Distributed Partial Constraint Satisfaction Problem

Katsutoshi Hirayama[1] and Makoto Yokoo[2]

[1] Kobe University of Mercantile Marine
5-1-1 Fukae-minami-machi, Higashinada-ku, Kobe 658, JAPAN
E-mail: hirayama@ti.kshosen.ac.jp
[2] NTT Communication Science Laboratories
2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02, JAPAN
E-mail: yokoo@cslab.kecl.ntt.co.jp

Abstract. Many problems in multi-agent systems can be described as *distributed Constraint Satisfaction Problems* (distributed CSPs), where the goal is to find a set of assignments to variables that satisfies all constraints among agents. However, when real problems are formalized as distributed CSPs, they are often over-constrained and have no solution that satisfies all constraints. This paper provides the *Distributed Partial Constraint Satisfaction Problem* (DPCSP) as a new framework for dealing with over-constrained situations. We also present new algorithms for solving *Distributed Maximal Constraint Satisfaction Problems* (DM-CSPs), which belong to an important class of DPCSP. The algorithms are called the *Synchronous Branch and Bound* (SBB) and the *Iterative Distributed Breakout* (IDB). Both algorithms were tested on hard classes of over-constrained random binary distributed CSPs. The results can be summarized as SBB is preferable when we are mainly concerned with the optimality of a solution, while IDB is preferable when we want to get a nearly optimal solution quickly.

1 Introduction

Many problems in AI can be formalized as *Constraint Satisfaction Problems* (C-SPs), and many researchers have investigated the problems and their algorithms for many years. However, as AI has begun to encounter more realistic problems in the real world, we have found that certain kind of problems in the real world cannot be handled in the conventional CSP framework, and several studies have been made in order to extend the traditional CSP framework.

In [14], Yokoo *et al.* presented a *distributed Constraint Satisfaction Problem* (distributed CSP) as the general framework for dealing with problems in multi-agent systems. A distributed CSP can be considered a CSP in which variables and constraints are distributed among multiple agents and the agents are required to satisfy all constraints by communicating with each other. Many problems in multi-agent systems, such as distributed interpretation problems[9], distributed resource allocation problems[3], distributed scheduling problems[11], and multi-agent truth maintenance systems[7], can be formalized as distributed CSPs.

On the other hand, when a problem designer tries to describe a real problem as a CSP, the resulting CSP is often over-constrained and has no solutions. For such an over-constrained CSP, almost all conventional CSP algorithms just produce a result that says there is no solution. If we are interested in solutions for practical use, the designer has to go back to the design phase and to find another design so that the CSP is not over-constrained. Freuder extended the CSP framework and provided a *Partial Constraint Satisfaction Problem* (PCSP), which is one of the approaches to over-constrained CSPs[4]. In a PCSP, we are required to find consistent assignments to an allowable relaxed problem.

Although a distributed CSP and a PCSP extend the traditional CSP framework in different directions, they are not mutually exclusive. It is not only possible to combine these extensions, but also beneficial because the problems in multi-agent systems can also be over-constrained. This paper provides a formal framework for over-constrained distributed CSPs, the *Distributed Partial Constraint Satisfaction Problem* (DPCSP), and presents two algorithms for solving *Distributed Maximal Constraint Satisfaction Problems* (DMCSPs), which belong to an important class of DPCSP. These algorithms are called the *Synchronous Branch and Bound* (SBB) and the *Iterative Distributed Breakout* (IDB).

This paper is organized as follows. Section 2 and Section 3 introduce the definition of a distributed CSP and a PCSP, respectively, and Section 4 defines a DPCSP and a DMCSP. Algorithms for solving DMCSPs are presented in Section 5, and Section 6 presents an experimental evaluation on randomly generated over-constrained distributed CSPs. Conclusions are given in Section 7.

2 Distributed Constraint Satisfaction Problem

A CSP consists of a pair (V, C), where V is a set of variables, each with a finite and discrete domain, and C is a set of constraints. The domain of a variable is a set of values, each of which can be assigned to the variable. Each constraint is defined over some subset of variables and limits the allowed combinations of variable values in the subset. Solving a CSP involves finding one set of assignments to variables that satisfies all constraints. In some cases, the goal is to find all sets of such assignments.

A distributed CSP can be considered a CSP in which variables and constraints are distributed among multiple agents. To put it formally,

- there exists a set of agents, $1, 2, \ldots, m$;
- for each variable x_j, an agent i is defined such that x_j belongs to i. We mean x_j belongs to i by belongs(x_j, i);
- a constraint C_l is known by an agent i. The predicate known(C_l, i) is used to express that.

We assume, in general, that an agent knows only those constraints relevant to the variables that belong to it. Note that some constraints known by an agent may include other agents' variables, not just its own variables. We refer to such a constraint as an *inter-agent constraint*. A distributed CSP is solved when the following conditions are satisfied for all agents. For each agent i,

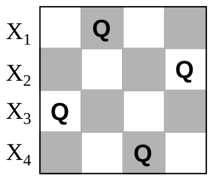

Fig. 1. Example of CSPs (4-queens problem)

– a variable x_j has a value d_j as its assignment for $\forall x_j$ belongs(x_j, i);
– a constraint C_l is true under the above assignments for $\forall C_l$ known(C_l, i).

Fig. 1 illustrates a 4-queens problem, which is a typical CSP. When we view this as a problem where each of four agents tries to determine each queen's position independently, this problem can be described as a distributed CSP.

The algorithms for distributed CSPs must find a solution as quickly as possible. An agent in a distributed CSP has only limited knowledge of the entire problem, and thus important things for the algorithms include how agents communicate with each other and what information is transferred.

3 Partial Constraint Satisfaction Problem

A PCSP is formally described as the following three components[4]:

$$\langle (P, U), (PS, \leq), (M, (N, S)) \rangle,$$

where P is a CSP, U is a set of 'universes', i.e., a set of potential values for each variable in P, $(PS, \leq)$ is a problem space with PS a set of CSPs and $\leq$ a partial order over PS, M is a distance function over the problem space, and (N, S) are necessary and sufficient bounds on the distance between P and some solvable member of PS. We leave the details of each component to [4] due to space limitations. A *solution* of a PCSP is a solvable problem P' from the problem space and its solution, where the distance between P and P' is less than N. Any solution will suffice if the distance between P and P' is not more than S, and all search can terminate when such a solution is found. An *optimal solution* to a PCSP is defined as a solution with a minimal distance between P and P', and the minimal distance is called an *optimal distance*.

4 Distributed Partial Constraint Satisfaction Problem

4.1 Motivation

It is likely that various application problems in multi-agent systems are over-constrained.

In a distributed interpretation problem[9], each agent is assigned a task to interpret a part of sensor data, produce possible interpretations, and help build a globally consistent interpretation through communicating possible interpretations among all of the agents. If an agent makes incorrect interpretations because of errors in the process—for example, noise on the sensor data—, there may be a situation where no globally consistent interpretation exists.

Multi-stage negotiation[3] is a kind of distributed resource allocation problem. Each agent in this problem has a goal (variable) and possible plans to achieve the goal (domain of the variable), and there can be resource conflicts between plan executions by different agents (constraints). The goal of this problem is to find a combination of plans that achieve the goals of all agents at a certain time. It is likely that all the goals cannot be achieved without violating some constraints if not enough resources are available.

While the ordinary distributed CSP framework does require satisfaction of all constraints among agents, it does not give any indication of how we should handle over-constrained distributed CSPs. Thus it makes sense to extend the distributed CSP framework to enable handling of over-constrained distributed CSPs. In [13], Yokoo proposed a method for over-constrained distributed CSPs by introducing *constraint hierarchy*[1] and relaxing the less important constraints if there exists no solution. This method can be applied to problems where constraints are hierarchically structured. However, we recognize that constraints are not always hierarchically structured, and this method is thus unsatisfactory for covering all problems. This research provides a new framework, called the DPCSP, for handling over-constrained distributed CSPs.

4.2 Definition

A DPCSP is formalized as:

- a set of agents, $1, 2, \ldots, m$;
- a PCSP for each agent i, $\langle (P_i, U_i), (PS_i, \leq), (M_i, (N_i, S_i)) \rangle$;
- a global distance function, G,

where P_i is agent i's original CSP that consists of variables belonging to i and constraints that are known by i, U_i is a set of 'universes', i.e., a set of potential values for each variable in P_i, $(PS_i, \leq)$ is a problem space for agent i with PS_i a set of CSPs and $\leq$ a partial order over PS_i, M_i is i's distance function over the problem space, and (N_i, S_i) are i's necessary and sufficient bounds on the distance between P_i and some solvable member of PS_i. The purpose of agent i is to find a solvable CSP, P'_i, from the problem space PS_i and its solution, where the distance between P_i and P'_i is less than N_i. Any solution will suffice for agent i if the distance between P_i and P'_i is not more than S_i.

A DPCSP is solved when each of the agents, say i, finds a solvable CSP from the problem space and its solution, such that the distance d_i between the solvable CSP and the original CSP is less than N_i. We refer to such solvable CSPs and their solutions as a *solution* to the DPCSP. Any solution to a DPCSP will suffice if every solution to an individual PCSP suffices. For each solution to a DPCSP, we define a global distance function $G(d_1, d_2, \ldots, d_m)$, which returns the distance of the solution. Using this function, an *optimal solution* is defined as the solution with a minimum distance, and we call the minimum distance an *optimal distance*.

In this paper, we specify the above setting for a DPCSP as follows:

- for each agent i, CSPs in PS_i are produced by removing possible combinations of constraints from P_i;
- the distance between P_i and P'_i (a solvable CSP in PS_i) is measured as the number of constraints removed from P_i;
- a global distance function, $G(d_1, d_2, \ldots, d_m)$, is specified by $\max_i d_i$.

We call this class of DPCSP *Distributed Maximal Constraint Satisfaction Problems* (DMCSPs). The goal of agent i for a DMCSP is to find a solvable CSP and its solution with the number of removed constraints less than N_i. To put it another way, the goal is to find assignments to the variables in P_i with the number of violated constraints in P_i less than N_i.

A DMCSP is solved when each agent, say i, finds assignments with the number of violated constraints less than N_i. We refer to the set of assignments as a *solution* to the DMCSP. Among solutions to the DMCSP, it is the *optimal solution* that minimizes $\max_i d_i$, where d_i is the number of violated constraints on P_i. We call such minimal value of $\max_i d_i$ an *optimal distance* for the DMCSP. An optimal solution to a DMCSP ensures that we cannot find a solution to the DMCSP, where each agent has assignments with the number of violated constraints less than the optimal distance.

A DMCSP seems to be a reasonable and important class of DPCSP, but we could define other classes of DPCSP. Those classes may include the one that consists of the same definition as a DMCSP except for G, for example, using $\sum_{i=1}^{m} d_i$ instead of $\max_i d_i$ for G. This class is designed to get an optimal solution with the total number of violated constraints over agents minimized. However, it allows an optimal solution in which the number of violated constraints is globally minimized while the violated constraints are concentrated on specific agents. We suppose that might not be a preferable feature for multi-agents systems in terms of equality among agents.

4.3 Example

Fig. 2 shows a distributed 2-coloring problem to illustrate a DMCSP. A node represents a variable and an agent that has the variable. An edge represents a constraint, which means the two connected nodes must be painted in different colors (black or white). An agent knows only the constraints that are relevant to its variable. For example, agent 1 knows only $\{a, c, d\}$. The original CSP for agent 1 (i.e., P_1) consists of a variable: $\{1\}$ with a domain of {black, white} and constraints: $\{a, c, d\}$. The current distance of agent 1 is one because it just violates the constraint d, and for other agents: one for agent 2, agent 5 and agent

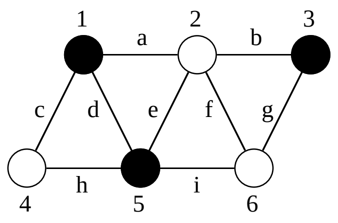

Fig. 2. Distributed 2-coloring problem

6, and zero for agent 3 and agent 4. Suppose $\forall i N_i = 2$, the DMCSP is already solved since the current distances for all agents are less than N_i. The DMCSP with $\forall i N_i = 1$, however, is not solved in this figure because the distances for agent 1, agent 2, agent 5, and agent 6 are not less than N_i. Note that there is no set of assignments that makes the maximal number of constraint violations over agents less than one, and thus the set of assignments in Fig. 2 is the optimal solution to the DMCSP where the optimal distance is one.

5 Algorithms for Distributed Maximal Constraint Satisfaction Problems

In this paper, we develop algorithms that find the optimal solution of a DMCSP.

The simplest algorithms for DMCSPs belong to the class called *centralized algorithms*. One of the centralized algorithms follows this procedure: agents run some leader election algorithm to elect one leader; send all distributed PCSPs to the leader; the leader solves those gathered PCSPs using some maximal constraint satisfaction algorithm[5], while others are idle. If we were interested only in efficiency and not in other aspects, the centralized algorithms might outperform other algorithms because they can make better use of the global knowledge of the entire problem. However, we believe such algorithms are not suitable for a distributed environment from a privacy and security standpoint (who on earth wants to expose an individual's schedule or private information to others?). We therefore develop algorithms on the assumption that each agent's knowledge of the entire problem should remain limited throughout the execution of the algorithms. The algorithms we present in this paper are the *Synchronous Branch and Bound* (SBB) and the *Iterative Distributed Breakout* (IDB).

5.1 Assumption

First of all, we assume the following conditions on communication among agents. These assumptions are quite reasonable for asynchronous communication systems.

- One agent sends messages to the others directly if it knows their addresses. At first an agent knows only the addresses of *neighbors*, a set of agents which share the same inter-agent constraints.
- Although the delay in delivering a message is finite, an upper bound is unknown.
- Between any two agents, messages are received in the order in which they were sent.

Next we introduce the following restrictions on problems. These are just for simplicity, and we can easily generalize our method for larger contexts.

- Each agent has exactly one variable.
- All constraints are binary, i.e., defined over two variables.

5.2 Synchronous Branch and Bound

The Synchronous Branch and Bound (SBB) is a simple algorithm that simulates the branch and bound method for Max-CSPs[5] in a distributed environment. In SBB, variable/agent and value ordering are fixed in advance, and a *path*, partial assignments for all variables, is exchanged among agents to be extended to a complete path. This extension process runs sequentially. To be concrete:

- the first agent in the ordering initiates the algorithm by sending a path that contains only its first value to the second agent;
- when receiving a path from the previous agent in the agent ordering, an agent evaluates the path and the first value of its domain in value ordering, and then sends the path plus the value as a new path to the next agent if its evaluation value is less than the current upper bound, or continues to try next values if the evaluation value is not less than the bound. If values are exhausted, it backtracks to the previous agent by returning the path;
- when receiving a path from the next agent in the agent ordering, an agent changes its assignment to the next value in its value ordering, reevaluates the new path, and sends it to the next if its evaluation value is less than the bound, or if not, continues to try next values. Another backtrack takes place if values are exhausted.

An element of a path actually consists of a variable, a value for the variable, and the number of constraint violations caused by the value. We measure the evaluation value of a path as the maximal number of constraint violations over the variables on the path, and the upper bound as the minimum evaluation value over those of complete paths found so far. Details of SBB are shown in Fig. 3.

Since SBB just simulates the branch and bound method in a distributed environment, it appears obvious that SBB is correct. Soundness is guaranteed since SBB terminates *iff* finding a complete path whose evaluation value is not more than a uniform initial value of S_i or finding no such complete path exists. With sequential control over agents and fixed variable/value orderings, SBB enables agents to do an exhaustive search in distributed search spaces. This

```
procedure **initiate** /* *done only by the first agent for starting the algorithm* */
    $d_i \leftarrow$ the first value in *domain*;
    $n_i \leftarrow$ known upper bound; *previous_path* $\leftarrow$ nil;
    send (**token**, $[[x_i, d_i, 0]]$, $n_i$) to the next agent;

**when** $i$ received (**token**, *current_path*, *ub*) from the previous agent **do**
    *previous_path* $\leftarrow$ *current_path*; $n_i \leftarrow ub$;
    *next* $\leftarrow$ **get_next**(*domain*);
    **send_token**; **end do**;

**when** $i$ received (**token**, *current_path*, *ub*) from the next agent **do**
    $[x_i, d_i, nv_i] \leftarrow$ the element related to $x_i$ in *current_path*; $n_i \leftarrow ub$;
    *next* $\leftarrow$ **get_next**(*domain* minus all elements up to $d_i$);
    **send_token**; **end do**;

procedur **send_token**
    **if** *next* $\neq$ nil **then**
        **if** $i$ = the last agent **then**
            *next_to_next* $\leftarrow$ *next*;
            **while** *next_to_next* $\neq$ nil **do**
                **when** max $nv_j$ in *new_path* $< n_i$ **do**
                    $n_i \leftarrow$ max $nv_j$ in *new_path*;
                    *best_path* $\leftarrow$ *new_path*; **end do**;
                **when** $n_i = 0$ **do**
                    terminate the algorithm; **end do**;
                *next_to_next* $\leftarrow$ **get_next**(*domain* minus all elements up to *next_to_next*);
            **end do**;
            send (**token**, *previous_path*, $n_i$) to the previous agent;
        **else**
            send (**token**, *new_path*, $n_i$) to the next agent; **end if**;
    **else**
        **if** $i$ = the first agent **then**
            terminate the algorithm;
        **else**
            send (**token**, *previous_path*, $n_i$) to the previous agent; **end if**; **end if**;

procedure **get_next**(*value_list*)
    **if** *value_list* = nil **then**
        **return** nil;
    **else**
        $d_i \leftarrow$ the first value in *value_list*; *new_path* $\leftarrow$ nil; *counter* $\leftarrow 0$;
        **if check**(*previous_path*) **then**
            **return** $d_i$;
        **else**
            **return get_next**(*value_list* minus $d_i$); **end if**; **end if**;

procedure **check**(*path*)
    **if** *path* = nil **then**
        add $[x_i, d_i, counter]$ to *new_path*;
        **return** *true*;
    **else**
        $[x_j, d_j, nv_j] \leftarrow$ the first element in *path*;
        **if** $[x_i, d_i]$ and $[x_j, d_j]$ are not consistent **then**
            *counter* $\leftarrow$ *counter* $+ 1$;
            **if** *counter* $\geq n_i$ or $nv_j + 1 \geq n_i$ **then**
                **return** *false*;
            **else**
                add $[x_j, d_j, nv_j + 1]$ to *new_path*;
                **return check**(*path* minus the first element); **end if**;
        **else**
            add $[x_j, d_j, nv_j]$ to *new_path*;
            **return check**(*path* minus the first element); **end if**; **end if**;
```

Fig. 3. Synchronous branch and bound. Variable(agent) and value ordering are given in advance, and both n_i and s_i for $\forall i$ have uniform values as their initial values. The initial value of s_i should be zero when searching for an optimal solution. The procedure should be initiated only by the first agent in the ordering.

ensures that SBB is complete, i.e., it eventually finds a sufficient solution or finds that there exists no such solution and terminates.

On the other hand, SBB does not allow agents to assign or change their variable values in parallel, and thus SBB cannot take advantage of parallelism.

5.3 Iterative Distributed Breakout

Outline We developed the *distributed breakout*[15] for solving distributed C-SPs. This method is characterized by hill-climbing in parallel while excluding neighbors' simultaneous action[6] and the *breakout* method[10] as a strategy for escaping from *quasi-local-minima*. In the distributed breakout, each agent first initializes its assignment arbitrarily, sends its assignment to neighbors with *ok*? messages, and then repeats the following:

- when knowing the current assignments of neighbors by receiving *ok*? messages, an agent evaluates its current assignment by counting the number of violated constraints and also measures the possible improvement of an evaluation value (called *improve*) if the agent changed the assignment to another. The value of *improve* is sent to neighbors with *improve* messages;
- when knowing the current *improves* of neighbors by receiving *improve* messages, an agent compares each of them with its own *improve*, and transfers the right to change an assignment by skipping its next change if the neighbor's *improve* is greater than its own *improve* or does not transfer this right if it's smaller. Ties are broken by comparing agent identifiers. Only the winners for the right to change actually change their assignments, and then all agents send the current assignments to neighbors with *ok*? messages.

This repeated process sometimes leads to a solution to a distributed CSP. However, it often gets stuck when some agent falls into a quasi-local-minimum, where it has at least one constraint violation and has no way to reduce the number of constraint violations. The distributed breakout provides an efficient way to escape from such a quasi-local-minimum. It just increases weights of violated constraints at a quasi-local-minimum and changes an assignment by evaluating the assignment as a weighted sum of violated constraints.

While each agent in the distributed breakout synchronizes its assignment change among neighbors, the overall assignment changes run in parallel. The method is thus especially efficient for critical problem instances with solutions. Another advantage is that it incorporates a procedure to detect whether the algorithm finds a global solution, in contrast with previous distributed constraint satisfaction algorithms that need to invoke the *snapshot* algorithm[2] for detection. On the other hand, one major drawback is that the distributed breakout is not complete, i.e., it may fail to find a solution even if one exists and also cannot determine that no solution exists.

The Iterative Distributed Breakout (IDB) is a method for DMCSPs in which a variant of the distributed breakout is repetitively applied to a DMCSP. The operation of IDB is: set a uniform constant value *ub* to each agent's necessary bound N_i and run the distributed breakout; if the distances of all agents become less than N_i, the agent that detects this fact sets its N_i to $ub-1$ and propagates its value to make N_i for all agents $ub-1$. This process is continued until some agent detects that a solution to a DMCSP with $\forall i\ N_i = S_i + 1$ is found.

Detail IDB is very similar to the distributed breakout. It does, however, introduce some extension for handling necessary bounds on distance. The bounds are exchanged by *ok?* and *improve* messages, both of which are also used in the distributed breakout. This paper focuses on the part that handles the necessary bounds and leaves details about the other parts, which are the same as in the distributed breakout, to [15].

- Before starting IDB, an agent i assigns a uniform value, say ub, to its necessary bound N_i. We currently give each agent a predetermined value.
- When receiving *ok?* messages from all neighbors, an agent i counts the number of violated constraints and then sets zero as the evaluation value of its current assignment if the number is less than N_i or, if not, the agent proceeds as in the distributed breakout. IDB thus permits an agent to have an assignment with the number of violated constraints less than N_i.
- For the distributed breakout, it is guaranteed that each agent is satisfied when some agent's *termination_counter* exceeds *diameter* (a diameter of graph). It is also guaranteed for IDB that each agent finds a solution to its individual PCSP with N_i when some agent's *termination_counter* exceeds *diameter*. The agent that finds this fact decreases its N_i by one and sends the new value with *improve* messages.
- An agent in the distributed breakout sets true to the variable *consistent iff* the numbers of violated constraints in itself and its all neighbors are zero. An agent in IDB, on the other hand, sets true to *consistent iff* the numbers of violated constraints in itself and its all neighbors are less than the necessary bounds.

The details of IDB are shown in Fig. 4.

We can prove inductively that the termination detection of each iteration of IDB is correct by the following fact: some agent i with $N_i = ub$ increases its *termination_counter* from d to $d+1$ *iff* each of i's neighbors has ub as the value of its necessary bound, has an assignment with the number of violated constraints less than ub, and has a *termination_counter* value of d or more.

While SBB is sequential in terms of value assignments, IDB enables parallel value assignments. However, IDB is not complete, i.e., it may fail to get an optimal solution to a DMCSP; besides, it cannot decide whether a solution is optimal or not even if it actually gets an optimal solution.

6 Evaluation

This section presents an experimental evaluation of SBB and IDB.

```
procedure **initiate** /* *done by every agent for starting the algorithm* */
    $current\_value \leftarrow$ the value randomly chosen from $domain$;
    $n_i \leftarrow$ known upper bound; $my\_termination\_counter \leftarrow 0$; $counter \leftarrow 0$;
    send (**ok?**, $x_i$, $current\_value$) to neighbors;
    **goto wait_ok? mode**;

**wait_ok? mode**
**when** $i$ received (**ok?**, $x_j$, $d_j$) **do**
    $counter \leftarrow counter + 1$; add ($x_j$, $d_j$) to $agent\_view$;
    **if** $counter = \#$ of neighbors **then**
        **send_improve**; $counter \leftarrow 0$;
        **goto wait_improve mode**;
    **else**
        **goto wait_ok? mode**; **end if**; **end do**;

procedure **send_improve**
    **if** # of currently violated constraints $< n_i$ **then**
        $current\_eval \leftarrow 0$;
    **else**
        $current\_eval \leftarrow$ weighted sum of violated constraints; **end if**;
    $my\_improve \leftarrow$ possible maximal improvement;
    $new\_value \leftarrow$ the value which gives the maximal improvement;
    **if** $current\_eval = 0$ **then**
        $consistent \leftarrow true$;
    **else**
        $consistent \leftarrow false$; $my\_termination\_counter \leftarrow 0$; **end if**;
    **if** $my\_improve > 0$ **then**
        $can\_move \leftarrow true$; $quasi\_local\_minimum \leftarrow false$;
    **else**
        $can\_move \leftarrow false$;
        **if** $current\_eval = 0$ **then**
            $quasi\_local\_minimum \leftarrow false$;
        **else**
            $quasi\_local\_minimum \leftarrow true$; **end if**; **end if**;
    send (**improve**, $x_i$, $my\_improve$, $current\_eval$, $my\_termination\_counter$, $n_i$) to neighbors;

**wait_improve mode**
**when** $i$ received (**improve**, $x_j$, $improve$, $eval$, $termination\_counter$, $ub$) **do**
    $counter \leftarrow counter + 1$;
    $my\_termination\_counter \leftarrow \min(termination\_counter, my\_termination\_counter)$;
    **when** $n_i \neq ub$ **do**
        $n_i \leftarrow \min(n_i, ub)$; $consistent \leftarrow false$; **end do**;
    **when** $improve > my\_improve$ **do**
        $can\_move \leftarrow false$; $quasi\_local\_minimum \leftarrow false$; **end do**;
    **when** $improve = my\_improve$ and $x_j$ precedes $x_i$ **do**
        $can\_move \leftarrow false$; **end do**;
    **when** $eval > 0$ **do**
        $consistent \leftarrow false$; **end do**;
    **if** $counter = \#$ of neighbors **then**
        **send_ok**; $counter \leftarrow 0$; clear $agent\_view$;
        **goto wait_ok? mode**;
    **else**
        **goto wait_improve mode**; **end if**; **end do**;

procedure **send_ok**
    **when** $consistent = true$ **do**
        $my\_termination\_counter \leftarrow my\_termination\_counter + 1$;
        **when** $my\_termination\_counter =$ diameter of the constraint graph **do**
            $n_i \leftarrow$ current global distance;
            **if** $n_i = 0$ **then**
                notify neighbors that a solution has been found; terminate the algorithm;
            **else**
                $my\_termination\_counter \leftarrow 0$; **end if**; **end do**; **end do**;
    **when** $quasi\_local\_minimum = true$ **do**
        increase the weights of violated constraints; **end do**;
    **when** $can\_move = true$ **do**
        $current\_value \leftarrow new\_value$; **end do**;
    send (**ok?**, $x_i$, $current\_value$) to neighbors;
```

Fig. 4. Iterative distributed breakout. Both n_i and s_i for $\forall i$ have uniform values as their initial values. The initial value of s_i should be zero when searching for an optimal solution. The procedure should be initialized by each agent before receiving any message.

problem class	median cycle	mean optimal distance
$\langle 10, 10, 18/45, 0.8\rangle$	3500	1.0
$\langle 10, 10, 18/45, 0.9\rangle$	18262	2.0
$\langle 10, 10, 27/45, 0.8\rangle$	46247	2.4
$\langle 10, 10, 27/45, 0.9\rangle$	499841	3.4
$\langle 10, 10, 36/45, 0.8\rangle$	336416	3.6
$\langle 10, 10, 36/45, 0.9\rangle$	1985700	5.0
$\langle 10, 10, 45/45, 0.8\rangle$	3435984	4.9
$\langle 10, 10, 45/45, 0.9\rangle$	21834077	6.0

Table 1. Median cycles for the Synchronous Branch and Bound for finding an optimal solution.

We tested both methods on *random binary distributed CSPs*, which are described as $\langle n, m, p_1, p_2\rangle$. One problem instance was generated by distributing variables and constraints of an instance of *random binary CSPs* with those 4 parameters. We distributed them such that each agent has exactly one variable and constraints relevant to the variable. The parameters of random binary CSPs are: n is the number of variables; m is the number of values for each variable; p_1 is the proportion of variable pairs that are constrained; and p_2 is the proportion of prohibited value pairs between two constrained variables. When generating an instance of random binary CSPs with $\langle n, m, p_1, p_2\rangle$, we randomly selected $n(n-1)p_1/2$ pairs of variables, and for each variable pair we set up a constraint such that randomly selected m^2p_2 pairs of values are prohibited.

In the experiments, we chose classes of random binary CSPs with $n = m = 10$, p_1 taking values from $\{18/45, 27/45, 36/45, 45/45\}$, and p_2 from $\{0.8, 0.9\}$. These classes of problems are known to be relatively hard ones for Max-CSPs[8]. Accordingly, we believe that they are suitable for the problems used to evaluate the methods.

Both SBB and IDB are implemented on a discrete event simulator that simulates concurrent activities of multiple agents. On the simulator, there exists a virtual agent called *manager*, which maintains a simulated clock and delivers messages among agents. One cycle of computation consists of: the manager gathers all messages issued by agents, increments one time unit (called *cycle*), and sends the messages to corresponding agents; agents then do their local computation and send messages. We evaluate the cost of algorithms in terms of cycles.

6.1 Cost of Finding an Optimal Solution

Since it is guaranteed that SBB finds an optimal solution, we can measure SBB's cost of finding an optimal solution as cycles to be consumed until SBB finds it. In the experiments, we applied SBB to each of 25 instances randomly generated for each class of problem. Note that we used *conjunctive width*

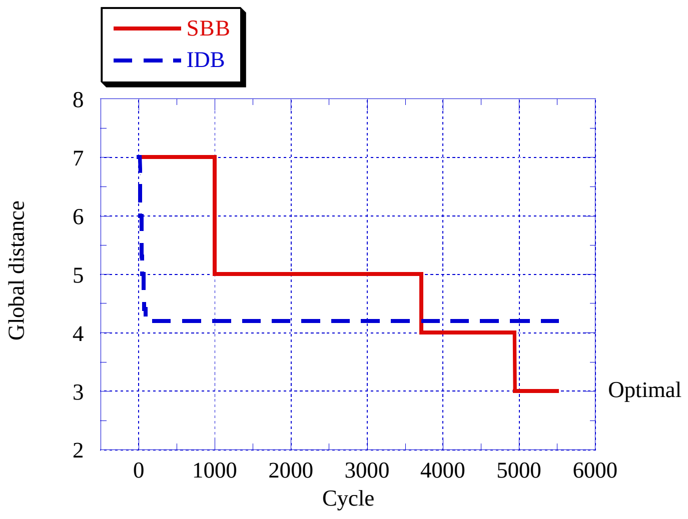

Fig. 5. Anytime curve for an instance of $\langle 10, 10, 27/45, 0.9\rangle$

heuristics(width/domain-size)[12] for variable ordering and lexical order for value ordering. Also note that the initial value of $\forall i N_i$ was set to the value of the maximum degree of a constraint graph minus one, and that of $\forall i S_i$ was zero. Table 1 illustrates the median cycle for finding an optimal solution and the mean optimal distance over 25 instances for each class. The cost of finding an optimal solution by SBB clearly seems to be very high.

On the other hand, IDB may fail to get an optimal solution, as stated above, and thus we cannot measure the cost in terms of cycles. However, we conducted an experiment to determine how often IDB fails to get an optimal solution. In this experiment, we ran 10 trials of IDB with randomly chosen initial assignments for each of 25 instances for the class of $n = m = 10, p_1 = 27/45, p_2 = 0.8$ (250 trials in total). Note that the initial values of $\forall i N_i$ and $\forall i S_i$ for IDB are the same as those for SBB. As a result, IDB obtained optimal solutions in 30 trials within the cycle at which SBB found the optimal solutions.

6.2 Anytime Curves

Next we compared IDB with SBB in terms of *anytime curves*. An anytime curve illustrates how the global distance (maximum number of constraint violations over agents) of the best solution found so far is improved as time proceeds. We show an anytime curve for each algorithm on the x-y plane with the x axis being the number of cycles passed by and the y axis being the global distance of the best solution found so far.

The thick line in Fig. 5 shows an anytime curve using SBB for an instance of a class of $n = m = 10, p_1 = 27/45, p_2 = 0.9$. For this instance, SBB finds an

problem class	nearly-optimal (optimal)	cycle for IDB	cycle for SBB
$\langle 10, 10, 18/45, 0.8\rangle$	2.2 (1)	100	417
$\langle 10, 10, 18/45, 0.9\rangle$	3.6 (2)	46	585
$\langle 10, 10, 27/45, 0.8\rangle$	3.6 (2)	508	2052
$\langle 10, 10, 27/45, 0.9\rangle$	4.2 (3)	196	3716
$\langle 10, 10, 36/45, 0.8\rangle$	4.6 (3)	3416	6360
$\langle 10, 10, 36/45, 0.9\rangle$	6.0 (4)	344	200145
$\langle 10, 10, 45/45, 0.8\rangle$	5.8 (4)	438	258753
$\langle 10, 10, 45/45, 0.9\rangle$	7.3 (6)	90	45696

Table 2. Cycles to find nearly-optimal solutions

optimal solution with the minimum cycles. The dotted line shows an anytime curve for IDB with the same instance. For IDB, the global distance at a certain cycle is averaged over the results of 10 trials with the same instance.

As shown in Fig. 5, while the curve of SBB eventually converges to the optimal distance, it declines relatively slowly. IDB, on the other hand, has a rapid drop at the beginning, and after that keeps steady at a *nearly optimal distance.* That is not peculiar to this instance but can be seen in other instances of this class or other classes. We conducted the same experiment with other classes and measured the number of cycles IDB consumes to reach a nearly optimal distance. We also measured the number of cycles SBB consumes to outperform the nearly optimal distance. Table 2 shows for each class the measured number of cycles for the nearly optimal distance with the real optimal distance in parentheses. We can see that IDB reaches the nearly optimal distance much sooner than does SBB for all classes.

7 Conclusions

We have presented the distributed partial constraint satisfaction problem as a new framework for dealing with over-constrained distributed CSPs. Since many problems in multi-agent systems can be described as distributed CSPs that are possibly over-constrained, we expect a DPCSP to have great potential in various applications.

We have also presented the synchronous branch and bound and the iterative distributed breakout for solving distributed maximal constraint satisfaction problems, which belong to a very important class of DPCSP. Our experimental results on random binary distributed CSPs show that SBB is preferable when we are concerned with the optimality of a solution, while IDB is preferable when we want to get a nearly optimal solution quickly. Our future work will include developing more efficient algorithms for DMCSPs and applying this framework to more realistic problems.

References

1. A. Borning, B. Freeman-Benson and M. Wilson. Constraint Hierarchies. In *Lisp and Symbolic Computation*, Vol. 5, pp. 223–270, 1992.
2. K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transaction on Computer Systems*, Vol. 3, No. 1, pp. 63–75, 1985.
3. S. E. Conry, K. Kuwabara, V. R. Lesser and R. A. Meyer. Multistage Negotiation for Distributed Constraint Satisfaction. *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No. 6, pp. 1462–1477, 1991.
4. E. C. Freuder. Partial Constraint Satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 278–283, 1989.
5. E. C. Freuder and R. J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, Vol. 58, No. 1–3, pp. 21–70, 1992.
6. K. Hirayama and J. Toyoda. Forming Coalitions for Breaking Deadlocks. In *Proceedings of First International Conference on Multi-Agent Systems*, pp. 155–162, 1995.
7. M. N. Huhns and D. M. Bridgeland. Multiagent Truth Maintenance. *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No. 6, pp. 1437–1445, 1991.
8. J. Larrosa and P. Meseguer. Phase Transition in MAX-CSP. In *Proceedings of the Twelfth European Conference on Artificial Intelligence*, pp. 190–194, 1996.
9. V. R. Lesser and D. D. Corkill. The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks. *AI Magazine*, Vol. 4, No. 3, pp. 15–33, 1983.
10. P. Morris. The Breakout Method for Escaping from Local Minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 40–45, 1993.
11. K. P. Sycara, S. Roth, N. Sadeh and M. Fox. Distributed Constrained Heuristic Search. *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No. 6, pp. 1446–1461, 1991.
12. R. J. Wallace and E. C. Freuder. Conjunctive Width Heuristics for Maximal Constraint Satisfaction. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 762–768, 1993.
13. M. Yokoo. Constraint Relaxation in Distributed Constraint Satisfaction Problem. In *5th International Conference on Tools with Artificial Intelligence*, pp. 56–63, 1993.
14. M. Yokoo, E. H. Durfee, T. Ishida and K. Kuwabara. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In *Proceedings of the Twelfth IEEE International Conference on Distributed Computing Systems*, pp. 614–621, 1992.
15. M. Yokoo and K. Hirayama. Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems. In *Proceedings of Second International Conference on Multi-Agent Systems*, pp. 401–408, 1996.

This article was processed using the LaTeX macro package with LLNCS style