12th IEEE International Conference on Distributed Computing Systems '92, pp.614–621

Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving

Makoto Yokoo [†]
Toru Ishida[†]
Edmund H. Durfee [‡]
Kazuhiro Kuwabara[†]

[†] NTT Communication Science Laboratories
Sanpeidani Inuidani, Seika-cho
Soraku-gun, Kyoto 619-02 Japan
yokoo/ishida/kuwabara@cslab.kecl.ntt.jp

[‡] Dept. of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109 U.S.A.
durfee@caen.engin.umich.edu

Abstract

Viewing cooperative distributed problem solving (CDPS) as distributed constraint satisfaction provides a useful formalism for characterizing CDPS techniques. In this paper, we describe this formalism and compare algorithms for solving distributed constraint satisfaction problems (DCSPs). In particular, we present our newly developed technique called asynchronous backtracking *that allows agents to act asynchronously and concurrently, in contrast to the traditional sequential backtracking techniques employed in constraint satisfaction problems. Our experimental results show that solving DCSPs in a distributed fashion is worthwhile when the problems solved by individual agents are loosely-coupled.*

1 Introduction

Cooperative distributed problem solving (CDPS) is a subfield of AI that is concerned with how a set of artificially intelligent agents can work together to solve problems. Recently, [9] has presented the idea of viewing CDPS as a distributed state space search in order to develop a general framework for CDPS. This concept is important because, without such general frameworks, it is very difficult to compare alternative approaches or to reproduce results obtained by one approach on slightly different problems. Our goal is to develop a framework for formalizing a subset of CDPS problems and methods by extending constraint satisfaction problems (CSPs) [10] to distributed multi-agent environments. In this paper, we define a *distributed constraint satisfaction problem* (DCSP) as a CSP in which multiple agents are involved. DCSPs are important for the following reasons:

- **Various CDPS problems can be formalized as DCSPs.**
  Multi-agent resource allocation problems described in [4], [8], in which tasks or resources must be allocated to agents so that inter-agent constraints are satisfied, can be formalized as DCSPs by viewing each task or resource as a variable and the possible assignments as values. Also, an interpretation problem by multiple agents such as [11] can be mapped into a DCSP framework by viewing possible interpretations as possible variable values. Distributed truth maintenance tasks described in [1] are essentially solving a DCSP where each variable can be either IN or OUT. By formalizing these problems as DCSPs, these problems can be solved by the general algorithms described in this paper.
- **DCSPs provide a formal framework for studying various CDPS methods.**
  There are various options in the methods for solving DCSPs, which influence the efficiency (e.g., the selection order of the values). Agents have to make decisions about these options and these decisions are interrelated. DCSPs serve as a basis for studies such as [6], in which agents exchange their local plans in order to make agents' decisions coherent.

In this paper, we define DCSPs and discuss alternative methods for solving them. In particular, we introduce our newly developed technique, called *asyn-*

chronous backtracking. Backtracking, which is a standard approach to solve CSPs, is essentially a sequential procedure. Our new algorithm allows agents to act concurrently and asynchronously. We experimentally show how our algorithm outperforms standard backtracking techniques. In addition, our experiments highlight issues of granularity in loosely-coupled distributed systems.

2 Distributed Constraint Satisfaction Problem (DCSP)

2.1 CSP

A CSP is formally defined as m variables $x_1, x_2, ..., x_m$, taking their values from a domains $D_1, D_2, ..., D_m$ respectively, and a set of constraints on their values. A constraint is defined by a predicate. That is, the constraint $P_k(x_{k1}, \ldots, x_{kj})$ is a predicate which is defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. This predicate is true iff the instantiations of these variables satisfy this constraint. Solving a CSP is equivalent to finding an assignment of values to all the variables such that all constraints are satisfied.

2.2 DCSP

In a DCSP, the variables of a CSP are distributed among agents. We assume the following communication model.

- Communication between agents is done by sending messages. An agent can send messages to other agents iff the agent knows the addresses of the agents[1].
- The delay in delivering a message is finite, though random. 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 instantiate their values. Constraints may exist between variables of different agents, and the instantiations of the variables must satisfy these inter-agent constraints. Formally, there exist n agents $1, 2, \ldots, n$. Each variable x_j belongs to one agent i (this relation is represented as $belongs(x_j, i)$). Constraints are also distributed among agents. The fact that the agent k knows the constraint predicate P_l is represented as $known(P_l, k)$.

We say that a DCSP is solved iff the following conditions are satisfied.

- $\forall i$, $\forall x_j$ belongs$(x_j, i) \Rightarrow x_j$ is instantiated to d_j, and $\forall\ k$, $\forall P_l$ known$(P_l, k) \Rightarrow P_l$ is true under the assignment $x_1 = d_1, x_2 = d_2, \ldots, x_m = d_m$.

Without loss of generality, we make the following assumptions while describing our algorithms for simplicity.

- Each agent has exactly one variable.
- Each agent knows all constraint predicates relevant to its variable.

3 Methods for DCSP

Methods for solving CSPs can be divided into two groups, namely backtracking algorithms and consistency algorithms [10]. Consistency algorithms are preprocessing procedures that are invoked before backtracking. Consistency algorithms in the ATMS framework [5] are essentially monotonic and can be applied straightforwardly to DCSP [13]. Therefore, in this paper, we focus on backtracking algorithms for DCSPs.

3.1 Centralized Backtracking

The most trivial algorithm for solving a DCSP is to select a leader agent among all agents, and gather all information about variables, their domains, and their constraints, into the leader agent. The leader then solves the CSP alone using standard backtracking algorithms. This approach is wasteful, both in terms of communication overhead (for selecting a leader and collecting the information at the leader) and in loss of parallelism (as the other agents sit idle while the leader solves the CSP).

3.2 Synchronous Backtracking

The standard backtracking algorithm for CSP can be simply modified to yield the synchronous backtracking algorithm for DCSP. Assume the agents agree on an instantiation order for their variables (such as agent 1 goes first, then agent 2, and so on). Each agent, receiving a partial solution (the instantiations of the preceding variables) from the previous agent, instantiates its variable based on the constraints that

[1]This model does not necessarily mean that the physical communication network must be fully connected (i.e., a complete graph). Unlike most parallel/distributed algorithm studies, in which the topology of the physical communication network plays an important role, we assume the existence of a reliable underlying communication structure among agents and do not care about the implementation of the physical communication network.

it knows about. If it finds such a value, it appends this to the partial solution and passes it on the next agent. If no instantiation of its variable can satisfy the constraints, then it sends a *backtracking* message to the previous agent.

While this algorithm does not suffer from the same communication overhead as the centralized method, it does suffer from inefficiencies by not taking advantage of parallelism. Because, at any given time, only one agent is receiving the partial solution and acting on it, the DCSP is still solved sequentially [2].

3.3 Asynchronous Backtracking

Our asynchronous backtracking algorithm removes the drawbacks of synchronous backtracking by allowing agents to run concurrently and asynchronously. Each agent instantiates its variable and communicates the variable value to relevant agents. To simplify this discussion, let us assume that constraints are binary.

We represent a DCSP in which all constraints are binary as a network, where variables are nodes and constraints are links between nodes[3]. Since each agent has exactly one variable, a node also represents an agent. We use the same identifier (id) for representing an agent and its variable. We also assume that every link (constraint) is *directed.* In other words, one of the two agents involved in a constraint is assigned that constraint, and receives the other agent's value. A link is directed from the value sending agent to the constraint evaluating agent. For example, in Figure 1 (a), there are three agents, x_1, x_2, x_3, with variable domains $\{1,2\}, \{2\}, \{1,2\}$ respectively, and the constraints $x_1 \neq x_3$ and $x_2 \neq x_3$.

Each agent instantiates its variable concurrently and sends the value to the agents which are connected by outgoing links. After that, agents wait for and respond to messages. Figure 2 describes procedures for receiving two kinds of messages. One kind is an *ok?* message, that a constraint evaluating agent receives from a value sending agent, asking whether the value chosen is acceptable (Figure 2 (i)). The second kind is a *nogood* message that a value sending agent receives, indicating that the constraint evaluating agent has found a constraint violation (Figure 2 (ii)).

[2]Recently, [3] presented a variation of synchronous backtracking called *Network Consistency Protocol*, in which agents construct a depth-first search tree. Agents act synchronously by passing *privilege*, but the agents which have the same parent in the search tree can act concurrently.

[3]It must be emphasized that this constraint network has nothing to do with the physical communication network. The link in the constraint network is not a physical communication link, but a logical relation between agents.

An agent has a set of values from the agents which is connected to by incoming links. These values constitute the agent's *agent_view*. The fact that x_1's value is 1 is represented by a pair of the agent id and the value, $(x_1, 1)$. Therefore, an agent_view is a set of these pairs, e.g., $\{(x_1, 1), (x_2, 2)\}$. If an *ok?* message is sent on an incoming link, the evaluating agent adds the pair to its agent_view and checks whether its own value assignment (represented as (*my_id, my_value*)) is *consistent* with its agent_view. Its own assignment is consistent with the agent_view if all constraints the agent evaluates are true under the value assignments described in the agent_view and (*my_id, my_value*), and all communicated *nogoods* are not *compatible* [4] with the agent_view and (*my_id, my_value*). If its own assignment is not consistent with the agent_view, the agent tries to change *my_value* so that it will be consistent with the agent_view.

A subset of an agent_view is called a *nogood* if the agent is not able to find *my_value* which is consistent with the subset. If an agent finds a subset of its agent_view is a nogood, the assignments of other agents must be changed. Therefore, the agent causes a *backtrack* (Figure 2 (iii)) and sends a *nogood* message to one of the other agents.

3.3.1 Avoiding Infinite Processing Loops

If agents change their values again and again and never reach a stable state, they are in an infinite processing loop. An infinite processing loop can occur if there exists a value changing loop of agents, such as if a change in x_1 causes x_2 to change, then this change in x_2 causes x_3 to change, which then causes x_1 to change, and so on. In the network representation, such a loop is represented by a cycle of directed links.

One way to avoid cycles in a network is to use a total order relationship among nodes. If each node has an unique id, and a link is directed from the smaller node to the larger node, there will be no cycle in the network. This means that each agent has an unique id, and for each constraint, the larger agent will be an evaluator, and the smaller agent will send an *ok?* message to the evaluator. Furthermore, if a nogood is found, a *nogood* message is sent to the largest agent in the nogood (Figure 2 (iii-a)). Similar techniques to this unique id method are used for avoiding deadlock in distributed database systems [12]. The knowledge each agent requires for this unique id method is much

[4]A nogood is compatible with the agent_view and (*my_id, my_value*) if all variables in the nogood have the same values in the agent_view and (*my_id, my_value*).

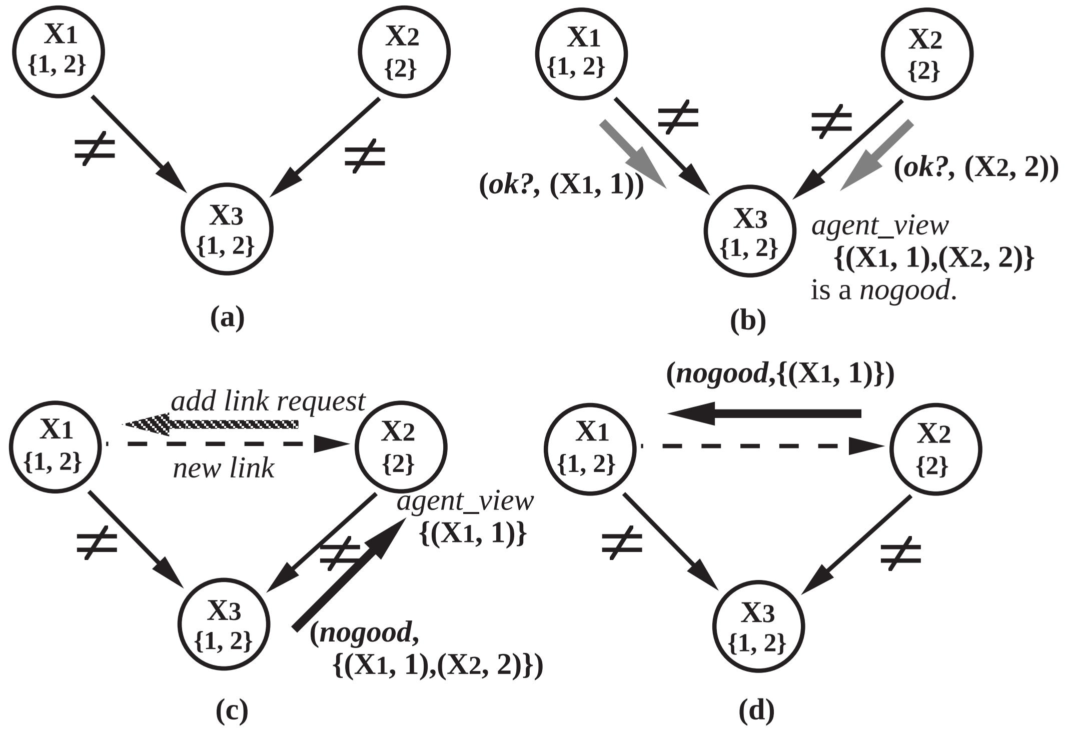

Figure 1: Example of constraint network

more local than that needed for synchronous backtracking. In synchronous backtracking, agents must act in a predefined sequential order. Such a sequential order can not be obtained easily just by giving an unique id to each agent.

3.3.2 Handling Asynchronous Changes

In the following, we show the methods for handling the difficulties which are caused by the asynchronous activities of agents.

Tolerating Inconsistent Agent_Views. Because agents change their instantiations asynchronously, an agent_view is subject to incessant changes. This can lead to potential inconsistencies, because a constraint evaluating agent might send a nogood message to an agent that has already changed the value of an offending variable as a result of other constraints. In essence, the nogood message may be based on obsolete information, and the value sending agent should not necessarily change its value again.

We introduce the use of *context attachment* to deal with these potential inconsistencies. In context attachment, an agent couples its message with the nogood that triggered it. This nogood is the context of backtracking. After receiving this message, the recipient only changes its value if the nogood is *compatible* with its current agent_view and its own assignment (Figure 2 (ii-a)). Since the nogood attached to a *nogood* message indicates the cause of the failure, asynchronous backtracking includes the function of dependency-directed backtracking in CSPs [10].

A nogood can be viewed as a new constraint derived from the original constraints. By incorporating such a new constraint, agents can avoid repeating the same failure again. For example, in Figure 1 (c), the nogood $\{(x_1, 1), (x_2, 2)\}$ represents a constraint between x_1 and x_2. Since there is no link between x_1 and x_2 originally, a new link must be added between them [5]. Therefore, after receiving the *nogood* message, agent x_2 asks x_1 to add a link between them. In general, even if all original constraints are binary, newly derived constraints can be among more than 2 variables. In such a case, one of the agents in the constraint will be an evaluator and links will be added between each of non-evaluator agents and the evaluator.

[5]Since a link in the constraint network represents a logical relation between agents, adding a link does not mean adding a new physical communication path between agents.

```
when received (ok?, (sender_id,value)) do — (i)
  add (sender_id,value) to agent_view;
  when my_value and agent_view are
    inconsistent do
      change my_value to a new consistent value; — (i–a)
      when can not find such a value do backtrack;
        change my_value to a new consistent value;
      end do;
      send (ok?, (my_id,my_value)) to its outgoing links;
  end do;
end do;

when received (nogood, sender_id, nogood) do — (ii)
  record nogood;
  when (id,value) where id is not connected
      is contained in nogood do
    request id to add a link from id to my_id
      and add (id,value) to agent_view;
  end do;
  if agent_view and my_value are incompatible
        with nogood — (ii–a)
    then send (ok?, (my_id,my_value)) to sender_id;
    else change my_value
        to a new consistent value; — (ii–b)
      when can not find such a value do backtrack;
        change my_value to a new consistent value;
      end do;
    send (ok?, (my_id,my_value)) to its outgoing links;
  end if;

procedure backtrack — (iii)
begin
  nogoods ← $\{V_s \mid V_s$=inconsistent subset of agent_view$\}$;
  when $\{\} \in$ nogoods do
    broadcast to other agents that there is
      no solution, terminate this algorithm;
  end do;
  for each $V_s = \{(id_1, v_1), \ldots\} \in$ nogoods do;
    select $(id_j, v_j)$ where
        $id_j$ is the largest in $V_s$; — (iii-a)
    send (nogood, my_id, $V_s$) to $id_j$;
    remove $(id_j, v_j)$ from agent_view;
  end do;
end backtrack;
```

Figure 2: Procedure for receiving messages

Interrupting Constraint Checking. In asynchronous backtracking, an agent_view may change while an agent is checking all applicable constraints to instantiate its variable. When this happens, continuing the local computation is useless since the value it chooses must be checked against the new agent_view anyway. Our asynchronous backtracking algorithm handles this problem by interrupting the consistency checks (performed at Figure 2 (i–a) and (ii–b)) whenever the agent_view changes. Furthermore, we have incorporated the technique of *backmarking* [7] to let an agent take advantage of as much of the interrupted computation's results as possible. Here is the basic idea of backmarking: Assume an agent does consistency checks between its variable x_i and the variables of other agents $x_1, \ldots, x_{i-1}$. If a consistency check for value d fails with the variable x_j, the agent marks the value d with x_j. Now, assume that the agent receives a message that changes the variable x_k, and value d is marked as x_j before the message is received. If $j < k$, the agent knows that assigning its variable with the value d will still lead to an inconsistency (since the value of x_j has not changed). Also, if $k < j$, then any checks the agent did with the value d and $x_1, \ldots, x_{k-1}$ are still satisfied and need not be rechecked. The agent thus avoids unnecessary constraint checking by having saved information from the interrupted computation.

3.3.3 Example

In Figure 1 (b), by receiving *ok?* messages from x_1 and x_2, the agent_view of x_3 will be $\{(x_1, 1), (x_2, 2)\}$. Since there is no possible value for x_3 consistent with this agent_view, this agent_view is a nogood. Agent x_3 chooses the largest agent in the agent_view, agent x_2, and sends a *nogood* message with the nogood, and removes $(x_2, 2)$ from the agent_view. By receiving this *nogood* message, agent x_2 records this nogood. This nogood, $\{(x_1, 1), (x_2, 2)\}$ contains agent x_1, which is not connected with x_2 by a link. Therefore, a new link must be added between x_1 and x_2. Agent x_2 requests x_1 to send x_1's value to x_2, and adds $(x_1, 1)$ to its agent_view (Figure 1 (c)). Agent x_2 checks whether its value is consistent with the agent_view. Since the nogood received from agent x_3 is compatible with its assignment $(x_2, 2)$ and its agent_view $\{(x_1, 1)\}$, the assignment $(x_2, 2)$ is inconsistent with the agent_view. The agent_view $\{(x_1, 1)\}$ is a nogood because x_2 has no other possible values. There is only one agent in this nogood, i.e., agent x_1, so agent x_2 sends a *nogood* message to agent x_1 (Figure 1 (d)).

3.3.4 Algorithm Soundness and Completeness

If there exists a solution, this algorithm reaches a stable state where all variable values satisfy all constraints, and all agents are waiting for an incoming message[6]. If no solution exists, this algorithm discovers this fact and terminates. For the agents to reach a stable state, all their variable values must perforce satisfy all constraints. Thus, the soundness of the algorithm is clear. Furthermore, the algorithm is complete, in that it finds a solution if one exists and terminates with failure when there is no solution.

A solution does not exist when the problem is overconstrained. In an overconstrained situation, our algorithm eventually generates a nogood corresponding to the empty set. Because a nogood logically represents a set of assignments which leads to a contradiction, an empty nogood means that any set of assignments leads to a contradiction. Thus, no solution is possible. Our algorithm thus terminates with failure if and only if an empty nogood is formed.

So far, we have shown that when the algorithm leads to a stable state the problem is solved, and when it generates an empty nogood, the algorithm terminates with failure. What remains is that we need to show that the algorithm must reach one of these conclusions in finite time. The only way that our algorithm might not reach a conclusion is at least one agent is cycling among its possible values in an infinite processing loop. Given our algorithm, we can prove by induction that this cannot happen as follows.

In the base case, assume that the agent with the lowest id, x_1, is in an infinite loop. Because it has the lowest id, x_1 only receives *nogood* messages. When it proposes a possible value, x_1 either receives a *nogood* message back, or else gets no message back. If it receives *nogood* messages for all possible values of its variable, then it will generate an empty nogood (any choice leads to a constraint violation) and the algorithm will terminate. If it does not receive a nogood message for a proposed value, then it will not change that value. Either way, it cannot be in an infinite loop.

Now, assume that agents x_1 to x_{k-1} ($k > 2$) are in a stable state, and agent x_k is in an infinite processing loop. In this case, the only messages agent x_k receives are *nogood* messages from agents whose ids are larger than k, and these *nogood* messages contain only the agent ids x_1 to x_k. Since agents x_1 to x_{k-1} are in a stable state, the *nogoods* agent x_k receives must be compatible with its agent_view, and so x_k will change instantiation of its variable with a different value. Because its variable's domain is finite, x_k will either eventually generate a value that does not cause it to receive a *nogood* (which contradicts the assumption that x_k is in an infinite loop), or else it exhausts the possible values and sends a *nogood* to one of $x_1 \ldots x_{k-1}$. However, this *nogood* would cause an agent we assumed as being in a stable state to not be in a stable state. Thus, by contradiction, x_k cannot be in an infinite processing loop.

[6]We should mention that the way to determine that agents as a whole have reached a stable state is not contained in this algorithm. To detect the stable state, distributed termination detection algorithms such as [2] are needed.

4 Evaluation

In this section, we compare the efficiency of the asynchronous, synchronous, and centralized backtracking algorithms. We simulate concurrent activity among the agents using 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 a constraint check. Because the asynchronous algorithm permits constraint checks in parallel, we expected this algorithm to outperform the centralized approach. However, to make the comparison more fair, we have to recognize that the multiagent algorithms depend on communication between agents. For this reason we introduced a communication delay t_d, such that a message issued at time t is available to the recipient at time $t + t_d$.

Given this model, we applied the algorithms to a well-studied constraint satisfaction problem, the n-queens problem. Each agent is assigned a queen to position in its column under the constraints that its queen not be threatened by the queens of other agents. By associating one queen per agent for 8- and 12-queens problems, we thus experimented with networks of 8 and 12 agents, respectively.

We analyzed performance in terms of the amount of simulated time required to solve the problems, and varied the communication delay. Our results are summarized in the graph shown in Figure 3. To make the comparisons fair, we included dependency-directed backtracking and backmarking in the synchronous and centralized systems as well as in our asynchronous mechanisms. When comparing asynchronous with synchronous backtracking, the additional parallelism of asynchronous backtracking makes our new algorithms 1.5 to 2 times as fast as synchronous backtracking. As communication delay increases, the time needs of both algorithms increase linearly.

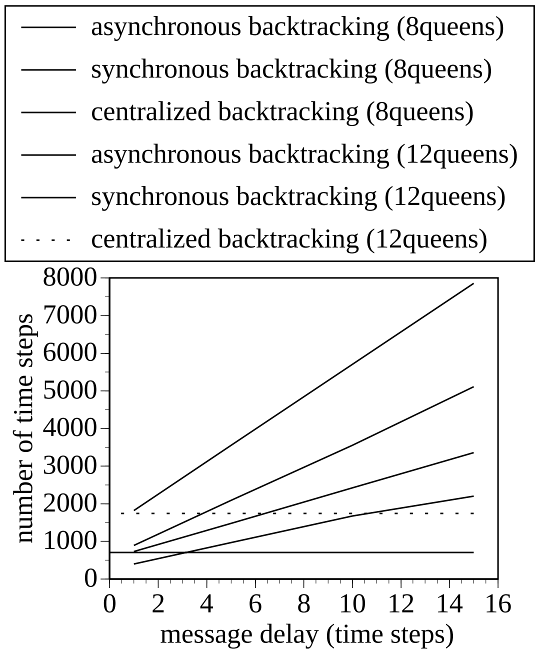

Figure 3: Comparison between asynchronous, synchronous and centralized backtracking (8, 12 queens problems)

An initially surprising result, however, was that our asynchronous algorithm only outperformed a centralized approach when message delays are very small. Given that the constraints being checked require very little computation, a message delay equivalent to 5 or more constraint checks would not be unreasonable to expect in a loosely-coupled distributed network. Our results indicated that our algorithm would be suitable for a tightly-coupled multiprocessor, but not a loosely-coupled system.

However, on reflection we realized that this result was not due to our algorithm, but instead to our problem decomposition. In essence, we were giving our agents tasks that were too small. Traditional cooperative distributed problem solving applications involve having agents work on large, nearly-independent subproblems. As a result, agents spend considerable time for local computation between sending messages. By having agents work on large-grained problems, the communication delays in a loosely-coupled network no longer dominate performance. For example, a typical problem involves having agents track vehicles moving through their different areas[6]. These agents each perform substantial processing independently as they interpret their own data, but they constrain each other at their borders since their pieces of tracks must fit together.

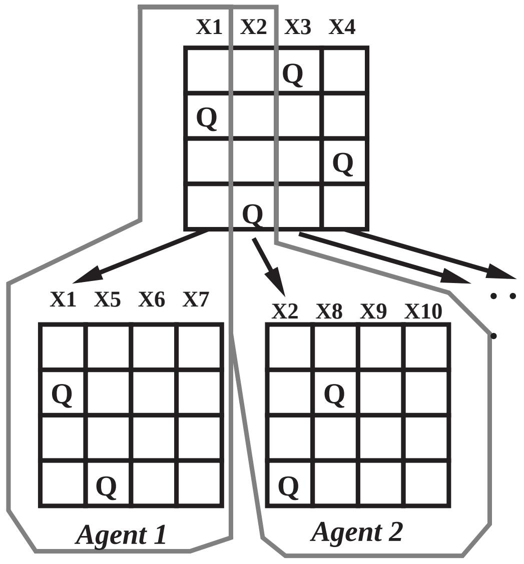

Figure 4: Example of loosely-coupled DCSP (hierarchical n-queens)

We have modified the n-queens problem to capture this "large-grained, nearly-independent" character by developing the hierarchical n-queens problem shown in Figure 4. As the figure shows, the problem involves $n + 1$ n-queens problems, in which the n of the problems are essentially independent, except that the positioning of the queen in the first column carries over the $n + 1$th problem. If each agent is given one of these nearly independent problems, then an agent can spend considerable time solving its own problem and needs to interact less frequently with other agents to check whether its local solution is compatible with the agents' shared problem. This is like having agents build major subassemblies on their own, but these subassemblies must "hook together" in a compatible way.

Figure 5 compares 3 experiments using the hierarchical 8-queens problem. One experiment uses centralized backtracking. The second experiment uses asynchronous backtracking where, as before, each queen has its own agent. This means that, in this experiment, 64 agents are in action. The third experiment uses asynchronous backtracking involving only 8 agents, where each is responsible for 8 queens, including one of the queens of the "shared" board. When message delay is small, the greater parallelism afforded by using 64 agents dominates. However, as delay increases, performance of the 64 agents experiment degrades rapidly due to the amount of communication

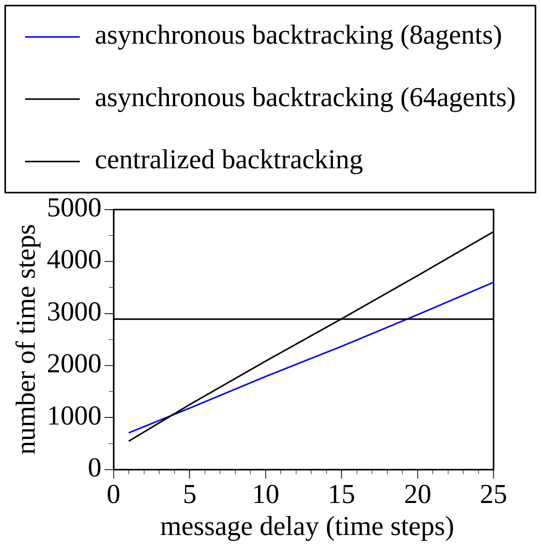

Figure 5: Comparison between asynchronous and centralized backtracking (loosely-coupled problem)

that the agents require. While using 8 agents was less effective with short message delays, increases in message delays impact the 8 agents experiment much less severely because the agents are working on larger individual problems and communicating less. As a result, using 8 agents outperforms the centralized case for delays up to 20 time units. These results confirm that, if the local problems are loosely-coupled, asynchronous backtracking outperforms centralized backtracking even if communications are relatively slow.

5 Conclusions

We presented the formalization of the distributed constraint satisfaction problem and described alternative methods for solving it, including our own asynchronous backtracking algorithm. Our experiments with these methods illustrated that solving constraint satisfaction problems in a loosely-coupled, distributed network is worthwhile if the local constraint satisfaction problems of each agent are large grained and nearly independent. These results formalize intuitions gained from past work in cooperative distributed problem solving.

Our ongoing research efforts involve introducing various heuristics proposed in CSP into asynchronous backtracking, and formalizing various CDPS methods (methods for achieving coherent behaviors, etc.) using the DCSP framework.

Acknowledgements
The authors wish to thank Drs. Kiyoshi Nishikawa and Ryohei Nakano for their support of this work.

References

[1] Bridgeland, D.M. and Huhns, M.N.: Distributed Truth Maintenance, *Proc. of AAAI-90*, pp. 72–77, 1990.

[2] Chandy, K.M. and Lamport, L : Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. on Computer Systems*, Vol. 3, No. 1, pp. 63–75, 1985

[3] Collin, Z., Dechter, R. and Kaiz, S. : On the Feasibility of Distributed Constraint Satisfaction, *Proc. of IJCAI-91*, pp. 318–324, 1991.

[4] Conry, S.E., Meyer, R.E. and Lesser, V.R. : Multistage Negotiation in Distributed Planning, In Alan H.Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann, pp. 367–384,1988.

[5] de Kleer, J : A Comparison of ATMS and CSP Techniques, *Proc. of IJCAI-89,* pp. 290–296, 1989.

[6] Durfee, E.H. and Lesser, V.R. : Using Partial Global Plans to Coordinate Distributed Problem Solvers, *Proc. of IJCAI-87*, pp. 875–883, 1987.

[7] Gasching, J. : A General Backtrack Algorithm That Eliminates Most Redundant Tests, *Proc. of IJCAI-77*, pp. 457, 1977.

[8] Kuwabara, K. and Lesser, V.R. : Extended Protocol for Multistage Negotiation, In M. Benda editor, *Proc. of 9th Workshop on Distributed Artificial Intelligence*, pp. 129–161, 1989.

[9] Lesser, V.R. : An Overview of DAI: Viewing Distributed AI as Distributed Search, *Journal of Japanese Society for Artificial Intelligence*, Vol. 5, No. 4, 1990.

[10] Mackworth, A.K. : Constraint Satisfaction, In S.C.Shapiro, editor, *Encyclopedia of Artificial Intelligence*, John Wiley & Sons, pp. 205–211, 1987.

[11] Mason, C.L. and Johnson, R.R. : DATMS: A Framework for Distributed Assumption Based Reasoning, In Les Gasser and M.N. Huhns, editors, *Distributed Artificial Intelligence Vol.II*, pp. 293–318, Morgan Kaufmann, 1989.

[12] Rosenkrantz, D.J., Stearns, R.E., and Lewis, P.M. : System Level Concurrency Control for Distributed Database Systems, *ACM Trans. on Database Systems*, Vol. 3, No. 2, pp. 178–198, 1978.

[13] Yokoo, M., Ishida, T. and Kuwabara, K. : Distributed Constraint Satisfaction for DAI Problems, In M. Huhns editor, *Proc. of 10th Workshop on Distributed Artificial Intelligence*, 1990.