Dynamic Prioritization of Complex Agents in Distributed Constraint Satisfaction Problems

Aaron Armstrong
Edmund Durfee

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109 USA
{armst, durfee}@umich.edu

Abstract

Cooperative distributed problem solving (CDPS) by loosely-coupled agents can be effectively modeled as a distributed constraint satisfaction problem (DCSP) where each agent has multiple local variables. DCSP protocols typically impose (partial) orders on agents to ensure systematic exploration of the search space, but the ordering decisions can have a dramatic effect on the overall problem-solving effort. In this paper, we examine several heuristics for ordering agents, and conclude that the best heuristics attempt to order agents based on the cumulative difficulty of finding assignments to their local variables. Less costly heuristics are sometimes also effective depending on the structure of the variables' constraints, and we describe the tradeoffs between heuristic cost and quality. Finally, we also show that a combined heuristic, with weightings determined through a genetic algorithm, can lead to the best performance.

Introduction

Cooperative distributed problem solving (CDPS) is often modeled as being done by a group of loosely-coupled computational agents involved in extensive local computations (Durfee, Lesser, and Corkill 1989; Luo, Hendry, and Buchanan 1993). Because these agents need to develop local solutions that together comprise one or more solutions to collective problems, they need to communicate intermittently about aspects of their local solutions to ensure compatibility. This may be usefully viewed as a distributed constraint satisfaction problem, where there are constraints between the local solutions of the different agents (Yokoo et al. 1992). The agents want to exchange enough information to identify and to rectify violations of constraints. Rapid delivery of pertinent information is essential for the agents to avoid computationally expensive dead-ends. The challenge is in controlling this exchange so that it does not swamp the agents with messages, and so that it efficiently results in convergence to consistent solutions.

One way of ensuring systematic exchange of partial solutions and of ensuring the identification of constraint violations is to order the agents, such that some agents make commitments to particular solutions around which others must work. If a work-around cannot be found, the system backtracks by asking agents up the pecking order to try different commitments. This strategy is the multi-agent version of a centralized, backtracking search. In fact, it is possible for backtracking to exploit parallelism, in cases where constraints are not highly constraining, by asynchronous backtracking (ABT) (Yokoo et al. 1992). With ABT, all agents in parallel pass their own variable assignments to relevant, lower priority agents and pass information on inconsistent combinations of value assignments (no-goods) to higher priority agents.

While instituting an ordering over the agents leads to systematic exploration, in the worst case there could still be an exhaustive search over the space of combinations of local solutions. To make this approach more effective, therefore, it can help if the agent ordering tends to focus search in more promising areas first. For example, highly constrained agents should have first choice.

Mapping this once again to the constraint satisfaction problem (CSP) framework, it would appear that ordering the agents is analogous to ordering the variables. In fact, this is the strategy that has generally been employed (Minton et al. 1990; Yokoo 1993), along with the typical assumption that each agent has one variable. The trouble in CDPS is that, to use communication bandwidth efficiently, the problem is distributed into a relatively small number of complex local problems—corresponding to a number of local CSPs. Realistically, the agents cannot be modeled as each having a single variable, but rather as each having multiple variables. In addition, considerations such as geographic distribution may suggest that we model the agents as each having a fixed set of variables, corresponding to a fixed problem decomposition (e.g. specified by resource availability). Now, even if variable ordering information is available, agents cannot be ordered strictly based on the variable ordering, because it is unclear how best to combine variable priorities to obtain a ranking of the agents. Further, even if good average-case methods for generating agent priorities were found, they could still be inferior to algorithms allowing dynamic priority assignment, since dynamic prioritization allows the CSP search process to discover and use additional information particular to the current problem. This research

This work has been supported, in part, by the National Science Foundation under PYI award 91-58473.

uses problems from a path planning domain to investigate the problems of how to prioritize dynamically and how to apply heuristics to ensembles of variables, so that DCSP algorithms can be efficiently used by the agents.

Distributed Constraint Satisfaction

In standard formulations of constraint-satisfaction, the problem is defined as one of instantiating an ordered set of variables V from a respective set of domains D such that a set of constraints C over the variables is satisfied. Figure 1 illustrates a CSP where V = (x_1, x_2, x_3, x_4), D = ({1, 2, 3}, {1, 2}, {1, 3}, {1}), C = { ($x_1 \neq x_2$), ($x_3 = 3$), ($x_3 = x_2 + 2$), ($x_3 \neq x_4$)}. A solution is (x_1, x_2, x_3, x_4) = (2, 1, 3, 1).

Backtracking search is a basic approach to solving CSPs. The variables are ordered and then the algorithm does a preorder walk of the implicit search tree. To reduce the size of the search space, we may use static or dynamic consistency methods to prune the tree, such as node, arc, and path consistency checks.

Asynchronous Backtracking

To allow us to study dynamic prioritization, we developed a DCSP protocol inspired by the asynchronous backtracking (ABT) (Yokoo et al. 1992) and weak-commitment (WC) protocols (Yokoo 1994). A key idea of asynchronous backtracking is to distribute the search problem and then allow the agents to work concurrently on their local problems. This creates potential parallelism by allowing each agent to actively guess solutions and by allowing agents to discover no-goods simultaneously. In ABT, each agent is responsible for a single variable, and the agents are usually related by a fixed, total order (though the communication is only between mutually constrained agents). They use periodic communication to synchronize constraint checking information. Each constraint is checked by the lowest priority agent among the set of agents involved in the constraint. Continuing the example above, the constraint ($x_1 \neq x_2$) would be checked by x_2, the constraint ($x_3 \neq x_4$) would be checked by x_4, and the other constraints would be checked by x_3. (Assuming a total order of $x_1 > x_2 > x_3 > x_4$.)

The process begins by having each agent assign a value to its variable and then passing that value to agents who are constrained by the agent's variable. Each agent, on receiving values from higher priority agents, checks to see if its own choice is compatible. If not, it tries to pick a new value. If it finds a legal new value, it passes on this change to its dependents and otherwise tells its parent that its parent's value is no-good. (A no-good is defined by a subset of the agents and their variables' instantiations such that some lower priority agent cannot instantiate its variable(s) without violating some constraint.) If an agent receives a no-good message, it records the no-good as a new constraint and tells the other (more important) agents involved in the no-good to keep the agent who is processing the no-good informed of any changes to their variables. The agent then tries to find a new value for its variable, as above. The algorithm terminates when the lowest priority agent has found a value consistent with all higher-priority agents—success. It also terminates when an agent discovers that it has an empty domain (e.g. from new constraints)—failure.

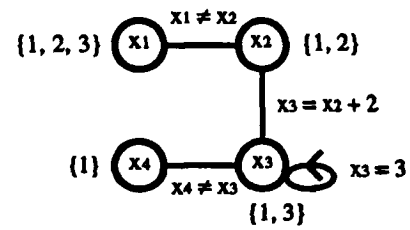

Figure 1

In the example above (figure 1), the agents choose (x_1, x_2, x_3, x_4) = (1, 1, 3, 1). Note that node constraints can be locally processed and so x_3 does not choose value 1. Each agent passes its information to agents checking mutual constraints. Agent (variable) x_2 receives x_1's message and changes its value to 2, sending along a message to x_3 stating the changes. Agent x_3 receives x_2's message and responds saying that $x_2 = 2$ is a no-good for x_2. Agent x_2 accordingly adds a self-loop constraint that $x_2 \neq 2$. Agent x_2 informs x_1 that $x_1 = 1$ is a no-good for x_1 and informs x_3 that it has now chosen $x_2 = 1$. Agent x_1 adds a self-loop constraint that $x_1 \neq 1$, chooses $x_1 = 2$, and sends this on to x_2. Agent x_2 receives this message and has already chosen $x_2 = 1$. The other two agents likewise do not need to change their assignments (a slight time savings from parallelism). The agents find the solution (x_1, x_2, x_3, x_4) = (2, 1, 3, 1).

Asynchronous Weak-Commitment Search

In (asynchronous) weak commitment search (Yokoo 1994, 1995), agents solve their local problems and check constraints in a manner similar to asynchronous backtracking, but whenever a no-good is discovered, the agent ordering is changed so that the agent who discovered the no-good now has highest priority. In weak-commitment search, the ordering is total and dynamic. Yokoo (1995) is not explicit in this regard, but many agents will need to be apprised of the situation whenever an agent is reprioritized so that constraints and no-goods will not be lost. Usually the lowest priority agent involved in a no-good stores the no-good information. Changing the order may then cause the no-good to be effectively forgotten, since another of the involved agents may become lower in the ranking and thus may eventually have to rediscover the no-good.

In our continuing example, the agents initially choose (x_1, x_2, x_3, x_4) = (1, 1, 3, 1). Agent x_2 then receives x_1's message ($x_1 = 1$) and changes its value to 2, sending along a message to x_3 stating the changes. Agent x_3 receives x_2's message and responds saying that $x_2 = 2$ is a no-good for x_2. The ordering then changes to (x_3, x_1, x_2, x_4). Agent x_3 sets its value to 3 and informs x_1. Agent x_1 retains its value of 1, causing x_2 to report a no-good of $x_1 = 1$. The priority then changes to (x_2, x_3, x_1, x_4). The agents choose (x_2, x_3, x_1, x_4) = (1, 3, 2, 1) and the problem is solved.

Summary

ABT and WC have similar benefits and a couple of problems. Their common benefit is the distribution of the problem and the result of possible parallelism. In addition, WC adds the ability to change bad orderings so that bad choices made by the high-priority agents will be discovered faster. One resulting problem of these distributed methods is the exponential space usage from the no-good storage. Another problem, the focus of our research, is the rigidness of the agent ordering. In ABT, the agent ordering is static and in WC, the ordering method is limited to a single heuristic.

Algorithm

Our algorithm was developed from the desire to use partial information from the search process to guide the remainder of the search, given our basic goal of investigating the problem of ordering agents with multiple variables. We generalized from ABT and WC to allow asynchronous local search with a more flexible reordering of the agent hierarchy.

Description

In our distributed algorithm, the variables and their domains are distributed among agents, each of which can check all constraints in which its variables are involved. There is also a central agent, which is responsible for starting and stopping the process and a no-good processor (possibly distributed), which keeps no-good information from being lost when the agents reorder.

The problem solving is divided into epoches, in which the central agent gives an initiation signal, the agents calculate and broadcast their priorities (thus establishing a total order of agents), and then the agents attempt to solve the problem. The epoch is terminated when a solution is discovered or when a no-good is discovered with fewer than m agents involved. (The parameter m is constant, typically small, during the search.) When $m = 0$, there will only be one epoch in the process, since a no-good involving zero agents implies that the problem has no solution. With this setting, the protocol reduces to a case of asynchronous backtracking in which communication is limited to an agent's immediate (typically two) neighbors in the total ordering. When $m > n$, where n is the number of agents, every no-good will cause a re-prioritization. With this setting and with the prioritization determined by a rapidly decaying number of no goods heuristic, the protocol reduces to weak-commitment search.

When a no-good is discovered, the variable assignments causing it and the IDs of the agents involved are sent to the no-good processor, which saves the no-good. Later, when an agent has found a tentative assignment for its variables, it consults the no-good processor to make sure that its assignment along with any known assignments of higher priority agents do not constitute a no-good. More on the no-good processor appears in the next subsection.

Each agent, after broadcasting and receiving priorities, constructs a tentative assignment of its variables. It consults with the no-good processor, then passes its assignment on to the next agent in the ranking. Gradually, information from higher ranking agents accumulates at each agent. This information is used to constrain the possible assignments of the agent's variables. All current information is passed on to the next agent in the ranking whenever a tentative assignment is made. In the case that the bottom agent is able to make an assignment to its variables and it has information from all higher ranking agents, the agent contacts the central agent to signal a solution to the problem. In the case that an agent cannot make an assignment to its variables, it has discovered a no-good. It will again contact the central agent, this time to signal a no-good (which may potentially trigger a new epoch). It will also signal the next-higher agent to find a new assignment for the higher agent's variables.

The No-Good Processor(s)

The issue of no-goods is crucially connected to completeness and to the space requirements of the DCSP algorithm. With asynchronicity, the algorithm could get stuck and continue to check and recheck the same variable assignments. To avoid this, as sections of the search tree are found not to contain a solution, data specifying the fruitless branches are recorded as no-goods to be avoided in the future. In asynchronous backtracking, when a no-good is discovered, the lowest priority agent involved in the no-good stores it in the form of an additional constraint.

In asynchronous weak-commitment search and in our protocol, using this method could result in the agents rediscovering and storing many copies of the same no-good, one for each time a different participating agent in the no-good had lowest priority. A no-good processor gives us the benefit of reduced storage costs (only one copy) and reduced search time (only discovered once). The tradeoff is in additional message passing. As mentioned before, we could distribute the no-good processor to reduce the computation and storage load on any particular process. In this case when an agent wants to check a collection of instantiations for no-goods, it sends the set of agents involved (and the variable assignments) to the no-good processors. The no-good processors are each responsible for mutually exclusive partitions of the power set of the agents. Each processor checks all subsets of the current list of agents corresponding to subsets in its piece of the partition. For example, with two agents and 2 no-good processors, processor A might store no-goods involving agents {1} and {2}, while processor B might store the no-goods for {1, 2}. If an agent wanted to check whether an assignment to the variables of agents {1, 2} was valid, processor A could check no-goods involving just agent 1 or just agent 2 and B could check no-goods involving both agents. Note further that if information from the higher-priority agents is consistent, the processors only have to check subsets involving the most recently added agent.

Performance Optimization

Now that the basic algorithm has been described and since it is a complete search technique, the next question is one of performance. Since CSP is NP-complete, we will merely attempt to improve the average case performance.

Heuristics. Obviously, to get good average performance from any search algorithm, we need to focus the search in more promising areas. These DCSP algorithms are focused by determining which agents have precedence and which values they prefer (choose first) for their variables. Value ordering heuristics, though important, were not the aim of this research and so were not used. We concentrated instead on the problem of deriving agent ordering heuristics from variable ordering heuristics which have been described in the literature, e.g. (Yokoo 1993; Minton et al. 1990). In much of the previous work, there had been a trivial derivation: since each agent had one variable, an ordering of the variables constituted an ordering of the agents. With multiple variables for an agent, there must be a method of combining the ordering information of single variables to produce agent-ordering information.

To improve the performance of our algorithm on test problems in the path-planning domain (discussed later in the paper), we investigated various heuristics. For uniformity and control purposes, we used two null heuristics—one static and the other dynamic. The remaining heuristics attempted to quantify the degree of constraint on an agent, equating "more constrained" with "more important."

- Random (but static). This heuristic would initially assign a random total order to the agents and hold the order fixed throughout the search. This was our standard for comparison.
- Random (dynamic). This heuristic randomly generated a different order with each reprioritization.
- A pseudo-heuristic which ordered the agents the same way that the dynamic random heuristic had ordered them when it had finished solving the same problem. This allowed us to examine the importance of ordering vs. building up knowledge of no-goods.
- The number of no-goods discovered or a decaying average of the number of no-goods discovered. These heuristics gave priority to agents which had discovered larger numbers of no-goods. The hope was that this would dynamically determine the most constrained agents. An exponentially decaying average is the basic heuristic of Yokoo's weak-commitment search (1994).
- Total or average number of different values in a single agent's domains. These heuristics gave priority to agents with fewer choices for their variables.
- A weighted average of the domain sizes. This heuristic is similar to the last one, but it gave more weight to variables representing important choices (e.g. more likely to conflict). In the path planning domain, the size of the domains of variables representing agent location at the middle of the path were considered to be most informative and so were given the greatest weight.
- Number of local solutions. This heuristic requires exhaustive constraint satisfaction internal to the agent to be performed first, effectively reducing the agent's set of variables to a single variable. If this is computationally infeasible, we could also estimate this number by checking some subset of the local problem for solutions. Another variation is to dynamically account for changes in the number of local solutions as no-goods accumulate and rule out solutions.

Combining Heuristics. In addition to investigating single heuristics, we also implemented an algorithm to automate performance tuning of combinations of heuristics. Priorities were assigned to agents by combining the heuristic values in a weighted sum. We used a genetic algorithm (Holland 1992) to search the space of heuristic weightings, to automatically discover which heuristics were effective, and if possible to exploit epistatic relations between them.

Example. In our toy example (figure 1), there are four problem-solving agents, each with one variable. (In our experiments, there are more variables for each agent—around 5-10.) The other two agents are the central agent and the no-good processor.

At the beginning, the central agent broadcasts an initiation message to all the agents. The agents calculate their priorities and broadcast them to each other. Suppose the ordering is ($A = x_1$, $B = x_2$, $C = x_3$, $D = x_4$), where we equate agents with single variables. The agents then choose (x_1, x_2, x_3, x_4) = (1, 1, 3, 1). Each checks with the no-good processor, which okays these choices. Agent A sends B a message that $x_1 = 1$. B discovers a conflict. It changes $x_2 = 2$ and passes (x_1, x_2) = (1, 2) to C. Agent C discovers a conflict. C determines further that there is no selection for x_3 that is consistent with the assignments to x_1 and x_2. It tells the no-good processor that the ordered agent set (A, B) cannot use the assignments (x_1, x_2) = (1, 2). Agent C sends a message to B telling it to try a different local solution and also sends a message to the central agent announcing a no-good involving (A, B).

If we assume the constant $m > n$, the central agent will start a new epoch for each no-good discovered. It instructs the agents to restart. They recalculate their priorities. Let us assume that agents who have discovered more no-goods and who have fewer solutions get higher priority. The ordering may then change to (C, D, B, A), where C's no-good increased its priority. The agents pick (x_1, x_2, x_3, x_4) = (1, 1, 3, 1). The no-good processor then forces A to choose 2 instead. The agents pass on information until A has received information from the others. Agent A now has a consistent assignment, minimal priority, and solutions from all higher priority agents, so the CSP has been solved. Agent A notifies the central agent, which broadcasts a halt message.

As this example illustrates, we can potentially benefit from the parallelism of computation as in ABT. We can also benefit from a good initial ordering and any reordering

of the variables, using dynamic information on the number of no-goods discovered and increasingly accurate estimates of the number of local solutions.

Experimental Evaluation

To evaluate our protocol and test our intuitions about various heuristics, we selected a simple family of problems and collected statistics on the performance of the algorithm using the different heuristics.

Problem Domain

We chose a problem in the domain of multiple-agent path planning as a source of the agents and their constraints. The n agents inhabit nodes in an arbitrary graph. Each agent starts at some node and must arrive at some other node. There are t time steps available for solving the problem. At each time step, each agent traverses an edge (possibly a self-loop). The solution to the problem consists of n paths between the n starts and goals. The paths must not conflict with one another (no simultaneous occupation of the same node or edge).

In figure 2 we have an example of a small problem. There are 3 agents, A, B, and C. This could potentially be a starting configuration. If the final configuration was (A, B, C) = (6, 4, 5), we could have a solution in which A uses path (1, 2, 3, 6); B uses path (7, 8, 5, 4); and C uses path (6, 9, 8, 5). In the CSP formulation, each agent might have 4 variables, corresponding to its position at times 1-4.

Topologies

We considered various topologies, since the topologies affect the kinds and numbers of constraints between variables more generally. We looked at randomly connected graphs, tree graphs, grid graphs, and hub graphs (each graph had multiple hubs). We did much of our work with grid graphs for ease of visualization and with hub graphs as a source of more difficult problems.

Experimental Analyses

We here summarize a few of our experimental results, focusing on the hub graph topology, but unless explicitly stated otherwise the reader can assume that the trends reported apply to the other topologies as well. We measured the impact of each agent-ordering heuristic on the time required by the agents to solve their path problems and on the amount of communication overhead incurred—the number of messages passed. The charts in figure 3 represent averages over 1200 randomly generated problems for the hub topology (with 12-16 nodes in each graph).

Effects of Agent Ordering. During the constraint satisfaction problem-solving, performance depends on the accumulation of additional constraints (discovery of no-goods) as well as on the ordering of agents. To get an appreciation of the relative influence of these factors, we can use the situation where the agents were given a fixed order for comparison. The order can be assigned randomly (in the charts, this is the "uniquifier" column) or can be based on the final ordering discovered by the dynamic random heuristic (in the chart, this is the "last run" heuristic). As shown in the charts (rightmost columns), a good ordering results in a time savings of 37% and a message savings of 32% compared to a random ordering. Similar savings occur in other topologies. Clearly, proper ordering can make a big difference.

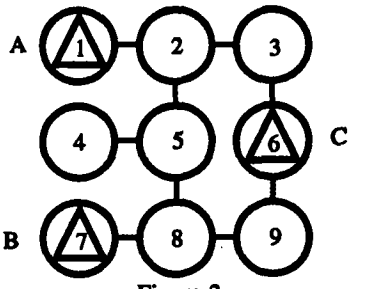

Figure 2

Effects of Dynamic Ordering. As just seen, agent ordering can make a big difference. If a random order starts out badly, performance will suffer. If agents are given a chance of reordering dynamically, their performance might improve even if the reordering is random (the "random" column in the charts). Our experiments bear this hypothesis out.

Performance of Heuristics Based on No-Goods. Yokoo's weak commitment search strategy would assume that an agent which discovers a no-good is highly constrained and should be moved to the front of the priority list. We evaluated this heuristic ("Decaying NG"), along with the variation of this heuristic that prioritized agents based on total no-goods discovered so far ("Num NGs"). Putting the most recently over-constrained agent first does shuffle the ordering to improve performance better than a random reshuffling, but we discovered that, at least for the DCSPs created in our domain across various topologies, it is better to consider more history: the total number of no-

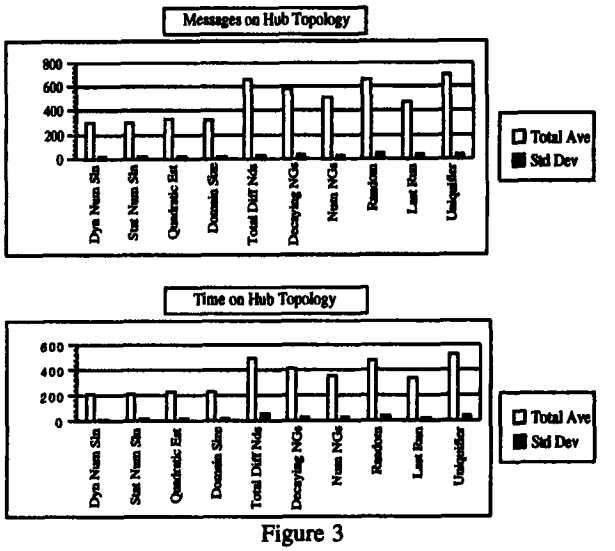

Figure 3

goods discovered heuristic generally performed better, and never performed worse, than the decaying no-goods heuristic that mimics Yokoo's.

Performance of Heuristics Based on Domain Size. A widely used heuristic for ordering variables in CSPs is to assign the most-constrained variable first. Our heuristics approximated this by assessing the total sizes of the domains of all of an agent's variables ("Total Diff Nds"), the average domain size of an agent's variables ("Domain Size"), and a quadratically weighted average of the domain sizes ("Quadratic Est"). The first did poorly overall, probably because the different agents tend to have similar domain sizes (similar path characteristics), but the latter two generally did well, though slightly less well than the number of solutions, discussed next.

Number of Local Solutions. Of course, a better approximation of the most-constrained variable heuristic is to treat each agent as having a single variable (as in previous DCSP analyses). In our case, this amounts to having an agent generate the set of legal solutions to its local CSP, and then ordering the agents in terms of which of them has fewer local solutions. We experimented with both static ("Stat Num Sln") and dynamic ("Dyn Num Sln") versions of this heuristic. Not surprisingly, it did very well across the board. Somewhat surprisingly, a static ordering did statistically as well as a dynamic ordering, but considering that usually constraints across groups of agents are discovered dynamically, we can see that dynamic changes to local solution sets may be minor.

Combinations of Heuristics. We used a generation-based genetic algorithm (GA) to study combinations of heuristics. Each of the 100 population members consisted of a concatenation of Grey-coded weights. The weights determined the amount of influence exerted by each heuristic. There was some difficulty with slow convergence because of the wide variation of the randomly generated problems.

After running the GA, we selected some of the top combination heuristics and ran them against several of our "pure" heuristics, measuring the number of time steps and the number of messages sent averaged over 1500 problems. Successful combinations placed high emphasis on the number of local solutions and on the number of no-goods. A high value was also accorded to the dynamic random heuristic. Combination heuristics proved to be modestly better than any pure heuristic.

In practice, such an algorithm could be used to gradually optimize the performance of a large CSP system. Heuristics and parameter settings could be guessed and as the system was used, it would adapt to the particular distribution of problems that it faced and identify the most useful heuristics with relatively little overhead.

Conclusions

While several researchers have recognized the similarities between cooperative distributed problem solving and distributed constraint satisfaction, the emphasis of the former on loosely-coupled agents solving substantial local problems has not been adequately addressed in the latter. In this paper, we describe a foray into this area of investigation. We have described a protocol that is assured of terminating and that generalizes asynchronous backtracking and weak commitment search to permit variations in the timing and criteria of agent reordering. Our empirical investigation using these capabilities has revealed that a good ordering is critical to performance, and that ordering based on the local solution spaces is most effective. However, because the computations for this heuristic amount to solving a substantial local CSP, more cost effective approximations are available that perform nearly as well. Yokoo's approach of placing the most recently over-constrained agent first is one possibility, although our results indicate that the total number of no-goods and the average domain size heuristics can be even more effective.

Many open problems remain, including characterizing the tradeoffs between local computation and the benefits of the heuristics. We also need to investigate further decentralization of the protocol and storage of no-goods. Finally, it might be the case that a good ordering of agents is not possible without redistribution of variables, leading to issues of negotiation and load-balancing.

References

Durfee, E., Lesser, V., and Corkill, D. 1989. Cooperative Distributed Problem Solving. In A. Barr, P. Cohen, and E. Feigenbaum (eds.). *The Handbook of Artificial Intelligence*, Volume IV, Addison-Wesley.

Holland, J. 1992. *Adaptation in Natural and Artificial Systems: an Introductory Analysis with Applications to Biology, Control, and AI*, 1st MIT Press ed. Cambridge, MA: MIT Press.

Luo, Q., Hendry, P., and Buchanan, J. 1993. Heuristic Search for Distributed Constraint Satisfaction Problems. Research report KEG-6-93, University of Strathclyde, UK.

Minton, S., Johnston, M., Philips, A., and Laird, P. 1990. Solving Large-Scale Constraint Satisfaction and Scheduling Problems using a Heuristic Repair Method. *Proc. of AAAI-1990*, 17-24.

Yokoo, M., Durfee, E., Ishida, T., and Kuwabara, K. 1992. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. *12th IEEE Int. Conf. on Dist. Computing Sys.*, 614-621.

Yokoo, M. 1993. Dynamic Variable/Value Ordering Heuristics for Solving Large-Scale Distributed Constraint Satisfaction Problems. *Proc. of 12th Int. Workshop on Distributed Art. Int.*, 407-422.

Yokoo, M. 1994. Weak-Commitment Search for Solving Constraint Satisfaction Problems. *Proc. of the 12th National Conf. on Art. Int.*, 313-318.

Yokoo, M. 1995. Asynchronous Weak-Commitment Search for Solving Large-Scale Distributed Constraint Satisfaction Problems. *Proc. of the 1st Int. Conf. on Multi-Agent Systems*, 467.