Abstract. In this paper, we extend the traditional formalization of a 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 the nature or an adversary. A Quantified DisCSP formalizes a situation where a team of agents is trying to make a robust plan against the nature or an adversary. In this paper, we present the formalization of such a Quantified DisCSP and develop an algorithm for solving it. This algorithm generalizes 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 totally symmetrical.

1 Introduction

A constraint satisfaction problem (CSP) [1] 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, the research on CSPs has a long and distinguished history in the AI literature.

A distributed CSP (DisCSP) [2, 3] is a CSP in which variables and constraints are distributed among automated agents. Various application problems in multi-agent systems that are concerned with finding a consistent combination of agent actions (e.g., distributed resource allocation problems [4], distributed scheduling problems [5], and multi-agent truth maintenance tasks [6]) can be formalized as DisCSPs.

On the other hand, a quantified constraint satisfaction problem (QCSP) [7] 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 satisfies 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 as the choice of the nature or an adversary. A QCSP can formalize application problems such 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. In a traditional DisCSP, all variables are existentially quantified, and the values of these variables are determined by a cooperative agent. On the other hand, in a quantified DisCSP, some variables are universally quantified and the values of these variables are determined by an adversarial agent. Accordingly, a quantified DisCSP can formalize the problems that involve the nature or an adversary. In a quantified DisCSP, a team of agents tries to make a robust plan against the nature or an adversary to satisfy all constraints. Furthermore, we present an algorithm for solving a quantified DisCSP. This algorithm is a generalization of the asynchronous backtracking algorithm [3] for solving 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, in this algorithm, the procedures executed by an adversarial agent in response to a good and to a nogood are symmetrical to the procedures executed by a cooperative agent in response to a nogood and to a good.

2 Related Research

2.1 Quantified Boolean Formulas

A quantified boolean formulas (QBF) is a generalization of a SAT in which some variables can be universally quantified. The definitions of a SAT and a QBF are as follows.

Satisfiability Problem A SAT is the problem of finding a solution that satisfies a given boolean formula. A boolean formula in a SAT is 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 SAT was the first known NP-complete problem.

Quantified Boolean Formulas 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 a 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 $\forall x$).

$$\exists x_1 \forall x_2 \exists x_3 (x_1 \lor x_2) \land (x_2 \lor \neg x_3)$$ (1)

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

$$Q_1 x_1 \cdots Q_n x_n$$ (2)

Please note that the order in the sequence is important. For example, $\exists x_1 \forall x_2 F$ is not equal to $\forall x_2 \exists x_1 F$. More specifically, the meanings of $\exists x_1 \forall x_2 F$ and $\forall x_2 \exists x_1 F$ are as follows.

- $\exists x_1 \forall x_2 F$ is true : There exists a value of $x_1$ such that for every value of $x_2$ $F$ is true. Note that there must be a single value for $x_1$, where $F$ is true regardless of the value of $x_2$.
- $\forall x_2 \exists x_1 F$ is true : For every value of $x_2$, there exists a value of $x_1$ such that $F$ is true. Note that we can choose the value of $x_1$ according to the value of $x_2$.

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 a universally quantified variable $x_i$, if an 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., Quaffle [8] and Qube [9], which are based on the DPLL algorithm [10], Skizzo [11], which is based on the Skolemization technique, and so on.

### 2.2 Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) is a problem of finding an assignment of values to variables that satisfies constraints. Each variable takes a value from a finite, discrete domain. A CSP can represent various application problems in AI, such as scheduling, planning, and so on.

A CSP with $n$ variables and $m$ constraints can be described as follows.

- A set of variables $X = \{x_1, x_2, \cdots, x_n\}$
- A set of domains of variables $D = \{D_1, D_2, \cdots, D_n\}$
- A set of constraints $C = \{C_1, C_2, \cdots, C_m\}$

The algorithm for solving CSPs finds an assignment of values to variables that satisfies all constraints or shows that there exists no solution.
2.3 Quantified CSP

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

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

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 (x_1 \neq x_3) \land (x_1 < x_4) \land (x_2 \neq x_3)$$  \hspace{1cm} (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 $a$.

There are several algorithms for solving a QCSP, most of which are extensions of QBF-based algorithms. One notable exception is an algorithm called QCSP-Solve [12], which introduces techniques specialized to a QCSP.

2.4 Distributed CSP

A DisCSP 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 the address of the agents.
- For the 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, there exist inter-agent constraints, and the value assignment must satisfy these inter-agent constraints.

**Asynchronous Backtracking Algorithm** We make the following assumptions while describing the algorithms for simplicity.

- Each agent has exactly one variable.
- Each agent knows all constraints relevant to its variable.
- All constraints are binary.
Under the above assumptions, a DisCSP can be represented as a network. In a network, agents are nodes and constraints are links. We assume a link is directed. More specifically, for two agents that have 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, the agent executes a certain procedure for each message type. In this algorithm, the two following types of messages are used.

- \((\text{ok?}, (x_j, \text{value}))\) : this message informs that the value of \(x_j\) is \text{value}.
- \((\text{nogood}, x_j, \text{nogood})\) : this message informs a new \text{nogood}. A \text{nogood} is a combination of values that causes constraint violation. For example, \(\text{nogood}\{(x_i, d_i), (x_j, d_j)\}\) represents the fact 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 \text{ok?} message, agent \(x_i\) tries to find a consistent value with higher-priority agents. If there exists no consistent value, agent \(x_i\) sends a \text{nogood} message to the agent, which has the lowest priority among agents whose priorities are higher than agent \(x_i\). The asynchronous backtracking algorithm allows agents to act asynchronously and concurrently without any global control, while guaranteeing the completeness of the algorithm.

3 Quantified Distributed CSP

3.1 Problem Definition

A quantified DisCSP is a quantified CSP where variables and constraints are distributed among agents. We assume each existentially quantified variable is owned by a separate agent, while all universally quantified variables are controlled by a single adversary. Also, a sequence of quantified variables defines the order of decision making, i.e., 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 a team of agents, who have existentially quantified variables, tries to find a plan for determining their variables so that all constraints are satisfied, regardless of the value assignments of the adversary.

3.2 Example of Quantified Distributed CSP

Let us consider a game-based example of a quantified DisCSP. A game called Noughts and Crosses (or, Tic-Tac-Toe) is played on a \(3 \times 3\) board. The two
players take turns placing a marker on any free slot. The first player is crosses (×), followed by the other player’s noughts (◦). The aim is to make a straight line connecting three markers. Here, we assume noughts are placed by the adversary. Also, there exists a team of agents, each of which is responsible for one particular move of crosses.

This problem can be formalized as a quantified DisCSP as follows.

- Variables: $x_1, \ldots, x_9$
- Domain: each variable has a domain $\{1, 2, \ldots, 9\}$, where each value represents a position on the board.
- Constraint: noughts must not form a line earlier than crosses.
- Quantifier sequence: $\exists x_1 \forall x_2 \cdots \exists x_9$ ($\exists x_i$ iff $i$ is odd, $\forall x_i$ iff $i$ is even.)

Variable $x_i$ represents the move of crosses if $i$ is odd, and it represents the move of noughts if $i$ is even. A team of agents wants to find a plan which never loses, regardless of the plan of the adversary.

Here, we introduce auxiliary constraints that represent the fact that a marker cannot be placed on a position which is already filled. Furthermore, the adversary must follow these auxiliary constraints, even while trying to violate the normal constraints.

Then we introduce the constraint shown in (4). \text{rule\_adversary}_i is the rule which adversarial agent $i$ should obey, \text{rule\_team}_i is the rule which cooperative agent $i$ should obey, and \text{adversary\_wins} is negation of the winning condition of the team.

\[
\text{rule\_adversary}_2(x_1, x_2) \land \text{rule\_adversary}_4(x_1, x_2, x_3, x_4) \land \\
\text{rule\_adversary}_6(x_1, x_2, x_3, x_4, x_5, x_6) \land \text{rule\_adversary}_8(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \\
\rightarrow (\text{rule\_team}_1(x_1) \land \text{rule\_team}_3(x_1, x_2, x_3) \land \text{rule\_team}_5(x_1, x_2, x_3, x_4, x_5) \land \\
\text{rule\_team}_7(x_1, x_2, x_3, x_4, x_5, x_6, x_7) \land \text{rule\_team}_9(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) \\
\land (\neg \text{adversary\_wins})
\] (4)

This is the constraint that represents the fact that the team of cooperative agents can achieve the aim if they obey the rule, unless adversarial agents violate the rule. If adversarial agents violate the rule, the constraint is satisfied.

4 Algorithm for Solving Quantified Distributed CSP

In this section, we present an algorithm for solving quantified DisCSPs. This algorithm is based on the asynchronous backtracking algorithm.

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, i.e., if $x_j$ appears before $x_i$, then $x_j$ has a higher priority than $x_i$. Note that the ordering among existentially quantified variables, whose positions are adjacent in the quantifier sequence, can be determined arbitrarily.

For simplicity, we assume agents know a DFS tree, which is determined by the priority order.

There exists a virtual agent for each universally quantified variable. A virtual agent imitates the actions of the adversary 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 in off-line. When the team actually plays against the adversary, it executes the plan obtained in the off-line search. In reality, one of the agents in the team should act as a virtual agent. Any team member can act as a virtual agent for a universally quantified variable $x_i$. However, to reduce communication costs, it would be better that an agent who is the parent or child of $x_i$ acts 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, which satisfies constraints that are 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 nogood and good

In this subsection, we compare the procedures executed at existentially/universally quantified variables for good/nogood.

**nogood**: a nogood is a combination of values that represents a contradiction. For example, nogood $\{(x_1, 1), (x_2, 2)\}$ represents the fact that $x_1 = 1 \land x_2 = 2 \rightarrow \bot$, i.e., $x_1 = 1 \land x_2 = 2$ causes a contradiction.

**Generation of a nogood**: 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 $x_2 = 1 \lor x_2 = 2$, nogood $\{(x_1, 1), (x_2, 1)\}$, and nogood $\{(x_1, 1), (x_2, 2)\}$.
- Assume that $x_2$ is a universally quantified variable. A nogood $\{(x_1, 1)\}$ can be generated from nogood $\{(x_1, 1), (x_2, 1)\}$.

Accordingly, a cooperative agent who has 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 who has 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.
**good**: A good is a combination of values that satisfies some constraints. For example, good \{ (x_1, 1), (x_2, 2) \}, which is generated by agent \( x_3 \), represents the fact that \( x_1 = 1 \land x_2 = 2 \) satisfies all constraints of \( x_3 \) and its descendants.

**Generation of a good**: 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 \( x_1 = 1 \land x_2 = 1 \) satisfies the constraint between \( x_1 \) and \( x_2 \).
- Assume \( x_2 \) is a universally quantified variable. A good \{ \( x_1, 1 \) \} can be generated from the fact that \( x_2 = 1 \lor x_2 = 2 \), good \{ \( x_1, 1 \), \( x_2, 1 \) \} and good \{ \( x_1, 1 \), \( x_2, 2 \) \}, which are sent from its only child \( x_3 \), if \( x_1 = 1 \land x_2 = 1 \) and \( x_1 = 1 \land x_2 = 2 \) satisfy the constraint between \( x_1 \) and \( x_2 \).

Consequently, a cooperative agent who has an existentially quantified variable sends a good message immediately 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 via ok? message to all descendants. After an agent sends its value, it waits for an incoming message. If an agent receives a message, it executes a procedure defined for each message type.

There are three types of messages, i.e., ok, nogood, and good messages. The procedures executed by a cooperative agent upon receiving messages are shown in Fig. 1, while the procedures executed by an adversarial/virtual agent upon receiving messages are shown in Fig. 2. Fig. 3 describes the procedures of backtrack and send good, which are used in the above procedures.

The procedures for a cooperative agent are basically the same as the procedures in asynchronous backtracking, except for the procedures for sending/receiving a good message. As in the asynchronous backtracking algorithm, when a cooperative agent receives ok? messages or nogood messages, the agent updates its agent_view and nogood_list and checks whether its current value is consistent with its agent_view. On the other hand, 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. When a cooperative agent receives a good message, the agent updates its good_list and sends a good message to its parent if the agent has received good messages for the same combination of values from all children.

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 constraint. 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 an adversarial/virtual agent receives an **ok**? message, the agent sends a **nogood** message if it finds one value that causes constraint violation.

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 sends a **nogood** message to its parent immediately, since it can choose the value described in the received **nogood** and violate some constraint.

When a cooperative agent receives **good** messages from all children for the same combination of values, it sends a **good** message to its parent immediately, 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 so that some constraints might be violated. 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.

For simplicity, we describe the algorithm so that it only checks whether the problem is solvable or not. To construct a plan for acting against the adversary, an agent that has an existentially quantified variable must record **goods** sent from its children.

```
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;
```

**Fig. 1.** Procedures of a cooperative agent upon receiving messages
when received (ok?, (x, value)) do
  add (x, value) to agent_view;
  check_agent_view_adversary;
end do;

when received (nogood, x, 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, good) do
  add good to good_list;
  check_agent_view_adversary;
end do;

procedure check_agent_view_adversary;
  when for some v ∈ Di, v and agent_view are inconsistent do backtrack; end do;
  when for all v ∈ Di, 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 ∈ Di so that some children have not send good message yet;
    change current_value to v;
    send (ok?, (x, current_value)) to descendants; end do;
end procedure;

Fig. 2. Procedures of an adversarial/virtual agent upon receiving messages

4.4 Example

We illustrate an example of an algorithm execution in Fig. 4. The figure represents a problem that consists of $Q = ∃x_1 ∀x_2 ∃x_3 ∃x_4, C = \{\text{nogood}\{(x_1, 1), (x_3, 1)\}, \text{nogood}\{(x_2, 1), (x_3, 2)\}, \text{nogood}\{(x_2, 2), (x_4, 1)\}\}$, and $D_1 = D_2 = D_3 = D_4 = \{1, 2\}$.

In Fig. 4 (b), after receiving ok? messages from x1 and x2, the agent_view of x3 and x4 will be \{(x_1, 1), (x_2, 1)\}. Since there exists no value for x3 that is consistent with this agent_view, x3 sends a nogood message to x2, who is the parent of x3. Also, since there exists a value consistent with this agent_view for x4, x4 sends a good message to its parent x2 (Fig. 4 (c)).

After receiving this nogood message and this good message, x2 sends a nogood message to x1 because x2 is an adversarial agent (Fig. 4 (d)).

After receiving this nogood message, x1 knows that $x_1 = 1$ causes constraint violation. Thus, $x_1$ changes its value to 2 and sends ok? messages to its descendants (Fig. 4 (e)).

By receiving this ok? message, x2, x3, x4 record this value to their agent_view. x3 sends a good message to x2 because $x_3 = 1$ satisfies its constraint. x4 also sends a good message to x2 because x4 can select a value that satisfies its constraint (Fig. 4 (f)).
procedure **backtrack**

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

procedure **send_good**

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

Fig. 3. Procedure for **backtrack** and **send_good**

Then x2 has received good messages for its value 1 from all children. Thus, x2 selects another value 2 (so that it might violate some constraints) and sends ok? messages (Fig. 4 (g)).

x3 and x4 receive this ok? message and record the value to their agent_view. Since both x3 and x4 can select a value that satisfies constraints, x3 and x4 send good messages to x2 (Fig. 4 (h)). After receiving these good messages, x2 sends a good message to x1 because every value for x2 satisfies related constraints (Fig. 4 (i)). Since x1 is a cooperative agent, x1 can select the current value. Thus, an empty good is generated. As a result, it is shown that this problem has a solution.

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. Thus, when an empty good is generated, the problem is solvable. Otherwise, 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 Theorem 1 and Theorem 2, respectively.
Fig. 4. Example of an algorithm execution
Theorem 1. If an empty good is generated in this algorithm, then the problem is solvable. On the other hand, 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. Consequently, when an empty nogood is generated, the problem is unsolvable.

We prove this fact by mathematical induction.

First, for the base case, we show that the procedure of a leaf agent is correct. A leaf agent receives only ok? messages. If the leaf agent is a cooperative agent, this agent generates a good and sends it to its parent only when it can select a value that is consistent with its agent_view. Furthermore, 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 there exists no value that causes constraint violations. Moreover, it generates a nogood and sends it to the parent if there exists at least one value that causes some constraint violations. It is clear that the generated good/nogood are correct.

Now, for the inductive case, let us assume that for agent $x_i$, all received good/nogood from its descendants are correct. Then, 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 $v \in D_i$, where it receives good messages from all children and the good is consistent with $x_i = v$ and agent_view, and $x_i = v$ 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 there exists at least one value $v \in D_i$, 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. Then, 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, then 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 mathematical induction. For the base case, it is clear that a leaf agent never fails to send a good or nogood message to its parent.

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 message 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 domain of the variable of the root 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.

5 Conclusion

In this paper, we introduced a generalization of a DisCSP called a quantified DisCSP, which formalizes a situation where a team of agents make a plan against
an adversary. Furthermore, we developed an algorithm for solving quantified DisCSPs, which is an extension of the classic asynchronous backtracking algorithm.

Issues for future study include extending the formalization to distributed constraint optimization problems and developing more efficient algorithms for quantified DisCSPs.

References