A Hybrid Solving Scheme for Distributed Constraint Satisfaction Problems

Carlos Eisenberg and **Boi Faltings**
Artificial Intelligence Laboratory (LIA)
Swiss Federal Institute of Technology (EPFL)
IN-Ecublens, 1015 Lausanne, Switzerland
{eisenberg, faltings}@lia.di.epfl.ch
http://liawww.epfl.ch/

Abstract

Algorithms for Distributed Constraint Satisfaction Problems fall into two categories: distributed complete search (DisCS) and distributed local search (DisLS). The advantage of DisCS is that it guarantees to find a solution or proves problem infeasibility. DisCS is good at solving tight problems with complex constraints, but suffers from false commitments early in the search tree, leading to exhaustive search and redundant communication when exploring infeasible subproblem spaces. DisLS in comparison does not commit to assignments and is excellent at solving large problems with constraints of local scope. DisLS however, is incomplete and has difficulties to deal with tightly-/ overconstrained problems and complex constraints, where the algorithm often gets caught in infinite cycles without finding a solution or proving problem infeasibility. In this paper we combine a DisLS algorithm with a DisCS algorithm, and profit from their complementary advantages. We present a new, hybrid search scheme and show that it outperforms both methods.

Introduction

Algorithms for Distributed Constraint Satisfaction Problems fall into two categories: distributed complete search (DisCS) and distributed local search (DisLS). The advantage of DisCS is that it is complete and guarantees to find a solution if one exists or can prove problem infeasibility. DisCS is usually good at solving tightly and overconstrained problems with complex constraints. Therefore a good strategy is always to start with the hardest part of the problem first. However, a big disadvantage of DisCS are hard variable value commitments, especially those that are made at the first nodes in the search tree. If these early commitments are dead ends, it invariably leads to exhaustive search and communication traffic when exploring infeasible subproblem spaces. DisLS in comparison, does not commit to assignments and is excellent at solving large scale problems with constraints of local scope. The disadvantage of DisLS however, is incompleteness and difficulties to deal with tightly and overconstrained problems as well as complex constraints. Under these conditions, the DisLS can easily get caught in infinite cycles without finding a solution or proving problem failure. As both search methods have complementary properties they lend themselves to be used in combination.

The breakout algorithm (BO) is a very successful, local search technique. Its origins go back to the work of (Minton et al. 1992) and (Morris 1993). Minton et. al. developed in 1992 a min-conflict heuristic, which iteratively repairs a given complete value assignment with the goal to minimize the conflicts (constraint violations). Minton et. al demonstrated the success of this technique by solving large scale scheduling problems and showed that it could outperform a traditional backtracking technique by several orders of magnitude. However, one major drawback of this method remained. The method could get stuck in local, non-solution minima, requiring expensive restarts with new initial assignments. However, Morris could eliminate this drawback and extended the heuristic by a breakout method. This method allowed the algorithm to escape from local non solution minima by assigning a weight to each constraint and when the algorithm is in a local non solution minimum to increase the weights of violated constraints. When the weights increased, the constraint conflicts also increased, and the algorithm can break out of the local minimum. Besides using the weights for escaping from local minima they can also be useful for localizing hard and unsolvable sub problems. Constraints with a relatively high weight, belong most likely to a hard or unsolvable subproblem. Hence, the constraint weights are also useful for establishing a fail first variable order and can therefore guide a systematic search method, such as backtracking.

In this paper we will present a new, hybrid search scheme for solving DisCSPs, where we combine the distributed breakout algorithm (DisBO), with the distributed backtracking algorithm (DisBT). We couple the two algorithms by a fail first variable order, which we extract from the DisBO algorithm and feed to DisBT. In this variable order, we sort the variables of potentially unsolvable subproblems at the top of the list.

By solving a large set of scheduling problems, we show that the proposed hybrid scheme works extremely well and outperforms the two individual methods by several orders of magnitude.

The rest of the paper is organized as follows. In section 2, we give definitions, discuss the properties of unsolvable sub-

problems and briefly recall the execution of the distributed breakout and distributed backtracking algorithm. In Section 3 we will present a solving scheme and propose a new hybrid algorithm:

- DisBOBT (Breakout with Backtracking) as a complete mixed algorithm for solving CSPs and identifying an unsolvable subproblem if it exists

In Section 4, we show the results of the experiments where we applied the DisBOBT algorithm to solve randomly generated scheduling problems. In Section 5, we briefly discuss algorithm improvements and variants. In section 6 we draw our conclusions.

Properties of the Breakout Algorithm

Definition 1 (Constraint Satisfaction Problem (CSP) P). *A finite, binary constraint satisfaction problem is a tuple $P =< X, D, C >$ where:*

- $X = \{x_1, .., x_n\}$ *is a set of n variables,*
- $D = \{d_1(x_1), .., d_n(x_n)\}$ *is a set of n domains, and*
- $C = \{c_1, .., c_p\}$ *is a set of p constraints, where each constraint $c_l(x_i, x_j)$ involves two variables x_i and x_j and is a function from the Cartesian product $d_i(x_i) \times d_j(x_j)$ to $\{0, 1\}$ that returns 0 whenever the value combination for x_i and x_j is allowed, and 1 otherwise (note that this is opposite from the classical formulation to simplify our formulas). We call the set $\{x_i, x_j\}$ $vars(c)$ and there is at most one constraint with the same set of variables.*

Definition 2 (Subproblem P_k). *A* subproblem *P_k of a problem P with k variables is defined as a tuple $P_k =< X_{P_k} \subseteq X, D_{P_k} \subseteq D, C_{P_k} \subseteq C >$ with the additional property that C_{P_k} contains all and only constraints between variables in X_{P_k}. We define the* size *of a subproblem as the number of constraints $|C_{P_k}|$.*

Definition 3 (Unsolvable Subproblems usp). *A subproblem P_k is* unsolvable *if there is no value assignment to variables in X_{P_k} that satisfies all constraints in C_{P_k}.*

An unsolvable subproblem P_k is minimal *if it becomes solvable by removing any one of its variables.*

A minimal unsolvable subproblem P_k is a smallest *unsolvable subproblem of P, if there is not another minimal unsolvable subproblem P'_l such that $size(P'_l) < size(P_k)$.*

Definition 4 (Distributed Constraint Satisfaction Problem (DisCSP) P_{dis}). *A distributed constraint satisfaction problem P_{dis} is a Constraint Satisfaction Problem where the variables, domains and constraints are distributed amongst a set of s agents. $\mathcal{A} = \{a_1, .., a_s\}$.*

Solving a CSP and DisCSP is equivalent to finding a value assignment for all variables, that simultaneously satisfies all the constraints in $\mathcal{C}$.

In our model each agent owns multiple variables, and distinguishes between private and public variables.

Definition 5 (Private and Public Variables).

- *Private variables are only visible to the owner agent and are not part of any public constraint. $\forall x_i \in \mathcal{X}_{priv}(\neg\exists c_j(x_k, x_l) \in \mathcal{C}_{pub} \wedge (x_i = x_k \vee x_i = x_l))$.*
- *Public variables are visible to all agents and are part of one or more public constraints. $\forall x_i \in \mathcal{X}_{pub}(\exists c_i(x_j, x_k) \in C_{pub} \wedge (x_i = x_j \vee x_i = x_k))$.*

In this context we also define the function $Owner(x_i)$, which returns the owner agent of x_i.

Note that we assume for the CSP problem graphs that they are at least connected and do not contain independent subproblems.

Definition 6 (Public and Private Constraints).

- *The private constraint set $\mathcal{C}_{priv} \subseteq \mathcal{C}$ includes all agent internal constraints, constraints that exclusively refer to variables that are owned by the same agent. $\forall c_i(x_j, x_k) \in \mathcal{C}_{priv}(Owner(x_j) = Owner(x_k))$.*
- *The public constraint set $\mathcal{C}_{pub} \subseteq \mathcal{C}$ includes all inter agent constraints. Inter agent constraints refer to variables that are owned by two or more agents. $\forall c_i(x_j, x_k) \in \mathcal{C}_{pub}(Owner(x_j) \neq Owner(x_k))$.*

Identifying Unsolvable Subproblems with the Breakout Algorithm

The breakout algorithm (Morris 1993) is a further development of the min-conflict algorithm (Minton et al. 1992) and is the basis for our work.

```
1: function breakout(CSP, cycle − limit, breakout − limit)
2: S ← random initial state
3: W ← vector of all 1
4: while S is not a solution ∧(cycle−limit > 0)∧(breakout−
   limit > 0) do
5:    if S is not a local minimum then
6:       make local change to minimize conflicts
7:       cycle − limit ← cycle − limit − 1
8:    else
9:       increase the weight of all currently violated constraints
10:      breakout − limit ← breakout − limit − 1
11: return(S, W)
```

Algorithm 1: Breakout algorithm.

Algorithm 1 shows the basic breakout algorithm. The state $S =< x_1 = v(x_1, S), ..., x_n = v(x_n, S) >$ is an assignment of values to all variables of the problem. It can be a solution when no constraint is violated, otherwise it has a number of conflicts with constraints. The breakout algorithm contains two essential steps: determining the local change that minimizes conflicts, and increasing the weights (called the breakout).

With every constraint, we associate a weight:

Definition 7 (Constraint weight w). *Each constraint is assigned a weight $w(c(x_i, x_j))$ or in short $w_{i,j}$. All weights are positive integer numbers and are set to 1 initially. The breakout algorithm uses the weights in order to escape from local non- solution minima.*

In Algorithm 1, the weights are grouped together in the weight vector W.

Conflict minimization consists of choosing a variable and a new value that reduces as much as possible the conflicts in the current state. For this, we compute for every variable its conflict value, defined as follows:

Definition 8 (Variable conflict value ω). *The conflict value $\omega(x_i, v_a, S)$ of variable x_i assigned the value v_a in state S, is the sum of weights of the constraints involving x_i that would be violated in a state S' that differs from S only in that $x_i = v_a$:*

$$\omega(x_i, v_a, S) = \sum_{c_l(x_i, x_j)} w(c_l) \cdot c_l(x_i = v_a, x_j = v(x_j, S))$$

where $v(x_j, S)$ is the value assigned to variable x_j in state S.

The best improvement is to the variable/value combination x_i, v_a such that $\omega(x_i, v(x_i, S), S) - \omega(x_i, v_a, S)$ is largest. If there is such a combination with an improvement greater than 0, the variable/value combination with the best improvement is chosen as the local improvement.

If no improvement is possible, the algorithm is in a local minimum. In this case, the algorithm increases the weight of each violated constraint by 1, and again attempts to compute the possible improvements. Increasing the weights of each violated constraint is what we term a *breakout step*. Since the current violations will gain more weight, eventually an improvement in the conflict value will be possible.

In general, one imposes a runtime limit on the algorithm: there can be a limit on the number of *iterations (cycles)*, i.e. the number of times variables are revised, or, on the number of *breakout steps*.

For the breakout algorithm, we can observe the following:

Lemma 1. *After m breakout steps, the sum of the constraint weights $w_{sum} = \sum_{c(x_i,x_j) \in C_{P_k}} w_{i,j}$ of an unsolvable subproblem P_k with $|C_{P_k}| = q$ constraints must be greater than or equal to $m + q$.*

Proof. If a subproblem is unsolvable, then in every breakout step, one or more of the subproblem constraints must be violated and the corresponding constraint weight is increased. The lower bound for w_{sum} can be derived by assuming that in every iteration only one constraint is violated, in this case the weight sum must be equal to $m + q$. □

Based on Lemma 1, we define:

Definition 9 (Weight sum condition for subproblem P_k). *We say that a subproblem P_k satisfies the weight sum condition if and only if after m iterations of the breakout algorithm, the condition of Lemma 1:*

$$\sum_{i=1}^{q} w(c_i) \geq m + q$$

is satisfied, where $c_i = c(x_s, x_t)$ are all the constraints of the constraint set C_{P_k} of the subproblem P_k, and $q = |C_{P_k}|$.

The weight sum condition is a powerful tool for searching unsolvable subproblems since by Lemma 1, any unsolvable subproblem must satisfy it:

Lemma 2. *After m breakout steps, an unsolvable subproblem with q constraints must satisfy the weight sum condition.*

Proof. The condition is ensured by Lemma 1. □

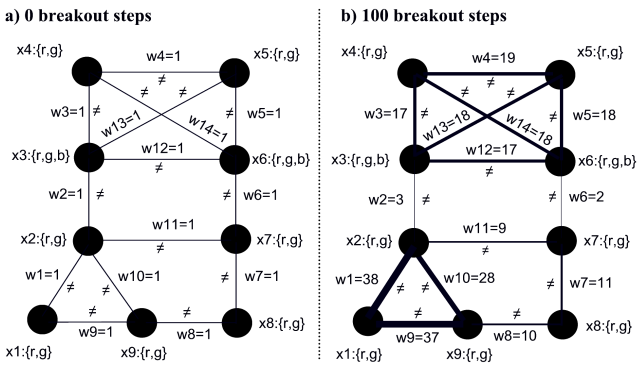

Figure 1: The constraint weight graph of an unsolvable problem containing three unsolvable sub-problems of size 3 (x_1, x_2, x_9), 4 (x_3, x_4, x_5, x_6) and 5 (x_1, x_2, x_7, x_8, x_9), after 0 and 100 breakout steps.

Thus, if after m iterations the breakout algorithm has not found a solution, and we suspect that the problem contains an unsolvable subproblem with 3 constraints, then we only have to consider subproblems whose weight sum is at least $m + 3$. If we apply this constraint in the problem of Figure 1, we find that the constraints of w1, w9, w10, whose sum is 103, are the only three constraints that satisfy the sum constraint and indeed describe an unsolvable subproblem of size 3: colouring a graph of 3 nodes with only 2 colours.

When applying the breakout algorithm to small problems that are entirely unsolvable, the condition can be already applied after a small number of breakout iterations. When unsolvable subproblems are embedded in a larger structure, as shown in Figure 1, there will also be many subproblems that satisfy the weight sum condition by accident. In this case, we may need to run the breakout algorithm for a certain minimum number of cycles before the unsolvable subproblem can be reliably identified.

Considering a randomly chosen individual constraint c, we can measure the probability that after m breakout steps c is violated in a breakout step as:

$$p(c = violated) = \frac{\sum_{c_l \in C} w(c_l) - 1}{m|C|} \quad (1)$$

When solving the problem, this probability will decrease during the first BO steps, since BO progressively eliminates conflicts. If the problem is solvable for the BO, then the probability eventually becomes 0. Otherwise, it will stabilize and converge towards a constant value. If this is the case and BO cannot solve the problem due to a hard or unsolvable subproblem P of size q, the constraints that belong to the unsolvable subproblem are identified by the fact that their probability of being violated is at least equal to $1/q$. Thus, the expected difference in weight between a constraint that is in the unsolvable subproblem and one that is not is[1]:

$$\delta = (1/q - p(c = violated)) \cdot m \quad (2)$$

[1]This ignores the fact that the constraints in the unsolvable subproblem itself increase the probability of constraint violations, so it is overly pessimistic.

As constraints belonging to the unsolvable subproblem can be identified only when their weights differ from the others by at least 1, we propose as a reasonable heuristic for choosing the number of breakout iterations for identifying subproblems of size q as:

$$m(q) \geq \frac{1}{1/q - p(c = violated)} \quad (3)$$

which means that the expected difference in weight is at least 1. When higher accuracy is desired, we can of course choose a higher expected weight difference and thus a larger number of iterations.

For example, in Figure 1, the total weight of the 14 constraints after $m = 100$ breakout iterations is 245, so that the probability:

$$p(c = violated) = 231/1400 \simeq 0.165$$

Thus, in this problem we could identify a subproblem of size 3 after approximately $1/(1/3 - 0.165) = 1/0.16833 \leq 6$ iterations, while for a subproblem of size 5 we would need about 30 iterations, and a subproblem of size 7 could not be identified with this reliability at all since $p(c = violated)$ is larger than $1/7$.

Note that due to equation 3 this method will work very well when p(c=violated) and q are small. In this case, the minimum number of required iterations m becomes small. This means that it is always easier to identify an unsolvable subproblem of size q than a larger one of size $q' > q$. Also, it is always easier to identify an unsolvable subproblem when the average number of constraint violations in a breakout step is small. These conditions are not unrealistic. In practice, problems are usually not excessively overconstrained and often contain only a few flaws of small size. With our method such flaws are easily identified and help to repair the problem. This method is therefore particularly well suited to deal with situations where there are small unsolvable subproblems. We now concentrate on developing a fail first variable order from the constraint weights for guiding a systematic search process such as backtracking.

Since high constraint weights indicate unsolvable or hard subproblems of small size, variables that are connected by these constraints, must be sorted at the top of the variable list so that a systematic search algorithm either fails early, or solves the hardest subproblem first. Besides the constraint weights the graph structure must be also considered. It is possible, that the highest constraint weights belong to different hard or unsolvable subproblems. We therefore order the variables in such a way that the next variable is always the variable, where the sum of the constraint weights of the constraints that connect this variable with the variable in the sorted variable list is the highest. This constraint weight directed, fail-first variable ordering heuristic is given in the following greedy algorithm:

Function **weight-ordering**(X, C) first assigns the empty set to the sorted variable set X_s. The function argmax in line 3 then takes as input the set of constraints and returns from these the one with highest weight. Then the two variables that are connected by this constraint are added to X_s. Then the function enters the main loop where, variable by variable is selected and added to X_s. It selects the next variable from the variables that are not yet selected and where the constraint weight sum of the constraints between that particular variable and variables in X_s, is the highest.

```
1: Function weight-ordering(X, C)
2: X_s ← {};
3: c_max(x_i, x_j) ← argmax_{c∈C}(w(c(x_i, x_j)));
4: X_s ← {x_i, x_j};
5: while |X_s| < |X| do
6:    X_p ← X \ X_s; max_sum ← 0;
7:    for all x_i ∈ X_p do
8:       sum ← 0;
9:       for all c(x_i, x_j) ∈ C ∧ x_j ∈ X_s do
10:         sum ← sum + w(c(x_i, x_j));
11:         if sum > max_sum then
12:            x_next ← x_i; max_sum ← sum;
13:   X_s ← X_s ∪ {x_next}
14: return X_s
```

Algorithm 2: Constraint weight variable order heuristic weight-ordering: orders the variables with respect to the highest constraint weight sum and the graph structure.

The Distributed Solving Scheme

The observed properties are not only useful for developing new central search methods, but equally for distributed methods. We are now going to implement a distributed version of the simple hybrid algorithm that we described in the last chapter.

The architecture of the distributed hybrid algorithm is as follows. We first try to solve the problem with the Distributed Breakout Algorithm (DisBO) and abort if no solution is available after exceeding the maximum number of cycles. Then, the agent, who is the owner of the constraint with the highest constraint weight starts a synchronous distributed backtracking (DisBT) process. The variable order of the DisBT process is equivalent to the one that we described in the central variable ordering function (Algorithm 2). However, in order to save messages when the problem fails already after the first few variables, the entire variable order is not determined beforehand, but incrementally, during the DisBT execution; every time the partial solution needs to be extended by a new variable, only then the next variable is selected. The DisBT process continues until it proves that the problem is infeasible or until a solution is found; therefore the distributed hybrid algorithm is complete.

Distributed Breakout Algorithm (DisBO)

The basis of our distributed hybrid algorithm is the distributed breakout algorithm (DisBO), see Yokoo et. al. and also Zhang et. al. (Yokoo 2001), (Yokoo et. al. 1996), (Zhang et. al. 2002). DisBO is described in two versions. The first version operates with quasi local minima, where an agent increases its constraint weights already, when one or more of its constraints are violated, and the possible improvements of all its variables as well as of the neighbour variables is 0. However, as Yokoo et. al. already point out, quasi local minima cannot guarantee a global minimum

and lead to the situation where agents increase their constraint weights too 'early' and therefore add 'noise' to the constraint weight system.

In the second version, called distributed breakout algorithm with broadcasting, the agents broadcast to all agents when they are in a quasi local minimum, and increase their weights only when all agents are in a quasi local minimum, which is then a real local minimum.

Since our variable ordering scheme is sensitive to 'noise' imposed upon the constraint weights, and also because 'noise' adversely affects the performance for solving dense problems, we implement a DisBO version, where we allow an increase in weights only in real minimum states. However, in order to reduce the message traffic caused by expensive broadcasting of quasi local minima states, we suggest to use instead a general synchronisation mechanism (see termination detection mechanism in (Yokoo 2001)). This mechanism we can then use for detecting a solution, a real local minima, and also the highest constraint weight of the problem.

We now briefly describe the DisBO algorithm in terms of the steps executed in every cycle:

Step 0: (Initialization). The agent assigns to each variable that he owns, $x_i \in X_{my} \wedge X_{my} \leftarrow \{x_i | x_i \in X \wedge owner(x_i) = self\}$ a value that is randomly chosen from its domain $d(x_i)$. Then he updates all neighbours that have a constraint with any of his public variables $x_i \in X_{pub}$ on the corresponding assignment $d(x_i)$. Then all constraint weights $w(c(x_i, x_j))$ are set to 1 and the cycle counter is initialised to 1.

Step 1: (Local Variable Revision). The agent revises all private variables $x_i \in X_{priv}$ that violate a constraint and assigns to them the first domain value that minimizes the conflict. If no such value exists, the assignment remains unchanged. The revision process continues until no more improvement is possible.

Step 2: (Determination of Variable Proposals). The agent considers its public variables that violate a constraint $x_i \in X_{pub}$ and determines for each the domain value that minimize the conflict value. The new assignment proposal together with the conflict reduction value, is then sent as a proposal to every neighbour agent (remark: two agents are neighbours if they share a constraint).

Step 3: (Evaluation of Variable Proposals). When the agent has received the proposals of all his neighbour agents, it compares all proposals and determine the winner proposal(s). The winner proposal(s) is the one with the greatest conflict reduction value. If two proposals that refer to neighbouring variables win, then the proposal, belonging to the lexicographic smaller agent, wins.

Step 4: (Variable Update). If the agent is the owner of a winner proposal, it updates the associated variable and updates the corresponding neighbours that have a constraint with the updated variable.

Step 5: (State Detection). At the end of each cycle the agent must detect the assignment state with its neighbours in order to proceed. This state detection method is described in the section 'Global State Detection Method'. Based on the detected state the agent will then either terminate, continue with the next cycle or increase the weights of violated constraints.The following states and consequent execution steps occur:

- **Solution.** The variable assignment is a *solution*and the agents terminate the algorithm and return the solution.
- **Local minimum.** When the variable assignment is in a local minimum, the agents increase all constraint weights by 1.
- **Maximal number of cycles.** If $mcyc > 0$, the agent counts down the cycle counter $mcyc$ by 1 and continues with DisBO by branching back to step 1. When $mcyc$ is equal to 1, the agent aborts the execution of DisBO and, if he is the owner of the constraint with the highest weight (see variable ordering scheme), he starts the DisBT process. In order to avoid an additional search process for this constraint, the agent includes during the last cycle, when $mcyc = 1$, his maximum constraint weight mw in the state detection messages. This ensures that the global highest constraint weight propagates, and hence determines the agent that starts DisBT.

Global State Detection Method

The global state detection method is used in the DisBO algorithm in order to detect two assignment states, a solution or a local minimum. This method is described by Yokoo et.al. see (Yokoo 2001) where it is used for detecting algorithm termination. In their example, an agent keeps a termination counter and updates it according to the following two rules:

1. If a constraint is violated the termination counter is set to 0, otherwise to 1. Then the counter value is sent to all neighbours.
2. When termination counter values are received from all neighbours, the termination counter is updated by the lowest termination counter value. If the new termination counter is greater than 0, the counter is increased by 1. Then the termination counter value is sent to all neighbour agents.

By inductive proof, one can show that when an agent's termination counter becomes d_{max}, which is equal to the shortest link distance between two agents that are furthest apart within the network (see Figure 2), that the termination counter value has fully propagated and that no agent within the distance d_{max} can have a termination counter equal to 0, or in other words, a constraint violation. Note that we assume that all agents know the value of d_{max}.

Besides detecting the algorithm termination, the state detection method is also used for detecting a real local minimum, where the value 0 represents the state in which the agent is not in a local minimum.

For the procedure **StateDetection**($mcyc$), which also starts the **SyncDisBT** function, an agent keeps the following information: $mcyc$ - cycle counter, tc - termination counter, lc - local minimum counter, sdc - state detection counter,

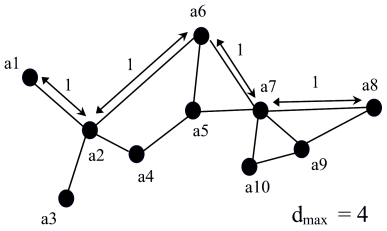

Figure 2: The distance d between two agents is defined as the shortest link distance between them. In a finite network of agents d_{max} is then defined as the greatest link distance d of two agents within the network. In the above agent network d_{max} is equal to 4, which is for example the shortest path between a_1 and a_8.

d_{max} - the shortest distance between the two agents that are furthest apart, c_{max} - the local constraint with the highest weight, mw - the maximum weight value and $mwref$ - the reference to the owner of mw (e.g. agent name) as second selection criteria when two constraints have the highest weight. The function myvar($c(x_i, x_j)$) returns to the agent the variable he owns, either x_i or x_j. The **state** messages contain the two counter values tc, lc and the tuple (mw, $mwref$) referring to the maximum weight and reference to the owner of that weight.

1: Procedure **StateDetection**($mcyc$)
2: $sdc \leftarrow 1$;$my\_lc \leftarrow 1$;$my\_tc \leftarrow 1$;
3: **if** $mcyc = 0 \wedge my\_mwref = self$ **then**
4: $x_1 \leftarrow myvar(c_{max}(x_i, x_j))$;
5: call **SyncDisBT**($x_1, \{\}$) and **exit** procedure;
6: **if** any $c_i \in C$ is violated **then** $my\_tc \leftarrow 0$;
7: **if** improvement during cycle **then** $my\_lc \leftarrow 0$
8: $c_{max} \leftarrow argmax_{c_i \in \{C_{pub} \cup C_{priv}\}}(w(c_i))$;
9: $my\_mw \leftarrow w(c_{max})$; $my\_mwref \leftarrow self$;
10: send **state** to all neighbours
11: $waitForState \leftarrow TRUE$;
12: **while** $waitForState$ **do**
13: **if** received (**state**, $tc, lc, (mw, mwref)$) **then**
14: $my\_tc \leftarrow min(tc, my\_tc)$; $my\_lc \leftarrow min(lc, my\_lc)$;
15: $(my\_mw, my\_mwref) \leftarrow$
$max((mw, mwref), (my\_mw, my\_mwref))$
16: **if** $my\_tc = 0$ **and** $my\_lc = 0$ **and** $mcyc > 1$ **then**
17: send **state** to all neighbours
18: $waitForState \leftarrow FALSE$;
19: **else**
20: **if** received a **state** message from all neighbours **then**
21: **if** $my\_tc = d_{max}$ **then**
22: terminate with solution
23: **if** $my\_lc = d_{max}$ **then**
24: increase weights of violated constraints
25: **if** $sdc = d_{max}$ **then**
26: $waitForState \leftarrow FALSE$;
27: **else**
28: send **state** message to all neighbours
29: $my\_tc \leftarrow my\_tc + 1$; $my\_lc \leftarrow my\_lc + 1$;
30: $sdc \leftarrow sdc + 1$

Distributed Backtrack Algorithm (DisBT) with constraint weight based variable ordering

The distributed systematic search algorithm is based on the synchronous distributed backtrack search algorithm (DisBT) from Yokoo et.al. (Yokoo 2001) and extended by a distributed constraint weight based variable ordering heuristic. For this algorithm we assume the same procedures, functions and messages as they are described for DisBT in (Yokoo 2001) and (Yokoo et. al. 1998).

The standard DisBT algorithm starts with a fixed variable order that the agents agree on before starting the backtrack process. However, for saving messages, we do not determine the entire variable order in advance, but order variables incrementally, every time a partial solution needs to be extended by a new variable. We select the new variable according to the same order rule used in the variable order function $weight - ordering(X, C)$ (Algorithm 2).

Function **SyncDisBT** carries out synchronous backtracking with an incremental addition of the variable, such that the sum of the weights of constraints leading back to earlier variables is the highest.

The partial assignment P to variables $x_1..x_{k-1}$ is passed to the agent responsible for x_k. It first gathers all values that are compatible with this partial assignment, and tests whether it has solved the entire problem. If it is the last variable and has found a consistent solution so far, then it calls function $FindVar$ to add the next variable; if there is no next variable it returns success. It then carries out a backtrack search with either the new or the next variable. If values are exhausted without success, the algorithm backtracks by returning failure. If backtracking reaches the first node, then the problem has been shown unsolvable.

For each variable x_i in the search, the owner agent keeps the following information:

- $Pred(x_i)$: predecessor variable in search order
- $Succ(x_i)$: successor variable in search order
- mw: value of the maximum sum of weights of a neighbour variable back into the problem

1: Function **SyncDisBT**(x_i, P)
2: $vals \leftarrow \{v | v \in d_i(x_i)$ such that no constraints with assignments in P are violated$\}$
3: **if** $(Succ(x_i) = NIL$ **and** $vals \neq \{\})$ **then**
4: $x_{next} \leftarrow FindVar(x_i, 0, NIL)$
5: **if** $x_{next} = NIL$ **then**
6: $x_i \leftarrow next(vals)$;
7: return $success$ to $Owner(Pred(x_i))$ and terminate
8: **else**
9: $AddVar(x_i, x_{next})$
10: **while** $vals$ has next **do**
11: $x_i \leftarrow next(vals)$;
12: invoke $Owner(Succ(x_i))$:
$r \leftarrow SyncDisBT(Succ(x_i), P \cup x_i)$
13: **if** $r = success$ **then**
14: return $success$ to $Owner(Pred(x_i))$ and terminate
15: **if** $Pred(x_i) = NIL$ **then**
16: inform all agents problem is unsolvable
17: **else**
18: return $failure$

Additionally, for every variable x_j not in the search process, its agent keeps the list of its neighbours that are part of the search process.

FindVar is a function that finds the neighbour with maximum weight sum and chains further back in the search tree. When the first node is reached, the maximum is found and the corresponding variable is handed back down to the last one.

1: Function **FindVar**(x_i,mw,x_{next})
2: **for** $x_n \in neighbours(x_i)$ **do**
3: invoke $Owner(x_n)$: $ws \leftarrow SumWeights(x_n)$
4: **if** $ws > mw$ **then**
5: $x_{next} \leftarrow x_n$; $mw \leftarrow ws$
6: **if** $Pred(x_i) = NIL$ **then**
7: return x_{next}
8: **else**
9: invoke $Owner(Pred(x_i))$:
$x_{next} \leftarrow FindVar(Pred(x_i), mw, x_{next})$
10: return x_{next}

AddVar is a procedure that adds variable x_{next} as the last one following x_{last} in the search process. It includes setting certain variables by the owner of x_{last} and others by the owner of x_{next} by message passing.

1: Procedure **AddVar**(x_{last}, x_{next})
2: **for** $x_n \in neighbours(x)$ **do**
3: inform $Owner(x_n)$ that x_{next} is now part of the search;
4: $Succ(x_{last}) \leftarrow x_{next}$; $Pred(x_{next}) \leftarrow x_{last}$
5: $Succ(x_{next}) \leftarrow NIL$;

SumWeights sums up the weights of the constraints going from variable x_i back to variables already in the search process:

1: Function **SumWeights**(x_i)
2: **if** Owner(x_i) is not part of the search process **then**
3: return sum of weights of constraints with neighbours in search process
4: **else**
5: return 0

Experiments and Results

For evaluating the hybrid algorithm DisBOBT we solve a large set of 1000 randomly generated scheduling problems and compare its performance in terms of exchanged messages with that of DisBO. We do not compare DisBOBT to DisBT, as DisBT requires unacceptable long execution times for solving the problems.

The scheduling problems are generated according to the KRFP (kernel resource feasibility problem) model, see (El Sakkout 2000), and are described by the following items:

- a schedule horizon: a project start and end date
- a set of tasks: each task has a variable start/ finish date and a fixed duration
- a set of precedence constraints: each precedence constraint links two tasks and determines their execution sequence.

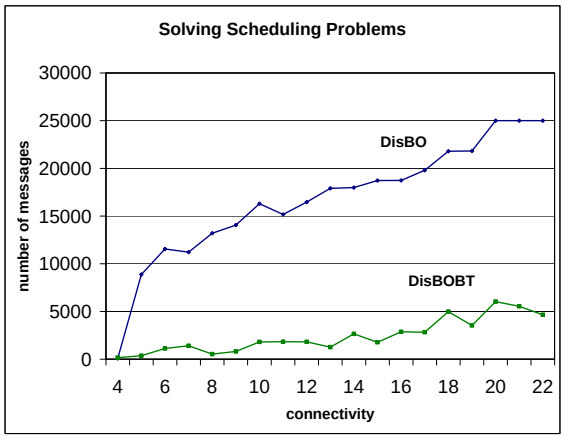

Figure 3: 1000 randomly generated scheduling problems solved with DisBO and DisBOBT.

- set of deadlines.
- a set of resource constraints: each project has a set of unary and discrete resources.

For the experiments, we generate schedules of different tightness. The tightness is controlled by randomly varying the number of deadlines, precedence and resource constraints. The generated problems have a connectivity between 4-22. The ratio of solvable to unsolvable problems (within the execution bounds) is 1:1. In all experiments the project start and finish date is fixed, and the number of tasks is always 25. The schedules are distributed amongst 5 agents. We limit the number of messages to 25,000 for both algorithms and start with systematic search (DisBT) after a fixed number of breakout steps when BO does not find a solution. By experimenting with different breakout bounds we find that the best compromise between systematic search and local search in terms of total number of constraint checks (DisBO+DisBT) is to start systematic search after 40 breakout steps. If we start systematic search after less breakouts, the number of constraint checks for DisBT goes up and for DisBO down. If we start systematic search after more breakouts the result is the opposite. However, in both cases the total number of constraint checks is always higher.

In the graph we draw the number of messages over the graph connectivity. The phase transition occurs approximately at a connectivity of 18.

The graph shows that DisBOBT clearly outperforms DisBO for all connectivity values by a factor of 10-30.

Algorithm Variants and Future Work

Dynamic Cycle Termination Control

The ability to identify hard and unsolvable subproblems is not dependent on the absolute weight values, but on the weight differences. As soon as constraint weights of an unsolvable subproblem differ from neighbour constraint weights, the unsolvable subproblem can be reliably identified. For the moment we are using a static cycle termination

control, that is optimal for the entire scheduling problem set, but not for each individual scheduling problem. In many cases aborting the algorithm earlier or later, is beneficial and saves a great deal of messages. We have implemented such a dynamic cycle termination control centrally, and improved the algorithm performance by a factor of 30. We are currently working on developing such a dynamic cycle control for the distributed hybrid algorithm.

Parallel Backtrack Search

In the presented algorithm the agents execute a single distributed backtracking process, where the hardest or unsolvable subproblem is sorted to the top of the variable list. However, a problem can contain several unsolvable subproblems, which are distributed within the problem and are quasi independent from each other by not sharing any constraint. For example, think of a project schedule, where independent resource constrained tasks are constrained by different tight deadlines and each representing a hard subproblem. In this case, only one of the hard sub problems is sorted to the top of the variable list, and if it is solvable, it will take a long time, due to ordering heuristic, before the backtrack search reaches the other hard sub problems. For balancing the search and tackling the solving of such quasi independent hard subproblems, we propose the execution of parallel backtracking processes, where each agent can decide to solve the problem from a different end as soon as he identifies a hard or unsolvable subproblem.

When we run parallel backtracking processes, we have to ensure that two partial solutions do not start to overlap and we solve them redundantly. To prevent this, the agent must monitor the set of labelled variables of the different search processes and terminate a processes, for example, when more than 50% of the variables overlap. Then partial solutions must be merged so as to not waste the search effort.

The idea of implementing parallel backtrack search as distributed constraint satisfaction algorithm is not new. Recently a parallel backtrack search for solving random distributed CSPs was presented by (Meisels et. al. 2002). This algorithm however performs parallel search on interleaving subtrees (see (Meseguer et. al. 1995)), where the search tree is divided into separate subtrees using the possible assignments of the first variable. This parallel search technique unfortunately is not suited to integrate the search of overlapping search spaces, i.e. it does not accommodate the merging of partial solutions.

Distributed Spanning Tree

A great deal of communication messages from the breakout algorithm originates from the state detection method in each cycle. As this method is based on a kind of 'flooding' mechanism, where an agent sends the same message to all his neighbours, the method produces a lot of redundant messages. We propose to replace this method by introducing a fixed max spanning tree communication structure, where each agent has one root agent and can have many sub agents. With such a max spanning tree, redundant messages can be eliminated.

Conclusions

In the first part of the paper we have presented a powerful identification scheme where the constraint weights assigned by the breakout algorithm are used to identify hard or unsolvable subproblems of a CSP. We have shown how this information can be used to identify a very efficient fail-first variable ordering, and thus to combine the breakout algorithm with backtrack search for a highly efficient overall CSP search algorithm.

In the second part of the paper we used the fail-first variable ordering heuristic for developing a hybrid distributed constraint satisfaction algorithm, DisBOBT. This algorithm combines the distributed breakout algorithm DisBO and the synchronous distributed backtracking algorithm DisBT and is complete. In order to couple the two algorithms, we extended DisBT by an incremental variable order function that selects the next variable according to the fail-first variable ordering heuristic.

We compared the performance of DisBOBT with that of DisBO by solving a large set of scheduling problems. Due to the termination guarantee and the efficient fail first variable order heuristic, DisBOBT outperforms DisBO and DisBT for all connectivity regions.

We are convinced that the presented hybrid algorithm and variable order scheme represent a platform for developing more powerful DisCSP algorithms in the future.

References

A. Meisels and R. Zivan: Parallel Backtrack Search on DisCSP," *In Proc. AAMAS-02 Workshop on Distributed Constraint Reasoning*, (2002).

P. Meseguer: Interleaved Depth-First Search," *Proceedings IJCAI-97* pp.1382-7, Japan, (1995).

S. Minton, M. Johnston, A. Philips and P. Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems," *Artificial Intelligence*, Vol. 58, P. 161-205, (1992).

P. Morris: The Breakout Method for Escaping from Local Minima," *Proceedings of AAAI-93*, Washington, DC, pp 40-45, (1993).

H. El Sakkout and M. Wallace: Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling," *Constraints*, Vol. 5/4, pp 359-388, (2000).

M. Yokoo: Distributed Constraint Satisfaction. Springer, (2001)

M. Yokoo, H. Durfee, T. Ishida, K. Kuwabara: The Distributed Constraint Satisfaction Problem: Formalization and Algorithms," *IEEE Transactions on knolwedge and data engineering*, Vol.10, No. 5, Sep. (1998)

Yokoo M. and Hirayama K.: Distributed breakout algorithm for solving distributed constraint satisfaction problems," *Proceeding of the Second International Conference on Multi-Agent Systems*, 401-408, (1996)

Zhang W. and Wittenburg L.: Distributed breakout revisited," *In Proc. 18-th National Conf. on Artificial Intelligence (AAAI-2002)*, Edmonton, Canada, 2002, pp.352-357.