PART 1

Background on Centralized and Distributed Constraint Reasoning

1

Constraint Satisfaction Problems

This chapter provides the state of the art in the area of centralized constraint programming. In section 1.1, we define the constraint satisfaction problem (CSP) formalism and present some academic and real examples of problems modeled and solved by centralized CSP. Typical methods for solving centralized CSP are described in section 1.2.

1.1. Centralized constraint satisfaction problems

Many real-world combinatorial problems in artificial intelligence arising from areas related to resource allocation, scheduling, logistics and planning are solved using constraint programming. Constraint programming is based on its powerful framework called CSP. A CSP is a general framework that involves a set of variables and constraints. Each variable can assign a value from a domain of finite possible values. Constraints specify the allowed values for a set of variables. Hence, a large variety of applications can be naturally formulated as CSPs. Examples of applications that have been successfully solved by constraint programming are picture processing [MON 74], planning [STE 81], job-shop scheduling [FOX 82], computational vision [MAC 83], machine design and manufacturing [FRA 87, NAD 90], circuit analysis [DEK 80], diagnosis [GEF 87], belief maintenance [DEC 88], automobile transmission design [NAD 91], etc.

Solving a CSP consists of looking for solutions to a constraint network, that is a set of assignments of values to variables that satisfy the constraints of the problem. A constraint represents restrictions on value combinations allowed for constrained variables. Many powerful algorithms have been designed for solving CSPs. Typical systematic search algorithms try to develop a solution to a CSP by incrementally instantiating the variables of the problem.

There are two main classes of algorithms searching solutions for CSPs, namely those of a look-back scheme and those of look-ahead scheme. The first category of search algorithms (look-back scheme) corresponds to search procedures checking the validity of the assignment of the current variable against the already assigned (past) variables. When the assignment of the current variable is inconsistent with assignments of past variables, a new value is tried. When no value remains, a past variable must be reassigned (i.e. change its value). Chronological backtracking (BT) [GOL 65], backjumping (BJ) [GAS 78], graph-based backjumping (GBJ) [DEC 90], conflict-directed backjumping (CBJ) [PRO 93] and dynamic backtracking (DBT) [GIN 93] are algorithms performing a look-back scheme.

The second category of search algorithms (look-ahead scheme) corresponds to search procedures that check forward the assignment of the current variable. In a look-ahead scheme, the not yet assigned (future) variables are made consistent, to some degree, with the assignment of the current variable. Forward checking (FC) [HAR 80] and maintaining arc consistency (MAC) [SAB 94] are algorithms that perform a look-ahead scheme.

Proving the existence of solutions or finding them in CSP are nondeterministic polynomial time (NP)-complete tasks. Thereby, numerous *heuristics* were developed to improve the efficiency of solution methods. Although being numerous, these heuristics can be categorized into two kinds: variable ordering and value ordering heuristics. Variable ordering heuristics address the order in which the algorithm assigns the variables, whereas the value ordering heuristics establish an order on which values will be assigned to a selected variable. Many studies have shown that the ordering of selecting variables and values dramatically affects the performance of search algorithms.

We present in the following an overview of typical methods for solving centralized CSPs after formally defining a CSP and give some examples of problems that can be encoded in CSPs.

1.1.1. *Preliminaries*

A CSP (or a constraint network) [MON 74] involves a finite set of variables, a finite set of domains determining the set of possible values for a given variable and a finite set of constraints. Each constraint restricts the combination of values that a set of variables it involves can assign. A solution is an assignment of values to all variables satisfying all constraints.

DEFINITION 1.1.– *A* constraint satisfaction problem *or a* constraint network *was formally defined by a triple* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$*, where:*

– $\mathcal{X}$ *is a set of* n variables $\{x_1, \ldots, x_n\}$*;*

– $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ *is a set of* n *current* domains*, where* $D(x_i)$ *is a finite set of possible values to which variable* x_i *may be assigned;*

– $\mathcal{C} = \{c_1, \ldots, c_e\}$ *is a set of* e constraints *that specify the combinations of values (or* tuples*) allowed for the variables they involve. The variables involved in a constraint* $c_k \in \mathcal{C}$ *form its* scope *(*`scope`$(c_k) \subseteq \mathcal{X}$*).*

During a search procedure, values may be pruned from the domain of a variable. At any node, the set of possible values for variable x_i is its *current domain*, $D(x_i)$. We introduce the particular notation of *initial domains* (or definition domains) $\mathcal{D}^0 = \{D^0(x_1), \ldots, D^0(x_n)\}$, which represents the set of domains before pruning any value (i.e. $\mathcal{D} \subseteq \mathcal{D}^0$).

The number of variables on the scope of a constraint $c_k \in \mathcal{C}$ is called a the arity of the constraint c_k. Therefore, a constraint involving one variable (respectively, two or n variables) is called a unary (respectively, *binary* or n-ary) constraint. In this book, we are concerned with binary constraint networks where we assume that all constraints are binary constraints (they involve two variables). A constraint in $\mathcal{C}$ between two variables x_i and x_j is then denoted by c_{ij}. c_{ij} is a subset of the Cartesian product of their domains (i.e. $c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$). A direct result of this assumption is that the connectivity between the variables can be represented with a constraint graph G [DEC 92].

DEFINITION 1.2.– *A binary constraint network can be represented by a* constraint graph $G = \{X_G, E_G\}$*, where vertices represent the variables of the problem (*$X_G = \mathcal{X}$*) and edges (*E_G*) represent the constraints (i.e.* $\{x_i, x_j\} \in E_G$ *iff* $c_{ij} \in \mathcal{C}$*).*

DEFINITION 1.3.– *Two variables are* adjacent *iff they share a constraint. Formally,* x_i *and* x_j *are adjacent iff* $c_{ij} \in \mathcal{C}$*. If* x_i *and* x_j *are adjacent, we also say that* x_i *and* x_j *are* neighbors*. The set of neighbors of a variable* x_i *is denoted by* $\Gamma(x_i)$.

DEFINITION 1.4.– *Given a constraint graph G, an* ordering $\mathcal{O}$ *is a mapping from the variables (vertices of G) to the set* $\{1, \ldots, n\}$*.* $\mathcal{O}(i)$ *is the ith variable in* $\mathcal{O}$*.*

Solving a CSP is equivalent to finding a combination of assignments of values to all variables in a way that all the constraints of the problem are satisfied.

In the following, we present some typical examples of problems that can be intuitively modeled as CSPs. These examples range from academic problems to real-world applications.

1.1.2. *Examples of CSPs*

Various problems in artificial intelligence can be naturally modeled as a CSP. We present here some examples of problems that can be modeled and solved by the CSP framework. First, we describe the classical n-queens problem. Next, we present the graph coloring problem. Finally, we introduce the problem of meeting scheduling.

1.1.2.1. *The n-queens problem*

The n-queens problem is a classical combinatorial problem that can be formalized and solved by a CSP. In the n-queens problem, the goal is to put n queens on an $n \times n$ chessboard so that none of them are able to attack (capture) any other. Two queens attack each other if they are located on the same row, column or diagonal on the chessboard. This problem is called a CSP because the goal is to find a configuration that satisfies the given conditions (constraints).

In the case of 4-queens ($n = 4$, Figure 1.1), the problem can be encoded as a CSP as follows[1]:

– $\mathcal{X} = \{q_1,\ q_2,\ q_3,\ q_4\}$, each variable q_i corresponds to the queen placed in the ith column;

– $\mathcal{D} = \{D(q_1), D(q_2), D(q_3), D(q_4)\}$, where $D(q_i)$=$\{1, 2, 3, 4\}$ $\forall i \in$ 1.4. The value $v \in D(q_i)$ corresponds to the row where the queen representing the ith column can be placed;

– $\mathcal{C} = \{c_{ij} : (q_i \neq q_j) \wedge (\mid q_i - q_j \mid \neq \mid i - j \mid)\ \forall\ i, j \in \{1, 2, 3, 4\}$ and $i \neq j\}$ is the set of constraints. A constraint between each pair of queens exists that forbids the involved queens to be placed in the same row or diagonal line.

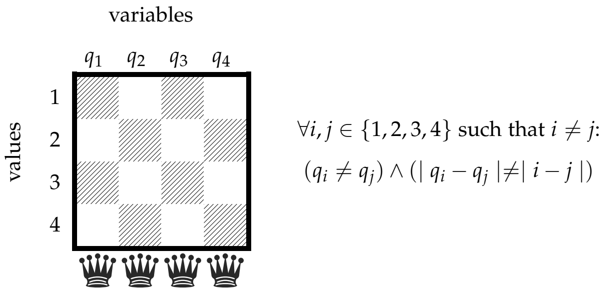

variables
q_1 q_2 q_3 q_4
values
1
2
3
4
$\forall i, j \in \{1, 2, 3, 4\}$ such that $i \neq j$:
$(q_i \neq q_j) \wedge (\mid q_i - q_j \mid \neq \mid i - j \mid)$

Figure 1.1. *The 4-queens problem*

The n-queen problem permits, in the case of $n = 4$ (4-queens), two configurations as solutions. We present the two possible solution in Figure 1.2. The first solution, Figure 1.2(a), is $(q_1 = 2, q_2 = 4, q_3 = 1, q_4 = 3)$, where we put q_1 in the second row, q_2 in the fourth row q_3 in the first row and q_4 is placed in the third row. The second solution, Figure 1.2(b), is $(q_1 = 3, q_2 = 1, q_3 = 4, q_4 = 2)$.

1.1.2.2. *The graph coloring problem*

Another typical problem is the graph coloring problem. Graph coloring is one of the most combinatorial problem studied in artificial intelligence because many real

1 This is not the only possible encoding of the n-queens problem as a CSP.

applications such as time-tabling and frequency allocation can be easily formulated as a graph coloring problem. The goal in this problem is to color all nodes of a graph so that any two adjacent vertices should get different colors where each node has a finite number of possible colors. The graph coloring problem is simply formalized as a CSP. Hence, the nodes of the graph are the variables to color and the possible colors of each node/variable form its domain. A constraint between each pair of adjacent variables/nodes exists that prohibits these variables from having the same color.

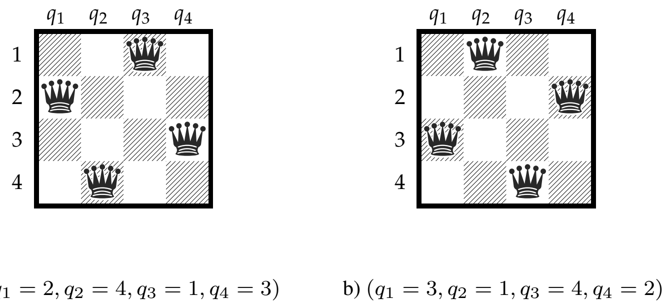

q_1 q_2 q_3 q_4
1
2
3
4
a) $(q_1 = 2, q_2 = 4, q_3 = 1, q_4 = 3)$
b) $(q_1 = 3, q_2 = 1, q_3 = 4, q_4 = 2)$

Figure 1.2. *The solutions for the 4-queens problem*

A practical application of the graph coloring problem is the problem of coloring a map (Figure 1.3). The objective in this case is to assign a color to each region so that no neighboring regions have the same color. An instance of the map coloring problem is illustrated in Figure 1.3(a), where we present the map of Morocco with its 16 provinces. We present this map-coloring instance as a constraint graph in Figure 1.3(b). This problem can be modeled as a CSP by representing each node of the graph as a variable. The domain of each variable is defined by the possible colors. A constraint exists between each pair neighboring regions. Therefore we get the following CSP:

– $\mathcal{X} = \{x_1, x_2, \ldots, x_{16}\}$;

– $\mathcal{D} = \{D(x_1), D(x_2), \ldots, D(x_{16})\}$, where $D(x_i) = \{red, blue, green\}$;

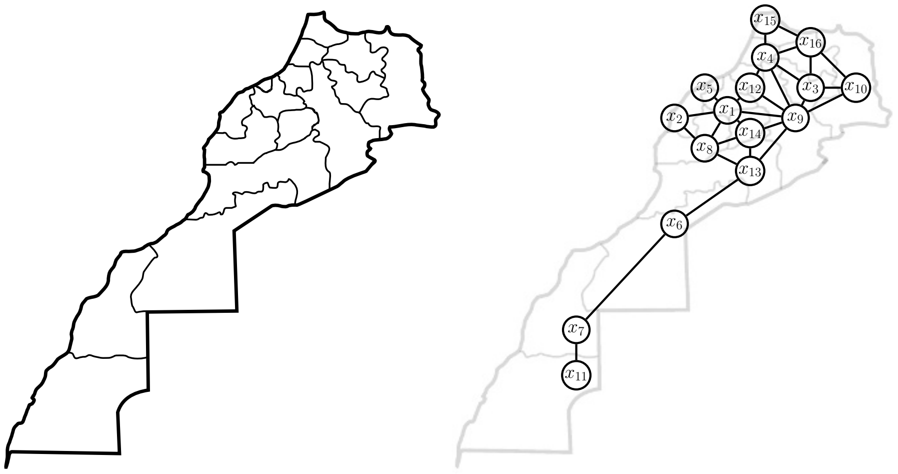
– $\mathcal{C} = \{c_{ij} : x_i \neq x_j \mid x_i$ and x_j are neighbors$\}$.

1.1.2.3. *The meeting scheduling problem*

The *meeting scheduling problem* (MSP) [SEN 95, GAR 96, MEI 04] is a decision-making process that consists of scheduling several meetings among various people with respect to their personal calendars. The MSP has been defined in many versions with different parameters (e.g. duration of meetings [WAL 02] and preferences of agents [SEN 95]). In MSP, we have a set of attendees, each with his/her own calendar (divided into time-slots), and a set of n meetings to coordinate. In general, people/participants may have several slots already filled in their calendars.

Each meeting m_i takes place in a specified location denoted by $location(m_i)$. The proposed solution must enable the participating people to travel among locations where their meetings will be held. Thus, an *arrival-time* constraint is required between two meetings m_i and m_j when at least one attendee participates in both the meetings. The arrival-time constraint between two meetings m_i and m_j is defined in equation [1.1]:

$$\mid time(m_i) - time(m_j) \mid - duration > TravelingTime(location(m_i), location(m_j)). \qquad [1.1]$$



a) *The 16 provinces of Morocco*

b) *The map coloring problem represented as a constraint graph*

Figure 1.3. *An example of the graph coloring problem*

The MSP [MEI 04] can be encoded in a centralized CSP as follows:

– $\mathcal{X} = \{m_1, \ldots, m_n\}$ is the set of variables where each variable represents a meeting;

– $\mathcal{D} = \{D(m_1), \ldots, D(m_n)\}$ is a set of domains, where $D(m_i)$ is the domain of variable/meeting m_i. $D(m_i)$ is the intersection of time-slots from the personal calendar of all agents attending m_i, that is $D(m_i) = \bigcap_{A_j \in \text{ attendees of } m_i} calendar(A_j)$;

– $\mathcal{C}$ is a set of arrival-time constraints. An arrival-time constraint for every pair of meetings (m_i, m_j) exists if there is an agent that participates in both meetings.

A simple instance of a MSP is illustrated in Table 1.1. There are four attendees: $Adam$, $Alice$, $Fred$ and Med, each having a personal calendar. There are four meetings to be scheduled. The first meeting (m_1) will be attended by $Alice$ and Med. $Alice$ and $Fred$ will participate in the second meeting (m_2). The agents attending the third meeting (m_3) are $Fred$ and Med, while the last meeting (m_4) will be attended by $Adam$, $Fred$ and Med.

Meeting	Attendees	Location
m_1	Alice, Med	Paris
m_2	Alice, Fred	Rabat
m_3	Fred, Med	Montpellier
m_4	Adam, Fred, Med	Agadir

Table 1.1. *A simple instance of the meeting scheduling problem*

The instance presented in Table 1.1 is encoded as a centralized CSP in Figure 1.4. The nodes are the meetings/variables (m_1, m_2, m_3, m_4). The edges represent binary arrival-time constraints. Each edge is labeled by the person, attending both meetings. Thus,

– $\mathcal{X} = \{m_1,\ m_2,\ m_3,\ m_4\}$;

– $\mathcal{D} = \{D(m_1),\ D(m_2),\ D(m_3),\ D(m_4)\}$;

- $D(m_1) = \{s \mid s$ is a slot in $calendar(Alice) \cap calendar(Med)\}$,

- $D(m_2) = \{s \mid s$ is a slot in $calendar(Alice) \cap calendar(Fred)\}$,

- $D(m_3) = \{s \mid s$ is a slot in $calendar(Fred) \cap calendar(Med)\}$,

- $D(m_4) = \{s \mid s$ is a slot in $calendar(Adam) \cap calendar(Fred) \cap calendar(Med)\}$;

– $\mathcal{C} = \{c_{12},\ c_{13},\ c_{14},\ c_{23},\ c_{24},\ c_{34}\}$, where c_{ij} is an arrival-time constraint between m_i and m_j.

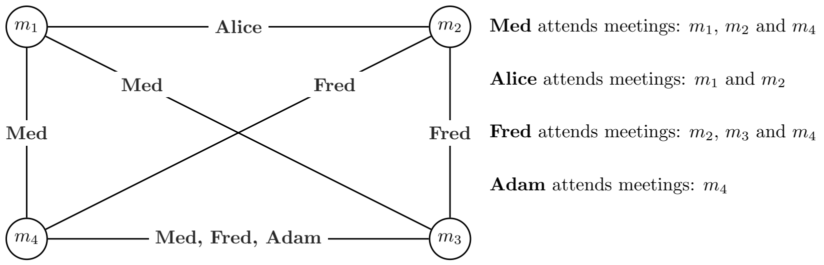

m_1
m_2
m_3
m_4
Alice
Med
Fred
Med
Fred
Med, Fred, Adam
Med attends meetings: m_1, m_2 and m_4
Alice attends meetings: m_1 and m_2
Fred attends meetings: m_2, m_3 and m_4
Adam attends meetings: m_4

Figure 1.4. *The constraint graph of the meeting scheduling problem*

The previous examples show the power of the CSP framework to easily model various combinatorial problems arising from different issues. In the following section, we describe the well-known generic methods for solving a CSP.

1.2. Algorithms and techniques for solving centralized CSPs

In this section, we describe the basic methods for solving CSPs. These methods can be considered under two broad approaches: constraint propagation and search. Here, we also describe a combination of those two approaches. In general, the search algorithms explore all possible combinations of values for the variables in order to find a solution of the problem, that is a combination of values for the variables that satisfies the constraints. However, the constraint propagation techniques are used to reduce the space of combinations that will be explored by the search process. Afterward, we present the main heuristics used to boost the search in the centralized CSPs. We particularly summarize the main variable ordering heuristics, while we briefly describe the main value ordering heuristics used in the CSPs.

1.2.1. *Algorithms for solving centralized CSPs*

Usually, algorithms for solving centralized CSPs search systematically through the possible assignments of values to variables in order to find a combination of these assignments that satisfies the constraints of the problem.

DEFINITION 1.5.– *An* assignment *of value v_i to a variable x_i is a pair (x_i, v_i) where v_i is a value from the domain of x_i, that is $v_i \in D(x_i)$. We often denote this assignment by $x_i = v_i$.*

Henceforth, when a variable is assigned a value from its domain, we say that the variable is assigned or instantiated.

DEFINITION 1.6.– *An* instantiation *$\mathcal{I}$ of a subset of variables $\{x_i, \dots, x_k\} \subseteq \mathcal{X}$ is an ordered set of assignments $\mathcal{I} = \{[(x_i = v_i), \dots, (x_k = v_k)] \mid v_j \in D(x_j)\}$. The variables assigned on instantiation $\mathcal{I} = [(x_i = v_i), \dots, (x_k = v_k)]$ are denoted by* $\texttt{vars}(\mathcal{I}) = \{x_i, \dots, x_k\}$.

DEFINITION 1.7.– *A* full instantiation *is an instantiation $\mathcal{I}$ that instantiates all the variables of the problem (i.e.* $\texttt{vars}(\mathcal{I}) = \mathcal{X}$*), and conversely we say that an instantiation is* a partial instantiation *if it instantiates in only a part.*

DEFINITION 1.8.– *An instantiation $\mathcal{I}$* satisfies *a constraint $c_{ij} \in \mathcal{C}$ if and only if the variables involved in c_{ij} (i.e. x_i and x_j) are assigned in $\mathcal{I}$ (i.e. $(x_i = v_i), (x_j = v_j) \in \mathcal{I}$) and the pair (v_i, v_j) is allowed by c_{ij}. Formally, $\mathcal{I}$ satisfies c_{ij} iff $[(x_i = v_i) \in \mathcal{I}] \wedge [(x_j = v_j) \in \mathcal{I}] \wedge [(v_i, v_j) \in c_{ij}]$.*

DEFINITION 1.9.– *An instantiation $\mathcal{I}$ is* locally consistent *iff it satisfies all of the constraints whose scopes have no uninstantiated variables in $\mathcal{I}$. $\mathcal{I}$ is also called a* partial solution. *Formally, $\mathcal{I}$ is locally consistent iff* $\forall c_{ij} \in \mathcal{C} \mid \texttt{scope}(c_{ij}) \subseteq \texttt{vars}(\mathcal{I})$ *and* $\mathcal{I}$ *satisfies* c_{ij}.

DEFINITION 1.10.– *A* solution *to a constraint network is a full instantiation $\mathcal{I}$, which is locally consistent.*

The intuitive way to search a solution for a CSP is to *generate and test* all possible full instantiations and check their validity (i.e. if they satisfy all constraints of the problem). The full instantiations satisfying all constraints are then solutions. This is the principle of the *generate & test* algorithm. In other words, a full instantiation is generated and then tested if it is locally consistent. In the generate & test algorithm, the consistency of an instantiation is not checked until it is full. This method drastically increases the number of combinations that will be generated. The number of full instantiations considered by this algorithm is the size of the Cartesian product of all the variable domains. Intuitively, one can check the local consistency of instantiation as soon as its respective variables are instantiated. In fact, this is the systematic search strategy of the chronological BT algorithm. We present the chronological BT in the following.

1.2.1.1. *Chronological backtracking*

The chronological BT [DAV 62, GOL 65, BIT 75] is the basic systematic search algorithm for solving CSPs. The BT is a recursive search procedure that incrementally attempts to extend a current partial solution (a locally consistent instantiation) by assigning values to variables not yet assigned, toward a full instantiation. However, when all values of a variable are inconsistent with previously assigned variables (a *dead-end* occurs), BT backtracks to the variable immediately instantiated in order to try another alternative value for it.

DEFINITION 1.11.– *When no value is possible for a variable, a* dead-end *state occurs. We usually say that the domain of the variable is* wiped out (DWO).

The pseudo-code of the BT algorithm is illustrated in algorithm 1.1. The BT assigns a value to each variable in turn. When assigning a value v_i to a variable x_i, the consistency of the new assignment with values assigned thus far is checked (line 6, algorithm 1.1). If the new assignment is consistent with previous assignments, BT attempts to extend these assignments by selecting another unassigned variable (line 7). Otherwise (the new assignment violates any of the constraints), another alternative value is tested for x_i if it is possible. If all values of a variable are inconsistent with previously assigned variables (a dead-end occurs), BT to the variable immediately preceding the dead-end variable takes place in order to check alternative values for this variable. In this way, either a solution is found when the last variable has been successfully assigned or BT can conclude that no solution exists if all values of the first variable are removed.

Algorithm 1.1. *The chronological backtracking algorithm.*

```
procedure Backtracking($\mathcal{I}$)
01. if ( isFull($\mathcal{I}$) ) then report $\mathcal{I}$ as solution;        /* all variables are assigned in $\mathcal{I}$ */
02. else
03.     select $x_i$ in $\mathcal{X} \setminus$ vars($\mathcal{I}$) ;        /* let $x_i$ be an unassigned variable */
04.     foreach ( $v_i \in D(x_i)$ ) do
05.         $x_i \leftarrow v_i$;
06.         if ( isLocallyConsistent($\mathcal{I} \cup \{(x_i = v_i)\}$) ) then
07.             Backtracking($\mathcal{I} \cup \{(x_i = v_i)\}$);
```

Figure 1.5 illustrates an example of running the BT algorithm on the 4-queens problem (Figure 1.1). First, variable q_1 is assigned to 1 (the first queen representing the queen to place in the first column, is placed in the first row of the 4×4 chessboard) and added to the partial solution $\mathcal{I}$. Next, BT attempts to extend $\mathcal{I}$ by assigning the next variable q_2. Because we cannot assign values 1 or 2 for q_2 as these values violate the constraint c_{12} between q_1 and q_2, we select value 3 to be assigned to q_2 ($q_2 = 3$). Then, BT attempts to extend $\mathcal{I} = [(q_1 = 1), (q_2 = 3)]$ by assigning the next variable q_3. No value from $D(q_3)$ exists that satisfies all of the constraints with $(q_1 = 1)$ and $(q_2 = 3)$ (i.e. c_{13} and c_{23}). Therefore, a BT is performed to the most recently instantiated variable (i.e. q_2) in order to change its current value (i.e. 3). Hence, variable q_2 is assigned to 4. Afterward, the value 2 is assigned to next variable q_3 because value 1 violates the constraint c_{13}. Then, the algorithm backtracks to variable q_3 after attempting to assign variable q_4 because no possible assignment for q_4 exists that is consistent with previous assignments $\mathcal{I} = [(q_1 = 1), (q_2 = 4), (q_3 = 2)]$. Thus, $q_3 = 2$ must be changed. However, no value consistent with $(q_1 = 1)$ and $(q_2 = 4)$ is available for q_3. Hence, another backtrack is performed to q_2. In the same way BT backtracks again to q_1 as no value for q_2 is consistent with $(q_1 = 1)$. Then, $q_1 = 2$ is selected for the first variable q_1. After that, q_2 is assigned to 4 because other values (1, 2 and 3) violate the constraint c_{12}. Next, $\mathcal{I}$ is extended by adding a new assignment $(q_3 = 1)$ of the next variable q_3 consistent with $\mathcal{I}$. Finally, an assignment, consistent with the extended partial solution $\mathcal{I}$, is sought for q_4. The first and the second values (row number 1 and 2) from $D(q_4)$ are not consistent with $\mathcal{I} = [(q_1 = 2), (q_2 = 4), (q_3 = 1)]$. Then, BT chooses 3 that is consistent with $\mathcal{I}$ to be instantiated to q_4. Hence, a solution is found because all variables are instantiated in $\mathcal{I}$, where $\mathcal{I} = [(q_1 = 2), (q_2 = 4), (q_3 = 1), (q_4 = 3)]$.

On the one hand, it is clear that we need only linear space to perform the BT. However, it requires time exponential in the number of variables for most nontrivial problems. On the other hand, the BT is clearly better than "generate & test" because a subtree from the search space is pruned whenever a partial instantiation violates a constraint. Thus, BT can detect early unfruitful instantiation compared to "generate & test".

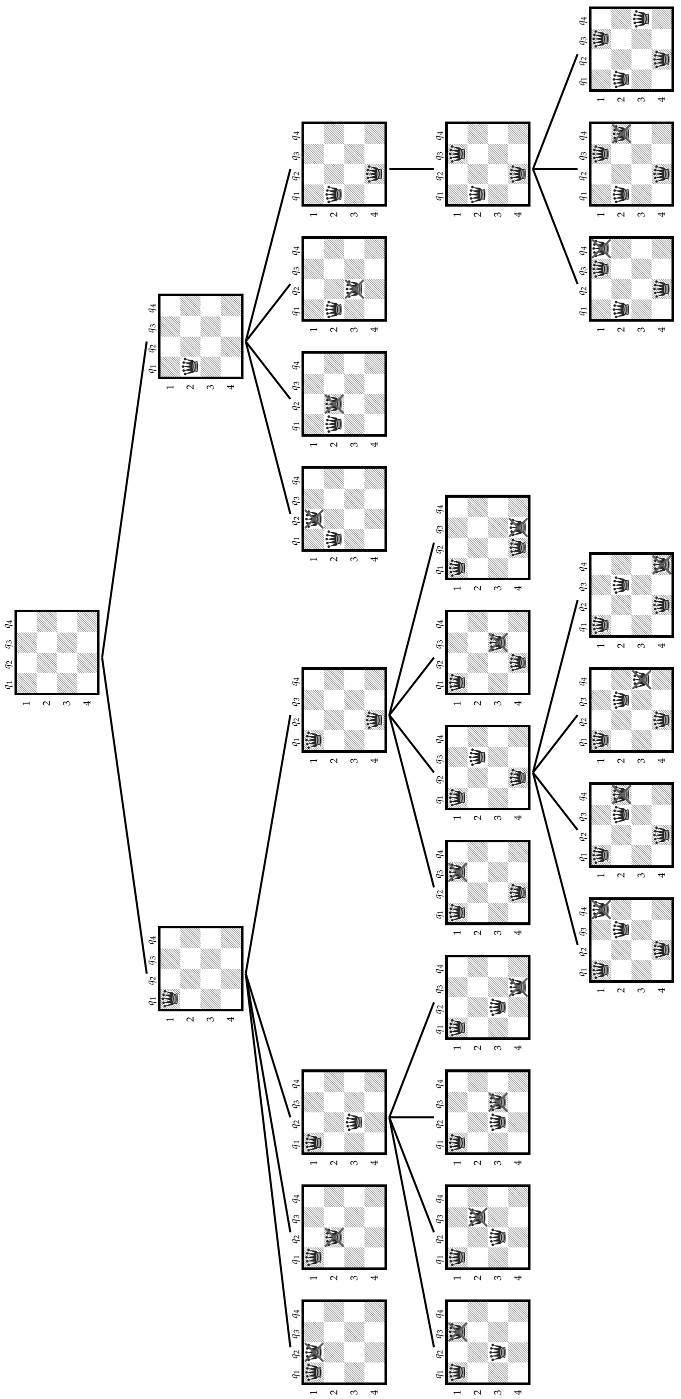


Figure 1.5. *The chronological backtracking algorithm running on the 4-queens problem*

Although the BT improves the “generate & test”, it still suffers from many drawbacks. The main drawback is the *thrashing* problem. Thrashing is the fact that the same failure due to the same reason can be rediscovered an exponential number of times when solving the problem. Therefore, a variety of refinements of BT have been developed in order to improve it. These improvements can be classified under two main schemes: look-back methods such as CBJ or look-ahead methods such as FC.

1.2.1.2. *Conflict-directed backjumping*

From the earliest work in the area of constraint programming, researchers were concerned by the trashing problem of the BT algorithm and then they proposed a number of techniques to avoid it. The BJ concept was one of the pioneer techniques used for this reason. Thus, several non-chronological BT (intelligent BT) search algorithms have been designed to solve centralized CSPs. In the standard form of BT, each time a dead-end occurs, the algorithm attempts to change the value of the most recently instantiated variable. However, BT chronologically to the most recently instantiated variable may not address the reason of the failure. This is no longer the case in the BJ algorithms that identify and then $jump$ directly to the responsible dead-end ($culprit$). Hence, the culprit variable is reassigned if it is possible or another jump is performed. Incidentally, the subtree of the search space where the thrashing may occur is pruned.

DEFINITION 1.12.– *Given a total ordering on variables $\mathcal{O}$, a constraint c_{ij} is* earlier *than c_{kl} if the latest variable in* `scope(`c_{ij}`)` *precedes the latest one in* `scope(`c_{kl}`)` *on $\mathcal{O}$.*

EXAMPLE 1.1.– Given the lexicographic ordering on variables ($[x_1, \ldots, x_n]$), the constraint c_{25} is earlier than constraint c_{35} because x_2 precedes x_3 since x_5 belongs to both scopes (i.e. `scope(`c_{25}`)` and `scope(`c_{35}`)`).

Gaschnig designed the first explicit non-chronological (BJ) algorithm in [GAS 78]. For each variable x_i BJ records the *deepest* variable with which it checks its consistency with the assignment of x_i. When a dead-end occurs on a domain of a variable x_i, BJ jumps back to the deepest variable, say x_j, against which the consistency of x_i is checked. However, if there are no more values remaining for x_j, BJ perform a simple backtrack to the last assigned variable before assigning x_j.[2] Dechter [DEC 90, DEC 02] presented the GBJ algorithm, a generalization of the BJ algorithm. Basically, GBJ attempts to jump back directly to the source of the failure by using only information extracted from the constraint graph. Whenever a dead-end occurs on a domain of the current variable x_i, GBJ jumps back to the most recent assigned variable (x_j) adjacent to x_i in the constraint graph. Unlike BJ, if a dead-end occurs again on a domain of x_j, GBJ jumps back to the most recent variable x_k

2 BJ cannot execute two “jumps” in a row, only performing steps back after a jump.

connected to x_i or x_j. Prosser [PRO 93] proposed the CBJ that rectifies the bad behavior of Gaschnig's algorithm.

The pseudo-code of CBJ is illustrated in algorithm 1.2. Instead of recording only the *deepest* variable, for each variable x_i CBJ records the set of variables that were in conflict with some assignment of x_i. Thus, CBJ maintains an *earliest minimal conflict set* for each variable x_i (i.e. $EMCS[i]$) where it stores the variables involved in the earliest violated constraints with an assignment of x_i. Whenever a variable x_i is chosen to be instantiated (line 3), CBJ initializes $EMCS[i]$ to the empty set. Next, CBJ initializes the current domain of x_i to its initial domain (line 5). Afterward, a consistent value v_i with the current search state is looked for the selected variable x_i. If v_i is inconsistent with the current partial solution, then v_i is removed from the current domain $D(x_i)$ (line 13), and x_j such that c_{ij} is the earliest violated constraint by the new assignment of x_i (i.e. $x_i = v_i$) is then added to the earliest minimal conflict set of x_i, that is $EMCS[i]$ (line 15). $EMCS[i]$ can be seen as the subset of the past variables in conflict with x_i. When a dead-end occurs on the domain of a variable x_i, CBJ jumps back to the last variable, say x_j, in $EMCS[i]$ (lines 16, 9 and 10). The information in $EMCS[i]$ is earned upwards to $EMCS[j]$ (line 11). Hence, CBJ performs a form of "intelligent backtracking" to the source of the conflict allowing the search procedure to avoid rediscovering the same failure due to the same reason.

Algorithm 1.2. *The conflict-directed backjumping algorithm.*

```
procedure CBJ($\mathcal{I}$)
01.  if ( isFull($\mathcal{I}$) ) then report $\mathcal{I}$ as solution;      /* all variables are assigned in $\mathcal{I}$ */
02.  else
03.     choose $x_i$ in $\mathcal{X} \setminus$ vars($\mathcal{I}$) ;                /* let $x_i$ be an unassigned variable */
04.     $EMCS[i] \leftarrow \emptyset$ ;
05.     $D(x_i) \leftarrow D^0(x_i)$ ;
06.     foreach ( $v_i \in D(x_i)$ ) do
07.        $x_i \leftarrow v_i$;
08.        if ( isConsistent($\mathcal{I} \cup (x_i = v_i)$) ) then
09.           $CS \leftarrow$ CBJ($\mathcal{I} \cup \{(x_i = v_i)\}$) ;
10.           if ( $x_i \notin CS$ ) then return $CS$ ;
11.           else $EMCS[i] \leftarrow EMCS[i] \cup CS \setminus \{x_i\}$ ;
12.        else
13.           remove $v_i$ from $D(x_i)$ ;
14.           let $c_{ij}$ be the earliest violated constraint by $(x_i = v_i)$;
15.           $EMCS[i] \leftarrow EMCS[i] \cup x_j$ ;
16.     return $EMCS[i]$ ;
```

When a dead-end occurs, the CBJ algorithm jumps back to address the culprit variable. During the BJ process, CBJ erases all assignments that were obtained since and then wastes a meaningful effort made to achieve these assignments. To overcome this drawback, Ginsberg have proposed DBT [GIN 93].

1.2.1.3. *Dynamic backtracking*

In the naive chronological of BT, each time a dead-end occurs the algorithm attempts to change the value of the most recently instantiated variable. Intelligent BT algorithms were developed to avoid the trashing problem caused by the BT. Although these algorithms identify and then jump directly to the responsible dead-end ($culprit$), they erase a great deal of the work performed thus far on the variables that are backjumped over. When backjumping, all variables between the culprit variable responsible for the dead-end and the variable where the dead-end occurs will be re-assigned.

Ginsberg proposed the DBT algorithm in order to keep the progress performed before BJ [GIN 93]. In DBT, the assignments of non-conflicting variables are preserved during the BJ process. Thus, the assignments of all variables following the culprit are kept and the culprit variable is moved so as to be the last among the assigned variables.

To detect the culprit of the dead-end, CBJ associates a conflict set ($EMCS[i]$) with each variable (x_i). $EMCS[i]$ contains the set of the assigned variables whose assignments are in conflict with a value from the domain of x_i. In a similar way, DBT uses nogoods to justify the value elimination [GIN 93]. Based on the constraints of the problem, a search procedure can infer inconsistent sets of assignments called nogoods.

DEFINITION 1.13.– *A* nogood *is a conjunction of individual assignments, which has been found inconsistent either because of the initial constraints or because of searching for all possible combinations.*

EXAMPLE 1.2.– The following nogood $\neg[(x_i = v_i) \wedge (x_j = v_j) \wedge \ldots \wedge (x_k = v_k)]$ means that assignments it contains are not simultaneously allowed because they cause an inconsistency.

DEFINITION 1.14.– *A* directed nogood *ruling out value v_k from the initial domain of variable x_k is a clause of the form $x_i = v_i \wedge x_j = v_j \wedge \ldots \rightarrow x_k \neq v_k$, meaning that the assignment $x_k = v_k$ is inconsistent with the assignments $x_i = v_i, x_j = v_j, \ldots$. When a nogood (ng) is represented as an implication (directed nogood), the* left-hand side*,* `lhs`(ng)*, and the* right-hand side*,* `rhs`(ng)*, are defined from the position of* $\rightarrow$.

In DBT, when a value is found to be inconsistent with previously assigned values, a directed nogood is stored as a justification of its removal. Hence, the current domain $D(x_i)$ of a variable x_i contains all values from its initial domain that are not ruled out by a stored nogood. When all values of a variable x_i are ruled out by some nogoods (i.e. a dead-end occurs), DBT resolves these nogoods producing a new nogood ($newNogood$). Let x_j be the last variable in the left-hand side of all these nogoods and $x_j = v_j$. In CBJ algorithm, x_j is the culprit variable. The `lhs`$(newNogood)$ is the conjunction of the left-hand sides of all nogoods except

$x_j = v_j$ and $\texttt{rhs}(newNogood)$ is $x_j \neq v_j$. Unlike the CBJ, DBT only removes the current assignment of x_j and keeps assignments of all variables between it and x_i because they are consistent with former assignments. Therefore, the work done when assigning these variables is preserved. The culprit variable x_j is then placed after x_i and a new assignment for it is searched for because the generated nogood ($newNogood$) eliminates its current value (v_j).

Because the number of nogoods that can be generated increases monotonically, recording all of the nogoods, as is done in dependency-directed backtracking algorithm [STA 77], requires an exponential space complexity. To keep a polynomial space complexity, DBT stores only nogoods compatible with the current state of the search. Thus, when BT to x_j, DBT destroys all nogoods containing $x_j = v_j$. As a result, with this approach, a variable assignment can be ruled out by at most one nogood. Because each nogood requires $O(n)$ space and there are at most nd nogoods, where n is the number of variables and d is the maximum domain size, the overall space complexity of DBT is in $O(n^2d)$.

1.2.1.4. *Partial order dynamic backtracking*

Instead of BT to the most recently assigned variable in the nogood, Ginsberg and McAllester [GIN 94] proposed the *partial order dynamic backtracking* (PODB), an algorithm that offers more freedom than DBT in the selection of the variable to put on the right-hand side of the generated nogood. PODB is a polynomial space algorithm that attempted to address the rigidity of DBT.

When resolving the nogoods that led to a dead-end, DBT always selects the most recently assigned variable among the set of inconsistent assignments to be the right-hand side of the generated directed nogood. However, there are clearly many different ways of representing a given nogood as an implication (directed nogood). For example, $\neg[(x_i = v_i) \wedge (x_j = v_j) \wedge \cdots \wedge (x_k = v_k)]$ is logically equivalent to $[(x_j = v_j) \wedge \cdots \wedge (x_k = v_k)] \rightarrow (x_i \neq v_i)$ meaning that the assignment $x_i = v_i$ is inconsistent with the assignments $x_j = v_j, \ldots, x_k = v_k$. Each directed nogood imposes ordering constraints called the set of *safety conditions* for completeness [GIN 94]. Since all variables on the left-hand side of a directed nogood participate in eliminating the value on its right-hand side, these variables must precede the variable on the right-hand side.

DEFINITION 1.15.– *The* safety conditions *imposed by a directed nogood ng, that is $S(ng)$, ruling out a value from the domain of x_j are the set of assertions of the form $x_k \prec x_j$, where x_k is a variable in the left-hand side of ng, that is $x_k \in \texttt{lhs}(ng)$.*

The PODB attempts to offer more freedom in the selection of the variable to put on the right-hand side of the generated directed nogood. In PODB, the only restriction to respect is that the partial order induced by the resulting directed nogood must safeguard the existing partial order required by the set of safety conditions, say S. In a later study, Bliek [BLI 98] shows that PODB is not a generalization of DBT and then proposes the *generalized partial order dynamic backtracking* (GPODB), a new algorithm that generalizes both PODB and DBT. To achieve this, GPODB follows the same mechanism of PODB. The difference between the two (PODB and GPODB) resides in the obtained set of safety conditions S' after generating a new directed nogood ($newNogood$). The new order has to respect the safety conditions existing in S'. While S and S' are similar for PODB, when computing S', GPODB relaxes all safety conditions from S of the form: $\texttt{rhs}(newNogood) \prec x_k$. However, both algorithms generate only directed nogoods that satisfy the already existing safety conditions in S. To the best of our knowledge, no systematic evaluation of either PODB or GPODB has been reported.

All algorithms presented so far incorporate a form of look-back scheme. Avoiding possible future conflicts may be more attractive than recovering from them. In the BT, BJ and DBT, we cannot detect that an instantiation is unfruitful until all variables of the conflicting constraint are assigned. Intuitively, each time a new assignment is added to the current partial solution (instantiation), one can look ahead by performing a forward check of consistency of the current partial solution .

1.2.1.5. *Forward checking*

The FC algorithm [HAR 79, HAR 80] is the simplest procedure of checking every new instantiation against the future (as yet uninstantiated) variables. The purpose of the FC is to propagate information from assigned to unassigned variables. Then, it is classified among those procedures performing a look-ahead.

The pseudo-code of FC procedure is presented in algorithm 1.3. FC is a recursive procedure that attempts to foresee the effects of choosing an assignment on the not-yet- assigned variables. Each time a variable is assigned, FC checks forward the effects of this assignment on the domains of future variables (`Check-Forward` call, line 6). So, all values from the domains of future variables, which are inconsistent with the assigned value (v_i) of the current variable (x_i), are removed (line 11). Future variables concerned by this filtering process are only those sharing a constraint with x_i, the current variable being instantiated (line 10). Incidentally, each domain of a future variable is filtered in order to keep only consistent values with past variables (variables already instantiated). Hence, FC does not need to check consistency of new assignments against already instantiated ones as opposed to chronological BT. The FC is then the easiest way to prevent assignments that guarantee later failure.

We illustrate the FC algorithm on the 4-queens problem (Figure 1.6). In the first iteration, the FC algorithm selects the first value of the domain (1), (i.e. ($q_1 = 1$)). Once, value 1 is assigned to q_1, FC checks forward this assignment. Thus, all values from domain of variables not yet instantiated sharing a constraint with q_1 (i.e. q_2, q_3 and q_4) will be removed if they are inconsistent with the assignment of q_1. Thus, the check-forward results in the following domains: $D(q_2) = \{3, 4\}$, $D(q_3) = \{2, 4\}$ and $D(q_4) = \{2, 3\}$. In the second iteration, the algorithm selects the first available value on the domain of q_2 (i.e. 3), then FC checks forward this new assignment (i.e. $q_2 = 3$). When checking forward ($q_2 = 3$), the assignment is rejected because a dead-end occurs on the $D(q_3)$ as values 2 and 4 for q_3 are not consistent with $q_2 = 3$. Thus, the FC algorithm then chooses $q_2 = 4$, which generates the following domains $D(q_3) = \{2\}$ and $D(q_4) = \{3\}$. Afterward, FC assigns the only possible value (2) for q_3 and checks forward the assignment $q_3 = 2$. The domain of q_4 (i.e. $D(q_4) = \{3\}$) is then filtered. Hence, value 3 is removed from $D(q_4)$ because it is not consistent with $q_3 = 2$. This removal generates a dead-end on $D(q_4)$, requiring another value for q_3. A backtrack to q_2 takes place because there is no possible value on $D(q_2)$. In a similar way, FC backtracks to q_1 requiring a new value.

Algorithm 1.3. *The forward checking algorithm.*

```
procedure ForwardChecking($\mathcal{I}$)
01. if ( isFull($\mathcal{I}$) ) then report $\mathcal{I}$ as solution;          /* all variables are assigned in $\mathcal{I}$ */
02. else
03.     select $x_i$ in $\mathcal{X} \setminus$ vars($\mathcal{I}$) ;               /* let $x_i$ be an unassigned variable */
04.     foreach ( $v_i \in D(x_i)$ ) do
05.         $x_i \leftarrow v_i$;
06.         if ( Check-Forward($\mathcal{I}$, $(x_i = v_i)$), ) then
07.             ForwardChecking($\mathcal{I} \cup \{(x_i = v_i)\}$);
08.         else
09.             foreach ( $x_j \notin$ vars($\mathcal{I}$) such that $\exists\, c_{ij} \in \mathcal{C}$ ) do restore $D(x_j)$ ;
function Check-Forward($\mathcal{I}$, $x_i = v_i$)
10. foreach ( $x_j \notin$ vars($\mathcal{I}$) such that $\exists\, c_{ij} \in \mathcal{C}$ ) do
11.     foreach ( $v_j \in D(x_j)$ such that $(v_i, v_j) \notin c_{ij}$ ) do remove $v_j$ from $D(x_j)$ ;
12.     if ( $D(x_j) = \emptyset$ ) then return false ;
13. return true;
```

A new assignment is generated for q_1 assigning it the next value 2. Next, $q_1 = 2$ is checked forward producing removals on the domains of q_2, q_3 and q_4. The obtained domains are as follows: $D(q_2) = \{4\}$, $D(q_3) = \{1, 3\}$ and $D(q_4) = \{1, 3, 4\}$. Afterward, the next variable is assigned (i.e. $q_2 = 4$) and checked forward producing the following domains: $D(q_3) = \{1\}$ and $D(q_4) = \{1, 4\}$. Next, variables are assigned sequentially without any value removal ($q_3 = 1$ and $q_4 = 3$). Thus, FC has generated a full, consistent instantiation and the solution is $\mathcal{I} = [(q_1 = 2), (q_2 = 4), (q_3 = 1), (q_4 = 3)]$.

The example (Figure 1.6) shows how the FC algorithm improves the BT and FC detects the inconsistency earlier compared to the chronological BT. Thus, FC prunes branches of the search tree that will lead to failure earlier than BT. This purpose allows us to reduce the search tree and (hopefully) the overall amount of time. This can be seen when comparing the size of the search tree of both algorithms on the example of the 4-queens (Figures 1.5 and 1.6). However, we have highlighted that when generating a new assignment, FC requires greater efforts compared to the BT.

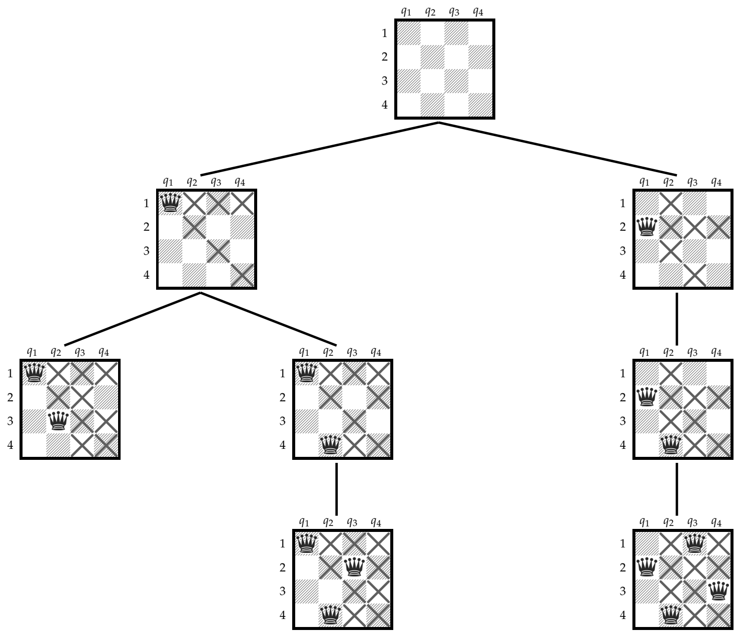


Figure 1.6. *The forward checking algorithm running on the 4-queens problem*

Unlike BT, FC algorithm enables us to prevent assignments that guarantee later failure. This improves the performance of BT. However, FC reduces the domains of future variables, checking only the constraints relating them to variables already instantiated. In addition to these constraints, we can also check the constraints relating future variables to each other. Incidentally, domains of future variables may be reduced and further possible conflicts will be avoided. This is the principle of the *full* look-ahead scheme or constraint propagation. This approach is called MAC.

1.2.1.6. *Arc consistency*

In CSPs, checking the existence of solutions is NP-complete. Therefore, the research community has devoted great interest to studying the *constraint propagation* techniques. Constraint propagation techniques are filtering mechanisms that aim to improve the performance of the search process by attempting to reduce the search space. They have been widely used to simplify the search space before or during the search. Thus, constraint propagation became a central process of solving CSPs [BES 06]. Historically, different kinds of constraint propagation techniques have been proposed: node consistency [MAC 77], AC [MAC 77] and path consistency [MON 74] . The oldest and most commonly used technique for propagating constraints in literature is the AC.

DEFINITION 1.16.– *A value $v_i \in D(x_i)$ is consistent with c_{ij} in $D(x_j)$ iff there exists a value $v_j \in D(x_j)$ such that (v_i, v_j) is allowed by c_{ij}. Value v_j is called a* support *for v_i in $D(x_j)$.*

Let us assume the constraint graph $G = \{X_G, E_G\}$ (see definition 1.2) associated with our CSP.

DEFINITION 1.17.– *An arc $\{x_i, x_j\} \in E_G$* (constraint c_{ij}) *is* arc consistent *iff $\forall\ v_i \in D(x_i), \exists\ v_j \in D(x_j)$ such that (v_i, v_j) is allowed by c_{ij} and $\forall\ v_j \in D(x_j), \exists\ v_i \in D(x_i)$ such that (v_i, v_j) is allowed by c_{ij}. A constraint network is arc consistent iff all its arcs (constraints) are arc consistent.*

A constraint network is arc consistent if and only if for any value v_i in the domain, $D(x_i)$, of a variable x_i there exist in the domain $D(x_j)$ of any adjacent variable x_j a value v_j that is compatible with v_i. Clearly, if an arc $\{x_i, x_j\}$ (i.e. a constraint c_{ij}) is not arc consistent, it can be made arc consistent by simply deleting all values from the domains of the variables in its scope for which there is not a support in the other domain. It is obvious that these deletions maintain the problem solutions since the deleted values are in no solution. The process of removing values from the domain of a variable x_i, when making an arc $\{x_i, x_j\}$ arc consistent is called *revising* $D(x_i)$ with respect to constraint c_{ij}. A wide variety of algorithms establishing AC on CSPs have been developed: AC-3 [MAC 77], AC-4 [MOH 86], AC-5 [VAN 92], AC-6 [BES 93, BES 94], AC-7 [BES 99], AC-2001 [BES 01c], etc. The basic and the most well-known algorithm is Mackworth's AC-3.

We illustrate the pseudo-code of AC-3 in algorithm 1.4. The AC-3 algorithm maintains a queue Q [3] of arcs to render arc consistent. AC-3 algorithm will return true once the problem is made arc consistent or false if an empty domain was generated (a domain is *wiped out*) meaning that the problem is not satisfiable. Initially, Q is filled with all ordered pair of variables that participates in a constraint. Thus, for each constraint c_{ij} ($\{x_i, x_j\}$) we add to Q the ordered pair (x_i, x_j) to revise the domain of x_i and the ordered pair (x_j, x_i) the revise the domain of x_j (line 8). Next, the algorithm loops until it is guaranteed that all arcs have been made arc consistent (i.e. while Q is not empty). The ordered pair of variables are selected and removed one by one from Q to revise the domain of the first variable. Each time an ordered pair of variables (x_i, x_j) is selected and removed from Q (line 10), AC-3 calls function `Revise(`x_i, x_j`)` to revise the domain of x_i. When revising $D(x_i)$ with respect to an arc $\{x_i, x_j\}$ (`Revise` call, line 11), all values that are not consistent with c_{ij} are removed from $D(x_i)$ (lines 2–4). Thus, only values having a support on $D(x_j)$ are kept in $D(x_i)$. The function `Revise` returns true if the domain of variable x_i has been reduced, false otherwise (line 6). If `Revise` results in the removal of values from $D(x_i)$, it can be the case that a value for another variable x_k has lost its support on $D(x_i)$. Thus, all ordered pairs (x_k, x_i) such that $k \neq j$ are added onto Q

3 Other data structures as queue or stack can perfectly serve the purpose.

so long as they are not already on Q in order to revise the domain of x_k. Obviously, the AC-3 algorithm will not terminate as long as there is any pair in Q. When Q is empty, we are guaranteed that all arcs have been made arc consistent. Hence, the constraint network is arc consistent. The while loop of AC-3 can be intuitively understood as constraint propagation process (i.e. propagation the effect of value removals on other domains potentially affected by these removals).

Algorithm 1.4. *The AC-3 algorithm.*

```
function Revise($x_i$, $x_j$)
01. $change \leftarrow$ false;
02. foreach ( $v_i \in D(x_i)$ ) do
03.    if ( $\nexists\ v_j \in D(x_j)$ such that $(v_i, v_j) \in c_{ij}$ ) then
04.       remove $v_i$ from $D(x_i)$ ;
05.       $change \leftarrow$ true;
06. return change ;
function AC-3()
07. foreach ( $\{x_i, x_j\} \in E_G$ ) do                 /* $\{x_i, x_j\} \in E_G$ iff $\exists\ c_{ij} \in \mathcal{C}$ */
08.    $Q \leftarrow Q \cup \{(x_i, x_j); (x_j, x_i)\}$ ;
09. while ( $Q \neq \emptyset$ ) do
10.    $(x_i, x_j) \leftarrow Q.pop()$ ;                 /* Select and remove $(x_i, x_j)$ from $Q$ */
11.    if ( Revise($x_i$, $x_j$) ) then
12.       if ( $D(x_i) = \emptyset$ ) then return false ;   /* The problem is unsatisfiable */
13.       else $Q \leftarrow Q \cup \{ (x_k, x_i) \mid \{x_k, x_i\} \in E_G,\ k \neq i,\ k \neq j \}$ ;
14. return true ;                                        /* The problem is arc consistent */
```

1.2.1.7. *Maintaining arc consistency*

Historically, constraint propagation techniques are used in a preprocessing step to prune values before a search. Thus, the search space that will be explored by the search algorithm is reduced because domains of all variables are refined. Incidentally, subsequent search efforts by the solution method will be reduced. Afterward, the search method can be called for searching a solution. Constraint propagation techniques are also used during search. This strategy is that used by the FC algorithm. FC combines backtrack search with a limited form of AC maintenance on the domains of future variables. Instead of performing a limited form of AC, Sabin and Freuder proposed [SAB 94] the MAC algorithm that establishes and maintains a *full AC* on the domains of future variables.

The MAC algorithm is a modern version of CS2 algorithm [GAS 74]. MAC alternates the search process and constraint propagation steps as is done in FC [HAR 80]. Nevertheless, before starting the search method, MAC makes the constraint network arc consistent. In addition, when instantiating a variable x_i to a value v_i, all the other values in $D(x_i)$ are removed and the effects of these removals are propagated through the constraint network [SAB 94]. MAC algorithm enforces AC in the search process as follows. At each step of the search, a variable assignment is followed by a filtering process that corresponds to enforcing AC. Therefore, MAC maintains the AC each time an instantiation is added to the partial solution. In other words, whenever a value v_i is instantiated to a variable x_i, $D(x_i)$ is reduced momentarily to a single value v_i (i.e. $D(x_i) \leftarrow \{v_i\}$) and the resulting constraint network is then made arc consistent.

Figure 1.7 shows the search process performed by the MAC procedure on the 4-queens problem. Obviously, MAC is able to prune the search space earlier than the FC. This statement can be seen in our example. For instance, when the first queen q_1 is selected to be placed in the first row (i.e. $q_1 = 1$), $D(q_1)$ is restricted to $\{1\}$. Afterward, the conflicts between the current assignment of q_1 and the future variables are removed (i.e. values $\{1, 2\}$, $\{1, 3\}$ and $\{1, 4\}$ are removed respectively from $D(q_2)$, $D(q_3)$ and $D(q_4)$). After that, MAC checks the conflicts among the future variables starting with the first available value (3) for next variable q_2. This, value is removed from $D(q_2)$ since it does not have a support in $D(q_3)$, its only support in $D(q_3)$ was value 1 that is already removed. The MAC algorithm follows with the last value 4 from $D(q_2)$, which has a support in c_{23} (i.e. 2). However, when MAC revises the next variable q_3 this only support (i.e. $2 \in D(q_3)$) for value $4 \in D(q_2)$ will be removed since it does not have a support in $D(q_4)$. Its only support in $D(q_4)$ was 4 that has already been removed from $D(q_4)$. This removal will lead to revisiting $D(q_2)$ and thus removing 4 from $D(q_2)$. A dead-end then occurs and we backtrack to q_1. Hence, value 2 is assigned to q_1. The same process follows until the result is reached on the right subtree.

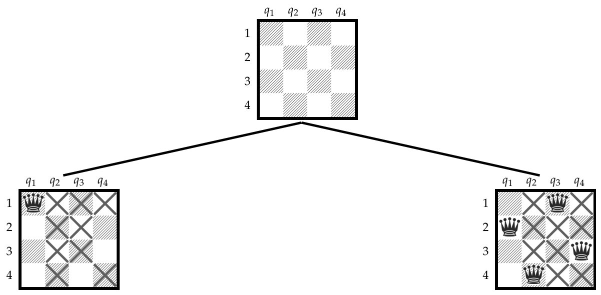


Figure 1.7. *The Maintaing arc consistency algorithm running on the 4-queens problem*

1.2.2. *Variable ordering heuristics for centralized CSPs*

Numerous efficient search algorithms for solving CSPs have been developed. The performance of these algorithms were evaluated in different studies and then shown to be powerful tools for solving CSPs. Nevertheless, because CSPs are in general NP-complete, these algorithms are still exponential. Therefore, a large variety of *heuristics* were developed to improve their efficiency, i.e. search algorithms solving CSPs are commonly combined with heuristics for boosting the search. The literature is rich in heuristics designed for this task. The order in which variables are assigned by a search algorithm was one of the early concerns for these heuristics. The order on variables can be either static or dynamic.

1.2.2.1. *Static variable ordering heuristics*

The first kind of heuristics addressing the ordering of variables was based on the initial structure of the constraint graph. Thus, the order of the variables can be determined prior to the search of solution. These heuristics are called static variable ordering (SVO) heuristics. When presenting the main search procedures (section 1.2), we always assumed, without specifying it each time, an SVO. Therefore, in the previous examples we have always used the lexicographic ordering of variables. That lexicographic ordering can be simply replaced by another ordering more appropriate to the structure of the network before starting the search.

SVO heuristics are heuristics that keep the same ordering on variables all along the search. This ordering is computed in a preprocessing step. Hence, this ordering only exploits (structural) information about the initial state of the search. Examples of such SVO heuristics are:

min-width: the *minimum width* heuristic [FRE 82] chooses an ordering that minimizes the width of the constraint graph. The *width* of a constraint graph is the minimum width over all orderings of variables of that graph. The *width* of an ordering $\mathcal{O}$ is the maximum number of neighbors of any variable x_i that occur earlier than x_i under $\mathcal{O}$. Because minimizing the width of the constraint graph G is NP-complete, it can be accomplished by a greedy algorithm. Hence, variables are ordered from last to first by choosing, at each step, a variable having the minimum number of neighbors (min degree) in the remaining constraint graph after deleting from the constraint graph all variables, which have been already ordered.

max-degree: the *maximum degree* heuristic [DEC 89] orders the variables in a decreasing order of their degrees in the constraint graph (i.e. the size of their neighborhood). This heuristic also aims at, without any guarantee, finding a minimum-width ordering.

max-cardinality: the *maximum cardinality* heuristic [DEC 89] orders the variables according to the initial size of their neighborhood. *max-cardinality* puts in the first position of the resulting ordering an arbitrarily variable. Afterward, other variables are ordered from second to last by choosing, at each step, the most connected variable with previously ordered variables. In a particular case, *max-cardinality* may choose as the first variable the one that has the largest number of neighbors.

min-bandwidth: the *minimum bandwidth* heuristic [ZAB 90] minimizes the bandwidth of the constraint graph. The *bandwidth* of a constraint graph is the minimum bandwidth over all orderings on variables of that graph. The *bandwidth* of an ordering is the maximum distance between any two adjacent variables in the ordering. Zabih claims that an ordering with a small bandwidth will reduce the need for BJ because the culprit variable will be close to the variable where a dead-end occurs. Many heuristic procedures for finding minimum bandwidth orderings have been developed and a survey of these procedures is given in [CHI 82]. However, there is currently little empirical evidence that *min-bandwidth* is an effective heuristic. Moreover, bandwidth minimization is NP-complete.

Another SVO heuristic that tries to exploit the structural information residing in the constraint graph is presented in [FRE 85]. Freuder and Quinn have introduced the use of pseudo-tree arrangement of a constraint graph in order to enhance the research complexity in centralized CSPs.

DEFINITION 1.18.– *A* pseudo-tree *arrangement* $T = (X_T, E_T)$ *of a constraint graph* $G = (X_G, E_G)$ *is a rooted tree with the same set of vertices as* G *(i.e.* $X_G = X_T$*) such that vertices in different branches of* T *do not share any edge in* G*.*

The concept of *pseudo-tree* arrangement of a constraint graph has been introduced to perform searches in parallel on independent branches of the pseudo-tree in order to improve the search in centralized CSPs. A recursive procedure for heuristically building pseudo-trees have been presented by Freuder and Quinn in [FRE 85]. The heuristic aims to select from G_X the minimal subset of vertices named *cutset* whose removal divides G into disconnected sub-graphs. The selected *cutset* will form the first levels of the pseudo-tree, while next levels are built by recursively applying the procedure to the disconnected sub-graphs obtained previously. Incidentally, the connected vertices in the constraint graph G belongs to the same branch of the obtained tree. Thus, the tree obtained is a pseudo-tree arrangement of the constraint graph. Once the pseudo-tree arrangement of the constraint graph is built, several search procedures can be performed in parallel on each branch of the pseudo-tree.

Although SVO heuristics are undoubtedly cheaper because they are computed once and for all, using this kind of variable ordering heuristics does not change the worst-case complexity of the classical search algorithms. On the other hand, researchers have expected that dynamic variable ordering (DVO) heuristics can be more efficient. DVO heuristics were expected to be potentially more powerful because they can take advantage of the information about the current search state.

1.2.2.2. *Dynamic variable ordering heuristics*

Instead of fixing an ordering as is done is SVO heuristics, DVO heuristics determine the order of the variables as search progresses. The order of the variables may then differ from one branch of the search tree to another. It has been shown empirically for many practical problems that DVO heuristics are more effective than choosing a good static ordering [HAR 80, PUR 83, DEC 89, BAC 95, GEN 96]. Hence, researchers in the field of constraint programming had so far mainly focused on such kind of heuristics. Therefore, many DVO heuristics for solving constraint networks have been proposed and evaluated over the years. These heuristics are usually combined with search procedures performing some form of look ahead (see sections 1.2.1.5 and 1.2.1.7) in order to take into account changes on not-yet-instantiated (future) variables.

The guiding idea of the most DVO heuristic is to select the future variable with the smallest domain size. Henceforth, this heuristic is named *dom*. Historically, Golomb and Baumert [GOL 65] were the first to propose the *dom* heuristic. However, it was popularized when it was combined with the FC procedure by Haralick and Elliott [HAR 80]. *dom* investigates the future variables (remaining sub-problem) and provides choosing as next variable the one with the smallest remaining domain. Haralick and Elliott proposed *dom* under the rubric of an intuition called the fail first principle: *"to succeed, try first where you are likely to fail"*. Moreover, they assume that *"the best search order is the one which minimizes the expected length or depth of each branch"* [HAR 80]. Thus, they estimate that minimizing branch length in a search procedure should also minimize search effort.

Many studies have been carried out to understand the *dom* heuristic, a simple but effective heuristic. Following the same principle of Haralick and Elliott saying that search efficiency is due to earlier failure, Smith and Grant [SMI 98] have derived from *dom* new heuristics that detect failures earlier than *dom*. Their study is based on an intuitive hypothesis saying that earlier detection of failure should lead the heuristic to lower search effort. Surprisingly, Smith and Grant's experiments refuted this hypothesis contrary to their expectations. They concluded that increasing the ability to fail early in the search did not always lead to increase its efficiency. In another work, Beck *et al.* (2005) showed that in FC (see section 1.2.1.5) minimizing branch depth is associated with an increase in the branching factor. This can lead FC to perform badly. Nevertheless, their experiments show that minimizing branch depth in MAC (see section 1.2.1.7) reduces the search effort. Therefore, Beck *et al.* do not overlook the principle of trying to fail earlier in the search. They propose to redefine failing early in a such way to combine both the branching factor and the branch depth as was suggested by Nadel [NAD 83] (for instance, minimizing the number of nodes in the failed subtrees).

In addition to the studies that have been carried out to understand the *dom*, considerable research effort has been spent on improving it by suggesting numerous variants. These variants express the intuitive idea that a variable that is constrained with many future variables can also lead to a failure (a dead-end). Thus, these variants attempt to take into account the neighborhood of the variables as well as their domain size. We present in the following a set of well-known variable ordering heuristics derived from *dom*:

dom+deg: a variant of *dom*, *dom+deg*, has been designed in [FRO 94] to break ties when all variables have the same initial domain size. *dom+deg* heuristic breaks ties by giving priority to the variable with the highest *degree* (i.e. the one with the largest number of neighbors).

dom+futdeg: another variant breaking ties of *dom* is the *dom+futdeg* heuristic [BRÉ 79, SMI 99]. Originally, *dom+futdeg* was developed by Brélaz for the graph coloring problem and then applied later to CSPs. *dom+futdeg* chooses a variable with smallest remaining domain (*dom*), but in case of a tie, it chooses from these the variable with the largest future degree, that is the one having the largest number of neighbors in the remaining sub-problem (i.e. among future variables).

dom/deg: both *dom+deg* and *dom+futdeg* use the domain size as the main criterion. The degree of the variables is considered only in case of ties. Alternatively, Bessiere and Régin [BES 96] combined *dom* with *deg* in a new heuristic called *dom/deg*. The *dom/deg* does not give priority to the domain size or degree of variables but uses them equally. This heuristic selects the variable that minimizes the ratio of current domain size to static degree. Bessiere and Régin have been shown that *dom/deg* gives good results in comparison with *dom* when the constraint graphs are sparse but performs badly on dense constraint graphs. They considered a variant of this heuristic which minimizes the ratio of current domain size to future degree *dom/futdeg*. However, they found that the performance of *dom/futdeg* is roughly similar to that of *dom/deg*.

Multi-level-DVO: a general formulation of DVO heuristics that approximates the constrainedness of variables and constraints, denoted *Multi-level-DVO*, have been proposed in [BES 01a]. *Multi-level-DVO* heuristics are considered as neighborhood generalizations of *dom* and *dom/deg* and the selection function for variable x_i they suggested is as follows:

$$H_{\alpha}^{\circledcirc}(x_i) = \frac{\sum\limits_{x_j \in \Gamma(x_i)} (\alpha(x_i) \circledcirc \alpha(x_j))}{\mid \Gamma(x_i) \mid^2}$$

where $\Gamma(x_i)$ is the set of x_i neighbors, $\alpha(x_i)$ can be any syntactical property of the variable such as *dom* or *dom/deg* and $\circledcirc \in \{+, \times\}$. Therefore, *Multi-level-DVO* take into account the neighborhood of variables which have shown to be quite promising. Moreover, they allow using functions to measure the weight of a given constraint.

dom/wdeg: conflict-driven variable ordering heuristics have been introduced in [BOU 04]. These heuristics learn from previous failures to manage the choice of future variables. A *weight* is associated with each constraint. When a constraint leads to a dead-end, its weight is incremented by one. Each variable has a weighted degree, which is the sum of the weights over all constraints involving this variable. This heuristic can simply select the variable with the largest weighted degree (*wdeg*) or incorporating the domain size of variables to give the domain-over-weighted-degree heuristic (*dom/wdeg*). *dom/wdeg* selects among future variables the variable with minimum ratio between current domain size and weighted degree. *wdeg* and *dom/wdeg* (especially *dom/wdeg*) have been shown to perform well on a variety of problems.

In addition to the variable ordering heuristics we presented here, other elegant dynamic heuristics have been developed for centralized CSPs in many studies [GEN 96, HOR 00]. However, these heuristics require extra computation and have only been tested on random problems. On other hand, it has been shown empirically that MAC combined with the *dom/deg* or the *dom/wdeg* can reduce or remove the need for BJ on some problems [BES 96, LEC 04]. Although the variable ordering heuristics proposed are numerous, we have yet to see any of these heuristics to be efficient in every instance of the problems.

Besides different variable ordering heuristics designed to improve the efficiency of search procedure, researchers developed many look-ahead value ordering (LVO) heuristics. This is because value ordering heuristics are a powerful way of reducing the efforts of search algorithms [HAR 80]. Therefore, the constraint programming community developed various LVO heuristics that choose which value to instantiate to the selected variable. Many designed value ordering heuristics attempt to choose the *least constraining* values next, that is the values that are most likely to succeed. Incidentally, values that are expected to participate in many solutions are privileged. Minton *et al.* [MIN 92] designed a value ordering heuristic, the *min-conflicts*, that attempts to minimize the number of constraint violations after each step. Selecting *min-conflicts* values first maximizes the number of values available for future variables. Therefore, partial solutions that cannot be extended will be avoided. Other heuristics try to select values maximizing the product first [GIN 90, GEE 92] or the sum of support in future domain after propagation [FRO 95]. Nevertheless, all these heuristics are costly. Literature is rich on other LVOs, to mention a few [DEC 88, FRO 95, MEI 97, VER 99, KAS 04].

1.3. Summary

We have described in this chapter the basic issues of centralized CSPs. After defining the CSP formalism and presenting some examples of academic and real combinatorial problems that can be modeled as CSPs, we reported the main existing algorithms and heuristics used for solving centralized CSPs.