ECAI 2016
G.A. Kaminka et al. (Eds.)
© 2016 The Authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/978-1-61499-672-9-1291

A Distributed Asynchronous Solver for Nash Equilibria in Hypergraphical Games [1]

Mohamed Wahbi and **Kenneth N. Brown** [2]

Abstract. Hypergraphical games provides a compact model of a network of self-interested agents, each involved in simultaneous subgames with its neighbors. The overall aim is for the agents in the network to reach a Nash Equilibrium, in which no agent has an incentive to change their response, but without revealing all their private information. Asymmetric Distributed constraint satisfaction (ADisCSP) has been proposed as a solution to this search problem. In this paper, we propose a new model of hypergraphical games as an ADisCSP based on a new global constraint, and a new asynchronous algorithm for solving ADisCSP that is able to find a Nash Equilibrium. We show empirically that we significantly reduce both message passing and computation time, achieving an order of magnitude improvement in messaging and in non-concurrent computation time on dense problems compared to state-of-the art algorithms.

1 Introduction

In many multi-agent problems, agents must interact with each other to achieve a global goal while maximising their own individual preferences. The *hypergraphical games* model [13] provides a compact representation of the problem, in which agent interactions are represented as normal-form strategic subgames, and the relationship topology between the agents is represented as a hypergraph. Each agent has a set of strategies, and a utility function specifying the agent's payoff under each possible combination of its own and neighboring agents' strategies in each subgame. A solution to a hypergraphical game is the selection of a strategy for each agent, such that the network is in equilibrium. Typically, the aim is to find a Nash Equilibrium (NE), in which no agent can improve its payoff by changing its strategy. To model more realistic problems, ϵ-approximate Nash Equilibria (ϵ-NE) are considered, in which no agent can improve its payoff by more than some minimum threshold ϵ.

Given the multi-agent setting, algorithms to compute solutions should be distributed, to avoid the need for agents to reveal potentially private information. Early work focused on identifying graph topologies which allowed polynomial-time solutions, based on algorithms for Bayesian Network inference. More recent approaches focus on arbitrary graphs with cyclic dependencies, and represent the problem as *asymmetric distributed constraint satisfaction* (ADisCSP), maintaining the individual utility functions as extensional table constraints [5]. ADisCSP allows agents to keep their strategic information private while optimizing their local utility, and allows the system to stabilize at an equilibrium by coordinating agents' decisions. However, for dense graphical games with large strategy sets, the encoding of the table constraints ([5]) becomes expensive in the number or size of messages that need to be exchanged.

Our contributions in this paper are as follows. We develop new approaches to finding approximate Nash Equilibria for hypergraphical games using the distributed constraint satisfaction framework. We develop a new model of a hypergraphical game as ADisCSP using a new global constraint, ϵ−`BRConstraint`, to represent an agent's requirement to find an approximate best strategy given the other decisions in its neighborhood. We introduce asymmetric asynchronous backtracking, AABT, a new algorithm for solving ADisCSP with global constraints using intelligent backtracking to avoid thrashing. AABT is then used to solve the problem of finding an ϵ-NE in hypergraphical games formulated as an ADisCSP. We compare the new model and algorithm empirically to previous state-of-the-art algorithms, and we show that we achieve significant reductions in non-concurrent computation time and an order of magnitude improvement in message passing.

The paper is organized as follows. Section 2 gives a brief overview of related works on game theory and distributed CSP. Section 3 introduces the necessary background, basic notation and terminology. We present our model of the problem of finding ϵ-NE in hypergraphical games as ADisCSP in Section 4. Section 5 introduces our new algorithm, AABT, for solving ADisCSP with global constraints, and we show our empirical results in Section 6.

2 Related Work

[8] introduced *graphical games*, a compact representation of n-player normal-form games and proposed NashTree, a dynamic programming algorithm for computing Nash equilibria in graphical games for which the underlying graph is a tree. NashTree consists of two phases: a table passing phase from the leaves to the root and an assignment passing phase from the root to the leaves. [15] proposed a constraint satisfaction generalization of NashTree for general graphical games using variable elimination. They transform the graphical game into a tree via triangulation, and then subsequently run NashTree algorithm on the resulting junction tree.

[12] introduced the NashProp algorithm, another generalization of NashTree for general graphical games based on belief propagation requiring no triangulation. In the table passing phase, NashProp proceeds in a series of rounds to maintain (generalized) arc consistency in the constraints network. In each round, every node will send a different binary-valued table to each of its neighbors in the graph. Each table represents the value combination of the sender and the receiver

[1] This work is funded by Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289.

[2] Insight Centre for Data Analytics, School of Computer Science and IT, University College Cork, Ireland
email: {mohamed.wahbi,ken.brown}@insight-centre.org

that the sender believes could be in an ϵ-NE. In the assignment passing phase, NashProp performs a synchronous search to find ϵ-NE [5, Section 5.1].

The pioneering algorithm for symmetric DisCSP was *asynchronous backtracking* (ABT) [18, 2]. ABT is an asynchronous algorithm executed autonomously by each agent, and is guaranteed to converge to a global consistent solution (or detect inconsistency) in finite time. [3] proposed two varieties of ABT for solving ADisCSP, namely ABT-2ph and ABT-1ph. ABT-2ph alternates the execution of ABT considering constraints in one direction following a total ordering on agents until the problem is solved or inconsistency is proved. ABT-1ph checks constraints asynchronously in both directions, by agents sending their proposed assignments to all their neighbors. In ABT-1ph, an agent only changes its assignment when it is inconsistent with a higher neighbor assignment. When it is inconsistent with a lower neighbor assignment, the conflict is reported to the lower neighbor to change its assignment. However, both algorithms were restricted to solving problems with binary constraints.

Grubshtein and Meisels proposed in [5] a model of graphical games as an ADisCSP with a unique private (global) table constraint for each agent. The table constraint contains all tuples (joint strategies) of the neighbors that satisfy the ϵ-NE condition. They also proposed *asynchronous Nash backtracking* (ANT), the first asynchronous algorithm for solving ADisCSP with global constraints capable of finding ϵ-NE. ANT is an extension of ABT-1ph that handles asymmetric global (non-binary) constraints. ANT achieves orders of magnitude improvements over NashProp. However, ANT only uses the global constraints as checkers. In addition its handling of global constraints produces chronological backtracks (thrashing), as all value assignments for agents in the constraint are considered, even if they are not the cause of the conflict. Recent developments in DisCSP have shown how to make more effective use of global constraints, exploiting their pruning power, and backjumping closer to the point of conflict [1, 16].

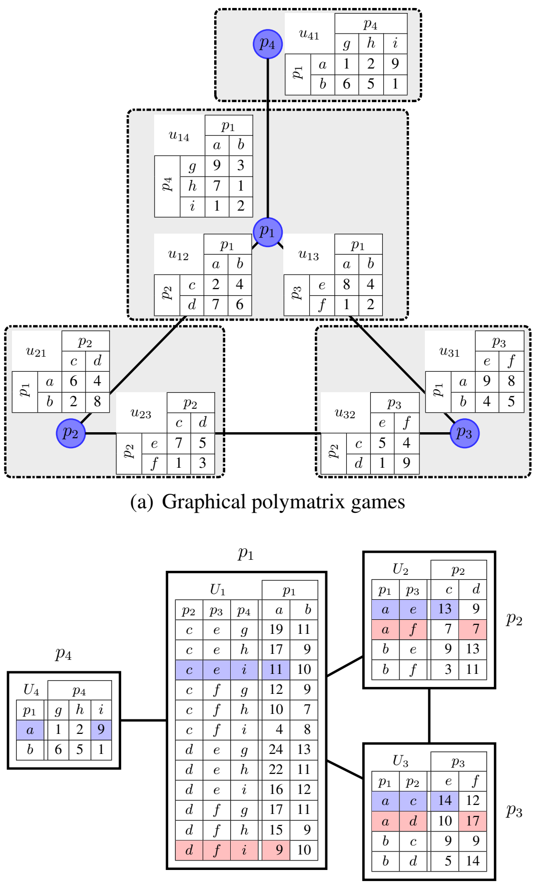
3 Preliminaries

In this section, we introduce some basic notation and terminology for game theory, and describe the framework of hypergraphical games before presenting the distributed constraint satisfaction formalism.

3.1 Game Theory

An n-player game in *normal-form* is a tuple $(\mathcal{P}, \{S_i, U_i\}_{p_i \in \mathcal{P}})$ where $\mathcal{P} = \{p_1, \ldots, p_n\}$ is a set of n players (agents). For each agent $p_i \in \mathcal{P}$, S_i is a finite set of actions or pure strategies, and $U_i : S \to \mathbb{R}$ where $S = \overset{n}{\underset{i=1}{\times}} S_i$, is a local utility hypermatrix/function that specifies the payoff for player p_i under each strategy profile $s \in S$. The joint strategic choice of all agents other than agent p_i is denoted by s_{-i}, and $s = (s_i, s_{-i})$. $U_i(s)$ is the utility that player p_i receives when, for all players $p_j \in \mathcal{P}$, p_j plays $s_j \in s$. The neighbors of agent p_i are all other agents in the normal-form game, i. e., $\mathcal{N}_i = \mathcal{P} \setminus p_i$. In the rest of the paper, we assume all agents have the same number of strategies, i. e., $\forall p_i \in \mathcal{P}, |S_i| = \mathrm{d}$. The representation size of each local utility hypermatrix is exponential in the number of players, i. e., $O(d^n)$. Motivated by scenarios where an agent's utility is directly dependent on only a subset of the total number of agents, researchers devoted a considerable effort to develop compact representations following the graphical game model [8].

A *hypergraph* is a pair (V, E) where V is a set of vertices, and E is a set of non-empty subsets of V called hyperedges. [13] introduced



u_{41}		p_4: g	h	i
p_1	a	1	2	9
	b	6	5	1

u_{14}		p_1: a	b
p_4	g	9	3
	h	7	1
	i	1	2

u_{12}		p_1: a	b
p_2	c	2	4
	d	7	6

u_{13}		p_1: a	b
p_3	e	8	4
	f	1	2

u_{21}		p_2: c	d
p_1	a	6	4
	b	2	8

u_{23}		p_2: c	d
p_2	e	7	5
	f	1	3

u_{31}		p_3: e	f
p_1	a	9	8
	b	4	5

u_{32}		p_3: e	f
p_2	c	5	4
	d	1	9

(a) Graphical polymatrix games

U_4	p_4: g	h	i
p_1			
a	1	2	9
b	6	5	1

U_1			p_1: a	b
p_2	p_3	p_4		
c	e	g	19	11
c	e	h	17	9
c	e	i	11	10
c	f	g	12	9
c	f	h	10	7
c	f	i	4	8
d	e	g	24	13
d	e	h	22	11
d	e	i	16	12
d	f	g	17	11
d	f	h	15	9
d	f	i	9	10

U_2		p_2: c	d
p_1	p_3		
a	e	13	9
a	f	7	7
b	e	9	13
b	f	3	11

U_3		p_3: e	f
p_1	p_2		
a	c	14	12
a	d	10	17
b	c	9	9
b	d	5	14

(b) Graphical game representation

Figure 1. An example of a hypergraphical game.

hypergraphical games, in which each agent p_i is involved in simultaneous (local) normal-form subgames, $\mathcal{G}_i$. A hypergraphical game is described by a hypergraph $(\mathcal{P}, E)$ where each hyperedge $h \in E$ represents an explicit subgame involving the players in $h \subseteq \mathcal{P}$. The strategy set of each agent p_i is the same in all subgames in $\mathcal{G}_i$. The payoff function of p_i is the sum of all p_i's payoffs in all $\mathcal{G}_i$. The neighbors of agent p_i, $\mathcal{N}_i$, is the union of its neighbors in all subgames in $\mathcal{G}_i$. The degree of agent p_i is denoted by $\kappa_i = |\mathcal{N}_i|$. The utility of an agent is directly dependent on its neighbors, i. e., $U_i : S_i \underset{p_j \in \mathcal{N}_i}{\times} S_j \to \mathbb{R}$. From now on, s_{-i} is the joint strategy of $\mathcal{N}_i$.

Hypergraphical games is a generalization of *graphical games* [8] where each agent is involved in exactly one subgame. In a graphical game, the representation size of p_i' utility is exponential in the degree of the agent $O(d^{\kappa_i})$. Hypergraphical games is also a generalization of *graphical polymatrix games* where each agent is involved in simultaneous 2-player games [6]. The representation size of p_i's utilities is $O(\kappa_i \cdot d^2)$.

In graphical polymatrix games, we can represent the utility function of agent p_i, U_i, by a set of (binary) utility functions u_{ij} for each $p_j \in \mathcal{N}_i$. The utility function $u_{ij}(s_i, s_j)$ represents the gain in utility of agent p_i when playing strategy $s_i \in S_i$ and agent p_j plays strategy $s_j \in S_j$, and p_i's overall utility function becomes:

$$U_i(s) = U_i(s_{-i}, s_i) = \sum_{p_j \in \mathcal{N}_i, s_j \in s} u_{ij}(s_i, s_j) \qquad (1)$$

For a strategy profile s, the *regret* $\delta_i(s)$ of an agent p_i is the highest additional reward p_i could have gained by changing its strategy, assuming its neighbors' strategy choices remain the same:

$$\delta_i(s) = \max_{s_i' \in S_i} \{U_i(s_{-i}, s_i') - U_i(s)\} \qquad (2)$$

In strategic games, each rational agent would ideally play its best strategy given a fixed joint strategy of other agents. A configuration in which all agents selected a best strategy is called a *Nash Equilibrium* (NE) . Specifically, a NE is a strategy profile s where each player's regret is 0 (i. e., $\forall p_i \in \mathcal{P}, \delta_i(s) = 0$). Given a joint strategy of other players s_{-i}, the *best response* of agent p_i, $BR_i(s_{-i})$, is the strategies which produces the maximal gain for agent p_i (i. e., $BR_i(s_{-i}) = \{s_i | \delta_i(s_{-i}, s_i) = 0\}$).

An *approximate Nash Equilibrium* (ϵ-NE) represents scenarios where agents are satisfied with choices whose payoff is sufficiently close to the maximum. Formally, a strategy profile s is an ϵ-NE if each player's regret is at most ϵ, i. e., $\forall p_i \in \mathcal{P}, \delta_i(s) \leq \epsilon$. Given a joint strategy of other players s_{-i}, the approximate best response set of agent p_i is defined as: ϵ-$BR_i(s_{-i}) = \{s_i | \delta_i(s_{-i}, s_i) \leq \epsilon\}$.

Figure 1 shows a simple hypergraphical game with 4 agents: p_1, p_2, p_3 and p_4 having the following strategy sets $S_1 = \{a, b\}$, $S_2 = \{c, d\}$, $S_3 = \{e, f\}$ and $S_4 = \{g, h, i\}$. Figure 1(a) shows the original representation in graphical polymatrix games and Figure 1(b) shows a representation of the same instance in graphical games. Agent p_1 is involved in three 2-player subgames with p_2, p_3 and p_4 represented respectively by the utilities u_{12}, u_{13}, and u_{14}. Agent p_2 is involved in two subgames with p_1 (u_{21}) and p_3 (u_{23}). Agent p_3 is involved in two subgames with p_1 (u_{31}) and p_2 (u_{32}). Agent p_4 is involved in one 2-player subgame with p_1 represented by utility u_{41}. This problem has one NE $[p_1 = a, p_2 = c, p_3 = e, p_4 = i]$ and one ϵ-NE in addition to the NE where $\epsilon = 3$, that is, $[p_1 = a, p_2 = d, p_3 = f, p_4 = i]$.

3.2 CSP & Asymmetric Distributed CSP

The *constraint satisfaction problem* (CSP) is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, where $D(x_i)$ is a finite set of values from which one value must be assigned to variable x_i, and $\mathcal{C}$ is a set of constraints. A constraint $c_k(X) \in \mathcal{C}$, on the ordered subset of variables $X = (x_{j_1}, \ldots, x_{j_k}) \subseteq \mathcal{X}$, is $c_k(X) \subseteq D(x_{j_1}) \times \cdots \times D(x_{j_k})$, and specifies the tuples of values which may be assigned simultaneously to the variables in X. $c_k(X)$ can be represented extensionally or intensionally. $|X|$ is the *arity* of $c_k(X)$, and X is its *scope*. A global constraint is defined on a set of variables and thus an instance of the constraint may have arbitrary arity. A *solution* is an assignment to each variable of a value from its domain, satisfying all the constraints.

Asymmetric distributed CSP (*ADisCSP*) [3] models problems where variables and constraints are held by distinct agents. ADisCSP is a 4-tuple $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}, \mathcal{D}$ and $\mathcal{C}$ are as above, and $\mathcal{A} = \{a_1, \ldots, a_m\}$ is a set of m agents. Each variable $x_i \in \mathcal{X}$ is controlled by a single agent in $\mathcal{A}$. During a solution process, only the agent which controls a variable can assign a value to this variable. In an ADisCSP, constraints are private and only the agent, a_i, that holds a constraint knows it while other agents involved in that constraint are only aware that a_i constrains their variable without knowing the nature of that constraint or its scope. We denote by $\mathcal{C}_i \subseteq \mathcal{C}$ all constraints held by a_i. As in CSP, a *solution* to an ADisCSP is an assignment to each variable of a value from its domain, satisfying all the constraints. For simplicity and without loss of generality, we assume each agent controls exactly one variable and use the two terms interchangeably (i. e., $m = n$).

4 Hypergraphical Games as ADisCSP

We now present a model of the problem of finding ϵ-NE in hypergraphical games as ADisCSP, where agents can control a local CSP that allows them to maintain the approximate best responses.

The straightforward modeling of a hypergraphical game $(\mathcal{P}, \{S_i, U_i\}_{p_i \in \mathcal{P}})$ into an ADisCSP is to represent each player $p_i \in \mathcal{P}$ by an agent $a_i \in \mathcal{A}$ having a single local variable x_i that can take its value from the strategy set of player p_i, i. e., $D(x_i) = S_i$. In the following we use the terms agent, player and variable interchangeably, (i. e., $a_k = p_k = x_k$). In addition, we consider a generic agent a_i. $\mathcal{X}_i = \{x_i, x_j | x_j \in \mathcal{N}_i\}$ denotes the variables (or copies) maintained by a_i. Agent a_i encodes the problem of finding an ϵ-NE (finding an ϵ-approximate best strategy) by a single constraint $c_i(\mathcal{X}_i)$ requiring that the regret, Eq. (2), is less than or equals ϵ, i. e., $c_i(\mathcal{X}_i) : \delta_i(s) \leq \epsilon$ where $s = (s_i, s_{-i})$ is a joint strategy (the assignments) of agents in $\mathcal{X}_i$. Thus, $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ where $\mathcal{C}_i = \{c_i(\mathcal{X}_i)\}$. The agent a_i that holds the constraint $c_i(\mathcal{X}_i)$ is the only agent that knows it, and must ensure it is satisfied, given the joint assignment of all its neighbors. To evaluate this constraint agent a_i needs to maintain local copies of its neighbors' variables in $\mathcal{X}_i$. Finding ϵ-NE distributively in a hypergraphical game is then equivalent to solving the ADisCSP model above.

Proposition 1. *Let $\mathcal{M}$ be an ADisCSP model of a hypergraphical game $\mathcal{H}$. A solution of $\mathcal{M}$ is an ϵ-NE of $\mathcal{H}$.*

Proof. (Sketch) A solution to $\mathcal{M}$ is an assignment to each variable of a value from its domain, satisfying all the constraints. Thus, each player is assigned a strategy and every agent a_i's private constraint, $c_i(\mathcal{X}_i)$, is satisfied. Satisfying each constraint $c_i(\mathcal{X}_i)$ means that each agent regret is less than ϵ ($\delta_i(s) \leq \epsilon$). Thus, each agent assignment is an approximate best response to its neighbors assignments. Hence, a solution of $\mathcal{M}$ is an ϵ-NE of $\mathcal{H}$. □

If agents are allowed mixed strategies (i.e. a probability distribution over deterministic strategies), then the discretization scheme proposed in [8] guarantees that an ϵ-NE always exists. Thus, the ADisCSP model of the hypergraphical game is always satisfiable. For more details about mixed strategies and the existence of an ϵ-NE, we refer the reader to [8, 12, 11, 5]. If the agents are restricted to pure strategies, it is possible that no ϵ-NE exists. In such circumstances, the ADisCSP model would have no solution, and any algorithm should report failure.

In the following we propose ϵ−`BRConstraint`, a new incremental global constraint to ensure an agent's value is an approximate best response, that is efficient in memory and allows efficient propagation. The ϵ−`BRConstraint` enforces $\delta_i(s) \leq \epsilon$ for each agent a_i (i. e., it is an implementation of $c_i(\mathcal{X}_i)$). For simplicity, we restrict our attention to graphical polymatrix games. However, our constraint and algorithm will represent any hypergraphical game including graphical games and graphical polymatrix games. Our experiments are against the state-of-the-art distributed algorithm for graphical games, and we experiment with the same class of problems that that algorithm was tested on. These problems happen to be graphical polymatrix games.

The Constraint ϵ−`BRConstraint`

When the system is at an ϵ-NE, each agent has assigned to its variable an approximate best strategy with respect to its neighbors assignments. Thus, at each step in the distributed algorithm, an agent a_i should prune all strategies that are dominated by others. In agent

a_i, ϵ−`BRConstraint` filters the domain of x_i to prune dominated strategies with respect to the (subset of) decisions of other agents in $\mathcal{N}_i$. ϵ−`BRConstraint`$(\epsilon, x_i, \{x_j\}_{x_j \in \mathcal{N}_i}, \{u_{ij}\}_{x_j \in \mathcal{N}_i})$ takes as parameters all variables in $\mathcal{X}_i$ and utility functions u_{ij} in addition to ϵ.[3] Before presenting the constraint behavior we first consider the auxiliary variables used inside ϵ−`BRConstraint` to filter dominated strategies. In addition to x_i and copies of neighbors' variables $x_j \in \mathcal{N}_i$, a_i has in $\mathcal{X}_i$ the following integer variables:

- $v_j[s_i]$ for each strategy $s_i \in D(x_i)$, and each utility u_{ij}. The variable $v_j[s_i]$ represents the gain in utility of agent a_i when playing strategy s_i with respect to x_j possible strategies. This variable ranges over the possible utilities in u_{ij} when a_i plays s_i, i. e., $D(v_j[s_i]) = \{u_{ij}(s_i, s_j) | s_j \in D(x_j)\}$.
- $y[s_i]$ for each strategy $s_i \in D(x_i)$. The variable $y[s_i]$ is used to maintain the minimum reward gained when a_i chooses to play strategy s_i.
$$y[s_i] = \sum_{x_j \in \mathcal{N}_i} \min\{D(v_j[s_i])\} \qquad (3)$$
- $z[s_i]$ for each strategy $s_i \in D(x_i)$. The variable $z[s_i]$ is used to maintain the maximum reward gained when a_i chooses to play strategy s_i.
$$z[s_i] = \sum_{x_j \in \mathcal{N}_i} \max\{D(v_j[s_i])\} \qquad (4)$$

For each $s_i \in D(x_i)$, we need to represent the relation between the values of the variables x_i, x_j and $v_j[s_i]$, and maintain $v_j[s_i] = u_{ij}(s_i, s_j)$. As soon as utility $v_j[s_i]$ is updated, some now inconsistent values of x_j can be removed from consideration. Similarly, when $D(x_j)$ is updated (e. g., a value is removed from $D(x_j)$), the utility variable $v_j[s_i]$ can be updated correspondingly. Each time a domain of x_j or $v_j[s_i]$ is changed, the domain of the other variable is updated to keep only values having a support on other variables, i. e., we need to ensure that $\exists\, s_j \in D(x_j) \wedge r \in D(v_j[s_i])$, $r = u_{ij}(s_i, s_j)$.[4]

We need also to make the correspondence between the values of $v_j[s_i]$ and those of $y[s_i]$ and $z[s_i]$ for each $s_i \in D(x_i)$ by ensuring Eq. (3) and Eq. (4). We maintain a support for $y[s_i]$ (resp. $z[s_i]$) on variable $v_j[s_i]$ and we only update that support and its corresponding reward on $y[s_i]$ (resp. $z[s_i]$) when the lower bound (resp. the upper bound) of the domain of $v_j[s_i]$ has changed.

By maintaining the above variables and properties on their domain changes, ϵ−`BRConstraint` is able to detect and remove from $D(x_i)$ the dominated strategies using the $y[s_i]$ and $z[s_i]$ variables. We say that a strategy $s_i \in D(x_i)$ is *dominated* if the largest utility that a_i can gain when choosing strategy s_i (i. e., $z[s_i]$) is lower than the minimal reward that a_i can gain by choosing another strategy s'_i where $s'_i = \arg\max_{s_i \in D(x_i)} \{y[s_i]\}$ taking ϵ in consideration. Specifically, a strategy $s_i \in D(x_i)$ is dominated iff:
$$z[s_i] < \max_{s'_i \in D(x_i)} \{y[s'_i]\} - \epsilon \qquad (5)$$
Removing dominated strategies s_i from $D(x_i)$ is safe because a_i will never play s_i, Eq. (5). Filtering in ϵ−`BRConstraint` is based on pruning all dominated strategies each time they are detected with changes on domains of variables in $\mathcal{X}_i$ without needing their full joint strategy (assignments).

[3] In the general hypergraphical games, u_{ij} will be replaced by payoffs matrix of each subgame.

[4] Maintaining $v_j[s_i]$ values follows the same scheme as the `element` constraint [14]. However, in the general case the index variable is a combination of indexes of all neighborhood values in the subgame.

x_2	x_3	x_4	x_1
c	e	g	b
c	e	h	b
c	e	i	b
c	f	g	b
c	f	h	b
c	f	i	a
d	e	g	b
d	e	h	b
d	e	i	b
d	f	g	b
d	f	h	b
d	f	i	a

(a) Table constraint encoding $c_1(\mathcal{X}_1)$ used in [5].

$D(v_2[a]) = \{2, 7\}$
$D(v_2[b]) = \{4, 6\}$ $\quad D(y[a]) = \{4..24\}$
$D(v_3[a]) = \{1, 8\}$ $\quad D(y[b]) = \{7..13\}$
$D(v_3[b]) = \{2, 4\}$ $\quad D(z[a]) = \{4..24\}$
$D(v_4[a]) = \{1, 7, 9\}$ $\quad D(z[b]) = \{7..13\}$
$D(v_4[b]) = \{1, 2, 3\}$

$v_2[a]$	$= u_{12}(a, s_2)$	$= [2, 7]$
$v_2[b]$	$= u_{12}(b, s_2)$	$= [4, 6]$
$v_3[a]$	$= u_{13}(a, s_3)$	$= [8, 1]$
$v_3[b]$	$= u_{13}(b, s_3)$	$= [4, 2]$
$v_4[a]$	$= u_{14}(a, s_4)$	$= [9, 7, 1]$
$v_4[b]$	$= u_{14}(b, s_4)$	$= [3, 1, 2]$

(b) ϵ-`BRConstraint` for $c_1(\mathcal{X}_1)$.

Figure 2. The encoding of ANT and AABT in a_1 of the constraint $c_1(\mathcal{X}_1)$ in the problem shown in Figure 1.

4.1 Memory requirements

In a hypergraphical game, in [5], $c_i(\mathcal{X}_i)$ is represented in extensional form using a table constraint containing all tuples (joint strategy) $s \in D(x_i) \bigtimes_{x_j \in \mathcal{N}_i} D(x_j)$ that satisfies $c_i(\mathcal{X}_i)$. A tuple s satisfies $c_i(\mathcal{X}_i)$ if it yields a maximal gain to agent a_i, i. e., $\delta_i(s) \leq \epsilon$. Thus, the utility functions of all subgames of a_i are encoded and represented by a large table constraint requiring a memory size of $O(d^{\kappa_i+1})$. The table constraint encoding the constraint $c_1(\mathcal{X}_1)$ of agent a_i of the example presented in Figure 1 is shown in Figure 2(a). All tuples satisfying $c_1(\mathcal{X}_1)$ are represented in that table.

In our new model, ϵ−`BRConstraint` maintains a_i's utility in each subgame in $\mathcal{G}_i$. Thus, the representation size in ϵ−`BRConstraint` is similar to that required by the hypergraphical game in each agent, i. e., $O(|\mathcal{G}_i| \cdot d^{\gamma+1})$ where γ is largest neighborhood in subgames $\mathcal{G}_i$ and $|\mathcal{G}_i|$ is the number of subgames in $\mathcal{G}_i$. In polymatrix games, this representation is polyspace $O(\kappa_i \cdot d^2)$. The encoding of the constraint $c_1(\mathcal{X}_1)$ of agent a_i of the example presented in Figure 1 is shown in Figure 2(b).

In ANT the handling of the global constraint does not exploit its filtering power. In addition, when a dead-end occurs, a no-good is produced from all neighbors' assignments and then sent to the lowest neighbor in the ordering. This produces chronological backtracks (thrashing). In the following, we propose a new algorithm for solving ADisCSP with global constraints, that exploits the pruning power of global constraints and produces no-goods closer to the (real) point of conflict.

5 Asymmetric Asynchronous BackTracking

Asymmetric asynchronous backtracking (AABT) is an asynchronous algorithm for solving ADisCSP that allows agents to keep their constraints private. In AABT agents operate asynchronously, but are subject to a known total priority order. AABT combines a distributed search procedure with a failure learning mechanism to perform intelligent backtracking. Intelligent backtracking techniques usually store an explanation for each value removal. Such explanations are computed on-the-fly on each domain reduction.

In AABT, each agent a_i tries to solve its local CSP_i defined by $\mathcal{X}_i$, their domains, and $\mathcal{C}_i$. Solving a CSP_i is achieved by interleaving search with propagation. The search can be regarded as the dy-

Algorithm 1: AABT algorithm running by agent a_i.

```
procedure AABT()
1.  $E_i^+ \leftarrow \{a_j \mid a_i \in \mathcal{N}_j\}$;  $end \leftarrow$ false;  $x_i \leftarrow nil$ ;
2.  assignVariable() ;
3.  while ( $\neg end$ ) do
4.      $msg \leftarrow$ getMsg();
5.      switch ( $msg.type$ ) do
6.          ok? : processOk($msg.var$, $msg.tag$);
7.          ngd : processNogood($msg.sender$, $msg.ngd$);
8.          adl : processAddLink($msg.sender$);
9.          stp : $end \leftarrow$ true;

procedure assignVariable()
10. while ( $x_i = nil \wedge \neg end$ ) do
11.     propagate($x_i = D(x_i).peek()$);
12.     if ( $\exists\, x_k \in \mathcal{X}_i \mid D(x_k) = \emptyset$ ) then
13.         repair() ; // An empty domain has been found
14.     else
15.         $t_i \leftarrow t_i$+1 ;
16.         sendMsg: ok?$\langle x_i = s_i, t_i \rangle$ to $E_i^+$ ;

procedure propagate($x_k = s'_k$)
17. remove($expl(x_l \neq s_l)$) s.t. $x_k = s_k \in expl(x_l \neq s_l)$ ;
18. $D(x_k) \leftarrow \{s'_k\}$ s.t. $expl(x_k \neq s_k) \leftarrow x_k = s'_k$;
19. $\mathcal{C}_i$.propagate();

procedure processOk($x'_j, t'_j$)
20. if ( $t'_j \geq t_j$ ) then
21.     propagate($x_j = x'_j$);
22.     if ( $\exists\, x_k \in \mathcal{X}_i \mid D(x_k) = \emptyset$ ) then repair();
23.     assignVariable() ;

procedure processNogood($a_j$, $ng$)
24. if ( $\forall x_l \in \{ng \cap \mathcal{X}_i\}, ng[x_l] = \mathcal{X}_i[x_l]$ ) then
25.     $Links \leftarrow \{ng \cup \mathcal{X}_i\} \setminus \mathcal{X}_i$ ;
26.     $\mathcal{X}_i \leftarrow \mathcal{X}_i \cup Links$ ;
27.     sendMsg: adl$\langle\rangle$ to $Links$ ;
28.     learn($ng$);
29. else if ( $ng[x_i] = x_i$ ) then
30.     sendMsg: ok?$\langle x_i = s_i, t_i \rangle$ to $a_j$;

procedure repair()
31. $ng \leftarrow \bigwedge_{s_k \in D(x_k)} expl(x_k \neq s_k)$ ;          /* $D(x_k) = \emptyset$ */
32. if ( $ng = \emptyset$ ) then
33.     sendMsg: stp$\langle\rangle$ to $\mathcal{A} \setminus a_i$ ;
34.     $end \leftarrow$ true;
35. else learn($ng$) ;

procedure learn($ng$)
36. Let $x_t$ be the lowest variable in $ng$ ;
37. if ( $x_t \neq x_i$ ) then sendMsg: ngd$\langle ng \rangle$ to $a_t$ ;
38. $expl(x_t \neq s_t) \leftarrow \{ng \setminus x_t = s_t\}$ ;
39. remove($expl(x_l \neq s_l)$) s.t. $x_t = s_t \in expl(x_l \neq s_l)$ ;
40. $\mathcal{C}_i$.propagate();
41. if ( $\exists\, x_k \in \mathcal{X}_i \mid D(x_k) = \emptyset$ ) then repair();
42. assignVariable() ;

procedure processAddLink($a_j$)
43. $E_i^+ \leftarrow E_i^+ \cup a_j$ ;
44. sendMsg: ok?$\langle x_i = s_i, t_i \rangle$ to $a_j$ ;
```

namic addition of constraints (decision constraints) to $\mathcal{C}_i$ and retractions (backtracks) [7]. For simplicity, we restrict ourselves to decision constraints that are of the form $x_k = s_k$, i. e., assignments of values to variables.[5] Each assignment is followed by the propagation of $\mathcal{C}_i$ with respect to $\mathcal{X}_i$'s domains. This propagation may result in a value removal ($x_k \neq s_k$) for a variable $x_k \in \mathcal{X}_i$. We define an *explanation*, $expl(x_k \neq s_k)$, of that value removal (i. e., $x_k \neq s_k$) by the set of decision constraints $x_j = s_j, \ldots, x_l = s_l$ (assignments), such that ($x_j = s_j \wedge \ldots \wedge x_l = s_l \wedge x_k = s_k$) is globally inconsistent. During search, a failure (a domain wipeout) may occur leading to *no-goods* computations. A no-good can be regarded as a subset of the assignments made so far that caused a failure (i. e., $\bigwedge_{s_k \in D(x_k)} expl(x_k \neq s_k)$ where $D(x_k) = \emptyset$).

AABT agents exchange the following message types:

ok?: used to notify its recipients of a new assignment associated with the current counter t_i of agent a_i, used to discard obsolete assignments.

ngd: used to report a no-good to another agent, requesting the removal of its value.

adl: used to request the additional of a link to the receiver.

The pseudo-code of AABT executed by each agent a_i is presented in Algorithm 1. In the following, s_i will represent the current value assigned to x_i and t_i the counter tagging s_i used for the timestamp mechanism. "$x_i = nil$" means that x_i is unassigned. After initialization, each agent assigns a value and informs all agents in E_i^+ of its decision (`assignVariable` call, line 2) by sending them **ok?** messages. E_i^+ are agents having a constraint involving a_i's variable (line 1). Then, a loop considers the reception of the possible message types. If no message is traveling through the network, the state of quiescence is reached meaning that a global solution is found. The solution is given by the current variables' assignments. The quiescence state can be detected by a specialized algorithm [4].

When an agent a_i receives an **ok?** message from a_j it calls procedure `processOk` (line 6). If the received assignment has a larger counter than that received beforehand, it is accepted, otherwise it is discarded (line 20). If the assignment is accepted, a_i calls procedure `propagate` to propagate $\mathcal{C}_i$ after adding the newly received assignment as a decision constraint $x_j = s'_j$, line 21. If the propagation results in a domain wipeout procedure `repair` is called to resolve the conflict, line 22. Finally, `assignVariable` is called (line 23) to assign a new consistent value to x_i if its value has been pruned by the propagation of $\mathcal{C}_i$.

When calling procedure `propagate`($x_k = s'_k$), the value of x_k is set to s'_k. Next, all explanations not relevant to this new assignment, i. e., $expl(x_k \neq s_k)$ containing $x_k = s_k$ where $s'_k \neq s_k$, are removed (line 17).[6] Then, the domain of x_k is reduced to a singleton $D(x_k) = \{s'_k\}$ with $x_k = s'_k$ as explanation, line 18. Finally, the constraints in $\mathcal{C}_i$ that might be affected by the domains' changes above are propagated, line 19.

When every value (s_k) of a variable $x_k \in \mathcal{X}_i$ is ruled out by an explanation $expl(x_k \neq s_k)$, the procedure `repair` is called to resolve the conflict (lines 13, 22 and 41). The conflict is resolved by computing a new no-good ng from the conjunction of these explanations, i. e., $expl(x_k \neq s_k)$, line 31. If the new no-good ng is empty, a_i terminates execution after sending a **stp** message to all agents in the system meaning that the problem is unsolvable (line 32). Otherwise,

[5] Agent a_i can not take decision for other agents variables x_j, i. e., $i \neq j$.

[6] The removal of some explanations $expl(x_k \neq s_k)$ does not imply the restoration of s_k to $D(x_k)$ unless the propagation of constraints involving x_k allows that.

it calls procedure `learn`(ng), line 35. Let x_t be the variable having the lowest priority in ng. If x_t is different than x_i, ng is reported to a_t through a **ngd** message, lines 36 and 37. In AABT, the backtracking target x_t is always that having the lowest priority in ng. Agent x_t can be a higher (as in ABT) but also a lower priority agent. Thus, we guarantee that the conflict is always reported to the agent with lowest priority in the conflict. Next, a new explanation from ng is used to justify the removal of the value of x_t, i. e., $expl(x_t{\neq}s_t) \leftarrow \{ng \setminus x_t = s_t\}$, line 38. All explanations containing the assignment of x_t ($x_t = s_t$) are removed because they are not valid anymore, line 39. The constraints in $\mathcal{C}_i$ are then propagated to check the new changes and if a failure occurs again `repair` is called, line 41. Finally, `assignVariable` is called to check if x_i needs to be assigned (line 42).

When a **ngd** message is received (line 7), a_i checks the validity of the received no-good (procedure `processNogood` call). If the assignments of the received no-good are consistent with those stored locally, this no-good is valid (line 24). Then, agent a_i sends a link request to non linked agents having variables in ng (lines 25 to 27). Next, a_i calls procedure `learn`(ng), line 28. If the ng is not valid but is consistent with the current assignment of x_i, a_i sends an **ok?** message to the generator of ng, lines 29 and 30.

When receiving an **adl** message (lines 43 and 44) a_i adds a new link to the request sender (a_j) and sends an **ok?** message to inform a_j of its assignment.

5.1 Privacy

In AABT, to use the power of the filtering algorithms of global constraints, each agent maintains a copy of its neighborhood domains. However, agents in AABT only require to know the initial domain of their neighbors and do not need their actual domain. For the case of the hypergraphical game as ADisCSP, an agent can construct the domain of a neighbor based on the payoff matrices of local subgames. At each stage of AABT, the privacy of agents' current domains is preserved. In AABT, agents exchange their values with their neighbors. Thus, in AABT there is no privacy of assignments. In AABT, agents do not have to share their constraints/scopes to solve the problem. Thus, AABT preserves privacy of constraints and privacy of agents topologies. However, during the solving process some information is exchanged (value assignments and no-goods) between agents, thus leaking information about agents' constraints.

A method to evaluate the constraint privacy using entropy as a quantitative measure for privacy loss has been proposed in [10, 3]. This method measures the percentage of the conflicts in a table constraint held by an agent that are revealed to another agent involved in a conflict induced from **ok?** and **ngd** messages. However, this method only applies to ADisCSP with binary table constraints. Thus, this method can not be used to evaluate the privacy loss in AABT. We believe that studying the privacy loss in distributed algorithms for solving DisCSP with global constraints (e. g., ANT and AABT) is an open research area that needs to be investigated.

5.2 Theoretical Analysis

Here we prove that AABT is sound, complete and terminates.

Theorem 1. *AABT terminates.*

Proof. AABT stops its execution in two cases: when an empty no-good has been generated meaning that the problem is unsolvable or when the network reaches a quiescent state reporting a solution. To prove that AABT terminates, we need to prove that AABT reaches one of these two cases in finite time, i. e., agents can never fall into an infinite loop cycling among their possible values. In the following, we prove by induction on the agent ordering that agents can never fall into an infinite loop. Let assume a lexicographic ordering on agents. The base case for induction ($i = 1$) is obvious. Unlike ABT, a_1 can receive **ok?** messages from its neighbors (having a lower priority). When receiving these **ok?** messages a_1 may generate new no-goods as results of propagating its constraints. Agent a_1 may generate three categories of no-goods. The first are empty no-goods. This category stops the algorithm execution. The second are no-goods containing a lower neighbor. Those no-goods are transmitted to the lowest agent involved in. The third category are singleton no-goods containing a value of a_1. Agent a_1 may also receive **ngd** messages from lower priority agents. Now, all no-goods contained in **ngd** messages a_1 receives are singleton because in AABT the generated no-good is always sent to the agent having the lowest priority in it (lines 36 and 37). Hence, when agent a_1 proposes a possible value, it will not change it unless it receives or itself produces singleton no-goods ruling out this value. Values in singleton no-goods are removed once and for all from the domain of a_1. Because its domain is finite, a_1 cannot fall into an infinite loop.

Now, assume that agents higher that agent a_i ($i > 2$), i. e., a_1 to a_{i-1} ($i > 2$), are in a stable state, i. e., they are all assigned values to their variables and do not change their values. In the following, we show that agent a_i never falls into an infinite loop. After processing **ok?** and **ngd** messages it receives, agent a_i may generate contradictions (no-goods) as results of propagating its local constraints. The categories of no-goods agent a_i may generate are: (i) empty no-goods, (ii) no-goods containing lower neighbors, (iii) no-goods containing the agents a_1 to a_{i-1}, and (iv) no-goods containing the agents a_1 to a_i. No-goods of (i) causes the algorithm to stop, those of (ii) are forwarded to a lower agent and can not cause a_i to fall in infinite loop. No-goods of (iii) breaks our assumption because they are forwarded to a higher agent that we assumed to be in a stable state, causing it to change its value. Thus, only (iv) may affect the termination of a_i. Agent a_i may also receive **ngd** messages from other agents (higher or lower). However, all no-goods contained in **ngd** messages a_i receives contain only the agents a_1 to a_i, i. e., category (iv), because in AABT no-goods are always sent to the lowest agent involved in. Since agents a_1 to a_{i-1} are in a stable state, these no-goods are valid for a_i (the assignments on these no-goods are consistent with those stored locally).[7] Thus, a_i will remove its value and will not assign it again while at least one of agents a_1 to a_{i-1} does not change its value. Because its domain is finite, a_i will either eventually change its assignment with a different value or exhaust the possible values and send a no-good to a higher agent, i. e., one of $a_1 \ldots a_{i-1}$. This no-good will cause an agent that we assumed to be in a stable state, not to be in a stable state. Hence, agent a_i can never fall into an infinite loop for a given stable state of a_1 to a_{i-1}. By induction we have that agents can never fall into an infinite loop and AABT is thus guaranteed to terminate. □

A global solution is reported when the network has reached quiescence, meaning that all agents are idle and no message is transmitting through the network. To prove that AABT is sound, we should first establish the two following lemmas about when the network reaches the quiescent state σ.

[7] Note here that we discuss only no-goods consistent with a_1 to a_i, because inconsistent ones are discarded.

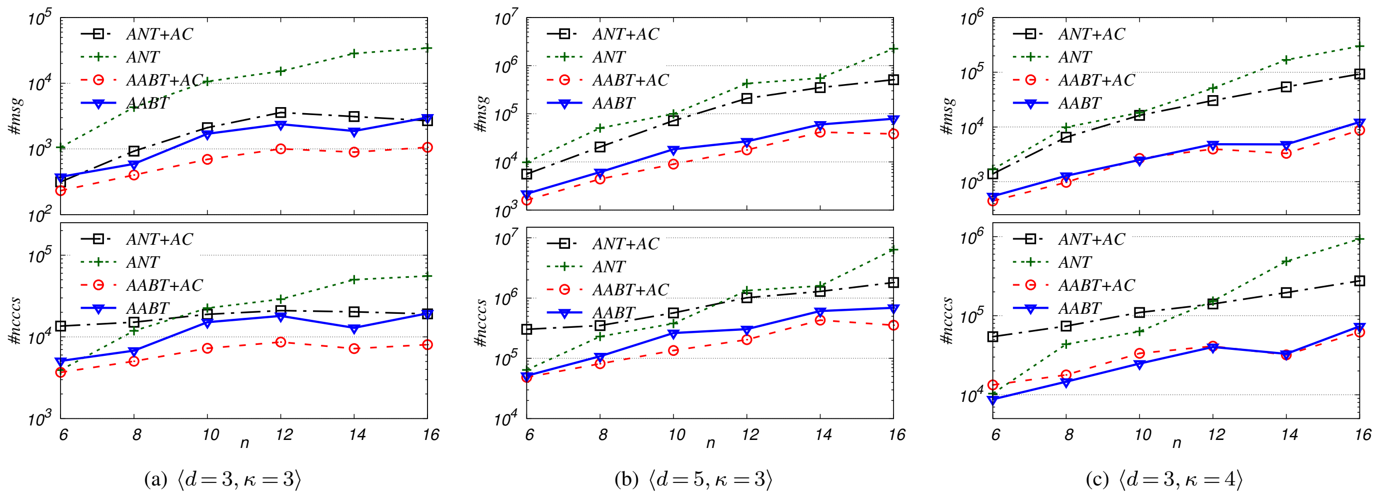


Figure 3. The $\#msg$ and $\#ncccs$ for solving random regular graphical polymatrix games when varying the number of agents in logarithmic scale.

Lemma 1. *When reaching a quiescent state σ, every agent is assigned, and its assignment is known by all its neighbors.*

Proof. (Sketch) Assume the system reaches a quiescent state σ. According to Algorithm 1, each time an agent a_i is unassigned, it immediately calls procedure `assignVariable` (lines 23 and 42). Before exiting its loop and being idle allowing the system to reach σ, `assignVariable` guarantees that either (i) an empty domain has been found when calling `repair` (line 13) meaning that the problem is unsolvable or (ii) a value is assigned to x_i. (i) contradicts our assumption. Thus, when reaching σ every agent a_i is assigned. Now, let v_i be the value assigned to x_i when reaching σ. Each neighbor of a_i, say a_j, should receive an **ok?** message from a_i containing its decision, i. e., $x_i = v_i$ (line 16). The only case where agent a_j removes $x_i = v_i$ is when it sends a no-good message to a_i. We can easily see that this message is either not obsolete, in which case a_i will change its value v_i and breaks our assumption, or obsolete, which means that an **ok?** message has not yet reached a_j which breaks our quiescence assumption. Hence, when reaching σ all agents know the assignments of all their neighbors and no two agents store different assignments for the same variable. □

Lemma 2. *When reaching σ, all constraints are satisfied.*

Proof. In AABT, each time a new value is assigned or a new message is received by an agent a_i, it immediately propagates its constraints $\mathcal{C}_i$. When generating an empty domain, a_i calls procedure `repair` to repair inconsistent assignments. Hence, after processing each message all assignments stored locally satisfy all constraints of the agent. □

Theorem 2. *AABT is sound.*

Proof. Direct from Lemmas 1 and 2. □

Theorem 3. *AABT is complete.*

Proof. In AABT, agents only store valid explanations and no-goods. In addition, all explanations and no-goods are generated by logical inferences from existing constraints. Thus, the empty no-good cannot be inferred if the network is satisfiable. □

Corollary 1. *AABT is a correct and complete solver for hypergraphical games.*

Proof. From Proposition 1 and Theorems 1 to 3. □

6 Empirical Study

In this section we experimentally compare AABT to ANT [5], AABT+AC and ANT+AC [5]. In AABT+AC and ANT+AC arc consistency (AC) is performed in a preprocessing phase before running AABT and ANT. This processing phase is the table passing phase of NashProp [12]. We do not compare the algorithms to NashProp because in [5] it has been shown that ANT achieves orders of magnitude improvements over NashProp. All the problems used for evaluating both algorithms were generated randomly and then modified to ensure that at least one pure strategy Nash equilibrium exists. All experiments were performed on the DisChoco 2.0 platform [17], in which agents are simulated by Java threads that communicate only through message passing. We evaluate the performance of the algorithms by communication load and computation effort. Communication load is measured by the total number of exchanged messages among agents during algorithm execution ($\#msg$) [9]. Computation effort is measured by the number of non-concurrent constraint checks ($\#ncccs$) [19]. $\#ncccs$ is the metric used in distributed constraint solving to simulate the computation time. Algorithms are evaluated on two random benchmarks: random regular graphical polymatrix games, and random graphical polymatrix games.

Random regular graphical polymatrix games: are characterized by $\langle n, d, \kappa \rangle$, where n is the number of players/agents, d is the number of strategies per agent, and κ represents the degree of each agent. For each value combination a utility was uniformly selected in $\{0, \ldots, 9\}$. We solved and report the average over 25 instances of four settings. The first and the fourth settings cover the experiments presented in [5]. In the three first settings, we fixed the number of strategies and the degree of each agent and varied the number of agents in the range 6..16 by a step of 2 to guarantee the graphs are κ-regular. In the first setting we generated 3 random connections for each agent ($\kappa = 3$) and fixed the number of strategies of each agent at $d = 3$. We then increase the number of strategies to $d = 5$ in the second setting and increase the degree of each agent to $\kappa = 4$ in the third setting.

Results are presented in Figure 3. Regarding the number of required messages to solve the problem ($\#msg$), AABT improves ANT by an order of magnitude in almost all instances of Figure 3.

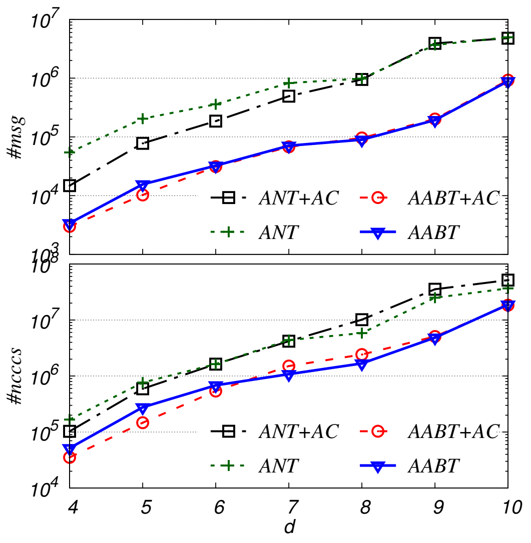


Figure 4. The $\#msg$ and $\#ncccs$ for solving random regular graphical polymatrix games when varying the number of strategies in logarithmic scale.

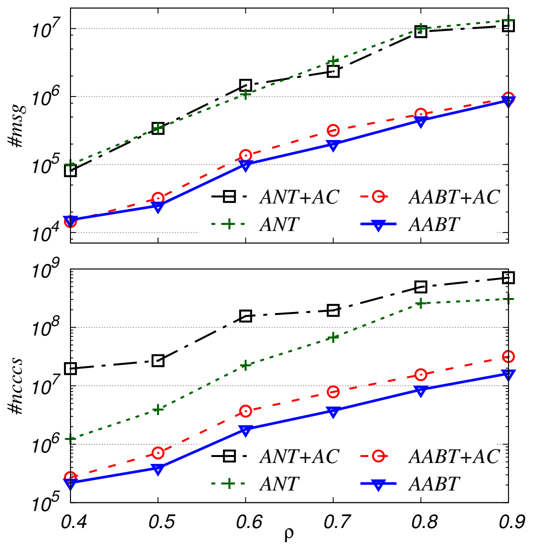


Figure 5. The $\#ncccs$ performed and $\#msg$ exchanged for solving problems where $\langle n = 10, d = 5, \rho\rangle$ in logarithmic scale.

In the first setting (Figure 3(a)), the improvement factor ranges between 3 for small problems and 15 on larger ones. In the second setting (Figure 3(b)), the factor of improvement ranges between 5 for small problems and 29 on larger ones. In the third setting (Figure 3(c)), the factor of improvement ranges between 3 for small problems and 35 on larger ones. For $\#ncccs$, AABT outperforms ANT, although the improvement is less significant than in $\#msg$. In small problems the two algorithms perform similarly. However, in larger problems, the factor of improvement ranges from 4 in the first setting ($d=3, \kappa = 3$) to 9 in the second ($d=5, \kappa = 3$) and 15 in the third ($d=5, \kappa = 4$). Regarding the AC, ANT+AC (resp. AABT+AC) always improves ANT (resp. AABT) in $\#msg$. For $\#ncccs$, AABT+AC always improves AABT, however, ANT+AC only improves ANT on problems with larger number of agents ($n > 10$). AABT always improves ANT+AC and the improvement is more significant (an order of magnitude improvement) in larger problems of the second and the third setting when increasing the number of strategies or the agents degrees. AABT+AC is clearly the best algorithm for solving all instances in the three settings.

In the fourth setting (Figure 4) we generated random 3-regular graphical polymatrix games ($\kappa=3$) where we fixed the number of agents at $n=10$ and varied the number of strategies of all agents in the range 4..10. Again, the results show that AABT outperforms ANT and ANT+AC in all instances. Regarding $\#ncccs$, the improvement factor ranges between 3 to 7. For $\#msg$, in instances with larger number of strategies the AABT improvement over ANT and ANT+AC is again an order of magnitude. AABT and AABT+AC algorithms perform almost similarly in all instances. ANT+AC shows small improvement over ANT on $\#msg$ for problems with smaller number of strategies.

Random graphical polymatrix games: are characterized by $\langle n, d, \rho\rangle$, where n is the number of players/agents, d is the number of strategies per player, ρ is the network connectivity defined as the ratio of existing binary utility functions. For each value combination a utility was uniformly selected from the set $\{0, \ldots, 9\}$. For each $\rho \in \{0.4, \ldots, 0.9\}$, we generated 25 instances in the class $\langle n=10, d=5, \rho\rangle$. We report the average over these instances in Figure 5. The results demonstrate that AABT improves ANT algorithm in both $\#msg$ and $\#ncccs$. This improvement is over an order of magnitude on dense problems, $\rho \geq 6$. In contrast to κ-regular games, the improvement on $\#ncccs$ is more significant than on $\#msg$. Specifically, the improvement factor on $\#ncccs$ when $\rho=.8$ is 30 while it is 22 regarding the $\#msg$. Regarding AC, our results confirm those of [5] that if pruning is limited, AC can create a significant overhead.

Summary: In all our experimentation, AABT always improves ANT and ANT+AC in both metrics $\#msg$ and $\#ncccs$. This improvement is more significant on harder problems (when increasing the number of players/agents and/or the number of strategies of each player and/or the degree of each agent). In sparse problems of κ-regular graph $\kappa=3$, AABT improves the $\#ncccs$ slightly compared to ANT.[8] However, this improvement is more significant on dense graphical polymatrix games. AABT+AC only improves AABT on sparse problems. AC is harmful for hard problems because it does not lead to domain filtering when problems are dense and/or large with a large number of strategies.

7 Conclusion

We studied the problem of finding an approximate Nash Equilibrium in hypergraphical games, an elegant framework for modeling collaboration in multi-agent systems within strategic environments. Our study is based on asymmetric distributed constraint satisfaction, an efficient tool for distributed problem solving allowing agents to keep their utilities private. We proposed a new model of hypergraphical games as an asymmetric DisCSP based on ϵ−`BRConstraint`, a new global constraint modeling hypergraphical games using the original compact representation of the subgames with a filtering algorithm of dominated strategies. Finally we introduced a new asynchronous algorithm for solving (asymmetric) DisCSP, in order to find Nash Equilibria for hypergraphical games. This achieves an order of magnitude improvement in messaging and in non-concurrent computation time on dense problems compared to state-of-the art algorithms.

[8] In our implementation, table constraints are presented in lexicographic ordering of tuples and we use a dichotomic search ($\log(\kappa_i + 1)$) to check the consistency of each tuple.

REFERENCES

[1] Christian Bessiere, Ismel Brito, Patricia Gutierrez, and Pedro Meseguer, 'Global constraints in distributed constraint satisfaction and optimization', *The Computer Journal*, (2013).

[2] Christian Bessiere, Arnold Maestre, Ismel Brito, and Pedro Meseguer, 'Asynchronous backtracking without adding links: a new member in the ABT family', *Artif. Intel.*, **161**, 7–24, (2005).

[3] Ismel Brito, Amnon Meisels, Pedro Meseguer, and Roie Zivan, 'Distributed Constraint Satisfaction with Partially Known Constraints', *Constraints*, **14**, 199–234, (2009).

[4] K. Mani Chandy and Leslie Lamport, 'Distributed Snapshots: Determining Global States of Distributed Systems', *ACM Trans. on Computer Systems*, **3**(1), 63–75, (1985).

[5] Alon Grubshtein and Amnon Meisels, 'Finding a nash equilibrium by asynchronous backtracking', in *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming*, CP'12, pp. 925–940, Berlin, Heidelberg, (2012). Springer-Verlag.

[6] Albert Xin Jiang and Kevin Leyton-Brown, 'A general framework for computing optimal correlated equilibria in compact games', *CoRR*, **abs/1109.6064**, (2011).

[7] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault, 'Maintaining arc-consistency within dynamic backtracking', in *Proceedings of CP'00*, pp. 249–261. Springer Berlin Heidelberg, (2000).

[8] Michael J. Kearns, Michael L. Littman, and Satinder P. Singh, 'Graphical models for game theory', in *Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, UAI'01, pp. 253–260, San Francisco, CA, USA, (2001). Morgan Kaufmann Publishers Inc.

[9] Nancy A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Series, 1997.

[10] Rajiv T. Maheswaran, Jonathan P. Pearce, Emma Bowring, Pradeep Varakantham, and Milind Tambe, 'Privacy loss in distributed constraint reasoning: A quantitative framework for analysis and its applications', *Autonomous Agents and Multi-Agent Systems*, **13**(1), 27–60, (2006).

[11] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani, *Algorithmic Game Theory*, Cambridge University Press, New York, NY, USA, 2007.

[12] Luis E Ortiz and Michael Kearns, 'Nash propagation for loopy graphical games', in *Advances in Neural Information Processing Systems 15*, eds., S. Becker, S. Thrun, and K. Obermayer, 817–824, MIT Press, (2003).

[13] Christos H. Papadimitriou and Tim Roughgarden, 'Computing correlated equilibria in multi-player games', *Journal of the ACM (JACM)*, **55**(3), 14:1–14:29, (August 2008).

[14] Pascal Van Hentenryck and Jean-Philippe Carillon, 'Generality vs. specificity: an experience with ai and or techniques', in *Proceedings of AAAI'88*, pp. 660–664, (1988).

[15] David Vickrey and Daphne Koller, 'Multi-agent algorithms for solving graphical games', in *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pp. 345–351, (2002).

[16] Mohamed Wahbi and Kenneth N. Brown, 'Global constraints in distributed csp: Concurrent gac and explanations in abt', in *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming*, CP'2014, pp. 721–737, Lyon, France, (2014). Springer International Publishing.

[17] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, and El Houssine Bouyakhf, 'DisChoco 2: A Platform for Distributed Constraint Reasoning', in *Proceedings of workshop on DCR'11*, pp. 112–121, (2011).

[18] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara, 'Distributed constraint satisfaction for formalizing distributed problem solving', in *Proceedings of ICDCS*, pp. 614–621, (1992).

[19] Roie Zivan and Amnon Meisels, 'Message delay and DisCSP search algorithms', *AMAI*, **46**(4), 415–439, (2006).