Dynamic Agent-Ordering and Nogood-Repairing in Distributed Constraint Satisfaction Problems

Lingzhong Zhou, John Thornton and **Abdul Sattar**
School of Information Technology,
Griffith University, Queensland, Australia
{l.zhou, j.thornton, a.sattar}@griffith.edu.au

Abstract

The distributed constraint satisfaction problem (CSP) is a general formalization used to represent problems in distributed multi-agent systems. To deal with realistic problems, multiple local variables may be required within each autonomous agent. A number of heuristics have been developed for solving such multiple local variable problems. However, these approaches do not always guarantee agent independence and have low efficiency search mechanisms.

In this paper, we are interested in increasing search efficiency for distributed CSPs. To this end we present a new algorithm using unsatisfied constraint densities to dynamically determine agent ordering during the search. Variables having a total order relationship only exist in the local agent. The independence of agents is guaranteed and agents without neighboring relationships can run concurrently and asynchronously. As a result of using nogoods to guarantee completeness, we developed a new technique called nogood repairing, which greatly reduces the number of nogoods stored and communication costs during the search, leading to further efficiency gains. In an empirical study, we show our new approach outperforms an equivalent static ordering algorithm and a current state-of-the-art technique in terms of execution time, memory usage and communication cost.

Introduction

The constraint satisfaction paradigm is a well recognized and challenging field of research in artificial intelligence, with many practical and important applications. A constraint satisfaction problem (CSP) is a problem with a finite number of variables, each of which has a finite and discrete set of possible values, and a set of constraints over the variables. A solution of a CSP is an instantiation of all variables for which all the constraints are satisfied.

When the variables and constraints of a CSP are distributed among a set of autonomous and communicating agents, this can be formulated as a distributed constraint satisfaction problem (distributed CSP), where agents autonomously and collaboratively work together to get a solution. A number of heuristics have been developed for solving distributed CSPs, such as synchronous backtracking, asynchronous backtracking (ABT) (Yokoo *et al.* 1998), asynchronous weak-commitment search (AWC) (Yokoo *et al.* 1998) and the distributed breakout algorithm (DB) (Yokoo & Hirayama 1996). However, these algorithms can only handle one variable per agent. In (Armstrong & Durfee 1997), dynamic prioritization is used to allow agents with multiple local variables in distributed CSPs. Here, each agent tries to find a local solution, consistent with the local solutions of higher priority agents. If no local solution exists, backtracking or modification of the prioritization occurs. The approach uses a centralized controller, where one agent controls the starting and ending of the algorithm, and a nogood processor which records all nogood information. However, these centralized mechanisms are often not appropriate for realistic distributed CSPs. In (Yokoo & Hirayama 1998), AWC search was extended to deal with multiple local variables in distributed CSPs. Although nogood learning is used (Hirayama & Yokoo 2000), their approach still requires a large space to store nogoods during the search.

In this paper, we integrate two new techniques: *Dynamic Agent Ordering* and *Nogood Repairing* to form a new algorithm (DAONR). Dynamic agent ordering uses constraint density measures to locally compute a *degree of unsatisfaction* for each agent. These values are used to dynamically set the order in which agents are allowed to change their particular variable instantiations. In effect, the agents' orders are decided naturally by their unsatisfied constraint densities during the search. By using a nogood repairing approach, all nogood repairs are tried inside a local agent without sending nogood messages to neighboring agents. As each local computation is independent from other agents, the benefits of parallelism are retained, resulting in an approach that is suitable for agent oriented design and efficient in terms of memory and communication cost.

In the rest of the paper, we formalize the definition of a distributed CSP. Then, we describe the new algorithm and investigate its performance in an empirical study. Finally, we discuss the possibility of using the new algorithm to solve other variants of distributed CSPs.

Distributed Constraint Satisfaction Problems

Formalization

In a distributed constraint satisfaction problem:

1. There exists an agent set A:

$$A = \{A_1, A_2, ..., A_n\},\ n \in Z^+;$$

2. Each agent has a variable set X_i and domain set D_i,

$$X_i = \{X_{i1}, X_{i2}, ..., X_{ip_i}\};$$
$$D_i = \{D_{i1}, D_{i2}, ..., D_{ip_i}\}, \quad \forall i \in [1, n],\ p_i \in Z^+;$$

3. There are two kinds of constraints over the variables among agents:
   (a) Intra-agent constraints, which are between variables of same agent.
   (b) Inter-agent constraints, which are between variables of different agents.

   Agent A_i knows all constraints related to its variables. A variable may involve both intra-agent and inter-agent constraints.

4. A solution S, is an instantiation for all variables that satisfies all intra-agent and inter-agent constraints.

Since agents are distributed in different locations or in different processes, each agent only knows the partial problem associated with those constraints in which it has variables. A global solution then consists of a complete set of the overlapping partial solutions for each agent. Communication among agents is necessary and important in distributed CSPs, since each agent only knows its variables, variable domains and related intra-agent and inter-agent constraints. Hence, to evaluate a search algorithm, we not only need to measure the search speed but also to consider the communication cost.

The Dynamic Agent Ordering Algorithm

Motivation

In a CSP, the order in which values and variables are processed significantly affects the running time of an algorithm. Generally, we instantiate variables that maximally constrain the rest of the search space. For instance, selecting the variable with the least number of values in its domain tends to minimize the size of the search tree. When ordering values, we try to instantiate a value that maximizes the number of options available for future instantiations.

The efficiency of algorithms for distributed CSPs is similarly affected by the order of value and variable selection. In the case where agents control multiple variables, the order in which agents are allowed to instantiate shared variables also becomes important. Agent communication and external computation (instantiating variables to be consistent with inter-agent constraints) is more costly than local computation (instantiating variables to be consistent with intra-agent constraints), and wrong or redundant computation can occur as a result of inappropriate agent ordering. It is therefore worth investigating agent orderings in order to develop more efficient algorithms.

The task of ordering agents is more complex than ordering variables, as more factors are involved, i.e. not only constraints and domains but also the structure of neighboring agents. Deciding on agent ordering is analogous to granting a priority to each agent, where the priority order represents a hierarchy of agent authority. When the priority order is static, the order of agents is determined before starting the search process, and the efficiency of the algorithm is highly dependent on the selection of variable values. If the priority order is dynamic, this can be used to control decision making for each agent and the algorithm is more able to flexibly exploit to the current search conditions.

We propose a new algorithm which uses constraint density (related to both intra-agent and inter-agent constraints) to order agents in a distributed CSP. When a search becomes stuck (i.e. an inconsistency is found), a nogood is generated, and the agent starts nogood repairing until a local solution found. The algorithm calculates the unsatisfied constraint densities and the degree of unsatisfaction for each agent, then broadcasts to the neighboring agents. Since the agent uses the local information and globally available information for local computation, it can still run asynchronously and concurrently.

Agent Ordering

To develop a dynamic agent ordering algorithm requires the specification of those features of the search space that should determine the ordering. In this study we develop a measure of the *degree of unsatisfaction* for each agent, such that the agent with the highest degree of unsatisfaction has the highest priority. In a standard CSP, the degree of unsatisfaction can simply be measured as the number of constraints unsatisfied divided by the total number of constraints. However, in a distributed CSP, we have the additional consideration of the relative importance of intra- versus inter-agent constraints. As inter-agent constraints affect variables in more than one agent, and these variables in turn can affect the intra-agent problems, we decided to develop separate measures for the intra- and inter-agent problems, such that the inter-agent constraints are given greater importance. To do this we looked at two problem features: (i) the degree of interconnectedness between constraints (or constraint density) and (ii) the degree of interconnectedness between inter-agent constraints and the intra-agent local problem.

To measure constraint density, we firstly divided the problem for a particular agent into an intra-agent constraint problem and an inter-agent constraint problem:

Intra-Agent Constraint Density: For the intra-agent problem, the maximum constraint density is simply defined as the ratio of the number of constraints over the number of variables, i.e. for agent i:

$$intraDensity_i = \frac{|intraC_i|}{|intraV_i|}$$

where $intraC_i$ is the set of intra-agent constraints for agent i and $intraV_i$ is the set of variables constrained by $intraC_i$. Assuming that each constraint has the same *tightness*[1], then we would expect a larger density to indicate a more constrained and hence more difficult problem.

[1] i.e. the ratio of the number unsatisfying assignments over the total number of possible assignments.

Inter-Agent Constraint Density: The constraint density measure for the inter-agent problem contains two additional features which increase the relative importance of the inter-agent measure in comparison to the intra-agent measure. Firstly, for agent i, instead of dividing by the total number of variables constrained by i's inter-agent constraints $interC_i$, we divide only by the number of variables that are constrained by $interC_i$ *and* controlled by i, i.e. $|interV_i|$.

In addition, when counting agent i's jth inter-agent constraint, $c_{i,j}$, we also count the number of *intra*-agent constraints $m_{i,j}$ that share a variable with $c_{i,j}$. This means the more interconnected $c_{i,j}$ is with the intra-agent problem, the larger the value of $m_{i,j}$ and the greater the effect of $c_{i,j}$ on the overall inter-agent constraint density, given by:

$$interDensity_i = \frac{|interC_i| + \sum_{j=1}^{|interC_i|} m_{i,j}}{|interV_i|}$$

The sum $staticDensity_i = intraDensity_i + interDensity_i$ now provides a general measure of the overall density of the problem for a particular agent. The greater this measure, the more constrained or difficult we would consider the problem to be.

Dynamic Constraint Density: The dynamic constraint density for a particular agent is based on the static density measure, except that only unsatisfied constraints are counted in the numerator. In this way the density of a current level of constraint violation during a search can be measured. Using the functions $intraUnsat(i, j)$, which returns one if the jth intra-agent constraint for agent i is unsatisfied, zero otherwise, and $interUnsat(i, j)$, which returns one if the jth inter-agent constraint for agent i is unsatisfied, zero otherwise, we define the following measures:

$$intraUnsat_i = \frac{\sum_{j=1}^{|intraC_i|} intraUnsat(i,j)}{|intraV_i|}$$

and

$$interUnsat_i = \frac{\sum_{j=1}^{|interC_i|} (interUnsat(i,j) \times (m_{i,j}+1))}{|interV_i|}$$

These measures then range from a value of zero, if all constraints are satisfied, to $intraUnsat_i = intraDensity_i$ and $interUnsat_i = interDensity_i$ if all constraints are unsatisfied. Combining these measures, we define:

$$dynamicDensity_i = intraUnsat_i + interUnsat_i$$

and

$$degreeUnsat_i = \frac{dynamicDensity_i}{staticDensity_i}$$

$degreeUnsat_i$ now ranges from a value of zero, if all constraints for agent i are satisfied, to one, if all constraints are unsatisfied, while embodying the increased importance of inter-agent constraints in the overall evaluation. It is this measure we use to dynamically decide agent priority in our proposed algorithm (for further details see (Zhou, Thornton, & Sattar 2003)).

Nogood Repairing

Constraint solvers often fall in a situation where there is no consistent assignment of variables at hand. This situation can arise because the values assigned to one or more variables conflict with each other, i.e., violates one or more constraints. A set of conflicting values is referred to as a *nogood*. Obviously, a nogood cannot be a subset of a solution.

While nogoods are not part of the solution(s), they provide useful information to the constraint solvers. They can be recorded, and referred to whenever necessary to avoid the same assignment of variables. This can result in significant computational advantages, especially in Distributed Constraint Satisfaction Problems. Whenever an agent detects a nogood, it is sent to all relevant (neighboring) agents so they can avoid repeating the same inconsistent assignment. However, this may affect storage for each agent and the communication load among agents. We propose that it is not necessary to send all nogoods to other agents as it may be possible to fix the nogood within the local agent. This can therefore reduce the communication load and save memory usage for each agent.

In this study, we classify nogoods into three categories:

1. *Intra-nogood*: This nogood is purely constructed by local variable instantiations;
2. *Mix-nogood*: The category of nogood is constructed by both local and neighboring agents' variable instantiations;
3. *Inter-nogood*: This nogood is purely constructed by the variable instantiations between neighboring agents.

We use different methods to deal with these nogoods. If a nogood is an intra-nogood, the local agent does not need to send any outgoing messages. It will reassign the related local variables to repair the nogood. If a nogood is a mixed nogood, the agent tries to reinstantiate related local variables. If the nogood cannot be repaired, a nogood message is sent to the related neighboring agents. If a nogood is an inter-nogood, it has to be sent to related neighboring agents immediately.

By using nogood repairing, local variables have to be assigned priorities, one of which represents an authority over each variable inside an agent. The initial priority value of each variable is 0. If the values of two variables are the same, the priority is decided by the numeric order of the variable indexes. This total order relationship only exists among local variables. There is no total order between variables in different agents. As a result, the independence of each agent can be guaranteed.

We illustrate the nogood repairing approach in Figure 1:

1. In Figure 1 (a), Agent 1's variable 4 cannot be instantiated with any color, as each color in its domain is inconsistent with the colors instantiated for variables 1, 2 and 3 respectively. This can be expressed as the nogood $\{A_1, v_4, \{A_1, v_1, \text{`}Y\text{'}\}, \{A_1, v_2, \text{`}R\text{'}\}, \{A_1, v_3, \text{`}B\text{'}\}\}$, where Agent 1's ($A_1$) variable v_4 cannot be instantiated with any color because A_1's variable v_1 is instantiated with 'Y', A_1's variable v_2 is instantiated with 'R' and A_1's variable v_3 is instantiated with 'B'. As this nogood only involves intra-agent constraints it is an intra-nogood.

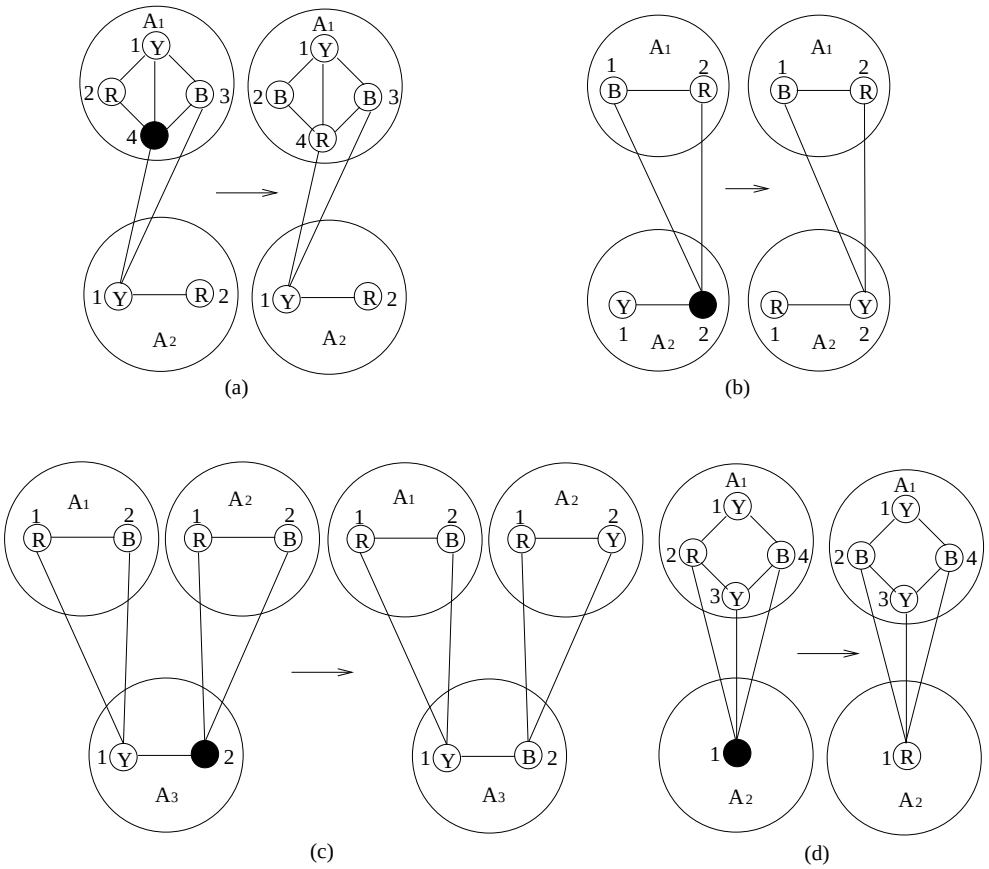

Figure 1: A Distributed 3-coloring Problem and Nogood Repairing

To effect a repair, the priority of v_4 will be increased to the highest priority in the nogood plus one. For example, assuming all variables' priorities are 0 in A_1, after this nogood is found, v_4's priority would be increased to 1. A_1 then randomly instantiates v_4 (e.g. with 'R'). In this case v_1's value will not be changed, since it is now consistent with v_4. However, v_2's value is inconsistent with v_4 and has to be changed (e.g to 'B'). Although v_3 has the lowest priority (by index order), its original value is still consistent with all other variables in the agent and so it is not changed. As result, this nogood is repaired internally and all constraints are satisfied.

2. In Figure 1 (b), Agent 2 discovers the mix-nogood $\{A_2, v_2, \{A_1, v_1, `B'\}, \{A_1, v_2, `R'\}, \{A_2, v_1, `Y'\}\}$. In previous algorithms (Yokoo & Hirayama 1998), A_2 will send the nogood to A_1, which will record it locally and so avoid repeating the instantiation. In our new algorithm, this nogood can be repaired inside A_2 without sending a nogood message. For example, assuming all A_2's variables have priority 0, A_2 will increase v_2's priority to 1, and change v_2's value (e.g. to 'Y'), to be consistent with A_1. Then A_2 can reassign v_1 (e.g. to 'R'), and the nogood is fixed. In this solution, only local computation occurs, no extra message is sent and no extra memory is needed to store the nogood.

   The case can still arise that after all local computation A_2 cannot find a consistent instantiation with A_1. When this occurs A_2 will send a nogood message to A_1. As A_1 was processed before A_2 it follows that A_1 has the higher priority. In order to repair the situation A_2's priority must be increased. This is achieved by setting A_2's $degreeUnsat$ to A_1's $degreeUnsat - \Delta$ [2] so that A_2's $degreeUnsat$ is slightly less than A_1's.

3. Figure 1 (c), shows a more detailed example of a mix-nogood that cannot be repaired by local computation. In this case A_3 cannot reassign any consistent color for v_2, and so it must increase its priority by decreasing its $degreeUnsat$. A_3 then sends a nogood message to A_2, and reassigns its variable v_2 to 'B'. A_2 then reassigns its variable v_2 to 'Y', repairing the nogood and finding a solution. This procedure can also be applied to the inter-nogood, as an inter-nogood can only be repaired by related neighboring agents. Figure 1(d) clearly shows this scenario.

Although the problem in Figure 1(b) only contains binary constraints, the nogoods generated during the search are non-binary. Dealing with these nogoods locally can re-

[2] Where Δ is the predefined precision of $degreeUnsat$, which in this case is 10^{-6}.

duce not only communication costs and memory storage but also computation costs. For example, in the absence of nogood repairing, when a local agent discovers a nogood relating to neighboring agents, each neighboring agent has to do the same nogood checking to avoid the inconsistent instantiation. This repeating of work can be avoided if the nogood can be repaired locally using our nogood repairing approach. In addition, dealing with a non-binary constraint is more complicated than dealing with a binary constraint. One local computation (using nogood repairing) instead of multiple local computations (without using nogood repairing) can greatly affect the search speed. This situation often happens in distributed CSPs.

Algorithm Implementation

The Dynamic Agent Ordering and Nogood Repairing (DAONR) algorithm was implemented as follows:

1. In the initial state, each agent concurrently instantiates their variables to construct a local solution, while checking consistency to guarantee that all intra-agent constraints are satisfied. Each agent then sends its local solution to its neighboring agents (i.e. those with which it shares at least one inter-agent constraint);
2. Each agent then starts to construct a local solution which attempts to satisfy both intra- and inter-agent constraints. In detail, each agent only considers inter-agent constraints with agents having higher priorities (lower $degreeUnsat$). Assuming the overall problem is satisfiable, if an agent is unable to instantiate a variable with any values in its domain, a nogood is discovered. Based on our nogood repairing approach (described in the previous section), an agent autonomously chooses the best option to repair the nogood, i.e. the agent will only send a nogood message if it cannot repair the nogood locally;
3. After assigning its own variables, an agent sends a message to neighboring agents. This message contains the $degreeUnsat$ value and the local instantiation of the agent.
4. The search will stop when each agent detects that its and all other agents' $degreeUnsat$ values are equal to zero.

The DAONR algorithm is shown in more detail in Algorithm *Dynamic Agent Ordering and Nogood Repairing*. Note that all variables from a neighboring agent, *iff* $degreeUnsat < local\_degreeUnsat$, have a higher priority than any local variables.

Algorithm *Dynamic Agent Ordering and Nogood Repairing*

1. **while** received(*Sender_id*, *variable_values*, *degreeUnsat*) do
2. calculate *local_degreeUnsat*;
3. **if** *local_degreeUnsat* and all other agents' *degreeUnsats* = 0
4. **then** the search is terminated;
5. **else** add (*Sender_id*, *variable_value*, *degreeUnsat*) to *agent_view*;
6. **if** *local_degreeUnsat* > *degreeUnsat*
7. **then** Assign_Local_Variables;

Algorithm *Assign_Local_Variables*

1. **if** local instantiation is consistent with *agent_view* from neighboring agents, and *degreeUnsat* < *local_degreeUnsat*
2. **then** send(*Sender_id*, *variable_values*, *local_degreeUnsat*) to neighboring agents;
3. **else** select an inconsistent variable v with the highest priority and assign a value from its domain;
4. **if** no value for this variable
5. **then if** nogood is new
6. **then** $Nogood\_Repairing(v)$;
7. **else** assign a value with minimal violations to the variables with lower priorities;
8. $Assign\_Local\_Variables$;

Algorithm *Nogood_Repairing (v)*

1. **if** nogood is intra-nogood or mix-nogood
2. **then** v's priority = the highest priority of local variables in nogood + 1;
3. **if** no values for v
4. **then** priority of the local agent = minimum of $degreeUnsat$ of related neighboring agents - Δ;
5. **else** assign a value with minimal violations to the variables with lower priorities;
6. **else** priority of the local agent = minimum of $degreeUnsat$ of related neighboring agents - Δ;

Experimental Evaluation

We evaluated our algorithm (DAONR) on a benchmark set of 3-coloring problems and against Asynchronous Weak-commitment search (AWC), recognized as the state-of-the-art for distributed CSPs where each agent has control over multiple variables (Yokoo & Hirayama 1998; Hirayama & Yokoo 2000). We implemented the latest version of AWC using nogood learning and obtained comparable results to those reported in (Hirayama & Yokoo 2000). All our results are averaged over 100 trails.

To simulate an autonomous agent environment we used an agent oriented design, implementing threads in FreeBSD that allow agents to run asynchronously and concurrently. All experiments were run on a Dell OptiPlex GX240 with a 1.6GHz P4 CPU and 256MB of PC133 DRAM. We used the same 3-coloring problem generator described in (Minton *et al.* 1992) and improved in (Yokoo & Hirayama 1998) to evaluate the performance of our algorithms. We chose this domain as the 3-coloring problem has been used in many other studies, and this type of problem is often used in connection with scheduling and resource allocation problems. To build the problem set, we randomly generated n agents with $(n \times 5)$ variables per problem. All instances were taken from the hard region of 3-coloring with a constraint to variable ratio of 2.7, assigning 50% of constraints as inter-agent and 50% as intra-agent constraints (within each problem). Each agent was also constrained to have at least one inter-agent constraint.

Figure 2 shows the average *degreeUnsat* for each of the

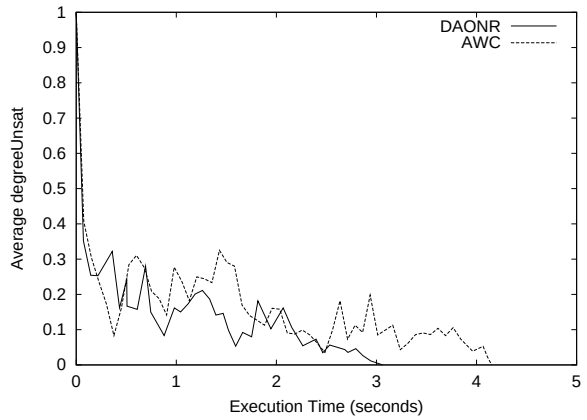

Figure 2: The average *degreeUnsat* plotted against time

Agents	Method	Checks	NG	LI	Time *(s)*
3	DAONR	88	2	5	0.025635
	AWC	128	2	4	0.040312
4	DAONR	107	3	11	0.033438
	AWC	151	14	9	0.073125
5	DAONR	1487	17	24	0.075625
	AWC	2710	43	31	0.132812
6	DAONR	4390	46	38	0.091562
	AWC	5340	72	69	0.170625
7	DAONR	9980	84	77	0.378125
	AWC	13410	203	119	0.714062
8	DAONR	14281	338	187	0.957425
	AWC	18344	827	254	1.376250
9	DAONR	18620	1063	403	1.113546
	AWC	21706	1814	436	2.005250
10	DAONR	25122	1750	753	3.062500
	AWC	29295	2533	1023	4.150650

Table 1: Results for distributed 3-coloring problems with n agents and n$\times 5$ Variables

two algorithms over the 10 agents with 50 variable problem set, and Table 1 shows the number of checks, the number of nogoods (NG) produced, the number of local instantiations (LI) broadcasted and the running time for all agents over the complete problem set. From these results it is clear that the new algorithm is considerably more efficient than AWC in terms of the number of messages sent and execution time.

Conclusion and Future Work

We have demonstrated a new algorithm that uses constraint density to dynamically order agents and a nogood repairing technique to increase the search speed, reduce communication cost and save memory usage in distributed CSPs. We argue that our algorithm is more feasible and offers greater agent independence than the existing algorithms for distributed CSPs, especially for situations with multiple local variables in each agent.

Our algorithm can be used to solve dynamic distributed CSPs and distributed over-constrained CSPs. Dynamic distributed CSPs are common in realistic problems, where constraints may be lost or added over time. By using our algorithm, real-time calculations can build new relations among agents, and changes of constraints and/or variables in one agent will not affect other agents' local computations. When a distributed CSP has no solutions, it is over-constrained. To deal with this kind of problem, we can setup a gate value (between 0 and 1) for the $degreeUnsat$ values. After all $degreeUnsat$ values reach the gate value, the problem is solved.

Finally, for problems where individual constraints have varying degrees of tightness, we can amend our constraint density measures to consider tightness directly. Currently we *count* the number of intra- and inter-agent constraints for each agent when calculating density. Alternatively, we can sum the tightness of these constraints, where tightness is defined as the number of possible unsatisfying assignments for a constraint divided by the total number of possible assignments.

References

Armstrong, A., and Durfee, E. 1997. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *The Fifteenth International Joint Conference on Artificial Intelligence*, 620–625.

Hirayama, K., and Yokoo, M. 2000. The effect of nogood learning in distributed constraint satisfaction. *The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*.

Minton, S.; Johnston, M. D.; Philips, A. B.; and Laird, P. 1992. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 161–205.

Yokoo, M., and Hirayama, K. 1996. Distributed breakout algorithm for solving distributed constraint satisfaction problems. *Proceedings of the Second International Conference on Multiagent Systems (ICMAS-96)* 401–408.

Yokoo, M., and Hirayama, K. 1998. Distributed constraint satisfaction algorithm for complex local problems. In *the Third International Conference on Multiagent Systems (ICMAS-98)*, 372–379.

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

Zhou, L.; Thornton, J.; and Sattar, A. 2003. Dynamic agent ordering in distributed constraint satisfaction problems. In *Proceedings of the 16th Australian Joint Conference on Artificial Intelligence, AI-2003, Perth*.