1

Distributed Constraint Satisfaction

In this chapter and the next we discuss cooperative situations in which agents collaborate to achieve a common goal. This goal can be viewed as shared between the agents or, alternatively, as the goal of a central designer who is designing the various agents. Of course, if such a designer exists, a natural question is why it matters that there are multiple agents; they can be viewed merely as end sensors and effectors for executing the plan devised by the designer. However, there exist situations in which a problem needs to be solved in a distributed fashion, either because a central controller is not feasible or because one wants to make good use of the distributed resources. A good example is provided by *sensor networks*. Such networks consist of multiple processing units, each with local sensor capabilities, limited processing power, limited power supply, and limited communication bandwidth. Despite these limitations, these networks aim to provide some global service. Figure 1.1 shows an example of a fielded sensor network used for monitoring environmental quantities like humidity, temperature and pressure in an office environment. Each sensor can monitor only its local area and, similarly, can communicate only with other sensors in its local vicinity. The question is what algorithm the individual sensors should run so that the center can still piece together a reliable global picture.

sensor network

Distributed algorithms have been widely studied in computer science. We concentrate on distributed problem-solving algorithms of the sort studied in artificial intelligence. We divide the discussion into two parts. In this chapter we cover distributed constraint satisfaction, where agents attempt in a distributed fashion to find a feasible solution to a problem with global constraints. In the next chapter we look at agents who try not only to satisfy constraints, but also to optimize some objective function subject to these constraints.

Later in this book we will encounter additional examples of distributed problem solving. Each of them requires specific background, however, which is why they are not discussed here. Two of them stand out in particular.

- In Chapter 7 we encounter a family of techniques that involve learning, some of them targeted at purely cooperative situations. In these situations the agents learn through repeated interactions how to coordinate a choice of action. This material requires some discussion of noncooperative game theory (discussed in Chapter 3) as well as general discussion of multiagent learning (discussed in Chapter 7).

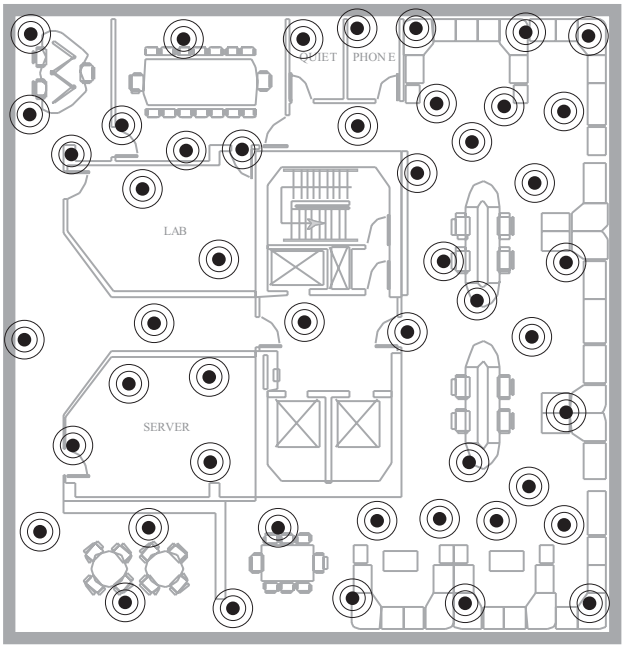

Figure 1.1 Part of a real sensor network used for indoor environmental monitoring.

- In Chapter 13 we discuss the use of logics of knowledge (introduced in that chapter) to establish the knowledge conditions required for coordination, including an application to distributed control of multiple robots.

1.1 Defining distributed constraint satisfaction problems

constraint satisfaction problem (CSP)

A *constraint satisfaction problem (CSP)* is defined by a set of variables, domains for each of the variables, and constraints on the values that the variables might take on simultaneously. The role of constraint satisfaction algorithms is to assign values to the variables in a way that is consistent with all the constraints, or to determine that no such assignment exists.

Constraint satisfaction techniques have been applied in diverse domains, including machine vision, natural language processing, theorem proving, and planning and scheduling, to name but a few. Here is a simple example taken from the domain of sensor networks. Figure 1.2 depicts a three-sensor snippet from the scenario illustrated in Figure 1.1. Each of the sensors has a certain radius that, in combination with the obstacles in the environment, gives rise to a particular coverage area. These coverage areas are shown as ellipses in Figure 1.2. As you can see, some of the coverage areas overlap. We consider a specific problem in this setting. Suppose that each sensor can choose one of three possible radio frequencies. All the frequencies work equally well so long as no two sensors with overlapping coverage areas use the same frequency. The question is which

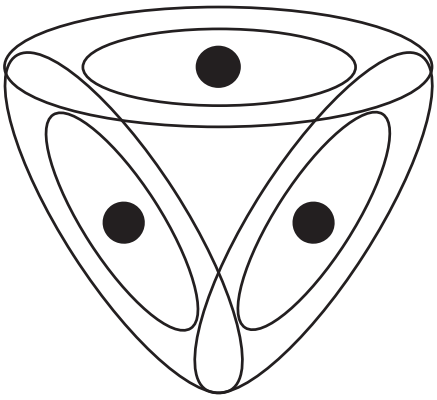

Figure 1.2 A simple sensor net problem.

algorithms the sensors should employ to select their frequencies, assuming that this decision cannot be made centrally.

The essence of this problem can be captured as a graph-coloring problem. Figure 1.3 shows such a graph, corresponding to the sensor network CSP above. The nodes represent the individual units; the different frequencies are represented by colors; and two nodes are connected by an undirected edge if and only if the coverage areas of the corresponding sensors overlap. The goal of graph coloring is to choose one color for each node so that no two adjacent nodes have the same color.

Formally speaking, a CSP consists of a finite set of variables $X = \{X_1, \ldots, X_n\}$, a domain D_i for each variable X_i, and a set of constraints $\{C_1, \ldots, C_m\}$. Although in general CSPs allow infinite domains, we assume here that all the domains are finite. In the graph-coloring example above there were three variables, and they each had the same domain, {*red*, *green*, *blue*}. Each constraint is a predicate on some subset of the variables, say, $X_{i_1}, \ldots, X_{i_j}$; the predicate defines a relation that is a subset of the Cartesian product $D_{i_1} \times \cdots \times D_{i_j}$. Each such constraint restricts the values that may be simultaneously assigned to the variables participating in the constraint. In this chapter we restrict the discussion to

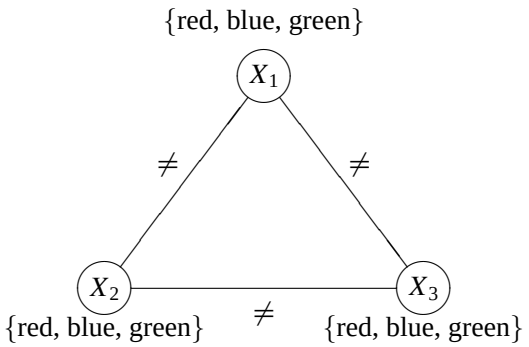

Figure 1.3 A graph-coloring problem equivalent to the sensor net problem of Figure 1.2.

binary constraints, each of which constrains exactly two variables. For example, in the map-coloring case, each "not-equal" constraint applied to two nodes.

Given a subset S of the variables, an *instantiation of* S is an assignment of a unique domain value for each variable in S; it is *legal* if it does not violate any constraint that mentions only variables in S. A *solution* to a network is a legal instantiation of all variables. Typical tasks associated with constraint networks are to determine whether a solution exists, to find one or all solutions, to determine whether a legal instantiation of some of the variables can be extended to a solution, and so on. We will concentrate on the most common task, which is to find one solution to a CSP, or to prove that none exists.

In a *distributed* CSP, each variable is owned by a different agent. The goal is still to find a global variable assignment that meets the constraints, but each agent decides on the value of his own variable with relative autonomy. While he does not have a global view, each agent can communicate with his neighbors in the constraint graph. A distributed algorithm for solving a CSP has each agent engage in some protocol that combines local computation with communication with his neighbors. A good algorithm ensures that such a process terminates with a legal solution (or with a realization that no legal solution exists) and does so quickly.

We discuss two types of algorithms. Algorithms of the first kind embody a least-commitment approach and attempt to rule out impossible variable values without losing any possible solutions. Algorithms of the second kind embody a more adventurous spirit and select tentative variable values, backtracking when those choices prove unsuccessful. In both cases we assume that the communication between neighboring nodes is perfect, but nothing about its timing; messages can take more or less time without rhyme or reason. We do assume, however, that if node i sends multiple messages to node j, those messages arrive in the order in which they were sent.

1.2 Domain-pruning algorithms

Under domain-pruning algorithms, nodes communicate with their neighbors in order to eliminate values from their domains. We consider two such algorithms.

filtering algorithm

In the first, the *filtering algorithm*, each node communicates its domain to its neighbors, eliminates from its domain the values that are not consistent with the values received from the neighbors, and the process repeats. Specifically, each node x_i with domain D_i repeatedly executes the procedure **Revise**(x_i, x_j) for each neighbor x_j.

procedure Revise(x_i, x_j)
forall $v_i \in D_i$ **do**
  if *there is no value* $v_j \in D_j$ *such that* v_i *is consistent with* v_j **then**
    delete v_i from D_i

arc consistency

The process, known also under the general term *arc consistency*, terminates when no further elimination takes place, or when one of the domains becomes

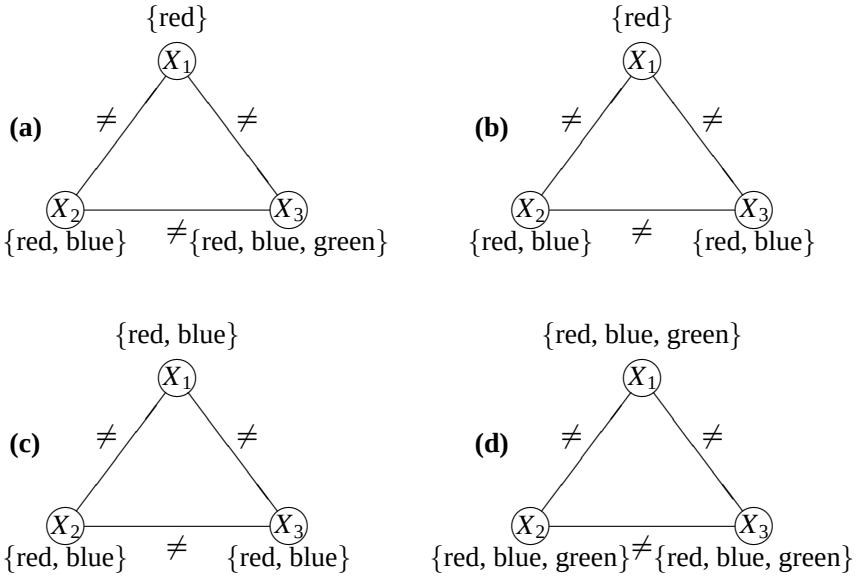

Figure 1.4 A family of graph coloring problems

empty (in which case the problem has no solution). If the process terminates with one value in each domain, that set of values constitutes a solution. If it terminates with multiple values in each domain, the result is inconclusive; the problem might or might not have a solution.

Clearly, the algorithm is guaranteed to terminate, and furthermore it is sound (in that if it announces a solution, or announces that no solution exists, it is correct), but it is not complete (i.e., it may fail to pronounce a verdict). Consider, for example, the family of very simple graph-coloring problems shown in Figure 1.4. (Note that problem (d) is identical to the problem in Figure 1.3.)

In this family of CSPs the three variables (i.e., nodes) are fixed, as are the "not-equal" constraints between them. What are not fixed are the domains of the variables. Consider the four instances of Figure 1.4.

(a) Initially, as the nodes communicate with one another, only x_1's messages result in any change. Specifically, when either x_2 or x_3 receive x_1's message they remove *red* from their domains, ending up with $D_2 = \{blue\}$ and $D_3 = \{blue, green\}$. Then, when x_2 communicates his new domain to x_3, x_3 further reduces his domain to $\{green\}$. At this point no further changes take place and the algorithm terminates with a correct solution.

(b) The algorithm starts as before, but once x_2 and x_3 receive x_1's message they each reduce their domains to $\{blue\}$. Now, when they update each other on their new domains, they each reduce their domains to {}, the empty set. At this point the algorithm terminates and correctly announces that no solution exists.

(c) In this case the initial set of messages yields no reduction in any domain. The algorithm terminates, but all the nodes have multiple values remaining.

And so the algorithm is not able to show that the problem is overconstrained and has no solution.

(d) Filtering can also fail when a solution exists. For similar reasons as in instance (c), the algorithm is unable to show that in this case the problem *does* have a solution.

In general, filtering is a very weak method and, at best, is used as a preprocessing step for more sophisticated methods. The algorithm is directly based on the notion of *unit resolution* from propositional logic. Unit resolution is the following inference rule:

unit resolution

$$\frac{\begin{array}{l} A_1 \\ \neg(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \end{array}}{\neg(A_2 \wedge \cdots \wedge A_n)}$$

To see how the filtering algorithm corresponds to unit resolution, we must first write the constraints as forbidden value combinations, called *Nogoods*. For example, the constraint that x_1 and x_2 cannot both take the value "red" would give rise to the propositional sentence $\neg(x_1 = red \wedge x_2 = red)$, which we write as the Nogood $\{x_1, x_2\}$. In instance (b) of Figure 1.4, agent X_2 updated his domain based on agent X_1's announcement that $x_1 = red$ and the Nogood $\{x_1 = red, x_2 = red\}$.

Nogood

$$\frac{\begin{array}{l} x_1 = red \\ \neg(x_1 = red \wedge x_2 = red) \end{array}}{\neg(x_2 = red)}$$

Unit resolution is a weak inference rule, and so it is not surprising that the filtering algorithm is weak as well. *Hyper-resolution* is a generalization of unit resolution and has the following form:

hyper-resolution

$$\frac{\begin{array}{l} A_1 \vee A_2 \vee \cdots \vee A_m \\ \neg(A_1 \wedge A_{1,1} \wedge A_{1,2} \wedge \cdots) \\ \neg(A_2 \wedge A_{2,1} \wedge A_{2,2} \wedge \cdots) \\ \qquad\qquad \vdots \\ \neg(A_m \wedge A_{m,1} \wedge A_{m,2} \wedge \cdots) \end{array}}{\neg(A_{1,1} \wedge \cdots \wedge A_{2,1} \wedge \cdots \wedge A_{m,1} \wedge \cdots)}$$

Hyper-resolution is both sound and complete for propositional logic, and indeed it gives rise to a complete distributed CSP algorithm. In this algorithm, each agent repeatedly generates new constraints for his neighbors, notifies them of these new constraints, and prunes his own domain based on new constraints passed to him by his neighbors. Specifically, he executes the following algorithm, where NG_i is the set of all Nogoods of which agent i is aware and NG_j^* is a set of new Nogoods communicated from agent j to agent i.

procedure ReviseHR(NG_i, NG_j^*)
repeat
  $NG_i \leftarrow NG_i \bigcup NG_j^*$
  let NG_i^* denote the set of new Nogoods that i can derive from NG_i and his domain using hyper-resolution
  if NG_i^* *is nonempty* **then**
    $NG_i \leftarrow NG_i \bigcup NG_i^*$
    send the Nogoods NG_i^* to all neighbors of i
    if $\{\} \in NG_i^*$ **then**
      stop
until *there is no change in i's set of Nogoods* NG_i

The algorithm is guaranteed to converge in the sense that after sending and receiving a finite number of messages, each agent will stop sending messages and generating Nogoods. Furthermore, the algorithm is complete. The problem has a solution iff, on completion, no agent has generated the empty Nogood. (Obviously, every superset of a Nogood is also forbidden, and thus if a single node ever generates an empty Nogood then the problem has no solution.)

Consider again instance (c) of the CSP problem in Figure 1.4. In contrast to the filtering algorithm, the hyper-resolution-based algorithm proceeds as follows. Initially, x_1 maintains four Nogoods—$\{x_1 = red, x_2 = red\}$, $\{x_1 = red, x_3 = red\}$, $\{x_1 = blue, x_2 = blue\}$, $\{x_1 = blue, x_3 = blue\}$—which are derived directly from the constraints involving x_1. Furthermore, x_1 must adopt one of the values in his domain, so $x_1 = red \vee x_1 = blue$. Using hyper-resolution, x_1 can reason:

$$\begin{array}{l} x_1 = red \vee x_1 = blue \\ \neg(x_1 = red \wedge x_2 = red) \\ \neg(x_1 = blue \wedge x_3 = blue) \\ \hline \neg(x_2 = red \wedge x_3 = blue) \end{array}$$

Thus, x_1 constructs the new Nogood $\{x_2 = red, x_3 = blue\}$; in a similar way he can also construct the Nogood $\{x_2 = blue, x_3 = red\}$. x_1 then sends both Nogoods to his neighbors x_2 and x_3. Using his domain, an existing Nogood and one of these new Nogoods, x_2 can reason:

$$\begin{array}{l} x_2 = red \vee x_2 = blue \\ \neg(x_2 = red \wedge x_3 = blue) \\ \neg(x_2 = blue \wedge x_3 = blue) \\ \hline \neg(x_3 = blue) \end{array}$$

Using the other new Nogood from x_1, x_2 can also construct the Nogood $\{x_3 = red\}$. These two singleton Nogoods are communicated to x_3 and allow him to generate the empty Nogood. This proves that the problem does not have a solution.

This example, while demonstrating the greater power of the hyper-resolution-based algorithm relative to the filtering algorithm, also exposes its weakness; the number of Nogoods generated can grow to be unmanageably large. (Indeed, we only described the minimal number of Nogoods needed to derive the empty Nogood; many others would be created as all the agents processed each other's messages in parallel. Can you find an example?) Thus, the situation in which we find ourselves is that we have one algorithm that is too weak and another that is impractical. The problem lies in the least-commitment nature of these algorithms; they are restricted to removing only provably impossible value combinations. The alternative to such "safe" procedures is to explore a subset of the space, making tentative value selections for variables, and backtracking when necessary. This is the topic of the next section. However, the algorithms we have just described are not irrelevant; the filtering algorithm is an effective preprocessing step, and the algorithm we discuss next is based on the hyper-resolution-based algorithm.

1.3 Heuristic search algorithms

A straightforward *centralized* trial-and-error solution to a CSP is to first order the variables (e.g., alphabetically). Then, given the ordering $x_1, x_2, \ldots, x_n$, invoke the procedure ChooseValue(x_1, {}). The procedure ChooseValue is defined recursively as follows, where $\{v_1, v_2, \ldots, v_{i-1}\}$ is the set of values assigned to variables $x_1, \ldots, x_{i-1}$.

procedure ChooseValue(x_i, $\{v_1, v_2, \ldots, v_{i-1}\}$)
$v_i \leftarrow$ value from the domain of x_i that is consistent with $\{v_1, v_2, \ldots, v_{i-1}\}$
if *no such value exists* **then**
  backtrack[1]
else if $i = n$ **then**
  stop
else
  ChooseValue(x_{i+1}, $\{v_1, v_2, \ldots, v_i\}$)

chronological backtracking

This exhaustive search of the space of assignments has the advantage of completeness. But it is "distributed" only in the uninteresting sense that the different agents execute sequentially, mimicking the execution of a centralized algorithm.

The following attempt at a distributed algorithm has the opposite properties; it allows the agents to execute in parallel and asynchronously, is sound, but is not complete. Consider the following naive procedure, executed by all agents in parallel and asynchronously.

1. There are various ways to implement the backtracking in this procedure. The most straightforward way is to undo the choices made thus far in reverse chronological order, a procedure known as *chronological backtracking*. It is well known that more sophisticated backtracking procedures can be more efficient, but that does not concern us here.

```
select a value from your domain
repeat
    if your current value is consistent with the current values of your
    neighbors, or if none of the values in your domain are consistent with
    them then
        do nothing
    else
        select a value in your domain that is consistent with those of your
        neighbors and notify your neighbors of your new value
until there is no change in your value
```

Clearly, when the algorithm terminates because no constraint violations have occurred, a solution has been found. But in all other cases, all bets are off. If the algorithm terminates because no agent can find a value consistent with those of his neighbors, there might still be a consistent global assignment. And the algorithm may never terminate even if there is a solution. For example, consider example (d) of Figure 1.4: if every agent cycles sequentially between red, green, and blue, the algorithm will never terminate.

We have given these two straw-man algorithms for two reasons. Our first reason is to show that reconciling true parallelism and asynchrony with soundness and completeness is likely to require somewhat complex algorithms. And second, the fundamental heuristic algorithm for distributed CSPs—the asynchronous backtracking (or ABT) algorithm—shares much with the two algorithms. From the first algorithm it borrows the notion of a global total ordering on the agents. From the second it borrows a message-passing protocol, albeit a more complex one, which relies on the global ordering. We will describe the ABT in its simplest form. After demonstrating it on an extended example, we will point to ways in which it can be improved upon.

ABT algorithm

1.3.1 *The asynchronous backtracking algorithm*

As we said, the asynchronous backtracking (ABT) algorithm assumes a total ordering (the "priority order") on the agents. Each binary constraint is known to both the constrained agents and is checked in the algorithm by the agent with the lower priority between the two. A link in the constraint network is always directed from an agent with higher priority to an agent with lower priority.

Agents instantiate their variables concurrently and send their assigned values to the agents that are connected to them by outgoing links. All agents wait for and respond to messages. After each update of his assignment, an agent sends his new assignment along all outgoing links. An agent who receives an assignment (from the higher-priority agent of the link), tries to find an assignment for his variable that does not violate a constraint with the assignment he received.

ok? messages are messages carrying an agent's variable assignment. When an agent A_i receives an **ok?** message from agent A_j, A_i places the received assignment in a data structure called *agent_view*, which holds the last assignment A_i

received from higher-priority neighbors such as A_j. Next, A_i checks if his current assignment is still consistent with his *agent_view*. If it is consistent, A_i does nothing. If not, then A_i searches his domain for a new consistent value. If he finds one, he assigns his variable that value and sends **ok?** messages to all lower-priority agents linked to him informing them of this value. Otherwise, A_i backtracks.

The *backtrack* operation is executed by sending a Nogood message. Recall that a Nogood is simply an inconsistent partial assignment, that is, assignments of specific values to some of the variables that together violate the constraints on those variables. In this case, the Nogood consists of A_i's *agent_view*.[2] The Nogood is sent to the agent with the lowest priority among the agents whose assignments are included in the inconsistent tuple in the Nogood. Agent A_i who sends a Nogood message to agent A_j assumes that A_j will change his assignment. Therefore, A_i removes from his *agent_view* the assignment of A_j and makes an attempt to find an assignment for A_j's variable that is consistent with the updated *agent_view*.

Because of its reliance on building up a set of Nogoods, the ABT algorithm can be seen as a greedy version of the hyper-resolution algorithm of the previous section. In the latter, all possible Nogoods are generated by each agent and communicated to all neighbors, even though the vast majority of these messages are not useful. Here, agents make tentative choices of a value for their variables, only generate Nogoods that incorporate values already generated by the agents above them in the order, and—importantly—communicate new values only to some agents and new Nogoods to only one agent.

Below is the pseudocode of the ABT algorithm, specifying the protocol for agent A_i.

when *received (**Ok?**, (A_j, d_j))* **do**
  add (A_j, d_j) to *agent_view*
  check_agent_view
when *received (**Nogood**,* nogood*)* **do**
  add *nogood* to Nogood list
  forall *(A_k, d_k) ∈* nogood*, if A_k is not a neighbor of A_i* **do**
    add (A_k, d_k) to *agent_view*
    request A_k to add A_i as a neighbor
  check_agent_view

procedure check_agent_view
when agent_view *and* current_value *are inconsistent* **do**
  if *no value in D_i is consistent with* agent_view **then**
    backtrack
  else
    select $d \in D_i$ consistent with *agent_view*
    current_value $\leftarrow d$
    send (**ok?**, (A_i, d)) to lower-priority neighbors

2. We later discuss schemes that achieve better performance by avoiding always sending this entire set.

procedure backtrack
nogood ← some inconsistent set, using hyper-resolution or similar procedure
if nogood *is the empty set* **then**
    broadcast to other agents that there is no solution
    terminate this algorithm
else
    select $(A_j, d_j) \in$ *nogood* where A_j has the lowest priority in *nogood*
    send (**Nogood**, *nogood*) to A_j
    remove (A_j, d_j) from *agent_view*
    check_agent_view

Notice a certain wrinkle in the pseudocode, having to do with the addition of edges. Since the Nogood can include assignments of some agent A_j, which A_i was not previously constrained with, after adding A_j's assignment to its *agent_view* A_i sends a message to A_j asking it to add A_i to its list of outgoing links. Furthermore, after adding the link, A_j sends an **ok?** message to A_i each time it reassigns its variable. After storing the Nogood, A_i checks if its assignment is still consistent. If it is, a message is sent to the agent the Nogood was received from. This resending of the assignment is crucial since, as mentioned earlier, the agent sending a Nogood assumes that the receiver of the Nogood replaces its assignment. Therefore it needs to know that the assignment is still valid. If the old assignment that was forbidden by the Nogood is inconsistent, A_i tries to find a new assignment similarly to the case when an **ok?** message is received.

1.3.2 *A simple example*

In Section 1.3.3 we give a more elaborate example, but here is a brief illustration of the operation of the ABT algorithm on one of the simple problems encountered earlier. Consider again the instance (c) of the CSP in Figure 1.4, and assume the agents are ordered alphabetically: x_1, x_2, x_3. They initially select values at random; suppose they all select *blue*. x_1 notifies x_2 and x_3 of his choice, and x_2 notifies x_3. x_2's local view is thus $\{x_1 = blue\}$, and x_3's local view is $\{x_1 = blue, x_2 = blue\}$. x_2 and x_3 must check for consistency of their local views with their own values. x_2 detects the conflict, changes his own value to *red*, and notifies x_3. In the meantime, x_3 also checks for consistency and similarly changes his value to *red*; he, however, notifies no one. Then x_3 receives a second message from x_2, and updates his local view to $\{x_1 = blue, x_2 = red\}$. At this point he cannot find a value from his domain consistent with his local view, and, using hyper resolution, generates the Nogood $\{x_1 = blue, x_2 = red\}$. He communicates this Nogood to x_2, the lowest ranked agent participating in the Nogood. x_2 now cannot find a value consistent with his local view, generates the Nogood $\{x_1 = blue\}$, and communicates it to x_1. x_1 detects the inconsistency with his current value, changes his value to *red*, and communicates the new value to x_2 and x_3. The process now continues as before; x_2 changes his value back to *blue*, x_3 finds no consistent

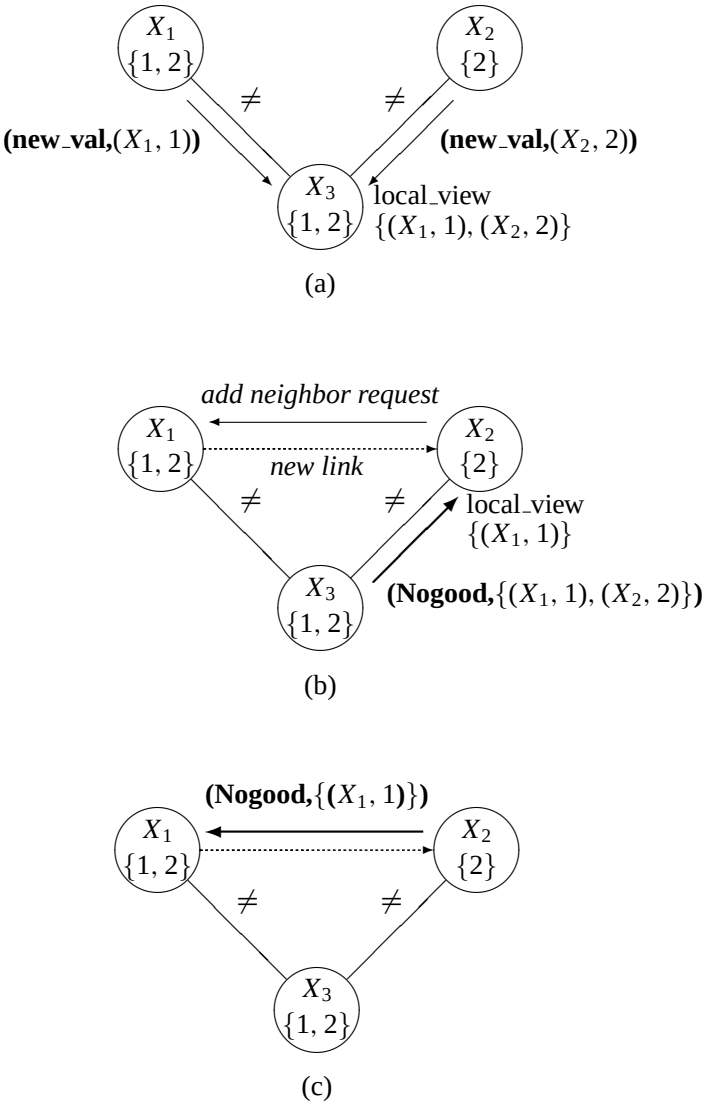

Figure 1.5 Asynchronous backtracking with dynamic link addition.

value and generates the Nogood $\{x_1 = red, x_2 = blue\}$, and then x_2 generates the Nogood $\{x_1 = red\}$. At this point x_1 has the Nogood $\{x_1 = blue\}$ as well as the Nogood $\{x_1 = red\}$, and using hyper-resolution he generates the Nogood $\{\}$, and the algorithm terminates having determined that the problem has no solution.

The need for the addition of new edges is seen in a slightly modified example, shown in Figure 1.5.

As in the previous example, here too x_3 generates the Nogood $\{x_1 = blue, x_2 = red\}$ and notifies x_2. x_2 is not able to regain consistency by changing his own value. However, x_1 is not a neighbor of x_2, and so x_2 does not have the value $x_1 = blue$ in his local view and is not able to send the Nogood $\{x_1 = blue\}$ to x_1. So x_2 sends

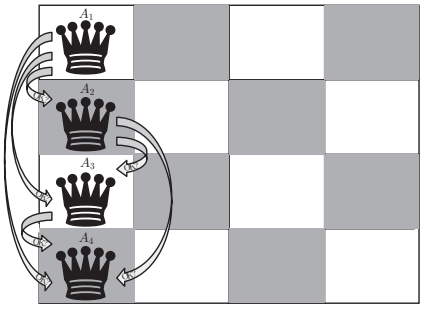

Figure 1.6 Cycle 1 of ABT for four queens. All agents are active.

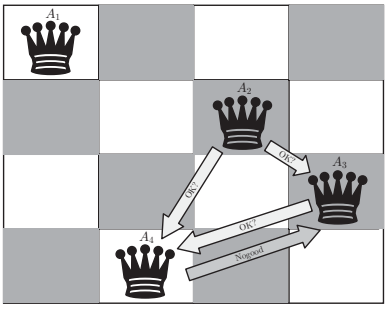

Figure 1.7 Cycle 2 of ABT for four queens. A_2, A_3, and A_4 are active. The Nogood message is $A_1 = 1 \wedge A_2 = 1 \rightarrow A_3 \neq 1$.

a request to x_1 to add x_2 to its list of neighbors and to send x_2 his current value. From there onward the algorithm proceeds as before.

1.3.3 *An extended example: the four queens problem*

In order to gain additional feeling for the ABT algorithm beyond the didactic example in the previous section, let us look at one of the canonical CSP problems: the n-queens problem. More specifically, we will consider the four queens problem, which asks how four queens can be placed on a 4×4 chessboard so that no queen can (immediately) attack any other. We will describe ABT's behavior in terms of cycles of computation, which we somewhat artificially define to be the receiving of messages, the computations triggered by received messages, and the sending of messages due to these computations.

In the first cycle (Figure 1.6) all agents select values for their variables, which represent the positions of their queens along their respective rows. Arbitrarily, we assume that each begins by positioning his queen at the first square of his row. Each agent 1, 2, and 3 sends **ok?** messages to the agents ordered after him: A_1 sends three messages, A_2 sends two, and agent A_3 sends a single message. Agent A_4 does not have any agent after him, so he sends no messages. All agents are active in this first cycle of the algorithm's run.

In the second cycle (Figure 1.7) agents A_2, A_3, and A_4 receive the **ok?** messages sent to them and proceed to assign consistent values to their variables. Agent A_3 assigns the value 3 that is consistent with the assignments of A_1 and A_2 that he receives. Agent A_4 has no value consistent with the assignments of A_1, A_2, and A_3, and so he sends a *Nogood* containing these three assignments to A_3 and removes the assignment of A_3 from his *Agent_view*. Then, he assigns the value 2 which is consistent with the assignments that he received from A_1 and A_2 (having erased the assignment of A_3, assuming that it will be replaced because of the Nogood message). The active agents in this cycle are A_2, A_3, and A_4. Agent A_2 acts according to his information about A_1's position and moves to square 3, sending two **ok?** messages to inform his successors about his value. As can be seen in Figure 1.7, A_3 has moved to square 4 after receiving the **ok?**

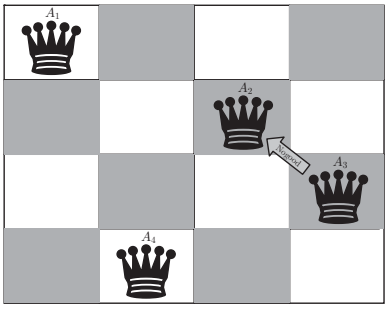

Figure 1.8 Cycle 3. Only A_3 is active. The Nogood message is $A_1 = 1 \rightarrow A_2 \neq 3$.

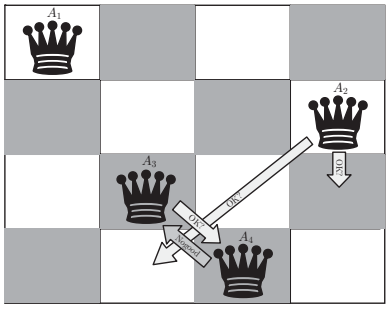

Figure 1.9 Cycles 4 and 5. A_2, A_3, and A_4 are active. The Nogood message is $A_1 = 1 \wedge A_2 = 4 \rightarrow A_3 \neq 4$.

messages of agents A_1 and A_2. Note that agent A_3 thinks that these agents are still in the first column of their respective rows. This is a manifestation of concurrency that causes each agent to act at all times in a form that is based only on his *Agent_View*. The *Agent_view* of agent A_3 includes the **ok?** messages he received.

The third cycle is described in Figure 1.8; only A_3 is active. After receiving the assignment of agent A_2, A_3 sends back a Nogood message to agent A_2. He then erases the assignment of agent A_2 from his *Agent_view* and validates that his current assignment (the value 4) is consistent with the assignment of agent A_1. Agents A_1 and A_2 continue to be idle, having received no messages that were sent in cycle 2. The same is true for agent A_4. Agent A_3 also receives the Nogood sent by A_4 in cycle 2 but ignores it since it includes an invalid assignment for A_2 (i.e., (2, 1) and not the currently correct (2, 4)).

Cycles 4 and 5 are depicted in Figure 1.9. In cycle 4 agent A_2 moves to square 4 because of the Nogood message he received. His former value was ruled out and the new value is the next valid one. He informs his successors A_3 and A_4 of his new position by sending two **ok?** messages. In cycle 5 agent A_3 receives agent A_2's new position and selects the only value that is compatible with the positions of his two predecessors, square 2. He sends a message to his successor informing him about this new value. Agent A_4 is now left with no valid value to assign and sends a Nogood message to A_3 that includes all his conflicts. The Nogood message appears at the bottom of Figure 1.9. Note that the Nogood message is no longer valid. Agent A_4, however, assumes that A_3 will change his position and moves to his only valid position (given A_3's anticipated move)—column 3.

Consider now cycle 6. Agent A_4 receives the new assignment of agent A_3 and sends him a Nogood message. Having erased the assignment of A_3 after sending the Nogood message, he then decides to stay at his current assignment (column 3), since it is compatible with the positions of agents A_1 and A_2. Agent A_3 is idle in cycle 6, since he receives no messages from either agent A_1 or agent A_2 (who are idle too). So, A_4 is the only active agent at cycle 6 (see Figure 1.10).

In each of cycles 7 and 8, one Nogood is sent. Both are depicted in Figure 1.11. First, agent A_3, after receiving the Nogood message from A_4, finds that he has

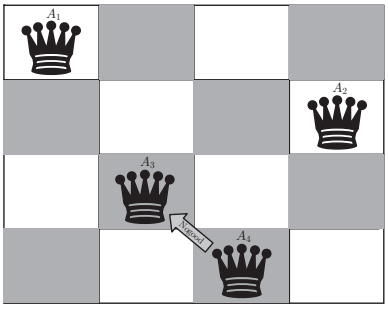

Figure 1.10 Cycle 6. Only A_4 is active. The Nogood message is $A_1 = 1 \wedge A_2 = 4 \rightarrow A_3 \neq 2$.

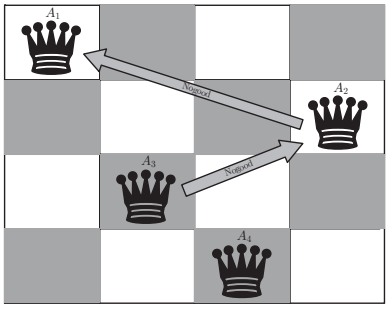

Figure 1.11 Cycles 7 and 8. A_3 is active in the first cycle and A_2 is active in the second. The Nogood messages are $A_1 = 1 \rightarrow A_2 \neq 4$ and $A_1 \neq 1$.

no valid values left and sends a Nogood to A_2. Next, in cycle 8, agent A_2 also discovers that his domain of values is exhausted and sends a Nogood message to A_1. Both sending agents erase the values of their successors (to whom the Nogood messages were sent) from their *agent_views* and therefore remain in their positions, which are now conflict free.

Cycle 9 involves only agent A_1, who receives the Nogood message from A_2 and so moves to his next value—square 2. Next, he sends **ok?** messages to his three successors.

The final cycle is cycle 10. Agent A_3 receives the **ok?** message of A_1 and so moves to a consistent value—square 1 of his row. Agents A_2 and A_4 check their *Agent_view*s after receiving the same **ok?** messages from agent A_1 and find that their current values are consistent with the new position of A_1. Agent A_3 sends an **ok?** message to his successor A_4, informing of his move, but A_4 finds no reason to move. His value is consistent with all value assignments of all his predecessors. After cycle 10 all agents remain idle, having no constraint violations with assignments on their *agent_views*. Thus, this is a final state of the ABT algorithm in which it finds a solution.

1.3.4 *Beyond the ABT algorithm*

The ABT algorithm is the backbone of modern approaches to distributed constraint satisfaction, but it admits many extensions and modifications.

A major modification has to do with which inconsistent partial assignment (i.e., Nogood) is sent in the backtrack message. In the version presented earlier, which is the early version of ABT, the full *agent_view* is sent. However, the full *agent_view* is in many cases not a minimal Nogood; a strict subset of it may also be inconsistent. In general, shorter Nogoods can lead to a more efficient search process, since they permit backjumping further up the search tree.

Here is an example. Consider an agent A_6 holding an inconsistent *agent_view* with the assignments of agents A_1, A_2, A_3, A_4 and A_5. If we assume that A_6 is

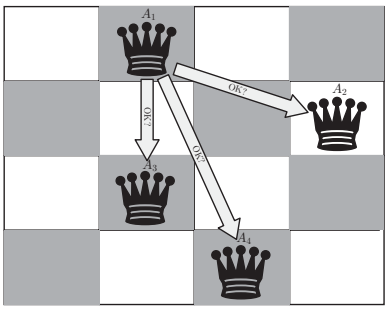

Figure 1.12 Cycle 9. Only A_1 is active.

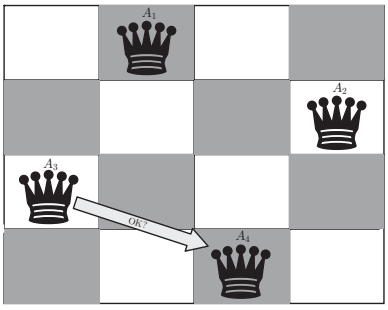

Figure 1.13 Cycle 10. Only A_3 is active.

only constrained by the current assignments of A_1 and A_3, sending a Nogood message to A_5 that contains all the assignments in the *agent_view* seems to be a waste. After sending the Nogood to A_5, A_6 will remove his assignment from the *agent_view* and make another attempt to assign his variable, which will be followed by an additional Nogood sent to A_4 and the removal of A_4's assignment from the *agent_view*. These attempts will continue until a minimal subset is sent as a Nogood. In this example, it is the Nogood sent to A_3. The assignment with the lower priority in the minimal inconsistent subset is removed from the *agent_view* and a consistent assignment can now be found. In this example the computation ended by sending a Nogood to the culprit agent, which would have been the outcome if the agent computed a minimal subset.

The solution to this inefficiency, however, is not straightforward, since finding a minimal Nogood is in general intractable (specifically, NP-hard). And so various heuristics are needed to cut down on the size of the Nogood, without sacrificing correctness.

A related issue is the number of Nogoods stored by each agent. In the preceding ABT version, each Nogood is recorded by the receiving agent. Since the number of inconsistent subsets can be exponential, constraint lists with exponential size will be created, and a search through such lists requires exponential time in the worst case. Various proposals have been made to cut down on this number while preserving correctness. One proposal is that agents keep only Nogoods consistent with their *agent_view*. While this prunes some of the Nogoods, in the worst case it still leaves a number of Nogoods that is exponential in the size of the *agent_view*. A further improvement is to store only Nogoods that are consistent with both the agent's *agent_view* and his current assignment. This approach, which is considered by some the best implementation of the ABT algorithm, ensures that the number of Nogoods stored by any single agent is no larger than the size of the domain.

asynchronous forward checking

concurrent dynamic backtracking

Finally, there are approaches to distributed constraint satisfaction that do not follow the ABT scheme, including *asynchronous forward checking* and *concurrent dynamic backtracking*. Discussion of them is beyond the scope of this book, but the references point to further reading on the topic.

1.4 History and references

Distributed constraint satisfaction is discussed in detail in Yokoo [2001], and reviewed in Yokoo and Hirayama [2000]. The ABT algorithm was initially introduced in Yokoo [1994]. More comprehensive treatments, including the latest insights into distributed CSPs, appear in Meisels [2008] and Faltings [2006]. The sensor net figure is due to Carlos Guestrin.