From: AIPS 1996 Proceedings. Copyright © 1996, AAAI (www.aaai.org). All rights reserved.

Algorithms for Solving Distributed Constraint Satisfaction Problems (DCSPs)

Gadi Solotorevsky and Ehud Gudes
Dept. of Mathematics and Computer Science
Ben-Gurion University of the Negev, Beer-Sheva, 84-105, Israel
Email: {gadi,ehud}@indigo.bgu.ac.il

Abstract

This paper investigates Constraint Satisfaction Problems (CSPs) that are distributed by nature, i.e., there is a division of the CSP into sub components (agents) that are connected via constraints, where each subcomponent includes several of the CSP variables with the constraints between them. We call such a problem a Distributed CSP (DCSP). In this paper we give a formal definition of DCSPs and present four algorithms for solving them. Two of the algorithms are based on the difference between the difficulty of solving the internal constraints in the CSP components (we call them the peripheral components) of the DCSP and the difficulty of solving the constraints between the different CSPs (the central component). The two other algorithms use local and global views of the DCSP respectively. All the algorithms permit the use of different techniques (CSP, knowledge based, and operation research algorithms) in solving each of the problem components. We probe that as long as all the selected techniques are sound and complete, our algorithms are sound and complete. The algorithms were tested in a real distributed environment; the results show that when there is a difference between the difficulty of solving the peripheral components and the central one, taking advantage of it may reduce significantly the amount of work (constraint checks and message passing) needed for solving the DCSP.

Introduction.

Constraint satisfaction problems (CSP) appear in many domains, e.g., image processing and resource allocation problems (Solotorevsky G., Gudes E., & Meisels A. 1994; Sycara K. *et al.* 1991). Constraint satisfaction problems are in general difficult to solve. The research in solving these problems was focused in two main models, the sequential, e.g, (Dechter R. & Pearl J. 1987; Feldman R. & Golumbic M. 1990; Kumar V. 1992), and the parallel (Collin Z. & Dechter R. 1990; Kasif S. & Delcher A.L. 1990). This paper is focused in a third model, the distributed model (Burke P. & Prosser P. 1989; Yokoo M. et. al. 1990; Yokoo M. 1995).

Intuitively, a distributed CSP is a CSP divided into CSP subproblems, which are connected via constraints. (Constraints may exist between variables of the same sub-problem as well as between variables belonging to different sub-problems.)

The origins of many CSPs are problems that are solved in real life by several agents, each of them working on a part of the problem (Levine P. & Pomerol J. 1990; Prosser P.and Conway C. & Muller M. 1992; Sycara K. *et al.* 1991), e.g., the problem of assignment of train-cars to trains in France (Levine P. & Pomerol J. 1990). Therefore, often a division of the CSP into sub-CSPs is already given. The starting point of this paper is a situation where a distributed CSP is given. We will develop algorithms for solving the distributed CSP using several agents that are connected by a communication network (i.e., no common memory, just message passing). The number of agents that will be used is equal or larger by a small constant, to the number of subproblems in the given division of the CSP.

Research in sequential CSP is concentrated on efficiently solving large CSP problems. Methods were suggested in the literature to reduce domains of variables (thus reducing search space) by pre-processing (Dechter R. & Pearl J. 1987; Kumar V. 1992), and to reduce backtracking by using constraint propagation (forward checking) algorithms (Prosser P. 1993; Kumar V. 1992). In solving DCSP problems we can use the same techniques but we have two additional complexities:

- Since each agent is working independently and in parallel, we want to optimize the performance of the slowest agent rather than optimizing each individual agent.
- We want to minimize the number of messages exchanged between the agents, in order to reduce communication overhead.
- We would like to minimize the amount of backtracking each agent performs as a result of actions of other agents.

The algorithms presented in this paper will address these issues. We also want to guarantee termination and successful finding of a solution if such a solution exists (completeness and soundness).

From: AIPS 1996 Proceedings. Copyright © 1996, AAAI (www.aaai.org). All rights reserved.

Note that we use the term DCSP to describe a family of problems that are distributed by nature, and not to describe distributed techniques to solve CSPs (see (Yokoo M. 1995)), i.e., though one may look on a DCSP as a CSP and vice versa, and though a DCSP may be solved using sequential methods while a CSP may be solved using distributed methods, they are ontologically different. DCSPs that are based on real life problems will often have different characteristics in their components, e.g., some components may be more constrained than others. In developing algorithms for solving DCSPs we will try to take advantage of these differences that are commonly inherent to real DCSPs; this is different from the work done in solving CSPs in a distributed manner, where such differences shouldn't be assumed.

The algorithms presented in this paper enable the use of different methodologies for different components of the same DCSP (including CSP, knowledge based, and OR methods), an example of a DCSP problem that will be enhanced by this, is the problem of building a timetable for a department at the university. In this problem the assignment of teachers and times to each class requires large amounts of knowledge, therefore a knowledge based system will be appropriate. However, the task of assigning rooms requires less knowledge therefore a CSP approach will be more suitable.

In the first section we will give the definition of DCSP (Distributed CSPs) problems. In the second section we will develop four distributed algorithms for solving DCSPs. We will prove for each algorithm that it is sound and complete.

Definitions

DCSPs, similarly to CSP problems, are problems that require the assignments of values to variables according to some constraints. In DCSPs there is a set of m groups $G_1, G_2 ..., G_m$ and a mapping function M.

For each $1 \leq i \leq m$ there exist in the i-th group n_i variables $X_{1_i}, X_{2_i}, ..., X_{n_i}$ with domains $D_{1_i}, D_{2_i}, ..., D_{n_i}$. Each D_{k_i} defines the set of values that X_{k_i} can get.

A **relation** R on these variables is a sub set of the Cartesian product:

$$R \subseteq D_{1_1} \times D_{2_1} \times ... \times D_{n_1} \times ... \times D_{1_m} \times D_{2_m} \times ... \times D_{n_m}$$

A **binary** constraint $R_{l_j k_i}$ between two variables X_{l_j}, X_{k_i} is a sub set of the Cartesian product between their domains, $R_{l_j k_i} \subseteq D_{l_j} \times D_{k_i}$. When $j = i$ the constraint is called **internal**, otherwise, when $j \neq i$ it is called **external**. The case where $k_i = l_j$ denotes a **unary constraint** on X_{l_j}.

M is a function that maps the variables in the set G_m into variables on the other G sets. Each variable in the G_m set is mapped to a single variable in one of the other sets, and no two variables in the m group are mapped to the same variable.

A **Distributed Binary CSP (binary DCSP)** is the set of variables

$$X_{1_1}, X_{2_1}, ..., X_{n_1}, ..., X_{1_m}, X_{2_m}, ..., X_{n_m}$$

and the set of binary and unary constraints on these variables.

A tuple P is a **Solution to a binary DCSP** if and only if:

1. All the binary and unary constraints behold in P

$$P \in \{x_{1_1}, ..., x_{n_1}, ..., x_{1_m}, ..., x_{n_m} \mid \forall_{k_i}(x_{k_i} \in D_{k_i}) \wedge \forall_{k_i l_j}(x_{k_i}, x_{l_j} \in R_{k_i l_j})\}$$

2. Each variable that belongs to the group G_m gets in P the same value that gets the variable on which it is mapped by M.

$$\forall_{1 \leq l \leq n_m}(M(X_{l_m}) = X_{k_i} \rightarrow x_{l_m} = x_{k_i})$$

A **canonical** DCSP (Figure 1) is a DCSP in which:

1. Each external constraint includes exactly one node of a group other than G_m and all the other nodes participating in the external constraint are members of G_m.
2. For each internal constraint between nodes that are all mapped into nodes in G_m, there exists in G_m exactly the same constraint between all the mapped nodes.

In **canonical** DCSPs both the CSP components of the DCSP ($G_1..G_{m-1}$) and the constraints between them (G_m) are represented as CSPs, . Therefore solving a canonical DCSP consists on solving m CSPs, plus taking care that the values given to the nodes in G_m are identical to the values given to the nodes on which they are mapped by M. This feature will be used in algorithms 3 and 4 in section 3.

An **explicit** DCSP is a DCSP whose G_m group is empty, see figure 3. Note that both concepts; an **explicit** DCSP, and a **canonical** DCSP are not restricted to DCSP that contains only binary and unary constraints.

The motivation behind the concepts of Canonical and Explicit DCSPs will become clearer when we present the algorithms. Basically a canonical DCSP representation allows checking the inter-nodes constraints using a single node (agent) and thus provides careful monitoring of effects of changes made by one node on other nodes.

Solving DCSP Problems

In the rest of this paper we assume a given DCSP with a division into $G_1, ..., G_m$ subproblems, a mapping function M, and m agents. Each agent A_i ($1 < i < m$) has a representation of all the variables and constraints in G_i (the internal constraints). For each pair of variables X_{l_j}, X_{k_i} that are mapped by M into X_{m_a}, X_{k_b} then representation of the constraint among them will be kept both in agents A_k, A_l, and A_m.

From: AIPS 1996 Proceedings. Copyright © 1996, AAAI (www.aaai.org). All rights reserved.

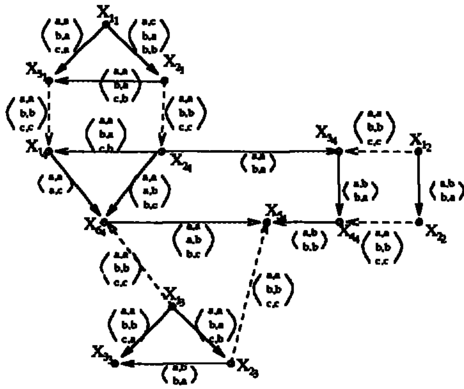

Figure 1: A canonical DCSP

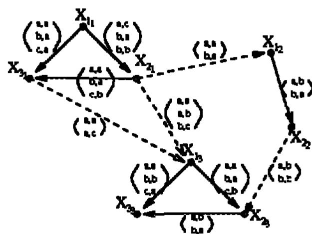

Figure 2: An explicit DCSP

In developing the algorithms for solving the DCSP, special emphasis is given to posing no restriction to the way used by each agent to solve the internal CSP for which it is responsible, the agents will only be restricted in the methods they should use for solving the external constraints. That is, in all the algorithms presented below, each agent can select any algorithm (e.g., Failure Directed Backjump (Prosser P. 1993), constraint propagation (Kumar V. 1992), etc.) for solving its internal CSP. The Algorithms in this paper ensure, as will be proven, that if every agent selects an algorithm that ensures to find a solution for the internal CSP if such exists, or otherwise to stop and return a negative answer, then the algorithms presented in this paper will find a solution to the DCSP when such solution exists or otherwise will halt giving a negative answer.

Low Level Procedures To represent the algorithms we will use four procedures:

- **solve_internal(G_i)** is a procedure that finds a solution to the CSP G_i with no concern for the external constraints.
- **propagate_external(G_i)** is a procedure that informs all the agents that are connected to G_i via constraints, about the values assigned to the variables of G_i.
- **update_propagate(G_i)** is a procedure that informs the agents that are connected to G_i via constraints, about the changes between the values of the variables reported by G_i the last time that propagate_external(G_i) or update_propagate(G_i) were used and the actual values in the variables of G_i. Only agents that are connected via constraints to variables whose values were changed are reported about the change.
- **external_conflict_backtrack(G_i)** is a procedure that seeks an alternative solution for G_i. It tries to find an alternative solution that involves a small amount of changes in the values gotten by variables connected by external constraints. The alternative solution must be a solution that is different from all the solutions found to G_i since the last call to solve_internal(G_i). Note that each call to solve_internal(G_i) enables external_conflict_backtrack(G_i) to find any solution to G_i, except, of course, the one found by solve_internal(G_i).

Algorithm 1. The simplest algorithm for solving a DCSP problem is using a sequential method, i.e., select one of the sequential algorithms for solving a CSP problem and use it on the DCSP (this is equivalent to synchronous backtracking in (Yokoo M. *et al.* 1992)); therefore, at any given time only one agent will be performing an action (checking a constraint, doing an assignment, or undoing an assignment). It is clear that if algorithm 1 uses a sequential algorithm that is sound and complete then algorithm 1 is sound and complete. The advantage of this approach over the transformation of the DCSP into a CSP and then solving it using a sequential algorithm, is that the knowledge acquisition and representation aren't damaged. The principal disadvantage of this approach is the huge waste of the working potential of the agents; at each moment $m-1$ agents won't be doing any work. Another disadvantage is that this method isn't aware of the fact that checking external constraints is much more expensive than checking internal constraints.

Algorithm 2. The second algorithm, similarly to the first one, uses a sequential algorithm in order to solve the DCSP, but in this algorithm an order between the agents is established, and all the nodes belonging to the same agent will be initialized one after the other. This strategy aims to reduce the amount of messages between nodes in different agents.

```
solve_dcsp2(i)
begin
 if (i==m+1)
   return true;
 else
  if solve_internal(G_i)
  begin
   propagate_external(G_i)
   solve_dcsp2(i+1)
  end
  else
   return backtrack_dcsp2(i-1)
end;
```

```
backtrack_dcsp2(int i)
begin
 if (i==0)
   return false
 else
  if (small_change_solution(G_i))
  begin
   update_propagate(G_i);
   return solve_dcsp(i+1);
  end
  else
   return backtrack_dcsp2(i-1)
end
```

The heuristic for ordering the agents can be selected according to the expected difficulty of solving the internal CSPs of each agent. The agents with the largest expected difficulty will be solved first. (There are many

From: AIPS 1996 Proceedings. Copyright © 1996, AAAI (www.aaai.org). All rights reserved.

possible methods for the estimation of the difficulty, e.g.,the number of nodes in the agents, etc..). Following is the pseudo-code that describes algorithm 2, it assumes that the order of the agents initialization is $A_1, A_2, ..., A_m$.

In order to demonstrate that algorithm 2 t is sound and complete, (and also algorithms 3 and 4, that will be soon described), we will look on the DCSP D as a CSP C in which for each group G_i with n_i nodes in the DCSP exists a node G'_i whose values domain are tuples of size n_i. A tuple is a legal value of G'_i (it is legal according to the unary constraint of G'_i) if and only if for every k, assigning the same values to g_k, as to the k location in the tuple yields a solution for the CSP G_i. Such a solution will be called the **equivalent solution of the tuple**.

An assignment of tuples to all the nodes in the CSP C will be a solution of C if and only if the equivalent solutions of all the tuples in D are a solution of D. I.e., in the CSP C, n_i and n_j that are legal values of G'_i and G'_j respectively (are legal according to the binary constraints between G'_i and G'_j), if an only if the equivalent solution of G_i and G_j is legal according to all the constraints between them. The above described interpretation of the DCSP as a CSP will be called the **naive interpretation**.

The meaning that the previously defined functions have according to the naive interpretation:

solve_internal(G'_i): finds a legal value for G'_i.

external_conflict_backtrack(G'_i): finds "a new" value for G'_i. The meaning of "new" value is a value that wasn't returned by this function since the last time that solve_internal(G'_i) was called.

update_propagate(G'_i)
and **propagate_external(G'_i):** are interpreted as an updating of the allowed values in the vertices that connect the node G'_i with other nodes.

Algorithm 2, according to the above presented interpretation, is equivalent to depth first search on a graph of a CSP (with a final number of nodes, and a final number of possible values for each node) with a fixed order of nodes. Since depth first search ensures, under those conditions, finding a solution if such exists or otherwise it halts giving a negative response, so will do algorithm 2.

Algorithm 3. Algorithms 3 and 4 work on canonical DCSPs, nevertheless they can be used to solve any DCSP after finding its equivalent canonical DCSP. The main idea in algorithm 3 is to find first the solution for G_m, i.e., to find a legal assignment of values for all the nodes connected to other nodes via external constraints, and then to find solutions for all the other G_i's that are compatible with the solution found for G_m. This will be done as following:

1. Find a solution for G_m.
2. Do constraints propagation from the found solution of G_m to all the other groups.
3. Find a solution for $G_1, ..., G_{m-1}$. Due to the constraint propagation, it is sufficient to find a local solution for each of $G_1, ..., G_{m-1}$.
4. If a solution for all of them was found, then a solution for the DCSP was achieved, otherwise:
   - if there is at least one G_i that we failed to solve, then seek a new solution for G_m and continue from 2.
   - otherwise halt and return false.

Note that the constraint propagation in stage 2 must also be done to groups whose current solution isn't immediately affected by the new solution of G_m. this is so both for achieving a correct backtracking and for finding a new solution.

The comprehension of algorithm 3 under the naive interpretation is that we test for G'_m each of the values that are legal according to its unary constraints; and for each of those values, we check if there are legal values, according to the unary constraints, for all the G'_is which are compatible, according to the binary constraints, with the value given to G'_m. And so on until a solution is found or until there isn't any new value for G'_m to test.

Since m is a final number and the number of values in the domain of each G_i is final, then the number of tuples for all the G_is (including $i = m$) is final; therefore this algorithm will find a solution if such exists, or otherwise will halt giving a false response.

```
solve_dcsp3()
begin
  solve_internal(G_m)
  propagate_external(G_m)
  if (|| solve_internal(G_1) and ... and solve_internal(G_{m-1}))
  /* parallel solve G_1...G_m if they all succeed then a solution is found,
  otherwise if one of these fails then without waiting for the
  others to answer do backtrack */
    return true
  else
    return backtrack_dcsp3()
end;
backtrack_dcsp3()
begin
  if (external_conflict_backtrack(G_m))
  begin
    update_propagate(G_m)
    if (|| solve_internal(G_1) and ... and solve_internal(G_{m-1}))
    /* parallel solve G_1...G_{m-1} if they all succeed solution found.
    otherwise if one of these fails then without waiting
    the others answer, then do backtrack */
      return true
    else
    return backtrack_dcsp3()
  end
  else
    return false
end;
```

The main advantages of algorithm 3 are that it enables performing distributed work (during the stage of finding solutions for $G_1, ..., G_{m-1}$) and doing so, by sending relatively small number of messages (they will be sent only after the stage when one of the agents fails to find a solution adequate to the solution of G_m).

The main disadvantage of this algorithm is that some times great parts of the performed distributed work , will be irrelevant, i.e., an agent may do a lot of work in finding a solution which will be found to be irrelevant due to the failure of another agent. This disadvantage will be reduced if the agents will use a method

From: AIPS 1996 Proceedings. Copyright © 1996, AAAI (www.aaai.org). All rights reserved.

that learns from its failures (e.g., by ussing a TMS), so when the agent will look for another solution, it will be helped by the knowledge previously acquired by him.

Algorithm 4. This algorithm uses agents $A_1, .., A_{m-1}$ which each tries to solve its own problem and they all use a central axe, A_m, only for synchronization in the backtrack stage. In this algorithm, each agent (excluding A_m) finds a solution for its local problem and then checks if this solution is compatible with the local solutions already found by the other agents. When the new solution is incompatible, then backtrack is performed and forward search is continued only after a compatible solution is found. Note that checking external constraints at the assignment stage, before local solutions are achieved, can be both difficult and wasteful due to the fact that the assignments values may change a great deal in this stage.

The approach in this algorithm is quite the opposite to the one in algorithm 3; there we first tried to find a solution for G_m, and afterwards to find compatible solutions for $G_1, .., G_{m-1}$; here we will first find solutions for $G_1, ..., G_{m-1}$ and only afterwards, we will test if they are a legal solution for G_m.

In this algorithm,

- Distributively search for solutions for the local problems of agents $A_1, ..., A_{m-1}$.
- When an agent finds a solution for its local problem, it updates A_m. A_m checks if the new solution (the $n+1$ solution) is compatible with the values of the solutions previously found by the other agents. If it is then:
  - If all the agents already found a solution, then a solution for the DCSP is found.
  - Otherwise if the solution is compatible but A_m hasn't already received solutions from all the other agents then he waits for them.

  Otherwise if the n+1 solution is incompatible with the n solutions already found, then backtrack is performed between the $n+1$ agents that gave the solutions. If the backtrack is successful, then A_m waits for new solutions (if needed), otherwise the algorithm halts and returns false. Note that this backtracking must be performed sequentially in order that one agent backtrack will not conflict with another agent backtrack, however, as soon as one backtrack results with a valid solution for G_m the process can stop.

```
solve_dcsp4()
begin
  if (||find_csp_solution4(G1) and,..., and find_csp_solution4(Gm-1))
    then DCSP solved
  else fail to solve DCSP
end.

find_csp_solution4(Gx)
begin
  solve_internal(Gx)
  if update_values_in_Gm(Gm) then
    return true
  else
  return false
end.
update_values_in_Gm(Gx)
/* this is a critical section that can be entered
only by one Agent at a time */
begin critical
  set the relevant values of Gx in Gm;
  if (Gm is not in a legal state) then
  while (Gm is not in a legal state) do
    select Gi for backtrack
    if (backtrack between the Gis which have given value s to Gm
      sets Gm to a legal state) then
      return true;
    end
    else
      return false.
  else
    return true;
end critical;
```

We can prove the soundness and completeness of this algorithm by using the naive interpretation in a similar way we did in algorithm 3, for a complete proof see (Solotorevsky G. & Gudes E. 1995).

The main disadvantage of this algorithm is that the backtrack is performed sequentially. This disadvantage may be overcome by using the agents for additional work in the stages when they are not active due to the sequential work, e.g., finding alternative local solutions that can be used in a future backtrack stage or exploring the local search tree using an algorithm that allows learning.

Evaluation and Testing of the Algorithms

Algorithms 3 and 4 were designed with view to two different families of DCSPs: the target of algorithm 4 is to solve DCSPs in which the central component is dominant to the other components; dominant in the sense that it is relatively difficult to solve it than to solve the other components. Algorithm 3 is aimed to problems in which the peripheral components are dominant. In this section we will test how well these algorithms are suited for their purposes. We will test the improvement achieved by exploiting the DCSPs nature as is done in algorithms 3 and 4 in comparison with algorithm 1 that completely ignores the DCSP distributed structure, and with algorithm 2 that makes a very naive use of the DCSPs distributed nature.

We opted to test our Algorithms in a real environment, as opposite to a simulation, this gives a more realistic appreciation of the algorithms performance, however this makes time measurements highly dependent of the activity in the Local Area Network (LAN), therefore we opted to use two parameters to evaluate the algorithms: the "Maximal Number of Constraints' Checks" (MNCC) and the number of messages sent by the agents. In the process of applying our algorithms to solve a DCSP there are some intervals in which the agents work in sequence and some other intervals in which they work in parallel. We defined MNCC as the sum of all the constraints' checks done in the sequential intervals plus the sum of the maximal number of constraints' checks done by one of the agents in each parallel interval.

Algorithms Implementation.

As we point out throughout the paper, the algorithms previously presented leave lots of freedom to how to

From: AIPS 1996 Proceedings. Copyright © 1996, AAAI (www.aaai.org). All rights reserved.

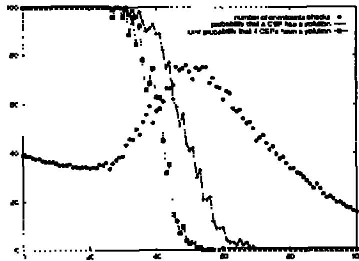

Figure 3: constraints checks and probability of CSPs to have at least one solution.

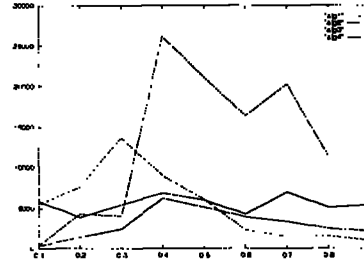

Figure 4: constraints checks for $p_2 = 40$.

implement the basic features, e.g., which technique to use in solve_internal(), or how to use the idle time of an agent to perform some kind of learning. A smart implementation of these techniques may enhance the performance of the algorithms, however the scope of this paper is to check the relative gains in using algorithms that exploit the inherent distribution of the DCSPs, and not to check and compare different implementations of these algorithms, therefore we opted for testing relatively non enhanced versions of the algorithms. In order to provide a fair comparison between the algorithms, we implemented in all of them the low level procedures in a similar way: Solve_internal finds a solution for a CSP using limited constraint propagation and simple backtracking. No heuristic is used for variable ordering. external_conflict_backtrack() finds the next solution according to the backtracking order.

In Algorithm 1 we used limited constraint propagation, failure directed backjumping and a random variable ordering. We decided to use here failure directed backjumping since using backtracking was far too slow. Nevertheless we used backtracking in the other algorithms to avoid a non standard implementation of backjumping that is needed in order that external_conflict_backtrack() works properly without skipping possible solutions.

The set of experiments.

To generate the set of experiments to test the algorithms, we parameterized both the central component of the canonical DCSPs, and the peripheral components. We used 8 parameters to characterize the DCSPs: *Cnum* - the number of CSP components in the DCSP. For each DCSP in the explicit representation of the DCSP we have 4 parameters for characterizing it (the same ones used in (Prosser P. 1994)): n - the number of variables, m - the number of values in each variable domain, p_1 - the probability that there is a constraint between a pair of variables, and p_2 - the conditional probability that a pair of values is inconsistent for a pair of variables, given that there is a constraint between the variables. The central CSP (in the canonical representation) is defined by 3 parameters: pe - the probability that a node in a peripheral CSP participates in an external constraint. pc - the conditional probability that given two variables that belong to two different CSPs, and that both were selected to participate in external constraints, then they are connected via a constraint. pv - the conditional probability that a pair of values is inconsistent for a pair of variables, given that the pair of variables is connected via an external constraint.

Graph 3 was created by testing CSPs with $n = 5$, $m = 4$, p_1=100. p_2 goes from 0 to 1 in steps of 0.1, at each step 100 tests were made. It shows both the average number of constraints' checks needed to find the first solution if there is a solution, and the probability of a CSP to have a solution, and the joint probability that 4 independent CSPs have solutions. E.g., when $p_2 = 50$ the average number of constrains checks needed to find a solution is about 80, and the probability that there is a solution is about 40. We solved the CSPs using limited constraint propagation. Note that we represent the CSPs by maintaining lists of legal pairs (instead of lists of illegal pairs), this is the reason that caused constraint propagation to take place even for a graph with no constraints ($p_2 = 0$).

In all our tests we used $n = 5$, $m = 4$, $p_1 = 100$. Since we are interested in the effects on the difficulty caused by the difference between the difficulty of solving the central CSP and the difficulty of solving the peripheral ones, then we opted for performing two sets of tests one with $p_2 = 0.4$ where it is relatively difficult to solve all the 4 peripheral CSPs, and the second with $p_2 = 0.2$, for which the peripheral CSPs are relatively simple to solve (see figure 8). In each set we did experiments with $pe = 0.3$, $pc = 0.5$ and pv running from 0.1 (easy) to 0.9 (difficult) in steps of 0.1, at each step we performed 50 tests. The tests were performed using several computers connected via a LAN.

Figures 4 and 5 show correspondingly the number of MNCC and messages that have been done by the algorithms for solving DCSPs with $p_2 = 40$, e.g., when pv is 30 algorithm 1 does 5000 MNCCs and sends 400 messages. Finding a solution for all the peripheral components, in each of the DCSPs solved in these figures, is quite difficult (see figure 3). When the central component is easy to solve, then finding a solution for it is not of much use for finding a solution for the whole

From: AIPS 1996 Proceedings. Copyright © 1996, AAAI (www.aaai.org). All rights reserved.

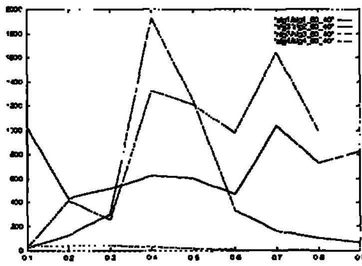

Figure 5: messages sent for $p_2 = 40$.

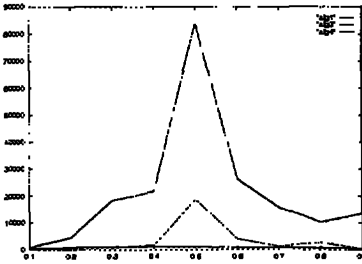

Figure 6: constraints checks for $p_2 = 20$.

DCSP; this is the reason why algorithm 3, which first seeks for the solution of the central component, isn't too effective in the region where p_2 goes from 0 to 30. However, when the central component becomes more difficult than the peripheral ones (p_2 40 to 80) then it is effective to solve it first. The number of messages sent by algorithm 3 remains low since it quickly learns from his failures which values are incompatible with the peripherals. Algorithm 4, that first solves the peripherals, performs a small amount of MNCCs when the central component is easy to solve, since most of the solutions that it finds (in a parallel process) for the peripherals, will be good for the central component; however when the central component becomes more difficult, many solutions will be rejected by it, which will lead to sequential backtracking that will greatly increase the number of messages. When the central component becomes very difficult, many possible assignments will be immediately rejected and this will reduce the number of MNCC and of messages.

Algorithm 1, that uses a global view of the DCSP, gives a relative low number of MNCC for all the DCSPs; however since it uses constraint propagation, it requires a high number of messages.

Algorithm 2 doesn't take advantage either of a global view of the DCSP or of parallelism. The result is a very high number of MNCC and messages (except for the zone where the central component is so easy that it can practically be ignored). Note that when the central component becomes very difficult, then the amount of messages and the MNCC decreases a bit, since many of the failures are due to peripheral components that are adjacent in the order selected by algorithm 2.

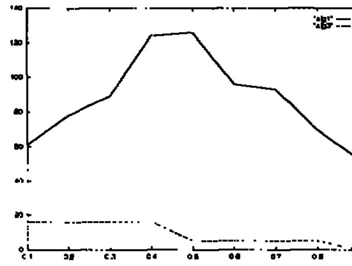

Figure 7: messages sent for $p_2 = 20$.

Figures 6 and 7 show correspondingly the number of MNCCs and messages that have been done by the algorithms for solving DCSPs with $p_2 = 20$, i.e., DCSPs where the peripheral components are easy to solve. Algorithms 2 and 4 performs very badly (their results didn't fit into the graphs scales therefore they were omited) the reason for this is that they both work according to the peripherals components, and afterward check if the result is correct for the central one, however there are many possible solutions for the peripheral components and only a few of them will fit to the central one. Algorithms 3 and 1 performed very well, algorithm 1 has some advantage in reducing the number of MNCCs while algorithm 3 has some advantage in reducing the number of messages.

Selecting an algorithm.

Selecting a suitable algorithm for a DCSP problem depends both on the problem characteristics, specially the relation between the difficulties of the peripheral components and the central one, and on the price expected for a constraint check vs. the price of a message. Obviously a message in a LAN is much more costly than a binary constraint check when an explicit representation is used; however in many DCSPs the constraints are not explicitly represented and may involve heavy procedures and Data Base queries. Table 8 shows which algorithms are recommended for which kinds of problems; of course the table isn't conclusive; it just recommends educated guests about the algorithms that could be selected, but for an individual problem it isn't assured that the recommended algorithm will be the best.

An interesting fact that becomes clear from table 8 and from figures 5 and 7 is that algorithm 3 is most of the time the best in keeping a low amount of messages. The reason for this behavior is that after it sends a solution to a peripheral component P it gets an answer; if there is a solution of P which is compatible to the solution found for the central component or if there is no solution. The agent that deals with the central component records this information, therefore the next time that a solution for the central component is found, that has the same values in the nodes connected to P, the agent of the central component knows whether P

From: AIPS 1996 Proceedings. Copyright © 1996, AAAI (www.aaai.org). All rights reserved.

type	reduce messages num	reduce MNCCs
difficulty in central component	1 or 3	3
difficulty in peripheral components	1 or 4	3
even difficulty (in difficult problems)	1 or 3 or 4	3

Figure 8: Which algorithm to use ?

has a compatible solution or not without sending any messages to P.

Note that there is a tradeoff between the use of parallelism and the use of a global view of the problem; when we tested algorithm 1 without constraint propagation (no parallelism or global view) it performed so badly that we opted to continue testing it only with the use of constraint propagation (global view); this resulted in amounts of MNCCs similar to those obtained from algorithms 3 and 4 (but with a higher communications overhead) which used parallelism but no global view. The amount of MNCCs performed by algorithms 3 and 4 can be easily reduced by the use of constraint propagation inside each component.

Discussion and Conclusions

We defined in this paper the concept of DCSP - Distributed Constraint Satisfaction Problem. The DCSP is an extension of the CSP concept. Previous works used distributed techniques for solving CSPs, but ignored the distribution as an inherent part of the problem. Our approach is distinct in that not only the distribution is a way to solve the problem but also is part of the problem definition. We believe that this will lead to a better definition an understanding of DCSPs.

The algorithms presented in this paper for solving DCSPs focus on the methods that the agents use for solving external constraints. However, little requirements are posed on how the agents solve the internal constraints. This feature enables each agent to use a different technique for solving the CSP for which he is responsible. Moreover, an agent may use a non CSPs' representation and solving technique, like an Operation Research or Knowledge Based method, for solving its internal CSP. Since the different methodologies often are suited for different CSPs, then using different techniques for different components of a DCSP may be very useful.

We proved that if the internal algorithms used by all the agents are sound and complete, then all the algorithms presented in this paper will have the same property. We proved it using the naive interpretation that enables looking at DCSP as a CSP whose nodes are the internal CSPs of the DCSP. Several other properties can be proven using this **naive interpretation**. For example, the property, "a DCSP whose components in the naive interpretation form a tree, can be solved without performing any backtrack between its' CSPs components", immediately follows from a similar proof for CSPs (Freuder E.C. 1982).

In future research we will apply the above algorithms to a real life distributed resource allocation problem such as in (Sycara K. *et al.* 1991) and report on their performance in this environment.

References

Burke P., and Prosser P. 1989. A distributed asynchronous system for predictive and reactive scheduling. Technical Report AISL-44, University of Strathclyde.

Collin Z., and Dechter R. 1990. A distributed solution to the network consistency problem. *Methodologies for Intelligent Systems* 242–251.

Dechter R., and Pearl J. 1987. Network-based heuristics for constraint satisfaction problems. *The AI Jou.* 34(1):1 37.

Feldman R., and Golumbic M. 1990. Interactive scheduling as a constraint labeling problem. *Annals of Mathematics and Artificial Intelligence* (1):49–73.

Freuder E.C. 1982. A sufficient condition for backtrack-free search. *Journal of the ACM* 29(4).

Kasif S., and Delcher A.L. 1990. Analysis of local consistency in parallel constraint-satisfaction networks. *The AI Jou.* (13).

Kumar V. . 1992. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine* 13:32–44.

Levine P., and Pomerol J. 1990. Railcar distribution at the french railways. *IEEE Expert* October:61–69.

Prosser P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* 9:268–299.

Prosser P. 1994. Binary constraint satisfaction problems: some are harder than others. In *Proceedings of the 11th European Conference on Artificial Intelligence*, 95 99.

Prosser P.and Conway C., and Muller M. 1992. A constraint maintenance system for the distributed resource allocation problem. *Intelligent Systems Engineering* 76 83.

Solotorevsky G., and Gudes E. 1995. Distributed constraint satisfaction problems - definitions and algorithms. Dept. of Mathematics and Computer Science Technical Report FC-9503, Ben-Gurion University, Israel.

Solotorevsky G.; Gudes E.; and Meisels A. 1994. Raps - a rule-nased language for specifying resource allocation and time-tabling problems. *IEEE Trans. DKE* 6:681 698.

Sycara K.; Roth F.; Sadeh N.; and Fox M. 1991. Resource allocation in distributed factory scheduling. *IEEE Expert* 29–40.

Yokoo M.; Durfee E.; Ishida T; and Kuwabara K. 1992. Distributed constraint satisfaction for formalizing distributed problem solving. In *IEEE Intern. Conf. Distrb. Comp. Sys.*, 614 – 621.

Yokoo M. et. al. 1990. Distributed constraint satisfaction for dai problems. In *10th Intern. Workshop on DAI.*

Yokoo M. 1995. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proc. 1st Intrnat. Conf. on Const. Progr.*, 88 – 102.