# CHAPTER 12

# Constraint Satisfaction

What is common between solving a sudoku or a crossword puzzle and placing eight queens on a chessboard so that none attacks another? They are all problems where each number or word or queen placed on the board is not independent of the others. Each constrains some others. Like a piece in a jigsaw puzzle that must conform to its neighbours. Interestingly, all these puzzles can be posed in a uniform formalism, *constraints*. The constraints must be respected by the solution – the constraints must be *satisfied*. And a unified representation admits general purpose solvers. This has given rise to an entire community engaged in *constraint processing*. Constraint processing goes beyond constraint satisfaction, with variations concerned with optimization. And it is applicable on a vast plethora of problems, some of which have been tackled by specialized algorithms like linear programming and integer programming.

In this chapter we confine ourselves to finite domain constraint satisfaction problems (CSPs) and study different approaches to solving them. We highlight the fact that CSP solvers can combine search and logical inferences in a flexible manner.

A constraint network $\mathcal{R}$ or a CSP is a triple,

$$\mathcal{R} = <X, D, C>$$

where $X$ is a set of variable names, $D$ is a set of domains, one for each variable, and $C$ is a set of constraints on some subsets of variables (Dechter, 2003). We will use the names $X = \{x_1, x_2, \ldots, x_n\}$ where convenient with the corresponding domains $D = \{D_1, D_2, \ldots, D_n\}$. The domains can be different for each variable and each domain has values that the variable can take, $D_i = \{a_{i1}, a_{i2}, \ldots, a_{ik}\}$. Let $C = \{C_1, C_2, \ldots, C_m\}$ be the constraints. Each constraint $C_i$ has a scope $S_i \subseteq X$ and a relation $R_i$ that is a subset of the cross product of the domains of the variables in $S_i$. Based on the size of $S_i$, we will refer to the constraints as unary, binary, ternary, and so on. A CSP is often depicted by a constraint graph and a matching diagram, as described in the examples to follow.

We will confine ourselves to finite domain CSPs, in which the domain of each variable is discrete and finite. We will also specify the relations in extensional form well suited for our algorithms. For example, given a common domain {1, 2, 3, 4} for each variable, if we have a



https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

binary constraint between two variables $x_i$ and $x_k$ in which the value in $x_i$ is smaller, then we represent it as

$$R_{ik} = \{<1, 2>, <1, 3>, <1, 4>, <2, 3>, <2, 4>, <3, 4>\}$$

The pairs in the relation are the allowable combination of values for the two variables respectively. For example, $x_i = 1$ and $x_k = 4$ are allowed. Note that we have adopted a naming convention for the relation as well, with the subscripts in $R_{ik}$ referring to the subscripts of the two related variables. We shall focus largely on binary constraint networks (BCNs) in this chapter.

An *assignment* $\mathcal{A}$ is a set of variable–value pairs, for example, $\{x_2 = a_{21}, x_4 = a_{45}, x_7 = a_{72}\}$. We also say that the assignment is an *instantiation* of the set of variables. An assignment to a subset of the variables is a partial assignment. Wherever there is no confusion, we will represent the assignment as a tuple $\mathcal{A} = <a_1, a_2, \ldots, a_p>$ where it is understood that these are the variables $x_1, x_2, \ldots, x_p$ instantiated.

An assignment $\mathcal{A}$ *satisfies* a constraint $C_i$ if $S_i \subseteq \{x_1, x_2, \ldots, x_p\}$ and $\mathcal{A}_{Si} \in R_i$ where $\mathcal{A}_{Si}$ is the projection of $\mathcal{A}$ onto $S_i$.

An assignment $\mathcal{A}$ is consistent if it satisfies all the constraints whose scope is covered by $\mathcal{A}$.

A solution to a CSP is a consistent assignment over all the variables in $X$. The CSP *expresses* the relation $\sigma_X$, also called sol($\mathcal{R}$), the solution relation, which is a relation on all the variables of $X$.

## 12.1 Constraints: Clearing the Fog

The solution sol($\mathcal{R}$) for a CSP is implicit in the network $\mathcal{R}$. Only that it is not explicitly specified. It is specified piecewise, like the description given by the blind men who are touching different parts of an elephant in an ancient parable in which each has only partial knowledge. The local constraints allow assignment of more values than the ones in the solution relation. There is a fog of possibilities that has to be cleared away for the solution to reveal itself. We begin with some examples to understand the problem.

### 12.1.1 The map colouring problem

The map colouring problem is a natural CSP. Political maps in school atlases demarcate the different regions using different colours. Such a colouring is the solution we seek. No two regions that share a boundary can have the same colour. This translates naturally into a set of binary constraints, and the map colouring problem is a BCN. Consider a small map of five regions *A, B, C, D*, and *E*, with the following pairs of regions sharing a boundary: <*A, B*>, <*B, C*>, <*B, D*>, <*C, D*>, and <*D, E*>. In our formulation, each region is a variable in the CSP. Each region has its own set of allowed colours as described in the domains below.

$\mathcal{R} = <X, D, C>$
$X = \{A, B, C, D, E\}, D = \{D_A, D_B, D_C, D_D, D_E\}, C = \{R_{AB}, R_{BC}, R_{BD}, R_{CD}, R_{DE}\}$
$D_A = \{b, g\}, D_B = \{r, b, g\}, D_C = \{b\}, D_D = \{r, b, g\}, D_E = \{r\}$
$R_{AB} = \{<b, r>, <b, g>, <g, r>, <g, b>\}$

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

$R_{BC} = \{<r, b>, <g, b>\}$
$R_{BD} = \{<r, b>, <r, g>, <b, r>, <b, g>, <g, r>, <g, b>\}$
$R_{CD} = \{<b, r>, <b, g>\}$
$R_{DE} = \{<b, r>, <g, r>\}$

Every CSP can be depicted as a *constraint graph*. The nodes in the graph are the variables in the CSP and an edge between two nodes says that the two variables participate in a constraint. This is true even when the constraint is ternary or higher. Constraint graphs are consulted by some algorithms in deciding the order of visiting variables.

Another diagram that is useful is the *matching diagram*. An edge in the matching diagram connects two *values* in two variables that together participate in some constraint. Figure 12.1 shows three views of the map colouring problem. On the left is the map showing the regions that share a boundary. In the centre is the constraint graph, where each region is represented by a node or a variable with an edge between two nodes that share a boundary. In the figure the nodes have the domains shown alongside, and the label on an edge represents the not-equal relation. The two related variables are only allowed different values. On the right is the matching diagram that makes the relation explicit, with every pair of *allowed* values being connected with an edge. Implicit in the matching diagram is the universal relation between nodes not connected in the constraint graph, for example, *A* and *C*. Any combination of values of such pairs of nodes is allowed, though not shown explicitly in the matching diagram.

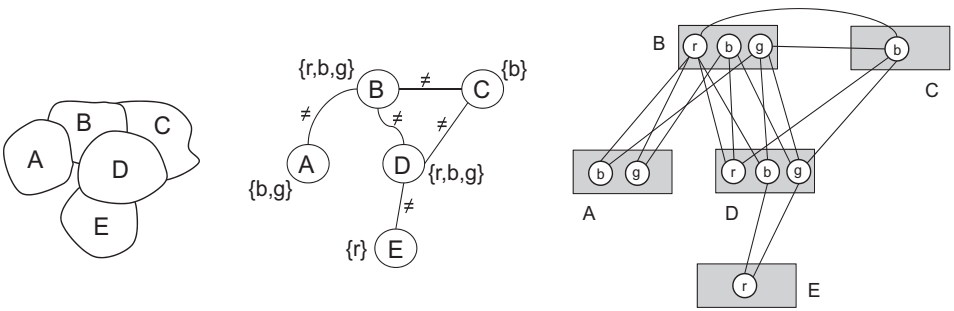



**Figure 12.1** A map colouring problem on regions *A, B, C, D*, and *E* is on the left. The constraint graph is in the centre and the matching diagram on the right. An edge in the matching diagram stands for an allowable pair of colours. For regions that are not adjacent, the matching diagram has an implicit universal relation where any combination of values is allowed.

The matching diagram shows pairs of *values* that can *possibly* occur together in a solution. When the fog clears, only the pairs that are part of a solution are left. We illustrate this phenomenon with the 6-queens problem in the next section.

### 12.1.2 The *N*-queens puzzle

The general task is to place $N$ queens on an $N \times N$ chessboard such that no queen attacks another. A queen attacks another in chess if the two are in the same row, same column, or the same diagonal. We can state this as a binary CSP by specifying constraints between any two queens.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

Thinking of a physical chessboard, the first thought is to have $N^2$ variables for the squares with each possibly having a queen. But we can exploit the knowledge that only one queen can be in one row and one column. This suggests a compact representation that is commonly used. Each row (or each column) can be a variable which will have one queen identified by the column (or row) in which it is. Figure 12.2 shows the 6-queens problem in which a queen has to be placed in each row. The row number becomes the variable, and the column number the value. In this representation there are six variables $X = \{1, 2, 3, 4, 5, 6\}$ and each $D_i = \{a, b, c, d, e, f\}$.

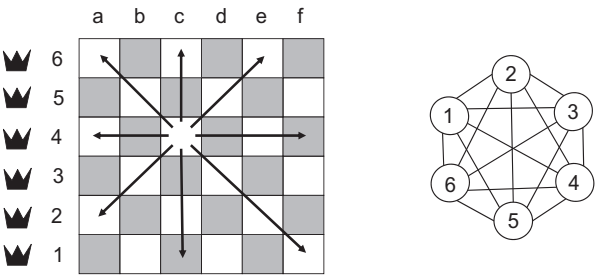



**Figure 12.2** The 6-queens problem is to place the six queens on a 6 × 6 chessboard such that no queen attacks another. The six queens must be on six different rows. We name each row as a *variable*, with the column names as *values*. The arrows show the squares attacked by a queen on square c4. The figure on the right is the constraint graph, which is a complete graph since each queen is constrained by every other queen.

As one can see, the constraint graph, shown on the right, is a complete graph. This is because every queen can potentially be attacked by every other queen. The pairwise allowed values are captured in the relations $C = \{R_{12}, R_{13}, R_{14}, R_{15}, R_{16}, R_{23}, R_{24}, R_{25}, R_{26}, R_{34}, R_{35}, R_{36}, R_{45}, R_{46}, R_{56}\}$. We describe $R_{12}$ and leave the other relations for the reader to complete.

$$
\begin{aligned}
R_{12} = \{&<a, c>, <a, d>, <a, e>, <a, f>,\\
&<b, d>, <b, e>, <b, f>,\\
&<c, a>, <c, e>, <c, f>,\\
&<d, a>, <d, b>, <d, f>,\\
&<e, a>, <e, b>, <e, c>,\\
&<f, a>, <f, b>, <f, c>, <f, d>\}
\end{aligned}
$$

Figure 12.3 shows a part of the matching diagram. The relations covered in the diagram are $R_{12}$, $R_{13}$, $R_{14}$, $R_{15}$, $R_{16}$, $R_{25}$, and $R_{36}$. Even with this subset of relations, one can see that there is a large number of combinations to choose from. As one can see, there is verily a fog of connections for each variable.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

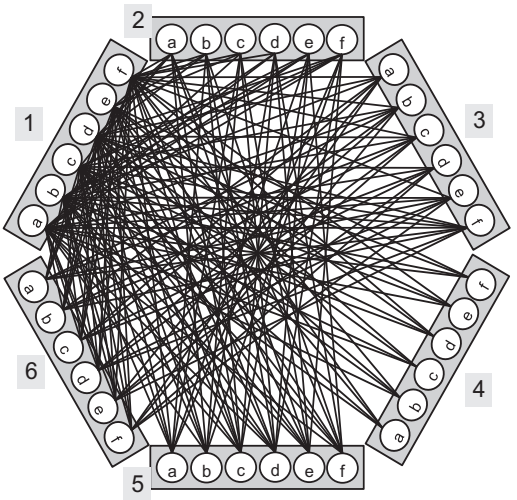



**Figure 12.3** The matching diagram for the 6-queens problem. Only edges for the relations $R_{12}$, $R_{13}$, $R_{14}$, $R_{15}$, $R_{16}$, $R_{25}$, and $R_{36}$ are drawn in the figure, giving rise to the higher density of edges on the left. A close scrutiny will reveal that there are three of four edges from a value for one variable to values in another variable.

In the solution, one value must be selected from the domain of each variable. Further, each value in each variable must have an edge connected to a value in every other variable that must be in the solution. The task of solving the CSP is to clear the fog and reveal the solution. Figure 12.4 shows one solution for the 6-queens problem.

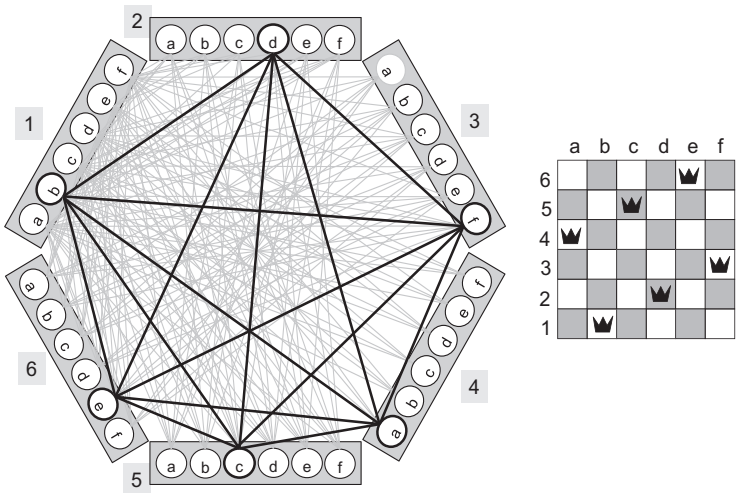



**Figure 12.4** A solution <*b*, *d*, *f*, *a*, *c*, *e*> for the 6-queens problem highlighted on the matching diagram. The solution is also shown on the board on the right.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

The solution <*b, d, f, a, c, e*> is also shown in the figure on the right.

Now we turn our gaze towards solving CSPs. The algorithms we are interested in are domain independent in nature, exemplifying the spirit of this book. The idea is again that users can pose their problems as a CSP, and then use a general off-the-shelf solver for solving the CSP. There is a two pronged strategy for solving a CSP. One is search. The idea here is that one picks the variables one at a time and assigns a value to the variable, which is consistent with earlier variables. The main problem faced by brute force search is combinatorial explosion, and we look at methods to mitigate that. The second is *consistency enforcement* or *constraint propagation*, which aims to prune the space being searched. Done to the extreme this can obviate the need for search altogether, but at a considerable cost. In practice, a judicious combination of the two works best. We begin with search.

## 12.2 Algorithm Backtracking

Search is the soul of solving CSPs. Most algorithms employ depth first search (DFS) wherein the algorithm picks one variable at a time and attempts to assign a consistent value to it. Which variable to pick next and which value to try for the variable will be questions we will address as we go along. For the moment we assume that the order $(x_1, x_2, \ldots, x_N)$ of the variables is given in advance for a CSP with $N$ variables.

The well known algorithm *Backtracking* is described below. The inputs to the algorithm are the three constituents of the CSP, the set of variables $X$, their domains $D$, and the constraints C. It builds an assignment $\mathcal{A}$ incrementally starting from scratch. The value for the next variable must satisfy any constraints that are defined over that variable and its predecessors. That is, $\mathcal{A}$ must be consistent at all times. If it cannot be extended to the next variable, then the algorithm backtracks and tries a different value for the last variable it assigned a value to. When it considers a new variable, it makes a copy of its domain and passes it to function *SelectValue* along with the partial assignment $\mathcal{A}$ constructed so far. SelectValue sifts through the values in the domain till it finds a value consistent with $\mathcal{A}$. The parent Backtracking accepts that value, augments the assignment, and moves on to the next variable. If SelectValue cannot find a consistent value, then Backtracking retreats to the previous variable and looks for another value.

**Algorithm 12.1.** Given an ordering of the variables, algorithm Backtracking picks variables one by one and incrementally builds the assignment $\mathcal{A}$. Function SelectValue takes a copy of the domain of the $i^{th}$ variable, the current assignment, and removes values that are not consistent with $\mathcal{A}$. When it finds a consistent value it returns the value to Backtracking which moves on to the next variable. If it cannot find a consistent value, Backtracking backtracks to look for another value for the previous variable.

Backtracking {X, D, C)
1. $\mathcal{A} \leftarrow []$
2. $i \leftarrow 1$
3. $D'_i \leftarrow D_i$

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

```
4.   while 1 ≤ i ≤ N
5.       a_i ← SELECTVALUE (D'_i, 𝒜, C)
6.       if a_i = null
7.          then   i ← i − 1
8.                 𝒜 ← tail 𝒜
9.          else   𝒜 ← a_i : 𝒜
10.                i ← i+1
11.                if i ≤ N
12.                   then D'_i ← D_i
13.  return REVERSE(𝒜)

SELECTVALUE(D'_i, 𝒜, C)
1.   while D'_i is not empty
2.         a_i ← head D'_i
3.         D'_i ← tail D'_i
4.         if CONSISTENT(a_i : 𝒜)
5.                then return a_i
6.   return null
```

Figure 12.5 shows the progress on the tiny map colouring problem from Figure 12.1. The order of variables is alphabetic. The very first choices for the variables *A, B*, and C are accepted, but when it comes to variable *D* only the third choice *g* works.

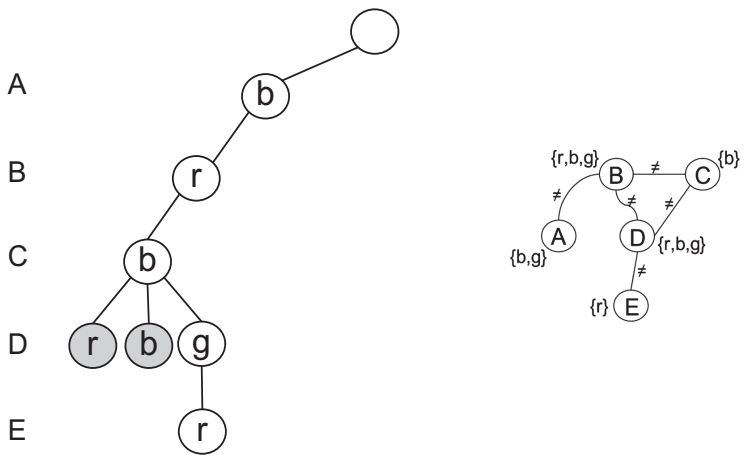



**Figure 12.5** BACKTRACKING does depth first search on the problem from Figure 12.1 and finds the solution <*b*, *r*, *b*, *g*, *r*>. On the way SELECTVALUE has rejected the values $D = r$ and $D = b$. The constraint graph is shown on the right.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

The order of processing variables will clearly impact the complexity of the search. There are essentially two approaches to deciding this order. One is a static approach that looks at the topology of the constraint graph to choose an order with fewer dead ends. We look at that next. The other is to dynamically choose the next variable to try, in tandem with constraint propagation. We will describe that after looking at the consistency enforcement algorithms.

### 12.2.1 Static variable ordering

The choice of a value for a variable is constrained by the other variables it is related to. If a variable $X$ is connected to only one other variable $Y$, then the moment one chooses a value for $Y$, a value for $X$ can be chosen, and that would be final. But imagine a variable $U$ related to three other variables $X$, $Y$, and $Z$. Then choosing values for $X$, $Y$, and $Z$ first may not leave a consistent value for $Z$. For example, if the domain of all four variables is $\{r, b, g\}$, then choosing $X = r$, $Y = b$, and $Z = g$ leaves no value for $U$. But choosing a value $U = r$ first allows for a choice of two values for each of $X$, $Y$, and $Z$. One can then hypothesize that variables of higher degree (connected to more other variables) should be assigned values earlier. This topological argument is the reason for choosing an ordering based on the degrees of the nodes. We begin with some definitions.

Given a CSP $<X, D, C>$ and an ordering $O$ of the variables $(x_1, x_2, \ldots, x_N)$, the *width* of a node in the ordering is the number of *parents* that it is connected to. A node $x_i$ is the parent of a node $x_k$ if the two have an edge between them in the constraint graph, and $x_i$ precedes $x_k$ in the ordering. The width of an ordering is the maximum of the width of all nodes in that ordering.

A min-width ordering of a graph is an ordering which has the lowest width (Dechter, 2003). A greedy algorithm described below produces the min-width ordering. It begins with an empty list $O$. In each cycle the algorithm plucks a node with the smallest degree along with its edges from the graph and concatenates it to $O$. As a result, the nodes with the smallest degree are placed in the end of the order, and the nodes with the largest degree at the front.

**Algorithm 12.2.** Algorithm MinWidth accepts a graph $<V, E>$ with $N$ nodes and returns a min-width ordering of the graph.

```
MinWidth (Graph = <V, E>, N)
1.    O ← []
2.    for i = N downto 1
3.        v ← a node in V with the smallest degree
4.        O ← v : O
5.        remove v from V
6.        remove edges to v from E
7.    return O
```

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

When a node has multiple parents, perhaps constraints can be imposed between them. Consider the example of variable $U$ having variables $X$, $Y$, and $Z$ as parents. Searching for values in the given order one can see that if $X$, $Y$, and $Z$ were originally unrelated, then adding the constraints $X = Z$, $X = Y$, and $Y = Z$ would have made finding a value for $U$ easier. One would not have to backtrack and try different values for $X$ or $Y$ or $Z$. As we shall see later, adding such constraints with the goal of minimizing backtracking is a strategy in consistency enforcement. It would be desirable to enforce enough consistency to make the search backtrack free. But the cost of achieving that could outweigh the savings.

In this context one can introduce the notion of an induced graph with an induced width, in which edges are added connecting parents of nodes in the order being imposed. Unfortunately, finding a min induced width ordering is NP-complete, but the following greedy algorithm often produces very good ones (Dechter, 2003). The greedy algorithm below is similar to Algorithm 12.2 except that before removing the selected node $v$ from the graph, all its parents are connected pairwise.

**Algorithm 12.3.** Algorithm *MinInducedWidth* is similar to MinWidth except that before plucking the node $v$ from the graph all its parents are connected pairwise.

```
MinInducedWidth (Graph = <V, E>, N)
1.    O ← []
2.    for i = N downto 1
3.        v ← a node in V with the smallest degree
4.        O ← v: O
5.        connect each pair of parents of v
6.        remove v from V
7.        remove edges to v from E
8.    return O
```

A variation that often performs better is to choose the node to be plucked using a different criterion. Instead of selecting the node with the lowest degree, one picks a node which has a minimum number of unconnected parents. Then only a few new edges will need to be added. This algorithm is called *MinFill*.

Figure 12.6 shows a few orderings for a small graph with seven nodes $X = \{A, B, C, D, E, F, G\}$ shown on the top. The first ordering in the figure is the alphabetic ordering ($A, B, C, D, E, F, G$). With this ordering Backtracking would assign a value for variable $A$ first and variable $G$ last. The alphabetic ordering has a width 3, because node $E$ has three parents $A$, $B$, and $D$. The second ordering is reverse alphabetic and has width 4 since $A$ has degree 4. The third ordering is the one produced by the MinWidth algorithm and has a width 2. The last one is the one produced by the MinInducedWidth algorithm. It also has a width 2, but has an additional edge connecting $D$ and $G$, the parents of $F$ which occurs later in the ordering.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

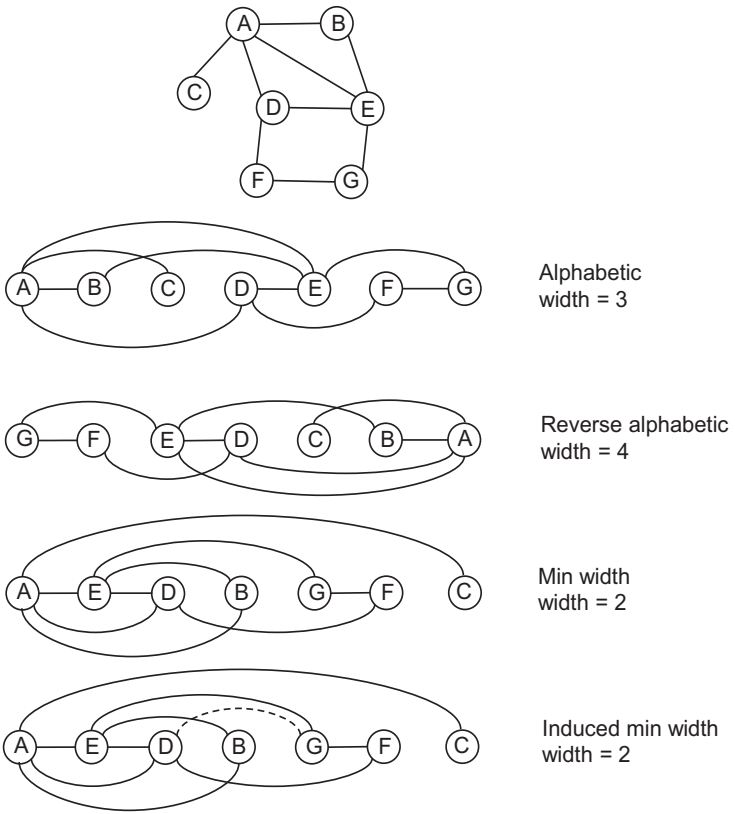



**Figure 12.6** A graph and some orderings. Both the min-width and min-induced-width orderings have a width 2. The alphabetic ordering has a width 3, and the reverse alphabetic ordering has the maximum width possible 4.

The reader should verify that if the given graph were to be a tree, then both the algorithms will produce an ordering of width 1. When we have a CSP ordering of width 1, then it is possible to do backtrack free search. This is because each node is constrained by only one parent who already has a value.

If the graph has cycles, then the minimum width possible is 2. This is the case for the example above.

### 12.2.2 Dynamic variable ordering

The search algorithm backtracks when it cannot find a value for a variable consistent with the earlier variables. This is because *all* values available in the domain of the variable may be conflicting with values assigned to earlier variables. This could be because there are too many parents, like when $X$, $Y$, and $Z$ are parents of variable $U$. But this could be also because there are too few values left in the domain of the current variable. For example, if the domain of $U$ has only one variable, then it could easily conflict with earlier variables. One strategy would be to assign

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

a value to this variable before considering the others. This is the approach taken in *dynamic variable ordering*, where the order in which variables are processed is decided on the fly.

This becomes even more relevant when the domains of future variables are pruned by the algorithm. We illustrate this with a cursory description of the algorithm ForwardChecking discussed later in more detail. The crux of the algorithm is that when it considers a value for a variable, it deletes values from future variables that would become inconsistent with the current assignment. We illustrate this with the small map colouring example from Figure 12.1. The domains and the constraints are reproduced below.

$D_A = \{b, g\}, D_B = \{r, b, g\}, D_C = \{b\}, D_D = \{r, b, g\}, D_E = \{r\}$
$R_{AB} = \{<b, r>, <b, g>, <g, r>, <g, b>\}$
$R_{BC} = \{<r, b>, <g, b>\}$
$R_{BD} = \{<r, b>, <r, g>, <b, r>, <b, g>, <g, r>, <g, b>\}$
$R_{CD} = \{<b, r>, <b, g>\}$
$R_{DE} = \{<b, r>, <g, r>\}$

We begin with the variable $C$ which is one of the two with the smallest domains.

1. $C = b$. Region $C$ is adjacent to regions $B$ and $D$. ForwardChecking deletes $b$ from their domains. Now $D_B = \{r, g\}$ and $D_D = \{g, r\}$ after pruning. Next, we consider $E$ which has the smallest domain.
2. $E = r$. Region $E$ is adjacent to $D$, and we prune the domain of $D$ to get $D_D = \{g\}$. Now $D$ becomes the smallest domain.
3. $D = g$. Only $B$ is a future variable related to $D$. $D_B = \{r\}$ after pruning.
4. $B = r$. This value in $B$ does not conflict with the values in $A$.
5. $A$ can be either blue or green.

As seen here, dynamic variable ordering considers those variables first which have the fewest values to choose from. And deleting values from future variables removes potentially conflicting choices. In the process, if a future variable becomes empty, the search algorithm can backtrack from the current variable itself. We will illustrate this in Section 12.4.

## 12.3 Constraint Propagation

A typical CSP describes the constraints in parts. A search algorithm wades through the constraints looking for an assignment. *Backtracking* happens when a partial assignment that satisfies some constraints cannot be extended to another variable and another constraint. For example, given the CSP $<\{X, Y, Z\}, \{D_X = D_Y = D_Z = \{1, 2, 3\}\}, \{R_{XY} = X < Y, R_{YZ} = Y < Z\}$, then choosing $X = 2$ allows us to choose $Y = 3$ but we cannot choose a value of $Z$.

*Constraint propagation* or *consistency enforcement* is the endeavour to *tighten* the CSP so that these kinds of dead ends do not arise. This can be done by pruning domains of variables in the simplest case, or by adding constraints to limit the choices to values that can be part of a solution. Done to an extreme, consistency enforcement can make search backtrack free. But at a prohibitive computational cost. Very often the best approach is to adopt a combination of reasoning and search that is optimal. In this chapter we study a few algorithms for consistency enforcement.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

We begin with the general notion of *i-consistency*. A network $\mathcal{R}$ is said to be *i-consistent* if every consistent assignment to any $i - 1$ variables can be extended to one more variable. A network is said to be *strongly i-consistent* if it is also *j-consistent* for all $j \leq i$.

A node is said to be 1-consistent or node consistent (NC) iff every variable $x$ in the domain satisfies all constraints $R_x$ on the variable. For example, if $R_x = \text{Even}(x)$, then there must be no odd value in any variable. Node consistency can be achieved by inspecting the domains of all variables and removing any values that do not satisfy some constraint.

### 12.3.1 Arc consistency

A variable $X$ is said to be *arc consistent* (AC) with respect to a variable $Y$ if there is an edge $(X, Y)$ in the constraint graph and for every value $a \in D_X$, there exists a value $b \in D_Y$ such that $<a, b> \in R_{XY}$. We say that $a$ supports $b$, and $b$ supports $a$. A simple algorithm REVISE($(X)$, $Y$) makes $X$ arc consistent to $Y$.

**Algorithm 12.4.** Algorithm REVISE prunes the domain of variable $X$, removing any value that is not paired to a matching value in the domain of variable $Y$.

```
REVISE((X), Y)
1.    for every a ∈ D_x
2.       if there is no b ∈ D_Y s.t. <a,b> ∈ R_XY
3.          then delete a from D_x
```

The worst case complexity of REVISE is $\mathcal{O}(k^2)$ where $k$ is the size of each domain. The worst case happens when no value in $X$ has a matching value in $Y$. An edge $(X, Y)$ in a constraint graph is said to be arc consistent iff both $X$ and $Y$ are arc consistent with respect to each other. A constraint network $\mathcal{R}$ is said to be arc consistent if all edges in the constraint graph are arc consistent. A node is said to be 2-consistent if an assignment to any variable can be extended to a consistent assignment to any other variable. Clearly, if a network is 2-consistent, it must be arc consistent as well. A simple brute force algorithm *AC-1* cycles through all edges in the constraint graph until no domain changes (Mackworth, 1977; Mackworth and Freuder, 1985).

**Algorithm 12.5.** Algorithm *AC-1* cycles through all edges repeatedly even if one value is removed from one variable.

```
AC-1 (X, D, C)
1.    repeat
2.       for each edge (x, y) in the constraint graph
3.            REVISE((x), y)
4.            REVISE((y), x)
5.    until no domain changes in the cycle
```

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

Let there be $n$ variables, each with domain of size $k$. Let there be $e$ edges in the constraint graph. Every cycle then has complexity $\mathcal{O}(ek^2)$. In the worst case, the network is not AC, and in every cycle exactly one element in one domain is removed. Then there will be $nk$ cycles. The worst case complexity of *AC-1* is therefore $\mathcal{O}(nek^3)$.

Before improving upon the arc consistency algorithm, we look at how deduction with *modus ponens* can be seen as constraint propagation. Let the knowledge base be $\{P, P \supset Q, Q \supset R, R \supset S\}$. Working with Boolean formulas each propositional variable has two values in its domain, 1 (*true*) and 0 (*false*). The truth table of the binary relation $X \supset Y$ can be represented by the constraint $\supset_{XY} = \{<0, 0>, <0, 1>, <1, 1>\}$. The CSP can then be viewed as

$<X, D, C>$ where

$X = \{P, Q, R, S\}$,

$D_P = D_Q = D_R = D_S = \{0, 1\}$

$R_P = \{<1>\}$

$R_{PQ} = R_{QR} = R_{RS} = \{<0, 0>, <0, 1>, <1, 1>\}$

First achieving node consistency prunes the domain of $P$ to $\{1\}$. Then achieving arc consistency prunes the rest of the variables to also contain only 1. The process is illustrated in Figure 12.7 with matching diagrams.

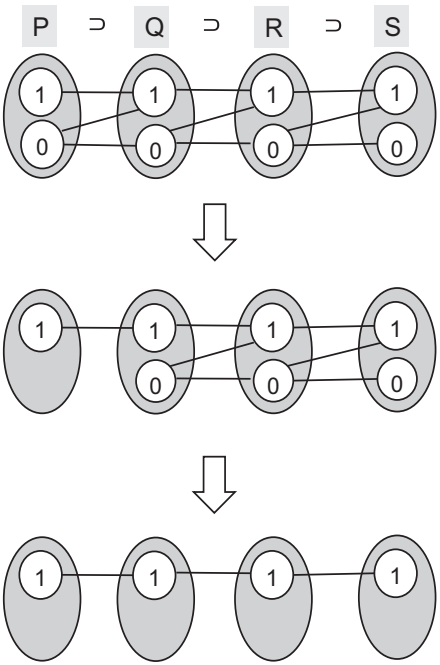



**Figure 12.7** Logical deduction can be seen as consistency enforcement. Given the variables $P$, $Q$, $R$, and $S$ and the constraints defined by the KB = $\{P, P \supset Q, Q \supset R, R \supset S\}$. Node consistency followed by arc consistency results in the domains of all variables having only 1. This amounts to deducing that $Q$, $R$, and $S$ are true.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

The algorithm AC-1 is an overkill. It makes unnecessary calls to REVISE. A better strategy is as follows. If REVISE((*X*), *Y*) removes some value *v* from the variable *X*, one need only check that all edges connected to *X* are still arc consistent. It is possible that the value *v* was the only support for some value *w* in a variable *W*. Then a call to REVISE((*W*), *X*) is needed. This is done by algorithm *AC-3* that pushes all such connected pairs of variables into a queue. A change in a variable is *propagated* to the connected variables. *Only those* are considered again for a call to REVISE.

**Algorithm 12.6.** Algorithm AC-3 begins by invoking REVISE for all edges in the constraint graph. After that, if the domain of a variable *P* has changed, then consistency w.r.t. *P* is enforced for all neighbours of *P*.

```
AC-3(X, D, C)
 1.   Q ← []
 2.   for each edge (N,M) in the constraint graph
 3.          Q ← Q ++ (N,M) : [(M,N)]
 4.   while Q is not empty
 5.      (P,T) ← head Q
 6.      Q ← tail Q
 7.         REVISE((P), T)
 8.         if Dp has changed
 9.              for each R ≠ T and (R,P) in the constraint graph
10.                Q ← Q ++ [(R,P)]
```

The complexity of AC-3 is $\mathcal{O}(ek^3)$ where $e$ is the number of edges and each domain is of size $k$. Of this, $k^2$ comes from REVISE. For each of the $e$ edges, it makes $2k$ calls to REVISE in the worst case if it deletes all values from the two connected variables.

One can be more frugal if one realizes that the call to REVISE can itself be an overkill. Just because a value $v$ has been deleted from a variable[1] $X$ why should one make a call REVISE ((*Y*), *X*) to check if every value in $Y$ is still supported by values in $X$? If one could keep track of the values in $Y$ that were being supported by $v \in D_X$, then if $v$ were the *only* support of a value $w \in D_Y$, then one can go ahead and delete $w$ from $D_Y$. Following this, we will have to check if $w$ in turn was the only support for some value in a connected variable. This is done by algorithm *AC-4* which, however, needs more bookkeeping to be done to keep track of individual support from values. The following data structures are used. Let $\mathcal{R} = <X, D, C>$ be the network, and let $x$ and $y$ be variables in $X$ (Dechter, 2003).

– The support set $S$ is a set of sets, one for each variable–value pair $<x, a>$, named $S_{<x, a>}$. For each variable–value pair $<x, a>$ the support set contains a list of supporting pairs from other variables. When a value $a \in D_x$ is deleted the set $S_{<x, a>}$ is instrumental in checking which values in other variables might have lost a support.

[1] We often say 'from a variable *X*' as a short form for 'from the domain of a variable *X*'.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

$S_{<x,a>} = \{ <y, b> \mid y \in X, b \in D_y \text{ and } <a, b> \in R_{xy} \}$

Given $e$ constraints and domain sizes $k$, computing $S$ is $\mathcal{O}(ek^2)$.

– Counter array *counter*. For each value $a \in D_x$, denoted by $<x, a>$, the counter array maintains the *number* of supports from a variable $y$. If the counter value becomes zero, then the value $a$ has to be removed from $D_x$.

  Counter$(x, a, y)$ = the number of values in $D_y$ that support the pair $<x, a>$

  The counter array can be constructed along with $S$, adding a constant amount of computation for each label.

– A queue $Q$ of labels without support that need to be processed. The unsupported variable–value pairs are added to this as and when they are identified.

The algorithm AC-4 begins by inspecting the network $\mathcal{R}$ setting up the records of links from the matching diagram in the set $S$, and a count of *how many* values from a variable $y$ support a value $a \in D_x$. This requires visiting both ends of all the $e$ edges in the network and all $k^2$ combination of values for the two connected variables. This is done in $\mathcal{O}(ek^2)$ time, and also requires $\mathcal{O}(ek^2)$ space. All this work is done upfront.

**Algorithm 12.7.** Algorithm AC-4 begins by inspecting all edges in the matching diagram and identifying the list of all supports for all variable–value pairs, and the count of number of supports for each value from another variable. It deletes a value with count 0 and then decrements the count of all connected values.

```
AC-4(X, D, C)
 1.  Q ← []
 2.  initialize S<x,a> and counter(x, a, y) for each Rxy in C
 3.  for each counter
 4.      if counter(x, a, y) = 0
 5.          Q ← Q ++ <x, a>
 6.  while Q is not empty
 7.      <x, a> ← head Q
 8.      delete a from Dx
 9.      for each <y, b> in S<x,a>
10.          counter(y, b, x) ← counter(y, b, x) − 1
11.          if counter(y, b, x) = 0
12.              Q ← Q ++ <y, b>
```

After the initialization, if there is a missing support for a value $a$ for variable $x$ (from some variable $y$), then $a$ is deleted from $D_x$ and then the set $S$ is inspected to decrement all counters for all related variable–value pairs $<y, b>$. If any counter becomes 0, then that variable–value pair is added to the queue $Q$ of values destined for deletion. In this manner, the propagation is extremely fine grained. Whenever a value is deleted, the algorithm pursues the links in the matching diagram, effectively deleting each such link. Then if a value in some domain is left without a link, that is added to the queue for deletion as well.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

The initialization step that creates the counters and the support pointers requires, at most, $\mathcal{O}(ek^2)$ steps. The number of elements in $S_{< x,a>}$ is of the order of $ek^2$ and each is accessed at most once. Therefore the time and space complexity of AC-4 is $\mathcal{O}(ek^2)$.

What can one say about a CSP on which arc consistency has been achieved and no domain is empty? If the constraint graph is a tree, then the CSP has a solution. This is because each variable is constrained by exactly one variable. Moreover, if one chooses the min-width ordering of the variables, the search will be backtrack free. If the constraint graph is not a tree, then it may be possible that the CSP has no solution. This is illustrated in Figure 12.8 where the network on the left has no solution even though it is arc consistent. But after removing one edge (BC) it becomes a tree, and this has two solutions.

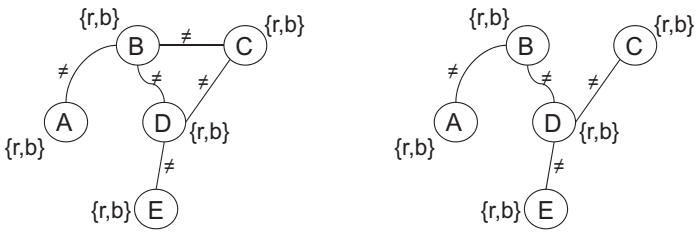



**Figure 12.8** The CSP on the left is arc consistent but does not have a solution. The network on the right is similar to the one on the left except that it has one edge (*B*,*C*) less which makes it a tree. This network has two solutions.

The reader is encouraged to try out different orderings of the network and verify that the min-width ordering for the tree can be solved in a backtrack free manner.

### 12.3.2 The Waltz algorithm

During the early 1970s, a bunch of students at MIT worked on the problem of *interpreting* line drawings. It started with Adolfo Guzman, a graduate student of Marvin Minsky's, who took up the problem of writing a program to look at a line drawing to ascertain how many objects were present in it. David Huffman then limited the line drawings to those of trihedral objects without any cracks and shadows, and where the viewpoint was such that a slight shift did not produce drastic changes in the image. A trihedral object is one in which exactly three straight line edges meet at every vertex. This also means that a vertex was created by three plane surfaces meeting at one point.

The objective was to label each line drawing with one of three kinds of labels. A convex edge is one where two faces are visible and the solid matter subtends an angle less than 180°, like the edge of a cube. Such an edge is to be labelled '+'. A concave edge is one where two faces are visible and the solid matter subtends an angle more than 180°, like the line where two walls in a room meet. This is labelled with '−'. The third kind of label is an arrow, when only one face is visible. The visible face is on the right when one travels along the direction of the arrow. Thus there are two different arrow labels. This task is also known as Huffman–Clowes labelling (Clowes, 1971; Huffman, 1971).

The visible vertices are of four kinds as shown in Figure 12.9.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

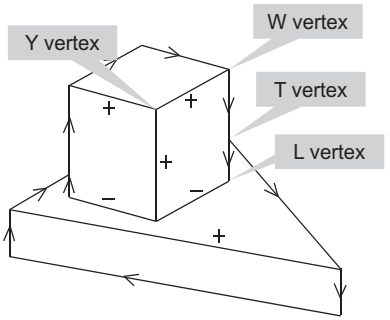



**Figure 12.9** The three kinds of edge labels and four kinds of vertices in trihedral objects.

Each edge in a line drawing can be labelled in one of four ways: +, −, →, and ←. Then a *W* or a *Y* or a *T* vertex can have $4^3 = 64$ different combined labels and an *L* vertex can have $4^2 = 16$. The interesting thing is that for trihedral objects without cracks or shadows, there are only 18 kinds of edge label combinations that are physically possible. These are shown in Figure 12.10.

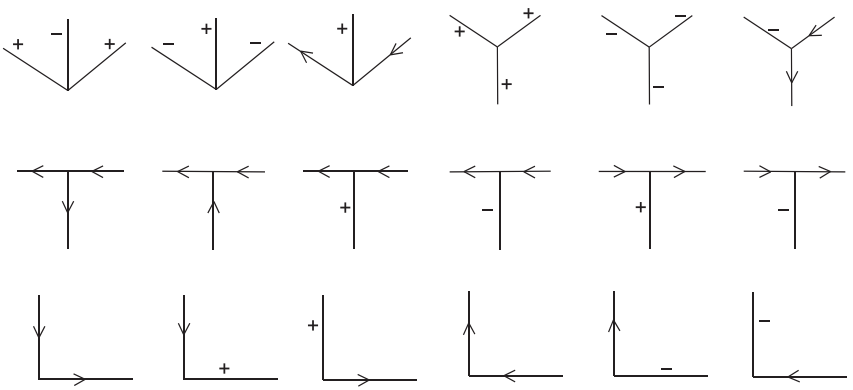


**Figure 12.10** The 18 different kinds of vertices possible in line drawings for trihedral objects without cracks or shadows. Some texts leave out the middle two *T* vertices because that configuration comes from a non-normal viewpoint.

Every edge in a line drawing connects two vertices but it can have only one label. This lays the foundation of constraint propagation. If one knows the label at one end, then that label must be propagated to the other end as well. And at the other end, the other edges impinging on the vertex will be constrained by possibilities shown in Figure 12.10.

The constraint propagation algorithm was written by David Waltz who extended the scope of objects manifold (Waltz, 1975). The WALTZ algorithm, as it is now known, could handle objects with more than 3-edge vertices, objects with cracks, and images with light and shadows. The number of edge labels shot up from 4 to 50-plus, and the number of valid vertices shot

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

up to thousands. The algorithm is somewhere between AC-1 and AC-3 and does propagation from vertex to vertex. We illustrate the propagation with a trihedral object shown on the left in Figure 12.11.

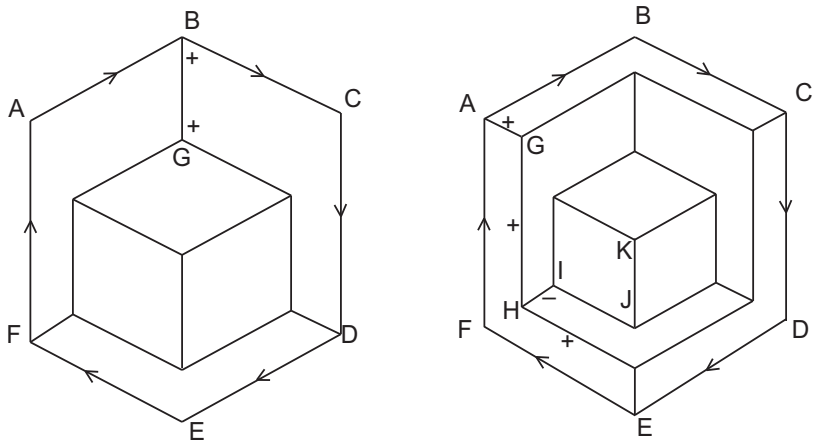



**Figure 12.11** The WALTZ algorithm begins by demarcating the two solid objects and marks the external sequence of edges *A–B–C–D–E–F* with arrows surrounding the solid material. In the diagram on the left the edge *B–G* in the *W* junction at *B* gets a label +. This + label must be the same at the *G* end of the edge *B–G* as well. The other two edges on vertex *G* can now only be +. This is the kind of propagation the WALTZ algorithm does. A similar process is followed in the line diagram on the right, but where one has two choices for the edge 1–*J* during propagation.

Both the objects in Figure 12.11 are solid objects, as careful observation will reveal. The WALTZ algorithm begins by isolating an object from the background. It does so by labelling the outermost edges with arrows, going in the clockwise direction along the outermost lines. In both the objects, these are $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, $D \rightarrow E$, and $E \rightarrow F$. Now there are only three kinds of *W* vertices as shown in Figure 12.10, and only one of them has two arrow labels. Consequently, the third edge in these vertices *must* be convex edges and can be labelled with a '+'.

This is illustrated for the edge *B–G* for the object on the left. We have labelled it twice to emphasize the fact that the label is propagated from the *W* vertex *B* to the *Y* vertex *G*. Now there is only one kind of *Y* vertex that has '+' labels, and all three of the edges impinging on it must be labelled '+'. This label can now be propagated along the two edges emanating from *G* to the connected *W* edges. This process continues and the entire set of edges can be labelled unambiguously.

For the object on the right, the labelling may involve backtracking. The reader should verify that the edge *H–I* must be labelled with a '−'. Now there are two possibilities of labelling the other two edges at vertex *I*. Either both must be '−' or both must be arrows. If *I–J* is a '−',

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

then the edge $J$–$K$ can only be a '+' given the constraints on $W$ vertices. This results in $K$ being labelled with '+++'. If it is to be an arrow, then the direction must be $I \rightarrow J$. But then there is no label possible for the edge $J$–$K$. So if the algorithm were to select $I \rightarrow J$, it would have to backtrack and select the label '−'.

The reader is encouraged to complete the labelling process for both objects.

### 12.3.3 Path consistency

A network $\mathcal{R}$ is said to be *path consistent* (PC) or 3-consistent if a consistent assignment to any two variables can be extended to any third variable. Consider the simple map colouring problem with five regions on the left. As discussed earlier, the network is arc consistent but does not have backtrack free search. The assignment $\{A = r, B = b, C = r\}$ is consistent but cannot be extended to variable $D$. Making it path consistent involves adding a new constraint $R_{BC} = \{<r, r>, <b, b>\}$ to the network as shown on the right. When that happens the assignment $\{A = r, B = b, C = r\}$ is no longer consistent and BACKTRACKING has to choose $C = b$ instead, which allows $D = r$.

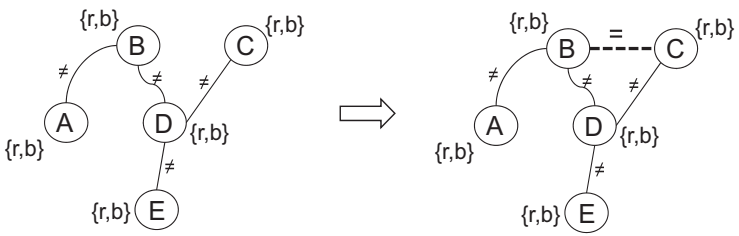



**Figure 12.12** The network on the left is not path consistent because an assignment $<B = r, C = b>$ cannot be extended to the variable $D$. Making it path consistent adds a new constraint $R_{BC} = \{<r, r>, <b, b>\}$ to the CSP. Now the variables $B$ and $C$ are related by the equality relation. Earlier, it was implicitly the universal relation.

The astute reader would have noticed that in the process of making the network 3-consistent we have introduced a new edge in the constraint graph for the relation $B = C$. The reader must also keep in mind that when the vertices $B$ and $C$ were not connected in the constraint graph, it meant that any value in $B$ was locally consistent with any value in $C$. That is, no constraint between $B$ and $C$ was specified, and $R_{BC}$ was a universal relation $\{<r, r>, <r, b>, <b, r>, <b, b>\}$. After the propagation this was pruned to $\{<r, r>, <b, b>\}$, and then an edge $B$–$C$ was introduced in the constraint graph. This is done by the algorithm *REVISE-3* which takes three variables $X$, $Y$, and $Z$, and removes any pair of values $<X = a, Y = b>$ when $a$ and $b$ are not connected to some value $c \in D_Z$ (Dechter, 2003). In other words, we are pruning the relation $R_{XY}$.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

**Algorithm 12.8.** Algorithm REVISE-3 prunes the relation $R_{XY}$, removing any edge $<a, b>$ that does not have a matching value in the domain of variable $Z$.

```
REVISE-3((X,Y), Z)
1.    for every <a, b> ∈ R_XY
2.       if there is no c ∈ D_Z s.t. <a, c> ∈ R_XZ and <b, c> ∈ R_YZ
3.          then delete <a, b> from R_XY
```

This is, in fact, an instance of the general case wherein making a network $N$-consistent induces a relation of arity $N - 1$, which essentially prunes an existing relation that could have been universal. This was the case also for arc consistency, because pruning the domain of a variable $X$ is equivalent to inducing a relation $R_X$ on the network. We will have more to say on this later.

The simplest algorithm to achieve path consistency is analogous to AC-1. It repeatedly considers *all* variable pairs $X$ and $Y$ and eliminates pairs of *values* $a \in D_X$ and $b \in D_Y$ that cannot be extended to a third variable $Z$. The algorithm is called *PC-1*.

**Algorithm 12.9.** Algorithm *PC-1* repeatedly calls REVISE-3 with *every pair* of variables for path consistency with *every* other variable, until no relation $R_{YZ}$ is pruned. The algorithm assumes that every pair of variables is related, even if by a universal relation which does not show up in the constraint graph.

```
PC-1(X, D, C)
1.    repeat
2.       for each x in X
3.          for each y and z in X
4.             REVISE-3((y, z), x)
5.    until no relation changes in the cycle
```

Let there be $n$ variables, each with domain of size $k$. The complexity of REVISE-3 is $\mathcal{O}(k^3)$ because the algorithm has to look at all values of the three variables. In each cycle the algorithm *PC-1* inspects $(n - 1)^2$ edges for each of the $n$ variables, requiring $\mathcal{O}(k^3)$ computations for each call to REVISE-3. Therefore, in each cycle, the algorithm will do $\mathcal{O}(n^3k^3)$ computations. In the worst case, PC-1 will remove one pair of values $<a, b>$ from some constraint $R_{xy}$. Then the number of cycles is $\mathcal{O}(n^2k^2)$, because there are $n^2$ pairs of variables, each having $k^2$ elements. Thus in the worst case, algorithm PC-1 will require $\mathcal{O}(n^5k^5)$ computations (Dechter, 2003).

Note that unlike AC-1, the algorithm PC-1 is not confined to working only with the edges in the constraint graph but considers all pairs of variables. It might be pertinent to remember that two variables in the constraint graph without an edge are related by the universal relation,

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

which means that any combination of values is allowed. Achieving path consistency may delete some elements from the universal relation, as illustrated in Figure 12.13.

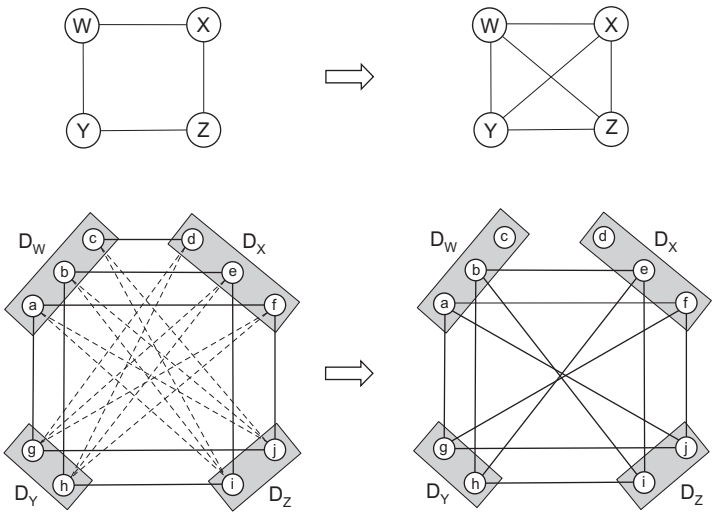



**Figure 12.13** The two figures on the left are the constraint graph and the matching diagram for a network with four variables *W, X, Y*, and *Z*. The dashed edges represent the implicit universal relations $R_{wz}$ and $R_{XY}$. On the right are the corresponding figures after the network is made path consistent. The relations $R_{wz}$ and $R_{XY}$ are now non-universal and show up in the constraint graph. The edge *c–d* is deleted along with eight edges from $R_{wz}$ and $R_{XY}$.

It can be observed that after achieving path consistency every edge in the matching diagram is part of a triangle with all other variables. The edge *c–d* in Figure 12.13 on the left gets deleted because it is not part of any triangle with values in variables *Y* and *Z*. For the same reason, four edges from each of the two implicit universal relations $R_{wz}$ and $R_{XY}$ are also deleted. In the network on the right, every edge is a part of two triangles.

Algorithm PC-1 looks at all triples of variables in every cycle. A better approach is to look only at variables where an edge deletion may have broken a triangle in the spirit of AC-3. Let the variables be an ordered set $X = \{x_1, x_2, \ldots, x_N\}$. Algorithm *PC-2* too tests every variable pair $<x_i, x_j>$ where $i < j$ against all other variables. It begins by enqueuing all such triples for calls to REVISE-3. Each pair of variables is added only once. If an edge $<a, b> \in R_{xy}$ is deleted by a call to REVISE-3, then PC-2 only checks for the triangles formed by all other variables with *x* and *y*. In the following algorithm, the indices 1, 2, …, *N* of the variables $x_1, x_2, \ldots, x_N$ are stored in the queue *Q* to enable only one of $<x_i, x_j>$ and $<x_j, x_i>$ to be added.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

**Algorithm 12.10.** Algorithm PC-2 begins by enqueuing all triples of distinct variables. It dequeues them one by one and calls REVISE-3((*x*,*y*), *z*). If any edge in *x*–*y* is deleted, then for every other variable *w*, PC-2 sets up calls to REVISE-3((*x*, *w*), *y*) and REVISE-3((*y*,*w*), *x*).

```
PC-2 (X, D, C)
 1.   Q ← []
 2.   for i ← 1 to N−1
 3.     for j ← i+1 to N
 4.       for each k s.t. k ≠ i and k ≠ j
 5.         Q ← Q ++ ((i, j), k)
 6.   while Q is not empty
 7.     ((i, j), k) ← head Q
 8.     Q ← tail Q
 9.       REVISE-3((x_i, x_j), x_k)
10.      if R_ij has changed
11.        for k ← 1 to N
12.          if k ≠ i and k ≠ j
13.            Q ← Q ++ ((i, k), j): [((j, k), i)]
```

Each call to REVISE-3 is $\mathcal{O}(k^3)$. The minimum number of calls is $\mathcal{O}(n^3)$, which is the number of distinct calls that can be made initially. In the worst case, one pair of values is deleted in each call to REVISE-3 in the while loop. In the worst case, $n^3$ calls to REVISE-3 are made. In each call to REVISE-3, at most $k^2$ edges can be removed. Hence the while loop can be executed at most $\mathcal{O}(n^3k^2)$ times and with REVISE-3 being $\mathcal{O}(k^3)$, the complexity of PC-2 is $\mathcal{O}(n^3k^5)$.

Like AC-1 and AC-3, both PC-1 and PC-2 rely on calls to REVISE-3, which can be a little bit of overkill. Like in AC-4, one can work at the value level, but we will not pursue those edges in the matching diagram. Mohr and Henderson (1986) have devised such an algorithm *PC-4* with complexity $\mathcal{O}(n^3k^3)$.

It is worth noting that path consistency does not automatically imply arc consistency. This is evident in Figure 12.13. The *CSP* is path consistent, but it is not arc consistent. In general, if a *CSP* is *i*-consistent it does not mean that it is $(i-1)$-consistent as well.

### 12.3.4 i-Consistency

The concept of consistency can be applied to any number of variables. Without going into the details we observe that the process involves defining a function *REVISE-i* in which a tuple $t_S$ from a set $S$ of $(i-1)$ variables is checked with one variable $X$ for consistency. If there is no value $v$ in $X$ that is consistent with the tuple then $t_S$ is deleted. This is equivalent to introducing a relation $R_S$ of arity $(i-1)$. In general, the complexity of REVISE-*i* *is* $\mathcal{O}(k^i)$, and algorithms for enforcing *i-consistency* have the worst case time complexity $\mathcal{O}((nk)^{2i}2^i)$ and space complexity of $O(n^ik^i)$ as described in (Dechter, 2003).

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

A network is said to be *strongly i-consistent* if it is *i-consistent* and is *j-consistent* for all $j < i$. If a network is strongly *i*-consistent, then the algorithm BACKTRACKING will find the solution in the first shot and will be backtrack free. However, the cost of achieving this can be prohibitive, and there are other approaches that one can employ to chip away at the complexity of depth first search. We first look at directional consistency.

### 12.3.5 Directional consistency

When a network is *i-consistent*, then a consistent assignment to *any* $(i - 1)$ variables can be extended to $i$ variables. In practice, though, we often have an order in which the algorithm BACKTRACKING searches. One only needs to ensure that the variables that *precede* a given variable $X$ have values that admit a consistent value for $X$. This brings in the notion of *directional consistency* in which consistency is *only* in the direction from parents to children, and not for *any* subset of variables, as illustrated in Figure 12.14.

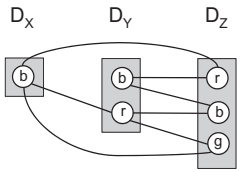



**Figure 12.14** The above network is neither arc consistent nor path consistent, but is both directionally arc consistent and directionally path consistent. Given the order *X*, *Y*, *Z*, BACKTRACKING finds a solution without backtracking.

Given an ordering $X = (x_1, x_2, \ldots, x_N)$, a network $<X, D, C>$ is said to be *directionally arc consistent* (DAC) if for every edge $<x_i, x_j>$ in the constraint graph such that $i < j$, variable $x_i$ is arc consistent with respect to variable $x_j$. DAC arc consistency can be achieved in a single pass, processing variables from the last to the first.

**Algorithm 12.11.** Algorithm DAC scans the variables from the last to first calling REVISE with all parents in the constraint graph.

```
DAC(X = [x1, x2, …, xn], D, C)
1.    for i ← N downto 2
2.       for j ← i−1 downto 1
3.          if Rij ∈ C
4.             REVISE((xj), xi)
```

*Directional path consistency* (DPC) is similar, except that it prunes binary relations and looks at all triples without any heed to the constraint graph. When it prunes an edge in the matching diagram, it adds a relation to the constraint graph. In the following algorithm, we have included DAC as well.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

**Algorithm 12.12.** Algorithm DPC does one pass from the last variable down to the first one. For each variable, it calls REVISE-3 with all the preceding variables, and then it also calls REVISE.

```
DPC(X = [x_1, x_2, ..., x_N], D, C)
1.    for i ← N downto 3
2.       for j ← i−1 downto 2
3.          for k ← j−1 downto 1
4.             REVISE-3((x_k, x_j), x_i)
5.             add R_kj to C
6.       for j ← i−1 downto 1
7.          if R_ij ∈ C
8.             Revise((x_j), x_i)
```

Figure 12.15 illustrates the DPC algorithm on a 4-variable 2-colour map colouring problem. To begin with, the constraint graph has three relations $R_{WY}$, $R_{XZ}$, and $R_{YZ}$. After the call REVISE-3((*X, Y*), *Z*) an induced relation $R_{XY}$ is added, and after REVISE-3((*W, X*), *Z*) the relation $R_{WX}$ is added.

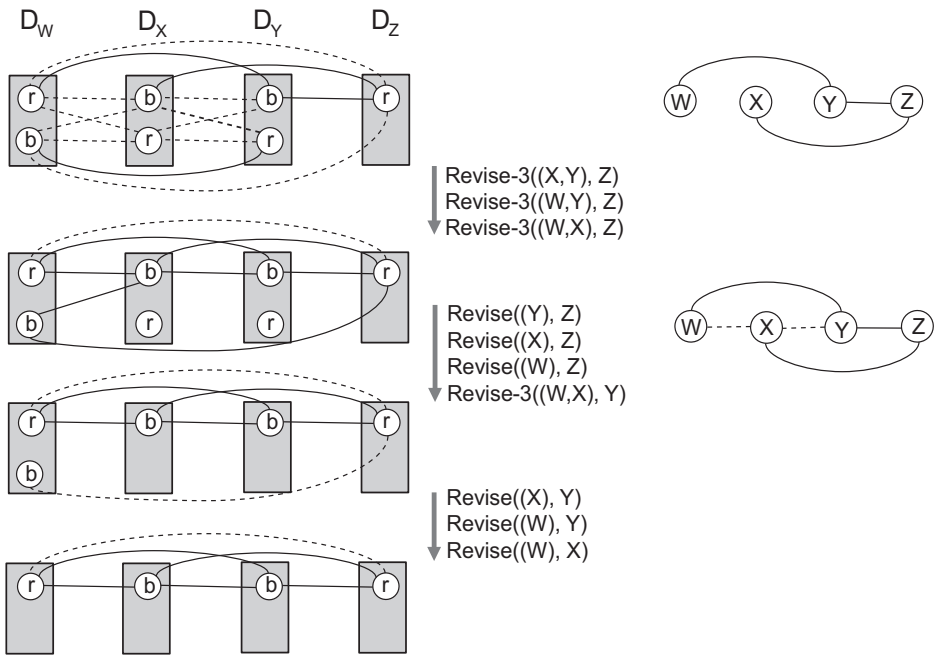



**Figure 12.15** DPC and DAC process the network in one pass from the last to the first node. The original matching diagram and the constraint graph are on the top. The dashed edges are from the universal relations. The revised versions are shown progressively below. The final network is strongly path consistent and backtrack free.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

The resultant network has an induced width 2. Observe that the edge <*r, r*> between variables *W* and *Z* is a remnant of the universal relation, and not a member of an induced relation. The induced width of the graph is 2, and for that DPC is sufficient for search to be backtrack free. If the induced width were to be higher, then a higher level of consistency would be required. This is neatly arrived at by the algorithm *AdaptiveConsistency*, which also processes the variables from the last to the first, but for each variable the degree of consistency is tailor-made based on the number of parents the node has. One must keep in mind that as the algorithm achieves the requisite consistency for a variable, it induces new relations on the parents, which may increase the width of some nodes. The algorithm is described below. In the literature a variation, called *bucket elimination*, that focuses on the relations explicitly is also popular.

**Algorithm 12.13.** Algorithm AdaptiveConsistency looks at the number *t* of parents of a variable *x,* and calls Revise-*t* to filter out value combinations from the parents *S*. It then adds edges between all parents and augments the constraint graph. It also induces a relation $R_s$ by deleting those sets of values in the parents that do not have a matching value in $D_x$.

AdaptiveConsistency($X = [x_1, x_2, \ldots, x_N]$, D, C)
1. E ← set of edges in the constraint graph
2. **for** i ← N **downto** 2
3. S ← Parents($x_i$)
4. t ← |S|
5. $R_s$ ← Revise-t(S, $x_i$)
6. **for** all $x_j$, $x_k$ ∈ S
7. E ← {<$x_j$, $x_k$>} ∪ E
8. C ← {$R_s$} ∪ C

The propagation techniques for combating combinatorial explosion seen so far are largely static and precede the search for solutions. Now we turn our attention to how some of these can be carried forward to the search phase itself. We have already mentioned dynamic variable ordering earlier. In the next section we look at ways for constraint propagation during search. Before picking a value for a variable, can we compute the impact on the domains of future variables?

Look before you leap.

## 12.4 Lookahead Search

Solving a CSP is often a mix of search and reasoning. Search tries out various assignments choosing variables and trying out values from their domains. Reasoning aims to compress the search space to minimize the work to be done by search. The consistency enforcement approaches described earlier process the CSP before Backtracking takes over. While looking for a value for a variable, Backtracking checks each value for consistency with the values

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

assigned to earlier variables. The algorithms in this section look ahead at future variables in addition to the ones in the past. Of course, as before, there is a cost to be paid for the extra reasoning one does.

Consider trying to solve an *N*-queens problem on a real chessboard or one drawn on a piece of paper. Every queen one places rules out all the squares it attacks for the other queens to be placed. Imagine marking those squares with a cross. In Figure 12.16 we illustrate how placing six queens row by row, this marking process can help narrow down search and backtrack even before a dead end is reached.

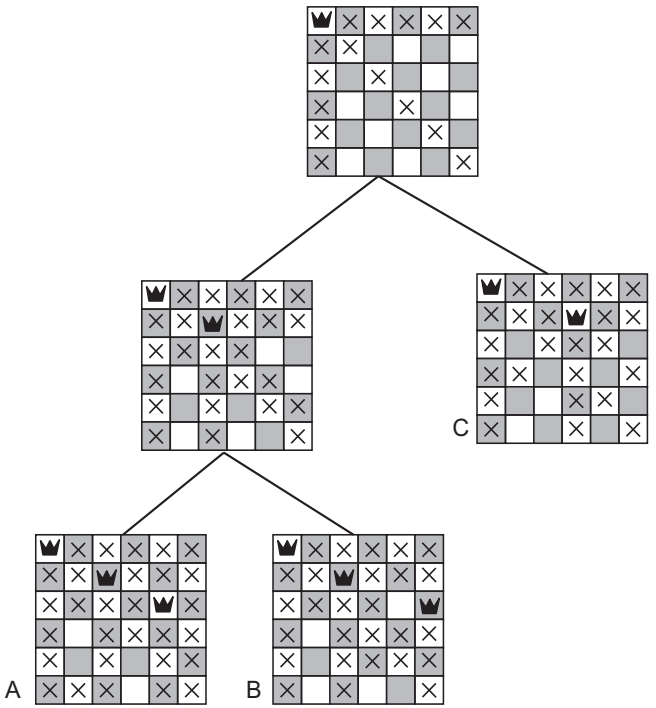



**Figure 12.16** After placing a queen in the corner of the top row, the crosses mark the squares no longer available. By the time search places the third queen in board position *A*, many squares in the bottom half are already marked. At this point, placing a queen in the fourth row would block the entire sixth row. The algorithm backtracks and tries position *B* with similar effect. It next goes back to trying a new value for the second queen in board position *C*.

Placing queens row by row in the first available position, one finds oneself in board position *A* after placing three queens. There is one unmarked square in row 4, but if one were to place a queen there, row 6 would be completely blocked. The next, and last, option, marked *B*, for the third queen would have a similar impact with row 5 being ruled out this time. Without even placing the fourth queen one, can backtrack and try another square for queen number 2 in board *C*. This is in essence the algorithm *forward checking* (FC).

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

### 12.4.1 Algorithm forward checking

Algorithm FC is a variation of BACKTRACKING in which the function called for selecting a value for the next variable does some lookahead. While considering a value $a_i$ for the variable $x_i$, the algorithm looks at all values in all future variables (Lines 4–7 of *SELECTVALUE-FC* in Algorithm 12.14) and deletes values that are not consistent with the proposed extension of the assignment $\mathcal{A}$. Only if no future domain is empty does it return the value $a_i$, else it undoes the deletions done with respect to this value (Lines 10–11). It is still in the while loop (Lines 1–11) and if there is another value available in the domain of $x_i$ it considers that next. If it emerges from the while loop without success, it returns *null* which triggers the parent algorithm to go back and look for another value for variable $x_{i-1}$ (Lines 7–10 in algorithm FC).

**Algorithm 12.14.** Given an ordering of the variables $x_1, \ldots, x_N$ algorithm FC is similar to algorithm BACKTRACKING except that the function SELECTVALUE-FC does more work pruning values from each future domain that is not consistent with the assignment $\mathcal{A}$ and the value $a_i$ being considered for variable $x_i$.

```
FORWARDCHECKING(X, D, C)
 1.  A ← []
 2.  for k ← 1 to N
 3.      D'_k ← D_k
 4.  i ← 1
 5.  while 1 ≤ i ≤ N
 6.      a_i ← SELECTVALUE-FC(D'_i, A, C)
 7.      if a_i = null
 8.          then    Undo lookahead pruning done while choosing a_{i-1}
 9.                  i ← i – 1          /* look for new value */
10.                 A ← tail A
11.          else    A ← a_i : A
12.                  i ← i + 1
13.  return REVERSE(A)

SELECTVALUE-FC(D'_i, A, C)
 1.  while D'_i is not empty
 2.          a_i ← head D'_i
 3.          D'_i ← tail D_i
 4.          for k ← i + 1 to N
 5.                  for each b in D'_k
 6.                      if not CONSISTENT(b : a_i : A)
 7.                          delete b from D'_k
 8.          if no D'_k is empty
 9.                  then return a_i
10.                  else for k ← i + 1 to N
11.                      undo deletes in D'_k
12.  return null
```

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

Algorithm FC does one pass over the future variables deleting values that are not going to be consistent with the current assignment. We illustrate the algorithm by following its progress on the matching diagram of a tiny example with five variables $x_1, \ldots, x_5$ processed in the given order. Figure 12.17 shows the constraint graph and the matching diagram at the start.

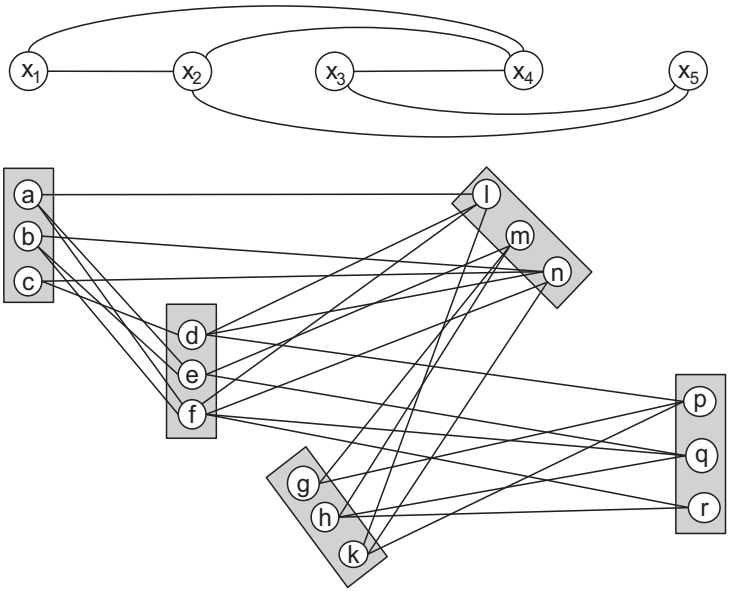



**Figure 12.17** A tiny CSP with five variables processed in the order $x_1$, $x_2$, $x_3$, $x_4$, and $x_5$. The constraint graph is shown at the top and the matching diagram at the bottom. Each domain has three values, selected in alphabetical order.

FC begins by calling for a value for variable $x_1$. SELECTVALUE-FC picks the first value $a$ from $x_1$. This value is connected to values $e$ and $f$ in $x_2$ but not connected to value $d$. Consequently, SELECTVALUE-FC removes $d$ from $D_{x2}$. Values deleted by SELECTVALUE-FC are shown with shaded circles in the figures that follow. In a similar manner, it also deletes values $m$ and $n$ from the domain of $x_4$. These are the only two variables which are related to $x_1$. SELECTVALUE-FC does no other pruning while considering value $a$. Then FC moves to $x_2$ and SELECTVALUE-FC tries the next *available* value $e$. This in turn deletes $l$ from $x_4$ and $p$ and $r$ from $x_5$. The situation at this point is shown in Figure 12.18.

At this point, the domain of variable $x_4$ has become empty and the algorithm undoes the deletions done with respect to $x_2 = e$ and backtracks to try another value.

When SELECTVALUE-FC tries the next value $x_2 = f$, it deletes $p$ from variable $x_5$ but that still has $q$ and $r$. It returns $x_2 = f$ to FC, which calls it again looking for a value for $x_3$. SELECTVALUE-FC tries $x_3 = g$ and $x_3 = h$ but both delete $l$ from $D_{x4}$. The next value $k$ does not, but it deletes $q$ and $r$ from the domain of $x_5$, which now becomes empty. The situation is shown in Figure 12.19. It has assigned values to the first three variables but does not even try to for the fourth.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

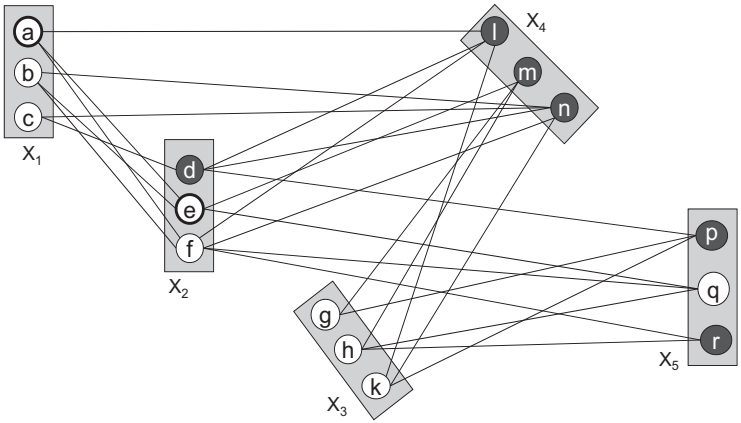



**Figure 12.18** When FC tries $x_1 = a$ and the first available value $x_2 = e$, it discovers that the domain of variable $x_4$ has become empty, because all three values are not consistent with tries $x_1 = a$ and $x_2 = e$. It will now undo deletion of values *l*, *p*, and *r* done while assigning $x_2 = e$ and will try the next value $x_2 = f$.

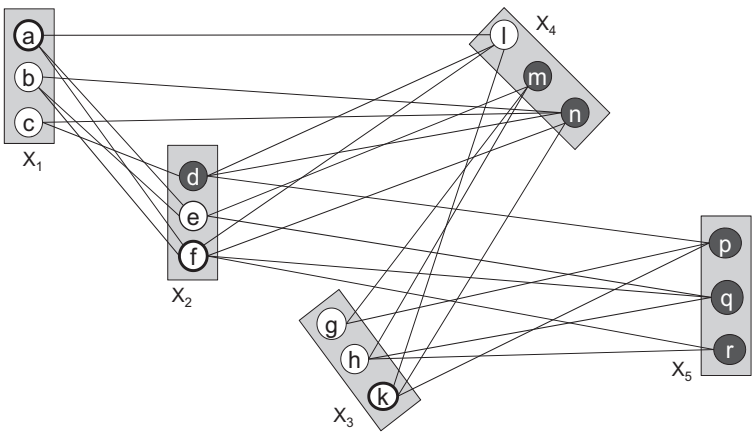



**Figure 12.19** When FC tries the next value $x_2 = f$ after undeleting values *l*, *p*, and *r*. This deletes *p* from $x_5$. It next tries values *g* and h for $x_3$ but both delete *l* in $x_4$. SELECTVALUE-FC next tries $x_3 = k$ but that deletes *q* and *r* from $x_5$, which becomes empty. There are no more values to backtrack to in $x_2$ and $x_3$ and it backtracks to $x_1$ and tries the value *b*.

SELECTVALUE-FC reports failure to find a value for $x_3$ and backtracks to $x_2$ but there is no other value available. It will next try $x_1 = b$. The reader is encouraged to verify that FC will backtrack because the $D_{x5}$ will again become empty after assigning the last possible value to $x_3$. The algorithm next tries $x_1 = c$ and eventually finds a solution with matching diagram as shown in Figure 12.20.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

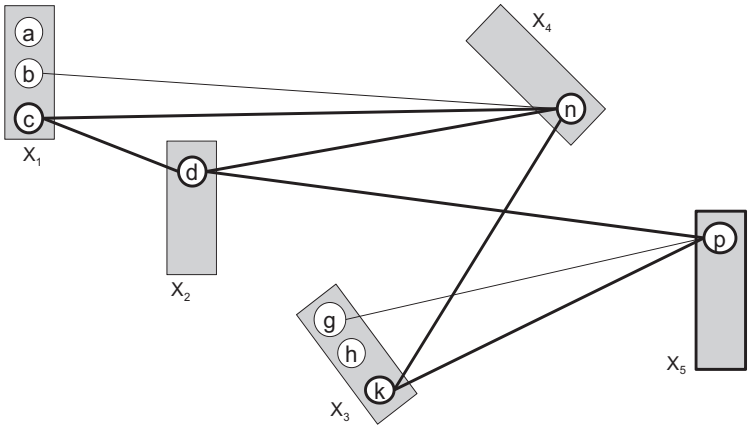



**Figure 12.20** The matching diagram at the point when FC finds the solution <*c, d, k, n, p*>. Note that values *a, b, g*, and *h* were not deleted because variables $x_1$ and $x_3$ do not have any parents in the given ordering.

In the diagram in Figure 12.20 there are still some unconnected values in the domains of $x_1$ and $x_3$ that have not been deleted. This is because there were no parents who could have done so. The next algorithm does a little bit more pruning of future domains.

### 12.4.2 Algorithm DAC-Lookahead

Algorithm *DAC-Lookahead*, also known as *PartialLookahead*, follows up with doing directional arc consistency on the future variables after SelectValue-DAC has deleted future values. In Algorithm 12.15 Lines 4–7 of SelectValue-DAC are the same as in SelectValue-FC, pruning future domains. Line 8 is the new addition where a call is made to DAC (Algorithm 12.11) for the future variable $\{x_{i+1}, \ldots, x_N\}$. Recall that DAC does one pass from the last variable to the first calling Revise with each connected parent.

**Algorithm 12.15.** Given an ordering of the variables $x_1,\ldots, x_N$ algorithm DAC-Lookahead does even more pruning than FC. After finding a value for variable *X*; consistent with the assignment $\mathcal{A}$, and deleting future nodes like in Algorithm 12.5, SelectValue-DAC does one pass of DAC on the future variables. If the current value *a*; does not work, it undoes the deletion done before trying the next value.

DAC-Lookhead(X, D, C)
1. $\mathcal{A} \leftarrow []$
2. **for** $k \leftarrow 1$ to N
3. $D'_k \leftarrow D_k$
4. $i \leftarrow 1$
5. **while** $1 \leq i \leq N$

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

```
6.      a_i ← SELECTVALUE-FC(D'_k, A, C)
7.      if a_i = null
8.        then      Undo lookahead pruning done while choosing a_{i-1}
9.                  i ← i – 1          /* look for new value */
10.                 A ← tail A
11.       else      A ← a_i : A
12.                 i ← i + 1
13.   return REVERSE(A)

SELECTVALUE-DAC(D'_i, A, C)
1.    while D'_i is not empty
2.        a_i ← head D'_i
3.        D'_i ← tail D'_i
4.        for k ← i + 1 to N
5.            for each b in D'_k
6.                if not CONSISTENT(b: a_i : A)
7.                    delete b from D'_k
8.        DAC ({x_{i+1}.. x_N}, D', C)
9.        if no domain is empty
10.               return a_i
11.       else for k ← i + 1 to N
12.               undo deletes in D'_k
13.   return null
```

The extra work done in DAC-LOOKAHEAD are these calls to DAC. We illustrate the effect of these on the tiny problem in Figure 12.17. DAC-LOOKAHEAD too begins by selecting $x_1 = a$ and deleting $d$ from $x_2$ and $m$, $n$ from $x_4$. As in the diagrams previously discussed, we show these as shaded circles in Figure 12.21, but we have deleted the edges emanating from them for clarity. Now DAC is called with the future variables $x_2$, $x_3$, $x_4$, and $x_5$ shown inside the dashed oval. Both $x_2$ and $x_3$ are arc consistent with respect to $x_5$ and no deletions happen. But values $e$ in $x_2$ and $g$, $h$ in $x_3$ do not have supporting values in $x_4$ and are deleted. The deletions by DAC are shown with cross marks, and the situation is as shown in Figure 12.21.

In the situation in Figure 12.21, DAC-LOOKAHEAD next tries the value $x_2 = f$. The future variables are now only $x_3$, $x_4$, and $x_5$ as shown inside the dashed oval in Figure 12.22. Forward checking deletes the value $p$ from the domain of $x_5$. This has a cascading effect when DAC kicks in with the value $k$ being deleted from $x_3$. The domain of $x_3$ is now empty and DAC-LOOKAHEAD retreats to $x_1$ and will try the value $b$.

Algorithm FC had looked at all values in the domain of $x_3$ before backtracking to $x_1$ to try the next value. Algorithm DAC-LOOKAHEAD retreated because it could not find a consistent value for $x_2$ without going to $x_3$. The next algorithm AC-LOOKAHEAD finds that it is unable to even assign $x_1 = a$.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

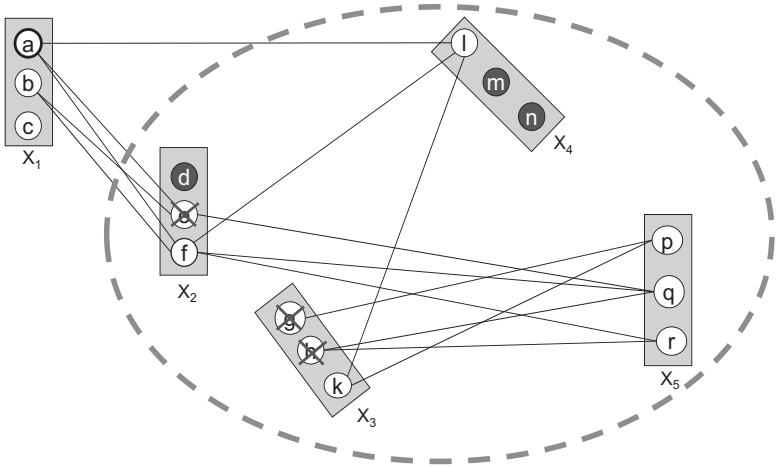



**Figure 12.21** Algorithm DAC-LOOKAHEAD begins like FC with $x_1 = a$ deleting values $d$, $m$, and $n$. The DAC component in *SELECTVALUE-DAC* kicks in for the remaining four variables $x_2$, $x_3$, $x_4$, and $x_5$ shown in the dashed oval resulting in $e$ being deleted from $x_2$ and $g$, $h$ being deleted from $x_3$. *DAC-LOOKAHEAD* will try $x_2 = f$ next.

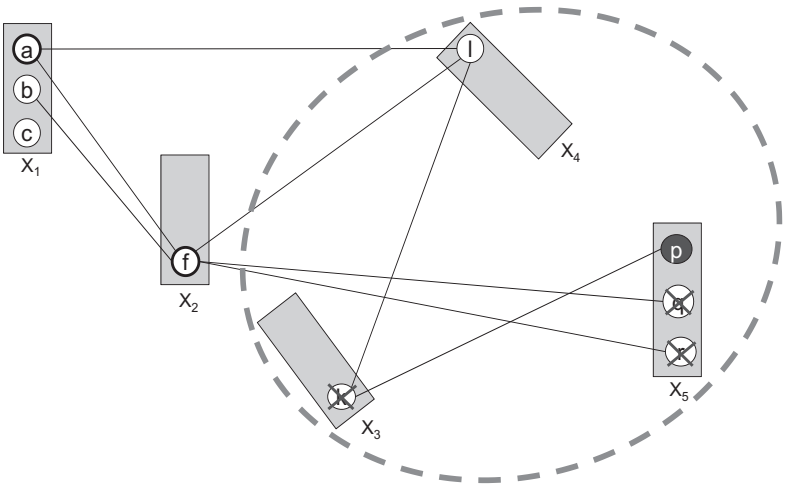



**Figure 12.22** When algorithm DAC-LOOKAHEAD picks value $f$ in $x_2$, forward checking deletes the value $p$ in $x_5$. The DAC component in SELECTVALUE-DAC kicks in for the remaining three variables $x_4$, $x_5$, and $x_5$, shown in the dashed oval resulting in $D_{x3}$ becoming empty. DAC-LOOKAHEAD retreats to variable $x_1$ without looking at $x_3$.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

### 12.4.3 Algorithm AC-Lookahead

The algorithm *AC-Lookahead* is similar to DAC-Lookahead except that in Line 8 of SelectValue-AC the algorithm calls for doing a full arc consistency of the future variables. This is clearly more work and also results in more pruning of the search space. We look at how the algorithm performs on the tiny problem from Figure 12.17.

Figure 12.23 shows the first part of the pruning phase when AC-Lookahead tries the first value *a* from the domain of $x_1$. Value *e* in $x_2$ has no support from $x_4$ and is deleted, as are values *g and h* in $x_3$. Likewise, the value *p* is deleted from the domain of $x_5$ because it does not have support from $x_2$.

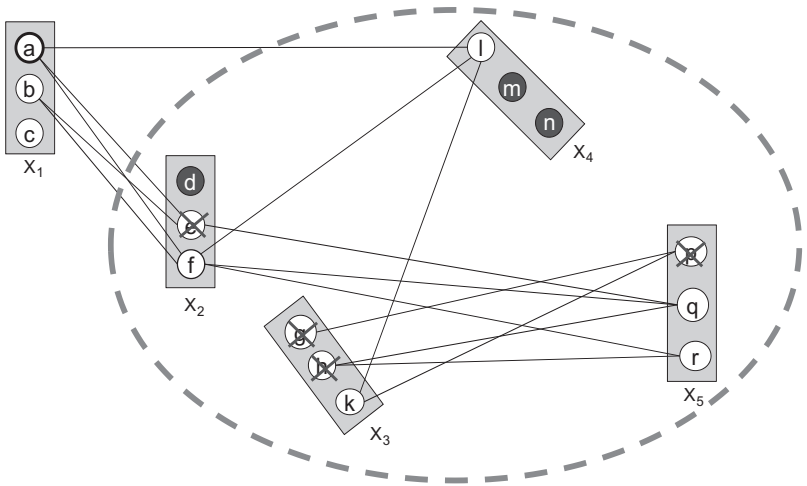



**Figure 12.23** Algorithm AC-Lookahead begins like FC with $x_1 = a$ deleting values *d, m*, and *n.* After this variables $x_2$, $x_3$, $x_4$, and $x_5$ are made arc consistent. Values *e* in $x_2$ without support in $x_4$, *g* and *h* in $x_3$ also without support in $x_4$, and *p* in $x_5$ without support in *x2* are the first to go, shown by cross marks.

The matching diagram at this stage is shown in Figure 12.24 where the pruning process continues after we have removed the pruned nodes from the figure. At this point, there are only five values remaining in the four future variables. Value *k* in variable $x_3$ is deleted because it has no support in $x_5$, and this results in *l* in $x_4$ and *p, q* in $x_5$ also being deleted, after which *f* goes from $x_2$.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

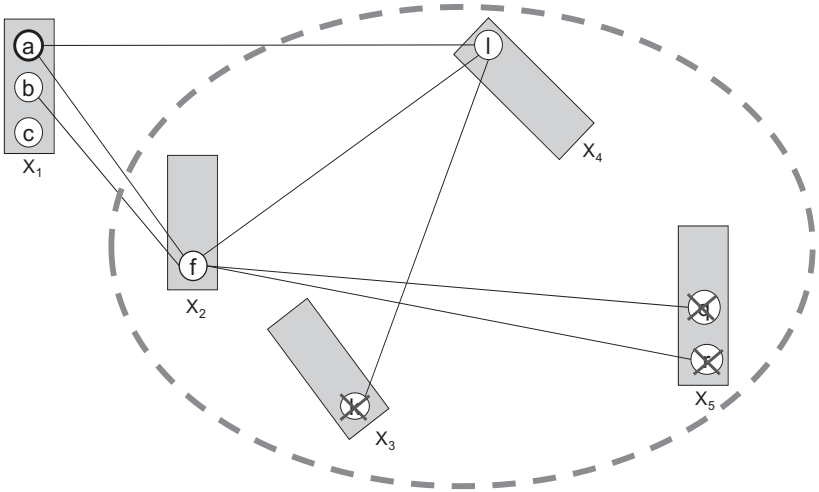



**Figure 12.24** Continuing from Figure 12.23 values $k$ in $x_3$ without support in $x_5$, $q$ and $r$ in $x_5$ without support in $x_3$ will go next. At this point, the domains of $x_3$ and $x_5$ have become empty, and $x_2$ and $x_4$ follow suit. Algorithm AC-Lookahead abandons the value $a$ for $x_1$ and moves on to $b$.

At this point, algorithm AC-Lookahead abandons the value $a$ it was considering for $x_1$ and moves on to the next value $b$. In practice, while implementing the algorithm one might exit as soon as one domain becomes empty. This is not reflected in our algorithm, where one blanket call is made to the algorithm for arc consistency.

The reader might have felt that AC-Lookahead perhaps does too much work. An algorithm we have not mentioned here is *FullLookahead*, which does a little bit less. This is like AC-Lookahead except that it does only one pass of calling Revise for every pair of future variables.

We now turn our attention to informed or intelligent backtracking.

## 12.5 Informed Backtracking

Given an ordering of variables, a search algorithm builds an assignment incrementally, looking for a consistent value for each variable. We take up the action when the algorithm is looking for a value for $x_i$ and has a partial assignment $\mathcal{A} = <a_1, a_2, \ldots, a_{i-1}>$. This assignment is consistent, which means that it satisfies all the constraints whose scope lies in the set $\{x_1, x_2, \ldots, x_{i-1}\}$. Now the algorithm seeks a value for $x_i$ that is consistent with $\mathcal{A}$. If it cannot find one, then it has reached a dead end. The search must retreat and try options other than $\mathcal{A} = <a_1, a_2, \ldots, a_{i-1}>$.

Algorithm Backtracking takes one step back and tries another value for $x_{i-1}$. The algorithm systematically tries different values for $x_{i-1}$ and will not miss a solution if one of them were to lead to one. This is called *chronological* backtracking, because the *last* variable that was assigned a value is looked at again. But the real reason behind the dead end may lie elsewhere,

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

and the work done trying different values for $x_{i-1}$ may be futile. Informed backtracking aims to reduce such unnecessary search and *jump back* to a variable where a different value may allow some value in $x_i$.

We say that the assignment $\mathcal{A} = <a_1, a_2, \ldots, a_{i-1}>$ is a *conflict set* with respect to $x_i$ if we cannot find a value $b$ in $D_i$ such that $<a_1, a_2, \ldots, a_{i-1}, b>$ is consistent. If no subset of $\mathcal{A} = <a_1, a_2, \ldots, a_{i-1}>$ is a conflict set with respect to $x_i$ we say that $\mathcal{A}$ is a minimal conflict set. We say that $<a_1, a_2, \ldots, a_{i-1}>$ is a *dead end* with respect to $x_i$, and $x_i$ is a *leaf dead end* variable. If in addition $<a_1, a_2, \ldots, a_{i-1}>$ cannot appear in *any* solution, we say that it is a *no-good*. It is possible for an assignment to be a no-good but not be a dead-end for any single variable. A minimal no-good is one which does not have any subset that is a no-good.

When $<a_1, a_2, \ldots, a_{i-1}>$ is a *conflict set* with respect to $x_i$, then search can jump back to *any* variable $x_j$ such that $j < i - 1$ in the quest for a solution. This process is called *backjumping*. We say that the jump back is *safe* if there is no $k$ between $j$ and $i - 1$ such that a new value for $x_k$ leads to a solution. Jumping back to a safe variable will thus not preclude any solution and affect the completeness of the algorithm. We say that a safe backjump to a variable $x_j$ is *maximal* if there is no $m < j$ such that a backjump to $x_m$ is safe.

The question is: given an assignment $<a_1, a_2, \ldots, a_{i-1}>$ that is a dead end for a variable, what is a safe and maximal backjump? We look at three well known algorithms for backjumping. Each collects differing kinds of data, based on which it decides the variable that is safe to jump back to. Each of them, however, arrives at a different answer to what is a maximal backjump that is safe.

### 12.5.1 Gaschnig's backjumping

*Gaschnig's backjumping* (GBJ) algorithm looks carefully at the current assignment $\mathcal{A} = <a_1, a_2, \ldots, a_{i-1}>$ while searching for a value for $x_i$. Before GBJ calls *SelectValue-GBJ* for a value for $x_i$, it sets a variable $latest_i$ to 0. SelectValue-GBJ starts with the first value $b \in D_i$ and scans $\mathcal{A}$ incrementally starting from $a_1$ to identify the index $k$ in $\mathcal{A}$ where the sub-tuple first becomes inconsistent with $b$. If this $k$ is greater than $latest_i$, its sets $latest_i$ to $k$. Then it moves on to the next value in $D_i$, where it could possibly increase the value of $latest_i$ further. If a value in $D_i$ were to be consistent with $\mathcal{A}$, the $latest_i$ would end up with the value $i - 1$.

If the call to SelectValue-GBJ were to return no value, then $latest_i$ would identify the *culprit* variable that GBJ needs to jump back to. We illustrate this process with the 8-queens example shown in Figure 12.25 in which five queens have been placed in rows 1–5, and GBJ is unable to place a queen in row 6.

When SelectValue-GBJ tried placing the sixth queen on square *a6*, it was being attacked by the queen in row 1. The variable $latest_6$ is set to 1. Then it tries square *b6* which is attacked by queens 3 and 5. If the culprit were one of these, then undoing the placement of queen 5 would not help, because queen 3 would still be attacking the square *b6*. So $latest_6$ is updated to 3. For square *c6* the earliest queen attacking is queen 2. But it would *not* be safe to jump back to queen 2 because a solution *might* have been possible by trying a new square for queen 3. Therefore, the latest that it is safe to jump back to is still 3, as reflected in the value of $latest_6$. It cannot jump back farther than 3, so it would be a *maximal* jump that is safe. Then looking at square *d6* this value is further increased to 4, where it stays over the next four squares in row 6.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

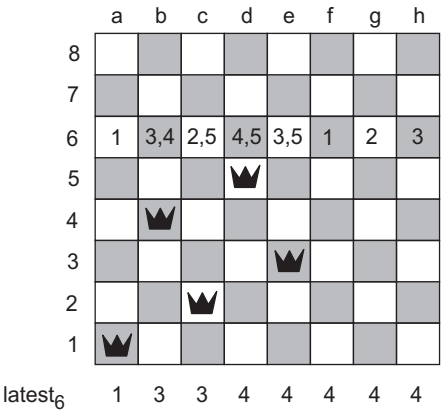



**Figure 12.25** SELECTVALUE-GBJ is unable to find a value (column name) for the sixth queen. The numbers in row 6 are the numbers of the queens attacking that square. In each of these, the earliest counts for each square. The value of $latest_6$ begins with 1 for square a6, becomes 3, the earlier queen attacking b6, and so on. The largest value is 4 from the square d6. GBJ would backtrack to the fourth queen.

The assignment <*a, c*, e, *b*> for the first four queens is a no-good. One of the queens must be relocated. It can only be queen 4 because a solution by relocating that could still be possible. Skipping queen 4 and relocating an earlier queen might miss a solution that relocating 4 might yield. So jumping back to queen 4 is both maximal and safe, and queen 4 is the culprit.

The algorithm GBJ is described below. Observe that the variable $latest_i$ is a global variable, initialized in the main program, set in the call to SELECTVALUE-GBJ, and used in the main program for jumping back when a *null* value is returned.

**Algorithm 12.16.** Algorithm GBJ is similar to algorithm BACKTRACKING except that it can jump back more than one step from leaf dead ends. The function SELECTVALUE-GBJ operates a global ratchet variable that identifies the maximal safe variable to jump back to when a value for $x_i$ cannot be found. For each value $a_i$ in $D_i$ it sets $latest_i$ to the index of the latest sub-tuple that is consistent with $a_i$ if that index is higher than the current value.

```
GBJ(X, D, C)
1.   A ← []
2.   i ← 1
3.   D'_i ← D_i
4.   while 1 ≤ i ≤ N
5.      latest_i ← 0
6.      a_i ← SELECTVALUE-GBJ(D'_i, A, C)
7.      if a_i = null
8.         then
9.            while i > latest_i
```

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

```
10.                 i ← i − 1
11.                 A ← tail A
12.           else
13.             A ← a_i : A
14.             i ← i + 1
15.             D'_i ← D_i
16.     return REVERSE(A)

SELECTVALUE-GBJ(D'_i, A, C)
 1.     while D'_i is not empty
 2.           a_i ← head D'_i
 3.           D'_i ← tail D'_i
 4.           consistent ← true
 5.           k ← 1
 6.           while k < i and consistent
 7.                 A_k ← take k A
 8.                 if k > latest_i
 9.                       latest_i ← k
10.                 if not CONSISTENT(a_i : A_k)
11.                       consistent ← false
12.                 else
13.                       k ← k + 1
14.           if consistent
15.                 return a_i
16.     return null
```

When SELECTVALUE-GBJ does return a value for $x_i$, the variable $latest_i$ has a value $i - 1$ and GBJ moves on to $x_{i+1}$. If at a later point GBJ were to backtrack to $x_i$, and if that had no value left in its domain, where would it backtrack to? This is known as an *internal dead end*. The value of $latest_i$ is $i - 1$, and hence GBJ would just move one step back. The algorithm GBJ thus does a safe and maximal backjump from a leaf dead end, but just moves one step back from an internal dead end.

### 12.5.2 Graph based backjumping

While GBJ pays no heed to the constraint graph when deciding where to jump back, *Graph based backjumping* (GBBJ) relies only on the relations between variables. So much so that when a dead end is reached, it concludes that a parent *must* be the culprit.

The algorithm defines and utilizes the following information relating to the constraint graph with a given ordering of variables. The set of ancestors $anc(x)$ of a node $x$ are all the nodes *connected* to $x$ which precede it in the ordering. Of these nodes, the parent of $x$, $parent(x)$, is the most recent ancestor. When the algorithm GBBJ encounters a leaf dead end, it jumps back to its parent, *assuming* that the parent is the source of the conflict.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

Figure 12.26 shows the graph from Figure 12.6 with the alphabetic ordering. Of the four nodes connected to node *E*, three are its ancestors in the given ordering. Of *A, B*, and *D*, the last one is the parent, and if node *E* were to be a leaf dead end, then it would try a new value of its parent *D*. This happens to be the last node visited, but that is not the case for nodes *C* and *F* who have only one ancestor and who is the parent. If *C* were to be a leaf dead end, GBBJ would try *A* next, and if *F* were to be a leaf dead end, GBBJ would try *D* next.

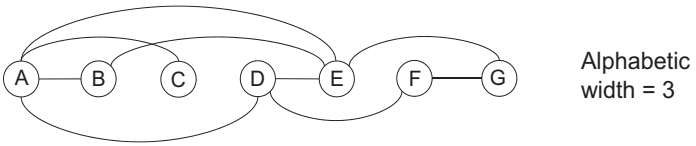



**Figure 12.26** On the alphabetic ordering in the graph from Figure 12.6 node *E* has three ancestors *A, B*, and *D* of which *D* is the parent. Nodes *C* and *F* have only one ancestor each which is the parent.

When GBBJ jumps back from a leaf dead end, it may encounter an *internal dead end*, which is the node it has jumped back to but which does not have any consistent value left. Where does it go next? Consider the case when node *G* were to be a leaf dead end and GBBJ jumped back to its parent *F*. If there is no value left in *F*, should it jump to its parent *D*? That would not be safe, because the original conflict in *G* might have been caused by *E*, its other ancestor. Bypassing *E* would not be safe. GBBJ handles this and similar cases as follows.

We say that GBBJ *invisits* a node *x* when it visits it in the forward direction, that is, after a node preceding it in the ordering. Starting from there, the *current session* of *x* includes all the nodes it has tried after invisiting *x* till the time it finds *x* to be an internal dead end.

We define the set of *relevant* dead ends *r(x)* of a node *x* as follows:

- If $x$ is a leaf dead end, then $r(x) = \{x\}$
- If $x$ is an internal dead end after jumping back from node $y$, then $r(x) = r(x) \cup r(y)$

The set of *induced ancestors* of a node *x* is defined as follows. Let *Y* be the set of relevant dead ends in the current session of *x*. Then the set of induced ancestors $I_x(Y)$ of node *x* is the union of all ancestors of nodes in *Y* which precede *x* in the ordering. The induced parent $P_x(Y)$ of node *x* is the latest amongst the induced ancestors of *x*. When *x* is a dead end, algorithm GBBJ jumps to the induced parent of a node *x*.

In Figure 12.26, when *G* turns out to be a leaf dead end, GBBJ tries its induced parent *F* (which is also its parent). The induced parent of *F* is *E* because *E* is the latest induced ancestor of *F*, an ancestor of *G*, which is a relevant dead end for *F*. So if *F* is an internal dead end, GBBJ will try *E* next. If *E* is an internal dead end too, the algorithm will try *D*. The current session of *D* includes *E, F*, and *G*, and hence the induced parent of *D* is *B*, which is where GBBJ will jump back to if *D* were to be a dead end too.

Contrast this with the case when *F* is a leaf dead end. GBBJ will jump back to *D* the parent of *F*. The relevant dead ends of *D* are *F* and *D* itself. The only induced ancestor of *D* is *A*. Remember that *E* is not a relevant dead end. If *D* were to be an internal dead end, the algorithm will now jump back to *A*.

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

The algorithm GBBJ shown below begins by computing the set of ancestors $anc(x)$ for each variable, given the ordering $x_1, x_2, \ldots, x_n$. Whenever it moves forward to a node $x_i$, it initializes the set of induced ancestors $l_i$ of $x_i$ to $anc(x_i)$ in Lines 6, 7 and 22, 23 of Algorithm 12.17. The index $p$ of the induced parent of the current node is always the latest node in the set of induced ancestors $l_i$ (Lines 7, 17, and 23). When it encounters a dead end $x_i$, which is when there is no consistent value for the current variable, it jumps back to induced parent $x_j$. It updates the induced ancestors of $x_j$ and identifies the induced parent to which it would jump back from $x_j$ if needed (Lines 12–17).

**Algorithm 12.17.** Algorithm GBBJ begins by computing the ancestors of each node. It keeps track of the induced ancestors when it jumps back to a node. It always jumps back to the induced parent on reaching a dead-end. The SELECTVALUE function is the simple one used in BACKTRACKING.

```
GBBJ(X, D, C)
 1.   A ← []
 2.   for k ← 1 to N
 3.      compute anc(k) the set of ancestors of x_k
 4.   i ← 1
 5.   D'_i ← D_i
 6.   I_i ← anc(i)
 7.   p ← latest node in I_i
 8.   while 1 ≤ i ≤ N
 9.      a_i ← SELECT VALUE(D'_i, A, C)
10.     if a_i = null
11.       then
12.         iprev ← i
13.         while i > p
14.               i ← i −1
15.               A ← tail A
16.         I_i ← I_i ∪ {l_iprev − {x_i}}
17.         p ← latest node in I_i
18.       else
19.         A ← a_i : A
20.         i ← i + 1
21.         D'_i ← D_i
22.         I_i ← anc(i)
23.         p ← latest node in I_i
24.   return REVERSE(A)

SELECTVALUE(D'_i, A, C)
1.   while D'_i is not empty
2.         a_i ← head D'_i
```

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

```
3.          D'_i ← tail D'_i
4.          if CONSISTENT(a_i : A)
5.                then return a_i
6.    return null
```

The jumping back behaviour of GBBJ is the same whether it does so from an internal dead end or a leaf dead end. This is an improvement over GBJ, which can jump back only from leaf dead ends. But GBBJ is conservative and assumes that if a node is a parent in the constraint graph, it *must* be the cause of the dead end, and jumps to that as an insurance. It will also not miss out on any solution and its jumps are safe. But it may jump back less than it needs to because it does not look at the *values* in the domains that lead to the conflict. The next algorithm makes use of both kinds of information, based on the values that conflict, and also the graph topology.

### 12.5.3 Conflict directed backjumping

The reason why search has to backtrack is that it cannot find a value for the next variable $x_i$ that is consistent with the current assignment $\mathcal{A}$ being constructed. A value $a_i$ of $x_i$ being considered conflicts with $\mathcal{A}$ because it conflicts with some constraint $R_S$ whose scope $S$ is within the set of variables $x_1 \ldots x_{i-1}$ already instantiated. As soon as the select value function in algorithm *conflict directed backjumping* (CDBJ) spots a conflict, it identifies the earliest constraint that conflicts with $a_i$.

Given an ordering of variables $x_1, x_2, \ldots, x_n$, a constraint $R_i$ is said to be *earlier* than a constraint $R_k$ if the latest variable in $R_i$ which is not in $R_k$ is earlier than the latest variable in $R_k$ which is not in $R_i$. That is, the latest in $S_i - S_k$ is earlier that the latest in $S_k - S_i$ where $S_i$ and $S_k$ are the scopes respectively of $R_i$ and $R_k$. For example, given $S_i = \{x_6, x_{10}\}$ and $S_k = \{x_1, x_3, x_7, x_8, x_{10}\}$, the constraint $R_i$ is earlier than $R_k$ because $x_6$ is earlier than $x_8$.

Algorithm CDBJ works with a set $emc(x_i)$ or $emc(\mathcal{A})$ of conflicting values called the *earliest minimal conflict set* associated with the variable $x_i$ that the algorithm is seeking to pick a value from. The variables $var\text{-}emc(x_i)$ in this set define the jumpback set $J_i$ for variable $x_i$. The jumpback set serves the same purpose that the ancestor set $anc(x_i)$ did in algorithm GBBJ, which is that the algorithm jumps back to the latest variable in this set. The difference is that GBBJ constructs the set based on the graph topology while CDBJ, like GBJ, does so based on an actual conflict of values. Both CDBJ and GBJ would jump back to the same variable from a leaf dead end, but CDBJ can jump back even from internal dead ends because it maintains an induced jumpback set $J_i$ when it jumps back based on all relevant dead ends, like in GBBJ. At the same time, the induced jumpback set in CDBG can be a subset of the induced ancestor set of GBBJ because CDBG only adds variables when it detects a real conflict, whereas GBBJ conservatively assumes that if the variables are connected, one of them must be the culprit. Thus GBBJ may jump back to a variable that CDBJ knows is not the actual culprit.

When algorithm CDBJ visits a new variable (Lines 2–4 and 17–19 in Algorithm 12.18), it initializes the jumpback set to the empty set. Then it calls *SELECTVALUE-CDBJ*, which, like *SELECTVALUE-GBJ*, scans the values in the assignment $\mathcal{A}$ incrementally checking for consistency

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

(Lines 5–13 of *SelectValue-CDBJ*). The moment the value $a_i$ from the domain of $x_i$ conflicts with $\mathcal{A}_k$, it adds the variables in the earliest conflict to the jumpback set $J_i$. Note that more than one constraint may simultaneously conflict with $a_i$ for a particular value of $k$. That is why one needs to select the earliest one amongst them (Lines 11–13). Having done that, it moves on to the next value in the domain $D_i'$ to test for consistency. For every value in $D_i'$ it finds a conflict, it adds the earliest conflict to the jumpback set $J_i$. If *SelectValue-CDBJ* cannot find a consistent value, then $x_i$ would be a leaf dead end and the parent program would jump back to the latest variable in the jumpback set $J_i$. Observe that like in algorithm GBJ we have assumed that $J_i$ is global data structure.

**Algorithm 12.18.** Algorithm CDBJ looks at actual conflicts of values a little bit like GBJ, but constructs the earliest minimal conflict set in *SelectValue-CDBJ* when it spots a conflict based on the earliest constraint that conflicts with a value in $x_i$. Like GBBJ it can combine the data gleaned from relevant dead ends in the main algorithm to be able to jump back from internal dead ends as well.

```
CDBJ(X, D, C)
 1.   A ←[]
 2.   i ← 1
 3.   D'i ← Di
 4.   Ji ← { }
 5.   while 1 ≤ i ≤ N
 6.      ai ← SelectValue(D'i, A, C)
 7.      if ai = null
 8.         then
 9.           iprev ← i
10.           p ← latest node in Ji
11.           while i > p
12.                i ← i − 1
13.                A ← tail A
14.           Ji ← Ji ∪ {Jiprev − {xi}}
15.        else
16.           A ← ai : A
17.           i ← i + 1
18.           D'i ← Di
19.           Ji ← { }
20.   return Reverse(A)

Select Value-CDBJ(D'i, A, C)
 1.   while D'i is not empty
 2.           ai ← head D'i
 3.           D'i ← tail D'i
 4.           consistent ← true
 5.           k ← 1
```

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

```
6.          while k < i and consistent
7.              A_k ← take kA
8.              if CONSISTENT(a_i : A_k)
9.                  k ← k + 1
10.             else
11.                 R_s ← earliest constraint with scope S causing the conflict
12.                 J_i ← J_i ∪ {S − {x_i}}
13.                 consistent ← false
14.         if consistent
15.             return a_i
16.     return null
```

Observe that when CDBJ jumps back to variable $x_i$ (Lines 10–14 of CDBJ), it is still in the current session of the variable, not yet having retreated from there. It might find a value for this variable and go forth to the next variable and onwards, till it strikes another dead end and again jumps back to $x_i$ from a relevant dead end. The merging of jumpack sets in Line 14 of CDBJ is similar to the process of computing the induced ancestors in GBBJ which enables the algorithm to jump back safely and maximally from the internal dead end as well.

## Summary

Constraints offer a uniform formalism for representing what an agent knows about the world. The world is represented as a set of variables each with its own domain of values, along with local constraints between subsets of variables. The fact that constraints are local obfuscates the world view. It is not clear what combination of values for each variable is globally consistent. The task of solving the CSP is to elucidate these values, which can be thought of as unearthing the solution relation that prescribes all consistent combinations of values.

There are two approaches to strive for this clarity, and constraint processing allows for the interleaving of both. On the one hand, there is constraint propagation or reasoning that eliminates infeasible combinations of values, in the process adding new constraints to the network. On the other is search, the basic strategy of problem solving by first principles.

We explored various combinations of techniques. This includes various levels of consistency that can be enforced. We looked at algorithms for arc consistency and path consistency. We also looked at the advantages of directional consistency and a little bit on ordering variables for search. Then we started with the basic search algorithm BACKTRACKING which essentially searches through combinations of values for variables. This can be augmented with look ahead methods that prune future variables, and by look back methods that make an informed choice of which variable to jump back to when a dead end is encountered. In all cases a dead end forces the search algorithm to retreat and undo some instantiations to try new combinations. One aspect we have not studied is no-good learning. Here, every time an algorithm jumps back to a culprit variable, the combination of conflicting values can be marked to be avoided in the

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

future. Clearly, no-good learning would be meaningful in large problems with huge search trees that can benefit with such pruning.

We have not looked at many applications despite having said that the CSPs present a very attractive formulation in which all kinds of problems can be posed, and then solved using some of the methods we have studied. We did mention in the chapter on planning that planning can be posed as a CSP, and illustrated the idea by posing it as satisfiability. Another frequent application is classroom scheduling and timetable generation which has its own group of interested researchers. Also, we have confined ourselves to finite domain CSPs with the general methods that they admit. We have not looked at specialized constraint solving problems and methods like linear and integer programming that have evolved as areas of study in themselves, beyond the scope of this book.

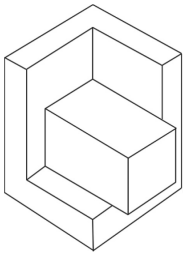
## Exercises

1. Which of the following statements are true regarding solving a CSP?
   - **a.** Values must be assigned to ALL variables such that ALL constraints are satisfied.
   - **b.** Values must be assigned to at least SOME variables such that ALL constraints are satisfied.
   - **c.** Values must be assigned to ALL variables such that at least SOME constraints are satisfied.
   - **d.** Values must be assigned to at least SOME variables such that at least SOME constraints are satisfied.
2. Pose the following cryptarithmetic problems as CSP:
   TWO + TWO = ONE
   JAB + JAB = FREE
   SEND + MORE = MONEY
3. Consider the following constraint network $R = <\{x_1, x_2, x_3\}, \{D_1, D_2, D_3\}, \{C\}>$ where $D_1 = D_2 = D_3 = \{a, b, c\}$ and $C = <\{x_1, x_2, x_3\}, \{<a, a, b>, <a, b, b>, <b, a, c>, <b, b, b>\}$. How many solutions exist?
4. Given a constraint satisfaction problem with two variables $x$ and $y$ whose domains are $D_x = \{1,2,3\}$, $D_y = \{1,2,3\}$, and constraint $x < y$, what happens to $D_x$ and $D_y$ after the REVISE$(x,y)$ algorithm is called?
   - **a.** Both $D_x$ and $D_y$ remain the same as before
   - **b.** $D_x = \{1,2\}$ and $D_y = \{1,2,3\}$
   - **c.** $D_x = \{2,3\}$ and $D_y = \{1,2\}$
   - **d.** $D_x = \{\}$ and $D_y = \{1,2,3\}$
5. Draw the search tree explored by algorithm BACKTRACKING for the 5-queens problem till it finds the first solution.
6. What is the best case complexity of REVISE$((X), Y)$ when the size of each domain is $k$?
7. Draw the matching diagram for the network in Figure 12.1 after it has been made arc consistent. Has the network become backtrack free?
8. What does one conclude when the domain of some variable $X$ while computing arc consistency becomes empty?

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

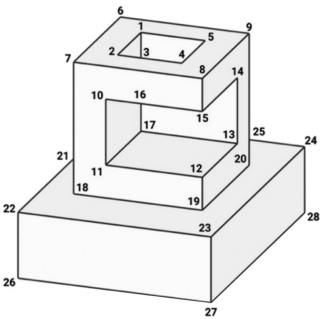
**9.** Try out different orderings for the two networks in Figure 2.8 and investigate how BACKTRACKING performs.

**10.** Is the following object a trihedral object? Label the edges, and explain your answer.

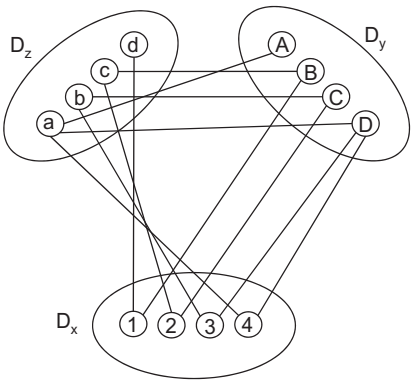


**11.** Draw trihedral objects to illustrate all the vertex labels shown in Figure 12.10.

**12.** [Baskaran] Label the edges in the following figure and identify each vertex type.

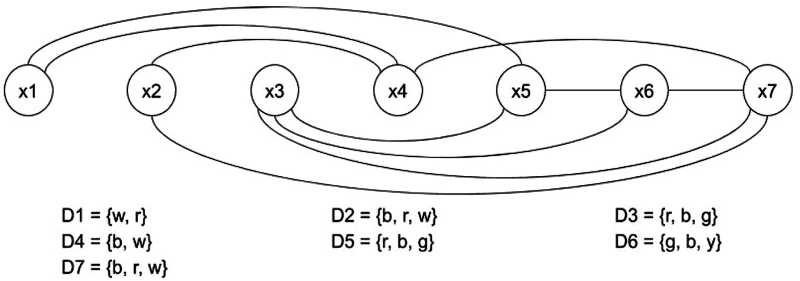



**13.** Given the CSP on variables $X = \{x, y, z\}$ and the relations $R_{xy}$, $R_{xz}$, $R_{yz}$ depicted in the matching diagram below, draw the matching diagram after the CSP has been made arc consistent. State the resulting CSP.




https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

**14.** Consider the following CSP for a map colouring problem. Answer the questions that follow.




**A.** The GeneralizedLookahead algorithm with SelectValue-FC is applied to the CSP in the figure, for the ordering $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$. If the algorithm chooses the assignments $x_1 = r$, $x_2 = b$, and $x_3 = b$, how many total values are pruned from the domains of the variables $x_4$, $x_5$, $x_6$, and $x_7$?

**B.** The GeneralizedLookahead algorithm with SelectValue-AC is applied to the CSP in the figure, for the ordering $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$. If the algorithm chooses the assignments $x_1 = r$, $x_2 = b$, and $x_3 = b$, how many total values are pruned from the domains of the variables $x_4$, $x_5$, $x_6$, and $x_7$?

**C.** The GeneralizedLookahead algorithm with SelectValue-PartialLookahead is applied to the CSP in the figure, for the ordering $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$. If the algorithm chooses the assignments $x_1 = r$, $x_2 = b$, and $x_3 = b$, how many total values are pruned from the domains of the variables $x_4$, $x_5$, $x_6$, and $x_7$?

**D.** The GeneralizedLookahead algorithm with SelectValue-FullLookahead is applied to the CSP in the figure, for the ordering $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$. If the algorithm chooses the assignments $x_1 = r$, $x_2 = b$, and $x_3 = b$, how many total values are pruned from the domains of the variables $x_4$, $x_5$, $x_6$, and $x_7$?

**15.** The objective of a $4 \times 4$ Sudoku puzzle is to fill a $4 \times 4$ grid so that each column, each row, and each of the four disjoint $2 \times 2$ subgrids contains all of the digits from 1 to 4.

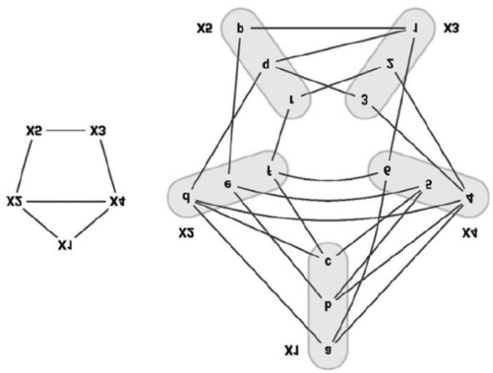
| | | | |
|---|---|---|---|
| x1 | x2 | x3 | x4 |
| x5 | x6 | x7 | x8 |
| x9 | x10 | x11 | x12 |
| x13 | x14 | x15 | x16 |

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press

The following figure depicts the domains of the variables for the given 4 × 4 Sudoku problem. Note that some cells have only one value in their domain. Show the order in which *dynamic variable ordering with forward checking* (DVFC) algorithm will fill in the values. Let the algorithm prefer variables in the order ($x_1$, $x_2$, …, $x_{16}$) at each tie break.

| | | | |
|---|---|---|---|
| 1 2 3 4 | 1 2 3 4 | 1 2 3 4 | 1 |
| 1 2 3 4 | 2 | 1 2 3 4 | 1 2 3 4 |
| 1 2 3 4 | 1 2 3 4 | 3 | 1 2 3 4 |
| 4 | 1 2 3 4 | 1 2 3 4 | 1 2 3 4 |

**16.** [Baskaran] The following figure shows a constraint graph of a binary CSP and a part of the matching diagram. When a pair of variables (like $X_1$ and $X_3$) do not have a constraint in the constraint graph then assume a *universal relation* in the matching diagram.

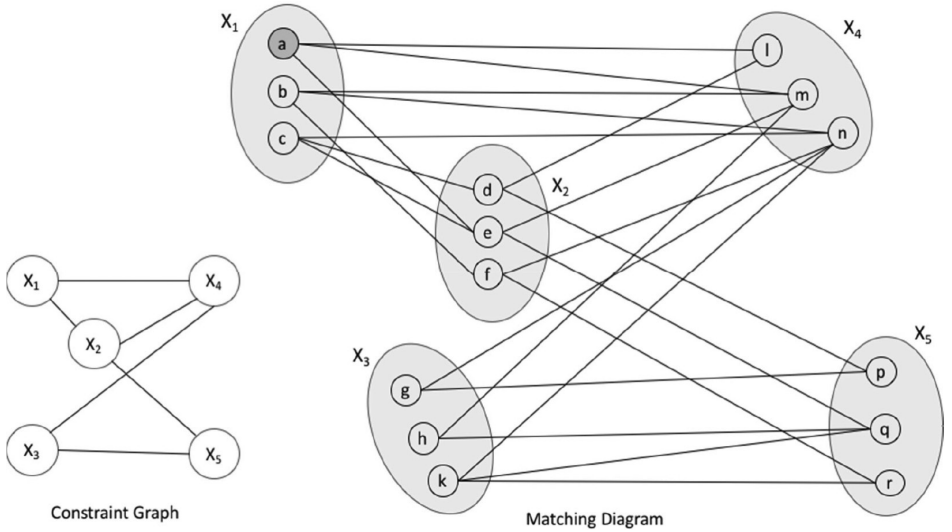



The FC algorithm begins by assigning $X_1 = a$. What are the *next* four values assigned to variables by the FC algorithm? List the values as a comma separated list in the order they are assigned.

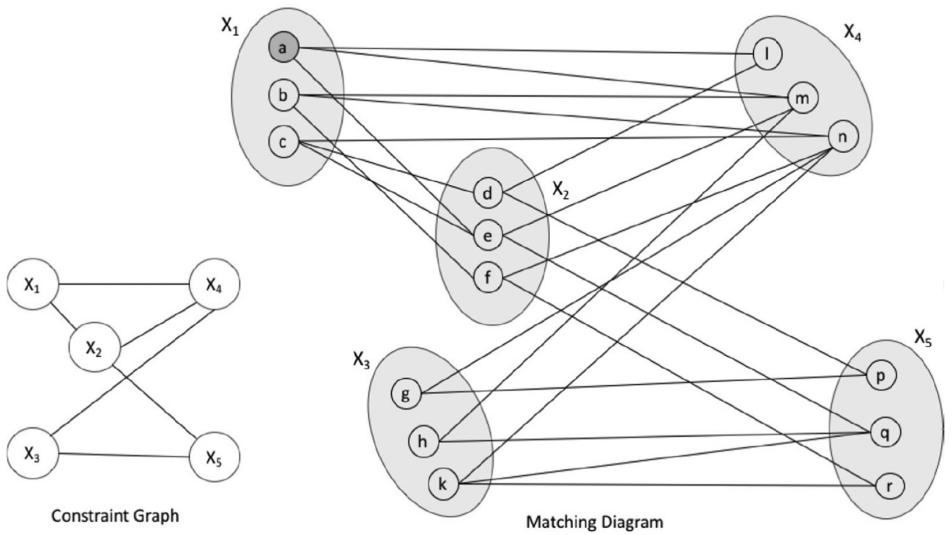
**17.** What is the first solution found by the FC algorithm for the above problem?

**18.** The following figure shows the constraint graph of a binary CSP on the left and a part of the matching diagram on the right. Please assume a *universal relation* in the matching diagram where there is no constraint between variables in the constraint graph. The variables, and their values, are to be considered in *alphabetical* order.
Algorithm FC is about to begin by assigning $X_1 = a$. What are the next six values assigned to variables? Draw the matching diagram at this point.
What is the first solution found by the algorithm?

https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press




**19.** Repeat the above question for the following CSP:




https://doi.org/10.1017/9781009284325.013 Published online by Cambridge University Press