Handbook of Constraint Programming
Edited by F. Rossi, P. van Beek and T. Walsh

Chapter 20

Distributed Constraint Programming

Boi Faltings

Constraint satisfaction and optimization problems often involve multiple participants. For example, producing an automobile involves a supply chain of many companies. Scheduling production, delivery and assembly of the different parts would best be solved as a constraint optimization problem ([35]). A more familiar task for most of us is *meeting scheduling*: arrange a set of meetings with varying participants such that no two meetings involving the same person are scheduled at the same time, while respecting order and deadline constraints ([18, 22]). Another application that has been studied in detail is coordinating a network of distributed sensors ([2]).

Such problems can of course be solved by gathering all constraints and optimization criteria into a single large CSP, and then solving this problem using a centralized algorithm. In practice there are many cases where this is not feasible, because it is impossible to bound the problem to a manageable set of variables.

For example, in meeting scheduling, once two people are planning a common meeting, this meeting is potentially in conflict with many other meetings either of them are planning and whose times are decided in parallel. A centralized solver does not know beforehand which of these potential conflicts will become important, and thus will have to gather information about all of them. Since any two people in the world are connected through on average six degrees, this constraint problem is likely to involve a substantial part of the world's population! In contrast, in a distributed solution, changes need to propagate only when there are conflicts between meetings. As such conflicts can usually be resolved by local adjustments, propagation will be limited and the problem can usually be solved with a reasonable amount of effort. This is a typical application where *distributed* algorithms for constraint satisfaction are attractive.

As an another example, consider a configuration system for vacations that composes information obtained from the internet. Even though there are only a finite number of elements to be composed, the number of information sources that could be considered is unboundedly large. Thus, even with a small number of variables it may be impossible for a constraint solver to know the entire space of admissible values and tuples for variables and constraints. Fortunately, the semantics of CSP allows to prove the validity of a solution

even without knowing the entire problem. The *open constraint satisfaction* formulation addresses this form of distributed CSP.

There are also other reasons why distributed constraint satisfaction may be necessary:

- cost of formalization: when problem solving is centralized, each participant will have to formulate its constraints on all imaginable options beforehand. This may be excessively complex: a part supplier, for example, might have to give all feasible delivery dates and quantities beforehand, requiring it to explicitly plan and evaluate a huge number of different scenarios.

  In contrast, when using open constraint satisfaction, agents are asked to evaluate only a minimal number of constraints. Furthermore, if solving is distributed they can use whatever software they have for this purpose.

- privacy: in a meeting scheduling scenario, the fact that person A is also meeting with person B may be private information that A wants to keep from another person C. When problem solution is centralized, the solver will see all meetings and constraints, and thus gains valuable private information that can easily be leaked or stolen.

  In contrast, a distributed solution can be constructed in such a way that agents only reveal information piecemeal when evaluating constraints, and other agents only see information they are required to see for the solving process.

- dynamicity: the problem may change dynamically in that new agents appear while others disappear. When a centralized solver is used, these changes would have to be managed by the central server, which may not be feasible.

- brittleness: a centralized solver creates a central point of failure that leads to brittleness of the entire system. In contract, when solving is distributed among different agents, it allows load balancing and redundant and thus fault-tolerant computation among different agents, leading to more reliable systems. Also, parallel execution might make the entire process more efficient.

A distributed algorithm involves a considerable amount of message exchange, which means that the overall solution may well be slower than a centralized process. In general, distributed techniques work well only when the problem is sparse and loose, i.e. each variable has constraints with only few other variables, and for each constraint there are many value combinations that satisfy it. When problems are dense and tight, usually the distributed solution requires so much information exchange among the agents that it would be much more efficient to communicate the problem to a central solver.

The topic has been studied for quite some time; Sycara et al. ([35]) has considered heuristics for distributed constrained search, and Dechter ([5]) has considered the feasibility of distributed constraint satisfaction in a network of identical agents. Since then, a considerable range of techniques has been developed.

This chapter gives an overview of the main techniques that have been developed for constraint satisfaction and optimization in distributed settings. It first covers methods for synchronous and asynchronous distributed solving by backtracking, local search and dynamic programming. It then describes methods for open constraint programming and optimization, and finally considers the issues of self-interest and privacy. In the interest of a

coherent description, I unified different algorithms, and thus occasionally present concepts with different names and simplifications. I believe that the algorithms illustrate the main techniques of the field in a concise and coherent manner.

20.1 Definitions

In this section, we formally define two variations to the standard formulation of constraint satisfaction problems. The first, which we call *distributed* constraint satisfaction, formalizes the fact that constraint solving happens under the control of different independent agents. The second, called *open* constraint satisfaction, formalizes the fact that information about the CSP is distributed among different agents.

A distributed constraint satisfaction problem is commonly defined as follows:

Definition 20.1. *A* distributed *constraint satisfaction problem (DisCSP) is a tuple* $< X, D, C, A >$ *where:*

- $X = \{x_1, .., x_n\}$ *is a set of* n *variables.*
- $D = \{d_1, .., d_n\}$ *is a set of* n *domains.*
- $C = \{c_1, .., c_m\}$ *is a set of* m *constraints.*
- $A = \{a_1, .., a_n\}$ *is a set of* n *agents, not necessarily all different.*

The main difference to a classical CSP, as defined in Chapter 1, is that every variable x_i is controlled by a corresponding agent a_i, meaning that this agent sets the variable's value. When an agent controls more than one variable, this would be modelled by a single variable whose values are combinations of values of the original variable. It is further assumed that agent a_i knows x_i's domain d_i and all constraints involving x_i, and that it can reliably communicate with all other agents. The values of n and m need not be known to any agent, thus allowing for problems of unbounded size as mentioned in the introduction. The main challenge is to develop distributed algorithms that solve the CSP by exchanging messages among the agents in A.

As in Chapter 1, each constraint c_j is a pair $< r_{s_j}, s_j >$ where s_j is a tuple of variables and r_{s_j} is a cost function $s_j \Rightarrow \{0, 1\}$ that maps every value combination of s_j into 0 if it is consistent, and 1 if it is not. A *solution* to the DisCSP is an assignment of values to all variables that is consistent for all relations.

A distributed constraint satisfaction problem can be extended to a distributed constraint optimization problem by letting the functions r_{s_j} map to $\Re^+$, representing a cost. A solution to the optimization problem is an assignment of values to all variables such that the sum of the costs of all constraints is minimized. Hard constraints can be incorporated by mapping consistent value combinations to cost 0, and inconsistent ones to cost ∞ (in practice, some very large number). It is possible to extend this formulation to general soft constraints (see Chapter 9) but little work has been done on that.

Note that the assumption that each agent controls a variable and knows all the constraints involving that variable may not be applicable to all situations. For example, in meeting scheduling agents usually do not have unilateral power to fix the time of a meeting, and do not know the constraints of the other participants. Silaghi et al. ([28]) have

proposed an alternative formulation where each agent applies its constraints find an agreement on the value of variables. Their *asynchronous aggregation search* (AAS) algorithm shows how the asynchronous backtracking techniques for distributed search can be modified to accommodate this formulation which can be more appropriate for applications.

The definition of open constraint satisfaction problems is somewhat more complex, since we need to take into account the fact that the problem itself varies. The definition thus follows that given for dynamic constraint satisfaction (Chapter 21), except that here its is the variable domains rather than the set of constraints that vary, and that the domains are monotonically increasing:

Definition 20.2. *An* open constraint satisfaction problem *(OCSP) is a possibly unbounded, partially ordered set* $\{CSP(0), CSP(1), ...\}$ *of constraint satisfaction problems, where CSP(i) is defined by a tuple* $< X, D(i), C >$ *where*

- $X = \{x_1, x_2, ..., x_n\}$ *is a set of* n *variables,*
- $D(i) = \{d_1(i), d_2(i), ..., d_n(i)\}$ *is the set of domains for CSP(i), with* $d_k(0) = \{\}$ *for all* k*, and*
- $C = \{(x_i, x_j), (x_k, x_l), ...\}$ *is a set of* m *binary constraints, given by the pairs of variables they involve and intensional relations between them.*

The set is ordered by the relation $\prec$ *where* $CSP(i) \prec CSP(j)$ *if and only if* $(\forall k \in [1..n])d_k(i) \subseteq d_k(j)$ *and* $(\exists k \in [1..n])d_k(i) \subset d_k(j)$.

An assignment *to an OCSP is a combination of value assignments from the corresponding domains to all variables. An assignment is* consistent *in instance* $CSP(i)$ *if and only if all intensional constraints are satisfied. A* solution *of an OCSP is an assignment that is consistent for some instance* $CSP(i)$ *and any instance* $CSP(j) \succ CSP(i)$.

Open CSP can be extended to open constraint optimization problems (OCOP) by adding a set of cost functions:

- $W(i) = \{w_1(i), w_2(i), ..., w_n(i)\}$ is a set of cost (weight) functions on the domains in D, where $w_i : d_i \rightarrow \Re^+$ gives the cost associated with each value in the domain d_i.

Note that in order to not require all costs to be known, they are associated with variable domains rather than constraints. Note also that the two variants can be combined, i.e. it is possible to have an open CSP where control of the variables is distributed among different agents.

In this chapter, we consider only unary and binary constraints. Most algorithms for DisCSP can be generalized to n-ary constraints.

20.2 Distributed Search

20.2.1 Synchronous Backtracking

The simplest algorithm for solving constraint satisfaction problems is backtrack search. In backtrack search, a partial assignment of values to a subset of variables $\{x_1, .., x_k\}$ is

iteratively extended by adding an assignment to another variable x_{k+1} such that all constraints with already assigned variables are satisfied. When no such extension is possible, the algorithm backtracks and changes one or more of the earlier assignments.

The backtrack search algorithm can be readily extended to a distributed algorithm by passing the partial assignments from agent to agent. Thus, agent a_k passes the partial assignment $\{x_1 = v_1, .., x_k = v_k\}$ to a_{k+1} who adds a consistent assignment for x_{k+1}, if possible, or otherwise returns a message to a_k signalling the need to backtrack. This is basically a centralized backtrack algorithm where the thread of control passes to different agents during the execution. Such an algorithm is described for example in [40].

Most of the well-known search techniques for centralized backtracking also apply to synchronous distributed backtracking:

- forward checking and higher degrees of consistency can be implemented by letting each agent a_i maintain a label that contains the admissible values for its variable x_i. Constraint propagation between different variables is implemented by sending messages between the corresponding agents.

- the variable ordering can be chosen statically or dynamically according to various heuristics, again by exchanging messages among agents.

- backjumping can be implemented by passing the backtrack not to the last involved agent, but to the one that is responsible for the variable to be backtracked to.

Considerable efficiency gains can be obtained by exploiting the parallelism inherent in the agents. Synchronous search can be extended with *asynchronous forward checking* ([17]). Here, forward checking is executed in parallel by sending messages to all agents that are responsible for unassigned variables, rather than treating them sequentially. This parallelism can also be extended to subsequent propagation that achieves higher degrees of consistency.

In *dynamic distributed backjumping* ([20]), instantiation continues in parallel with forward checking, and agents are informed of domain wipeouts by nogood messages. An additional heuristic identifies potential conflicts with variable assignments and orders values to avoid these conflicts. These two modifications bring improvements of about 1-2 orders of magnitude in cycles, constraint checks and number of messages.

However, synchronous backtracking has essentially the same restrictions as centralized solutions, except that there may be a small advantage in privacy in that constraints do not have to be communicated to any other parties. Furthermore, they do not exploit the potential for parallel execution among the different agents, as essentially only one agent is active at any one time.

20.2.2 Asynchronous Backtracking

Most of the research effort in distributed constraint satisfaction has focussed on asynchronous distributed search algorithms. These are characterized by the fact that all agents are active in parallel and only coordinate as needed to ensure consistency of the constraints their variables are involved in.

The first category of asynchronous search algorithms are *asynchronous backtracking* algorithms that perform a systematic exploration of the entire search space. The first of

Algorithm 20.1: ABT-opt: Asynchronous backtracking adapted for optimization.

```
1:  Procedure receive-add-link(xj)
2:  add xj.agent to self.lower-agents
3:  call(receive-ok(xj.agent,self.x))

1:  Procedure receive-ok(var)
2:  for ng ∈ nogoods do
3:     if ∃vc ∈ conds(ng) vc.agent=var.agent ∧ vc.v ≠ var.v then
4:        eliminate ng from nogoods
5:  replace v ∈ self.agentview s.th. v.agent = var.agent by var
6:  adjust-value

1:  Procedure receive-nogood(new-ng)
2:  for var ∈ new-ng.cond do
3:     if (cv ← x ∈ self.agentview s.th. x.agent = var.agent) = NIL then
4:        add var to self.agentview
5:        call(var.agent,receive-add-link(self.x))
6:     else
7:        if cv.v ≠ var.v then
8:           return
9:  for ng ∈ self.nogoods s.th. ng.v = new-ng.v ∧ new-ng.tag ∩ ng.tag = ng.tag do
10:    eliminate ng from self.nogoods
11: self.nogoods ← self.nogoods ∪{new-ng}
12: adjust-value

1:  Procedure adjust-value
2:  old-value ← self.x.v ; self.cost ← ∞
3:  for v ∈ self.domain do
4:     δ ← r(v) ; LB ← 0 ; tag ← { self } ; exact ← true
5:     for xj ∈ self.agentview do
6:        δ ← δ + r(xj,v)
7:     for ng ∈ self.nogoods s.th. ng.v = v do
8:        LB ← LB + ng.cost
9:        tag ← tag ∪ ng.tag
10:       exact ← exact ∧ ng.exact
11:    exact ← exact ∧ (tag ∩ self.lower-agents = self.lower-agents)
12:    if δ + LB ≤ self.cost then
13:       self.x.v ← v ; self.cost ← δ + LB; self.tag ← tag; self.exact ← exact
14: if (self.cost ≠ 0) ∨ self.exact then
15:    if (self.agentview = φ) ∧ self.exact terminate(self.cost)
16:    xj ← lowest priority variable in self.agentview
17:    call(xj.agent,receive-nogood(nogood(xj.v,
       agentview\xj,self.tag,self.cost,self.exact))
18: if self.x.v ≠ old-value then
19:    for a ∈ self.lower-agents do
20:       call(a,receive-ok(self.x))
```

`self`:

`x`	own `variable`
`domain` (constant)	domain of the own variable
`r` (constant)	`r(xj,vi)` gives the cost $r_{\{xj,self.x\}}(xj.v, vi)$ (associated with the constraint with `xj`) and `r(vi)` gives the cost $r_{\{self.x\}}(vi)$ if there is unary constraint on the own variable, and 0 otherwise
`agentview`	set of `variable`s
`lower-agents`	set of pointers to agents
`nogoods`	set of `nogood`s
`cost`	estimate of the cost of self and lower priority variables
`tag`	set of agents, used to keep track of received nogoods
`exact`	true if cost is exact and false if only a lower bound

`variable`:

`v`	value
`agent`	pointer to responsible agent

`nogood`:

`v`	variable value that nogood refers to
`cond`	set of `variable`
`tag`	set of pointers to agents
`cost`	cost of the nogood
`exact`	true/false depending on whether cost is exact or only a lower bound.

1: **Procedure** ABT-opt
2: self.x.v ← NIL; self.lower-agents ← {lower priority agents sharing a constraint}
3: self.agentview ← {higher priority variables sharing a constraint}
4: adjust-value

Figure 20.1: Data structures and main procedure for ABT-opt.

these algorithms, Asynchronous Backtracking(ABT), was published by Yokoo et al. ([37], and it has become a reference in the field on which many other algorithms are built. It was formulated for binary constraints and here we will also assume that all constraints are binary.

Figure 20.1 shows the data structures used for the ABT algorithm. Each agent has a data structure `self` that contains its own variable, constants that represent the domain of its variable as well as the constraints it has with other variables, and several fields that are used to store the current state of the search. The other data structures are `variables` and `nogoods` that will be explained in more detail below.

Algorithm 20.1 shows the main procedures of the ABT algorithm. Its presentation is slightly adapted so that it can also be used for optimization, following ([31]) and ([34]). It uses the construct **call** to indicate that the agent sends a message to another agent, thus invoking the receiving agent's procedure for receiving this message. These invocations are

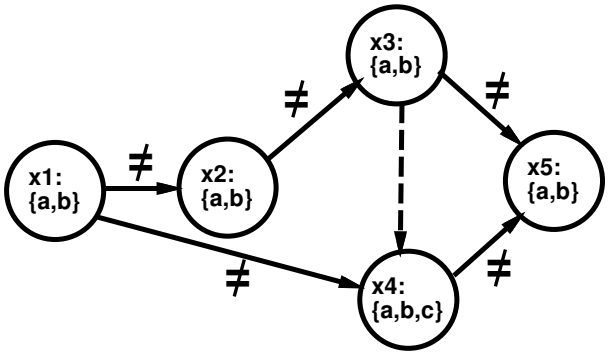

Figure 20.2: Example of a distributed constraint satisfaction problem. Circles represent different agents, directed arrows represent constraints between variables. Here, all constraints are inequality constraints. The dashed arrow indicates a link that might have to be added during the execution and is not part of the problem.

made asynchronously, i.e. they do not return any values and the invoking process does not wait for them to finish. However, it is assumed that no invocations are lost, all invocations by the same agent are handled in the order in which they were made, and no agents crash[1]

ABT explores a backtrack search tree with a fixed variable ordering that we assume to be $x_1, x_2, .., x_n$, without loss of generality. The ordering is assumed to be known to all agents, and establishes *priorities* in that a variable x_i has priority over another variable x_j whenever $i < j$. Note that this does not imply that any agent has knowledge of the entire problem: the order could be established for example by assigning each agent a unique number (e.g. serial number of the processor and the process id of the agent) and letting the ordering be identical to this numbering.

The variable ordering is used to decide a direction for each constraint: the agent controlling the first variable in the ordering is called the *value-sending* agent, and the other is called the *constraint-evaluating* agent.

As an example, consider the problem shown in Figure 20.2, where variables are assumed to be ordered according to their index. The constraints are thus directed as indicated by the arrows.

The algorithm is initially called by invoking each agent with the procedure `ABT-opt`, shown at the bottom of Figure 20.1. This initializes the own value, identifies the initial set of lower priority agents, and initializes the `agentview` to the higher priority variables, which are initialized with values that could be randomly chosen. The agent then picks an own value by calling `adjust-value`, which also causes it to be sent in `OK` messages to the lower priority agents. Once all these messages have been received, every agent has a correct `agentview`. During the search, agents continue to set their value asynchronously whenever they receive new information.

When an agent receives an `OK` message, it is invoked through the procedure `receive-ok`. The agent keeps a record of nogoods that identify assignment combinations that have

[1] When agents do crash, the algorithm still terminates, but the constraints enforced by crashed agents are not necessarily satisfied in the final result.

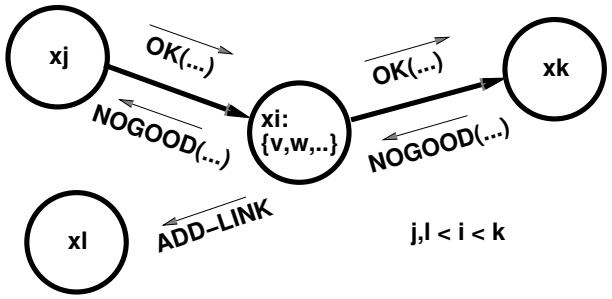

Figure 20.3: Messages used in asynchronous backtracking. The agent responsible for x_i receives `OK` messages from higher priority agents a_j ($j < i$) and `nogood` message from lower priority agents a_k ($k > i$) when their variables have common constraints with x_i. It may send `add-link` messages to request the value of other higher priority variables x_l.

been found to lead to inconsistencies or costs as explained further below. In order to avoid a combinatorial explosion in its memory requirements, the agent first eliminates any nogoods that are no longer valid given the new assignment. It then integrates the received value into its `agentview`. This is a data structure that keeps track of all values of higher priority agents that are relevant for its search.

Next, it adjusts its own value to optimally satisfy the constraints. This is done in the procedure `adjust-value`. It first evaluates the constraints with earlier variables to determine which values in its domain cause the least amount of inconsistency. For each value `v` in the domain, the variable δ is used to compute a lower bound on the cost that this value would imply. We assume here that the constant `r` returns a value of 0 if the combination given as an argument is allowed by the constraint, and 1 otherwise, so that the cost measures the number of constraint violations. The agent first sums the number of constraint violations with higher priority variables given in `agentview`(Steps 5-6). In LB, it adds any nogoods that may exist on the value, indicating violations in lower priority variables(Steps 7-10). The `tag` and `exact` variables are used for termination detection and are explained later (see 20.2.4).

The agent computes the number of violations in cost as δ + LB, and chooses the value with the smallest cost(Steps 12-13). Ideally, this cost is 0, in which case it has found a consistent value. If no consistent value can be found, the minimum cost is > 0, and the agent sends a `nogood` to the lowest higher-priority agent in its agent view(Steps 14-17). Finally, if this leads to choosing a different value, it is sent to the lower priority agents via `OK` messages(Steps 18-20).

In the example, assume that all variables are initially set to `a`. We consider one possible execution where all messages take about the same time, so that it can be understood as a sequence of parallel rounds. However, such synchrony is not required to obtain the correct results. In the first round, agents receive the following messages and make the following changes:

	message(s)	action
a_2	OK(x_1=a)	$x_2 \leftarrow$ b
a_3	OK(x_2=a)	$x_3 \leftarrow$ b
a_4	OK(x_1=a)	$x_4 \leftarrow$ b
a_5	OK(x_3=a)	$x_5 \leftarrow$ b
	OK(x_4=a)	

which leads to a second round:

	message(s)	action
a_3	OK(x_2=b)	$x_3 \leftarrow$ a
a_5	OK(x_3=b)	$x_5 \leftarrow$ a
	OK(x_4=b)	

and a third round:

	message(s)	action
a_5	OK(x_3=a)	inconsistent!

If no value consistent with the agent view can be found, as is the case for a_5, the agent can conclude that the current agent view is responsible for this failure. It constitutes a *nogood*: a combination of assignments that makes it impossible to assign a value to the variable, and thus cannot be part of any solution of the CSP. The nogood is constructed as a *resolvent* of all constraints and existing nogoods. It is sent in a `nogood` message to the agent responsible for the lowest priority variable in the nogood.

Nogoods are generated whenever an agent a_i does not find a consistent value for its variable x_i. In this case, each value in d_i entails at least one constraint violation, i.e. the minimum cost for all values is > 0. In this case, the agent concludes that its current agent view does not allow any solution, and passes this up as a `nogood` to the lowest priority agent in its agentview.

Here, d_5 has possible values `a` and `b`. where:

$$x_3 = \texttt{a} \quad \Rightarrow \quad x_5 \neq \texttt{a}$$
$$x_4 = \texttt{b} \quad \Rightarrow \quad x_5 \neq \texttt{b}$$

Thus, the minimum cost is equal to 1, so that the current agent view, $(x_3 = \texttt{a}, x_4 = \texttt{b})$ is passed up as a nogood with `v = b`, `cond` = $(x_3 = \texttt{a})$, `tag` = x_5 and `cost = 1` to a_4, the lowest priority agent in the agent view.

For each of the values $v \in d_i$, agent a_i stores one or more nogoods whose `cost` field gives a lower bound on the cost that will be incurred by variables in the field `nogood.tag`:

$$(x_j = v_j \wedge x_k = v_k \wedge ... \wedge x_i = v) \Rightarrow \mathit{cost}(\texttt{nogood.tag}) \geq \texttt{nogood.cost}$$

which, assuming that x_i is the lowest-priority variable, is written in explanatory form as:

```
nogood.cond ⊆ self.agentview ∧ nogood.v = self.x.v ⇒
cost-sum(nogood.tag) ≥ nogood.cost
```

and stored with agent a_i. In the version of ABT shown here, agents ensure that all nogoods for the same value refer to disjoint tags and that they are all applicable in the current agent view.

An agent receives a `nogood` by being invoked through the `receive-nogood` message. Note that an agent may receive a nogood that contains variables that it previously had no constraint with, and thus are not part of its agent view. In order to decide whether the nogood is applicable, it thus has to add a link with this variable so that it will be informed whenever it changes. This is done using an `add-link` message(Steps 2-5).

An agent applies its nogoods when checking for consistency of its current value assignment. It has to only apply those nogoods that are consistent with the current agent view. In fact, any nogoods that are no longer consistent with the agent view can be discarded as they will in any case be rediscovered should the agent view again become compatible with it. This means that the amount of storage required at each agent grows at most linearly with the size of the domain and the number of variables. As a consequence, when an agent receives a nogood which is no longer applicable to the current agentview, it discards it(Steps 7-8).

Next, it eliminates all existing nogoods that already cover part or all of the variables covered by the nogood just received, as indicated by the tags. These may exist because the agent may have already received nogoods resulting from partial propagation among the lower-priority agents. Finally, it adds the nogood to the nogood list, and adjusts its current value. This may result in further nogood messages being passed to higher priority agents(Steps 9-10).

In this example, a_4 has now received a nogood that involves x_3, and it can no longer evaluate its applicability. Thus, it sends an `add-link` message to a_3 and is informed that the current value of x_3 is `a`, so that the nogood is indeed applicable and its own value is no longer consistent. Thus, a_4 now searches for a new value for x_4 and finds value `c`, which now makes the entire problem consistent, and no further messages result.

Bessière et al. [3] show that adding links can be skipped if after sending a nogood of nonzero value, all nogoods involving a variable that is not linked with the current one are immediately discarded. If they are indeed required, they will eventually be rediscovered, but the algorithm will repeat a portion of the search and thus become less efficient.

Algorithm 20.1 terminates when the highest priority agent derives a nogood that is either exact or has non-zero cost. In the first case, the algorithm has found an assignment without constraint violation and thus a solution to the CSP. In the second case, the problem has no solution. The termination is initiated by a procedure **terminate** that should also inform the other agents that the process is now terminated. For more detail see Section 20.2.4.

When agents operate asynchronously, it can happen that some agents change value faster than others. This can lead to an agent receiving several `OK` messages for the same variable before processing them. In this case, only the last message has to be kept. Timestamps can be used to identify this when message delivery times are not predictable.

20.2.3 Asynchronous Distributed Constraint Optimization

Constraint optimization most commonly uses the branch-and-bound algorithm. This algorithm is difficult to adapt to an asynchronous, distributed setting because it requires agreement among all agents on a global upper bound on the solution cost. However, optimization can be carried out purely on the basis of lower bound propagation.

As has been observed in [31], the asynchronous backtracking algorithm (Algorithm 20.1) can also be used for constraint optimization by assuming that constraints no longer

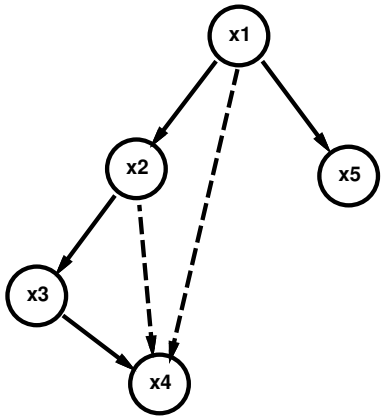

Figure 20.4: DFS tree ordering.

just return values of 0 (consistent) and 1 (inconsistent), but a more general cost measure in $\Re^+$. The algorithm works exactly as in the constraint satisfaction problem. However, while in the case of constraint satisfaction, most of the time a variable's nogood ends up having a value of 0 and is thus not transmitted, in this case most nogoods will have a positive cost and thus travel up the hierarchy.

To show the correctness of the optimization, consider in more detail the computation of local costs. An agent responsible for variable x_i the local cost $\delta(v)$ of value v of its own variable as

$$\delta(v) = r_{\{x_i\}}(v) + \sum_{x_j \in agentview} r_{\{x_i,x_j\}}(v, v_j)$$

i.e. the cost of this value at the variable itself plus the sum of all costs for the constraints with higher priority variables. It uses this value to compute for each value v the *lower bound*

$$LB(x_i = v) = \delta(v) + \sum_{ng \in compatible-nogoods(x_i=v)} ng.cost$$

as the sum of the local value and all nogoods for this value that are compatible with the current agent view. It always chooses its own value as the value v that minimizes the lower bound. If this is different from the current value, it sends `OK` message to lower priority agents.

The nogoods that agents receive express lower bounds on the total cost of constraints evaluated by lower priority agents. They supersede any earlier nogoods that refer to the same or a subset of the variables. Thus, Algorithm 20.1 checks for this in lines 9 and 10 of procedure `receive-nogood` and discards any obsolete nogoods.

However, it would still be possible for several nogoods to refer to the same variable and thus for the corresponding costs to be summed up multiple times. When we only consider constraint *satisfaction*, this is not a problem because it is not necessary to compute the exact number of violations, but only to detect that some violation is present. However, in optimization such overlaps lead to inaccurate results.

We now show that if agents are ordered in a *DFS tree* order, such overlaps can never occur. A DFS tree is a rooted and directed spanning tree of the constraint graph such that any edge not in the tree, called a *back edge*, can only exist between an ancestor and a descendant in the tree, but never to a sibling or descendant of a sibling. Figure 20.4 shows an example of a DFS tree consisting of 5 nodes. Tree-edges are shown as solid lines, and back-edges as dashed lines. Any graph can be ordered as a DFS tree, for example using depth-first search traversal to find the tree-edges and order. This process can also be carried out as a distributed algorithm.

In a DFS tree, backedges from a node x_i can only lead to nodes x_j that are on the path from the root to x_i. Since agents always send nogoods to their lowest-priority ancestor, no nogood can ever be sent between agents connected by a back-edge, but only along tree edges. Thus, there is always a unique path from any agent to any other agent, and so a given variable x_k can participate in only one chain of nogoods. This means that the nogoods sent by the children always give an exact bound on the actual cost of the lower priority agents.

Thus, ordering the agents as a DFS tree simplifies the algorithm and guarantees non-overlapping nogoods. A similar observation underlies the AND/OR search trees discovered recently ([15]).

This algorithm is essentially the ADOPT algorithm described in [19]. However, ADOPT also includes a mechanism of *backtrack thresholds* to avoid excessive recomputation of nogoods. It addresses the problem that the algorithm frequently recomputes earlier partial solutions as the bounds change.

Consider a variable x_i and three states s_1, s_2 and s_3 such that all higher-priority variables have identical values, but x_i changes first from `a` to `b`, then back to `a` since the nogood for `b` turns out to be bigger than that of `a`. When x_i changes state, all lower-priority variables discard their nogoods. Thus, when x_i changes back to `a`, they will again have to search to re-establish the solution they had already reached earlier. However, x_i's ancestor still has a nogood for value `a`, and can indicate this as the cost of the best solution that the lower-priority agents must find. The lower-priority agents can use this information to speed up their search that reconstructs the optimal solution for value `a`. Backtrack thresholds involve some further bookkeeping issues that are addressed in detail in the ADOPT algorithm ([19]).

20.2.4 Termination Detection

In the context of the ABT algorithm as we have described it here, termination detection can be achieved by also propagating upper bounds on the quality of the solution. This has been first proposed in the ADOPT algorithm ([19]). Note first that in ABT-opt, when all agents below x_k have received all nogood messages, the lower bound computation returns the exact cost of the subproblem below the sending agent. Thus, we include in the nogood an extra field that indicates if the nogood is exact or not. This is then used by higher priority agents to see if their costs are exact: only if they have received exact nogoods covering all lower priority agents will their costs become exact as well. Note that exact nogoods are passed on even when their cost is zero so that the receiving agents can tell whether their cost is exact.

Finally, when the highest-priority agent derives an exact nogood, the algorithm terminates. In a constraint satisfaction problem, if the cost is non-zero, the problem has no

consistent solution. Otherwise, the current assignment to all variables is a consistent or optimal solution.

20.2.5 Soundness, Termination and Completeness of ABT

Soundness and termination of the ABT algorithm can be proven inductively as follows. Consider variable x_k and assume as inductive hypothesis that as long as variables $x_1..x_{k-1}$ do not change value, variables $x_k..x_n$ will converge in finite time on an assignment whose cost is minimal given the values of $x_1..x_{k-1}$, and that this cost is transmitted to x_k as a nogood with the exact field set to `true`.

Clearly, the inductive hypothesis holds for x_n since there are no lower priority variables, it chooses its optimal value instantly and transmits its exact cost to its parent.

Now consider variable x_k. It will change value only when the nogood for its current value increases so that its cost becomes greater than that of another value. As the nogoods form lower bounds on the optimal costs, they cannot increase beyond this optimal cost. Since each nogood is the sum of costs taken from a finite set, this implies a bounded number of increases. Thus, x_k must eventually reach quiescence, and by the inductive hypothesis, it will receive exact nogoods with the costs of the optimal assignments for $x_{k+1}..x_n$. Since the algorithm chooses the value of x_k to minimize the cost of $x_k..x_n$ given the value of $x_1..x_{k-1}$, x_k can only stabilize on the optimal value and then sends an exact nogood with the optimal cost of $x_k..x_n$ given the values of its ancestors. Thus, it also satisfies the inductive hypothesis. By induction, the hypothesis also holds for x_1, which proves soundness and termination of the algorithm.

For the constraint satisfaction case, completeness follows from termination and the fact that the algorithm finds an optimal solution. If there is an assignment that satisfies all constraints, it has cost 0 and so the algorithm will terminate with an assignment that has cost 0 and violates no constraints.

20.2.6 Performance Evaluation

The complexity of constraint satisfaction algorithms is commonly measured by counting the number of constraint checks. In asynchronous search algorithms, a more accurate measure of the expected execution time is to count the number of *concurrent constraint checks*, given as the smallest number of cycles required when each agent can execute a constraint check in parallel, thus considering the interdependency of their execution. Meisels et al. ([16]) presents an algorithm called CCA for computing the number of concurrent constraint checks during a simulation run of an algorithm.

In a distributed execution, sending a message often takes much longer than a constraint check. For example, sending a message through e-mail can take minutes, amounting to millions of constraint checks. Thus, many researchers measure message complexity as the main measure of expected execution time. Here again, one can simply count the total number of messages, or obtain a measure of *concurrent messages* that more accurately reflects the interdependencies among them.

An issue here is also the size of messages, as some algorithms may be able to package information into fewer but larger messages. This applies particularly to techniques based on dynamic programming, described later in this chapter.

It has been customary in distributed systems research to show a graph of complexity vs. the ratio between the time required to send a message and the time required for a constraint check. This measure has been used in the original paper on ABT ([37]) and in several other algorithm evaluations, and is the most comprehensive performance measure since it also makes apparent parallelism between message delivery and computation.

20.3 Improvements and Variants

20.3.1 Agents Controlling Constraints Instead of Variables

In ABT, it was assumed that each variable is under the control of an agent, and that this agent knows all constraints relevant to that variable. In many applications, variables are public knowledge and it is necessary to generate a consensus among agents as to their value. On the other hand, agents are free to set constraints as they wish.

Silaghi et al. ([28]) have shown how ABT can be adapted to this situation. It involves treating a dual problem where agents exchange constraints or parts of constraints rather than variable assignments. To represent them efficiently, their *asynchronous aggregation search*(AAS) algorithm uses aggregations of values that are described next.

20.3.2 Value Aggregation to Reduce Message Traffic

When variables in a DisCSP have large domains, it is often the case that several values behave the same with respect to constraints. It is then more useful to aggregate them into a single value that can be treated in a single message.

This idea is developed in the AAS algorithm ([28, 33]). AAS is similar to ABT, but uses the dual of the original problem so that agents are now responsible for constraints, and variables are shared between agents that have constraints on them. Each agent decomposes the space of value combinations of a constraint into equivalent groups such that all value combinations within them have the same cost. These can be considered the values of the dual variables, and the algorithm then performs the ABT algorithm on this dual problem. Some complications occur since the decompositions may have to be refined during search. On randomly generated problems, aggregation brings improvements of several orders of magnitude in search efficiency ([33]).

20.3.3 Distributed Consistency Maintenance

One of the most successful techniques in (centralized) CSP is consistency, in particular arc consistency. They can be adapted to asynchronous settings as well, but labels now have to refer to the context of higher priority variables that has been used to generate them so that they can be reset whenever this context is no longer valid.

The MHDC algorithm ([33]) maintains arc consistency during distributed search in AAS by adding a separate type of message called a *propagate* message. It again results in very significant performance gains on randomly generated problems.

20.3.4 Asynchronous Weak-Commitment Search

An important weakness of ABT is that it uses a static variable order, which is known to lead to inefficient search in CSP. In asynchronous weak-commitment search (AWC) [38]), agent priorities are dynamically adjusted so that whenever a backtrack occurs, the agent initiating the backtrack becomes the highest priority agent. This focusses the search on the most difficult parts of the problem space, and AWC is reported to be significantly (at least 1 order of magnitude) more efficient than ABT. However, a major drawback is that AWC is complete only when all nogoods are stored, leading to exponential storage requirements.

20.3.5 Asynchronous Reordering

Reordering is one of the most powerful techniques for speeding up search algorithms for constraint satisfaction and optimization. In asynchronous search, reordering is significantly more complex as there is no central view of the problem.

A first algorithm that uses reordering is AWC, described above. AWC needs to store an exponential number of nogoods to be complete and is thus not considered practical.

However, it is possible to allow a more limited form of reordering if agents are only allowed to change the orders of lower priority agents. Such reorderings do not affect the validity of the nogoods that have been received from these agents, and thus termination in a finite number of steps after the last reordering is still guaranteed. This has been proposed by Silaghi et al. ([29]) in the ABTR algorithm and more recently by Zivan and Meisels ([42]) in the ABT_DO algorithm.

In these algorithms, each agent can impose a new ordering of the agents below itself, and inform these lower-priority agents of the new order. When an agent receives a message informing it of a new order, it adjusts its `agent-view` to add all agents that now have a higher priority, and discards all nogoods that mention agents that now have a lower priority.

When several agents propose reorderings, their priority is decided using a signature scheme. It consists of a set of counters, one for each position in the ordering: $(c_1..c_n)$. When the k-th agent in the current ordering proposes a new order, the signature of this new order is derived from the old one by keeping all c_i,$i < k$ the same, increasing c_k by 1, and setting all c_j,$j > k$ to 0. Priority between orderings can now be decided by comparing their signatures lexicographically, i.e. letting l be the first position where two signatures differ, the signature with a higher c_l has higher priority.

Note that given the restrictions on allowable orders, the highest priority agent can never leave its position. Silaghi et al. ([30]) show a protocol based on proxy agents that allows general reorderings, but at the expense of a more complex algorithm where roles are exchanged between agents.

Both Silaghi et al. [29] and Zivan and Meisels ([42]) report gains in efficiency for certain reordering heuristics; however, these gains are not nearly as significant as what can be observed in centralized algorithms.

20.3.6 Storing Nogoods

One of the main problems with asynchronous search is that in order to limit the amount of storage required, nogoods are erased as soon as they become inapplicable due to changes in the agent view. This means that the algorithms derive the same information over and

over again. Much efficiency can be gained by systematically storing all nogoods that are discovered during search. This is particularly interesting when variables are ordered as a DFS tree so that nogoods form tight bounds on the cost of possible solutions. It has been shown experimentally ([33]) that such storage can tremendously increase the efficiency of asynchronous backtracking algorithms.

The amount of memory required for systematically storing all nogoods can be bounded using the following consideration. The maximum number of nogoods that need to be stored at an agent is equal to the size of its own domain times the number of possible contexts, i.e. assignment combinations to higher-priority variables that are the target of edges or back-edges from lower-priority variables. It can be shown ([24]) that for any variable, the number of variables in this context can never exceed the *induced width* of the DFS tree ordering. Thus, the maximum amount of space required at any agent is exponential in the induced width of this ordering.

In many practical distributed problems, this width is actually not very large. For example, in meeting scheduling, most meetings are between people in similar groups. It has been shown ([22]) that this leads to graphs with relative low induced width. Other examples, such as sensor networks, also typically have low induced width.

20.3.7 Cooperative Mediation

Another way to deal with the complexity of distributed optimization problems is to detect particularly difficult parts and solves those in a centralized fashion. The *optimal asynchronous partial overlay* (OptAPO) algorithm ([14]) dynamically calls upon certain agents to mediate by determining the optimal solution for itself and its neighbours using a centralized branch-and-bound algorithm. When message delivery is slow, as is usually the case, this can bring significant performance increases over algorithms based on asynchronous backtracking such as ADOPT.

20.3.8 Distributed Dynamic Programming

A fundamental problem with distributed backtracking algorithms is that they explore the search space sequentially by changing variable assignments. As variables are distributed, each change in assignment requires message exchange between agents. Since the search space has exponential size in the number of variables, the algorithms inevitably require an exponentially growing number of messages. Messages are costly and slow to send, so this is usually unacceptable.

Dynamic programming techniques such as *bucket elimination* ([7]) are interesting as they allow exploring all assignments in parallel. Thus, instead of sequentially exploring all assignments of a variable x_i and passing this on to a lower-priority variable x_j, variable x_j sends a single message to x_i that gives the optimal cost for each of the possible values of x_i. In the DPOP algorithm ([24]), agents are arranged in a DFS tree, as described above, and each agent communicates with its direct parent/children in the tree. Children send UTIL messages to their parents, while parents send VALUE messages to their children. Each UTIL message specifies, for each possible value combination of the parent and possibly a number of ancestors the optimal cost for the sending variable and all its descendants in the pseudotree. Value messages are similar to OK messages in that they specify the value assigned to the parent variable.

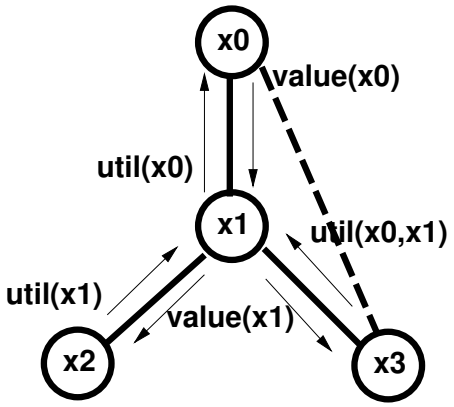

Figure 20.5: Example of a distributed optimization problem and its solution using DPOP.

Agents use the following rules:

1. an agent that has received UTIL messages from all its children and VALUE messages from all its parents decides on its optimal value and sends VALUE messages to all its children.
2. otherwise, if it has received UTIL messages from all its children it constructs a UTIL message to its parent.

Figure 20.5 shows an example of a distributed constraint optimization problem solved using the DPOP algorithm. Assume that each variable can take values w(hite) and b(lack), and that we have the following constraints:

$c(x_0, x_3) =$

x_0 \ x_3	w	b
w	3	0
b	3	3

$c(x_0, x_1) =$

x_0 \ x_1	w	b
w	1	0
b	2	2

$c(x_1, x_2) =$

x_1 \ x_2	w	b
w	1	0
b	0	1

$c(x_1, x_3) =$

x_1 \ x_3	w	b
w	2	0
b	0	2

Initially, only agents a_2 and a_3 satisfy the second behavior rule and send the following messages to their parent a_1:

$UTIL(x_1) =$

x_1	
w	b
0	0

$UTIL(x_0, x_1) =$

x_0 \ x_1	w	b
w	0	2
b	3	3

They give the lowest costs that can be obtained for these values of x_1 and x_0 given the best choices for x_2 and x_3, and are obtained by combining the constraints on x_2 and x_3,

respectively, using the bucket elimination operation ([7]). As soon as these messages have been received by a_1, a_1 also satisfies the second behavior rules and generates the following UTIL message to its parent a_0:

$$UTIL(x_0) = \begin{array}{cc} x_0 & \\ \hline w & b \\ 1 & 3 \end{array}$$

a_0 has no parents and has received UTIL messages from all its children, so it satisfies the first behavior rule, decides its value to be $\mathtt{w}$ and sends this to a_1 as a VALUE message. a_1 now has all the required information to decide on its best value, $\mathtt{w}$, and sends VALUE messages $(x_0 = \mathtt{w}, x_1 = \mathtt{w})$ to x_2 and x_3 who can finally decide on their own values $\mathtt{b}$. Propagation then stops since no agent satisfies any of the rules. All agents know that they have decided on the optimal value so that no further termination detection is necessary.

It is interesting to consider solving the same example as above using asynchronous backtracking. When nogoods are systematically stored for all combinations of higher-priority agents, agents exchange exactly the same information as in dynamic programming, but through a sequence of nogoods. The memory required in each agent to store all nogoods is identical to the size of the largest message in dynamic programming. When nogoods are not stored systematically, a high price is paid for rediscovering nogoods over and over again. Thus, in the distributed case we can understand backtracking and dynamic programming as two related approaches.

In the DPOP algorithm, the number of messages grows only linearly with the size of the problem. However, messages may become very large. It can be shown ([24]) that the maximum message size is exponential in the induced width of the pseudotree ordering used. This growth can be dealt with using a technique similar to that of mini-bucket elimination ([8]). Here, we need to identify higher priority variables that are involved in the highest-dimensional messages. These variables will then change their values incrementally while informing the lower-priority agents, similar to what is done in asynchronous backtracking. While this reintroduces the problem of message explosion due to the state changes of these variables, growth is much more moderate since in a problem with low width, there are only few such variables.

The dynamic programming formulation also has several other advantages:

- it is possible to limit memory consumption by dropping dimensions of UTIL messages, and propagate upper and lower bounds ([25]). This allows computing solutions that are optimal within these bounds.

- it is possible to stop propagation of UTIL messages when the differences between values are insignificant, either because they have no influence on the rest of the problem or because their influence can be bounded by an approximation tolerance ([25]).

- for settings where the problem undergoes dynamic changes, it is possible to incrementally adapt the solution using a self-stabilization technique ([23]): each agent that observes a change initiates new UTIL messages that propagate through the network and initiate changes wherever necessary. All agents know simultaneously when the new optimal value has been reached and can change to this value without any further synchronization mechanism, thus achieving super-stabilization. Such a

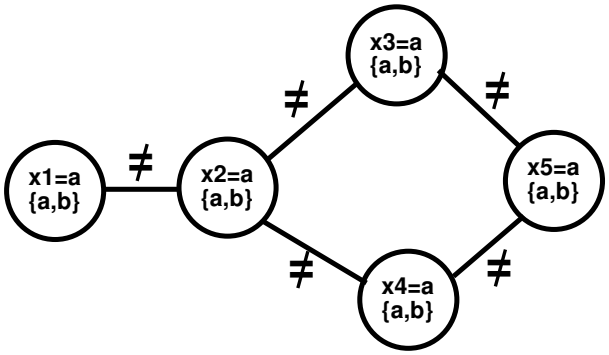

Figure 20.6: Distributed local search.

property is interesting for using distributed optimization as a method for controlling a distributed system.

It can be argued that for problems with high width constraint graphs, the distributed CSP approach does not offer significant advantages and would be solved better by centralizing the problem. As the search is likely to explore a large portion of the possible states, agents need to reveal information about their constraints to many other agents and so there is little privacy advantage. Furthermore, due to the high cost of message exchange, it is unlikely that the parallelism inherent in the distributed algorithm will bring any practical advantage.

20.4 Distributed Local Search

Recall that local search (see Chapter 5) works by starting with an initial configuration where values are assigned to all variables, and then makes incremental modifications to reduce the number of inconsistencies or minimize the cost of the configuration. Each modification is applied to individual variables or small sets of variables. Because of this local nature, they can be carried out by a single agent. This makes local search extremely well suited to distributed implementation.

Researchers have considered distributed local search algorithms where moves are restricted to modifications of single variables, and each variable is controlled by a single agent. Figure 20.6 shows an example of such a problem. We assume that initially, all variables are set to the value `a`, and thus all inequality constraints are violated.

The basic algorithm is *hill-climbing*: make local changes to the variables such that the number of constraint violations is reduced as much as possible. However, changes must be coordinated so that neighbouring agents in the constraint graph never change value at the same time. Otherwise, in the example each agent would change its local value to `b`, but not reduce constraint violations at all.

Algorithm 20.2 implements distributed hill-climbing with coordination. It requires synchronous execution with two rounds of message exchange with the set of neighbours N(x) in each cycle: first, to exchange the actual values, and later to exchange the best improvements. Note that each agent only has to know N(x), but nothing about the rest of

the problem, so that in spite of synchronous execution this algorithm can still be applied to unbounded problems.

Algorithm 20.2: Distributed local search algorithm.

```
1:  v(x) ← initial-value; tc1(x) ← 0; tc2(x) ← 0
2:  repeat
3:     send current value v(x) to all neighbours in N(x)
4:     receive current values v(xj) from all xj ∈ N(x)
5:     currentCost ← Σ_{x_j ∈ n(x)} c(x,xj=v(xj))
6:     if currentCost ≠ 0 then tc1(x) ← 0 else tc1(x) ← tc1(x)+1
7:     dmax ← 0; vmin ← NIL
8:     for v ∈ d do
9:        δ ← currentCost − Σ_{x_j ∈ n(x)} c(x=v,xj=v(xj))
10:       if δ > dmax then
11:          dmax ← δ; vmin ← v
12:    if dmax ≠ 0 then tc2(x) ← 0 else tc2(x) ← tc2(x)+1
13:    send improvement dmax and termination counts tc1(x), tc2(x) to all xj ∈ N(x)
14:    receive improvements dm(xj) and tc1(xj), tc2(xj) from all xj ∈ N(x)
15:    for xj ∈ N(x) do
16:       if dm(xj) > dmax ∨ (dm(xj)=dmax ∧ xj ≻ self) then
17:          vmin ← NIL
18:       tc1(x) ← min(tc1(x),tc1(xj)+1); tc2(x) ← min(tc2(x),tc2(xj)+1)
19:    if vmin ≠ NIL then
20:       v(x) ← vmin
21: until tc2(x) > max-dist
22: if tc1(x) > max-dist then success else failure
```

Thus, if an agent finds a neighbour that obtains a bigger improvement than itself, it will not change its value. The effect of this simple coordination is that no neighbours ever change value simultaneously. For the example, a consistent solution is achieved in the first round of execution:

var	current-cost	vmin	dmax	change
x_1	1	b	1	-
x_2	3	b	3	b
x_3	2	b	2	-
x_4	2	b	2	-
x_5	2	b	2	b

Note that only x_2 and x_5 change values: x_2 wins over its neighbours because it has the best improvement, and x_5 because it has the highest index. Since they are not neighbours, they can change in the same cycle.

To detect termination, Algorithm 20.2 uses two termination counters `tc1` and `tc2`. `tc1` measures the minimum distance of any variable that could be involved in a constraint violation. `tc2` measures the minimum distance of any variable that could be unable to make further improvement. Agents extend their knowledge by exchanging counters in the

second message exchange (Steps 13-14), and update their distances by taking the minimum of their values in (Step 18).

The algorithm must terminate when no agent can find any improvement. This is the case when the distance of any such agent is larger than the maximum distance of any agent in the constraint graph, given by the constant max-dist. This constant must be global knowledge of the problem, but can be an overestimation without affecting the correctness of the algorithm. If at termination the minimum distance of an agent with a violation, `tc1(x)`, also exceed this distance, then a consistent solution has been found (Step 22).

It is well-known that hillclimbing algorithms can easily get stuck in local minima where no local improvement is possible, but the best solution has not yet been found. Two types of solutions to this problem have been given for distributed local search.

The first solution is distributed stochastic search, where with some probability the algorithm also accepts changes that do not result in an improvement in the quality of the configuration. A detailed description and analysis of several such algorithms is given in [44]. The implementation of these techniques is a straightforward modification of the hillclimbing procedure given above. One way to do this is to insert a step following the computation of δ in Step 9:

9a. **if** $\delta \leq 0$ **then** with probability p, $\delta \leftarrow 1$

where the probability might be varied as optimization progresses.

Another solution is to adjust the problem topology using the breakout algorithm [39, 13]. Here, we associate with each constraint a weight that is initially set to 1 and varies over time. In the hillclimbing procedure, we do not simply sum the costs, but multiply each constraint by its weight.

Whenever the main loop of the algorithm terminates (Step 21) and the algorithm has not found a consistent solution (`tc1(x)` $\leq$ `max-dist`), the agent increases the weight of all currently violated constraints by 1. This has the effect of making the current optimum less attractive and thus driving the search to a different configuration in subsequent moves. Then, the procedure is restarted from the beginning. This process repeats until either a consistent solution is found or some timeout limit is reached. Basharu et al. ([1]) report that resetting weights periodically or in response to observing agent behavior further improves performance.

While the breakout algorithm often performs quite well at getting search out of local optima, there are simple situations that it fails to solve, as pointed out in [43]. Figure 20.7 shows an example of such a situation. Here, each node represents a variable that must be colored either black or white, and each arc is an inequality constraint between neighbouring nodes. Note that the problem is solvable by coloring the nodes alternatively black and white. Assume that the breakout algorithm starts in the configuration shown on the top left. There are conflicts between nodes 2 and 3, and between nodes 6 and 7, that both cannot be eliminated by a local change. Thus, a breakout step increases the constraint weights from 1 to 2, as indicated in the next step on the right. Now, the algorithm can make an improvement in the weighted sum of constraint violations by changing the color of nodes 2 and 6. However, this generates a similar situation to the initial one, with 2 local minima. The algorithm again increases the weight, makes changes to variables 1 and 5, and the cycle continues until finally we reach the same situation as the initial one, except that all constraint weights have increased by 1. Thus, the breakout algorithm will never find a solution to this problem, but infinitely cycle and increase the constraint weights.

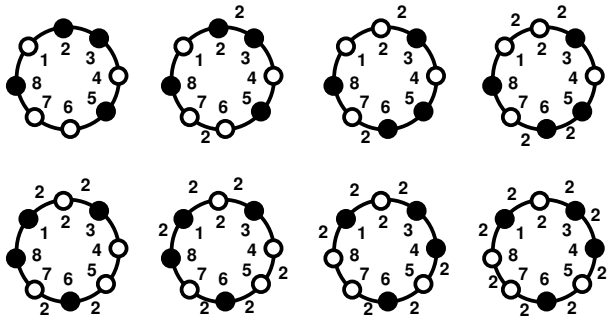

Figure 20.7: Problem that is unsolvable for the distributed breakout algorithm.

Note that such a behavior can be detected by the fact that constraint weights are continuously increasing in some subproblem. In fact, if there is a subproblem that the breakout algorithm is unable to solve - either because it has no solution or because of the algorithm's incompleteness - it must be the case that during each breakout cycle, at least one of the constraints has its weights increased. If the subproblem is small, it is possible to identify this subproblem and then solve it using a complete backtrack search algorithm. Such schemes, as described in [9], can be applied to solve large-scale distributed CSP with hundreds of variables.

20.5 Open Constraint Programming

In open constraint programming, the set of variables may be bounded and commonly known, but variable domains and admissible constraint tuples are distributed among a possibly unboundedly large set of information sources, so that the problem can never be completely centralized. Using transformations such as hidden-variable encoding (see Chapter 11), constraints can be treated as tuple-valued variables. It is therefore sufficient to consider distributed variable domains.

As an example of a problem requiring such an approach, consider a configuration system for financial portfolios. It can obtain information about available financial products from a large set of information sources. Furthermore, many of these products are themselves configured on demand by their providers. It is thus not possible to place a bound on the space of possible parts that can be considered in such a configuration.

The challenge in open constraint programming is to solve such a problem without knowing the complete domains. Algorithms are defined based on a model where variable domains are discovered incrementally by querying a mediator. We say that an algorithm for open constraint programming is complete if it always terminates with a solution when there is one; however, a complete algorithm may never terminate when there are unboundedly large domains. Note that a more restricted version that requires domains to be finite has been proposed as *interactive* constraint satisfaction in [6].

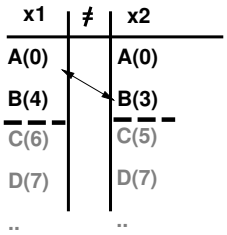

Figure 20.8: The principle underlying open constraint optimization.

Open constraint satisfaction Open constraint satisfaction is feasible since by the semantics of constraint satisfaction, a solution to a CSP remains a solution even when values are added to the domains of one or more variables:

Lemma 20.3. *Let A be a consistent assignment to an instance $CSP(i)$ of an OCSP. Then A is also a consistent assignment to all instances $CSP(j), CSP(i) \prec CSP(j)$ of the same OCSP.*

Proof. As the domains of $CSP(i)$ are contained in those of $CSP(j)$, A is also an assignment in $CSP(j)$. As the constraints remain the same, it remains consistent. □

Thus, if we find a consistent assignment to an instance $CSP(i)$, we have found a solution to the OCSP, and do not need to examine any further values.

Algorithms for open constraint satisfaction incrementally query information sources for additional domain values until they find either a solution, or detect a subproblem where all domains have been completely obtained and that has no solution. The key issue is to query values in a balanced way so that the problem becomes solvable as quickly as possible, and to detect unsolvable problems even in the presence of unbounded domains without falling into infinite queries of these values. Centralized algorithms for open CSP can be found in [12].

Open constraint optimization In open constraint optimization, a solution is not just any variable assignment that satisfies the constraints, but an assignment that maximizes a utility (or, equivalently, minimizes a cost). We assume that utilities are given by additional soft constraints that are formulated on the values of individual, possibly tuple-valued variables.

Open constraint optimization is feasible under the condition that queries to the mediator always return the most preferred values or value combinations first. It is shown in [12] that if this condition does not hold, it is not possible to prove that a solution is optimal without retrieving the entire domains of variables, and thus not possible to have a general algorithm for solving open constraint optimization problems.

The reason for this fact is illustrated by Figure 20.8. It shows a COP with just two variables x_1 and x_2, connected by an inequality constraint. The variables can take different values with their costs shown in parentheses. The optimal solution is $x_1 = a, x_2 = b$ with a total cost of $0 + 3 = 3$. To show that this solution is indeed optimal, it is sufficient to know the first two values of each domain, since:

- any solution that would use a value of x_1 beyond the second value (b) would have cost of at least 4 for x_1 and 0 for x_2 (cost of the best possible value), yielding a sum of 4 which is more than the 3 we get in the proposed solution.

- any solution that would use a value of x_2 beyond the second value (b) would have cost of at least 3 for x_2 and 0 for x_1, yielding a total of 3 which again is no better than the proposed solution.

Based on this principle, it is possible to construct algorithms that determine the optimal solution without querying the entire variable domains. While there can not be any deterministic algorithm that always examines only the minimal number of values necessary to prove optimality, it is possible to come close to this limit using techniques based on the A^* algorithm ([12]).

Algorithm 20.3: fo-opt: an incremental algorithm for solving OCOP.

```
1:  Function fo-opt(OCOP)
2:  For i ∈ {1..n}, d_i ← (more(x_i))
3:  OPEN ← {(first(d_1),..,first(d_n))}
4:  loop
5:    M ← {a ∈ OPEN|cost(a) = min_{b∈OPEN} cost(b)}
6:    a ← lexicographically smallest element of M
7:    remove a from OPEN
8:    if consistent(a) then
9:      return a
10:   else
11:     c ← c(x_k,...,x_l) such that max(k,...,l) is the smallest and c is violated in a (first
        violated constraint)
12:     for j ∈ vars(c) do
13:       if a(j) = last(d_j) then
14:         d_j ← append(d_j,more(x_j))
15:       nxj ← succ(a(j),d_j)
16:       b ← (a(1),..,a(j-1),nxj,a(j+1),..,a(n))
17:       if b ∉ OPEN then
18:         OPEN ← OPEN ∪{b}
```

Algorithm 20.3 is an example of such an algorithm. Search nodes are complete (but possibly inconsistent) assignments to all variables. Initially, the algorithm uses the function **more** to query the best value for all variables, and thus becomes the initial search node.

Following the best-first search heuristic, the OPEN list of nodes is kept ordered by decreasing utility, and the best node is chosen to be expanded next(Steps 5-7). When this node is a consistent assignment, an optimal solution has been found(Steps 8-9).

Successors to a search node could be generated by assigning one of the variables the next best value, thus giving each node n successors. When the domain is not sufficiently known, it is queried to obtain the new value. However, as shown in [12], it is only necessary to generate successors that include new values for the variables involved in the first violated constraint, where constraints are ordered according to the highest variable they

involve, according to some fixed ordering. This leads to a significantly lower memory consumption as well as a much smaller number of value queries, and significantly improves the performance of the algorithm which now comes close to the minimal number of queries. Algorithm 20.3 thus picks out the first violated constraint in Step 11 and generates the successors in Step 15. Note that the function **more** is used to query the next best domain value if necessary.

Algorithm 20.3 is guaranteed to produce the optimal solution because search nodes are explored in the order of non-increasing utility. Thus, when a consistent solution is found, it will necessarily be the one with the minimum possible cost.

While Algorithm 20.3 is a centralized algorithm, open constraint optimization can also be carried out by distributed algorithms. In particular, [27] shows how open constraint optimization can be integrated with the DPOP algorithm (Section 20.3.8) to produce a distributed constraint optimization algorithm that can deal with unbounded domains and exchanges significantly less information than the DPOP, ADOPT and ABT algorithms.

20.6 Further Issues

20.6.1 Incentive-Compatibility

Agents may have conflicting interests regarding the solution to a distributed CSP. If they are allowed to post any hard or soft constraints they like, it is in their best interest to enforce their preferences by exaggerating their constraints. If all agents adopt this behavior, the solution computed by the algorithm will no longer be meaningful. Another problem is that agents can manipulate the outcome by not correctly executing the distributed optimization algorithm ([21]).

Both problems can be avoided by mechanisms where agents are required to pay a tax corresponding to the constraints they impose on others. The tax is calculated so that it is in the best interest of agents to report their constraints truthfully. Such mechanisms are called *truthful* or *incentive-compatible* mechanisms.

Another property that is important in multi-agent settings is that of *individual rationality*. It means that each agent is better off by participating in the joint mechanism rather than remaining on its own. If a tax scheme is used, it means that the amount of tax an agent may be forced to pay is never greater than the gain it gets out of influencing the choice of the algorithm.

A well-known mechanism for incentive-compatibility is the *Vickrey-Clarke-Groves tax*(VCG) mechanism. It can be shown that it is the only general mechanism that guarantees both incentive-compatibility and individual rationality for all agents. Its application for multi-agent decision making has first been proposed in [10] and its application to distributed CSP described in [11]. [26] describes in detail how the DPOP algorithm can be combined with a completely distributed VCG tax mechanism, resulting in a scheme that is completely resistant to manipulation.

In the VCG mechanism, each agent pays the difference in cost to all other agents between the optimal solution when it is present and the solution when it is not:

$$payment(A) = \sum_{r_k \in R - R_A} r_k(v^*_R) - r_k(v^*_{R-R_A})$$

where R is the set of all relations, R_A is the set of relations imposed by agent A, and v^*_R, $v^*_{R-R_A}$ are the solutions that minimize the sum of costs in R and $R - R_A$, respectively. The tax must be paid to an uninterested party (charity).

The following argument shows why this tax makes it optimal for an agent to tell the truth:

- suppose that agent A overstates the importance of his constraints. Then it will gain an advantage for those cases where his claimed cost is higher than his real cost. However, it turns out that in these cases, he will also have to pay a tax which is higher than the benefit it gets out of having his solution chosen - consequently, this behavior is not rational for the agent.

- suppose on the other hand that the agent understates its costs. Then it will save the tax in those cases where it would fall between its stated and its true cost for the value that is chosen. However, in all these cases the tax would be lower than the loss it incurs by having this value chosen, so again it is not individually rational for the agent to act this way.

It has been shown ([21, 26]) that VCG taxes can also eliminate the potential for agents to manipulate the outcome by unfaithfully executing a distributed search algorithm.

The VCG tax mechanism applies only to constraint optimization with soft constraints. A hard constraint can cause an unbounded amount of utility loss to the remaining agents, and thus by the principle of VCG taxes an unbounded amount of tax for the agent that imposes it. Such a tax may be considered to violate that agent's individual rationality. Therefore, in general hard constraints should only be used to model commonly verifiable knowledge.

20.6.2 Privacy

One of the possible motivations for using distributed constraint satisfaction is to protect the privacy of agents' constraints. The ultimate protection is achieved when solving a CSP using cryptographic techniques. Here, no agent learns anything about other agents' constraints except that a certain combination of assignments - the final solution - is consistent with all constraints.

Secure distributed constraint satisfaction, as described by Yokoo in [41], is based on cryptographic techniques that achieve three properties:

1. constraints are encrypted, and consistency of a value assignment is decided without decrypting the constraints;

2. values are permuted in a random-looking way so that no agent can tell what value corresponds to what position;

3. algorithms search the entire search space so that no information can be drawn from the time it takes to find a solution.

Each agent encrypts all its constraints by generating a constraint matrix that includes encrypted versions of the elements 1 (consistent) and another number z (inconsistent). Using randomized encryption techniques, each of these is made to look like a completely random number so that an observer cannot tell whether two cells contain identical elements.

When the algorithm checks for the consistency of an assignment, it collects the relevant constraint entries and multiplies them together. Thanks to a second property of the encryption scheme, that of being homomorphic, the product of the encrypted numbers is equivalent to the encryption of their product. Thus, the product can subsequently be decrypted and checked for whether it contains a 1 - meaning that all constraints were 1, and thus satisfied - or another number, meaning that at least one constraint was not satisfied. This decryption must in fact be done by passing the result through all involved agents, and thus invariably leads to a very large number of messages.

To ensure that no agent can know what values were found consistent or inconsistent, the scheme furthermore involves a *permutation* of all domain values. Each agent permutes all constraint matrices referring to its own values with the same permutation of domain values, and applies a renewed randomization of the encryption so that the permutation cannot be discovered by comparing values.

Search can use a centralized or decentralized algorithm, but each constraint check requires a cooperative decryption of the result of multiplying the relevant constraints. When a consistent solution is found, each agent can apply the permutation of its domain in reverse and thus finds out what its value was.

In subsequent work it has been noted that since the computation time of the search algorithm reveals information about the constraints, and that even this protocol is not entirely secure. For an example of a protocol that is also secure against this kind of attack, see [32].

Cryptographic privacy protection is very costly to implement, and so far has not been used in practical applications of realistic size. In principle, all distributed CSP algorithms provide some level of privacy protection, since constraints are only revealed to neighbouring agents, and even here only when backtracking is required. An analysis of privacy loss is found for example in [36]. Researchers have also explored whether constraints could be shared by participants, allowing them to be enforced even though no agent knows the entire constraint ([4]).

20.7 Conclusion

Many applications of constraint satisfaction and optimization occur in settings with multiple agents and may even be unbounded. In that case, it is no longer feasible to solve them by centralizing all problem information on a single server. Distributed constraint satisfaction techniques address such naturally unbounded problems. The localized nature of constraint satisfaction is a major advantage in such settings; classical optimization techniques such as linear programming are not easily applied in a distributed and possibly asynchronous manner.

Unboundedness can occur in two ways. The first is that the set of variables and constraints involved in the problem is not bounded. This is the classical distributed constraint satisfaction problem, and it occurs for example in meeting scheduling. The second is that the admissible values and value tuples are unbounded. This is the *open* constraint satisfaction problem, and it occurs for example in product or supply chain configuration. This chapter has presented an overview of the main algorithms that have been developed for these scenarios.

Applications of distributed constraint satisfaction algorithms are just beginning to appear. Since many of these problems were impossible to solve by computer before, the

technology is in fact an enabler for future applications that are now beginning to be explored.

Bibliography

[1] M. Basharu, I. Arana and H. Ahriz: "Solving DisCSPs with Penalty Driven Search," *Proceedings of the 20th AAAI*, pp. 47-52, 2005

[2] R. Béjar, C. Domshlak, C. Fernández, C. Gomes, B. Krishnamachari, B. Selman, and M. Valls: "Sensor networks and distributed CSP: communication, computation and complexity," *Artificial Intelligence* **161**(1-2), pp. 117-147, 2005

[3] C. Bessière, A. Maestre, I. Brito and P. Meseguer: "Asynchronous backtracking without adding links: a new member in the ABT family," *Artificial Intelligence* **161**(1-2), pp. 7-24, 2005

[4] I. Brito and P. Meseguer: "Distributed Forward Checking," *Proceedings of the 9th CP*, Springer LNCS 2833, pp. 801-806, 2003

[5] Z. Collin, R. Dechter and S. Katz: "On the Feasibility of Distributed Constraint Satisfaction," *Proceedings of the 12th IJCAI*, pp. 319-324, Sydney, 1991

[6] R. Cucchiara, M. Gavanelli, E. Lamma, P. Mello, M. Milano, and M. Piccardi: "Constraint propagation and value acquisition: why we should do it interactively," *Proceedings of the 16th IJCAI*, pp.468-477, 1999

[7] R. Dechter: "Bucket elimination: A unifying framework for reasoning," *Artificial Intelligence* **113**, pp.41-85, 1999

[8] R. Dechter and I. Rish: "Minibuckets: A general scheme for approximating inference," *Journal of ACM*, pp. 107-153, 2003

[9] C. Eisenberg and B. Faltings: "Hybrid Solving Method for Large-Scale Distributed Constraint Satisfaction Problems, in W. Zhang and V. Sorge (eds.): *Distributed Problem Solving and Reasoning in Multi-agent Systems*, IOS Press, pp. 19-33, 2004

[10] E. Ephrati and J. S. Rosenschein: "The Clarke tax as a consensus mechanism among automated agents," *Proceedings of the 9th AAAI*, pp. 173-178, 1991

[11] B. Faltings: "Incentive-compatible Open Constraint Optimization," *Proceedings of the 4th ACM Conference on Electronic Commerce*, 2003

[12] B. Faltings and S. Macho-Gonzalez: "Open Constraint Programming," *Artificial Intelligence* **161**(1-2), pp. 181-208, 2005

[13] K. Hirayama, M. Yokoo, "Coordinated Multi-agent Local Search", *Artificial Intelligence* **161**(1-2), pp. 89-116, 2005

[14] R. Mailler and V. Lesser: "Solving Distributed Constraint Optimization Problems Using Cooperative Mediation," *Proceedings of the 3rd AAMAS*, pp. 438-445, 2004

[15] R. Marinescu and R. Dechter: "AND/OR Branch-and-Bound for Graphical Models," *Proceedings of the 19th IJCAI*, pp. 224-229, 2005

[16] A. Meisels, E. Kaplansky, I. Razgon and R. Zivan: "Comparing Performance of Distributed Constraints Processing Algorithms," *Proceedings of 3rd Workshop on Distributed Constraint Reasoning* pp. 86-93, 2002

[17] A. Meisels and R. Zivan: "Asynchronous Forward-Checking for DisCSPs," in W. Zhang and V. Sorge (eds.): *Distributed Problem Solving and Reasoning in Multi-agent Systems*, IOS Press, pp. 93-107, 2004

[18] P. J. Modi, M. Veloso: "Multiagent Meeting Scheduling with Rescheduling," *5th Workshop on Distributed Constraint Reasoning*, 2004

[19] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo: "An Asynchronous Complete Method for Distributed Constraint Optimization," *Artificial Intelligence* **161**(1-2), pp. 149-180, 2005

[20] V. Nguyen, D. Sam-Haroud and B. Faltings: "Dynamic Distributed Backjumping," *Recent Advances in Constraints*, Springer LNAI 3419, pp. 71-85, 2005

[21] D. C. Parkes and J. Shneidman: "Distributed implementations of Vickrey-Clarke-Groves mechanisms," *Proceedings of the 3rd AAMAS*, pp. 261-268, 2004

[22] A. Petcu and B. Faltings: "An Efficient Constraint Optimization Method for Large Multiagent Systems, " *AAMAS Workshop on large-scale multi-agent systems*, 2005.

[23] A. Petcu and B. Faltings: "Superstabilizing, Fault-containing Multiagent Combinatorial Optimization," *Proceedings of the 20th AAAI*, pp. 1406-1411, 2005

[24] A. Petcu and B. Faltings: "A Scalable Method for Multiagent Constraint Optimization," *Proceedings of the 19th ICJAI*, pp 266-271, 2005

[25] A. Petcu and B. Faltings: "Approximations in Distributed Optimization," *Proceedings of the 11th CP*, Springer LNCS 3709, pp. 802-806, 2005

[26] A. Petcu, B. Faltings and D. Parkes: "MDPOP: Faithful Distributed Implementation of Efficient Social Choice Problems," *Proceedings of the 5th AAMAS*, 2006

[27] A. Petcu and B. Faltings: "ODPOP: An Algorithm for Open Distributed Constraint Optimization," *AAMAS 06 Workshop on Distributed Constraint Reasoning*, 2006

[28] M. Silaghi, D. Sam-Haroud and B. Faltings: "Asynchronous Search with Aggregations," *Proceedings of the 17th AAAI*, pp. 917-922, 2000

[29] M. Silaghi, D. Sam-Haroud and B. Faltings: "ABT with Asynchronous Reordering," *Proceedings of 2nd A-P Conference on Intelligent Agent Technology* IEEE press, pp. 54-63, 2001

[30] M. Silaghi, D. Sam-Haroud and B. Faltings: "Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering," EPFL Technical Report 01/364, 2001

[31] M. Silaghi: "Asynchronously Solving Distributed Problems with Privacy Requirements," Ph.D. Thesis 2601, EPFL, 2002

[32] M. Silaghi: "Meeting Scheduling System Guaranteeing n/2-Privacy and Resistant to Statistical Analysis (Applicable to any DisCSP)," *Proceedings of the 3rd International Conference on Web Intelligence*, IEEE press, pp. 711-715, 2004

[33] M. Silaghi and B. Faltings: "Asynchronous Aggregation and Consistency in Distributed Constraint Satisfaction," *Artificial Intelligence* **161**(1-2), pp. 25-54, 2005

[34] M. Silaghi and M. Yokoo: "Nogood-based Asynchronous Distributed Optimization (ADOPT-ng)," *Proceedings of the 5th AAMAS*, 2006

[35] K. Sycara, S.F. Roth, N. Sadeh-Koniecpol, and M.S. Fox: "Distributed Constrained Heuristic Search," IEEE Transactions on Systems, Man, and Cybernetics, **21**(6), pp. 1446-1461, 1991

[36] R. J. Wallace and E. C. Freuder: "Constraint-based reasoning and privacy/efficiency tradeoffs in multi-agent program solving," *Artificial Intelligence* **161**(1-2), pp. 209-227, 2005

[37] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara: "Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving", *Proceedings of the 12th ICDCS*, pp.614-621, 1992.

[38] M. Yokoo: "Weak-commitment Search for Solving Constraint Satisfaction Problems", *Proceedings of the 12th AAAI*, pp.313–318, 1994.

[39] M. Yokoo, K. Hirayama: "Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems" *Proceedings of the 2nd ICMAS*, pp.401–408, 1996

[40] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara: "Distributed Constraint Satisfaction Problem: Formalization and Algorithms," *IEEE Trans. on Knowledge and Data Engineering* **10**(5), 1998

[41] M. Yokoo, K. Suzuki, and K. Hirayama, "Secure Distributed Constraint Satisfaction: Reaching Agreement without Revealing Private Information", *Artificial Intelligence* **161**(1-2), pp. 229-246, 2005

[42] R. Zivan and A. Meisels: "Dynamic Ordering for Asynchronous Backtracking on DisCSPs," *Proceedings of the 11th CP*, Springer LNCS 3709, pp. 32-46, 2005

[43] W. Zhang and L. Wittenburg: "Distributed breakout revisited," *Proceedings of the 18th AAAI*, pp.352-357, 2002

[44] W. Zhang, G. Wang, Z. Xing and L. Wittenberg: "Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks," *Artificial Intelligence* **161**(1-2), pp. 55-87, 2005