Cooperative Problem Solving against Adversary: Quantified Distributed Constraint Satisfaction Problem

Satomi Baba, Atsushi Iwasaki, Makoto Yokoo
Kyushu University
Fukuoka 819-0395, Japan
{s-baba@agent., iwasaki@, yokoo@}is.kyushu-u.ac.jp

Marius Călin Silaghi
Florida Institute of Technology
Melbourne, FL 32901, United States
msilaghi@fit.edu

Katsutoshi Hirayama
Kobe University
Kobe 658-0022, Japan
hirayama@maritime.kobe-u.ac.jp

Toshihiro Matsui
Nagoya Insutitute of Technology
Nagoya 466-8555, Japan
matsui.t@nitech.ac.jp

ABSTRACT

In this paper, we extend the traditional formalization of distributed constraint satisfaction problems (DisCSP) to a quantified DisCSP. A quantified DisCSP includes several universally quantified variables, while all of the variables in a traditional DisCSP are existentially quantified. A universally quantified variable represents a choice of nature or an adversary. A quantified DisCSP formalizes a situation where a team of agents is trying to make a robust plan against nature or an adversary. In this paper, we present the formalization of such a quantified DisCSP and develop an algorithm for solving it by generalizing the asynchronous backtracking algorithm used for solving a DisCSP. In this algorithm, agents communicate a value assignment called a *good* in addition to the *nogood* used in asynchronous backtracking. Interestingly, the procedures executed by an adversarial/cooperative agent for *good*/*nogood* are completely symmetrical. Furthermore, we develop a method that improves this basic algorithm. Experimental evaluation results illustrate that we observe an easy-hard-easy transition by changing the tightness of the constraints, while very loose problem instances are relatively hard. The modification of the basic algorithm is also effective and reduces the cycles about 25% for the hardest problem instances.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Algorithms

Cite as: Cooperative Problem Solving against Adversary: Quantified Distributed Constraint Satisfaction Problem, Satomi Baba, Athushi Iwasaki, Makoto Yokoo, Marius Silaghi, Katsutoshi Hirayama and Toshihiro Matsui, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. 781-788

Keywords

distributed constraint satisfaction problem, quantified constraint satisfaction problem

1. INTRODUCTION

A constraint satisfaction problem (CSP) [10] is the problem of finding an assignment of values to variables that satisfies all constraints. Each variable takes a value from a discrete finite domain. A variety of AI problems can be formalized as CSPs. Consequently, research on CSPs has a long and distinguished history in AI literature.

A distributed CSP (DisCSP) [13] is a CSP where variables and constraints are distributed among agents. Various application problems in multi-agent systems that are concerned with finding a consistent combination of agent actions (e.g., sensor networks [6]) can be formalized as DisCSPs. Also, many algorithms for solving a DisCSP have been developed (e.g., Asynchronous Backtracking [13], Distributed Dynamic Backtracking [2], Asynchronous Partial Overlay [11]).

A common assumption used in traditional DisCSP studies is that all agents are cooperative and everything in the world is under control of the team of agents. If external factors exist, they are already fixed and do not change when the team is searching/executing its plan.

In practice, there are many situations where this assumption does not hold. For example, some uncontrollable random factors exist that are not fixed in the planning time. Even worse, these factors might be controlled by an adversary who acts against the team. In this work, we introduce the framework of quantified DisCSPs to overcome this limitation by introducing universally quantified variables that represent a choice of nature or adversary.

There are other extensions of the CSP framework to model problem solving in an open environment. For example, in a dynamic CSP, constraints and variables can change over time [12]. In an open CSP, the domain of a variable is not known in advance and must be obtained through information gathering [5]. The quantified (distributed) CSP framework is different from these approaches in the point that possible dynamic changes or uncertainty in problem solving come from an adversary.

A quantified constraint satisfaction problem (QCSP) [3] is an extension of a CSP in which some variables are universally quantified. The goal of a QCSP is to find the assignments of values to existentially quantified variables that satisfy all constraints, regardless of the choice of universally quantified variables. While solving a CSP is generally NP-complete, solving a QCSP is generally PSPACE-complete. In a QCSP, a universally quantified variable can be considered the choice of nature or an adversary. A QCSP can formalize such application problems as planning under uncertainty and playing a game against an adversary.

In this paper, we present a quantified DisCSP, which is a combination of a DisCSP and a quantified CSP by introducing universally quantified variables in DisCSP. In a quantified DisCSP, a team of agents tries to make a robust plan against nature or an adversary.

Furthermore, we present a basic algorithm for solving a quantified DisCSP, which is a generalization of the classic asynchronous backtracking algorithm [13] for a DisCSP. In this algorithm, agents communicate a value assignment of a subset of variables called a *good*, which satisfies some constraints, as well as a value assignment of a subset of variables called a *nogood*, which violates some constraints. Interestingly, the procedures executed by an adversarial agent in response to a *good* and a *nogood* are symmetrical to the procedures executed by a cooperative agent in response to a *nogood* and a *good*. We also show a method that improves the basic algorithm by creating smaller *nogoods* and sending *nogoods* to non-direct ancestors.

Experimental evaluation results illustrate that as in traditional DisCSPs, we observe an easy-hard-easy transition by changing the tightness of the constraints, while very loose problem instances are relatively hard, since the team of agents still needs to consider all possible value assignments of universally quantified variables. Also, the modification of the basic algorithm is effective and can reduce cycles about 25% for the hardest problem instances.

2. RELATED RESEARCH

2.1 Quantified Boolean Formulas

A quantified boolean formula (QBF) is a generalization of a SAT in which some variables can be universally quantified. A SAT is the problem of finding a solution that satisfies a given boolean formula. A boolean formula in a SAT is in conjunctive normal form (CNF). A CNF is a conjunction of clauses, and a clause is a disjunction of literals (a boolean variable or the negation of a variable).

A QBF is a generalization of a SAT in which variables can be either universally or existentially quantified. The meanings of quantifiers are as follows:

- $\exists x F$: There exists a value of x in {True, False} such that F is true.
- $\forall x F$: For every value of x in {True, False}, F is true.

A QBF has form QF as represented in (1), where F is a propositional formula expressed in CNF and Q is a sequence of quantified variables such as ($\exists x$ or $\forall x$).

$$\exists x_1 \forall x_2 \exists x_3 (x_1 \vee x_2) \wedge (x_2 \vee \neg x_3) \tag{1}$$

Q consists of n pairs, where each pair consists of quantifier Q_i and variable x_i, as represented in (2).

$$Q_1 x_1 \cdots Q_n x_n \tag{2}$$

Note the importance of sequence order. For example, the meanings of $\forall x \exists y\ loves(x, y)$ and $\exists y \forall x\ loves(x, y)$ are quite different. $\forall x \exists y\ loves(x, y)$ means any x loves some y, where y can be different for each x. On the other hand, $\exists y \forall x\ loves(x, y)$ means particular person y is loved by everybody.

The goal of a QBF is to assign, for every value of universally quantified variables, the values of existentially quantified variables so that the boolean formula is true. For universally quantified variable x_i, if existentially quantified variable x_j appears before x_i, then the boolean formula must be true for every value of x_i. If x_j appears after x_i, then the choice of x_j can be a function of x_i.

Various algorithms for solving QBF have been developed, e.g., Qube [8] based on the DPLL algorithm [4], sKizzo [1] based on the Skolemization technique, and so on.

2.2 Quantified CSP

A constraint satisfaction problem (CSP) is a problem of finding an assignment of values to variables that satisfies constraints. A CSP is described with n variables $x_1, x_2, \cdots, x_n$ and m constraints $C_1, C_2, \cdots, C_m$. Each variable x_i takes a value from a dicrete finite domain D_i.

A QCSP [3] is a generalization of a CSP in which some variables are universally quantified. Also, it is a generalization of a QBF. Solving a QCSP is PSPACE-complete.

A QCSP has form QC as in (3), where C is a conjunction of constraints and Q is a sequence of quantified variables.

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 (x_1 \neq x_3) \wedge (x_1 < x_4) \wedge (x_2 \neq x_3) \tag{3}$$

The semantics of a QCSP QC can be defined recursively as follows.

- If C is empty then the problem is true. If Q is of the form $\exists x_1 Q_2 x_2 \cdots Q_n x_n$, then QC is true iff there exists some value $a \in D(x_1)$ such that $Q_2 x_2 \cdots Q_n x_n C[(x_1, a)]$ is true. If Q is of the form $\forall x_1 Q_2 x_2 \cdots Q_n x_n$, then QC is true iff for each value $a \in D(x_1)$, $Q_2 x_2 \cdots Q_n x_n$ $C[(x_1, a)]$ is true. $C[(x_1, a)]$ is a constraint C where x_1 is instantiated to value a.

Several algorithms solve QCSP, most of which are extensions of QBF-based algorithms. One notable exception is an algorithm called QCSP-Solve [7], which introduces techniques specialized to a QCSP.

2.3 Distributed CSP

A DisCSP [13] is a CSP in which variables and constraints are distributed among agents.

We assume the following communication model.

- Agents communicate by sending messages.
- An agent can send messages to other agents iff the agent knows their addresses.
- For transmission between any pair of agents, messages are received in the order in which they were sent.

Each agent has some variables and tries to determine their values. However, inter-agent constraints exist, and the value assignment must satisfy them.

Asynchronous Backtracking Algorithm

We make the following assumptions while describing the asynchronous backtracking algorithms [13] for simplicity.

- Each agent has exactly one variable.
- Each agent knows all constraints involving its variable.
- All constraints are binary.

Under the above assumptions, a DisCSP can be represented as a network, where agents are nodes and constraints are links. We assume a link is directed. More specifically, for two agents with a constraint relationship, one agent checks the constraint after receiving the other agent's value. Thus, the direction of the link is set from the agent that sends its value to the agent that checks the constraint. We assume the priority order of variables/agents is determined by the alphabetical order of the variable identifiers. The direction of the link is determined by this priority order.

The asynchronous backtracking algorithm is a basic algorithm for solving a DisCSP. In this algorithm, each agent determines its value asynchronously and concurrently and sends this value to related agents connected by outgoing links. Then each agent waits for incoming messages. If an agent receives a message, it executes a certain procedure for each message type. In this algorithm, the following two types of messages are used.

- (**ok?**, (x_j, *value*)) : this message conveys that the value of x_j is *value*.
- (**nogood**, x_j, *nogood*) : this message conveys a new *nogood*. A *nogood* is a combination of values that causes a constraint violation. For example, *nogood*$\{(x_i, d_i), (x_j, d_j)\}$ represents that a combination of (x_i, d_i) and (x_j, d_j) causes a constraint violation.

In the asynchronous backtracking algorithm, by receiving an **ok?** message, agent x_i tries to find a consistent value with higher-priority agents. If no consistent value exists, agent x_i sends a **nogood** message to the agent with the lowest priority among agents whose priorities are higher than agent x_i. The asynchronous backtracking algorithm is guaranteed to be complete.

3. QUANTIFIED DISTRIBUTED CSP

3.1 Problem Definition

A quantified DisCSP is a quantified CSP where the variables and constraints are distributed among agents. We assume each existentially quantified variable is owned by a separate agent, and all universally quantified variables are controlled by a single adversary. The sequence of quantified variables defines the order of decision making. If x_i appears before x_j in the sequence, when determining the value of x_j, the value assignment of x_i is observable for the agent who owns x_j. We assume that a team of agents with existentially quantified variables tries to find a plan for determining its variables so that all constraints are satisfied, regardless of the value assignments of the adversary.

3.2 Example of Quantified Distributed CSP

Consider contingency planning as an example of a quantified DisCSP. The goal is creating a communication network where n nodes are connected with each other (directly or indirectly) under the following requirements.

- The final network must be robust against k node failures; even to the failure of k nodes, the remaining network must be connected.

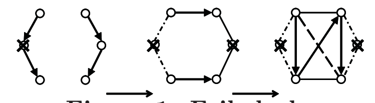
Figure 1: Failed plan

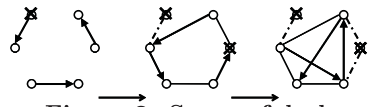
Figure 2: Successful plan

- The final network must be constructed within T steps.
- In each step, each node can place at most one bidirectional communication link (which connects the node with another node).
- The total number of links cannot exceed l.
- An adversary can make any one node fail after each step (except for the final T-th step).
- The adversary can make any k nodes fail after T steps (i.e., it can make $T + k - 1$ nodes fail in total).

In this problem, we assume each agent is a node. This problem can be formalized as a quantified DisCSP as follows.

- Variables: x_{ij}, where $1 \leq i \leq n$, $1 \leq j \leq T$, and y_j, where $1 \leq j \leq T + k - 1$. i means the node id and j means a time step where $j \leq T$. For y_j where $j \geq T$, y_j represents an adversary's action after T steps.
- Domain: x_{ij} takes value from $\{0, \ldots, i-1, i+1, \ldots, n\}$. Each value represents a node to which x_{ij} sets a link. When x_{ij} takes value 0, x_{ij} doesn't set a link.
  y_j takes value from $\{0, \ldots, n\}$. Each value represents a node that fails. When y_j takes value 0, no node fails.
- Constraint: The total number of links is less than l.
  The remaining nodes are connected with each other after the choice of y_{T+k-1}.
- Quantifier sequence: $\exists x_{11} \cdots \exists x_{n1} \forall y_1 \ldots \exists x_{1T} \cdots \exists x_{nT}$ $\forall y_T \ldots \forall y_{T+k-1}$

We show examples of the plans in Figs. 1 and 2, where $n = 6, T = 3, k = 2, l = 9$. In Fig. 1, after the second step, the adversary has caused the left and right nodes to fail. Since already 6 links have been used, the agents can only use 3 more links. Thus, they cannot create a network that is robust against two node failures in the third step.

On the other hand, in Fig. 2, the agents can create a network that is robust against two node failures in the third step. The agents must carefully decide the order of placing links to achieve this goal.

4. ALGORITHM FOR SOLVING QUANTIFIED DISTRIBUTED CSP

In this section, we present an algorithm for solving quantified DisCSPs. As an initial step, we develop an algorithm that is an extension of the classic asynchronous backtracking algorithm. Although many works exist on more sophisticated and efficient algorithms for DisCSPs, we chose the asynchronous backtracking algorithm as the basis of our new algorithm since it is the most basic and clear algorithm for solving DisCSPs. We believe the techniques/ideas utilized in this new algorithm are effective when developing more efficient quantified DisCSP algorithms based on other DisCSP algorithms.

4.1 Basic Ideas

We introduce the following ideas to extend the asynchronous backtracking algorithm for quantified DisCSPs.

- The priority order among agents is determined based on the sequence of quantified variables; if x_j appears before x_i, then x_j has a higher priority than x_i. Note that the order among existentially quantified variables, whose positions are adjacent in the quantifier sequence, can be determined arbitrarily.
- For simplicity, we assume agents know a Depth First Search (DFS) tree, which is determined by the priority order.
- A virtual agent, who exists for each universally quantified variable, imitates the adversary's actions but cooperates in searching for a plan with its team of cooperative agents. In a sense, the team of cooperative agents is making a plan *off-line*. When the team actually plays against the adversary, it executes the plan obtained in this off-line search. In reality, one of the agents on the team should act as a virtual agent. Any team member can act as a virtual agent for universally quantified variable x_i. However, to reduce communication costs, an agent who is the parent or child of x_i in the DFS tree should act as the virtual agent.
- Agents communicate **good** messages as well as **ok?** and **nogood** messages. A *good* is a value assignment of a subset of variables that satisfies constraints owned by the sender and its descendants. Interestingly, the procedures executed by adversarial agents for *good* and *nogood* are symmetrical to the procedures executed by cooperative agents for *good* and *nogood*, as described in the next subsection.

4.2 Generation of nogood/good

In this subsection, we compare the procedures executed at the existentially/universally quantified variables for generating new *good*/*nogood*, and show they are logically correct.

A *nogood* is a combination of values that represents a contradiction. For example, *nogood* $\{(x_1, 1), (x_2, 2)\}$ represents that$x_1 = 1 \wedge x_2 = 2$, causes a contradiction.

A new *nogood* is generated in the following cases.

- Assume x_2 is an existentially quantified variable. A *nogood* $\{(x_1, 1)\}$ can be generated from $x_2 = 1 \vee x_2 = 2$, *nogood* $\{(x_1, 1), (x_2, 1)\}$, and *nogood* $\{(x_1, 1), (x_2, 2)\}$.

  This is a standard resolution procedure, where from $x_1 = 1 \wedge x_2 = 1 \rightarrow \bot$, $x_1 = 1 \wedge x_2 = 2 \rightarrow \bot$, and $x_2 = 1 \vee x_2 = 2$, we derive $x_1 = 1 \rightarrow \bot$.
- Assume that x_2 is a universally quantified variable. A *nogood* $\{(x_1, 1)\}$ can be generated from *nogood* $\{(x_1, 1), (x_2, 1)\}$.

  This procedure means from $x_1 = 1 \wedge x_2 = 1 \rightarrow \bot$ and $x_2 = 1$, we derive $x_1 = 1 \rightarrow \bot$.

Accordingly, a cooperative agent with an existentially quantified variable sends a **nogood** message only after it finds out that all of its possible values cause contradiction (either by its own constraints or by received *nogoods*).

On the other hand, a virtual/adversarial agent with a universally quantified variable will send a **nogood** message immediately after it finds out that at least one of its possible values causes a contradiction.

A *good* is a combination of values that satisfies constraints. For example, *good* $\{(x_1, 1), (x_2, 2)\}$, which is generated by agent x_3, represents that $x_1 = 1 \wedge x_2 = 2$ satisfies all the constraints of x_3 and its descendants.

A *good* is generated in the following cases.

- Assume x_2 is an existentially quantified variable. A *good* $\{(x_1, 1)\}$ can be generated from *good* $\{(x_1, 1), (x_2, 1)\}$, which is sent from its only child x_3, if $x_1 = 1 \wedge x_2 = 1$ satisfies the constraint between x_1 and x_2.

  This procedure means that $\{(x_1, 1), (x_2, 1)\}$ satisfies all constraints related to x_3 and its descendants as well as the constraint between x_1 and x_2, so we conclude that $\{(x_1, 1)\}$ can satisfy all constraints related to x_2 and its descendants (assuming x_2 chooses its value to 1).
- Assume x_2 is a universally quantified variable. A *good* $\{(x_1, 1)\}$ can be generated from $x_2 = 1 \vee x_2 = 2$, *good* $\{(x_1, 1), (x_2, 1)\}$ and *good* $\{(x_1, 1), (x_2, 2)\}$ sent from its only child x_3, if $x_1 = 1 \wedge x_2 = 1$ and $x_1 = 1 \wedge x_2 = 2$ satisfy the constraint between x_1 and x_2.

  This procedure means that both $\{(x_1, 1), (x_2, 1)\}$ and $\{(x_1, 1), (x_2, 2)\}$ satisfy all constraints related to x_3 and its descendants, as well as the constraint between x_1 and x_2, and $x_2 = 1 \vee x_2 = 2$, so we conclude that $\{(x_1, 1)\}$ can satisfy all constraints related to x_2 and its descendants (regardless of the choice of x_2).

Consequently, a cooperative agent who has an existentially quantified variable immediately sends a **good** message after it receives a *good* for at least one of its possible values (and the value also satisfies its own constraints). On the other hand, a virtual/adversarial agent who has a universally quantified variable sends a **good** message only after it receives **good** messages for all of its possible values (and each of these values satisfies its own constraints).

4.3 Details of Algorithm

In this algorithm, as in the asynchronous backtracking algorithm, each agent determines its value asynchronously and concurrently. For simplicity, we assume an agent sends its value assignment by **ok?** messages to all descendants. After an agent sends its value, it waits for an incoming message. If an agent receives a message, it executes the procedure defined for each message type.

There are three types of messages; **ok?**, **nogood**, and **good**. Each agent sends its *agent_view* as a *good/nogood*, where the *agent_view* contains the value assignments of ancestors. The procedures executed by a cooperative agent upon receiving messages are shown in Fig. 3, and the procedures executed by an adversarial/virtual agent upon receiving messages are shown in Fig. 4. Fig. 5 describes **backtrack** and **send_good** procedures that are used in the above procedures. As in the asynchronous backtracking algorithm, when a cooperative agent receives an **ok?** message, the agent updates its *agent_view* and checks whether its current value is consistent with its *agent_view*. When a cooperative agent is a leaf agent, it sends a **good** message to the parent if the current value is consistent with its *agent_view*. On the other hand, when an adversarial/virtual agent receives

```
**when received** (**ok**?, $(x_j, value)$) **do**
  add $(x_j, value)$ to *agent_view*;
  **check_agent_view**;
  **when** agent is a leaf and,
    *agent_view* contains all ancestors **do**
    **send_good**; **end do**; **end do**;

**when received** (**nogood**, $x_j$, *nogood*) **do**
  add *nogood* to *nogood_list*
  **check_agent_view**; **end do**;

**when received** (**good**, $x_j$, *good*) **do**
  add *good* to *good_list*
  **when** received consistent *good* from all children
    and *agent_view* contains all ancestors **do**
    **send_good**; **end do**; **end do**;

procedure **check_agent_view**;
  **when** *current_value* and *agent_view* are inconsistent **do**
    change *current_value* to a new consistent value;
    **when** cannot find such a value **do backtrack**; **end do**;
  send (**ok**?, $(x_i, current\_value)$) to descendants;
```

Figure 3: Procedures of cooperative agent upon receiving messages

an **ok?** message, it sends a **nogood** message if it finds one value that causes a constraint violation.

While a cooperative agent tries to find a value that satisfies all constraints, an adversarial/virtual agent tries to find a value that violates some constraints. Thus, the procedures executed by an adversarial/virtual agent upon receiving **good** and **nogood** messages are symmetrical to the procedures executed by a cooperative agent upon receiving **good** and **nogood** messages.

When a cooperative agent receives a **nogood** message, it searches for another value that satisfies constraints. If it cannot find any consistent value, it sends a **nogood** message to its parent. On the other hand, when an adversarial agent receives a **nogood** message, it does not search for another value that satisfies constraints but immediately sends a **nogood** message to its parent, since it can choose the value described in the received *nogood* and violate some constraints.

When a cooperative agent receives **good** messages from all children for the same combination of values, it immediately sends a **good** message to its parent, since it can choose the value described in the received *good* and satisfy the constraints. On the other hand, when an adversarial agent receives a **good** message, it searches for another value to violate some constraints. If it cannot find such a value, i.e., it receives **good** messages for all possible values, it then sends a **good** message to its parent.

This algorithm terminates when the root agent generates an empty *good* or *nogood*. A *good*/*nogood* represents a set of value assignments, where existentially/universally quantified variables can satisfy/violate constraints regardless of the choice of universally/existentially quantified variables. Thus, the problem is solvable/unsolvable when an empty *good*/*nogood* is found.

For simplicity, we describe the algorithm so that it only checks whether the problem is solvable. To construct a plan for acting against the adversary, a cooperative agent must record *goods* sent from its children.

```
**when received** (**ok**?, $(x_j, value)$) **do**
  add $(x_j, value)$ to *agent_view*;
  **check_agent_view_adversary**; **end do**;

**when received** (**nogood**, $x_j$, *nogood*) **do**
  add *nogood* to *nogood_list*;
  **when** *nogood* is consistent with *agent_view* and
    *current_value*, and *agent_view* contains all ancestors **do**
    **backtrack**; **end do**; **end do**;

**when received** (**good**, $x_j$, *good*) **do**
  add *good* to *good_list*;
  **check_agent_view_adversary**; **end do**;

procedure **check_agent_view_adversary**;
  **when** for some $v \in D_i$,
    $v$ and *agent_view* are inconsistent **do**
    **backtrack**; **end do**;
  **when** for all $v \in D_i$,
    received *good* messages from all children (or a leaf agent),
    and *agent_view* and $v$ are consistent **do**
    **send_good**; **end do**;
  **otherwise do** choose $v \in D_i$ so that
    some children have not sent a *good* message yet;
    change *current_value* to $v$;
    send (**ok**?, $(x_i, current\_value)$) to descendants; **end do**;
```

Figure 4: Procedures of adversarial/virtual agent upon receiving messages

4.4 Example

We illustrate an example of an algorithm execution in Fig. 6, which represents a problem that consists of $Q = \exists x_1 \forall x_2 \exists x_3 \exists x_4$, $C = \{nogood\,\{(x_1,1),(x_3,1)\}, nogood\,\{(x_2,1),(x_3,2)\}, nogood\,\{(x_2,2),(x_4,1)\}\}$, and $D_1 = D_2 = D_3 = D_4 = \{1,2\}$.

By receiving **ok?** messages from x_1 and x_2, the *agent_view* of x_3 and x_4 will be $\{(x_1,1),(x_2,1)\}$ (Fig. 6 (b)). Since there is no value for x_3 that is consistent with this *agent_view*, x_3 sends a **nogood** message to its parent x_2. Also, since there is a value for x_4 that is consistent with this *agent_view*, x_4 sends a **good** message to its parent x_2 (Fig. 6 (c)).

After receiving these **nogood** and **good** messages, x_2 sends a **nogood** message to x_1 because x_2 is an adversarial agent (Fig. 6 (d)). By receiving this **nogood** message, x_1 knows that $x_1 = 1$ has caused a constraint violation. Thus, x_1 changes its value to 2 and sends **ok?** messages to its descendants (Fig. 6 (e)). After receiving this **ok?** message, x_2, x_3, and x_4 record this value to their *agent_view*. x_3 and x_4 send **good** messages to x_2 because x_3 and x_4 can select a value that satisfies their constraints. (Fig. 6 (f)).

Now x_2 has received **good** messages for its value 1 from all children. Thus, x_2 selects another value 2 (so that it might violate some constraints) and sends **ok?** messages (Fig. 6 (g)).

x_3 and x_4 receive this **ok?** message and record the value to their *agent_view*. Since both x_3 and x_4 can select a value that satisfies constraints, x_3 and x_4 send **good** messages to x_2 (Fig. 6 (h)). After receiving these **good** messages, x_2 sends a **good** message to x_1 because every value for x_2 satisfies related constraints (Fig. 6 (i)). Since x_1 is a cooperative agent, x_1 can select the current value. Thus, an empty *good* is generated and this problem is solvable.

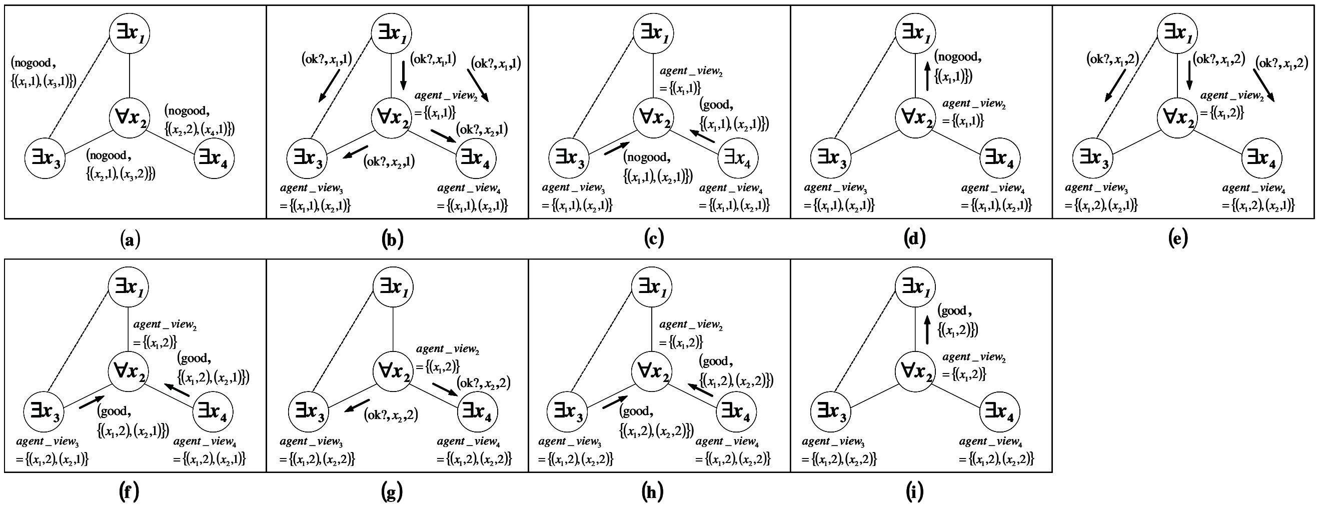
(a) (b) (c) (d) (e) (f) (g) (h) (i)
∃x1 ∀x2 ∃x3 ∃x4
(nogood, {(x1,1),(x3,1)})
(nogood, {(x2,2),(x4,1)})
(nogood, {(x2,1),(x3,2)})
(ok?,x1,1)
(ok?,x2,1)
agent_view2 = {(x1,1)}
agent_view3 = {(x1,1),(x2,1)}
agent_view4 = {(x1,1),(x2,1)}
(nogood, {(x1,1),(x2,1)})
(good, {(x1,1),(x2,1)})
(nogood, {(x1,1)})
(ok?,x1,2)
agent_view2 = {(x1,2)}
agent_view3 = {(x1,2),(x2,1)}
agent_view4 = {(x1,2),(x2,1)}
(good, {(x1,2),(x2,1)})
(ok?,x2,2)
agent_view3 = {(x1,2),(x2,2)}
agent_view4 = {(x1,2),(x2,2)}
(good, {(x1,2),(x2,2)})
(good, {(x1,2)})

Figure 6: Example of algorithm execution

```
procedure backtrack
  nogood ← agent_view
  when nogood = {} do
    broadcast to other agents that the problem is unsolvable;
      terminate this algorithm; end do;
  send (nogood, $x_i$, nogoods) to its parent $x_j$;
  remove ($x_j$, $d$) from agent_view;

procedure send_good
  good ← agent_view
  when good={} do
    broadcast to other agents that the problem is solvable;
      terminate this algorithm; end do;
  send (good, $x_i$, good) to its parent
```

Figure 5: Procedure for backtrack and send_good

4.5 Algorithm Correctness and Completeness

This algorithm terminates by concluding that the problem is unsolvable when an empty *nogood* is generated at the root agent. On the other hand, it terminates by concluding the problem is solvable when an empty *good* is generated at the root agent. Therefore, to show that this algorithm is correct/complete, it suffices to show the following facts.

- The procedures for generating new *nogood*/*good* are logically correct. When an empty *good* is generated, the problem is solvable. When an empty *nogood* is generated, the problem is unsolvable.
- The algorithm does not stop before an empty *nogood* or an empty *good* is generated at the root agent, and this algorithm never enters an infinite processing loop.

We prove these two facts in Theorems 1 and 2, respectively.

THEOREM 1. *If an empty* good *is generated in this algorithm, then the problem is solvable. If an empty* nogood *is generated, then the problem is unsolvable.*

PROOF. We show that the procedures for generating new *nogood*/*good* are logically correct. Therefore, when an empty *good* is generated, the problem is solvable. When an empty *nogood* is generated, the problem is unsolvable.

We prove this by structural induction. First, for the base case, we show that the procedure of a leaf agent is correct. A leaf agent only receives **ok?** messages. If the leaf agent is a cooperative agent, it generates a *good* and sends it to its parent only when it can select a value that is consistent with its *agent_view*. Moreover, it generates a *nogood* and sends it to its parent only when it cannot select any consistent value. It is clear that the generated *good*/*nogood* is correct.

If the leaf agent is an adversarial/virtual agent, this agent generates a *good* and sends it to the parent only when no value exists that causes constraint violations. Moreover, it generates a *nogood* and sends it to the parent if at least one value exists that causes some constraint violations. It is clear that the generated *good*/*nogood* are correct.

Now, for the inductive case, assume that for agent x_i, all received *goods*/*nogoods* from its descendants are correct. We derive that the *good*/*nogood* generated by x_i are correct.

If x_i is a cooperative agent, it generates a *nogood* identical to its *agent_view* only when, for each of its values $v \in D_i$, either of two conditions holds: (i) v and *agent_view* violate some constraints related to x_i, or (ii) a *nogood* that is consistent with $x_i = v$ and *agent_view* is sent from its child. Consequently, assuming the *nogoods* sent from its children are correct, this newly generated *nogood* is also correct. Also, x_i generates a *good* identical to its *agent_view* only when there exists at least one value $v \in D_i$, which satisfies the following conditions: (i) x_i receives **good** messages from all children, (ii) each *good* is consistent with $x_i = v$ and *agent_view*, and (iii) $x_i = v$ and *agent_view* satisfy its own constraints. Thus, assuming the *goods* sent from its children are correct, this newly generated *good* is also correct.

If x_i is an adversarial/virtual agent, it generates a *nogood* identical to its *agent_view* when at least one value $v \in D_i$ exists, where either of two conditions holds: (i) v and *agent_view* violate some constraints related to x_i, or (ii) a *nogood* that is consistent with $x_i = v$ and *agent_view* is sent from its child. Therefore, assuming the *nogoods* sent from its children are correct, this newly generated *nogood* is also correct. Also, x_i generates a *good* identical to its *agent_view* only when, for each of its values $v \in D_i$, it receives **good** messages from all children and the *good* is consistent with $x_i = v$ and *agent_view*, and v and *agent_view* satisfy its own constraints. Thus, assuming the *goods* sent from its children are correct, this newly generated *good* is also correct.

From the above facts, the procedures for generating new *nogood*/*good* at each agent are logically correct. When an empty *good* is generated at the root agent, the problem is solvable. On the other hand, when an empty *nogood* is generated at the root agent, the problem is unsolvable. □

THEOREM 2. *The algorithm does not stop before the root agent generates an empty* good *or* nogood*, and never enters an infinite processing loop.*

PROOF. To prove Theorem 2, we first show that when an agent sends an **ok?** message, it receives a *good* or *nogood* message from each of its children. We show this fact by structural induction. For the base case, it is clear that a leaf agent never fails to send a *good*/*nogood* message.

For the inductive case, we show that agent x_i always sends a *good* or *nogood* message to its parent, assuming that it always receives *good* or *nogood* messages from its children.

If x_i is a cooperative agent, when x_i receives an **ok?** message from its parent, it sends a *good* to the parent if it receives **good** messages for its current value and its *agent_view* from all children. On the other hand, if x_i receives a *nogood* from a child, x_i changes its value and sends **ok?** messages. The descendants send a *nogood* or *good* in reply to this message. Since the domain of the variable of x_i is finite, x_i cannot change its value forever. Eventually, it will send a *nogood* message to its parent.

If x_i is an adversarial/virtual agent, when x_i receives an **ok?** message from its parent, it sends a *nogood* to the parent if it receives a **nogood** messages for its current value and its *agent_view* from any child. On the other hand, if x_i receives a *good* from all of its children, x_i changes its value and sends **ok?** messages. The descendants send a *nogood* or *good* in reply to this message. Since the domain of the variable of x_i is finite, x_i cannot change its value forever. Eventually, it will send a *good* message to its parent.

Thus, each child of the root agent always sends a *nogood* or *good* message if the root agent sends an **ok?** message.

If the root is a cooperative agent, when it receives **good** messages for its current value and its *agent_view* from all children, it generates an empty *good*. On the other hand, if it receives a *nogood* from a child, it changes its value and sends **ok?** messages. The descendants send a *nogood* or *good* in reply to this message. Since the variable domain is finite, it cannot change its value forever. Eventually, it will generate an empty *nogood* and the algorithm terminates.

If the root is an adversarial/virtual agent, when it receives a **nogood** message for its current value and its *agent_view* from one of its children, it generates an empty *nogood*. On the other hand, if it receives *good* messages from all of its children, it changes its value and sends **ok?** messages. The descendants send a *nogood* or *good* in reply to this message. Since the domain of the variable of the root is finite, it cannot change its value forever. Eventually, it will generate an empty *good* and the algorithm terminates. □

4.6 Improvement of Basic Algorithm

We show a method for improving the basic algorithm. its main ideas are as follows.

- Generate a smaller *nogood* based on a resolvent-based learning technique described in [9].
- Skip sending a new *nogood* to a universally quantified variable/agent.

The modified algorithm is correct and complete. We omit the proof due to space limitations.

4.6.1 Resolvent-based Learning

We generate a smaller *nogood* based on a resolvent-based learning technique described in [9] as follows.

Assume agent i has variable x_i, whose domain is D_i. Each value $d \in D_i$ is violating one or more constraints under the current *agent_view*. For each value $d \in D_i$, agent i chooses one *nogood* as follows.

- From the *nogoods* related to d, choose the smallest one. If multiple smallest *nogoods* exist, then compare the priority of the lowest priority agents in these *nogoods*, and choose the one with the higher priority.

Then we generate a new *nogood* by taking the union of the selected nogoods.

In the basic algorithm, the whole *agent_view* is used to create a new *nogood*. Thus, the new *nogood* might contain variables irrelevant to the current failure, so the new *nogood* is always sent to the direct parent, even though the parent is not responsible for the current failure. By utilizing this resolvent-based learning technique, an agent can create a smaller *nogood* that contains less irrelevant variables and can send the *nogood* directly to an ancestor who is not a direct parent. Thus, the efficiency of the algorithm can be improved.

4.6.2 Skipping Universally Quantified Variable

When the recipient of a new *nogood* is a universally quantified variable, we can skip sending this new *nogood* to the universally quantified variable/virtual agent. We remove the agent from the *nogood* and send the new *nogood* to the lowest priority agent in the *nogood*. This operation is correct since if we send the *nogood* to the universally quantified variable, it will eventually remove itself from the *nogood* and send a new *nogood* to its parent.

Furthermore, this operation can be done repeatedly as long as the lowest priority variable in the *nogood* is a universally quantified variable. As a result, we can reduce the number of required cycles.

5. EXPERIMENTS

In this section, we show the experimental evaluation results of the basic and modified algorithms.

We used a discrete event simulation, where each agent maintains its own simulated clock. An agent's time is incremented by one simulated time unit whenever it performs one computation cycle. One cycle consists of reading all incoming messages, performing local computation, and sending messages. We assume that a message issued at time t is available to the recipient at time $t + 1$. We analyzed the performance for the number of cycles required to solve the problem.

We created problem instances based on the model described in [7]. In this model, the sequence of quantified variables Q is divided into three blocks. The first and third

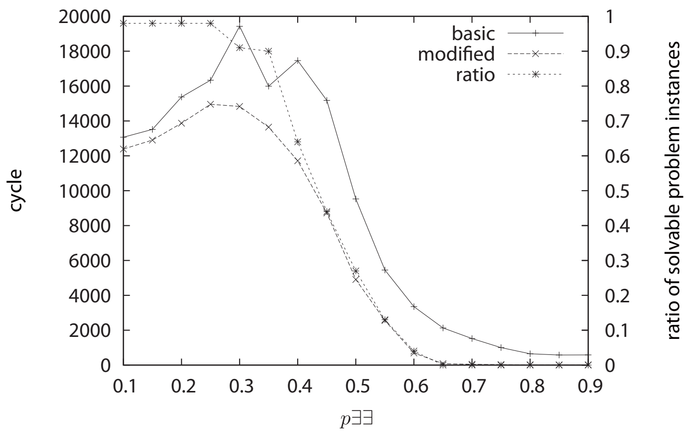

Figure 7: Required Cycles ($n = 15$, $n_\forall = 5$, $n_{pos} = 6$, $d = 5$, $p = 0.35$, $p_{\forall\exists} = 0.5$)

blocks consist of existentially quantified variables, and the second block consists of universally quantified variables. A problem instance is generated based on seven parameters, i.e., $<n, n_\forall, n_{pos}, d, p, p_{\exists\exists}, p_{\forall\exists}>$. The meanings of these parameters are as follows. n is the number of variables, $n_\forall$ is the number of universally quantified variables, and n_{pos} is the position of the first universally quantified variable in sequence Q. Furthermore, d represents the domain size, which is the same for all variables, and p represents the number of binary constraints as a fraction of all possible constraints. We assume a constraint is given as *nogoods*. $p_{\exists\exists}$ represents the number of *nogoods* in the form of $\exists x_i \exists x_j, c_{ij}$ as a fraction of all possible tuples. $p_{\forall\exists}$ is a similar quantity for $\forall x_i \exists x_j, c_{ij}$ constraints, described below. The other two types of constraints are not generated since they can be removed by preprocessing.

If many *nogoods* exist in the form of $\forall x_i \exists x_j, c_{ij}$, most problem instances are insolvable. To generate enough solvable instances, *nogoods* in the form of $\forall x_i \exists x_j, c_{ij}$ are restricted in the following way [7].

First, we generate a random total bijection from one domain to the other. All tuples that are not in this bijection are excluded in the *nogoods*. From this total bijection, choose $p_{\forall\exists}$ fraction of tuples as *nogoods*.

In these experiments, we chose the following parameters: $n = 15$, $n_\forall = 5$, $n_{pos} = 6$, $d = 5$, $p = 0.35$, $p_{\forall\exists} = 0.5$. Then we varied $p_{\exists\exists}$ from 0.10 to 0.90. Fig. 7 shows the required cycles of the basic and modified algorithms. Each data point represents the average of 100 problem instances.

We observed an easy-hard-easy transition by changing $p_{\exists\exists}$, but when $p_{\exists\exists}$ is small (i.e., the constraints are loose), the problem instances are relatively hard compared to the cases where $p_{\exists\exists}$ is large (i.e., the constraints are tight). This is because even if the constraints are loose, the team of agents still needs to consider all possible value assignments of the universally quantified variables.

We also observed that our modification of the basic algorithm is effective, in particular, when $p_{\exists\exists}$ becomes large, because the number of generated *nogoods* increases as the constraints become tight. In the hardest problem instances (where $p_{\exists\exists}$ is about 0.3), the modified algorithm can reduce cycles by about 25%.

6. CONCLUSION

In this paper, we introduced a generalization of a DisCSP called a quantified DisCSP, which formalizes a situation where a team of agents makes a plan against an adversary. Furthermore, we developed an algorithm for solving quantified DisCSPs, which is an extension of the classic asynchronous backtracking algorithm, and developed a method for improving this basic algorithm. Our experimental evaluation results illustrate that we observe an easy-hard-easy transition by changing the tightness of constraints, while very loose problem instances are relatively hard. The modification of the basic algorithm is effective and reduced cycles about 25% for the hardest problem instances.

Our future works include developing more efficient complete algorithms for quantified DisCSPs and developing a real-time algorithm that can make a decision to meet a deadline even if a complete solution cannot be found.

7. REFERENCES

[1] M. Benedetti. sKizzo: A suite to evaluate and certify QBFs. In *CADE*, pages 369–376, 2005.

[2] C. Bessiere, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. In *IJCAI DCR Workshop*, 2001.

[3] H. M. Chen. *The computational complexity of quantified constraint satisfaction*. PhD thesis, Cornell University, 2004.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[5] B. Faltings and S. Macho-Gonzalez. Open constraint programming. *Artificial Intelligence*, 161(1-2):181–208, 2005.

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

[7] I. P. Gent, P. Nightingale, and K. Stergiou. QCSP-Solve: A solver for quantified constraint satisfaction problems. In *IJCAI*, pages 138–143, 2005.

[8] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. In *IJCAR*, pages 364–369, 2001.

[9] K. Hirayama and M. Yokoo. The effect of nogood learning in distributed constraint satisfaction. In *ICDCS*, pages 169–177, 2000.

[10] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. John Wiley & Sons, 1992.

[11] R. Mailler and V. Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 25:529–576, 2006.

[12] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *AAAI*, pages 307–312, 1994.

[13] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.