described in Section 3.4, the MPC protocol is executed by all n agents. In this section we propose two variants of the compare_CPA_cost_to_upper_bound sub-protocol, which invoke the MPC protocol with smaller number of executing parties. By doing so, we put stricter limitations on the size of coalitions among the interacting parties; however, by invoking smaller scale instances of the MPC protocol, the overall runtime of PC-SyncBB reduces significantly and, as a consequence, it can be executed in larger problem settings.

Being able to rely on a smaller set of parties who conduct the secure computation has implications on the communication and computation costs of the protocol. The overall communication complexity of securely computing Eq. (7) is $O(n^2|C|)$, when using the Ben-Efraim-Omri protocol, or $O(n|C|)$ when using the Damgård-Nielsen protocol. Therefore, smaller number of parties imply smaller communication complexities. In particular, fixing that number to a constant means that the communication complexity does no longer depend on the number of agents, what may permit better scaling. As a side effect, since each random-secret-sharing and each secure-multiplication requires the parties to perform interpolation over a set of points whose size equals the number of participants, reducing this number improves the efficiency of those computations as well.

Note that the round complexity would not be affected since in both the Ben-Efraim-Omri and the Damgård-Nielsen cases, the protocols are constant-round (i.e., the round complexity is independent of the number of parties).

We refer the reader to Section 5.2.3 for the concrete improvement in performance for committees of 5,7, and 11 parties (i.e., with coalition size of 2,3, and 5, respectively).

4.1. A variant immune to coalitions of size $\leq c < (n-1)/2$

The first variant that we describe herein assumes that the coalition size is $\leq c$ for some constant $c < (n-1)/2$. Such a limitation on the coalition size enables higher efficiency in terms of runtime and communication costs. The idea is to delegate the inequality verification that is carried out in the sub-protocol compare_CPA_cost_to_upper_bound to a randomly selected committee of agents $\mathcal{C} \subset \mathcal{A} = \{A_1, A_2, \dots, A_n\}$, where $|\mathcal{C}| = 2c+1$. The inequality verification is carried out exactly as described in Section 3.4; however, as the number of interacting parties is $2c+1 < n$, the runtime and communication costs of the MPC protocol will be smaller than those in the original variant (in which the MPC protocol is carried out by all n agents). Under the assumption that the number of colluding agents is no larger than c, such a variant provides the same privacy guarantees as the original variant, but with reduced costs.

To that end, whenever the sub-protocol compare_CPA_cost_to_upper_bound is called, the n agents select a committee $\mathcal{C} \subset \mathcal{A}$, where $|\mathcal{C}| = 2c+1$. The selection is made randomly and independently each time the sub-protocol is called. We defer for later the description of the selection process. Assume that the selected committee is

$$\mathcal{C} = \{A_{i_0}, A_{i_1}, \dots, A_{i_{2c}}\} \ , \quad \text{where } \ 1 \leq i_0 < i_1 < \cdots < i_{2c} \leq n\,.$$

We recall that the computation that the agents need to perform at this stage is the one described by the function f at Eq. (7). That is, agent A_k has secrets a_k and b_k. To perform that verification in a more efficient manner that involves only the $2c+1$ committee members and not all n agents, agents $A_k \in \mathcal{A} \setminus \mathcal{C}$ 'deal' their secrets to agents in $\mathcal{C}$. Specifically, A_k splits a_k and b_k to $2c+1$ random shares, $a_{k,i_0}, \dots, a_{k,i_{2c}}$ and $b_{k,i_0}, \dots, b_{k,i_{2c}}$ respectively, so that

$$a_k = \sum_{j=0}^{c} a_{k,i_j} \mod S\,, \quad \text{and} \quad b_k = \sum_{j=0}^{c} b_{k,i_j} \mod S\,,$$

and then he sends a_{k,i_j} and b_{k,i_j} to agent $A_{i_j} \in \mathcal{C}$, $0 \leq j \leq 2c$. Now, each A_{i_j}

updates his own secret input to the MPC protocol as follows:

$$a_{i_j} \leftarrow \left(a_{i_j} + \sum_{A_k \in \mathcal{A} \setminus \mathcal{C}} a_{k,i_j} \right) \bmod S \quad \text{and} \quad b_{i_j} \leftarrow \left(b_{i_j} + \sum_{A_k \in \mathcal{A} \setminus \mathcal{C}} b_{k,i_j} \right) \bmod S \,.$$

After such an update, the committee members alone can compute Eq. (7) by invoking the MPC protocol, as described in Section 3.4, only that this time the number of interacting parties is smaller ($2c + 1$ instead of n).

It remains only to discuss the details of committee selection. To that end, we adopt the method of counter-mode encryption [35]. First, all n agents select a random 256-bit key K in the symmetric block cipher AES [36]. Specifically, each agent A_i selects his own random key K_i and then K is set to $\oplus_{i=1}^{n} K_i$, i.e., the bitwise XOR. For the purpose of performing this joint computation, each agent can just broadcast his own key to all other agents. Such a joint key K can be used by the agents in order to produce, each one independently on his own, the very same stream Σ of pseudorandom bits, as follows:

$$K \mapsto \Sigma := AES_K(0) || AES_K(1) || AES_K(2) || \cdots \,.$$

Then, whenever a new committee is to be selected, each of the agents can recover on his own the same committee by running independently Algorithm 3. (Here, $q := \lceil \log_2 n \rceil$, is the number of bits needed to identify one of the n agents.)

Algorithm 3 – Committee Selection Algorithm

for all $0 \le k \le n-1$ **do**
  $Selected(k) \leftarrow$ **false**
$CommSize \leftarrow 0$
$\mathcal{C} \leftarrow \emptyset$
while $CommSize < 2c + 1$ **do**
  $h \leftarrow$ next q bits from Σ
  if $h < n$ and $\neg Selected(h)$ **do**
    $Selected(h) \leftarrow$ **true**
    $CommSize \leftarrow CommSize + 1$
    $\mathcal{C} \leftarrow \mathcal{C} \bigcup \{A_{h+1}\}$

4.2. A mediated variant immune to any coalition size

In this variant we also delegate the verification of Eq. (7) to a committee of size $2c + 1$, and assume that no more than c of them may collude. However, in this case the committee is external to $\mathcal{A}$ and is fixed. Such a variant follows the computation model that is known in cryptography as "the mediated model", see e.g. [22, 23, 24]. In that model, there is a set of interacting agents that execute an MPC protocol. They export some of the computations to an external mediator (that can consist of several independent parties). The external mediators are expected to act honestly (namely, perform the computations that are delegated to them correctly), but at the same time they are not allowed access to private inputs of the agents.

Let $\mathcal{C} = \{B_0, B_1, \cdots, B_{2c}\}$ be the committee. Here, agent A_k splits his secret inputs a_k and b_k to random shares, $a_{k,0}, \ldots, a_{k,2c}$ and $b_{k,0}, \ldots, b_{k,2c}$ respectively, so that

$$a_k = \sum_{j=0}^{c} a_{k,j} \mod S\,, \quad \text{and} \quad b_k = \sum_{j=0}^{c} b_{k,j} \mod S\,,$$

and then he sends $a_{k,j}$ and $b_{k,j}$ to committee member $B_j \in \mathcal{C}$, $0 \leq j \leq 2c$. Then, B_j defines $\alpha_j = \sum_{k=1}^{n} a_{k,j}$ and $\beta_j = \sum_{k=1}^{n} b_{k,j}$. From that point, the committee members proceed to verify Eq. (7) by invoking the MPC protocol with the secret input α_j and β_j, $0 \leq j \leq 2c$.

We note that this variant is immune to any coalition among the agents themselves, since any such coalition that does not include all agents cannot use the shares that they hold in the CPA cost and in the upper bound in order to infer information on those values, nor on private information of agents outside the coalition. However, as stated earlier, it is assumed that among the external committee of mediators there is an honest majority (i.e., if some of them collude, the number of colluding mediators is at most c).

5. Experimental evaluation

We divide the experimental evaluation to two main parts. We begin by evaluating in Section 5.1 the runtime and communication complexity of the compare_CPA_cost_to_upper_bound sub-protocol, which is a central and computationally expensive part of PC-SyncBB. Subsequently, we evaluate the performance of the full PC-SyncBB algorithm in Section 5.2.

5.1. Evaluation of the compare_CPA_cost_to_upper_bound sub-protocol

For efficiency and reproducibility we use the implementation of Chida et al. [32] for the Damgård-Nielsen protocol [31], and the Ben-Efraim [37] implementation for the Ben-Efraim-Omri protocol [34], and compare them over identical settings. Both implementations are open source and available online. The executions were over LAN with EC2 machines of type `c5.large` in Amazon's North Virginia data center, with every agent running on a separate machine.

We measured performance for various values of n, where q (the maximum value of a single binary constraint) was set to 100. Hence, the maximum cost of any solution is $q_\infty := \binom{n}{2}q + 1$.

We now discuss the setting of the parameter S (the size of the group $\mathbb{Z}_S$ in which all computations take place). In the execution of the Ben-Efraim-Omri protocol, we set S to be the smallest power of 2 greater than $2q_\infty$, as required by [25]. In that case, the number of agents, n, fully determines S and the corresponding number of bits, $\ell = \log S$. Consequently, the value of n also determines the size of the circuit C that the protocol uses. In the execution of the Damgård-Nielsen protocol, on the other hand, the domain $\mathbb{Z}_S$ has to be a field. Hence, we set S to be a prime p larger than q_∞. As it turns out, selecting the prime p to be a Mersenne prime (a prime of the form $p = 2^t - 1$ for some integer $t > 1$) is advantageous in secret-sharing-based protocols, since multiplication of two field elements in such cases can be done without performing an expensive division (in case the multiplication result exceeds the modulus). We used two selections of Mersenne primes: $p_1 := 2^{13} - 1$ and $p_2 := 2^{31} - 1$. In experiments with $n \leq 13$ we set $S = p_1$ (since for those values of n, $p_1 > q_\infty$), while in experiments with $13 < n \leq 19$ we used $S = p_2$ (since $p_2 > q_\infty$ for this range of n values).

Table 1 shows a comparison of the runtimes of the two protocols. For the

Ben-Efraim-Omri protocol, the table presents the input bit length, $\ell = \log S$, as a function of n, under the above stated assumption of $q = 100$. It also presents the size of the corresponding Boolean circuit size, denoted $|C_{f'}|$, being the number of `AND` gates in it. Finally, it shows the runtime for securely evaluating that circuit in order to compute Eq. (8), where we separate between the overall runtime (namely, offline and online phases combined) and the runtime of the online phase alone (a number which may be of significance in settings where there is a sufficient time for running the offline phase ahead, and then execute in real time only the online phase). As for the Damgård-Nielsen protocol, the table presents the number of bits in the prime S that we used for each n (i.e., either $S = p_1$ or $S = p_2$ as discussed above), the size of the corresponding arithmetic circuit, denoted $|C_f|$ (counting the number of multiplication gates), and the runtime for securely evaluating that circuit in order to compute Eq. (7). All runtimes are in milliseconds and they represent an average over 100 executions.

The number of multiplication gates in the arithmetic circuit C_f is given by $279 \cdot \log p + 5$ in a circuit of depth 15. Thus, when $p = 2^{13} - 1$ we have $|C_f| = 3632$, whereas when $p = 2^{31} - 1$ we have $|C_f| = 8654$. Note that the Boolean circuits are much smaller than the arithmetic ones, in terms of number of "expensive" gates (i.e., `AND` gates in the Boolean case, and multiplication gates in the arithmetic case). Nonetheless, the Damgård-Nielsen protocol is much faster than the complete (i.e., offline and online) Ben-Efraim-Omri protocol, as is evident from the runtimes in Table 1. There are two reasons for that:

(1) As mentioned above, the underlying operations in the Damgård-Nielsen protocol are simple arithmetic field operations, i.e., addition and multiplication of (up to) 32-bit field elements. On the other hand, the underlying operations in the Ben-Efraim-Omri protocol are expensive AES [36] pseudorandom functions. Although the latter functions are implemented via CPU intrinsic instructions, they are still much more expensive than integer arithmetic.

(2) We execute the implementation by Chida et al. [32] of the Damgård-Nielsen protocol. That implementation introduced a very important optimization to the Damgård-Nielsen protocol. Specifically, the communication required

in their implementation for a sum-of-products operation equals the communication required by a single multiplication gate. That is, while in the original Damgård-Nielsen protocol, in order to securely compute a sub-circuit described as $\sum_{i=1}^{m} x_i \cdot y_i$, each party has to communicate $O(m)$ field elements, in the optimization by Chida et al. the corresponding communication cost is only $O(1)$.

Having said that, the Ben-Efraim-Omri protocol can be a better choice in settings where the agents may prepare in advance by generating upfront a sufficient number of garbles circuits for evaluating f' (Eq. (8)), and then during the real time computation perform only the online phase.

n	**Ben-Efraim-Omri**				**Damgård-Nielsen**		
	ℓ	$\|C_{f'}\|$	**total time**	**time online**	$\log p$	$\|C_f\|$	**time**
5	11	184	7.21	0.51	13	3632	4.3
6	12	255	10.23	0.68	13	3632	4.9
7	13	336	13.25	0.85	13	3632	6.6
8	13	392	17.37	1.08	13	3632	11.1
9	13	448	21.5	1.3	13	3632	12.7
10	14	549	27.7	1.45	13	3632	12.8
11	14	610	33.9	1.6	13	3632	16.5
12	14	671	41.75	2	13	3632	16.8
13	14	732	49.6	2.4	13	3632	18.5
14	15	858	62.05	2.45	31	8654	20.1
15	15	924	74.5	2.5	31	8654	20.7
16	15	990	85.75	2.6	31	8654	21.9
17	15	1056	97	2.7	31	8654	23.1
18	15	1122	117.95	3.15	31	8654	23.8
19	16	1278	138.9	3.6	31	8654	26

Table 1: Runtime in milliseconds, depending on the number of parties n for the Ben-Efraim-Omri and Damgård-Nielsen protocols.

Table 2 presents a comparison of the concrete communication complexity between the two protocols, denoted BO and DN. For BO, the circuit size refers to the number of `AND` gates in the Boolean circuit that computes Eq. (8), whereas for DN the circuit size refers to the number of multiplication gates in the circuit that computes Eq. (7). Communication rounds refer to the number of times the parties have to interact. For BO, κ refers to the computational security parameter, which equals 128 in their implementation. As mentioned above, for DN, $\log p = 13$ for $n \leq 13$ and $\log p = 31$ for $13 < n \leq 19$.

The communication analysis, like the runtime analysis, suggests that in cases where the agents may prepare upfront, the Ben-Efraim-Omri protocol may be a better choice. Indeed, as can be seen from Table 2, the number of communication rounds in the online phase in that protocol is only 2, comparing to 17 in Damgård-Nielsen, and the number of messages in that online phase is smaller than the corresponding number in Damgård-Nielsen for $n \leq 8$.

	Ben-Efraim-Omri		**Damgård-Nielsen**
Circuit size	$\lvert C_{f'}\rvert = (n-1)(1+5(\ell-2))$		$\lvert C_f\rvert = 279 \log p + 5$
	offline	online	
Communication rounds	3	2	17
Number of messages	$3n^2$	$2n^2$	$17n$
Total communication	$8n^2\kappa\lvert C_{f'}\rvert$	$n^2\kappa\ell$	$3n\lvert C_f\rvert \log p$

Table 2: Communication complexities of the Ben-Efraim-Omri and Damgård-Nielsen protocols. The total communication measure is in bits. Recall that the circuit size $|C_{f'}|$ was calculated in Section 3.4.2, while the circuit size $|C_f|$ is as reported in Nishide and Ohta [33] (see our discussion in Section 3.4.3). In addition, round, message, and communication complexities are taken from the reports of Ben-Efraim-Omri [34] and Damgård-Nielsen [31].

5.2. Evaluation of the full PC-SyncBB algorithm

Now we turn to the performance evaluation of the full PC-SyncBB algorithm. In order to asses the toll of privacy preservation, we compare PC-SyncBB to other algorithms that maintain the Branch & Bound structure – P-SyncBB [18] that preserves privacy only under the assumption of non-colluding agents, and the basic insecure SyncBB [5].

The conclusion from our discussion in Section 5.1 indicates that there is no clear "winner" regarding the protocol to be used in compare_CPA_cost_to_upper_bound. While the Damgård-Nielsen protocol [31] is overall faster, the Ben-Efraim-Omri protocol [34] may be advantageous in applications that allow offline computations. Consequently, we present three results for PC-SyncBB: (*i*) PC-SyncBB-BO, which is the overall effort using the Ben-Efraim-Omri protocol, (*ii*) PC-SyncBB-BO-online, which represents only the online computation when using the Ben-Efraim-Omri protocol, and (*iii*) PC-SyncBB-DN, which is the overall effort using the Damgård-Nielsen protocol.

The algorithms were implemented[6] and executed in the AgentZero simulator[7] [38], running on a hardware comprised of an Intel i7-6820HQ processor and 32GB memory, except for the calls to the compare_CPA_cost_to_upper_bound procedure that were executed on the machines from Amazon's North Virginia data center, in order to simulate as realistically as possible a truly distributed environment. We followed the *simulated time* [39] approach in all the subsequent experiments. The results are shown in a logarithmic scale and are the average over 50 problem instances (for each setting/benchmark).

5.2.1. Runtime performance in various benchmarks

In accordance with the experiments of P-SyncBB [18], we use four benchmarks for evaluating the performance of PC-SyncBB – random DCOPs, graph coloring problems, scale-free networks, and meeting scheduling problems.

The first benchmark consists of unstructured randomly generated DCOPs on which we perform two experiments. In the first experiment, presented in Figure 1, we fix the number of agents to $n = 7$ and the domain sizes to $d = 6$, and vary the constraint density $0.3 \leq p_1 \leq 0.9$. (Using lower density values $p_1 < 0.3$ results in unconnected constraint graphs.) As can be clearly seen, constraint density only mildly affects the runtime performance of all the eval-

[6] `https://github.com/grinshpo/PCSyncBB_implementation_and_experiments`

[7] AgentZero Tutorial, including installation instructions: `https://docs.google.com/document/d/1B19TNQd8TaoAQVX6njo5v9uR3DBRPmFLhZuKOH9Wiks/view`

uated algorithms. However, the toll of privacy preservation is evidently high, with each layer of protection adding about two orders of magnitude to the runtime. Specifically, the online part of PC-SyncBB-BO requires about one order of magnitude more time than P-SyncBB, and PC-SyncBB-DN is faster than the overall (offline+online) PC-SyncBB-BO.

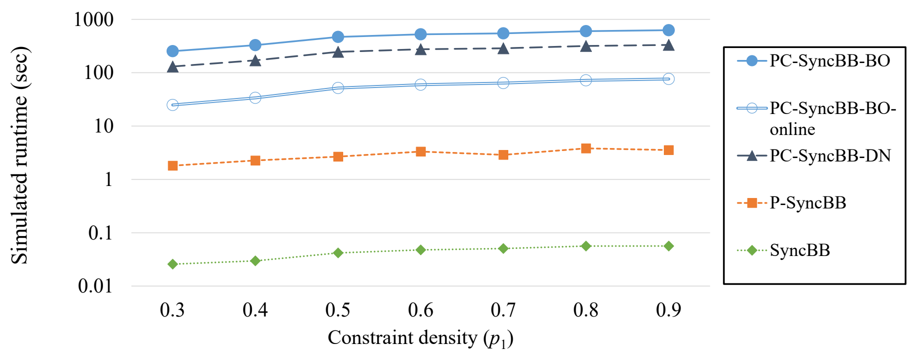

Figure 1: Runtime performance in random DCOPs ($n = 7$, $d = 6$, varying p_1).

In the second experiment, shown in Figure 2, we fix the constraint density to $p_1 = 0.3$ and the domain sizes to $d = 6$, and vary the number of agents $5 \leq n \leq 9$. It is clear that the number of agents has a major effect on the performance of all the evaluated algorithms, in accordance with known results regarding the scalability of Branch & Bound algorithms in computationally hard problems. Interestingly, P-SyncBB scales slightly better, probably due to its inherent use of sorted value ordering.

Next, we evaluate the scalability in more structured benchmarks. The second benchmark consists of distributed 3-color graph coloring problems in which each pair of equal values of constrained agents imposes a random and *private* cost of up to $q = 100$. The structure in these problems lies in the diagonal constraint matrices between every pair of neighboring agents.

Figure 3 depicts the runtime performance in distributed 3-color graph coloring problems ($p_1 = 0.4$, $5 \leq n \leq 19$) and shows similar scalability trends to those of random DCOPs. However, the small domain size ($d = 3$) enables

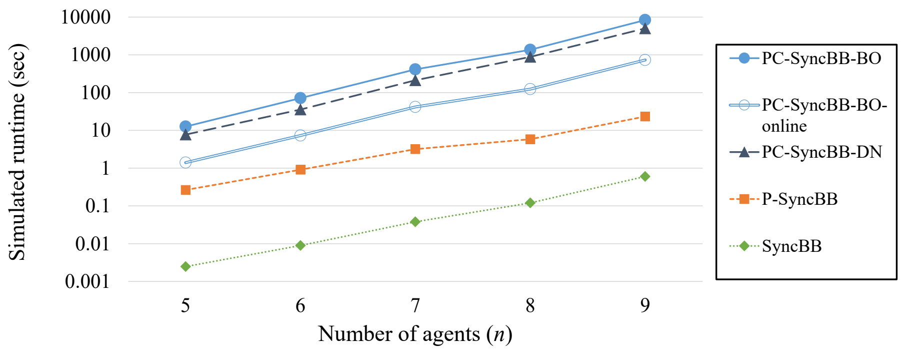

Figure 2: Runtime performance in random DCOPs ($p_1 = 0.3$, $d = 6$, varying n).

running problems of larger size, which clearly show that PC-SyncBB-DN scales better than PC-SyncBB-BO.

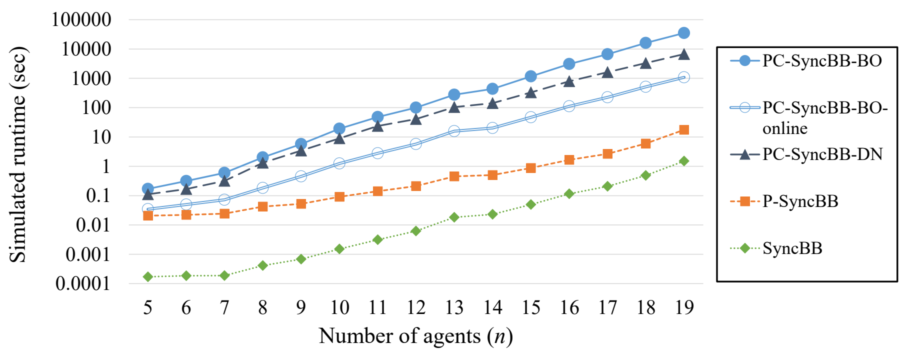

Figure 3: Runtime performance in 3-color graph coloring problems ($p_1 = 0.4$, varying n).

Similar trends are also witnessed in the other benchmarks. Figure 4 presents the runtime performance in scale-free networks ($7 \leq n \leq 13$, domains of size $d = 5$), which are structured networks that are generated according to the Barabási-Albert model [40]. As part of the Barabási-Albert network construction procedure, we use an initial set of $m_0 = 4$ connected agents, and connect every new added agent to $m = 2$ existing agents in the network in a probability that is proportional to the number of links that the existing agents already have. The Barabási-Albert model is commonly used for the representation of large networks with hubs; the results in Figure 4 indicate that the PC-SyncBB algorithm is not suitable for such networks.

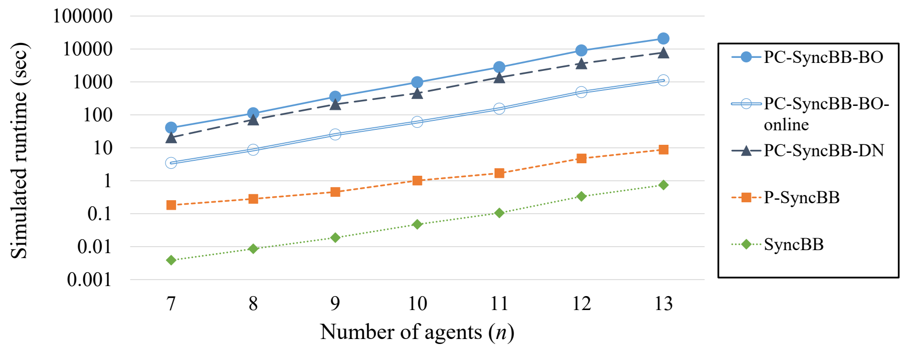

Figure 4: Runtime performance in scale-free networks ($m_0 = 4$, $m = 2$, $d = 5$, varying n).

Finally, we examine the scalability of runtime performance in distributed meeting scheduling problems, which are highly structured real-world problems. We construct the problems similarly to the PEAV (Private Events As Variables) formulation [2], which is aimed for scenarios where privacy is a concern. The PEAV formulation generates multiple-variable agents. However, the presentation of most DCOP algorithms assumes a single variable per agent, so for simplicity and clarity reasons we follow the experimental setting of the P-SyncBB experiments [18], which uses the *decomposition* method that turns each variable into a *virtual agent* [41].

As in the setting of Grinshpoun and Tassa [18], inspired by the meeting scheduling experiments of Léauté and Faltings [14], the number of meetings m is varied, while the number of participants per meeting is fixed to 2. For each meeting, participants were randomly drawn from a common pool of 3 agents. The goal is to assign a time to each meeting among $d = 8$ available time slots. Two types of preferences are considered – *time preference* (a cost of $0, \ldots, 3$ for each time slot) and *meeting importance* (a cost of $5, \ldots, 9$ for each meeting). Figure 5 presents the runtime performance in the described meeting scheduling problems. Clearly, this highly structured setting shows similar scalability trends to those of the other benchmarks.

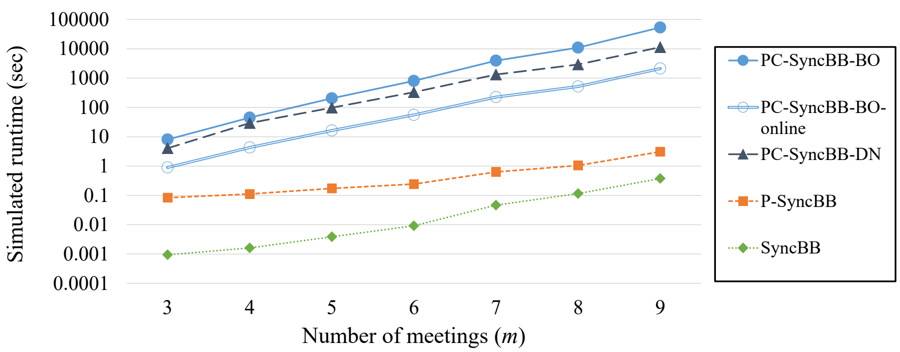

Figure 5: Runtime performance in meeting scheduling problems with varying number of meetings m. Each meeting includes 2 agents (out of a pool of 3 agents in total). There are $d = 8$ available time slots for the meetings. Costs are given to time preferences $[0, \dots, 3]$ and meetings importance $[5, \dots, 9]$.

5.2.2. Communication complexity

Communication complexity in DCOPs is traditionally measured in terms of the total number of messages exchanged throughout the solving process. However, in cases where there are considerable differences in message sizes, it is also desired to evaluate the network load, i.e., the overall size of exchanged information. Such metric is common when evaluating inference-based DCOP algorithms [42, 43], which typically involve messages of exponential size. The privacy-preserving algorithms herein also require exchanging large messages, because, for instance, an encrypted number typically requires using a dynamic-size structure, such as BigInteger (Java), instead of a primitive `int`. Hence, we use herein both the number of messages and the network load metrics to evaluate the communication complexity.

We start by returning to the setting of Figure 1 (random DCOPs, $n = 7$ agents, domain sizes of $d = 6$, and constraint densities of $0.3 \leq p_1 \leq 0.9$), and counting the total number of sent messages in this setting. The results, given in Figure 6, show that like in the case of runtime performance, the communication complexity is also only mildly affected by the varying constraint density. Similarly to the trends in Figure 1, the overhead of the PC-SyncBB versions is about two orders of magnitude compared to P-SyncBB. However, the differences between the PC-SyncBB versions are mild, with PC-SyncBB-DN exchanging only a slightly higher number of messages than PC-SyncBB-BO-online.

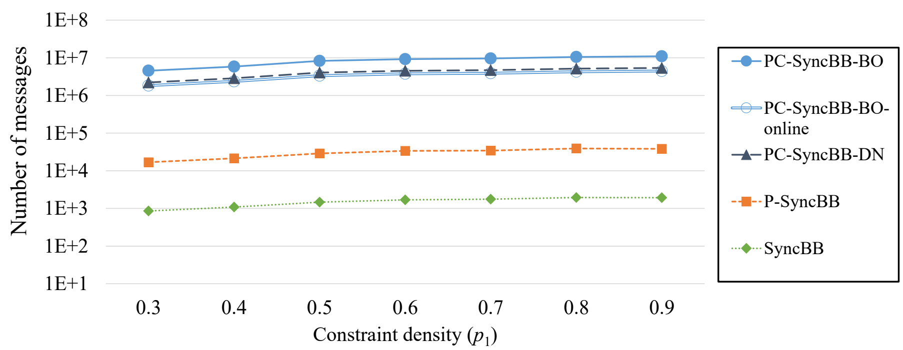

Figure 6: Total number of messages in random DCOPs ($n = 7$, $d = 6$, varying p_1).

Figure 7 displays the network load results (in kilobytes) for the same setting. Here, the differences between the versions of PC-SyncBB are more substantial, and especially the gap between the online part and overall network load of PC-SyncBB-BO.

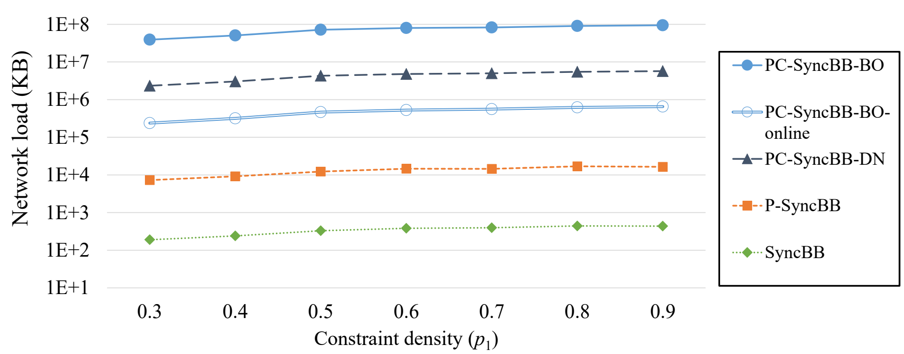

Figure 7: Network load in random DCOPs ($n = 7$, $d = 6$, varying p_1).

Next, in order to evaluate the scalability of the algorithms in terms of communication, we return to the setting of Figure 3 (distributed 3-color graph coloring problems, $p_1 = 0.4$, $5 \leq n \leq 19$). Figure 8 depicts the total number of messages for this setting, and shows a very interesting phenomenon –

PC-SyncBB-DN exchanges less messages even than just the online part of PC-SyncBB-BO, for $n \geq 9$. Nevertheless, the communication complexity in terms of the number of messages is rather similar between the versions of PC-SyncBB, and between the online and offline phases of PC-SyncBB-BO.

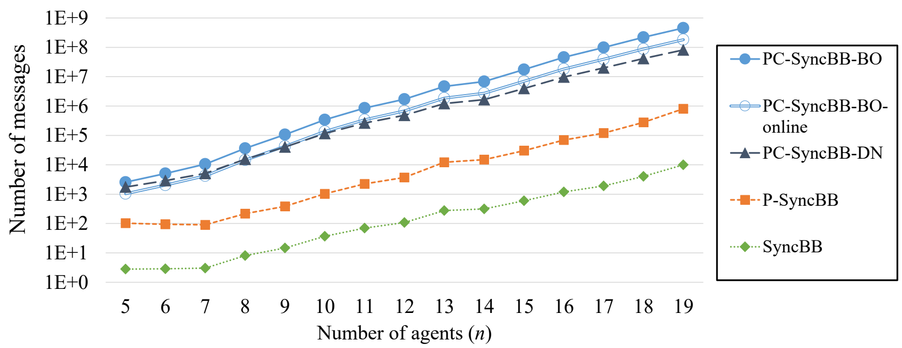

Figure 8: Total number of messages in 3-color graph coloring problems ($p_1 = 0.4$, varying n).

The story is quite different when considering the network load in this setting, see Figure 9. Regarding network load, PC-SyncBB-online scales considerably better than the overall PC-SyncBB-BO that also includes the offline phase. In fact, it is just over an order of magnitude larger than that of P-SyncBB. Also interesting is the network load of PC-SyncBB-DN, which rises steeply when moving from $n = 13$ to $n = 14$. This is due to the required change in the Mersenne prime used, see Section 5.1.

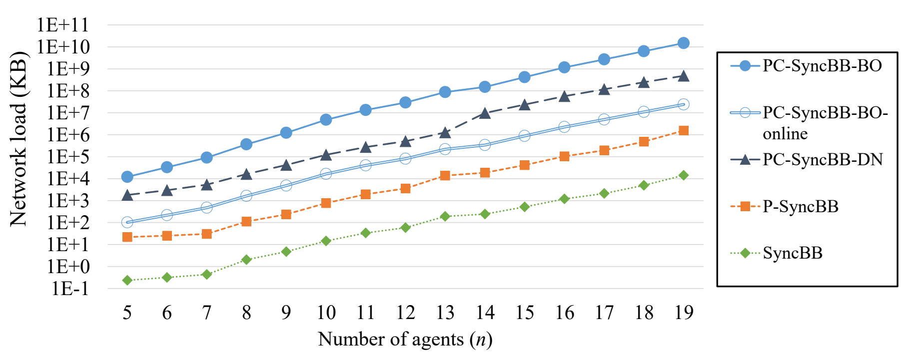

Figure 9: Network load in 3-color graph coloring problems ($p_1 = 0.4$, varying n).

Communication complexity shows very similar scalability trends in other benchmarks (random DCOPs, scale-free networks, meeting scheduling), hence such graphs are omitted.

5.2.3. Varying size of committees

In Section 4.1 we introduced a variant of our algorithm that enables immunity against coalitions of size c smaller than $(n-1)/2$. Here we conduct experiments on varying size of coalitions.

In order to show meaningful results, we chose to focus on the graph coloring benchmark, which enables running problems with relatively large number of agents. Figure 10 depicts the runtime performance on the same distributed 3-color graph coloring problems as in Figure 3 ($p_1 = 0.4$, $5 \leq n \leq 19$). For coalitions of size 1 we use the P-SyncBB algorithm [18]. We also depict the runtime when running PC-SyncBB while limiting the maximal coalition sizes to 2, 3, and 5, by PC-SyncBB-c2, PC-SyncBB-c3, and PC-SyncBB-c5, respectively. Finally, PC-SyncBB depicts the runtime of the standard PC-SyncBB algorithm, without limitation on the maximal coalition size (except for the honest majority assumption). In all versions of PC-SyncBB herein, we use the Damgård-Nielsen [31] variant. The same trends between the Damgård-Nielsen and Ben-Efraim-Omri protocols that were shown in Figure 3 also hold here, so we omit the Ben-Efraim-Omri results for clarity of presentation.

The results in Figure 10 show that limiting coalition sizes improves the runtime performance by up to 5-6 times (according to the difference between PC-SyncBB and PC-SyncBB-c2 for $n = 19$ agents). Such a difference is indeed significant, so running PC-SyncBB according to an a priori known maximal number of colluders is a good idea. Another interesting phenomenon is that the runtime of P-SyncBB is orders of magnitude shorter than that of PC-SyncBB-c2. This comes as no surprise, since P-SyncBB does not need to incorporate protection mechanisms against coalitions, and is thus inherently faster. Hence, whenever there is no suspicion of colluding agents, P-SyncBB should be used for improved performance.

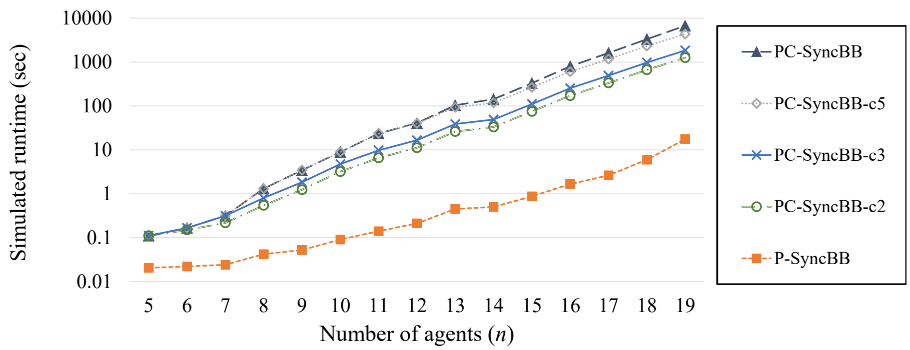

Figure 10: Runtime performance for different coalition sizes in 3-color graph coloring problems ($p_1 = 0.4$, varying n).

The trends are similar, although gaps are smaller, when considering the total number of messages metric, see Figure 11.

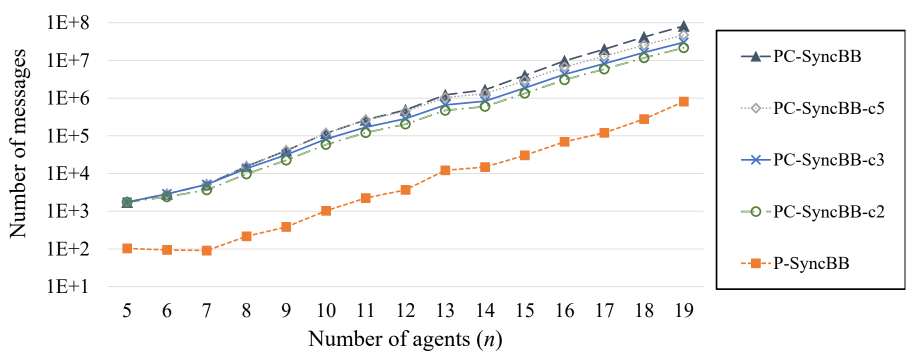

Figure 11: Total number of messages for different coalition sizes in 3-color graph coloring problems ($p_1 = 0.4$, varying n).

Figure 12 displays the corresponding results for the network load measure. Here, we can see that assuming smaller sizes of coalitions can drastically reduce the network load. This is due to the change in the Mersenne prime used for problems with $n \geq 14$. Such a change is no longer required when limiting the size of coalitions to up to $c = 6$.

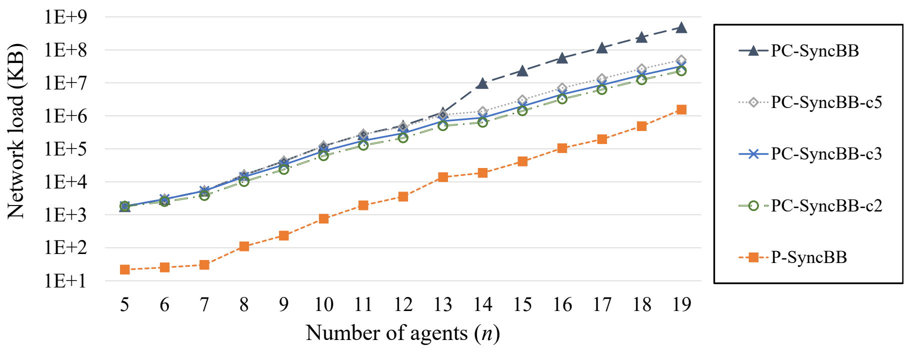

Figure 12: Network load for different coalition sizes in 3-color graph coloring problems ($p_1 = 0.4$, varying n).

6. Conclusion

We proposed herein PC-SyncBB, the first privacy-preserving DCOP algorithm which is secure against coalitions. It is based on the complete SyncBB algorithm. We showed how the agents can simulate all of SyncBB's input-dependent operations while preserving the privacy of their sensitive constraint, topology and assignment/decision information, even in the presence of coalitions smaller than half the number of agents. We analyzed the properties of the algorithm and evaluated its performance. Our experiments demonstrate that PC-SyncBB is feasible for moderately-sized problems.

We also proposed two additional variants of the algorithm. The first variant is secure against coalitions of size $\leq c$, for some constant $c < (n-1)/2$. Our experiments show that this variant improves the performance of the algorithm, and especially the network load. The second variant incorporates the mediated model. As a consequence, it is immune to agent coalitions of any size, but it relies on an external committee of mediators with an honest majority.

A major limitation on the scalability of PC-SyncBB is due to the cryptographic MPC protocols invoked by the compare_CPA_cost_to_upper_bound sub-protocol. In the preliminary version of this study [25], we used a garbled-circuit-based protocol – the Ben-Efraim-Omri (BO) protocol [34]. We chose an

MPC protocol that is based on a Boolean circuit because the desired function that needs to be computed in compare_CPA_cost_to_upper_bound, Eq. (7), can be represented by Eq. (8), which strongly suggests using a Boolean circuit. However, an arithmetic circuit representation turns out to be much more efficient. Specifically, the solution that we present here computes Eq. (7) by a secure emulation of an arithmetic circuit; the secure emulation of that circuit is carried out by the Damgård-Nielsen (DN) protocol [31]. Our experiments show that the DN-based implementation is more efficient than the BO-based implementation. However, the BO-based implementation can be significantly more efficient in applications that allow offline computations.

As explained in the Introduction, all existing privacy-preserving DCOP algorithms base their security on assuming solitary conduct of the agents. Alas, such an assumption may not always hold, and if indeed two or more corrupted agents collude, they may breach the privacy guarantees of those algorithms. This study is the first one that addresses this risk of privacy-breach, by introducing the first privacy-preserving algorithm that is secure against coalitions. We chose to depart from SyncBB, the first DCOP algorithm [5], because it is a complete algorithm (i.e., it outputs an optimal solution), and its information flow enabled a rather simple cryptographic reconstruction that achieves the desired properties. However, it is necessary to develop privacy-preserving and collision-secure implementations of other DCOP algorithms. We believe that immediate efforts should be invested in incomplete DCOP algorithms, that could offer better scalability.

This work can be extended in future work in several directions. We assume in this study that all constraints are binary. This common assumption is widely used in DCOP literature, and in fact in all prior art on privacy-preserving DCOPs, following the equivalence result of Rossi et al. [44]. Nevertheless, modeling a problem with only binary constraints is not always efficient [45]; thus, the extension of PC-SyncBB to an algorithm that handles non-binary constraints is an important future prospect. Another important extension is that to the *asymmetric* DCOP model (ADCOP) [46], in which a constraint may impact differently the agents that share it, as is commonly the case in many multi-agent settings. The asymmetry of constraints introduces an additional type of privacy concern, namely *internal constraint privacy* [47], hence privatizing an asymmetric version of SyncBB is an interesting challenge. However, the move from DCOP to ADCOP is not trivial and requires substantial algorithmic effort, e.g. [42, 46, 48, 49], and is thus left for a separate thorough study. Finally, another interesting future research direction is to devise secure implementations of SyncBB that are secure in the presence of malicious agents, i.e., agents that might deviate from the prescribed protocol in attempt to extract sensitive information on other agents.

Appendix A. Illustrating the operation of PC-SyncBB

In this section we illustrate the operation of PC-SyncBB on a small problem. We consider a setting with $n = 4$ agents that control four variables, X_1, X_2, X_3, X_4. The constraint graph between those variables is depicted in Figure A.13. As can be seen there, four out of the six pairs of variables are constrained. Hence $\Gamma(X_2, X_4) = \Gamma(X_3, X_4) = 0$, while $\Gamma(X_i, X_j) = 1$ otherwise.

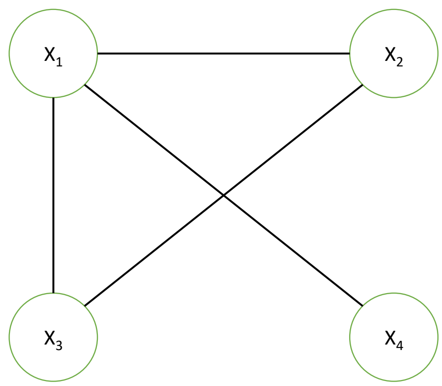

Figure A.13: The constraint graph of the exemplified DCOP over four variables.

We assume that the upper bound on any single binary constraint cost is $q = 10$. Hence, the upper bound that we set in PC-SyncBB on the cost of any solution is $q_\infty = \binom{4}{2} \cdot q + 1 = 61$. Consequently, all computations will be carried out modulo $S = 67$, which is the smallest prime larger than q_∞.

For simplicity, we shall assume that all domains are the same, $D_k = \{10, 20, 30\}$, and that the fixed order on all of them is $\mathbf{u}_k = (10, 20, 30)$, $1 \leq k \leq 4$. The constraint matrices between every pair of constrained variables are given in

Table A.3.

X_2

X_1	10	20	30
10	5	6	4
20	7	9	1
30	10	4	0

X_3

X_1	10	20	30
10	2	3	4
20	1	2	1
30	3	4	0

X_4

X_1	10	20	30
10	9	8	9
20	7	6	10
30	10	7	0

X_3

X_2	10	20	30
10	0	6	7
20	0	6	3
30	0	9	5

Table A.3: The constraint matrices between every pair of constrained variables

Next, we begin the description of PC-SyncBB's run on the above problem. We do so by a sequence of 52 "snapshots" of that run, which correspond to one possible execution of PC-SyncBB. (Recall that PC-SyncBB is an algorithm in which the agents make randomized decisions, so it may have several execution scenarios on the very same problem.)

Each snapshot shows:

- The cost of the current partial assignment, $cost(CPA)$, and the current upper bound (being the minimal cost of a full solution that was found so far), UB.
- The list of agents, where the currently active agent is marked by blue.
- The list of variables and their currently assigned value (for the variables that are included in the CPA).
- Shares of pairwise costs, i.e. $\text{sCPA}_k(t)$ for all pairs $1 \leq kconstraint \neq t \leq n = 4$.

- Shares of the upper bound, sUB_k.
- The current optimal setting for each variable X_k.
- The random ordering by which each agent A_k traverses his domain D_k. Recall that this ordering is denoted $\mathbf{w}_k$, and it is generated at random by A_k whenever he begins a new traversal over his domain.
- The pointer p_k that points to a value in the ordering $\mathbf{w}_k$; the current assignment to X_k is given by $\mathbf{w}_k(p_k)$ and it is marked by blue.

For convenience, we mark in each snapshot the values that had changed in that step by red.

Snapshot 1 illustrates the initialization of variables as done in PC-SyncBB's procedure **init**: all shares of pairwise costs are initialized to zero (Line 1), all pointers p_k are set to zero (Line 2), and the shares of the upper bound are set according to Lines 3-6.

Snapshot 2 illustrates the execution of the **assign_CPA** procedure by A_1. He generates a new random ordering $\mathbf{w}_1$ over his domain, increments p_1 to 1 and assigns to X_1 the value $\mathbf{w}_1(p_1) = 20$. As the current CPA involves only X_1, all $\mathrm{sCPA}_k(t)$ shares, $1 \leq k \neq t \leq 4$, remain zero, and $cost(CPA) = 0$. That cost is compared to UB, which is currently set to $q = 61$. Since that comparison (PC-SyncBB, Line 21) yields a **true** value, the search torch is passed on to A_2 (Line 24).

Snapshot 3 illustrates the similar actions that A_2 performs in **assign_CPA**. Note that here $cost(CPA) = 1$, since that is the cost when $X_1 = 20$ and $X_2 = 30$. This value is encoded in the two shares $\mathrm{sCPA}_1(2)$ and $\mathrm{sCPA}_2(1)$. Agents A_1 and A_2 compute those shares, securely, by running update_shares_in_CPA (Line 15). The resulting random shares in the illustrated example are, as shown in the snapshot, $\mathrm{sCPA}_1(2) = 33$ and $\mathrm{sCPA}_2(1) = 35$. Indeed, the sum of those two values modulo $S = 67$ equals $cost(CPA) = 1$. Note that all other shares are still zero, since only X_1 and X_2 have assigned values at this point. Since $cost(CPA) = 1$ is still smaller than $UB = 61$, the search torch is passed on to

A_3.

Snapshots 4-5 describe the assignments to X_3 and X_4. At this stage we have a full set of assignments, as shown in Snapshot 5. The overall cost of that full assignment is $cost(CPA) = 8$, as can be easily verified against Table A.3. Recall that $cost(CPA)$ is not stored anywhere; its only existence is through the shares $\text{sCPA}_k(t)$. We leave it to the reader to verify that for each pair of constrained variables, X_t and X_k, where $t < k$, the sum of $\text{sCPA}_t(k)$ and $\text{sCPA}_k(t)$ modulo S equals $C_{t,k}(X_t, X_k)$ (as given in Table A.3), in accord with Eq. (2). Consequently, the sum of all those shares, over all $1 \leq k \neq t \leq n = 4$, equals $cost(CPA) = 8$ modulo $S = 67$ (see Eq. (3)).

When the last agent A_4 triggers an execution of the comparison procedure compare_CPA_cost_to_upper_bound (Line 17) in order to compare $cost(CPA) = 8$ to $UB = 61$, the result is **true**.

Snapshots 6: As a result, a message **NEW_OPTIMUM_FOUND** is broadcast (Line 18). Consequently, all shares sUB_k are updated according to Line 31 in PC-SyncBB. For example, $\text{sUB}_2 = \text{sCPA}_2(1) + \text{sCPA}_2(3) + \text{sCPA}_2(4) = 35 + 42 + 0 = 10 \bmod S = 67$. Now, the sum of all those shares equals the new upper bound, namely, $\sum_{k=1}^{4} \text{sUB}_k = 44 + 10 + 18 + 3 = 75$, equals $UB = 8 \bmod S = 67$. Additionally, all agents store their optimal setting in the solution that was just found to be the optimal solution thus far in the search (PC-SyncBB, Line 32).

Snapshots 7: Here, A_4 proceeds to assign the next value in his domain (Line 19). To that end he increments p_4 to 2 (Line 10) and assigns $X_4 \leftarrow \mathbf{w}_4(p_4) = 10$ (Line 14). As X_4 is constrained only with X_1 then only the two shares $\text{sCPA}_1(4)$ and $\text{sCPA}_4(1)$ are updated. Now they equal 66 and 8, respectively, so that their sum modulo $S = 67$ equals 7, which is $C_{1,4}(X_1 = 20, X_2 = 10) = 7$. Now, in wake of that assignment, $cost(CPA)$ grew by 1, from 8 to 9. Therefore, the comparison in this case (Line 17) yields a **false** value. In that case, all that is left to do is to proceed to assign the next value from D_4 (Line 19).

In **Snapshot 8** A_4 proceeds to assign the next value from D_4 tp X_4. That assignment does not produce a new optimum either. Hence, in **Snapshot 9** A_4

executes once again **assign_CPA** (Line 19). But this time, since he completed a full scan over D_4, he backtracks (Line 12). As part of backtracking, A_4 zeros his share with each of the agents in $I_4^- = \{A_1\}$ (Line 26); in our case that amounts to zeroing $\text{sCPA}_4(1)$. In addition, he sends a **ZERO_SHARE_MSG** message to A_1 (Line 27), who, subsequently, sets $\text{sCPA}_1(4) = 0$ (Line 35). As a result, the remaining shares add up to $cost(CPA) = 2 \bmod S = 67$, which is indeed the cost of the reduced CPA over the first three variables, as was the case in Snapshot 4. Next, A_4 sends a **BACKTRACK_MSG** message to A_3 (Line 28); upon receiving that message, A_3 calls the **assign_CPA** procedure (Line 36).

Snapshot 10: Here, A_3 assigns the next value from $\mathbf{w}_3$ to X_3. The resulting $cost(CPA)$ is 12. Indeed, by Table A.3, $C_{1,2}(X_1 = 20, X_2 = 30) = 1$, $C_{1,3}(X_1 = 20, X_3 = 20) = 2$, and $C_{2,3}(X_2 = 30, X_3 = 20) = 9$, and the sum of the above three costs is 12. The reader can see that exactly four shares $\text{sCPA}_k(t)$ changed (all those that relate to the changed variable X_3) and the sum of all shares indeed add up to 12. As that value is greater than $UB = 8$, the comparison in Line 12 issues a **false** answer. Hence, there is no need to examine the rest of this search path, and thus A_3 proceeds to examine the next value in his domain (Line 22).

Snapshot 11: Here, A_3 assigns $X_3 \leftarrow \mathbf{w}_3(3) = 30$. The resulting $cost(CPA)$ is 7, which is smaller than $UB = 8$ (i.e., the comparison in Line 21 issued a **true** result). As a consequence, A_3 sends a **CPA_MSG** message to A_4 (Line 24).

Snapshot 12: Upon receiving the **CPA_MSG** message, A_4 prepares for a new traversal of his domain D_4: he sets $p_4 \leftarrow 0$ (Line 33), calls assign_CPA (Line 34), generates a new random ordering $\mathbf{w}_4$ over D_4 (Line 9), increment p_4 to 1 (Line 10), and then sets a new value to X_4 (Line 14). The resulting cost of the CPA turns out to be 17, which is larger than UB (**false** in Line 17). Hence, A_4 calls once again assign_CPA (Line 19).

Snapshots 13-14: Here, A_4 tests the remaining two values in his domain against the current assignments of X_1, X_2, X_3. Neither of them produces a new optimum.

In **Snapshot 15** A_4 backtracks. In doing so, the two shares sCPA$_1(4)$ and sCPA$_4(1)$ are zeroed, and then $cost(CPA)$ reduces to 7, as it was in Snapshot 11, since now the CPA involves only X_1, X_2, X_3.

Then, in **Snapshot 16** also A_3 backtracks since he had exhausted his domain. Once again, all shares that relate to X_3 are zeroed and $cost(CPA)$ reduces to 1, its historic value from Snapshot 3, since now the CPA involves only X_1, X_2.

We arrive at **Snapshot 17** where A_2 increments p_2 in order to test the next assignment from his domain. The examination of the CPA $X_1 = \mathbf{w}_1(1)$ and $X_2 = \mathbf{w}_2(2)$ proceeds in **Snapshots 18-21**. That search yields no new optimum. In **Snapshot 22** the CPA $X_1 = \mathbf{w}_1(1)$ and $X_2 = \mathbf{w}_2(3)$ is tested. Since the cost of that single binary constraint is 9, which is larger than $UB = 8$, that CPA is abandoned very quickly. After A_2 backtracks in **Snapshot 23**, A_1 advances his variable assignment to the next value in $\mathbf{w}_1$, which is 30 (**Snapshot 24**).

The search proceeds in that manner. A new optimum of $UB = 7$, which improves on the previous optimum of $UB = 8$, is found in **Snapshot 30**. Another improvement, to $UB = 3$, is reached in **Snapshot 40**. The search terminates in **Snapshot 52** with a message of **COMPLETE** (Lines 30 and 37-38 in PC-SyncBB).

Snapshot 1

Agents	A_1	A_2	A_3	A_4
Variables	X_1	X_2	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 61$	$sUB_2 = 0$	$sUB_3 = 0$	$sUB_4 = 0$
OptimalSetting	–	–	–	–
Pointer	$p_1 = 0$	$p_2 = 0$	$p_3 = 0$	$p_4 = 0$
Random ordering				

Snapshot 2

Cost(CPA) = 0 < UpperBound = 61 = true

Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	X_2	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 61$	$sUB_2 = 0$	$sUB_3 = 0$	$sUB_4 = 0$
OptimalSetting	–	–	–	–
Pointer	$p_1 = 1$	$p_2 = 0$	$p_3 = 0$	$p_4 = 0$
Random ordering	$\mathbf{w}_1 = (20,30,10)$			

Snapshot 3

1380

Cost(CPA) = 1	<	UpperBound = 61	= true	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 35$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 61$	$sUB_2 = 0$	$sUB_3 = 0$	$sUB_4 = 0$
OptimalSetting	–	–	–	–
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 0$	$p_4 = 0$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (30,10,20)$		

Snapshot 4

Cost(CPA) = 2	<	UpperBound = 61	= true	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 10$	X_4
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 8$ $sCPA_1(4) = 0$	$sCPA_2(1) = 35$ $sCPA_2(3) = 42$ $sCPA_2(4) = 0$	$sCPA_3(1) = 60$ $sCPA_3(2) = 25$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 61$	$sUB_2 = 0$	$sUB_3 = 0$	$sUB_4 = 0$
OptimalSetting	–	–	–	–
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 1$	$p_4 = 0$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (30,10,20)$	$\boldsymbol{w}_3 = (10,20,30)$	

Snapshot 5

Cost(CPA) = 8	<	UpperBound = 61 = true		
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 10$	$X_4 = 20$
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 8$ $sCPA_1(4) = 3$	$sCPA_2(1) = 35$ $sCPA_2(3) = 42$ $sCPA_2(4) = 0$	$sCPA_3(1) = 60$ $sCPA_3(2) = 25$ $sCPA_3(4) = 0$	$sCPA_4(1) = 3$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 61$	$sUB_2 = 0$	$sUB_3 = 0$	$sUB_4 = 0$
OptimalSetting	–	–	–	–
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 1$	$p_4 = 1$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (10,20,30)$	$\mathbf{w}_4 = (20,10,30)$

Snapshot 6

Cost(CPA) = 8		UpperBound = 8	NEW_OPTIMUM_FOUND	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 10$	$X_4 = 20$
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 8$ $sCPA_1(4) = 3$	$sCPA_2(1) = 35$ $sCPA_2(3) = 42$ $sCPA_2(4) = 0$	$sCPA_3(1) = 60$ $sCPA_3(2) = 25$ $sCPA_3(4) = 0$	$sCPA_4(1) = 3$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 1$	$p_4 = 1$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (10,20,30)$	$\mathbf{w}_4 = (20,10,30)$

Snapshot 7

Cost(CPA) = 9	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 10$	$X_4 = 10$
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 8$ $sCPA_1(4) = 66$	$sCPA_2(1) = 35$ $sCPA_2(3) = 42$ $sCPA_2(4) = 0$	$sCPA_3(1) = 60$ $sCPA_3(2) = 25$ $sCPA_3(4) = 0$	$sCPA_4(1) = 8$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 1$	$p_4 = 2$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (10,20,30)$	$\mathbf{w}_4 = (20,10,30)$

Snapshot 8

Cost(CPA) = 12	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 10$	$X_4 = 30$
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 8$ $sCPA_1(4) = 5$	$sCPA_2(1) = 35$ $sCPA_2(3) = 42$ $sCPA_2(4) = 0$	$sCPA_3(1) = 60$ $sCPA_3(2) = 25$ $sCPA_3(4) = 0$	$sCPA_4(1) = 5$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 1$	$p_4 = 3$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (10,20,30)$	$\mathbf{w}_4 = (20,10,30)$

Snapshot 9

Cost(CPA) = 2	UpperBound = 8		ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 10$	X_4
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 8$ $sCPA_1(4) = 0$	$sCPA_2(1) = 35$ $sCPA_2(3) = 42$ $sCPA_2(4) = 0$	$sCPA_3(1) = 60$ $sCPA_3(2) = 25$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 1$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (30,10,20)$	$\boldsymbol{w}_3 = (10,20,30)$	

Snapshot 10

Cost(CPA) = 12 <	UpperBound = 8	= false		
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 20$	X_4
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 18$ $sCPA_1(4) = 0$	$sCPA_2(1) = 35$ $sCPA_2(3) = 60$ $sCPA_2(4) = 0$	$sCPA_3(1) = 51$ $sCPA_3(2) = 16$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 2$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (30,10,20)$	$\boldsymbol{w}_3 = (10,20,30)$	

Snapshot 11

Cost(CPA) = 7	<	UpperBound = 8	= true	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 30$	X_4
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 26$ $sCPA_1(4) = 0$	$sCPA_2(1) = 35$ $sCPA_2(3) = 39$ $sCPA_2(4) = 0$	$sCPA_3(1) = 42$ $sCPA_3(2) = 33$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 3$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (10,20,30)$	

Snapshot 12

Cost(CPA) = 17	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 30$	$X_4 = 30$
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 26$ $sCPA_1(4) = 8$	$sCPA_2(1) = 35$ $sCPA_2(3) = 39$ $sCPA_2(4) = 0$	$sCPA_3(1) = 42$ $sCPA_3(2) = 33$ $sCPA_3(4) = 0$	$sCPA_4(1) = 2$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 3$	$p_4 = 1$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (10,20,30)$	$\mathbf{w}_4 = (30,20,10)$

Snapshot 13

Cost(CPA) = 13	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 30$	$X_4 = 20$
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 26$ $sCPA_1(4) = 4$	$sCPA_2(1) = 35$ $sCPA_2(3) = 39$ $sCPA_2(4) = 0$	$sCPA_3(1) = 42$ $sCPA_3(2) = 33$ $sCPA_3(4) = 0$	$sCPA_4(1) = 2$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 3$	$p_4 = 2$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (10,20,30)$	$\mathbf{w}_4 = (30,20,10)$

Snapshot 14

Cost(CPA) = 14	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 30$	$X_4 = 10$
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 26$ $sCPA_1(4) = 14$	$sCPA_2(1) = 35$ $sCPA_2(3) = 39$ $sCPA_2(4) = 0$	$sCPA_3(1) = 42$ $sCPA_3(2) = 33$ $sCPA_3(4) = 0$	$sCPA_4(1) = 60$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 3$	$p_4 = 3$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (10,20,30)$	$\mathbf{w}_4 = (30,20,10)$

Snapshot 15

Cost(CPA) = 7		UpperBound = 8	ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	$X_3 = 30$	X_4
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 26$ $sCPA_1(4) = 0$	$sCPA_2(1) = 35$ $sCPA_2(3) = 39$ $sCPA_2(4) = 0$	$sCPA_3(1) = 42$ $sCPA_3(2) = 33$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 3$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (10,20,30)$	

Snapshot 16

Cost(CPA) = 1		UpperBound = 8	ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 30$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 33$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 35$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 1$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$		

Snapshot 17

Cost(CPA) = 7	<	UpperBound = 8	= true	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 10$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 34$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 40$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 2$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$		

Snapshot 18

Cost(CPA) = 15	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 10$	$X_3 = 20$	X_4
Shares of pairwise costs	$sCPA_1(2) = 34$ $sCPA_1(3) = 1$ $sCPA_1(4) = 0$	$sCPA_2(1) = 40$ $sCPA_2(3) = 60$ $sCPA_2(4) = 0$	$sCPA_3(1) = 1$ $sCPA_3(2) = 13$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 2$	$p_3 = 1$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (20,30,10)$	

Snapshot 19

Cost(CPA) = 15	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 10$	$X_3 = 30$	X_4
Shares of pairwise costs	$sCPA_1(2) = 34$ $sCPA_1(3) = 34$ $sCPA_1(4) = 0$	$sCPA_2(1) = 40$ $sCPA_2(3) = 1$ $sCPA_2(4) = 0$	$sCPA_3(1) = 34$ $sCPA_3(2) = 6$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 2$	$p_3 = 2$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (20,30,10)$	

Snapshot 20

Cost(CPA) = 8	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 10$	$X_3 = 10$	X_4
Shares of pairwise costs	$sCPA_1(2) = 34$ $sCPA_1(3) = 2$ $sCPA_1(4) = 0$	$sCPA_2(1) = 40$ $sCPA_2(3) = 20$ $sCPA_2(4) = 0$	$sCPA_3(1) = 66$ $sCPA_3(2) = 47$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 2$	$p_3 = 3$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$	$\mathbf{w}_3 = (20,30,10)$	

Snapshot 21

Cost(CPA) = 7		UpperBound = 8	ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 10$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 34$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 40$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 2$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$		

Snapshot 22

Cost(CPA) = 9	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	$X_2 = 20$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 5$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 4$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 3$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$		

Snapshot 23

Cost(CPA) = 0	UpperBound = 8		ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 20$	X_2	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 1$	$p_2 = 4$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$			

Snapshot 24

Cost(CPA) = 0 <	UpperBound = 8 = true			
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	X_2	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 2$	$p_2 = 4$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$			

Snapshot 25

Cost(CPA) = 10	<	UpperBound = 8	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 10$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 55$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 22$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 2$	$p_2 = 1$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (10,20,30)$		

Snapshot 26

Cost(CPA) = 4	<	UpperBound = 8	= true	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 1$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (10,20,30)$		

Snapshot 27

Cost(CPA) = 14	<	UpperBound = 8 = false		
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	$X_3 = 20$	X_4
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 2$ $sCPA_1(4) = 0$	$sCPA_2(1) = 1$ $sCPA_2(3) = 30$ $sCPA_2(4) = 0$	$sCPA_3(1) = 2$ $sCPA_3(2) = 43$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 1$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (10,20,30)$	$\boldsymbol{w}_3 = (20,10,30)$	

Snapshot 28

Cost(CPA) = 7	<	UpperBound = 8 = true		
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	$X_3 = 10$	X_4
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 2$ $sCPA_1(4) = 0$	$sCPA_2(1) = 1$ $sCPA_2(3) = 20$ $sCPA_2(4) = 0$	$sCPA_3(1) = 1$ $sCPA_3(2) = 47$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 2$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (10,20,30)$	$\boldsymbol{w}_3 = (20,10,30)$	

Snapshot 29

Cost(CPA) = 7	<	UpperBound = 8	= true	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	$X_3 = 10$	$X_4 = 30$
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 2$ $sCPA_1(4) = 32$	$sCPA_2(1) = 1$ $sCPA_2(3) = 20$ $sCPA_2(4) = 0$	$sCPA_3(1) = 1$ $sCPA_3(2) = 47$ $sCPA_3(4) = 0$	$sCPA_4(1) = 35$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 44$	$sUB_2 = 10$	$sUB_3 = 18$	$sUB_4 = 3$
OptimalSetting	20	30	10	20
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 2$	$p_4 = 1$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	$\mathbf{w}_4 = (30,10,20)$

Snapshot 30

Cost(CPA) = 7		UpperBound = 7	NEW_OPTIMUM_FOUND	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	$X_3 = 10$	$X_4 = 30$
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 2$ $sCPA_1(4) = 32$	$sCPA_2(1) = 1$ $sCPA_2(3) = 20$ $sCPA_2(4) = 0$	$sCPA_3(1) = 1$ $sCPA_3(2) = 47$ $sCPA_3(4) = 0$	$sCPA_4(1) = 35$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 2$	$p_4 = 1$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	$\mathbf{w}_4 = (30,10,20)$

Snapshot 31

Cost(CPA) = 17	< UpperBound = 7	= false		
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	$X_3 = 10$	$X_4 = 10$
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 2$ $sCPA_1(4) = 9$	$sCPA_2(1) = 1$ $sCPA_2(3) = 20$ $sCPA_2(4) = 0$	$sCPA_3(1) = 1$ $sCPA_3(2) = 47$ $sCPA_3(4) = 0$	$sCPA_4(1) = 1$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 2$	$p_4 = 2$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	$\mathbf{w}_4 = (30,10,20)$

Snapshot 32

Cost(CPA) = 14	< UpperBound = 7	= false		
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	$X_3 = 10$	$X_4 = 20$
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 2$ $sCPA_1(4) = 8$	$sCPA_2(1) = 1$ $sCPA_2(3) = 20$ $sCPA_2(4) = 0$	$sCPA_3(1) = 1$ $sCPA_3(2) = 47$ $sCPA_3(4) = 0$	$sCPA_4(1) = 66$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 2$	$p_4 = 3$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	$\mathbf{w}_4 = (30,10,20)$

Snapshot 33

Cost(CPA) = 7	UpperBound = 7		ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	$X_3 = 10$	X_4
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 2$ $sCPA_1(4) = 0$	$sCPA_2(1) = 1$ $sCPA_2(3) = 20$ $sCPA_2(4) = 0$	$sCPA_3(1) = 1$ $sCPA_3(2) = 47$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 2$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	

Snapshot 34

Cost(CPA) = 7 <	UpperBound = 7 = false			
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	$X_3 = 30$	X_4
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 1$ $sCPA_2(3) = 1$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 2$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 3$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	

Snapshot 35

Cost(CPA) = 4	UpperBound = 7		ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 20$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 1$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 2$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$		

Snapshot 36

Cost(CPA) = 0	< UpperBound = 7	= true		
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$		

Snapshot 37

Cost(CPA) = 13	<	UpperBound = 7	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	$X_3 = 20$	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 2$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 5$ $sCPA_2(4) = 0$	$sCPA_3(1) = 2$ $sCPA_3(2) = 4$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 1$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (10,20,30)$	$\boldsymbol{w}_3 = (20,10,30)$	

Snapshot 38

Cost(CPA) = 3	<	UpperBound = 7	= true	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	$X_3 = 10$	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 26$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 21$ $sCPA_2(4) = 0$	$sCPA_3(1) = 44$ $sCPA_3(2) = 46$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 2$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (10,20,30)$	$\boldsymbol{w}_3 = (20,10,30)$	

Snapshot 39

Cost(CPA) = 3	<	UpperBound = 7	= true	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	$X_3 = 10$	$X_4 = 30$
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 26$ $sCPA_1(4) = 30$	$sCPA_2(1) = 0$ $sCPA_2(3) = 21$ $sCPA_2(4) = 0$	$sCPA_3(1) = 44$ $sCPA_3(2) = 46$ $sCPA_3(4) = 0$	$sCPA_4(1) = 37$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 37$	$sUB_2 = 21$	$sUB_3 = 48$	$sUB_4 = 35$
OptimalSetting	30	20	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 2$	$p_4 = 1$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	$\mathbf{w}_4 = (30,20,10)$

Snapshot 40

Cost(CPA) = 3		UpperBound = 3	NEW_OPTIMUM_FOUND	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	$X_3 = 10$	$X_4 = 30$
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 26$ $sCPA_1(4) = 30$	$sCPA_2(1) = 0$ $sCPA_2(3) = 21$ $sCPA_2(4) = 0$	$sCPA_3(1) = 44$ $sCPA_3(2) = 46$ $sCPA_3(4) = 0$	$sCPA_4(1) = 37$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 2$	$p_4 = 1$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	$\mathbf{w}_4 = (30,20,10)$

Snapshot 41

Cost(CPA) = 10	<	UpperBound = 3	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	$X_3 = 10$	$X_4 = 20$
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 26$ $sCPA_1(4) = 4$	$sCPA_2(1) = 0$ $sCPA_2(3) = 21$ $sCPA_2(4) = 0$	$sCPA_3(1) = 44$ $sCPA_3(2) = 46$ $sCPA_3(4) = 0$	$sCPA_4(1) = 3$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 2$	$p_4 = 2$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	$\mathbf{w}_4 = (30,20,10)$

Snapshot 42

Cost(CPA) = 13	<	UpperBound = 3	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	$X_3 = 10$	$X_4 = 10$
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 26$ $sCPA_1(4) = 5$	$sCPA_2(1) = 0$ $sCPA_2(3) = 21$ $sCPA_2(4) = 0$	$sCPA_3(1) = 44$ $sCPA_3(2) = 46$ $sCPA_3(4) = 0$	$sCPA_4(1) = 5$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 2$	$p_4 = 3$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	$\mathbf{w}_4 = (30,20,10)$

Snapshot 43

Cost(CPA) = 3 UpperBound = 3 **ZERO_SHARE_MSG**

Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	$X_3 = 10$	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 26$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 21$ $sCPA_2(4) = 0$	$sCPA_3(1) = 44$ $sCPA_3(2) = 46$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 2$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	

Snapshot 44

Cost(CPA) = 5 < UpperBound = 3 = false

Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	$X_3 = 30$	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 2$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 3$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 3$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (10,20,30)$	$\mathbf{w}_3 = (20,10,30)$	

Snapshot 45

Cost(CPA) = 0	UpperBound = 3		ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	$X_2 = 30$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 2$	$p_2 = 3$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (10,20,30)$		

Snapshot 46

Cost(CPA) = 0	UpperBound = 3		ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 30$	X_2	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 2$	$p_2 = 4$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$			

Snapshot 47

Cost(CPA) = 0 < UpperBound = 3 = true

Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 10$	X_2	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 3$	$p_2 = 4$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$			

Snapshot 48

Cost(CPA) = 4 < UpperBound = 3 = false

Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 10$	$X_2 = 30$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 5$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 66$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 3$	$p_2 = 1$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$	$\boldsymbol{w}_2 = (30,10,20)$		

Snapshot 49

Cost(CPA) = 5	<	UpperBound = 3	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 10$	$X_2 = 10$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 62$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 10$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 3$	$p_2 = 2$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$		

Snapshot 50

Cost(CPA) = 6	<	UpperBound = 3	= false	
Agents	A_1	A_2	A_3	A_4
Variables	$X_1 = 10$	$X_2 = 20$	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 3$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 3$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 3$	$p_2 = 3$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\mathbf{w}_1 = (20,30,10)$	$\mathbf{w}_2 = (30,10,20)$		

Snapshot 51

Cost(CPA) = 0	UpperBound = 3		ZERO_SHARE_MSG	
Agents	A_1	A_2	A_3	A_4
Variables	X_1=10	X_2	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	30	30	10	30
Pointer	$p_1 = 3$	$p_2 = 4$	$p_3 = 4$	$p_4 = 4$
Random ordering	$\boldsymbol{w}_1 = (20,30,10)$			

Snapshot 52

Cost(CPA)	**UpperBound = 3**		**COMPLETE**	
Agents	A_1	A_2	A_3	A_4
Variables	X_1	X_2	X_3	X_4
Shares of pairwise costs	$sCPA_1(2) = 0$ $sCPA_1(3) = 0$ $sCPA_1(4) = 0$	$sCPA_2(1) = 0$ $sCPA_2(3) = 0$ $sCPA_2(4) = 0$	$sCPA_3(1) = 0$ $sCPA_3(2) = 0$ $sCPA_3(4) = 0$	$sCPA_4(1) = 0$ $sCPA_4(2) = 0$ $sCPA_4(3) = 0$
Shares of best known global cost	$sUB_1 = 56$	$sUB_2 = 21$	$sUB_3 = 23$	$sUB_4 = 37$
OptimalSetting	**30**	**30**	**10**	**30**
Pointer	$p_1 = 4$	$p_2 = 4$	$p_3 = 4$	$p_4 = 4$
Random ordering				

Acknowledgements

This work was partially supported by the Ariel Cyber Innovation Center in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office. The authors would also like to thank Vadim Levit for his help with running the experiments.

References

[1] P. Meseguer, J. Larrosa, Constraint satisfaction as global optimization, in: IJCAI, 1995, pp. 579–584.

[2] R. Maheswaran, M. Tambe, E. Bowring, J. Pearce, P. Varakantham, Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling, in: AAMAS, 2004, pp. 310–317.

[3] A. Farinelli, A. Rogers, A. Petcu, N. Jennings, Decentralised coordination of low-power embedded devices using the max-sum algorithm, in: AAMAS, 2008, pp. 639–646.

[4] F. Lezama, J. Palominos, A. Rodríguez-González, A. Farinelli, E. Munoz de Cote, Agent-based microgrid scheduling: An ICT perspective, Mobile Networks and Applications 24 (5) (2017) 1682–1698.

[5] K. Hirayama, M. Yokoo, Distributed partial constraint satisfaction problem, in: CP, 1997, pp. 222–236.

[6] A. Gershman, A. Meisels, R. Zivan, Asynchronous forward bounding for distributed COPs, Journal of Artificial Intelligence Research 34 (2009) 61–88.

[7] R. Mailler, V. Lesser, Solving distributed constraint optimization problems using cooperative mediation, in: AAMAS, 2004, pp. 438–445.

[8] P. Modi, W. Shen, M. Tambe, M. Yokoo, ADOPT: asynchronous distributed constraint optimization with quality guarantees, Artificial Intelligence 161 (2005) 149–180.

[9] A. Petcu, B. Faltings, A scalable method for multiagent constraint optimization, in: IJCAI, 2005, pp. 266–271.

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

[11] H. Katagishi, J. Pearce, Kopt: Distributed DCOP algorithm for arbitrary k-optima with monotonically increasing utility, in: DCR, 2007.

[12] B. Ottens, C. Dimitrakakis, B. Faltings, DUCT: An upper confidence bound approach to distributed constraint optimization problems, ACM Transactions on Intelligent Systems and Technology (TIST) 8 (5) (2017) 69.

[13] W. Zhang, G. Wang, Z. Xing, L. Wittenburg, Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks, Artificial Intelligence 161 (1-2) (2005) 55–87.

[14] T. Léauté, B. Faltings, Protecting privacy through distributed computation in multi-agent decision making, Journal of Artificial Intelligence Research 47 (2013) 649–695.

[15] M. Silaghi, D. Mitra, Distributed constraint satisfaction and optimization with privacy enforcement, in: IAT, 2004, pp. 531–535.

[16] M. Ben-Or, S. Goldwasser, A. Wigderson, Completeness theorems for non-cryptographic fault-tolerant distributed computation, in: STOC, 1988, pp. 1–10.

[17] R. Greenstadt, B. Grosz, M. Smith, SSDPOP: improving the privacy of DCOP with secret sharing, in: AAMAS, 2007, pp. 171:1–171:3.

[18] T. Grinshpoun, T. Tassa, P-SyncBB: A privacy preserving branch and bound DCOP algorithm, Journal of Artificial Intelligence Research 57 (2016) 621–660.

[19] T. Tassa, T. Grinshpoun, R. Zivan, Privacy preserving implementation of the Max-Sum algorithm and its variants, Journal of Artificial Intelligence Research 59 (2017) 311–349.

[20] T. Grinshpoun, T. Tassa, V. Levit, R. Zivan, Privacy preserving region optimal algorithms for symmetric and asymmetric DCOPs, Artificial Intelligence 266 (2019) 27–50.

[21] C. Kiekintveld, Z. Yin, A. Kumar, M. Tambe, Asynchronous algorithms for approximate distributed constraint optimization with quality bounds, in: AAMAS, 2010, pp. 133–140.

[22] J. Alwen, J. Katz, Y. Lindell, G. Persiano, A. Shelat, I. Visconti, Collusion-free multiparty computation in the mediated model, in: CRYPTO, 2009, pp. 524–540.

[23] J. Alwen, A. Shelat, I. Visconti, Collusion-free protocols in the mediated model, in: CRYPTO, 2008, pp. 497–514.

[24] J. Schneider, Lean and fast secure multi-party computation: Minimizing communication and local computation using a helper, in: SECRYPT, 2016, pp. 223–230.

[25] T. Tassa, T. Grinshpoun, A. Yanai, A privacy preserving collusion secure DCOP algorithm, in: IJCAI, 2019, pp. 4774–4780.

[26] A. Chechetka, K. Sycara, No-commitment branch and bound search for distributed constraint optimization, in: AAMAS, 2006, pp. 1427 – 1429.

[27] T. Grinshpoun, T. Tassa, A privacy-preserving algorithm for distributed constraint optimization, in: AAMAS, 2014, pp. 909–916.

[28] P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in: Eurocrypt, 1999, pp. 223–238.

[29] A. Shamir, How to share a secret, Commun. ACM 22 (11) (1979) 612–613.

[30] A. Yao, Protocols for secure computation, in: FOCS, 1982, pp. 160–164.

[31] I. Damgård, J. B. Nielsen, Scalable and unconditionally secure multiparty computation, in: CRYPTO, 2007, pp. 572–590.

[32] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, A. Nof, Fast large-scale honest-majority MPC for malicious adversaries, in: CRYPTO, 2018, pp. 34–64.

[33] T. Nishide, K. Ohta, Multiparty computation for interval, equality, and comparison without bit-decomposition protocol, in: PKC, 2007, pp. 343–360.

[34] A. Ben-Efraim, E. Omri, Concrete efficiency improvements for multiparty garbling with an honest majority, in: LATINCRYPT, 2017, pp. 289–308.

[35] H. Lipmaa, P. Rogaway, D. Wagner, Comments to NIST concerning AES-modes of operations: CTR-mode encryption (October 2000).

[36] J. Daemen, V. Rijmen, The Design of Rijndael: AES - The Advanced Encryption Standard, Information Security and Cryptography, Springer, 2002.

[37] A. Ben-Efraim, The Ben-Efraim-Omri protocol implementation, `https://github.com/cryptobiu/Protocols/tree/master/Concrete_Efficiency_Improvements_to_Multiparty_Garbling_with_an_Honest_Majority`.

[38] B. Lutati, I. Gontmakher, M. Lando, A. Netzer, A. Meisels, A. Grubshtein, AgentZero: A framework for simulating and evaluating multi-agent algorithms, in: Agent-Oriented Software Engineering, 2014, pp. 309–327.

[39] E. Sultanik, R. N. Lass, W. C. Regli, DCOPolis: a framework for simulating and deploying distributed constraint reasoning algorithms, in: AAMAS (demos), 2008, pp. 1667–1668.

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

[41] M. Yokoo, K. Hirayama, Algorithms for distributed constraint satisfaction: A review, Autonomous Agents and Multi-Agent Systems 3 (2) (2000) 185–207.

[42] Y. Deng, Z. Chen, D. Chen, W. Zhang, X. Jiang, AsymDPOP: complete inference for asymmetric distributed constraint optimization problems, in: IJCAI, 2019, pp. 223–230.

[43] D. Chen, Y. Deng, Z. Chen, W. Zhang, Z. He, HS-CAI: A hybrid DCOP algorithm via combining search with context-based inference, in: AAAI, 2020.

[44] F. Rossi, C. J. Petrie, V. Dhar, On the equivalence of constraint satisfaction problems., in: ECAI, 1990, pp. 550–556.

[45] F. Bacchus, X. Chen, P. Van Beek, T. Walsh, Binary vs. non-binary constraints, Artificial Intelligence 140 (1-2) (2002) 1–37.

[46] T. Grinshpoun, A. Grubshtein, R. Zivan, A. Netzer, A. Meisels, Asymmetric distributed constraint optimization problems, Journal of Artificial Intelligence Research 47 (2013) 613–647.

[47] T. Grinshpoun, When you say (DCOP) privacy, what do you mean? - categorization of DCOP privacy and insights on internal constraint privacy, in: ICAART, 2012, pp. 380–386.

[48] D. Chen, Y. Deng, Z. Chen, Z. He, W. Zhang, A hybrid tree-based algorithm to solve asymmetric distributed constraint optimization problems, Autonomous Agents and Multi-Agent Systems 34 (2) (2020) 1–42.

[49] R. Zivan, T. Parash, L. Cohen-Lavi, Y. Naveh, Applying Max-sum to asymmetric distributed constraint optimization problems, Autonomous Agents and Multi-Agent Systems 34 (1) (2020) 1–29.