Chapter 3

Distributed Constraint Satisfaction

In recent years, an increasing interest has arisen about problems where information is distributed among different computers. If this information cannot be centralized in a single computer, the classical CSP model is inadequate for these problems, because it assumes centralized solving. Distributed Constraint Satisfaction (DisCSP) consider constraint problems, where knowledge (i.e., domains, variables and constraints) is distributed among communicating agents and cannot be centralized for different reasons (i.e. prohibitive costs of constraint translation or security/privacy issues).

This chapter gives an overview of existing research DisCSP. First, we formally introduce the Distributed Constraint Satisfaction problem (Section 3.1) and provide some examples (Section 3.2). Then, we describe a set of solving methods for DisCSP (Section 3.3). We also discuss some issues to evaluate (Section 3.4) and implement (Section 3.5) DisCSP solving algorithms.

3.1 What is a Distributed Constraint Satisfaction Problem?

A distributed CSP (DisCSP) is a CSP whose variables, domains and constraints are distributed among automated agents. There exit two distributed models for representing DisCSPs: the variable-based model [YDIK92] and the constraint-based model [SSHF00].

Definition 3.1.1. A variable-based model is a distributed model in which each variable belongs to one agent and constraints are shared between agents.

Definition 3.1.2. A constraint-based model is a distributed model in which each constraint belongs to one agent and shared variables in two constraints not belonging to the same agent are duplicated.

In this thesis, we focus on problems that are modeled using the variable-based model, which is the most frequently found in the related literature. According to the variable-based model, the formal definition of a finite DisCSP is as follows.
Definition 3.1.3. A finite DisCSP is defined by a 5-tuple \((\mathcal{X}, D, C, A, \phi)\), where \(\mathcal{X}\), \(D\) and \(C\) are the same as in CSP (see Definition 2.1.1 in Chapter 2), and

- \(\mathcal{A} = \{1, \ldots, p\}\) is a set of \(p\) agents,
- \(\phi : \mathcal{X} \to \mathcal{A}\) is a function that maps each variable to its agent.

Each variable belongs to one agent. The distribution of variables divides \(C\) in two disjointed subsets, \(C_{\text{intra}} = \{c_{ij} | \phi(x_i) = \phi(x_j)\}\), and \(C_{\text{inter}} = \{c_{ij} | \phi(x_i) \neq \phi(x_j)\}\), called intra-agent and inter-agent constraint sets, respectively. An intra-agent constraint \(c_{ij}\) is known by the agent owner of \(x_i\) and \(x_j\), but it is unknown by the other agents. Usually, an inter-agent constraint \(c_{ij}\) is known by the agents \(\phi(x_i)\) and \(\phi(x_j)\) [YDIK98, HBQ98].

Similar to CSP, a solution of a DisCSP is an assignment of values to variables satisfying every constraint (although DisCSP literature focuses mainly on solving inter-agent constraints). Distributed CSPs are solved by the collective and coordinated action of agents \(\mathcal{A}\). We assume the following communication model ([Yok01]):

- agents communicate by sending messages;
- an agent can send messages to other agents if it knows their addresses;
- the delay in delivering a message is finite, though random;
- for a given pair of agents, messages are delivered in the ordering they were sent.

From now on, we identify the agent number with its variable index \((\forall x_i \in \mathcal{X}, \phi(x_i) = i)\) in DisCSPs where every agent owns exactly one variable. In this situation, all constraints are inter-agent constraints, so \(C = C_{\text{inter}}\) and \(C_{\text{intra}} = \emptyset\). We use the terms "variables" and "agents" interchangeably.

Next, we discuss some examples of DisCSP.

3.2 Example of DisCSPs

Here, we describe three examples for DisCSP: the Distributed n-queens problem, the Distributed n-pieces m-chessboard problem and the Distributed Sensor-Mobile problem.

3.2.1 Distributed n-queens Problem

The n-queens problem is a frequent example in Constraint Satisfaction. In the distributed version of this problem, queens are represented by autonomous agents [YDIK98]. Similar to the n-queens problem (see Chapter 2, Section 2.2.1), the distributed n-queens problem consists of \(n\) queens which must be located in a \(n \times n\) chessboard in such a way that no queen attacks any other.

We model this problem as a DisCSP by having the same number of agents and queens. In this formulation, each queen is represented by an agent which holds a variable. Besides,
3.2. EXAMPLE OF DISCSPS

each agent is associated to one row of the \( n \times n \) chessboard. The variable held by an agent corresponds to the position of the queen in the row. There exits an inter-agent constraint between each pair of variables. Every constraint is explicitly defined by the set of allowed combinations of positions where the involved two queens can be located, i.e. those combinations of positions where the two queens are not placed in the same column or diagonal (by construction, two queens cannot be at the same row).

3.2.2 Distributed n-pieces m-chessboard Problem

The \( n \)-pieces \( m \)-chessboard problem is an extension of the \( n \)-queens problem [BM03]. This problem consists of \( n \) chess pieces and a \( m \times m \) chessboard and the goal is to put all pieces on the chessboard in such a way that no piece attacks any other. Similar to the distributed \( n \)-queens problem, in the distributed version of \( n \)-pieces \( m \)-chessboard, pieces are represented by autonomous agents.

One can formalize the distributed \( n \)-pieces \( m \)-chessboard problem as a DisCSP as follows. Every chess piece is represented by an agent, which holds a variable. The domains of each variable contain \( m \times m \) values, each one corresponds to a position of the \( m \times m \) chessboard. Analogous to the distributed \( n \)-queens problem, there exits a constraint between each pair of variables. Constraints depend on the way the involved chess pieces attack one another. Each constraint enumerates the set of combinations of positions where the two involved chess pieces can be located without attacking each other.

3.2.3 Distributed Sensor-Mobile Problem

Inspired by a real distributed resource allocation problem, [FBKG02] introduces the Distributed Sensor-Mobile problem (SensorDCSP). It consists of a set of sensors \( \{s_1, s_2, ..., s_n\} \) and set of mobiles \( \{m_1, m_2, ... m_k\} \). Sensors are required to cooperate for tracking mobiles. Each mobile must be tracked by 3 sensors. Each sensor can track at most one mobile. A solution is an assignment of three distinct sensors to each mobile. This assignment must satisfy two sets of constraints: visibility and compatibility constraints.

Figure 3.1 presents an example of SensorDCSP. This example includes 6 sensors (\( s_1, s_2, s_3, s_4, s_5 \) and \( s_6 \)) and 3 mobiles (\( m_1, m_2 \) and \( m_3 \)). This figure includes the visibility constraints (visibility graph (a)), compatibility constrains (compatibility graph (b)) and also a possible solution for this instance (given in graph (c)). One mobile is visible for a sensor if and only if there exits a directed arc between them in the visibility graph. Two sensors are compatible if and only if they are linked by an arc in the compatibility graph. In the solution, the three sensors assigned to each mobile are the sensors that form a triangle where the mobile is inside.

In general, finding a solution for the sensor-mobile problem is NP-complete [FBKG02]. Note that this problem can be easily reduced to the problem of partitioning a graph into cliques of size three, which has been proven to be NP-complete [KH83]. In contrast, instances of this problems, in which every pair of sensors is compatible, can be solved in polynomial time [FBKG02].
CHAPTER 3. DISTRIBUTED CONSTRAINT SATISFACTION

One can formulate SensorDCSP as a DisCSP, as follows. Each agent represents one mobile. Each agent includes three variables, one for each sensor that is required to track the corresponding mobile. The domain of a variable is the set of compatible sensors. There is a binary constraint between each pair of variables in the same agent. These intra-constraints must guarantee that sensors assigned to a mobile are different but compatible. There exits a binary constraint between the variables of different agents. These inter-constraints make every sensor be selected by at most one agent.

Furthermore, we present two other examples of problems that can be modeled as DisC-SPs in Part IV Applications. These problems are the Distributed Meeting Scheduling and Distributed Stable Matching problems, which will be studied in Chapter 10 and Chapter 11, respectively.

3.3 Algorithms for Solving DisCSP

The most trivial algorithm for solving a DisCSP is to gather all the information about the problem (variables, domains and constraints) into a single agent. Then, this agent solves the problem using a centralized algorithm (Chapter 2). In some cases, however, this approach is not convenient. The cost of collecting the whole problem into a single agent could be very high. Furthermore, in some applications, agents may desire to keep their local information as private as possible. In some other problems, agents may desire to participate actively during the solving process. For example, an agent may want to decide dynamically which value is more suitable for it, at any given time.

A distributed algorithm for DisCSP does not consider the centralized solving approach. In a distributed algorithm, all agents cooperate for finding a globally consistent solution. The solution involves assignments of all agents to all their variables. Agents exchange messages containing information about their assignments which allow them to check the consistency of assignments with respect to the problem constraints.

Depending on the model we assume about the timing of events in the distributed system,
3.3. **ALGORITHMS FOR SOLVING DISCSP**

we obtain different types of algorithms. In [Lyn97], three timing models are considered, which are informally described as follows:

1. **The synchronous model.** “This is the simplest model to describe, to program and to reason about. We assume that components (agents) take steps simultaneously, that is, that execution proceeds in synchronous rounds.”

2. **The asynchronous model.** “We assume that separate components (agents) take steps in arbitrary ordering, at arbitrary relative speeds.”

3. **The partially synchronous model.** “We assume some restrictions on the relative timing of events, but execution is not completely lock-step as it is in the synchronous model.”

These three timing models generate three types of algorithms for solving DisCSP. Broadly speaking, a synchronous algorithm is based on the notion of privilege, a token that is passed among agents. Only one agent is active at any time, the one having the privilege, while the rest of agents are waiting. When the process in the active agent terminates, it passes the privilege to another agent, which now becomes the active agent. In an asynchronous algorithm every agent is active at any time, and they do not have to wait for any event. A partially synchronous algorithm is in between these two types. An agent running a partially synchronous algorithm may be required to wait for some special event, but not for every event.

To solve a DisCSP instance, the three types of algorithms differ in their functionality and efficiency. Considering functionality, asynchronous algorithms are the most general and portable, because they impose no assumptions on the timing of computation steps. Usually, they are more robust and offer more privacy than the other two types. Regarding efficiency, defined generally as the amount of resources required to compute a solution, there is some debate as to which type of algorithm is more efficient. Literature concentrates on asynchronous algorithms for solving DisCSP. In following subsections, we describe several existent synchronous and asynchronous algorithms for DisCSP. Only a few partially synchronous algorithms have been studied. In Chapter 6, we present an algorithm that is basically asynchronous although it requires that some agents to synchronize their actions under certain conditions.

### 3.3.1 Synchronous Search

The simplest backtracking algorithm for DisCSP is derived from Chronological Backtracking algorithm (*BT*) (Section 2.3, Chapter 2). *BT* searches for a solution by continuously trying to make the extension of the current partial solution (which does not involve all the variables) into a total one.

The distributed version of this algorithm for DisCSP is called *SBT*. This algorithm was first presented in [YDIK92]. In this work, it is assumed that agents only have one variable. *SBT* requires a static instantiation ordering of agents. Agents exchange two kinds of messages: **ok?** and **ngd**. Following the static ordering, agents try to extend a partial
solution into a total one by adding consistent assignments for unassigned variables. Initially, the partial solution is empty. The first agent in the ordering assigns any value to its variable and inserts this assignment to the partial solution, which is sent, via an ok? message, to the second agent in the ordering. In general, when a variable receives an ok? message from the preceding agent in the ordering, it tries to assign a consistent value to its variable according to the constraints with the previously assigned variables. If such a value exits, the agent includes this valuation in the partial solution and sends the partial solution to the next agent in the ordering. Otherwise, the agent sends a ngd message to the preceding agent in the ordering, which causes the receiver to search for a new consistent value.

DisBT terminates either because all variables are included in the partial solution or because every value for the variable of the first agent has been discarded. In the former, the partial solution constitutes a solution for the problem, while in the latter, the problem is unsatisfiable. SBT is correct, complete and terminates.

The Conflict Backjumping (CBJ) algorithm is another CSP method that can be easily implemented in distributed settings in a synchronous form. As seen in Section 2.3, Chapter 2, CBJ improves the performance of BT. In CBJ, when a variable cannot find a consistent value with its constraints and the preceding variable assignments, the algorithm backtracks not to the previously assigned variable, but to the closest preceding culprit.

The term SCBJ represents the distributed version of CBJ. This algorithm was introduced in [ZM03]. SCBJ requires a total ordering among agents and uses the same kinds of messages as SBT: ok?, ngd. The only difference between SBT and SCBJ occurs when an agent does not have a valid value for one of its variables. In this situation, the SBT agent sends a ngd message to the previous agent in the ordering while the SCBJ agent sends a ngd message to the closest culprit agent. For an agent j, the closest culprit agent is the agent nearest to j, that appears before j in the ordering and holds a variable which forbids at least one value in j’s domain. The ngd message includes a nogood, that is the set of previously assigned variables which causes the sender to not have consistent values in its domain.

Analogous to SBT, SCBJ is correct, complete and terminates. SCBJ is experimentally compared with other novel synchronous methods in Chapter 4.

3.3.2 Asynchronous Search

Complete Algorithms

The pioneering works of Yokoo and colleagues propose two of the most famous asynchronous algorithms for DisCSP: the Asynchronous Backtracking (ABT) and the Asynchronous Weak-Commitment search (AWC) algorithm [YDIK92, Yok95, YDIK98]. Both methods assume the variable-based model.

These algorithms also require a total ordering among agents, which is static for ABT and dynamic for WCS. This ordering define the priority of the agents. Agents that appear first in the ordering have higher priority, while agents that appear last in the ordering have lower priority.
ABT and AWC assume that every constraint is directed following the total ordering among agents. For each constraint, the lowest priority agent involved in the constraint is the agent that evaluates the constraint and, therefore, is called the constraint-evaluating agent. The rest of agents in the constraint, that is, the agents involved in the constraint with higher priority than the constraint-evaluating agent, are called the value-sending agents because they send their assignments to the constraint-evaluating agent, which will check the consistency of the constraint. For each constraint, one link exits from each value-sending agent to the constraint-evaluating agent.

The neighbors of an agent $A_i$ refer to the set of agents that share constraints with $A_i$. The higher priority agents of an agent $A_i$ is the set of agents that are neighbors of $A_i$ and appear before $A_i$ in the ordering. Conversely, the lower priority agents of agent $A_i$ is the set of agents that are neighbors of $A_i$ and appear after $A_i$ in the ordering.

ABT and AWC use three kinds of messages to solve a problem: \textit{ok?}, \textit{ngd} and \textit{addl}. They initially assume links between each agent and its neighbors. In both algorithms, agents start the search by assigning initial values to their variables. Each time an agent assigns a new value to one of its variables, it uses an \textit{ok?} to inform its lower priority agents of the new assignment. Each agent stores the last received assignments from its higher priority agents in the agent view. When an agent does not find a value for one of its variables that is consistent with the assignments in the agent view, the agent backtracks. Both ABT and AWC have different ways of performing backtracking.

In the ABT algorithm presented in [YDIK92], when an agent cannot find a consistent value for one of its variables, it finds all the minimal nogoods of its agent view, that is, every nogood that does not contain another nogood. Then, the agent sends one \textit{ngd} message for every of those minimal nogoods. Each nogood is sent to the agent with the lowest priority among those that appear in the nogood that the message contains. After sending these messages, the agent forgets the assignments of those agents to which it sent a \textit{ngd} message.

A \textit{ngd} message causes the recipient agent to record the received nogood as a new constraint and to try to find a new value consistent with the agent view and with all recorded nogoods. To simplify the computation of all the minimal nogoods that are in the agent view of an agent, [YDIK98] propose to use the whole agent view of the agent as a single non-minimal nogood. This nogood is sent in a \textit{ngd} message to the closest agent in the agent view.

Because as agents act concurrently and asynchronously, a \textit{ngd} message may be obsolete when received by an agent. In this situation, the agents ignores the message and sends again its current value to the message sender. If an agent $A_j$ receives a \textit{ngd} message, it may be that the received nogood includes a variable $x_i$ belonging to $A_i$ which does not share a constraint with $A_j$. In this case, $A_j$ will send an \textit{addl} message to $A_i$ informing to $A_i$ that, each time $x_i$ changes its value $A_j$ must be informed via an \textit{ok?} message.

In ABT, the priority ordering of agents is determined at the beginning and it is maintained statically during the execution of the algorithm. This ordering identifies the agents that act as constraint-evaluating agents and the agents that act as value-sending agents. The static ordering has the drawback that any value proposed by a value-sending agent will not be changed unless an exhaustive search is performed by the constraint-evaluating agent.
agent, which could be costly in large-scale problems.

AWC can be seen as a modification of ABT, working with a dynamic priority ordering of agents. In AWC, in addition to the variable’s assignment, every ok? message also includes the priority value of the sender agent. When the current assignment is not consistent with the agent view, the agent selects a new consistent assignment that minimizes the number of constraint violations with lower priority agents. If an agent cannot find a consistent value with its agent view, it generates a new nogood, it sends the ngd message to all its neighbors and increases its priority one unit over the maximal priority of its neighbors. Then, it finds a consistent value with the assignments of higher priority agents and informs its neighbors via ok? messages. If no new nogood can be generated, the agent waits for the next message.

In both algorithms, nogoods are exchanged and stored by agents. This may lead to extra message passing and extra memory usage. ABT and AWC are correct, complete and terminate [Yok01]. However, the completeness of AWC may be affected if all nogoods are not stored. This causes AWC to have an exponential-space complexity.

The original ABT presented by Yokoo and colleagues has been developed further and studied in number of other works [YDIK98, HBQ98, BMM01, BMBM05]. A new version of ABT which does this without adding new links is one of the contributions of this thesis. This algorithm is presented in Chapter 5.

Next, we mention some other asynchronous and complete algorithms that have been proposed to solve DisCSP:

• DIBT is a distributed asynchronous backtracking algorithm which performs graph-based backjumping without nogood storage [HBQ98, Ham99]. In its original formulation, DIBT is not complete [Yok00, BMM01]. A revised version of this algorithm is discussed in Chapter 5.

• The Distributed Maintaining Asynchronously Consistency for ABT (DMAC-ABT), presented in [SSHF01a], is a complete protocol for maintaining asynchronously consistency. DMAC-ABT is a generic algorithm that can be easily integrated into more complex versions of ABT.

• The Asynchronous Aggregation Search (AAS) assumes the constraint-based model [SSHF’00, SF05]. The algorithm and its later versions can be seen as generalizations of ABT. They are based on the exchange of sets of partial solutions among agents. Unlike the variable-based model, where variables are distributed among agents, AAS considers the dual case where constraints are controlled by a single agent. The use of AAS may cause that the problem to be solved has to be translated into a new one. This transformation could be inadequate in many naturally distributed problems where the initial problem structure must remain unchanged. This approach is especially suitable for problems with arithmetic constraints, where variables are common to multiple agents.

• Following the idea of AWC, other works propose new algorithms which allow agents to change dynamically the priority ordering during search. Alternative dynamic variable
orderings for \textit{DisCSP} are investigated in [AD97]. [SSHF01b] say that a large number of reordering operations in an asynchronous algorithm may be costly. They suggest performing only a finite number of reordering operations. However, the experimental results presented in the work show minor improvements to static ordering \textit{ABT}. More recently, [ZM05b] propose a generic method for dynamic ordering in asynchronous backtracking (\textit{ABT-DO}). Agents in the algorithm choose orders dynamically and asynchronously. At the same time, each agent acts according to the most updated ordering it knows. An array of counters represents each suggested ordering of its priority with respect to others orders. Each time an agent replaces the assignment of its variables, it may propose a new ordering. In this algorithm, an agent can propose only orderings which affect to agents with lower priority than itself. Thus, the first agent in the ordering will never be reordered. The combination of \textit{ABT-DO} and heuristic inspired by the idea used for dynamic backtracking in \textit{CSPs} [Gin93] was found to be very effective.

- Recent works propose new approaches to \textit{DisCSP}. In the Concurrent Backtracking Search (\textit{ConBT}) several search processes asynchronously scan disjoined parts of the search space. The search that each process performs is completely synchronous and when an agent cannot find a consistent value, it backtracks following a chronological ordering [ZM04a]. In contrast to \textit{ConBT}, in the Concurrent Dynamic Backtracking (\textit{ConDB}) when an agent cannot find a consistent value, it backtracks to the closest agents involved in the the conflict [ZM04b].

\section*{Incomplete Algorithms}

Considering asynchronous and incomplete approaches for \textit{DisCSP}, the Distributed Break-out algorithm (\textit{DisBO}) [Yok01] is the method that has been studied the most in the literature. The algorithm is inspired by the Break-out algorithm (\textit{BO}) for \textit{CSP}. Similar to the original Break-out algorithm, \textit{DisBO} assumes that the original constraints of the problem and the inconsistency found during the search, are represented as nogoods (i.e. a subset of conflicting variable values). Every conflict has a weight associated, which initially is equal to 1.

In \textit{BO}, a \textit{flawed solution} containing some constraint violations is revised by local changes until all constraints are satisfied. The evaluation of the flawed solution is defined by the sum of the weights of all conflicts that appears in it. The algorithm moves from one flawed solution to another one if the new one has better evaluation than the preceding one.

\textit{DisBO} agents exchange two kinds of messages among them: \texttt{ok?} and \texttt{improve}. The former is used by an agent to exchange the current valuation of its variables while the latter is used to reveal the maximal improvement it will get if it changes the assignments of its variables. In contrast to \textit{BO}, in \textit{DisBO} more than one variable may change its value in the flawed solution at a time, which allows to the algorithm to take advantage of the parallelism. Each agent communicates only with its neighboring agents, that is, the agents that share a constraint with them. Neighboring agents exchange values of possible improvements, and
only the agent that can maximally improve the evaluation value may change its value. Note that it is possible for two non-neighboring agents to change its assignments concurrently.

Since agents process only local information, they cannot detect when the whole system is trapped in a local-minimum like in BO. Alternatively, DisBO agents work with the concept of quasi-local-minimum, which is a weaker condition than a local-minimum but can be detected via local communications. An agent is said to be in a quasi-local-minimum if the agent is violating some constraint and the possible improvement of it and all its neighboring agents is 0. When an agent is trapped in a quasi-local-minimum, it increases by 1 the weights of all constraint violations which help to the system to jump from a possible real local minimum.

3.4 Comparing Algorithmic Performance

Some debate has happened about the parameters for measuring the performance evaluation of the algorithms [MRKZ02]. We consider that solving distributed problems requires search effort from individual agents, plus the usage of a global network that implements message passing. The search effort can be measured as the total number of constraint checks performed by the set of agents during one execution, or as the CPU time that an agent requires to complete such execution (network communication time is not included in this measure). The network usage is measured as the total number of messages exchanged during the resolution of a problem. The cost in time of exchanging one message is usually higher than the cost of performing a constraint check, although the exact relation depends on the implementation and the network. Because of that, we tend to consider better algorithms those which exchange less messages.

In synchronous algorithms, the computation effort is measured by the total number of constraint checks (cc), and the global communication effort is evaluated by the total number of messages exchanged among agents (msg) [Lyn97]. In asynchronous algorithms, search effort is measured by the number of ”non concurrent constraint checks” (ncce), which was defined in [MRKZ02], following Lamport’s logic clocks [Lam78]. Each agent has a counter for its own number of constraint checks. The number of non concurrent constraint checks is computed by attaching to each message the current counter of the constraint checks of the sending agent. When an agent receives a message, it updates its counter to the higher value between its own counter and the counter attached to the received message. When the algorithm terminates, the highest value among all the agent counters is taken as the number of concurrent constraint checks. Informally, this number approximates the longest sequence of constraint checks not performed concurrently. As for synchronous search, we evaluate the global communication effort as the total number of messages exchanged among agents (msg).

Note that, in synchronous algorithms, all constraint checks are performed sequentially. Therefore, ncce = cc. In this thesis we use the term ncce to refer to the computation cost in both kinds of algorithms.
3.5. **SIMULATOR**

3.4.1 **Random Binary *DisCSP***

Random binary problems are commonly used as sample problems to evaluate the performance of methods for solving *CSP* and *DisCSP*. A binary random *CSP* class is characterized by \( (n, m, p_1, p_2) \) where \( n \) is the number of variables, \( m \) the number of values per variable, \( p_1 \) the network connectivity defined as the ratio of existent constraints, and \( p_2 \) the constraint tightness defined as the ratio of forbidden value pairs. The constrained variables and the forbidden value pairs are randomly selected [Smi94]. Similarly, a binary random *DisCSP* is defined by \( (n, m, p_1, p_2) \), where each element has the same meaning as for random binary *CSP*. In addition, each variable is assigned to one agent.

3.5 **Simulator**

Ideally, to evaluate a new algorithm one should have \( n \) dedicated processors connected to a common network on which tests would be done. However, this setting is often not available in most of our labs. Even if there is a number of computers available, the workload of each computer and the load of the communication network are out of the control of the experimenter, and these aspects have a significant impact on the efficiency of the algorithms. Because of that, we consider that simulation on a single computer is a suitable alternative in order to make most of the experimentation in *DisCSP* algorithms. After that, some algorithms can be tested in a real setting, assuming the resources needed to perform a field test. In the following, we consider the different options for *DisCSP* algorithms when are evaluated by simulation on a single computer.

Usually, *DisCSP* algorithms are described in terms of agents. An agent is an autonomous entity that contains a part of the problem, is able to perform its own reasoning process and to communicate with other agents. In a multi-task computer (for instance, a desktop with Linux operating system (OS)), a direct option is to implement each agent as a different task, all having the same priority. The OS scheduler is in charge of activating / deactivating the agents, that take control of the CPU as any other task in the system. Communication among agents is performed using a standard task communication facilities (usually implemented using disk storage). This approach is relatively simple to implement but presents some drawbacks. First, it depends on the OS, so results obtained in computers with different OS may not be directly comparable. Second, even using the same computer and the same implementation, it is difficult to reproduce exactly the same results when repeating the same experiments. There are some sublet factors (such as the mail server, the network load, the disk storage) which change between executions and are out of the experimenter’s control. Because of that, the exact reproduction of previous results is almost impossible with this approach.

To overcome this obstacle, an alternative is to use a simulator that offers the same facilities as the OS, but allows one complete control. This simulator allows agents to execute, perform the scheduling among agents and provide communication facilities. With this approach, results are reproducible, the same experiment generates the same results (providing random elements are initialized with the same seed).
The first simulator of this kind appears in the seminal work of Yokoo [YDIK92, YDIK98]. Each agent keeps its own clock, which is incremented at each cycle of computation. One cycle for an agent consists of reading all its incoming messages, processing them and writing all messages generated as answers. It is assumed that a message sent at time $t$ is available to the receiver at time $t + 1$. This means a kind of synchronicity in the activation of agents, which is somehow contradictory with the evaluation of asynchronous procedures.

Another scheduling policy is to activate agents randomly: a random number between 1 and $n$ determines the identifier of the agent to activate. When this agent terminates, the same process selects the next agent to activate. This approach seems to be more adequate to evaluate asynchronous procedures. In this work, every algorithm is designed and implemented following this approach.

### 3.6 Summary

In this chapter we have discussed the basic issues of Distributed Constraint Satisfaction. We formally define the problem and present problems that can be modeled as DisCSPs. We also describe the main existent methods for solving DisCSP. At the end of the chapter, we discuss some issues that have to be considered when evaluating and implementing DisCSP procedures.