chapter 4

Directional Consistency

Law is order, and good law is good order.
Aristotle, Politics

How do we explain how people perform so well on tasks that are theoretically intractable? One explanation is to assume that intelligent behavior is actually grounded in approximation methods that are based on idealized, easy-to-solve models. In other words, we assume people intuitively transform hard tasks into a series of more manageable, simple tasks, which, taken together, approximate the original task. Some real-life problems may naturally fall into such easy classes and can thus be solved efficiently. Likewise, some difficult problems may be transformed into simplified versions that are not too distant from the original problems. Following this line of reasoning, one approach in artificial intelligence is to imitate the assumed human reasoning process and find ways to idealize (i.e., simplify) the task environment. These simplifications can then be used to provide either approximate solutions or heuristic advice to guide the search toward an exact solution of the original problem. We will indeed see that although the general constraint satisfaction problem is hard, there are many subclasses of constraint networks that are easy to process (some of which we already saw in Chapter 3.)

In general, a problem class is considered *easy* when it allows a solution in polynomial time. Because of the popularity of backtracking search as the primary problem-solving method for constraint satisfaction problems, a CSP is considered easy if it is backtrack-free, that is, if backtracking search (the focus of Chapters 5 and 6) can solve the problem without encountering any dead-ends. In such a case, a solution is produced in time linear in the size of the problem, as defined by the number of variables and the overall size of the constraints.

As mentioned in Chapter 3, the primary means by which a constraint problem can avoid dead-ends is by making its representation more explicit via *inference*. If a bounded, polynomial-time inference method generates a backtrack-free representation, the whole problem can be solved efficiently. This concept has prompted a theoretical investigation into the level of local consistency that suffices for ensuring a backtrack-free search, leading to topological and constraint properties for which a restricted level of consistency enforcing is sufficient for transforming such networks into backtrack-free representations.

Let's first define the notion of backtrack-free search:

DEFINITION 4.1 **(backtrack-free search)**

A constraint network is backtrack-free relative to a given ordering $d = (x_1, \ldots, x_n)$ if for every $i \leq n$, every partial solution of $(x_1, \ldots, x_i)$ can be consistently extended to include x_{i+1}. •

The various successful approaches for identifying tractable classes of constraint satisfaction problems can be divided into two main groups. The first is *tractability by restricted structure*, which is based solely on the structure of the constraint graph of the problem, independently of the actual constraint relations. This class will be the focus of the current chapter, as well as Chapters 8 and 9. The second group, *tractability by restricted constraint relations*, or constraint languages that identify classes that are tractable thanks to special properties of the constraint relations, is the focus of Chapter 11.

Tractability due to restricted structure can be reasoned from the topological properties of the constraint graph and its hypergraph. We will start by reviewing relevant graph concepts.

4.1 Graph Concepts: Induced Width

Topological characterization is centered on the graphical parameter known as *induced width*. Given an undirected graph $G = (V, E)$, an *ordered graph* is a pair (G, d), where $V = \{v_1, \ldots, v_n\}$ is the set of nodes, E is a set of arcs over V, and $d = (v_1, \ldots, v_n)$ is an ordering of the nodes. The nodes adjacent to v that precede it in the ordering are called its *parents*. The *width of a node* in an ordered graph is its number of parents. The *width of an ordering d*, denoted $w(d)$, is the maximum width over all nodes. The *width of a graph* is the minimum width over all the orderings of the graph.

EXAMPLE 4.1 Figure 4.1 presents a constraint graph G over six nodes, along with three orderings of the graph: $d_1 = (F, E, D, C, B, A)$, its reversed ordering $d_2 = (A, B, C, D, E, F)$, and $d_3 = (F, D, C, B, A, E)$. Note that we depict the orderings from bottom to top, so that the first node is at the bottom of the figure and the last node is at the top. The arcs of the graph are depicted by the solid lines. The parents of A along d_1 are $\{B, C, E\}$. The width of A along d_1 is 3, the width of C along d_1 is 1, and the width of A along d_3 is 2. The width of these three orderings are $w(d_1) = 3$, $w(d_2) = 2$, and $w(d_3) = 2$. The width of graph G is 2. •

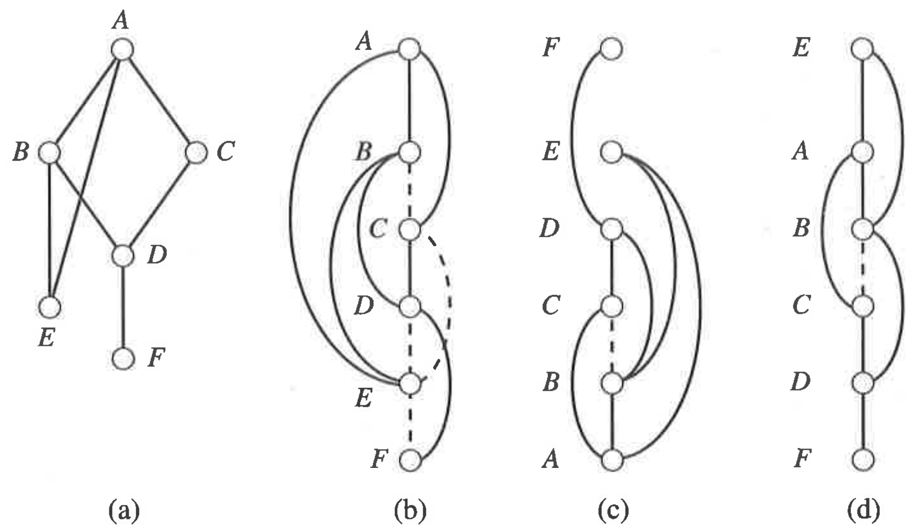


Figure 4.1 (a) Graph G, and three orderings of the graph: (b) $d_1 = (F, E, D, C, B, A)$, (c) $d_2 = (A, B, C, D, E, F)$, and (d) $d_3 = (F, D, C, B, A, E)$. Broken lines indicate edges added in the induced graph of each ordering.

The *induced graph* of an ordered graph (G, d) is an ordered graph (G^*, d) where G^* is obtained from G as follows: The nodes of G are processed from last to first (top to bottom) along d. When a node v is processed, all of its parents are connected. The *induced width of an ordered graph*, (G, d), denoted $w^*(d)$, is the width of the induced ordered graph (G^*, d). The *induced width of a graph*, w^*, is the minimal induced width over all its orderings.

EXAMPLE 4.2 Consider again Figure 4.1. For each ordering d, (G, d) is the graph depicted without the broken edges, while (G^*, d) is the corresponding induced graph that includes the broken edges. We see that the induced width of B along d_1 is 3, and that the overall induced width of this ordered graph is 3. The induced widths of the graph along orderings d_2 and d_3 both remain 2, and, therefore, the induced width of the graph G is 2. •

A rather important observation is that a graph is a tree (has no cycles) if and only if it has a width-1 ordering. The reason a width-1 graph cannot have a cycle is that, for any ordering, at least one node on the cycle would have two parents, thus contradicting the width-1 assumption. And vice versa: if a graph has no cycles, it can always be converted into a rooted directed tree by directing all edges away from a designated root node. In such a directed tree, every node has exactly one node pointing to it—its parent. Therefore, any ordering in which every parent node precedes its child nodes in the rooted tree has a width of 1. Furthermore, given

```
MIN-WIDTH (MW)
Input: A graph $G = (V, E)$, $V = \{v_1,\dots,v_n\}$.
Output: A min-width ordering of the nodes $d = (v_1,\dots,v_n)$.
1. for $j = n$ to 1 by $-1$ do
2.     $r \leftarrow$ a node in $G$ with smallest degree.
3.     Put $r$ in position $j$ and $G \leftarrow G - r$.
       (Delete from $V$ node $r$ and from $E$ all its adjacent edges)
4. endfor
```

Figure 4.2 The MIN-WIDTH (MW) ordering procedure.

an ordering having a width of 1, its induced ordered graph has no additional arcs, yielding an induced width of 1, as well. In summary, we have the following:

PROPOSITION 4.1 A graph is a tree iff it has induced width of 1. ●

4.1.1 Greedy Algorithms for Finding Induced Widths

Finding a minimum-width ordering of a graph can be accomplished by the greedy algorithm MIN-WIDTH (see Figure 4.2). The algorithm orders variables from last to first as follows: In the first step, a variable with a minimum number of neighbors is selected and put last in the ordering. The variable and all its adjacent edges are then eliminated from the original graph, and selection of the next variable continues recursively with the remaining graph. Ordering d_2 of G in Figure 4.1(c) could have been generated by a min-width ordering.

PROPOSITION 4.2 Algorithm MIN-WIDTH (MW) finds a minimum-width ordering of a graph.

Proof See Exercise 2. ●

Though finding the min-width ordering of a graph is easy, finding the minimum *induced width* of a graph is hard (NP-complete). Nevertheless, deciding whether there exists an ordering whose induced width is less than a constant k takes $O(n^k)$ time.

A decent greedy algorithm, obtained by a small modification to the MIN-WIDTH algorithm, is the MIN-INDUCED-WIDTH (MIW) algorithm (Figure 4.3). It orders the variables from last to first according to the following procedure: The algorithm selects a variable with minimum degree and places it last in the ordering. The algorithm next connects the node's neighbors in the graph to each other, and only

```
MIN-INDUCED-WIDTH (MIW)
Input: A graph $G = (V, E)$, $V = \{v_1,\dots,v_n\}$.
Output: An ordering of the nodes $d = (v_1,\dots,v_n)$.
1. for $j = n$ to 1 by $-1$ do
2.     $r \leftarrow$ a node in $V$ with smallest degree.
3.     Put $r$ in position $j$.
4.     Connect $r$'s neighbors: $E \leftarrow E \cup \{(v_i, v_j) | (v_i, r) \in E, (v_j, r) \in E\}$.
5.     Remove $r$ from the resulting graph: $V \leftarrow V - \{r\}$.
```

Figure 4.3 The MIN-INDUCED-WIDTH (MIW) procedure.

```
MIN-FILL (MF)
Input: A graph $G = (V, E)$, $V = \{v_1,\dots,v_n\}$.
Output: An ordering of the nodes $d = (v_1,\dots,v_n)$.
1. for $j = n$ to 1 by $-1$ do
2.     $r \leftarrow$ a node in $V$ with smallest fill edges for his parents.
3.     Put $r$ in position $j$.
4.     Connect $r$'s neighbors: $E \leftarrow E \cup \{(v_i, v_j) | (v_i, r) \in E, (v_j, r) \in E\}$.
5.     Remove $r$ from the resulting graph: $V \leftarrow V - \{r\}$.
```

Figure 4.4 The MIN-FILL (MF) procedure.

then removes the selected node and its adjacent edges from the graph, continuing recursively with the resulting graph. The ordered graph in Figure 4.1(c) could have also been generated by a min-induced-width ordering of G. In this case, it so happens that the algorithm achieves the overall minimum induced width of the graph, w^*. Another variation yields a greedy algorithm known as MIN-FILL. Rather than order the nodes in order of their min-degree, it uses the *min-fill set*, that is, the number of edges needed to be filled so that its parent set is fully connected, as an ordering criterion. This min-fill heuristic, described in Figure 4.4, was demonstrated empirically to be somewhat superior to the MIN-INDUCED-WIDTH algorithm. The ordered graph in Figure 4.1(c) could have been generated by a min-fill ordering of G, while the ordering d_1 or d_3 in parts (a) and (d) could not.

The notions of width and induced width, and their relationships with various graph parameters, have been studied extensively in the past two decades. Here we will focus only on those aspects that are relevant to constraint processing.

4.1.2 Chordal Graphs

Computing the induced width for chordal graphs is easy. A graph is *chordal* if every cycle of length at least 4 has a chord, that is, an edge connecting two non-adjacent vertices. For example, G in Figure 4.1(a) is not chordal since the cycle (A, B, D, C, A) does not have a chord. The graph can be made chordal if we add the edge (B, C) or the edge (A, D).

Many difficult graph problems become easy on chordal graphs. For example, finding all the maximal (largest) *cliques* (completely connected subgraphs) in a graph—an NP-complete task on general graphs—is easy for chordal graphs. This task (finding maximal cliques in chordal graphs) is facilitated by using yet another ordering procedure called the *max-cardinality ordering* (Tarjan and Yannakakis 1984). A max-cardinality ordering of a graph orders the vertices from *first to last* according to the following rule: The first node is chosen arbitrarily. From this point on, a node that is connected to a maximal number of already ordered vertices is selected, and so on. (See Figure 4.5.)

A max-cardinality ordering can be used to identify chordal graphs. Namely, a graph is chordal iff in a max-cardinality ordering each vertex and all its parents form a clique. You can thereby enumerate all maximal cliques associated with each vertex (by listing the sets of each vertex and its parents, and then identifying the maximal size of a clique). Notice that there are at most n cliques: each vertex and its parents are one such clique. Consequently, when using a max-cardinality ordering of a chordal graph, the ordered graph is identical to its induced graph, and therefore its width is identical to its induced width. Also:

PROPOSITION 4.3 If G* is the induced graph of a graph G, along some ordering, then G* is chordal.

Proof See Exercise 3. ●

```
MAX-CARDINALITY (MC)
Input: A graph $G = (V, E)$, $V = \{v_1, \dots, v_n\}$.
Output: An ordering of the nodes $d = (v_1, \dots, v_n)$.
1. Place an arbitrary node in position 0.
2. for $j = 1$ to $n$ do
3.     $r \leftarrow$ a node in $G$ that is connected to a largest subset of nodes
           in positions 1 to $j - 1$, breaking ties arbitrarily.
4. endfor
```

Figure 4.5 The MAX-CARDINALITY (MC) ordering procedure.

EXAMPLE 4.3 We see again that G in Figure 4.1(a) is not chordal since the parents of A are not connected in the max-cardinality ordering in Figure 4.1(d). If we connect B and C, the resulting induced graph is chordal. ●

4.1.3 *k*-Trees

A subclass of chordal graphs is *k-trees*. A k-tree is a chordal graph whose maximal cliques are of size $k+1$, and it can be defined recursively as follows: (1) A complete graph with k vertices is a k-tree. (2) A k-tree with r vertices can be extended to $r+1$ vertices by connecting the new vertex to all the vertices in any clique of size k.

k-trees were investigated extensively in the graph-theoretical literature. It was shown, for example, that a graph can be *embedded* in a k-tree if and only if it has an induced width $w^* \leq k$ (Arnborg 1985).

4.2 Directional Local Consistency

We now return to the primary target of this chapter: determining the amount of inference that can guarantee a backtrack-free solution.

The level of inference applied to a given constraint network can be restricted in a variety of ways. A general approach discussed in Chapter 3 is bounding the number of variables that participate in an inference to yield propagation methods such as arc-consistency or path-consistency. Another orthogonal approach is to restrict inference relative to a given ordering of the variables, in anticipation of subsequent processing by search.

Indeed, securing full arc-consistency, full path-consistency, and full i-consistency is sometimes unnecessary if a solution is going to be generated by search along a fixed variable ordering. Consider, for example, the task of applying search on a problem whose ordered constraint graph is given in Figure 4.6. To ensure that the search algorithm encounters no dead-ends when assigning values using ordering $d = (x_1, x_2, x_3, x_4)$, we need only make sure that any assignment to x_1 will have at

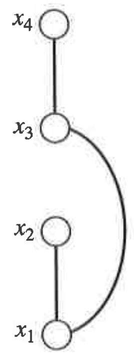


Figure 4.6 An ordered constraint graph.

one consistent corresponding value in x_2 and x_3, and that, subsequently, any ment to x_3 will have at least one consistent corresponding value in x_4. This can be achieved by making x_1 arc-consistent relative to x_2 and x_3, and x_3 arc-consistent relative to x_4. We don't need to ensure that x_2 and x_3 are arc-consistent relative to x_1, or that x_4 is arc-consistent relative to x_3. In other words, arc-consistency is required only in the direction to be exploited by the search algorithm.

In this section we develop the idea of *directional consistency*—restricting inference to a given variable ordering. We present algorithms for enforcing varying levels of directional consistency. We will see how graphical properties of the constraint graph can shed light on the design and analysis of such algorithms. We will further show that directional inference leads to a general and complete variable elimination algorithm called ADAPTIVE-CONSISTENCY. In Chapter 8 these ideas are extended to relational consistency.

Since we start with directional arc- and path-consistency that in their simplest form are relevant only to binary constraints, we will initially assume that the networks in question are binary, and then generalize to arbitrary constraints, to include the notions of generalized arc-consistency and relational consistency (in Chapter 8).

4.2.1 Directional Arc-Consistency

DEFINITION 4.2 **(directional arc-consistency)**

A network is *directional arc-consistent* relative to order $d = (x_1, \ldots, x_n)$ iff every variable x_i is arc-consistent relative to every variable x_j such that $i \leq j$. ●

An algorithm DAC for achieving directional arc-consistency along ordering $d = (x_1, \ldots, x_n)$ is given in Figure 4.7. It processes the variables in reverse order of d. When processing x_i, all the binary constraints incident to x_i, R_{ki} such that $k \leq i$, are considered and the corresponding domains D_k of x_k are tightened.

```
DAC($\mathcal{R}$)
Input: A network $\mathcal{R} = (X, D, C)$, its constraint graph $G$, and an ordering
    $d = (x_1,\ldots,x_n)$.
Output: A directional arc-consistent network.
1.  for $i = n$ to 1 by -1 do
2.      for each $j < i$ such that $R_{ji} \in \mathcal{R}$, do
3.          $D_j \leftarrow D_j \cap \pi_j (R_{ji} \bowtie D_i)$, (this is REVISE($(x_j), x_i$)).
4.  endfor
```

Figure 4.7 Directional arc-consistency (DAC).

EXAMPLE 4.4 Assume that the constraints and the domains of the problem in Figure 4.6 are specified below.

$$D_1 = \{red, white, black\}$$
$$D_2 = \{green, white, black\}$$
$$D_3 = \{red, white, blue\}$$
$$D_4 = \{white, blue, black\}$$
$$R_{12}: \; x_1 = x_2$$
$$R_{13}: \; x_1 = x_3$$
$$R_{34}: \; x_3 = x_4$$

Using the ordering $d = (x_1, x_2, x_3, x_4)$, the algorithm processes the variables in the reverse order along the ordered graph in Figure 4.6. Starting with x_4, DAC first revises x_3 relative to x_4, deleting *red* from D_3 (since *red* of x_3 has no match in D_4), yielding $D_3 = \{white, blue\}$. Since x_4 has only this one constraint, processing proceeds with x_3, and the constraint R_{13} is tested to ensure that x_1 is arc-consistent relative to x_3. As a result, *red* and *black* are eliminated from D_1 since they have no equals in the updated domain of D_3. When x_2 is next processed, nothing changes because x_1, with its current domain $D_1 = \{white\}$, is already arc-consistent relative to x_2. The final resulting domains are $D_1 = \{white\}$, $D_2 = \{green, white, black\}$, $D_3 = \{white, blue\}$, $D_4 = \{white, blue, black\}$, yielding a directional arc-consistent network relative to the given ordering.

Is the resulting network also full arc-consistent? Checking the constraint R_{31}, we see a violation of arc-consistency: variable x_3 is not arc-consistent relative to x_1; *blue* in D_3 has no match in D_1. Nevertheless, if we now try to assign values in the forward direction of ordering $d = (x_1, x_2, x_3, x_4)$, we will assign $x_1 = white$ (the only value in x_1's domain), then $x_2 = white$ (the only way to satisfy the equality constraint between x_1 and x_2), and similarly, $x_3 = white$ and $x_4 = white$. We see that despite lack of full arc-consistency a consistent assignment was made to every variable, and that no dead-end was encountered. ●

There is a distinct computational advantage to enforcing directional arc-consistency rather than full arc-consistency: each constraint is processed exactly once. Indeed, we have the following:

PROPOSITION 4.4 Given a network $\mathcal{R}$, and an ordering of its variables d, algorithm DAC generates a directional arc-consistent network relative to d, with time complexity of $O(ek^2)$, where e is the number of binary constraints (i.e., number of arcs) and k bounds the domain size.

Proof Because of the processing order, once the arc-consistency of x_i relative to a later x_j is enforced, the domain of x_j will not be revised again. In addition, even if the domain of x_i is subsequently revised (in order to enforce arc-consistency with another, intermediate variable), it can only shrink in size; no new members will be added to the domain of x_i, and therefore the enforced arc-consistency of x_i relative to x_j will be maintained. Since there are e binary constraints, each processed just once, and since the complexity of REVISE is $O(k^2)$, the complexity of DAC is bounded by $O(ek^2)$. •

Algorithm DAC seems optimal for achieving directional arc-consistency. To merely *verify* directional arc-consistency, each constraint needs to be inspected at least once, requiring $O(ek^2)$ consistency tests.

EXAMPLE 4.5 Let's now examine another constraint network. This time the constraints are between every pair of variables and are all not-equal constraints ($R_{ij} : x_i \neq x_j, i \neq j$), and the domains will all be {*red*, *blue*}. This network is already full arc-consistent, and so, by definition, it is also directional arc-consistent for any ordering. Therefore, applying DAC in any order will not change the domains of the variables. Does the network's directional arc-consistent standing guarantee a backtrack-free search for a consistent solution? Well, we see that it is possible to make the consistent partial assignment $(\langle x_1, red\rangle, \langle x_2, blue\rangle)$. However, there is no assignment to x_3 that satisfies both $R_{23} : x_2 \neq x_3$ and $R_{13} : x_1 \neq x_3$, and therefore a dead-end is encountered. We see that like full arc-consistency, DAC is insufficient to guarantee that every problem will be backtrack-free; higher levels of consistency may be necessary. •

4.2.2 Directional Path-Consistency

The same principle of restricting inference to a specific ordering can also be applied to path-consistency and to i-consistency in general. These consistency properties can be achieved more efficiently relative to one specific ordering than can the corresponding full consistency.

DEFINITION 4.3 **(directional path-consistency)**

A network $\mathcal{R}$ is *directional path-consistent* relative to order $d = (x_1, \ldots, x_n)$ iff for every $k \geq i, j$, the pair $\{x_i, x_j\}$ is path-consistent relative to x_k. •

EXAMPLE 4.6 Consider again the graph-coloring problem whose ordered constraint graph along $d = (x_1, x_2, x_3, x_4)$, having the domains {*red*, *blue*}, is depicted in Figure 4.8(a). As we saw in Chapter 3, this network is arc-consistent, but not path-consistent. For example, the (universal) constraint between

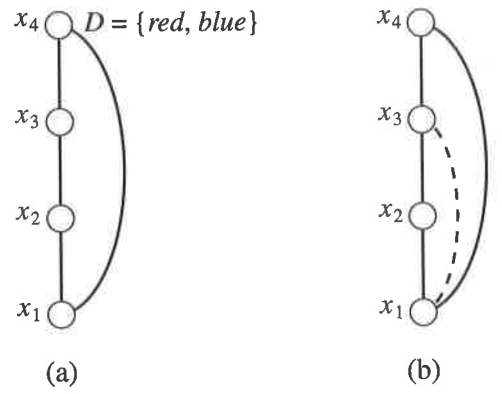


Figure 4.8 (a) An ordered constraint graph and (b) its induced graph.

x_3 and x_1 allows the assignment $x_1 = red$ and $x_3 = blue$, while there is no assignment to x_4 that both satisfies the relevant constraints and agrees with this instantiation. Enforcing full path-consistency on this network requires adding the constraints $R_{13} : x_1 = x_3$ and $R_{24} : x_2 = x_4$. However, for directional path-consistency relative to d only, we only need to add the constraint $x_1 = x_3$. This allows a solution to be assembled along order d without encountering dead-ends. •

As implied by the previous example, when the network is not directional path-consistent, directional path-consistency can be enforced. Algorithm DPC (Figure 4.9) achieves strong directional path-consistency (namely, both directional arc-consistency and directional path-consistency). The DPC algorithm processes the variables in reverse order of d. When processing variable x_k, it makes all the constraints R_{ij}, $i, j < k$, path-consistent relative to x_k, and it also enforces arc-consistency on all the variables x_i, $i < k$, relative to x_k. Step 3 of the algorithm is equivalent to REVISE$((x_i), x_k)$, and performs directional arc-consistency. Step 5 is equivalent to REVISE-3$((x_i, x_j), x_k)$: recording or updating the binary constraints inferred by a later third variable.

A new feature of this algorithm is its explicit reference to the underlying constraint graph; the algorithm manages not only the changes made to the constraints but also the changes made to the constraint graph, namely, adding new arcs that correspond to the added new constraints.

THEOREM 4.1 Given a binary network $\mathcal{R}$ and an ordering d, algorithm DPC generates a largest equivalent, strong, directional path-consistent network relative to d. The time and space complexity of DPC is $O(n^3k^3)$, where n is the number of variables and k bounds the domain sizes.

```
DPC($\mathcal{R}$)
Input: A binary network $\mathcal{R} = (X, D, C)$ and its constraint graph $G = (V, E)$,
    $d = (x_1, \ldots, x_n)$.
Output: A strong directional path-consistent network and its graph $G' = (V, E')$.
Initialize: $E' \leftarrow E$.
1.  for $k = n$ to 1 by -1 do
2.      (a) $\forall\, i \leq k$ such that $x_i$ is connected to $x_k$ in the graph, do
3.          $D_i \leftarrow D_i \cap \pi_i (R_{ik} \bowtie D_k)$ (REVISE $((x_i), x_k)$)
4.      (b) $\forall\, i, j \leq k$ such that $(x_i, x_k), (x_j, x_k) \in E'$ do
5.          $R_{ij} \leftarrow R_{ij} \cap \pi_{ij} (R_{ik} \bowtie D_k \bowtie R_{kj})$ (REVISE-3$((x_i, x_j), x_k)$)
6.          $E' \leftarrow E' \cup (x_i, x_j)$
7.  endfor
8.  return the revised constraint network $\mathcal{R}$ and $G' = (V, E')$.
```

Figure 4.9 Directional path-consistency (DPC).

Proof To prove the above theorem, all we need to show is that when DPC terminates every pair (x_i, x_j) is path-consistent relative to x_k, assuming $i \leq j \leq k$ in d. We know that when x_k was processed (step 5), the pair (x_i, x_j) was made path-consistent relative to x_k. We also know that this condition may be violated only if the domain of x_k is later reduced, or if the two binary constraints R_{ik} and R_{jk} are changed. However, by the time x_k is processed, these two constraints already have their final form determined because they can be affected only when processing variables appearing later than x_k, and those were all processed before x_k. Regarding complexity, the number of times the inner loop (steps 4, 5, and 6) is executed for variable x_i is at most $O(n^2)$ (the number of different pairs of earlier neighbors of variable x_i), and each step is at most $O(k^3)$, yielding an overall complexity of $O(n^3k^3)$. Since the computation of steps 2 and 3 is completely dominated by the computation of steps 4, 5, and 6, we find that the overall complexity is $O(n^3k^3)$. ●

4.2.3 Directional *i*-Consistency

In the previous two subsections we restricted our attention to binary networks because the constraints recorded were binary or unary only. Generalizing to directional i-consistency must account for constraints with larger scopes. We can lift the restriction of binary constraints and apply DAC or DPC to a general network, but we must restrict operation of these two algorithms to their binary subnetworks only.

With this approach, algorithm DAC still yields a directional arc-consistent network, and DPC yields a directional path-consistent network.

DEFINITION 4.4 **(directional *i*-consistency)**

A network is *directional i-consistent* relative to order $d = (x_1, \ldots, x_n)$ iff every $i-1$ variables are i-consistent relative to every variable that succeeds them in the ordering. A network is *strong directional i-consistent* if it is directional j-consistent for every $j \leq i$. ●

When extending directional consistency to i-consistency, we consider subnetworks defined on i variables. Given a general constraint network, algorithms for enforcing directional i-consistency can be obtained by generalizing DPC, replacing the composition operator in DPC (step 5) by the REVISE-i operator. A description of the generalized algorithm is given in Figure 4.10. Note that in this algorithm we use a generic REVISE procedure that operates on any size set as its first parameter. If the parent set of a variable has no more than $i-1$ variables, the procedure records a single constraint over the parent set (steps 2–4). Otherwise, we record a constraint over every subset of size $i-1$ of the parent set (step 6). Algorithm DIC$_i$ enforces

```
Directional i-consistency (DIC$_i$($\mathcal{R}$))
Input: A network $\mathcal{R} = (X, D, C)$, its constraint graph $G = (V, E)$, $d = (x_1, \ldots, x_n)$.
Output: A strong directional $i$-consistent network along $d$ and its graph
    $G' = (V, E')$.
Initialize: $E' \leftarrow E$, $C' \leftarrow C$.
1. for $j = n$ to 1 by -1 do
2. let $P = parents(x_j)$.
3.     if $|P| \leq i - 1$ then
4.         Revise $(P, x_j)$
5.     else, for each subset of $i - 1$ variables $S$, $S \subseteq P$, do
6.         Revise $(S, x_j)$
7.     endfor
8.     $C' \leftarrow C' \cup$ all generated constraints.
9.     $E' \leftarrow E' \cup \{(x_k, x_m) | x_k, x_m \in P\}$ (connect all parents of $x_j$)
10. endfor
11. return $C'$ and $E'$.
```

Figure 4.10 Algorithm directional i-consistency (DIC$_i$).

strong directional i-consistency (see Exercise 9). Its complexity will be addressed later using induced width.

EXAMPLE 4.7 Applying DIC3 to the network $\mathcal{R} = \{R_{xyz}\}$ in Example 3.9, along ordering $d = (x, y, z)$, will add the constraint $R_{xy} = \{(\langle x, 0\rangle\langle y, 0\rangle)\}$ in addition to the constraint $R_x = \{(\langle x, 0\rangle)\}$. •

Note that DIC3 and DPC are identical if the input network is binary, but they handle ternary constraints differently.

4.2.4 Graph Aspects of Directional Consistency

Neither directional arc-consistency nor full arc-consistency can change the constraint graph. Higher levels of directional consistency do change the constraint graph, although to a lesser extent than their nondirectional counterparts. For example, applying DPC to the network whose ordered graph is given in Figure 4.8(a) results in a network having the graph in Figure 4.8(b), where arc (x_1, x_3) is added. If we apply DPC to the problem in Figure 4.6, no constraint will be added, since the changes are only those caused by arc-consistency. Full path-consistency, on the other hand, will make the constraint graph of Figure 4.6 complete. Before continuing with this section, you should refresh your memory of the graph concepts defined in Section 4.1, if necessary.

During processing by DPC, a variable x_k only affects the constraint between a pair of earlier variables when it is constrained via binary constraints[1] and is thus connected to both earlier variables in the graph. In this case, a new constraint and a corresponding new arc may be added to the graph. Algorithm DPC recursively connects the parents of every two nodes in the ordered constraint graph, thus generating the *induced ordered graph*. Indeed, the graph in Figure 4.8(b) is the induced graph of the ordered graph in Figure 4.8(a).

PROPOSITION 4.5 Let (G, d) be the ordered constraint graph of a binary network $\mathcal{R}$. If *DPC* is applied to $\mathcal{R}$ relative to order d, then the graph of the resulting constraint network is subsumed by the induced graph (G^*, d).

Proof Let G be the original constraint graph of $\mathcal{R}$, and let G_1 be the constraint graph of the problem generated by applying DPC to $\mathcal{R}$ along d. We prove the above claim by induction on the variables along the reverse ordering of $d = (x_1, \ldots, x_n)$. The induction hypothesis is that all the arcs incident to $x_n, \ldots, x_i$ in G_1 appear also in (G^*, d). The claim is true for x_n (the induction base step), since its connectivity is equivalent in both graphs.

1. Remember that, as defined, DPC manipulates only the binary constraints.

Assume that the claim is true for $x_n, \ldots, x_i$, and we will show that it holds also for $i - 1$ (i.e., for $x_n, \ldots, x_{i-1}$). Namely, we will show that if (x_{i-1}, x_j), $j < i - 1$, is an arc in G_1, then it is also in (G^*, d). There are two cases: either x_{i-1} and x_j are connected in G_1 (i.e., they have a binary constraint in $\mathcal{R}$, and therefore they will stay connected in (G^*, d)), or a binary constraint over $\{x_{i-1}, x_j\}$ was added by DPC. In the second case, this new binary constraint was obtained while processing some later variable x_t, $t > i - 1$. Since a constraint over x_{i-1} and x_j is generated by DPC, both x_j and x_{i-1} must be connected to x_t in G_1 and, following the induction hypothesis, each will also be connected to x_t in (G^*, d). Therefore, x_{i-1} and x_j will become connected when generating the induced graph (G^*, d) (i.e., when connecting the parents of x_t). •

The induced graph and its induced width can be used to refine the worst-case complexity of DPC along d. Since DPC processes only pairs of variables selected from the set of *current* parents in the ordered constraint graph, the number of such pairs can be bounded by the parent set size of each variable, namely, by $w^*(d)$. We conclude the following:

THEOREM 4.2 Given a binary network $\mathcal{R}$ and an ordering d, the complexity of DPC along d is $O((w^*(d))^2 \cdot n \cdot k^3)$, where $w^*(d)$ is the induced width of the ordered constraint graph along d.

Proof Proposition 4.5 asserts that when a variable x is being processed by DPC it is connected to at most $w^*(d)$ parents. Therefore the number of triplets that a variable can share with its parents is $O(w^*(d)^2)$. Since processing each triplet is $O(k^3)$, and since there are n variables altogether, we get the complexity bound claimed above. •

Consequently, orderings with a small induced width allow DPC to be more efficient. Rather than being governed by cubic complexity, DPC is linear in the number of variables for networks whose constraint graph has bounded induced width.

The complexity of general DIC$_i$ can be bounded using the induced width as well. Given a general network whose constraint scopes are bounded by i, applying DIC$_i$ in any ordering connects the parents of every node in the ordered *primal* constraint graph (restricted to constraints of arity i or less), yielding again its induced graph. We can now extend Proposition 4.5 and Theorem 4.2 to the general i-consistency case.

PROPOSITION 4.6 Given a network $\mathcal{R}$ whose constraint arity is bounded by i, if DIC$_i$ is applied to $\mathcal{R}$ relative to order d, then the primal graph of the resulting constraint network is subsumed by the induced graph (G^*, d).

Proof See Exercise 8. •

The complexity of the algorithm is determined by the REVISE procedure. If the algorithm records a constraint on a set of size j, its complexity is $O((2k)^j)$. Since the size of the parent set is bounded by $w^*(d)$, and the number of its subsets of size i is bounded by $O((w^*(d))^i)$, we conclude the following:

THEOREM 4.3 Given a general constraint network $\mathcal{R}$ whose constraints' arity is bounded by i, and an ordering d, the complexity of DIC$_i$ along d is $O(n(w^*(d))^i \cdot (2k)^i)$. •

4.3 Width versus Local Consistency

In Example 4.6, we saw that DPC changed the network so that a solution could be found in a backtrack-free manner. However, it is easy to come up with examples where DPC (or even full path-consistency) would not suffice for making the network backtrack-free. Clearly, it would be highly desirable to have a criterion that could identify, *in advance*, the level of consistency sufficient for generating a backtrack-free representation for a given constraint network. Such a criterion can be provided, based on the induced width of the network's graph. Let's start with the special case of width 1.

4.3.1 Solving Trees: Case of Width 1

In the example graph of Figure 4.6, we saw that directional arc-consistency generated a backtrack-free network. However, it will not generate a backtrack-free network when we add a not-equal constraint between x_2 and x_4 augmenting Example 4.4. Notice also that the graph of Figure 4.6 did not have cycles, although it did once we added the constraint. Indeed these examples illustrate the general characteristics that any arc-consistent tree-structured binary network is backtrack-free for a variety of orderings. Moreover, if a tree network is not arc-consistent, arc-consistency can be enforced. We will use the concept of width to express this relationship between graphs and local consistency. Remember (from Section 4.1) that a graph is a tree iff it has a width-1 ordering.

However, as we observed before, attaining full arc-consistency is not necessary for achieving backtrack-free solutions on trees. For example, if the constraint network in Figure 4.11 is assigned values (by backtrack search) along ordering $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$, we need only make sure that any value assigned to variable x_1 will have at least one consistent value in x_2. Notice that the tree in Figure 4.11, along $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$, has a width of 1, while along the ordering $d_2 = (x_4, x_5, x_6, x_7, x_2, x_3, x_1)$, the width is 2. We can conclude the following:

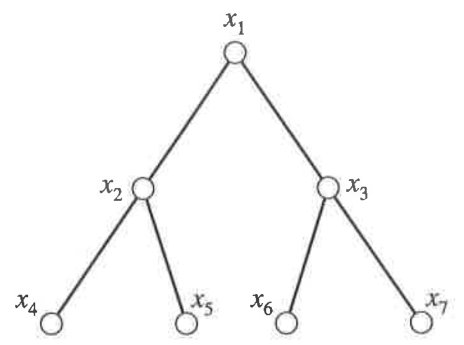


Figure 4.11 A tree network.

```
TREE-SOLVING
Input: A tree network $T = (X, D, C)$.
Output: A backtrack-free network along an ordering $d$.
1.  Generate a width-1 ordering, $d = x_1, \ldots, x_n$ along a rooted tree.
2.  let $x_{p(i)}$ denote the parent of $x_i$ in the rooted ordered tree.
3.  for $i = n$ to 1 do
4.      REVISE $((x_{p(i)}), x_i)$;
5.      if the domain of $x_{p(i)}$ is empty, exit (no solution exists).
6.  endfor
```

Figure 4.12 TREE-SOLVING algorithm.

THEOREM 4.4 **(width 1 and directional arc-consistency)**

Let d be a width-1 ordering of a constraint tree T. If T is directional arc-consistent relative to d, then the network is backtrack-free along d.

Proof Consider a width-1 ordering $d = (x_1, \ldots, x_n)$. Let's assume that a subset of variables $x_1, \ldots, x_i$ was instantiated consistently and that we now need to instantiate x_{i+1}. Since d is a width-1 ordering, there is only one parent variable x_j ($j < i$) that may constrain x_{i+1}. Since x_j is arc-consistent relative to x_{i+1}, it must have a legal value consistent with the current assignment to x_j, and this value provides a consistent extension to the partial solution. •

So, if we have a width-1 ordering of a binary constraint network, we can apply algorithm DAC along that ordering, thus enforcing directional arc-consistency, and then find a backtrack-free solution. The tree-solving algorithm in Figure 4.12 presents these steps explicitly. Step 1 generates a *rooted-directed* tree that corresponds to various width-1 orderings. Steps 3–5 apply directional arc-consistency. Clearly (from Theorem 4.4) the tree-solving algorithm is complete for trees, and its complexity is $O(nk^2)$, the complexity of DAC.

Interestingly enough, if we apply DAC relative to a width-1 order d and *then* apply DAC relative to the reverse order of d, we will achieve full arc-consistency for binary trees in $O(nk^2)$ steps. In contrast, if algorithm AC-3 had been applied to a tree, its worst-case performance is $O(nk^3)$. If algorithm AC-4 is applied to trees, it also has a complexity of $O(nk^2)$, but at the cost of a much more involved implementation.

4.3.2 Solving Width-2 Problems

Can we also make a width-2 network backtrack-free? To some extent, the answer is yes. Let's extend the relationship observed in Theorem 4.4:

THEOREM 4.5 **(width 2 and directional path-consistency)**

If $\mathcal{R}$ is directional arc- and path-consistent along d, and if it also has width 2 along d, then it is backtrack-free along d.

Proof To ensure that a width-2 ordered constraint network is backtrack-free, it is required that each variable selected for instantiation will have some values in its domain that are consistent with all previously chosen values. Suppose that $x_1, x_2, \ldots, x_k$ were already instantiated. Having a width-2 ordering implies that variable x_{k+1} is constrained with at most two previous variables, x_i and x_j, $i, j \le k$. Since the problem is directional path-consistent, for any assignment of values to x_i and x_j, there exists a consistent assignment for x_{k+1}. If x_{k+1} is constrained by only one previous variable, directional arc-consistency ensures the existence of a consistent extension to x_{k+1} as well. •

If a problem has a width-2 ordering but is *not* directional path-consistent, then we may consider enforcing directional path-consistency by DPC. However, as we saw in Section 4.2.2, applying DPC may add arcs to the problem's graph and increase its width, now its induced width $w^*(d)$, above 2. Therefore, applying DPC to width-2 problems does *not* guarantee a backtrack-free solution, unless $w^*(d) = 2$. A ring is a good example of a problem whose width and induced width is 2. Figure 4.13 shows an ordered ring and its induced graph. Both graphs have a width of 2.

THEOREM 4.6 A binary constraint network $\mathcal{R}$ having an induced width of 2 can be solved in linear time in the number of variables namely in $O(nk^3)$.

Proof Let d be an ordering of a binary constraint network $\mathcal{R}$ for which $w(d) = w^*(d) = 2$. Such a problem can be solved by first applying DPC

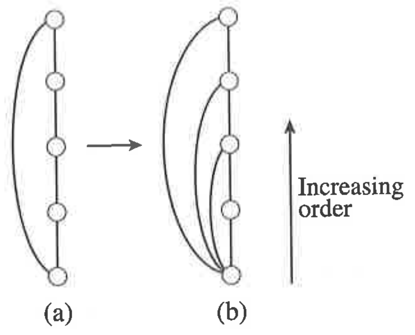


Figure 4.13 An induced width-2 graph. (a) Width-2. (b) Its induced width-2 graph.

relative to d, followed by a backtrack-free value assignment. For problems having an induced width of 2, DPC is bounded by $O(nk^3)$ steps. •

How easy is it to determine if a graph has an ordering with an induced width of 2? Clearly, enumerating all orderings may be hard. Fortunately there is a linear time algorithm for recognizing such graphs: the MIN-INDUCED-WIDTH (MIW) algorithm described in Section 4.1. Remember, MIW orders the nodes in decreasing order. It selects a node with smallest degree, places it last in the ordering, connects its neighbors, removes it from the graph, and continues recursively. If at anytime a selected node's degree is larger than 2, the graph must have an induced width higher than 2 (see Exercise 14).

4.3.3 Solving Width-*i* Problems

The relationship between graph width and the tractability of the problem can be extended to general nonbinary networks. A problem is *backtrack-free* along d if the *level* of its directional strong consistency along this order is greater than the *width* of the ordered constraint graph.

The intuition behind this claim rests on the fact that when backtracking works along a given ordering, it only tests the consistency of the relevant constraints among past and current variables. If these past constraints already ensure that a *locally consistent* partial solution will remain consistent relative to future variables, a dead-end will not occur. When a future variable is constrained with many past variables (i.e., when it has a high *width*) the required level of local consistency on past variables is higher. Generalizing previous theorems yields the following:

THEOREM 4.7 **(Width $i-1$ and directional i-consistency)**

Given a general network $\mathcal{R}$, if its ordered constraint graph along d has a width of $i-1$, and if it is also strong directional i-consistent, then $\mathcal{R}$ is *backtrack-free* along d.

Proof Assume a network with width $i-1$ along d that is strong i-consistent. Then, given any partial solution $\bar{a} = (\langle x_1, a_1\rangle, \ldots, \langle x_{i-1}, a_{l-1}\rangle)$ and given the next variable x_l, because the width is $i-1$, x_l is connected to a subset of variables S_i, $S_i \subseteq \{x_1, \ldots, x_l\}$, of at most $i-1$ variables. Because of strong i-consistency, the partial assignment $\bar{a}$ restricted to S_i, $\bar{a}[S_i]$, which is of length $i-1$ or less, is consistent with some value of x_l. Therefore, there exists a consistent extension of $\bar{a}$ to x_l. •

Because most problem instances do not satisfy the desired relationship between width and local consistency, we may want to increase the level of strong directional consistency until it matches the width of the problem. Specifically, if a width-$(i-1)$ problem is not i-consistent, algorithms enforcing directional i-consistency should be applied.

However, as in the case of width 2, algorithm DIC$_i$ augments the network with additional constraints (either binary or nonbinary), yielding a denser constraint graph. The resulting graph is identical to (or subsumed by) the induced ordered graph along the processed ordering. In other words, enforcing directional i-consistency guarantees a backtrack-free solution if the network's *induced* width along the processed order is $i-1$. This leads us to a simple procedure. Given a problem, select an ordering having a small width $w(d)$, compute its induced width $w^*(d)$, and then apply strong directional $(w^*(d)+1)$-consistency. The resulting network must satisfy the desired relationship—and is therefore backtrack-free.

4.4 Adaptive Consistency and Bucket Elimination

Algorithm ADAPTIVE-CONSISTENCY (ADC1) in Figure 4.14 implements the adaptive procedure suggested at the end of the previous section. Given an ordering d, ADC1 establishes directional i-consistency recursively, changing levels from node to node to *adapt* to the changing width of nodes at the time of processing. The algorithm is just DIC$_i$, when i is adaptive. This approach works because by the time a node is processed, its final induced width is determined, and the matching level of consistency can be achieved. The procedure may impose new constraints over certain subsets of variables, as well as tighten existing constraints.

Another way to look at adaptive consistency is as a variable elimination algorithm. That is, at each step, one variable and all its related constraints are solved, and a constraint is inferred on all the rest of the participating variables. We next provide an alternative description of adaptive consistency that employs a data structure, called *buckets*, which provides a convenient way for describing variable elimination algorithms, avoiding an explicit reference to the constraint graph. This description highlights important properties of the algorithm and unifies variable elimination algorithms for a variety of tasks. The idea is to associate a bucket with each variable

```
ADAPTIVE-CONSISTENCY (ADC1)
Input: A constraint network $\mathcal{R} = (X, D, C)$, its constraint graph $G = (V, E)$,
    $d = (x_1, \ldots, x_n)$.
output: A backtrack-free network along $d$.
Initialize: $C' \leftarrow C$, $E' \leftarrow E$
1. for $j = n$ to 1 do
2.     Let $S \leftarrow parents(x_j)$.
3.     $R_S \leftarrow$ REVISE $(S, x_j)$ (generate all partial solutions over $S$ that can be
         extended to $x_j$).
4.     $C' \leftarrow C' \cup R_S$
5.     $E' \leftarrow E' \cup \{(x_k, x_r) | x_k, x_r \in parents(x_j)\}$ (connect all parents of $x_j$)
6. endfor
```

Figure 4.14 Algorithm ADAPTIVE-CONSISTENCY—version 1.

and, given an ordering, to place each constraint into the bucket of the variable that appears latest in its scope. By doing so, we will have collected in the various buckets all the constraints that share the same latest variable in their scope. Subsequently, buckets are processed in reverse order. Processing a bucket means solving a subproblem and recording its solutions as a new constraint. This operation is equivalent to the REVISE procedure. The newly generated constraint is placed in the bucket of *its* latest variable. The *bucket elimination* algorithm adaptive consistency ADC in Figure 4.15 does precisely this. You can verify that the two descriptions of adaptive consistency coincide. From our earlier discussion, we obtain the following:

PROPOSITION 4.7 Let $\mathcal{R}$ be a constraint network and d an ordering for which $w^*(d) = i$. Then (1) applying adaptive consistency is identical to applying strong directional $(i+1)$-consistency along d, and (2) the constraint graph of the resulting network has width bounded by i. •

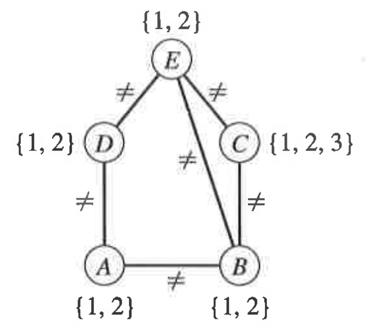
THEOREM 4.8 Given a network $\mathcal{R}$, adaptive consistency (either version ADC1 or ADC) determines the consistency of $\mathcal{R}$, and if the network is consistent, it also generates an equivalent representation $E_d(\mathcal{R})$ that is backtrack-free along d.

Proof From Proposition 4.7 it follows that ADC generates a width-i problem that is $(i+1)$-consistent, for some i, and thus, from Proposition 4.7, it follows that the resulting network $E_d(\mathcal{R})$ is backtrack-free along the order of processing. •

```
ADAPTIVE-CONSISTENCY (ADC)
Input: A constraint network $\mathcal{R}$, an ordering $d = (x_1,\ldots,x_n)$.
Output: A backtrack-free network, denoted $E_d(\mathcal{R})$, along $d$, if the empty
   constraint was not generated. Else, the problem is inconsistent.
1.  Partition constraints into $bucket_1,\ldots,bucket_n$ as follows:
    for $i \leftarrow n$ downto 1, put in $bucket_i$ all unplaced constraints mentioning $x_i$.
2.  for $p \leftarrow n$ downto 1 do
3.      for all the constraints $R_{S_1},\ldots,R_{S_j}$ in $bucket_p$ do
4.          $A \leftarrow \bigcup_{i=1}^{j} S_i - \{x_p\}$
5.          $R_A \leftarrow \pi_A(\bowtie_{i=1}^{j} R_{S_i})$
6.          if $R_A$ is not the empty relation then add $R_A$ to the bucket
               of the latest variable in scope $A$,
7.          else exit and return the empty network
8.  return $E_d(\mathcal{R}) = (X, D, bucket_1 \cup bucket_2 \cup \cdots \cup bucket_n)$
```

Figure 4.15 ADAPTIVE-CONSISTENCY as a bucket elimination algorithm.

EXAMPLE 4.8 Consider the graph-coloring problem depicted in Figure 4.16 (domains are numbers). The figure shows a schematic execution of adaptive consistency using the bucket data structure for the two orderings $d_1 = (E, B, C, D, A)$ and $d_2 = (A, B, C, D, E)$. The initial constraints, partitioned into buckets for both orderings, are displayed in the figure to the left of the double bars, while the constraints generated by the algorithm are displayed to the right of the double bars, in their respective buckets. Focusing on ordering d_2, adaptive consistency proceeds from E to A and imposes constraints on the parents of each processed variable, which are those variables appearing in its bucket. Processing the bucket of E, the problem, composed of three constraints in the buckets, is solved and the solution is projected over D, C, B, recording the ternary constraint R_{DCB}, which is placed in the bucket of D. Next, the algorithm processes D's bucket, which contains $D \neq A$ and the new constraint R_{DCB}. Joining these two constraints and projecting out D yields a constraint R_{ACB}, which is placed in the bucket of C, and so on. Processing the buckets along ordering d_1 causes the generation of a different set of constraints. Observe that while only binary constraints are created along order d_1, it is possible that ternary constraints are generated along ordering d_2. Notice also that along ordering d_1 two



Ordering d_1

$Bucket(A)$: $A \neq D, A \neq B$

$Bucket(D)$: $D \neq E \;||\; R_{DB}$

$Bucket(C)$: $C \neq B, C \neq E$

$Bucket(B)$: $B \neq E \;||\; R^1_{BE}, R^2_{BE}$

$Bucket(E)$: $||\; R_E$

Ordering d_2

$Bucket(E)$: $E \neq D, E \neq C, E \neq B$

$Bucket(D)$: $D \neq A \;||\; R_{DCB}$

$Bucket(C)$: $C \neq B \;||\; R_{ACB}$

$Bucket(B)$: $B \neq E \;||\; R_{AB}$

$Bucket(A)$: $||\; R_A$

Figure 4.16 Execution of ADC along two orderings.

constraints on scope B and E are generated, in the bucket of C and in the bucket of D, denoted R^1_{BE} and R^2_{BE}, respectively, in the figure.

Furthermore, the constraint R^1_{BE} means $B \neq E$ and is displayed—for illustration only—in the bucket of B, since there is already an identical original constraint. Also, the constraint R^2_{BE} is the universal constraint and should therefore not be recorded at all; we chose to display it only to illustrate the general case. When processing along ordering d_2, we only indicated the scheme of the constraints (i.e., their scope), leaving out their explicit description. ●

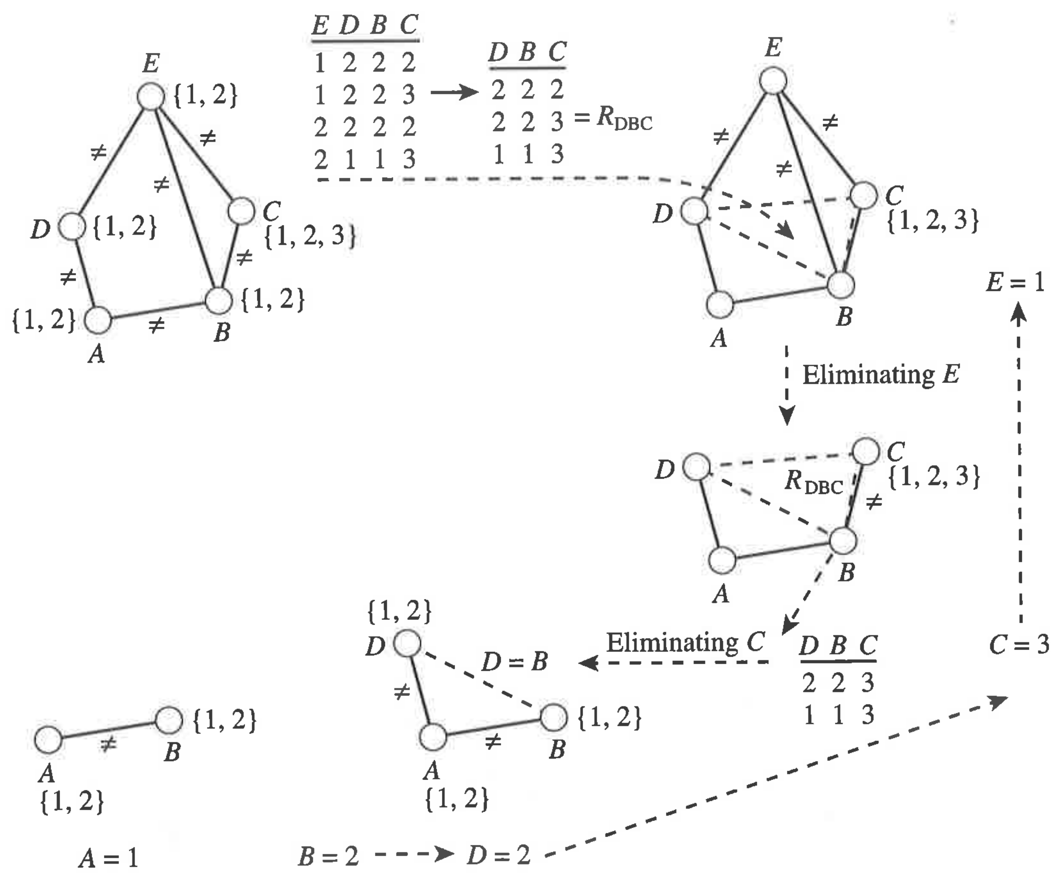


Figure 4.17 A schematic variable elimination and solution generation process is backtrack-free.

An alternative graphical illustration of the algorithm's performance along d_2 is given in Figure 4.17. The figure shows, through the changing graph, how constraints are generated in one ordering, and how a solution is created in the reverse order.

Generating the induced graph along the orderings $d_1 = E, B, C, D, A$ and $d_2 = A, B, C, D, E$ leads to the two graphs in Figure 4.18. The broken arcs are the newly added arcs. The induced width along d_1 and d_2 are 3 and 2, respectively, suggesting different complexity bounds for adaptive consistency. Algorithm ADC is linear in the number of buckets n and in the time to process each bucket. However, since processing each bucket amounts to generating all the solutions of a subproblem, its complexity is exponential in the number of variables appearing in the bucket. The important observation is that the number of variables in a bucket is bounded by

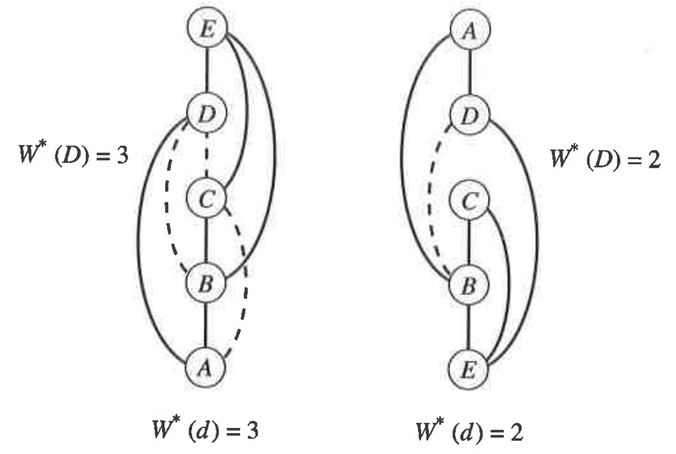


Figure 4.18 The induced width along the orderings: $d_1 = A, B, C, D, E$ and $d_2 = E, B, C, D, A$.

the number of parents of the corresponding variable in the induced ordered graph, namely, by the induced width.

THEOREM 4.9 The time and space complexity of ADAPTIVE-CONSISTENCY is $O(n \cdot (2k)^{w*+1})$ and $O(n \cdot k^{w*})$, respectively, where n is the number of variables, k bounds the domain size, and $w*$ is the induced width along the order of processing. When r bounds the number of constraints, the complexity can be bounded by $O(rk^{w^*(d)+1})$.

Proof The number of constraints (relations) in each bucket will increase to at most $2^{w^*(d)+1}$ relations because there are at most $w^* + 1$ variables in a bucket. Therefore testing that many constraints over all $O((k)^{w^*+1})$ tuples yields the overall complexity of $O(n \cdot (2k)^{w^*(d)+1})$. Alternatively, since the total number of function input and those generated is bounded by $2r$, and since the computation in a bucket is $O(r_i k^{w^*(d)+1})$, where r_i is the number of functions in a bucket, the total over-all buckets is $O(rk^{w^*(d)+1})$. ●

The analysis in this chapter yields a class of tractable problems based on the induced width of the constraint graph. Problems having bounded induced width $(w^* \leq b)$ for some constant b can be solved in polynomial time. In particular, when applied to trees, ADC coincides with DAC (directional arc-consistency), as demonstrated in Figure 4.19. Since the graph is cycle-free, when ordered along $d = (A, B, C, D, E, F, G)$, its width and induced width are 1. Indeed, as demonstrated by the schematic execution along d in Figure 4.19, ADC generates only unary relationships.

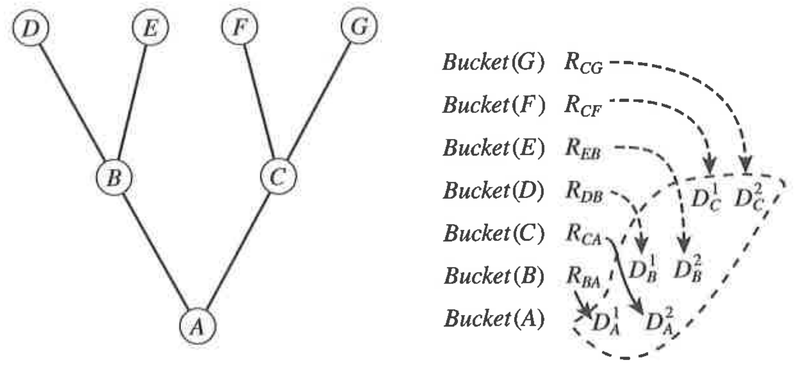


Figure 4.19 Schematic execution of ADAPTIVE-CONSISTENCY on a tree network. D_X denotes unary constraints over X.

Although finding the induced width of a graph is hard, deciding if the width is less than a constant b can be done in polynomial time. Consequently, w^* yields a characterization of a subclass of constraint networks that is tractable. In summary, we have the following:

THEOREM 4.10 **(w*-based tractability)**

The class of constraint problems whose induced width is bounded by a constant b is solvable in polynomial time and space. ●

ADAPTIVE-CONSISTENCY should be evaluated not merely as a procedure for deciding consistency, but also as a compilation algorithm since it transforms the constraint problem into an equivalent network from which *every* solution can be assembled in linear time. Therefore, when the output network can be substantially compacted, ADC may be cost-effective for compilation, even if it takes substantial time to generate the compiled network. However, when the worst-case space complexity reflects the actual output network, the value of adaptive consistency—both as a one-time solution process and as a compilation algorithm—is governed by the same induced-width parameter.

4.5 Summary

This chapter introduced the notion of bounded directional consistency algorithms such as directional arc-, path-, and i-consistency. These are incomplete inference algorithms that can sometimes decide inconsistency and that are designed as preprocessing algorithms to be used before backtracking search. As we'll see in the next chapter, such algorithms can also be interleaved with search. We also presented a relationship between induced width and consistency levels that guarantees

a backtrack-free solution: if the problem has width i and it is $(i + 1)$-consistent, then it is backtrack-free. This condition yields the identification of tractable problem classes based on the induced width: if a problem has induced width b, then it can be made backtrack-free by $\text{DIC}_{(b+1)}$, which coincides with adaptive consistency.

While finding the induced width of a problem is NP-complete, determining if the induced width of a graph is less than a constant b can be done in polynomial time, exponential in b (Arnborg 1985), and therefore the induced width indeed identifies tractability classes for constraint satisfaction problems. In particular, trees and problems having induced width 2 can be solved linearly. Finally, the inference algorithm ADAPTIVE-CONSISTENCY (ADC) was shown to make any problem backtrack-free relative to a given variable ordering. It is described as a variable elimination algorithm using the bucket data structure.

The chapter also provides a background section to graph concepts such as width and induced width. Related graph concepts (e.g., tree width) will be presented in Chapter 9. Chapter 8 extends algorithms appearing in this chapter to the more general consistency notion of relational consistency.

4.6 Bibliographical Notes

The fundamental relationship between width and consistency level that guarantees a backtrack-free solution was introduced by Freuder (1982). This relationship was extended by Dechter and Pearl (1987b) to the more restricted concept of directional consistency. It also led to algorithm ADAPTIVE-CONSISTENCY and to the identification of the induced width as the principal graph parameter that controls the algorithms's complexity. A similar elimination algorithm was introduced earlier by Seidel (1981). It was observed that these algorithms belong to the class of dynamic programming algorithms as presented in Bertele and Brioschi (1972). In Dechter and Pearl (1989), the connection between ADAPTIVE-CONSISTENCY and tree-clustering algorithms was made explicit, as will be shown in Chapter 9.

The analysis of the role of graph-based parameters in the complexity of various variable elimination algorithms, and the connection between tree width, induced width, hypertrees, and join-trees was observed independently in the areas of relational databases (Maier 1983), dynamic programming (Bertele and Brioschi 1972), and graph theory (Arnborg 1985; Corneil, Arnborg, and Proskourowski 1987; Arnborg and Proskourowski 1989).

In their book on nonserial dynamic programming, Bertele and Brioschi (1972) show the dependence of variable elimination algorithms for solving optimization tasks on a graph parameter (which they called "dimension" and what we call "induced width"). They suggest several greedy heuristics, including the MIN-INDUCED-WIDTH (not under this name) ordering, and show that it is complete for graphs having an induced width of 2. Montanari (1974) also observes that series-parallel binary constraint networks are tractable, a class that is identical to networks having an induced width of 2. In the field of relational databases the concept of join-tree was defined and observed as a desired representation. The connection between join-trees and chordal graphs was identified (Beeri et al. 1983), and the max-cardinality order was shown to be an identifier of chordal graphs by Tarjan and Yannakakis (1984). The practical value of the min-fill heuristic has been experimentally shown to produce elimination orders with small induced width by Kjaerulff, (1990, 1992).

Finally an extensive analysis of related concepts defined over hypertrees and the notion of tree width, and their connection with a variety of graph properties, were comprehensively analyzed in several papers by Arnborg and his colleagues (Arnborg 1985; Corneil, Arnborg, and Proskourowski 1987; Arnbourg and Proskourowski 1989). In particular, Arnborg proved that finding the tree width of an arbitrary graph is NP-complete. Nevertheless, deciding whether there exists an ordering whose induced width is less than a constant b takes $O(n^b)$ time. A more recent analysis is given by Bodlaender (1997). Approximation algorithms with some good guarantees have been and continue to be developed (Bar-Yehuda, Becker, and Geiger 1999; Shoiket and Geiger 1997).

4.7 Exercises

1. (*) Show that it takes $O(n^b)$ time to decide whether a graph G has an ordering whose induced width is less than a constant b.[2]
2. Prove that algorithm MIN-WIDTH achieves min-width ordering of a graph.
3. Let (G^*, d) be an ordered induced graph. Prove that G^* is chordal.
4. Prove that if you apply arc-consistency on a tree, from leaves to root and back, you get an arc-consistent network.
5. Prove that the tree-solving algorithm in Figure 4.12 is optimal.
6. Generate a directional path-consistent 4-queens problem along the columns ordered from left to right.
7. Consider the graph in Figure 4.20.
   (a) What is the induced width of the graph? Provide an ordering having minimum induced width.
   (b) Assume that the graph expresses a binary constraint network with some arbitrary constraints (e.g., not-equal). Provide a complexity bound using

2. An asterisk (*) indicates a relatively difficult problem.

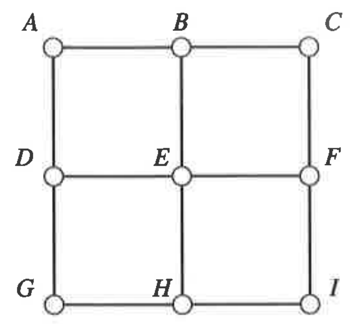


Figure 4.20 Grid with nine nodes.

   the induced width for applying algorithm DPC along the optimal induced-width ordering of this problem.
   (c) Can algorithm DPC always determine consistency of every constraint problem having an $n \times n$ grid constraint graph?
8. Prove Proposition 4.6.
9. Prove that algorithm DIC$_i$ generates a strong directional i-consistent network.
10. Consider 2-CNF formulas (conjunction of clauses of length 2).
    (a) Describe a DAC-type algorithm for enforcing directional arc-consistency of 2-CNF formulas using resolution.
    (b) Describe a DPC-type algorithm for enforcing directional path-consistency on 2-CNFs using resolution.
11. Consider the graph-coloring problem given in Figure 4.21. The constraints are not-equal constraints and the domains are indicated inside the nodes in the graph.
    (a) Generate a directional strong path-consistent network for this problem.
    (b) Generate a backtrack-free problem using adaptive consistency.
    (c) Find a solution to the problem.
12. Consider the crossword puzzle in Figure 4.22.
    (a) Model the problem as a binary CSP with the words as variables. Draw its constraint graph.
    (b) Generate a min-induced-width and max-cardinality ordering of the constraint graph. Generate the induced graph along these orderings. What is the w^* of this problem?

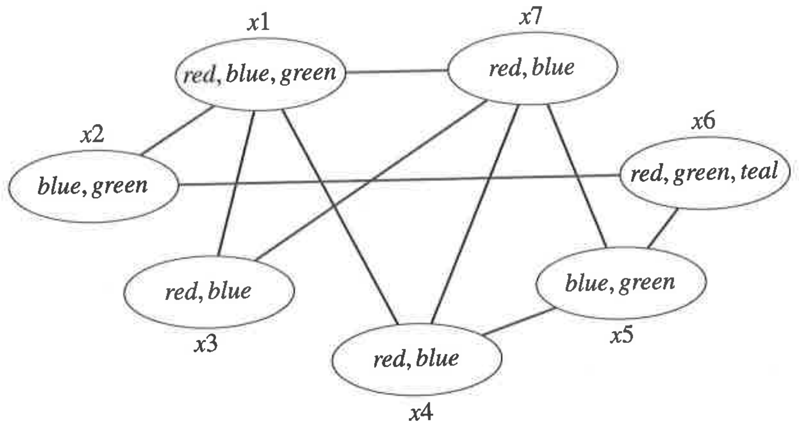
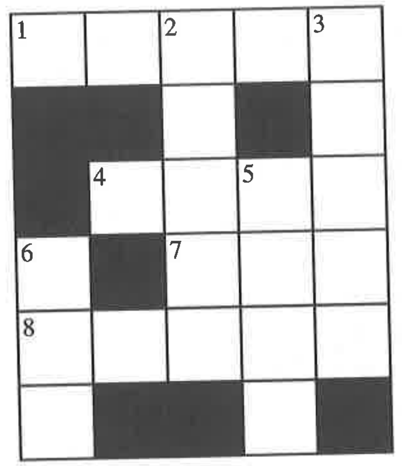


Figure 4.21 A coloring problem.



Word list	
aft	laser
ale	lee
eel	line
heel	sails
hike	sheet
hoses	steer
keel	tie
knot	

Figure 4.22 A crossword puzzle.

(c) What level of directional *i*-consistency is guaranteed to generate a backtrack-free representation for the ordered graphs that you picked, assuming you know nothing about the constraints themselves?

(d) Using the min-induced-width ordering, show the constraints that will be recorded for this problem when applying

i. directional arc-consistency

ii. directional path-consistency

iii. adaptive consistency (in this case, show the constraint subsets generated for all buckets, and also the actual constraints generated when processing the first three buckets)

13. Consider the crypto-arithmetic problem TA + DB = GBA when using the formulation of the problem with carries.

(a) Draw the primal constraint graph of the problem.

(b) Find an ordering of the variables using the MIN-INDUCED-WIDTH and MIN-FILL algorithms. What is the width of the orderings you generated? Compute the induced graph and the induced width of the orderings generated.

(c) Hand-simulate algorithm ADAPTIVE-CONSISTENCY on the ordering you created in (b). First describe the schemes of the initial partitioning into buckets, then show how new relations are created. Describe only the scopes of the relations.

(d) Bound the complexity of the algorithm on the ordering of your choice.

(e) Code algorithm ADAPTIVE-CONSISTENCY and apply it to this problem.

14. Prove that the MIW algorithm (Figure 4.3) can decide if a graph has an induced width of 2.