Chapter 4

Directional Consistency

4.1 Introduction

Although the general constraint satisfaction problem is hard, there are many subclasses of constraint networks that are easy to process. The identification of tractable classes has attracted significant effort in Artificial Intelligence. It is believed that intelligent behavior will be achieved through the utilization of idealized models of the task's environment. In particular, to explain how people perform so well on tasks that are theoretically intractable, we can assume that approximation methods based on idealized, easy to solve, models compose a significant portion of intelligent activity. Some real-life problems may fall into easy classes, and can thus be solved efficiently. Some difficult problems may be transformed into simplified versions which are not too distant from the original problems. Such simplifications can then be used to provide either approximate solutions or heuristic advice to guide the search for an exact solution of the original problem.

In general, a problem class is considered *easy* when it admits 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, namely, 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 size of the constraints.

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

We first define the notion of backtrack-free search:

Definition 4.1.1 (backtrack-free search) *A constraint network is backtrack-free relative to a given ordering* $d = (x_1, ..., x_n)$ *if for every* $i \leq n$*, every partial solution of* $(x_1, ..., x_i)$ *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: *Tractability due to restricted structure* which is based solely on the structure of the constraint scopes of the problem, independently of the actual constraint relations. This class will be the focus of the current chapter as well as Chapter 9. *Tractability due to restricted constraint relations*, or constraint languages which identifies classes that are tractable due to special properties of the constraint relations, which 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 therefore start by providing a review of graph theory concepts that will be used in subsequent sections.

4.2 Graph concepts: induced width and hypertrees

One way of characterizing tractable classes is via topological properties of the constraint graph. Topological characterization is centered on the graphical parameter known as *induced width*.

4.2.1 The induced width

Given an undirected graph $G = (V, E)$, an *ordered graph* is a pair (G, d), where $V = \{v_1, ..., v_n\}$ is the set of nodes, E is a set of arcs over V, and $d = (v_1, ..., v_n)$ 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 4.2.1 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 as proceeding 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 only. 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, while the

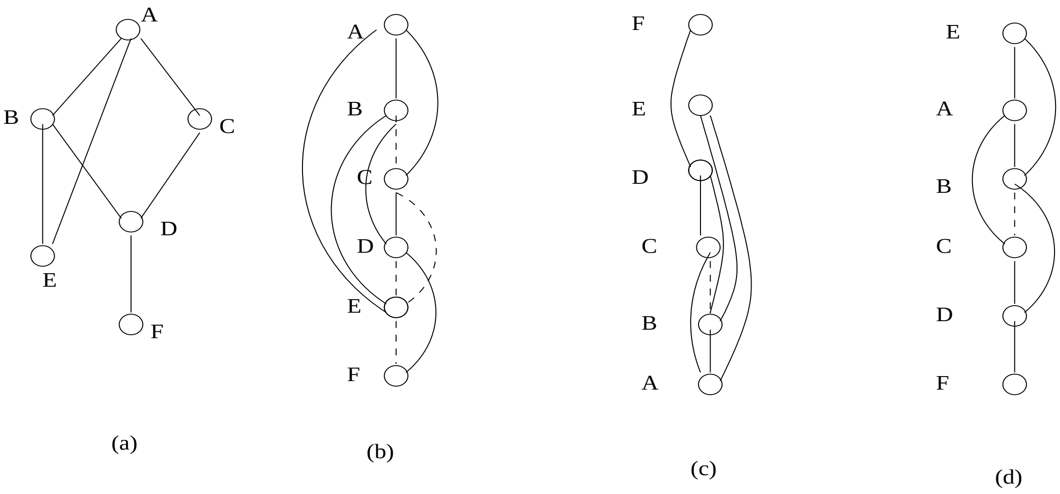


Figure 4.1: (a) Graph G, and three orderings of the graph; (b) $d_1 = (F, E, D, C, B, A)$, (c) $d_2 = (A, B, C, D, E, F)$, and (d) $d_3 = (F, D, C, B, A, E)$. Broken lines indicate edges added in the induced graph of each ordering.

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. □

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 x 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 4.2.2 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 as well. 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, which is 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 an ordering having width of 1, its induced-ordered graph has no additional arcs, yielding an induced width of 1, as well. In summary,

Proposition 4.2.3 *A graph is a tree iff it has width and induced width of 1.* □

Finding a minimum-width ordering of a graph can be accomplished by the greedy algorithm called *min-width* (see Figure 4.2). The algorithm orders variables from last to first as follows: In its first step, a variable with minimum number of neighbors is selected and put last in the ordering. The variable and all its adjacent edges are 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.

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

Figure 4.2: The min-width (MW) ordering procedure

Proposition 4.2.4 *[52] Algorithm min-width* (MW) *finds a minimum width ordering of a graph.*

Proof: See exercises. □

While finding the min-width ordering of a graph is easy, finding the minimum *induced width* of a graph is hard (NP-complete [7]). Nevertheless, deciding whether there exists an ordering whose induced width is less than a constant k, takes $O(n^k)$ time. Approximation algorithms with some good guarantees have been and continue to be developed [2, 147].

A fairly good 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 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 been generated by a min-induced-width ordering of G. In this case, it so happens that the algorithm achieves the minimum induced width of the graph, w^*. The notions of width and induced width and their relationships with various graph parameters have been studied extensively [136]. Here we will focus only on those aspects that appear relevant to constraint processing.

MIN-INDUCED-WIDTH (MIW)
input: a graph $G = (V, E)$, $V = \{v_1, ..., v_n\}$
output: An ordering of the nodes $d = (v_1, ..., v_n)$.
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.
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: $G \leftarrow G - r$.

Figure 4.3: The min-induced-width (MIW) procedure

Chordal graphs

Computing the induced width is easy for chordal graphs. A graph is said to be *chordal* if every cycle of length four has a chord, namely an edge connecting two nonadjacent 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, which is 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* [152]. 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 next, and so on. (See Figure 4.4.)

It can be shown that 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. One can thereby enumerate all maximal cliques associated with each vertex (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 is one such clique. In addition, 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 4.2.5 *If G^* is the induced graph of a graph G, along some ordering, then G^* is chordal.* □

Proof: See exercises.

Example 4.2.6 We see again that G in Figure 4.1(a) is not chordal since in the ordering in Figure 4.1(d) which is a max-cardinality ordering, the parents of A are not connected. Once we connect B and C, the resulting induced graph is chordal. □

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

Figure 4.4: The max-cardinality (MC) ordering ordering procedure

k-trees. A *subclass* of chordal graphs are *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 the vertices in any clique of size k. In particular, one-trees are ordinary trees. *k-trees* were investigated extensively in the graph theoretical literature. In particular, it was shown that a graph can be *embedded* in a k-tree if and only if it has an induced width $w^* = k$ [7].

4.3 Directional consistency

We will now get back to our primary target in this chapter, namely determining the amount of inference that can guarantee a backtrack-free solution.

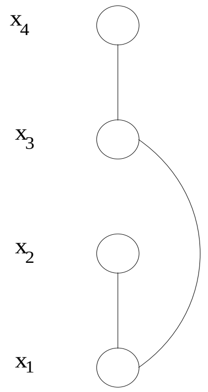


Figure 4.5: An ordered constraint graph

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

Indeed, securing full arc-consistency, full path-consistency, and, in general, full i-consistency is frequently more than is neccessary if a solution is going to be generated by search. Consider, for example, the task of performing a backtrack-search on a problem whose ordered constraint graph is given in Figure 4.5. 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 least one consistent corresponding value in x_2 and x_3, and that, subsequently, any assignment 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, regardless of whether x_2 and x_3 are arc-consistent relative to x_1 or x_4 is arc-consistent relative to x_3. In other words, arc-consistency is required in only one direction, the one which will be exploited by the search algorithm.

In this section, we develop the idea of restricting inference to a given variable ordering, which we call *directional-consistency*, and 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 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 are relevant only to binary constraints, we will initially assume that the networks have binary constraints only, and then generalize to arbitrary constraints.

4.3.1 Directional arc-consistency

Definition 4.3.1 (directional arc-consistency) *A network is* directional-arc-consistent *relative to order $d = (x_1, ..., x_n)$ 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 $d = (x_1, ..., x_n)$ is given in Figure 4.6. 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.

Example 4.3.2 Assume that the constraints and the domains of the problem in Figure 4.5 are specified below.

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

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 4.5. 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 processed next, nothing changes, since 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. Namely, 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 a consistent assignment was made to every variable, and that no dead-end was encountered. □

The advantage of enforcing directional-arc-consistency rather than enforcing full arc-consistency is computational; each constraint is processed exactly once. Indeed,

```
DAC(R)
Input: A network R = (X, D, C), its constraint graph G, and an ordering d = (x_1, ...., x_n).
Output: A directional arc-consistent network.
    for i = n to 1 by −1 do
        for each j < i s.t. R_ji ∈ R,
            D_j ← D_j ∩ π_j(R_ji ⋈ D_i), (this is revise((x_j), x_i)).
    end-for
```

Figure 4.6: Directional arc-consistency (DAC)

Proposition 4.3.3 *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: Due to the order of processing, 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 $O(ek^2)$. $\square$

Algorithm DAC seems optimal for achieving directional-arc-consistency: just to *verify* directional-arc-consistency, each constraint should be inspected once, requiring $O(k^2)$ consistency tests in the worst case.

Example 4.3.4 Let us now examine another constraint network, one whose ordered graph is depicted in Figure 4.7. This time the constraints will all be *inequality* constraints ($R_{ij} : x_i \neq x_j, i \neq j$), and the 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 this (and any) ordering. Therefore, applying DAC will not cause any changes to the domains of the variables.

Does this guarantee a backtrack-free search for a consistent solution? We see that it is possible to make the consistent partial assignment $(\langle x_1, red\rangle, \langle x_2, blue\rangle, \langle x_3, red\rangle)$. However, there is no assignment to x_4 that satisfies both $R_{24} : x_2 \neq x_4$ and $R_{34} : x_3 \neq x_4$, and therefore a dead-end is encountered. Thus we see that, like full-arc-consistency, DAC is not sufficient to guarantee that every problem will be backtrack-free; higher levels of consistency may therefore be necessary. $\square$

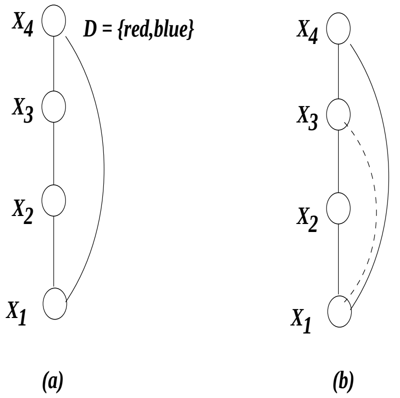


Figure 4.7: An ordered constraint graph (a) and its induced graph (b).

4.3.2 Directional path-consistency

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

Definition 4.3.5 (directional path-consistency) *A network $\mathcal{R}$ is* directional path-consistent *relative to order $d = (x_1, ..., x_n)$ iff for every $k \geq i, j$, the pair $\{x_i, x_j\}$ is path-consistent relative to x_k.*

Example 4.3.6 Consider a 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 4.7(a). This network is already arc-consistent; however, it is not path-consistent. For example, the (universal) constraint between 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, if we enforce only directional-path-consistency relative only to d, then only the constraint $x_1 = x_3$ needs to be added in order that a solution can be assigned along order d without encountering any dead-ends. □

As implied by the above example, when the network is not directional-path-consistent, the corresponding consistency level can be enforced. Algorithm DPC (Figure 4.8) achieves strong directional-path-consistency (namely, both directional arc-consistency and directional path-consistency). A new feature of this algorithm is its explicit reference to the underlying constraint graph. The algorithm not only manages the changes made to the constraints but also the changes made to the constraint graph, namely, the addition of new arcs that correspond to adding new constraints.

DPC($\mathcal{R}$)
Input:A binary network $\mathcal{R} = (X, D, C)$ and its constraint graph $G = (V, E)$, $d = (x_1, ..., x_n)$.
Output:A 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) $\forall\ i, j \leq k$ s.t. (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 4.8: Directional path-consistency (DPC)

The 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.

Theorem 4.3.7 *Given a binary network $\mathcal{R}$ and an ordering d, algorithm DPC generates an 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.*

Proof: All we need to show is that 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 3), 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 reduced later or if the two binary constraints R_{ik} and R_{jk} are changed. However, by the time x_k is processed, these two constraints 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 the algorithm's 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 $O(k^3)$ yielding an overall compexity of $O(n^3k^3)$. Since the computation of steps 2 and 3 is completely dominated by the computation of steps 4,5,6, we get that the overall complexity is $O(n^3k^3)$. $\square$

4.3.3 Directional i-consistency

In the previous two subsections we restricted our attention to binary networks because the constraint 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 restrict operation of these two algorithms to their binary subnetworks only. Algorithm DAC still yields a directional arc-consistent network, and DPC yields a directional path-consistent network.

Definition 4.3.8 (directional i-consistent) *A network is* directional i-consistent *relative to order $d = (x_1, ..., x_n)$ iff every $i-1$ variables are i-consistent relative to every variable that succeeds all of them all in the ordering. A network is strong directional i-consistent if it is directional j-consistent for every $j < i$.*

When extending directional-consistency to i-consistency, we consider all 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 algorithm is given in Figure 4.9. Note, that we use a generic Revise procedure that operates on any size set as its first parameter. If the parent set of a variable has less than $i-1$ variables the procedure records a single constraint over the parent set (steps 2 and 3). Otherwise, we record a constraint over every subset of size $i-1$ of the parent set (step 5). Algorithm DIC_i enforces strong directional-i-consistency (see exercises). Its complexity will be addressed later using the induced-width.

Example 4.3.9 Applying DIC_3 to the network $\mathcal{R} = \{R_{xyz}\}$ in Example 3.4.4, along ordering $d = (x, y, z)$ will add the constraint $R_{xy} = \{(< x, 0 >< y, 0 >)\}$ in addition to the constraint $R_x = \{(< x, 0 >)\}$. $\square$

Note that DIC_3 and DPC are identical if the input network is binary, while otherwise they are not because of different handling of ternary constraints.

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** $|P| < i - 1$ 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.
8. $E' \leftarrow E' \cup \{(x_k, x_m) | x_k, x_m \in P\}$ (connect all parents of x_j)
9. **endfor**.
10. **return** C' and E'.

Figure 4.9: Algorithm directional i-consistency (DIC_i)

4.3.4 Graph aspects of directional consistency

Neither directional arc-consistency nor full arc-consistency change the constraint graph. Higher levels of directional-consistency do, although to a lesser extent than their nondirectional counterparts. For example, applying DPC to the network whose ordered graph is given in Figure 4.7(a) results in a network having the graph in Figure 4.7(b), where arc (x_1, x_3) is added. If we apply DPC to the problem in Figure 4.5, no constraint will be added, and the changes are only those caused by arc-consistency. Full path-consistency, on the other hand, will make the constraint graph of Figure 4.5 complete. The reader should refresh its memory on graph concepts defined in Section 4.2

Specifically, during processing by DPC, the only time a variable x_k affects the constraint between a pair of earlier variables is when it is constrained with both via binary constraints[1] and is thus connected to both in the graph. In only 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 4.7(b) is the induced graph of the ordered graph in Figure 4.7(a).

Proposition 4.3.10 *Let (G, d) be the ordered constraint graph of a binary network $\mathcal{R}$. If*

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

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 claim by induction on the variables along the reverse ordering of $d = (x_1, ..., x_n)$. The induction hypothesis is that all the arcs incident to $x_n, ..., x_i$ in G_1 appear also in (G^*, d). The claim is true for x_n (the induction base step), since its connectivity is the same in both graphs. Assume that the claim is true for $x_n, ..., x_i$, and we will show that it holds also for $i-1$, namely, 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 (namely, 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 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, by 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) (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 for each variable, DPC processes all and 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,

Theorem 4.3.11 *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: Since, from Proposition 4.3.10, when a variable x is being processed by DPC it is connected to at most $w^*(d)$ parents, the number of triplets 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. □

Consequently, orderings having a small induced width allow a more efficient performance of DPC. In particular, instead of 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 similarly bounded using the induced width. 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 4.3.10 and Theorem 4.3.11 to the general i-consistency case.

Proposition 4.3.12 *Given a network $\mathcal{R}$ whose constraint arity are 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). (for a proof see exercise)* $\square$

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,

Theorem 4.3.13 *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 $O(n(w^*(d))^i \cdot (2k)^i)$.* $\square$

4.4 Width vs. local consistency

We saw that DPC changed the network in Example 4.3.6 so that a solution could be found in a backtrack-free manner. However, it will be easy to create an example for which 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*, for a given constraint network, what level of consistency is sufficient for generating a backtrack-free representation. Fortunately, such a characteristic can be provided, based on the induced width of the network's graph. We will start with the special case of width 1.

4.4.1 Solving trees: case of width-1

We saw that while directional arc-consistency in the example of graph 4.5 generated a backtrack-free network, it did not generate a backtrack-free network when we added the constraint between x_2 and x_4. Notice also that the graph of the first problem 4.5 did not have cycles, however, once we added the constraint, it did. Indeed these examples generalize. Any tree-structured binary network which is arc-consistent is backtrack-free for a variety of orderings. Moreover, if a tree-network is not arc-consistent, arc-consistency can be enforced. We will express this relationship between graphs and local consistency using the concept of a width. Remember, as shown in Section 4.2, a graph is a tree iff it has a width-1 ordering.

Theorem 4.4.1 *If a binary constraint network has a width of 1 and if it is arc-consistent, then it is backtrack-free along any width-1 ordering.*

However, as observed before, full arc-consistency is more than is necessary for achieving backtrack-free solutions on trees. For example, if the constraint network in Figure 4.10

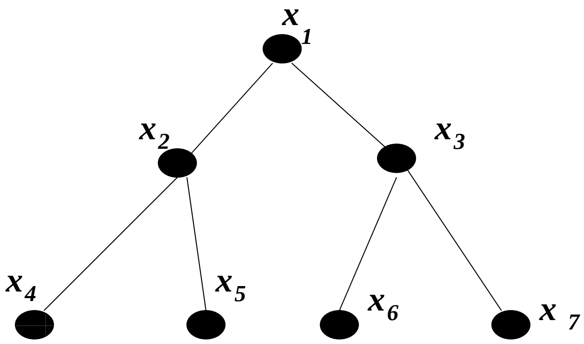


Figure 4.10: A tree network

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. This can be achieved by making the directed arc (x_1, x_2) arc-consistent, regardless of whether (x_2, x_1) is arc-consistent. Notice that the tree in Figure 4.10, along $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ has a width 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 that all we need is *directional arc-consistency.*

Theorem 4.4.2 (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.* $\Box$

Proof: Consider a width-1 ordering $d = (x_1, ..., x_n)$. Let's assume that a subset of variables $x_1, ..., x_i$ were instantiated consistently and that we need to instantiate x_{i+1}. Since d is a width-1 ordering, there is only one parent variable x_j with which x_{i+1} may be constrained. 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. $\Box$

Thus if we have a width-1 ordering of a binary constraint network, we can apply algorithm DAC along that ordering, enforcing directional arc-consistency, and then find a solution backtrack-freely. The tree-solving algorithm in Figure 4.11 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 4.4.2) the tree-solving algorithm is complete for trees and its complexity is $O(nk^2)$, the complexity of DAC.

Interestingly, 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

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, $d = x_1, \ldots, x_n$.
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 4.11: Tree-solving algorithm

$O(nk^2)$ steps. In contrast, if algorithm AC-3 had been applied to the tree, its worst case performance is $O(nk^3)$. If algorithm AC-4 is applied to trees, it has complexity of $O(nk^2)$, but at the cost of a much more involved implementation.

4.4.2 Solving width-2 problems

Can we also make a width-2 network backtrack-free? To some extent we can. The relationship observed in Theorem 4.4.2 can be extended:

Theorem 4.4.3 (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 consistent with all previously chosen values. Suppose that x_1,x_2, ..., 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 $i, j \leq k$. 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 with only one previous variable, directional arc-consistency ensures the existence of a consistent extension to x_k as well. □

If a problem has a width-2 ordering but it is not directional path-consistent, then we may want to enforce directional path-consistency by DPC. However, as was shown in Section 4.3.2, applying DPC may add arcs to the problem's graph and increase its width, now its induced width $w^*(d)$, to something greater than 2. Therefore, applying DPC to width-2 problems does *not* guarantee a backtrack-free solution, unless $w^*(d) = 2$,. A ring

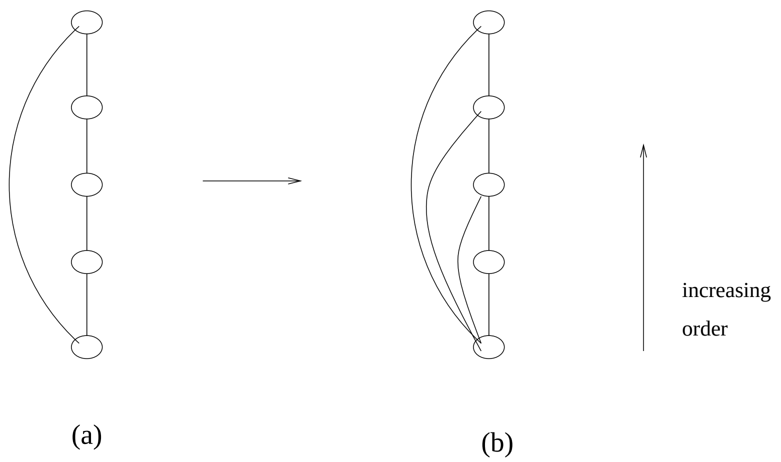


Figure 4.12: An induced width-2 graph

constitutes an example of a problem whose width and induced-width is 2. Figure 4.12 shows an ordered ring and its induced graph. Both graphs have a width of 2.

Theorem 4.4.4 *A binary constraint network $\mathcal{R}$ having induced width of 2 can be solved in linear time of $O(nk^3)$ by DPC.*

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

Is it easy to recognize if a graph has an ordering with induced width of 2? Clearly, enumerating all orderings may be hard. Luckily, there is a linear time algorithm for recognizing such graphs. The algorithm is the min-induced-width (MIW) algorithm described in Section 4.2. To remind, it orders the nodes in decreasing order. It selects a node with smallest degree, place it last in the ordering, connects its neighbors, removes it from graph, and continues, recursively. If at anytime a selected node's degree is larger than 2, the graph must have induced width higher than 2 (see exercises).

4.4.3 Adaptive-consistency: solving width-i problems

The relationship between the width of the graph and the tractability of the problem can be extended to general non-binary networks as follows. A problem is *backtrack-free* along d if the *level* of its directional strong consistency along this order is greater then 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 tests the consistency of the relevant constraints among past and current variables only. 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. Formally,

Theorem 4.4.5 (Width (i-1) and directional i-consistency) *Given a general network $\mathcal{R}$. If its ordered constraint along d has a width of $i-1$ and if it is also strong directional i-consistent, then it is backtrack-free along d.*

Proof: Assume that the problem has a width $i-1$ along d and that it is strong i-consistent. Then, given any partial solution $\bar{a} = (\langle x_1, a_1\rangle, ..., \langle x_{i-1}, a_{l-1}\rangle)$ and given the next variable x_l, because the width is $i-1$, x_l is connected to a subset of variables S_i, $S_i \subseteq \{x_1, ..., x_l\}$, 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. $\Box$.

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 can be applied.

However, as in the case of width-2, algorithm DIC_i augments the network with additional constraints (either binary or non-binary ones), yielding a denser constraint graph. The resulting graph is identical (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 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.

Adaptive consistency. Algorithm *adaptive-consistency (AC)* in Figure 4.13 implements this idea in a uniform way. Given an ordering d, it establishes *directional i-consistency* recursively, whose level changes from node to node to *adapt* to its *width* at the time of processing. This is feasible 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 subsets of variables as well as tighten existing constraints. When algorithm adaptive-consistency terminates, it yields a backtrack-free constraint network.

ADAPTIVE-CONSISTENCY (AC)
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 $S \leftarrow parents(x_j)$, and let i be the size of S.
3. $R_S \leftarrow Revise(S, x_j)$ (generate all partial solutions over S that can extend 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)
5. **endfor**.

Figure 4.13: Algorithm adaptive-consistency

Example 4.4.6 Consider a problem having the ordered graph in Figure 4.14a along $d = (x_1, x_2, x_3, x_4, x_5)$. Assume that the constraints scope are:

$$\{x_3, x_4, x_5\}, \{x_2, x_5\}\{x_4, x_2\}, \{x_3, x_1\}, \{x_2, x_1\}.$$

Adaptive-consistency proceeds from x_5 to x_1 and imposes constraints on the parents of each processed variable. Variable x_5 is chosen first and because it has 3 parents, x_2, x_3, x_4, the algorithm records a ternary constraints R'_{234} on these variables (using constraints R_{345} and R_{52}). Variable x_4 is processed next and, having width two, the algorithm solves the subproblem containing the constraint R_{42}, and the new generated constraints R'_{234} and projects the set of all solutions on x_3 and x_2. A constraint R'_{23} and an arc (x_2, x_3) are added. When the algorithm reaches node x_3, its width is two and, therefore 3-consistency is enforced on x_3's parents $\{x_2, x_1\}$. The arc (x_1, x_2) already exists so this operation may merely tighten the original constraint, R_{12}. The resulting graph is given in Figure 4.14b. $\square$

4.5 Bucket-elimination

We now give an alternative description of adaptive-consistency using a data structure, called *buckets*, which provides a convenient way of organizing the algorithm's steps, 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, and given an ordering, to

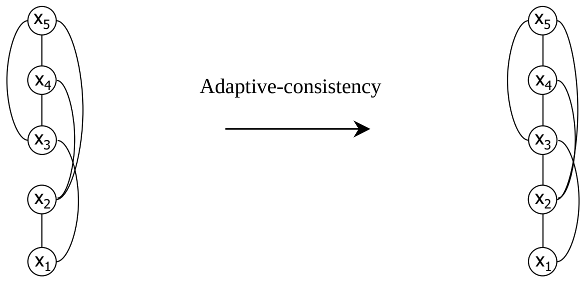


Figure 4.14: A constraint graph before (a) and after (b) *adaptive consistency.*

place each constraint into the bucket of the variable that appears latest in its scope. Doing so, we will have collected together in the various buckets all the constraints that have the same latest variable in their scope. Then, the Revise operation merely solves the subproblem defined by each bucket.

The *bucket-elimination* algorithm AC in Figure 4.15 does just this. In its first step, AC partitions all the constraints into ordered buckets, where the bucket of x_i contains all the constraints having x_i as the latest variable in their scope. Then, buckets are processed in reverse order. Processing a bucket means solving a subproblem and recording its solutions as a new constraint, after the bucket's variable is projected out. The newly generated constraint is placed in the bucket of *its* latest variable. The reader should verify that the two descriptions of adaptive-consistency coincide. It follows from earlier discussion that,

Proposition 4.5.1 *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 4.5.2 *Given a network $\mathcal{R}$, adaptive-consistency decides 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 4.5.1 it follows that AC generates a width-i problem that is $(i+1)$-consistent, for some i, and thus, from Theorem 4.4.5, it follows that the resulting network $E_d(\mathcal{R})$ is backtrack-free along the order of processing. $\Box$

Example 4.5.3 Consider the graph coloring problem depicted in Figure 4.16 (the colors are denoted by numbers). Figure 4.17 shows a schematic execution of adaptive-consistency using the bucket data structure for the two orderings $d_1 = (E, B, C, D, A)$

ADAPTIVE-CONSISTENCY (AC)

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

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.
for $p \leftarrow n$ **downto** 1 **do**
  for all the constraints $R_{S_1}, \ldots, R_{S_j}$ in $bucket_p$ **do**
    $A \leftarrow \bigcup_{i=1}^{j} S_i - \{x_p\}$
    $R_A \leftarrow \Pi_A(\bowtie_{i=1}^{j} R_{S_i})$
    if R_A is not the empty relation **then** add R_A to the bucket of the latest variable in scope A,
    else exit and return the empty network
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

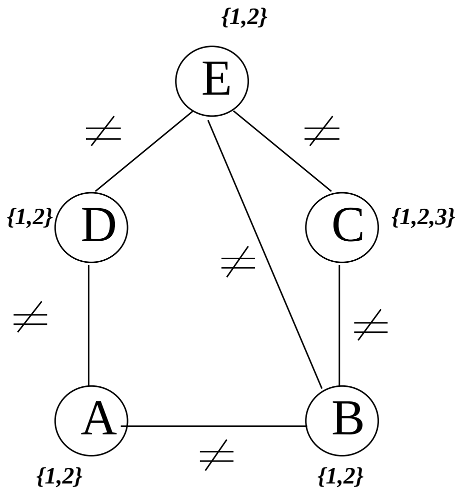


Figure 4.16: A modified graph coloring problem

$d_2 = (A, B, C, D, E)$. The constraints generated by the algorithm are displayed to the right of the double bar in the buckets into which they are placed. While along order d_1 only binary constraints are created, along ordering d_2, potentially ternary constraints are generated.

Notice that for the ordering d_1 the constraint $B \neq E$, generated in the bucket of D, is displayed in the bucket of B (in parentheses) for illustration only, since there is already an identical original constraint. Also, the constraint R_{BE} recorded when processing bucket C is the universal constraint and therefore should not be recorded at all. We chose to display the constraint 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.

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 new added arcs. The induced width along d_1 and d_2 are 2 and 3 respectively, suggesting different complexity bounds for adaptive-consistency. □

From this point forward, when referring to algorithm adaptive-consistency (AC), we mean its bucket description. Algorithm AC 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 and the number of constraints appearing in the bucket. The number of variables in a bucket is bounded by the number of parents of the corresponding variable in the induced ordered graph.

Theorem 4.5.4 *The time and space complexity of Adaptive-consistency is $O(n \cdot (2k)^{w*+1})$ and $O(n \cdot k^{w*})$ repectively where, n is the number of variables, k bounds the domain size and $w*$ is the induced-width along the order of processing.*

Proof: The number of relations in each bucket will increase to at most $2^{w^*(d)+1}$ relations. Since solving a subproblem on $w^* + 1$ variables is $O((2k)^{w^*+1})$, the overall complexity is bounded by $O(n \cdot (2k)^{w^*(d)+1})$. □

As a result, problems having bounded induced width ($w^* \leq b$) for some constant b, can be solved in polynomial time. In particular, AC when applied to trees, coincides with DAC as demonstrated in Figure 4.19. The Figure depicts a constraint graph that has no cycles. When the graph is ordered along $d = (A, B, C, D, E, F, G)$, its width and induced width equal 1. Indeed, as is demonstrated by the schematic execution along d, AC generates only unary relationships and is therefore very efficient.

To summarize, the analyses in this chapter yields a class of tractable problems based on the induced-width of the constraint graph.

For ordering d_1, the initial partitioning is:
$bucket(A)$: $A \neq D$, $A \neq B$
$bucket(D)$: $D \neq E$
$bucket(C)$: $C \neq B$, $C \neq E$
$bucket(B)$: $B \neq E$
$bucket(E)$:

After processing:
$bucket(A)$: $A \neq D$, $A \neq B$
$bucket(D)$: $D \neq E$, $||$ $D = B$
$bucket(C)$: $C \neq B$, $C \neq E$
$bucket(B)$: $B \neq E$ $||$ $(B \neq E)$, (R_{BE})
$bucket(E)$: $||$ R_E

For ordering d_2, the initial partitioning is:
$bucket(E)$: $E \neq D$, $E \neq B$, $E \neq C$
$bucket(D)$: $D \neq A$
$bucket(C)$: $C \neq B$
$bucket(B)$: $B \neq A$
$bucket(A)$:

After processing we get:
$bucket(E)$: $E \neq D$, $E \neq B$, $E \neq C$
$bucket(D)$: $D \neq A$, $||$ R_{DCB}
$bucket(C)$: $C \neq B$ $||$ R_{ACB}
$bucket(B)$: $B \neq A$ $||$ R_{AB}
$bucket(A)$: $||$ R_A

Figure 4.17: Execution along the orderings $d_1 = E, B, C, D, A$ and $d_2 = A, B, C, D, E$.

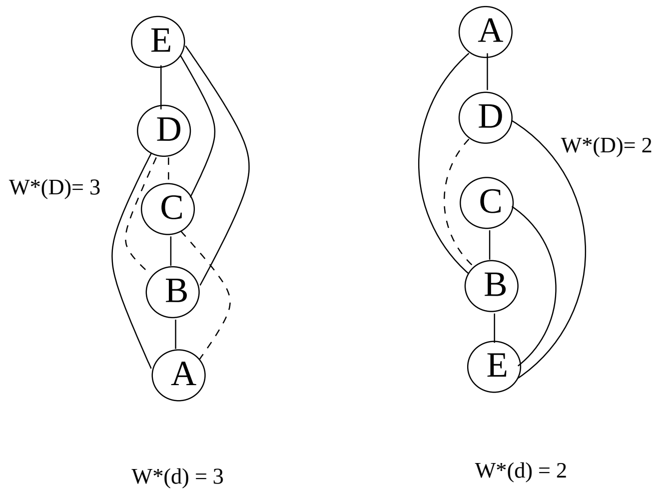


Figure 4.18: The induced width along the orderings: $d_1 = A, B, C, D, E$ and $d_2 = E, B, C, D, A$

Theorem 4.5.5 (w*-based tractability) *Constraint problems whose induced-width is bounded by t are polynomially solvable.*

Although finding the induced-width of a graph is hard, deciding if the width is less than a constant t can be done in polynomial time [7]. Consequently, Theorem 4.5.4 yields a legitimate tractable class.

Adaptive-consistency should not be evaluated merely as a procedure for deciding consistency, but also as a compilation algorithm. It transform the constraint problem into an equivalent network, from which *every* solution can be assembled in linear time. Therefore, when the output network can be compacted substantially, AC 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.

4.6 Summary

The chapter introduced the notion of bounded directional consistency algorithms, such as directional arc-, path-, and i-consistency. These are incomplete inference algorithm that can sometime decide inconsistency and which are designed as preprocessing algorithms to be used prior to backtracking search. As we will show in Chapter 5, such algorithms can also be interleaved with search. The chapter also presented the relationship between

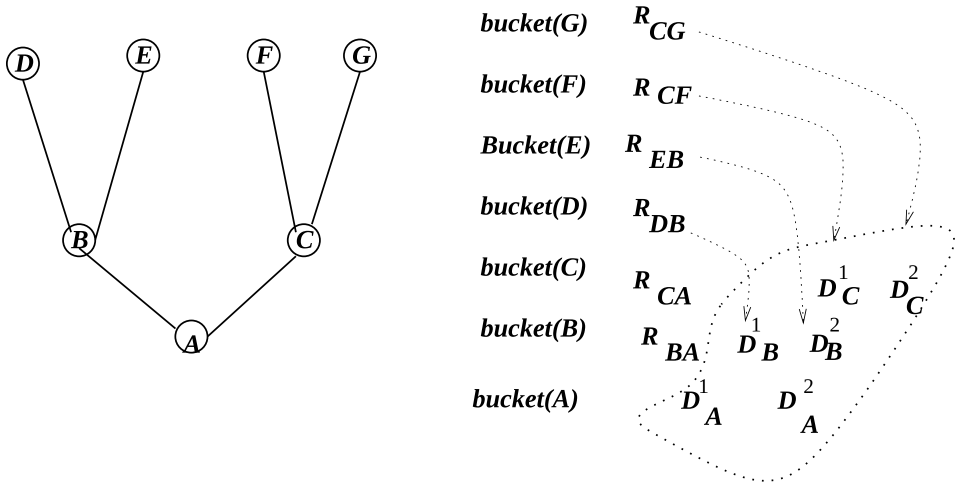


Figure 4.19: Schematic execution of adaptive-consistency on a tree network. D_X denotes unary constraints over X.

induced width and consistency level that guarantee 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 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 t can be done in polynomial time (exponential in t [7]), 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 and quadratically, respectively. Finally, a complete inference algorithm adaptive-consistency is shown to make any problem backtrack-free relative to a given variable-ordering.

The chapter also devotes a background section on graph-concepts such as induced-graphs and the parameters width and induced-width. Related graph-concepts (e.g., tree-width) will be presented in Chapter 9. Chapter 8 extends algorithms appearing in this section to the more general consistency notions of relational-consistency, and to bucket-elimination algorithms.

4.7 Chapter notes

The fundamental relationship between width and consistency level that guarantee a backtrack-free solution was introduced by Freuder [52]. This relationship was extended by Dechter [43] to the more restricted concept of directional consistency. It also lead to algorithm Adaptive-consistency and to the identification of the induced-width as the principle graph-parameter that controls its complexity. A similar elimination algorithm was introduced earlier by Zeidel [168]. It was observed that these algorithms belong to the class of Dynamic programming algorithms as presented in [14]. In [44], the connection between Adaptive-consistency and tree-clustering algorithms was made explicit [44] 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, hypert-trees and join-trees was observed independently in the areas of relational database [107], Dynamic programming [14] and graph-theory [7, 136, 6].

In their book on non-serial dynamic programming [14] Bertele and Briochi show the dependence of variable-elimination algorithms for solving optimization tasks to a graph parameter which is what is called here *induced-width* They suggest several greedy heuristics including the Min-induced-width (not under this name) heuristics and show that it is optimal for graphs having induced-width of 2. Montanari [115] also observes that series-parallel binary constraint networks are tractable, a class which is identical to networks having induced-width of 2. In Relational database the concept of join-tree was defined and observed as a desired representation for relational databases. The connection between join-trees and chordal graphs was identified [13] and the max-cardinality order was shown to be an identifier of chordal graphs by [152].

Finally an extensive analysis of related concepts defined over hypre-trees and the notion of tree-width, its connection with a variety of graph properties were fully analyzed in several papers by Arnborg and his colleagues [7, 136, 6]. In particular Arnborg proved that finding the tree-width of an arbitrary graph is NP-complete.

4.8 Exercises

1. (*) Show that it takes $O(n^k)$ time to decide whether a graph G has an ordering whose induced width is less than a constant k.

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 4.11, is optimal.

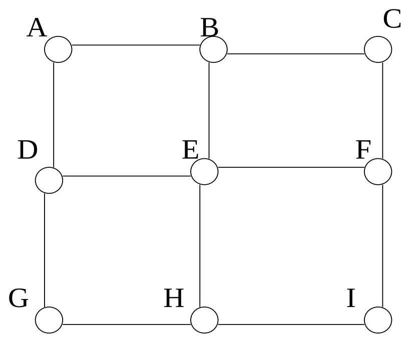


Figure 4.20: Grid with nine nodes

6. Generate a directional-path-consistent 4-queen 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 constraints (e.g., inequalities). Provide a complexity bound using the induced width for applying algorithm DPC along the optimal induced-width ordering of this problem.

   (c) Can algorithm DPC always decide consistency of every constraint problem having the grid constraint graph?

8. Prove Proposition 4.3.12.

9. Prove that algorithm DIC_i generates a 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 modified graphs coloring problem given in Figure 4.21. The constraints are inequation constraints, and the domains are indicated inside the nodes in the graph.

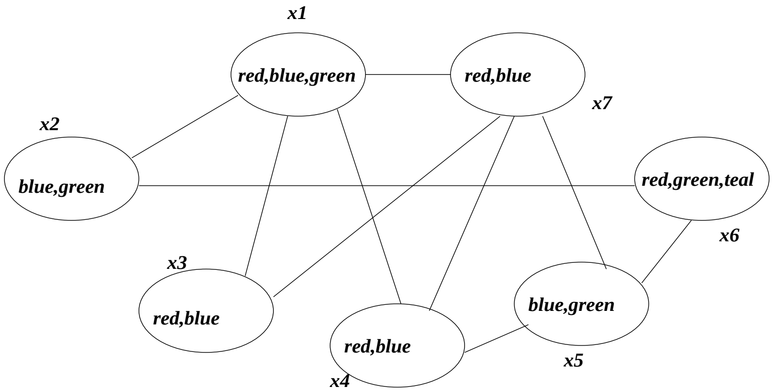


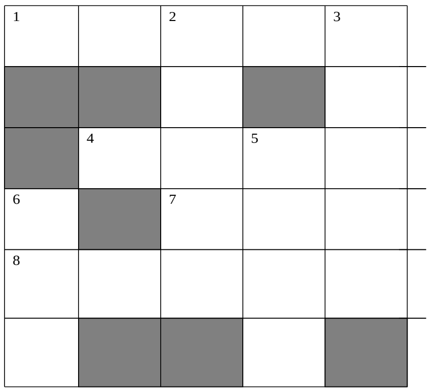
Figure 4.21: A modified coloring problem.

(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 again the crossword puzzle:



Word List

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

(a) Model the problem as a binary CSP, namely the words are the variables (the problem has 8 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.

(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 which will be recorded for this specific problem by 1. directional arc-consistency, 2. directional path-consistency 3. adaptive consistency (Show the constraints subsets generated for all buckets, and the actual constraints generated when processing the first three buckets).

13. Consider the crypt-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 algorithm. What is the width of the ordering you generated? Compute the induce graph and the induced width of the ordering generated.

    (c) Hand-simulate algorithm adaptive-consistency on the ordering you created 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 to this problem.

14. Prove the ordering MIW can decide if a graph has an induced-width 2.