Chapter 3

Consistency-enforcing algorithms: constraint propagation

3.1 Introduction

This chapter deals with the processing and solving of constraint systems by inference. Inference, as it is applied to constraints, narrows the search space of possible partial solutions by creating equivalent, yet more explicit, networks. Consider the constraint network having three variables x, y, z, all with domains $D = \{0, 1\}$, and three constraints: (1) $x = y$, (2) $y = z$, and (3) $x \neq z$. From the first two constraints, we can infer that $x = z$. Since this inferred constraint is inconsistent with the constraint $x \neq z$, we can conclude that the set of three constraints is inconsistent.

Alternatively, consider the set of just two constraints $\mathcal{R} = \{x = y, y = z\}$. From $\mathcal{R}$ we can infer $x = z$, which we can add to the set of constraints, resulting in the new network $\mathcal{R}' = \{x = y, y = z, x = z\}$. The two networks $\mathcal{R}$ and $\mathcal{R}'$ are equivalent, meaning that they have the same set of solutions, but $\mathcal{R}'$ is more explicit than $\mathcal{R}$, including a relationship not made explicit in $\mathcal{R}$.

Suppose now that we want to generate a solution for these networks by assigning values to the variables along the ordering x, z, y. Using $\mathcal{R}$, we can generate the partial solution $(x = 1, z = 0)$ and will only then realize that there is not a consistent extension to y. In other words, the two values in the domain of the remaining variable, y, are each inconsistent with at least one value assignment in the partial solution. Therefore, we have to backtrack and to try another assignment for z. This time, only $z = 1$ remains, which yields a full solution.

Assume now that we assign values along the same ordering while consulting network $\mathcal{R}'$. Since the constraint $x = z$ is explicit in $\mathcal{R}'$, it will not allow us to assign a value to z that differs from the value assigned to x. Therefore, the partial assignment above is not a partial solution of $\mathcal{R}'$, and we will avoid the dead-end we encountered with $\mathcal{R}$.

In general, as we have more and more explicit and tight (but equivalent) constraint networks, the search space of all possible partial solutions becomes more restricted, and consequently, any search will become more efficient. At the extreme, the problem may become explicit enough (by inferring more and more constraints or by tightening existing ones) so that the search will be completely directed to a solution, and no dead-end will be encountered.

Indeed, constraint inference can be used to find a complete solution. It can deduce constraints from the given initial set until either an inconsistency is encountered, or until an equivalent constraint network is created from which every solution can be derived by a depth-first search that encounters no dead-ends.

However, completely solving the problem by inference is frequently too hard, requiring the addition of an exponential number of new constraints. A more conservative approach is to infer only a restricted set of new constraints, yielding a search space which is reduced, but is not necessarily backtrack-free, and which will allow finding a solution, or concluding inconsistency, relatively quickly. Algorithms that perform a bounded amount of constraint inference are informally called either *local consistency enforcing*, *bounded inference*, or *constraint propagation* algorithms.

Since consistency enforcing algorithms were introduced to assist search, they are defined in terms of extending a solution of a fixed length $i-1$, by one more variable. Intuitively, a consistency-enforcing algorithm will make any partial solution of a small subnetwork extendible to some surrounding network. The primary characteristic by which consistency inference algorithms are distinguished is the *size* of the subnetwork involved in the inference. This "size" can be defined by the *number of variables* or by the *number of constraints* involved. If we infer constraints based on pairs of variables, it is called *arc-consistency*. Arc-consistency ensures that any legal value in the domain of a single variable has a legal match in the domain of any other selected variable. If we infer constraints based on subnetworks of size three, it is called *path-consistency*. Path-consistency ensures that any consistent solution to a two-variable subnetwork is extendible to any third variable. And, in general, if we infer constraints based on subnetworks having i variables, it is called *i-consistency*. Algorithms that enforce i-consistency guarantee that any consistent instantiation of $i-1$ variables is extendible to any i^{th} variable. Finally, if a network is i-consistent for all i, it is *globally consistent*.

Deciding the level of consistency that should be enforced on the network before search is not a clear-cut choice. Generally speaking, any search algorithm will benefit from representations that have a high level of consistency. However, as we will show, the complexity of enforcing i-consistency is exponential in i, as this is the time and space needed to infer a constraint based on i variables. Thus, there is a trade-off between the effort spent on preprocessing by i-consistency and that spent on subsequent search.

Experimental analyses which enforce local consistency before and during search focus on this trade-off.

This chapter as well as Chapter 8 cover the basics of local consistency methods. Here we present consistency enforcing methods whose level is determined by the number of variables involved. Most of the chapter focuses on arc and path-consistency which are inherently binary-networks concepts. Higher levels of local-consistency that are applicable to non-binary constraints, are discussed towards the end. In Chapter 8, we focus on local consistency definitions and methods based on the number of constraints involved, and therefore, the entire exposition is applied to the general, non-binary, framework.

We first review the definition of consistency of a partial solution, which is central to this section.

Definition 3.1.1 (partial solution) *Given a constraint network $\mathcal{R}$, an assignment of values to a subset of the variables $S = \{x_1, ..., x_j\}$ given by $\bar{a} = (< x_1, a_1 >, < x_2, a_2 >, ..., < x_j, a_j >)$ is consistent relative to $\mathcal{R}$ iff it satisfies every constraint R_{S_i} such that $S_i \subseteq S$. The assignment $\bar{a}$ is also called a partial solution of $\mathcal{R}$. The set of all partial solutions of a subset of variables S is denoted by $\rho(S)$.*

3.2 Arc-consistency

Note that the minimal network has the following *local-consistency* property: any value in the domain of a single variable can be extended consistently by any other variable (follows immediately from Proposition 2.3.16). This property is termed *arc-consistency*, and it can be satisfied by non-minimal networks as well. It can be enforced on any network by an efficient computation that, because of its local and distributed character, is often called *propagation*.

We will demonstrate the notion of *arc-consistency* with the following example. Note that we speak both of a constraint being arc-consistent (or not) relative to a given variable and of a variable being arc-consistent (or not) relative to other variables. In both cases, the underlying meaning is the same.

Example 3.2.1 Consider the variables x and y whose domains are $D_x = D_y = \{1, 2, 3\}$ and the single constraint R_{xy} expressing the relation $x < y$. We can describe the constraint R_{xy} pictorially in a *matching diagram* as in Figure 3.1a, where the domain of each variable is an enclosed set of points, and there are arcs connecting points that correspond to consistent pairs of values. (Do not confuse this type of diagram with the constraint graph of the network.) In this case, the value $3 \in D_x$ has no consistent matching value in D_y. We say, therefore, that the constraint R_{xy} is not *arc-consistent* relative to x. Similarly R_{xy} is not arc-consistent relative to y, since $y = 1$ has no consistent match in x. Using

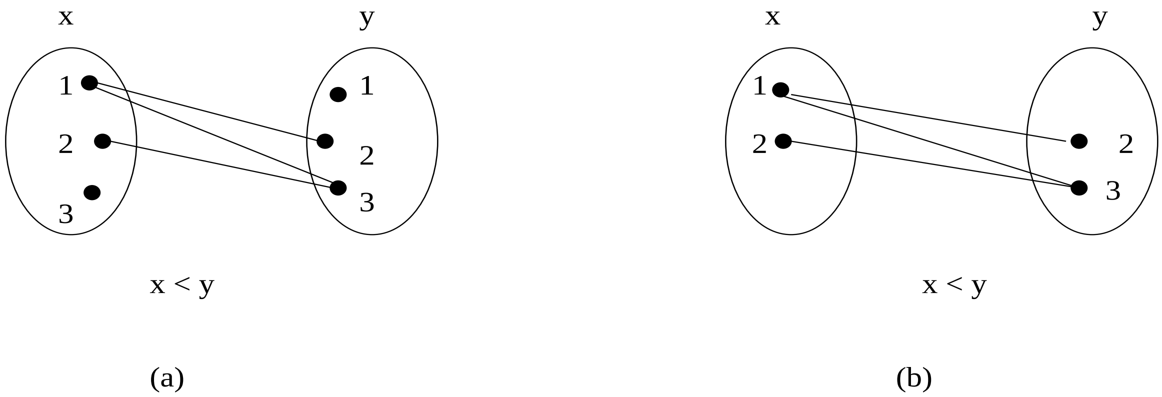

Figure 3.1: A matching diagram describing the arc-consistency of two variables x and y. In (a) the variables are not arc-consistent. In (b) the domains have been reduced, and the variables are now arc-consistent.

matching diagrams, a constraint is not arc-consistent if any of its variables has *lonely* values. Indeed, we see that in the matching diagram of Figure 3.1a both x and y have lonely values.

If, however, we shrink the domains of both x and y such that $D_x = \{1, 2\}$ and $D_y = \{2, 3\}$, then x is arc-consistent relative to y, and y is arc-consistent relative to x. The matching diagram of the arc-consistent constraint network is depicted in Figure 3.1b. If we shrink the domains even further to $D_x = \{1\}$ and $D_y = \{2\}$, we will still have an arc-consistent constraint. However, the latter is no longer equivalent to the original constraint since we may have deleted solutions from the whole set of solutions. □

Definition 3.2.2 (arc-consistency) *Given a constraint network $\mathcal{R} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, with $R_{ij} \in C$, a variable x_i is* arc-consistent *relative to x_j if and only if for every value $a_i \in D_i$ there exists a value $a_j \in D_j$ such that $(a_i, a_j) \in R_{ij}$. The subnetwork (alternatively, the arc) defined by $\{x_i, x_j\}$ is arc-consistent if and only if x_i is arc-consistent relative to x_j and x_j is arc-consistent relative to x_i. A network of constraints is called* arc-consistent *iff all of its arcs (e.g., subnetworks of size 2) are arc-consistent.*

Example 3.2.1 demonstrates that we may make a constraint arc-consistent by shrinking the domains of the variables in its scope. Clearly, if a value does not participate in a solution of a two-variable subnetwork, it will not be part of a complete solution. We wish, however, to eliminate only an amount which ensures that the set of the network's solutions will not be affected. The simple procedure *Revise*$((x_i), x_j)$ in Figure 3.2, if applied to two variables, x_i and x_j, returns the largest domain D_i of x_i for which x_i is arc-consistent

```
REVISE((x_i), x_j)
input: a subnetwork defined by two variables X = {x_i, x_j}, a distinguished variable x_i,
    domains: D_i and D_j, and constraint R_ij
output: x_i arc-consistent relative to x_j
1. for each a_i ∈ D_i
2.      if there is no a_j ∈ D_j such that (a_i, a_j) ∈ R_ij
3.           then  delete a_i from D_i
4.      endif
5. endfor
```

Figure 3.2: The Revise procedure

relative to x_j. It simply tests each value of x_i and eliminates those values having no match in x_j.

Since each value in D_i is compared, in the worst case, with each value in D_j, Revise has the following complexity:

Proposition 3.2.3 *The complexity of Revise is $O(k^2)$, where k bounds the domain size.* □

The Revise procedure can also be described using composition; namely, lines 1, 2, and 3 can be replaced by

$$D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j). \tag{3.1}$$

In this case, D_i stands for the one-column relation over x_i. (Please consult Section 1.3 for the definitions of join and project operators.) Remember that the subscript i is shorthand for variable x_i.

Arc-consistency may be imposed on some pairs of variables, on all pairs from some subset of variables, or over the whole network. Arc-consistency of the whole network may be accomplished by applying the Revise procedure to all pairs of variables. However, applying Revise to each pair of variables just once is sometimes not enough to ensure the arc-consistency of a network.

Example 3.2.4 Consider, for instance, the matching diagram of a three-variable constraint network depicted in Figure 3.3a. Without even knowing the nature of the constraint between y and z, we can see that the two are arc-consistent relative to one another because each value in the domains of the two variables can be matched to an element from the other. However, their arc-consistency is violated in the process of making the adjacent constraints arc-consistent. Specifically, in the process of making $\{x, z\}$ arc-consistent, we

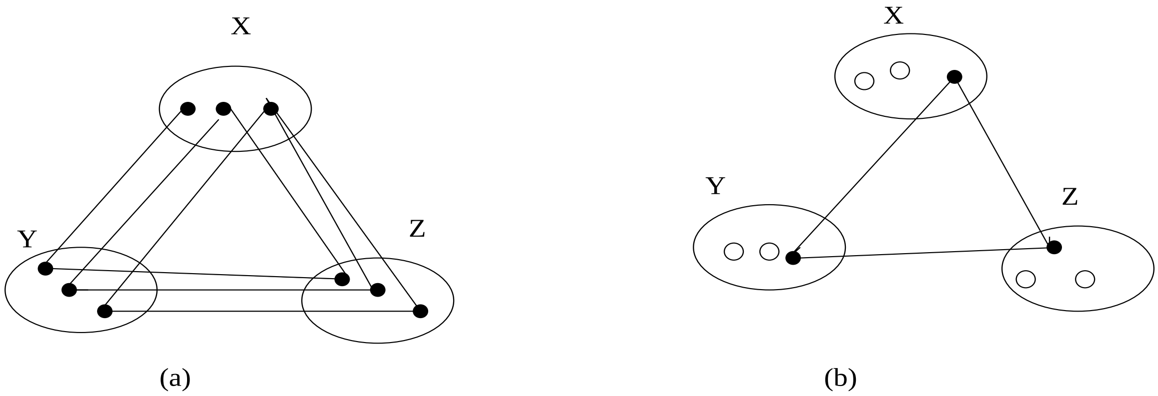

Figure 3.3: (a) Matching diagram describing a network of constraints that is not arc-consistent (b) An arc-consistent equivalent network.

must delete a value from the domain of x, which will make x no longer arc-consistent relative to y. Consequently, Revise may need to be applied more than once to each constraint until there is no change in the domain of any variable in the network. □

Algorithm *Arc-consistency-1 (AC-1)*, a brute-force algorithm that enforces arc-consistency over the network, is given in Figure 3.4. The algorithm applies the Revise rule to all pairs of variables that participate in a constraint until a full cycle ensures that no domain has been altered. The arc-consistent equivalent of the network presented using the matching diagram of Figure 3.3a is given in Figure 3.3b. Occasionally, arc-consistency may discover inconsistency.

AC-1($\mathcal{R}$)
input: a network of constraints $\mathcal{R} = (X, D, C)$
output: $\mathcal{R}'$ which is the loosest arc-consistent network equivalent to $\mathcal{R}$
1. **repeat**
2.   **for** every pair $\{x_i, x_j\}$ that participates in a constraint
3.   Revise($(x_i), x_j$) (or $D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)$)
4.   Revise($(x_j), x_i$) (or $D_j \leftarrow D_j \cap \pi_j(R_{ij} \bowtie D_i)$)
5.   **endfor**
6. **until** no domain is changed

Figure 3.4: Arc-consistency-1 (AC-1)

Example 3.2.5 Consider a binary network over three variables $\{x, y, z\}$ where the domains of all the variables are $\{1, 2, 3\}$, and the constraints are $x < y$, $y < z$, $z < x$. Since AC-1 revises the pairs of variables that participate in each constraint, we can speak of the algorithm as if we are systematically processing each constraint. Indeed, once we have finished processing a pair of variables, we can say both that the variables are arc-consistent relative to one another and that the constraint itself is arc-consistent. Applying arc-consistency to the variables that participate in constraints in the sequence R_{xy}, R_{yz}, R_{zx}, we get first (when revising R_{xy} in both directions) that D_x is reduced to $\{1, 2\}$ and D_y to $\{2, 3\}$. Then, processing constraint R_{yz}, the domain of y is reduced further to $\{2\}$ and the domain of z to $\{3\}$. When R_{zx} is processed, the domain of z will become empty. As a result, subsequent processing will empty the domains of y and z, and we can conclude that the network is inconsistent.

□

Clearly, algorithm *AC-1* generates an equivalent arc-consistent network, and when an empty domain is encountered, the network has no solution. AC-1 has the following complexity:

Proposition 3.2.6 *Given a constraint network $\mathcal{R}$ having n variables, with domain sizes bounded by k, and e binary constraints, the complexity of AC-1 is $O(enk^3)$.*

Proof: One cycle through all the binary constraints (steps 2-5 of AC-1) takes $O(ek^2)$. Since, in the worst case, one cycle may cause the deletion of just one value from one single domain, and since, overall, there are nk values, the maximum number of such cycles is nk, resulting in the overall bound of $O(n \cdot ek^3)$. □

Algorithm AC-1 can be improved. There is no need to process all the constraints if only a few domains were reduced in the previous round. The idea behind the next version of arc-consistency is to maintain a queue of constraints to be processed. Initially, each pair of variables that participates in a constraint is placed in the queue twice (once for each ordering of the pair of variables). Once an ordered pair of variables is processed, it is removed from the queue, and it will be placed back on the queue only if the domain of its second variable is modified due to the processing of adjacent constraints. We name this algorithm *AC-3* to synchronize with the names already established in the community. Another version called *AC-2* provides only a minor advancement in between these two versions. Therefore, algorithm *Arc-consistency-3 (AC-3)*, which uses the queue data structure, is presented in Figure 3.5.

Example 3.2.7 Consider a three-variable network: z, x, y with $D_z = \{2, 5\}$, $D_x = \{2, 5\}$, $D_y = \{2, 4\}$. There are two constraints: R_{zx} specifying that z evenly divides x, and R_{zy} specifying that z evenly divides y. The constraint graph of this problem is depicted in

AC-3($\mathcal{R}$)
input: a network of constraints $\mathcal{R} = (X, D, C)$
output: $\mathcal{R}'$ which is the largest arc-consistent network equivalent to $\mathcal{R}$
for every pair $\{x_i, x_j\}$ that participates in a constraint $R_{ij} \in \mathcal{R}$
    $queue \leftarrow queue \cup \{(x_i, x_j), (x_j, x_i)\}$
endfor
while $queue \neq \{\}$
    select and delete (x_i, x_j) from $queue$
    $Revise((x_i), x_j)$
    if $Revise((x_i), x_j)$ causes a change in D_i
        then $queue \leftarrow queue \cup \{(x_k, x_i), i \neq k\}$
    endif
endwhile

Figure 3.5: Arc-consistency-3 (AC-3)

Figure 3.6a. Assume that we apply AC-2 to the network. We put $(z, x), (x, z), (z, y)$, and (y, z) onto the queue. Processing the pairs (z, x) and (x, z) does not change anything, since the domains of z and x are already arc-consistent relative to R_{zx}. When we process (z, y), we delete 5 from D_z, and consequently (x, z) is placed back on the queue. Processing (y, z) causes no further change, but when (x, z) is revised, 5 will be deleted from D_x. At this point, no further constraints will be inserted to the queue (Remember, no constraint already in the queue need be inserted), and the algorithm terminates with the domains $D_x = \{2\}$, $D_z = \{2\}$, $D_y = \{2, 4\}$. □

Algorithm AC-3 processes each constraint at most $2k$ times, where k bounds the domain size, since each time it is re-introduced into the queue, the domain of a variable in its scope has just been decremented by at least one value, and there are at most $2k$ such values. Since there are e binary constraints and processing each one takes $O(k^2)$, we can conclude that

Proposition 3.2.8 *The time complexity of* AC-3 *is* $O(ek^3)$. □

Is AC-3's performance optimal? It seems that no algorithm can have time complexity below ek^2, since just verifying the arc-consistency of a network requires ek^2 operations in the worst case[1]. Indeed, algorithm Arc-consistency-4 (AC-4) achieves this optimal

[1] For a proof for the special case of tree-networks see [42].

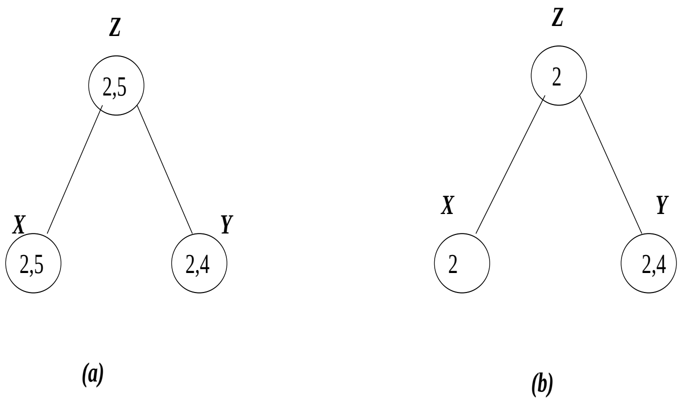

Figure 3.6: A three variable network, with two constraints: z divides x and z divides y (a)before and (b)after AC-2 is applied.

performance. The algorithm no longer uses Revise or the composition operator as an atomic operation. Rather, it exploits the underlying detailed structure of the constraint relation and tunes its operation to that level.

The algorithm associates each value a_i in the domain of x_i with the amount of *support from variable* x_j, that is, the number of values in the domain of x_j that are consistent with value a_i. Then, a value a_i is removed from the domain D_i if it has no support from some other variable. The algorithm maintains a list $LIST$ of currently unsupported variable-value pairs, a counter array $counter(x_i, a_i, x_j)$ of supports to a_i from x_j, and an array $S_{(x_j,a_j)}$ which points to all values in other variables that are supported by $< x_j, a_j >$. In each step , the algorithm picks up an unsupported value from the List, adds it to the *removed* list M, and updates all the supports of values that can be affected. Those values that became unsupported as a result are placed in the $LIST$. If a_j is unsupported, the counters of values that it supports will be decremented. The algorithm is given in Figure 3.7.

Example 3.2.9 Consider the problem in Figure 3.6. Initializing the $S_{(x,a)}$ arrays (indicating all the variable-value pairs that each $< x, a >$ supports), we have :
$S_{(z,2)} = \{< x, 2 >, < y, 2 >, < y, 4 >\}$, $S_{(z,5)} = \{< x, 5 >\}$, $S_{(x,2)} = \{< z, 2 >\}$, $S_{(x,5)} = \{< z, 5 >\}$, $S_{(y,2)} = \{< z, 2 >\}$, $S_{(y,4)} = \{< z, 2 >\}$.
For counters we have: $counter(x, 2, z) = 1$, $counter(x, 5, z) = 1$, $counter(z, 2, x) = 1$, $counter(z, 5, x) = 1$, $counter(z, 2, y) = 2$, $counter(z, 5, y) = 0$, $counter(y, 2, z) = 1$, $counter(y, 4, z) = 1$. (Note that we do not need to add counters between variables that are not directly constrained, such as x and y.) Finally, $List = \{< z, 5 >\}$, $M = \emptyset$. Once $< z, 5 >$ is removed from $List$ and placed in M, the counter of $< x, 5 >$ is updated to $counter(x, 5, z) = 0$, and $< x, 5 >$ is placed in $List$. Then, $< x, 5 >$ is removed from

AC-4($\mathcal{R}$)
input: a network of constraints $\mathcal{R}$
output: $\mathcal{R}'$ an arc-consistent network equivalent to $\mathcal{R}$
Initialization: $M \leftarrow \emptyset$,
    initialize $S_{(x_i,c_i)}$, $counter(i, a_i, j)$ for all R_{ij}
    for all counters
        if $counter(x_i, a_i, x_j) = 0$ (if $< x_i, a_i >$ is unsupported by x_j)
            then add $< x_i, a_i >$ to $LIST$
        endif
    endfor
while $LIST$ is not empty
    choose $< x_i, a_i >$ from LIST, remove it, and add it to M
    for each $< x_j, a_j >$ in $S_{(x_i,a_i)}$
        decrement $counter(x_j, a_j, x_i)$
        if $counter(x_j, a_j, x_i) = 0$
            then add $< x_j, a_j >$ to $LIST$
        endif
    endfor
endwhile

Figure 3.7: Arc-consistency-4 (AC-4)

List and placed in M. Since the only value it supports is $< z, 5 >$ and since $< z, 5 >$ is already in M, the *List* remains empty and the process stops. □

The initialization step that creates the counter of supports and the pointers requires at most $O(ek^2)$ steps. The number of elements in $S_{(xj,a_j)}$ is on the order of ek^2. We conclude,

Proposition 3.2.10 *The time and space complexity of AC-4 is $O(ek^2)$.*

Proof: (exercise 4).

Since worst-case complexity often overestimates, and since average-case analysis is hard to achieve, it is sometime useful to introduce additional, more refined parameters into the analysis. Instead of using $O(k^2)$, which is the size of the universal relation (recall that arc-consistency is relevant only to the binary constraints), as a bound for each relation size, we can use a *tightness* parameter t that stands for the maximum number of tuples in each binary relation. Frequently the constraints can be quite tight, as in the case of functional constraints where $t = O(k)$.

Revisiting our worst-case analysis, the Revise procedure can be modified to have a complexity of $O(t)$. Consequently, the complexity of AC-1 can be modified to $O(n \cdot k \cdot e \cdot t)$, that of AC-3 to $O(e \cdot k \cdot t)$, and that of AC-4 to $O(e \cdot t)$.

Analyzing the best-case performance of these algorithms can also be insightful. The best case of AC-1 and AC-3 is $e \cdot k$ steps, because the problem may already be arc-consistent. The best case of AC-4 is still ek^2, which is the time necessary to create the special data structures in its initialization phase. Consequently, when the constraints are loose (i.e., when t is closest to k^2), AC-1 and AC-3 may frequently outperform AC-4, even though, in the worst case, AC-4 is optimal.

3.3 Path-consistency

We have seen in Example 3.2.5 that arc-consistency can sometime decide inconsistency by discovering an empty domain. However, arc-consistency is not complete for deciding consistency because it makes inferences based on networks of size 2 only.

Example 3.3.1 Consider for instance the three variables x, y, z whose domains are $\{red, blue\}$ and the constraints $x \neq y$, $y \neq z$, $z \neq x$. This constraint network is arc-consistent as is, without reducing any domains, and no domain must be reduced to the empty set to reveal an inconsistency. However, although the constraint R_{xy}, $x \neq y$, allows the assignment $(< x, red >, < y, blue >)$, no color for z will be consistent with both $< x, red >$ and $< y, blue >$. In this case, we say that R_{xy} is not *path-consistent* relative to z, and, more generally, that the subnetwork associated with the three variables $\{x, y, z\}$ is not 3-consistent. There is an inference we can do over this network: since our domain size is 2, $x \neq y$ and $y \neq z$ imply $x = z$, but $x = z$ is not explicit in this network. Not only that, it contradicts another explicit constraint. □

The consistency level that is violated here is called *path-consistency* or 3-consistency, and it requires consistency relative to paths of length 3. It is interesting to note that in the minimal network such a property is maintained; any consistent pair of values can be consistently extended by any third variable (see exercise 6).

In general, the notion of path-consistency involves inferences using subnetworks having three variables. However, since the original definition involves binary constraints only, it is relevant only to the binary constraints of the network. An extension to non-binary constraints is presented later and it constitute only a minor change which may be relevant when the network has ternary constraints as well.

Definition 3.3.2 (Path-consistency) *Given a constraint network $\mathcal{R} = (X, D, C)$, a two variable set $\{x_i, x_j\}$ is path-consistent relative to variable x_k if and only if for every consistent assignment $(< x_i, a_i >, < x_j, a_j >)$ there is a value $a_k \in D_k$ s.t. the assignment $(< x_i, a_i >, < x_k, a_k >)$ is consistent and $(< x_k, a_k >, < x_j, a_j >)$ is consistent. Alternatively, a binary constraint R_{ij} is path-consistent relative to x_k if and only if for every pair $(< x_i, a_i >, < x_j, a_j >) \in R_{ij}$ and a_i and a_j are from their respective domains, there is a value $a_k \in D_k$ s.t. $(a_i, a_k) \in R_{ik}$ and $(a_k, a_j) \in R_{kj}$. A subnetwork over three variables $\{x_i, x_j, x_k\}$ is path-consistent if and only if for any permutation of (i, j, k), R_{ij} is path consistent relative to x_k. A network is* path-consistent *if and only if for every R_{ij} (including universal relations) and for every $k \neq i, j$, R_{ij} is path-consistent relative to x_k.*

Pictorially, a connected pair of points in the matching diagram, (denoting a legal pair of values in R_{xy}) satisfies path-consistency iff it can be extended to a triangle with a third value as is shown in Figure 3.8b. When a three-variable subnetwork is not path-consistent, we can enforce path-consistency by making the necessary inferences. If in the input specification there is no constraint between x and y (meaning that everything is allowed), a new constraint can be deduced by the length-3 path from x to y through z. Thus, the necessary inferences in this case are adding binary constraints or tightening binary constraints (deleting tuples from the relation). In the coloring example, (see the matching diagram in Figure 3.8a), if we begin with R_{xy}, this implies emptying the constraint R_{xy} and declaring inconsistency.

Example 3.3.3 Consider the network of three variables $\{x, y, z\}$ where each variable has the domain $\{1, 2\}$, that has two equality constraints: $R_{xz}: \ x = z$, and $R_{yz}: \ y = z$.

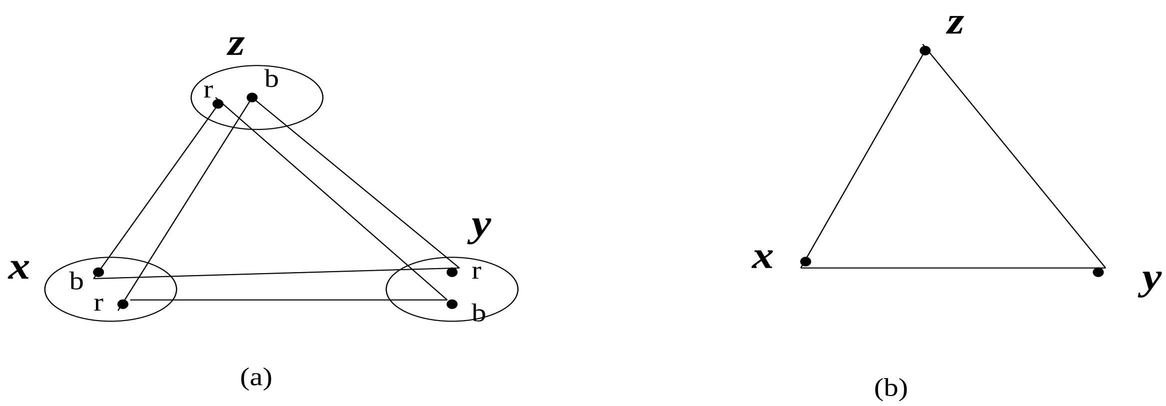

Figure 3.8: (a) The matching diagram of a 2-value graph coloring problem. (b) Graphical picture of path-consistency using the matching diagram.

To make this network path-consistent, the constraint R_{xy}; $x = y$ should be inferred and added to the network. □

In analog to Revise (which can also be called revise-2 as it deals with two variables only) for achieving arc-consistency, we will define a procedure *Revise-3*$((x, y), z)$ (Figure 3.9) that takes a pair of variables (x, y) and their constraint R_{xy}, which we wish to modify, (and which can also be the universal constraint, allowing all possible pairs), and a third variable, z, and returns the loosest constraint R'_{xy} satisfying path-consistency. It tests if each pair of consistent values in R_{xy} can be extended consistently to a value of z, and if not, it deletes the violating pair.

Revise-3$((x, y), z)$
input: a three-variable subnetwork over (x, y, z), R_{xy}, R_{yz}, R_{xz}.
output: R_{xy} path-consistent with z.
1. **for** each pair $(a, b) \in R_{xy}$
2. **if** no value $c \in D_z$ exists such that $(a, c) \in R_{xz}$ and $(b, c) \in R_{yz}$
3. **then** delete (a, b) from R_{xy}.
4. **endif**
5. **endfor**

Figure 3.9: Revise-3

.

Revise-3($(x, y), z$), can be expressed succinctly as the composition,

$$R_{xy} \leftarrow R_{xy} \cap \pi_{xy}(R_{xz} \bowtie D_z \bowtie R_{zy})$$

Clearly, Revise-3 makes the minimal changes necessary to make R_{xy} path-consistent relative to z. The reader can verify that its performance is characterized by

Proposition 3.3.4 *The complexity of* Revise-3 *is $O(k^3)$, where k bounds the domain size.* □

Proof: The claim assumes that testing the consistency $(a, b) \in R_{xy}$ can be done in constant time using hash tables. Otherwise a *log k* factor should be introduced. When using the tightness parameter t, the worst-case complexity of Revise-3 is $O(t \cdot k)$ and its best-case performance may take just t steps. □

Path-consistency can be applied to a subnetwork, or it can be fully enforced with respect to all subnetworks of size 3, yielding a *path-consistent network.* The following two algorithms, *Path-consistency-1* (PC-1) and *Path-consistency-3* (PC-3) enforce path-consistency on a network in an manner analogous to AC-1 and AC-3. The first applies Revise-3 in a brute-force manner to every triplet of variables until there is one full cycle with no change, as shown in Figure 3.10. Its complexity can be analyzed in a similar manner to AC-1.

PC-1($\mathcal{R}$)
input: a network $\mathcal{R} = (X, D, C)$.
output: $\mathcal{R}'$ a path consistent network equivalent to $\mathcal{R}$.
1. **repeat**
2.   **for** $k \leftarrow 1$ to n
3.   **for** $i, j \leftarrow 1$ to n
4. $R_{ij} \leftarrow R_{ij} \cap \pi_{ij}(R_{ik} \bowtie D_k \bowtie R_{kj}) /^* \ (Revise - 3((i, j), k))$
5.   **endfor**
6.   **endfor**
7. **until** no constraint is changed.

Figure 3.10: Path-consistency-1 (PC-1)

Proposition 3.3.5 *Given a network $\mathcal{R}$, PC-1 generates an equivalent path-consistent network, with complexity $O(n^5 k^5)$ or $O(n^5 \cdot t^2 \cdot k)$ when k bounds the domain sizes and t bounds the tightness of the constraints.*

Proof: Since in each cycle (steps 3, 4 and 5) we may eliminate only one pair of values from one constraint, the number of cycles is $O(n^2k^2)$ ($O(n^2 \cdot t)$, resp.). Since we have $O(n^3)$ triplets of variables, and since processing each by Revise-3 is $O(k^3)$ (resp, $O(t \cdot k)$), each cycle costs $O(n^3k^3)$ (resp, $O(n^3 \cdot t \cdot k)$), yielding the resulting bound. □

Algorithm PC-3 in Figure 3.11 improves on PC-1 by maintaining a queue of ordered triplets that need to be processed (or reprocessed). Once a constraint R_{ij} is modified, deleting a pair of its values, all the triplets involving x_i or x_j and any third variable x_k should be reprocessed.

Note that in order to make the network fully path-consistent, all ordered triplets should be included in the initial queue, even those containing pairs of variables that have no direct binary constraint.

PC-3($\mathcal{R}$)
input: a network $\mathcal{R} = (X, D, C)$.
output: $\mathcal{R}'$ a path consistent network equivalent to $\mathcal{R}$.
1. $Q \leftarrow \{(i,k,j) \mid 1 \le i < j \le n, 1 \le k \le n, k \ne i, k \ne j \}$
2. **while** Q is not empty
3.   select and delete a 3-tuple (i,k,j) from Q
4. $R_{ij} \leftarrow R_{ij} \cap \pi_{ij}(R_{ik} \bowtie D_k \bowtie R_{kj})$ /* (Revise-3($(i,j),k$))
5. $Q \leftarrow Q \cup \{(l,i,j)(l,j,i) \mid 1 \le l \le n, l \ne i, l \ne j\}$
6. **endwhile**

Figure 3.11: Path-consistency-3 (PC-3)

Proposition 3.3.6 *Given a network $\mathcal{R}$, algorithm PC-3 generates an equivalent path-consistent network, and its complexity is $O(n^3k^5)$ ($O(n^3 \cdot t^2 \cdot k)$, resp.), where k and t are the domains' and constraints' tightness, respectively.*

Proof: exercise 12. □

Example 3.3.7 Consider the graph-coloring problem on a four-variable network where the domains contain only two possible colors $D_1 = D_2 = D_3 = D_4 = \{red, blue\}$, and the constraints are $x_1 \ne x_2$, $x_2 \ne x_3$, $x_3 \ne x_4$, $x_4 \ne x_1$. The constraint graph of this problem is depicted in Figure 3.12a. The network is currently arc-consistent, but not path-consistent. For example, the universal constraint between x_3 and x_1 allows the assignment $x_1 = red$ and $x_3 = blue$, yet there is no assignment to x_4 satisfying the not-equal constraints with this instantiation. Enforcing path-consistency will require adding the constraint $x_1 = x_3$. Similarly, path-consistency will add the constraint $x_2 = x_4$.

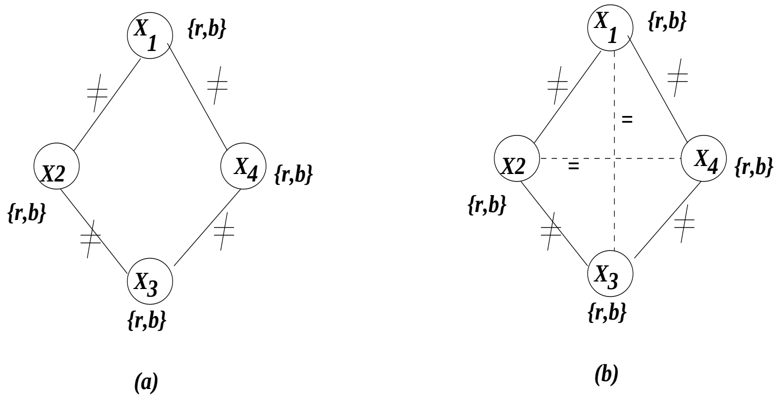

Figure 3.12: A graph-coloring graph (a) before path-consistency (b) after path-consistency

The path-consistent constraint network is depicted in Figure 3.12b. If we generate a path-consistent network by applying PC-1 to the original network, its first cycle applies Revise-3 to four triplets, generating the two equality constraints. Subsequently, a full cycle will be executed to verify that nothing changes. This requires a second processing of each triplet. If we enforce path-consistency by PC-3, we may be able to process each triplet once, if the right ordering is picked. If we apply Revise-3 to (x_1, x_3, x_2), namely to the universal constraint between x_1 and x_3 first, each triplet would be processed just once. $\square$

Like its arc-consistent counterpart, PC-3 is not optimal. An optimal algorithm akin to AC-4 can be devised. It requires operating on the relation level and maintaining *supports* for pairs of values. An algorithm exploiting such low-level consistency maintenance, which we will call *PC-4*, is available [112], and its complexity bound is $O(n^3k^3)$ or $O(n^3tk)$. It is an optimal algorithm, since even verifying path-consistency will take at least $\Omega(n^3k^3)$.

Regarding best-case performance, we observe that PC-1, PC-3, and PC-4 have properties that parallel those of arc-consistency. Algorithms PC-1 and PC-3 can be as good as $n^3 \cdot t$ and $n^3 \cdot k^2$ respectively, while algorithm PC-4 (which was not presented explicitly) requires an order of n^3k^3 (or $n^3 \cdot t \cdot k$) in the best case as well because of its initialization (see exercises).

We conclude by giving an alternative definition to path-consistency that may explain the origin of the term.

Definition 3.3.8 (A path-consistent constraint) *A constraint R_{ij} is path-consistent relative to the path of length m through the nodes $(i = i_0, i_1, ..., i_m = j)$ if for any pair $(a_i, a_j) \in R_{ij}$ there is a sequence of values $a_{il} \in D_{i_l}$ such that $(a_i = a_{i_0}, a_{i_1}) \in R_{i_0 i_1}$, $(a_{i_0}, a_{i_1}) \in R_{i_0 i_1}$, ..., and $(a_{i_{m-1}}, a_{i_m} = a_j) \in R_{i_{m-1} i_m}$.*

It can be shown that if we consider all triplets, whether or not they contain universal constraints, then a network is path-consistent (Definition 3.3.2) relative to *all* paths of length 3 iff it is path-consistent relative to paths of any length (Definition 3.3.8). On the other hand, if we require path-consistency to hold relative to triangles in the constraint graph only, then the two definitions are not the same (see exercise 14).

3.4 Higher levels of i-consistency

Arc- and path-consistency[2] algorithms process subnetworks of size 2 and 3 respectively. We now generalize the concept of local consistency to subnetworks of size i. In this case non-binary constraints will come into play as well.

Definition 3.4.1 (i-consistency, global consistency) *Given a general network of constraints $\mathcal{R} = (X, D, C)$, a relation $R_S \in C$ where $|S| = i - 1$ is i-consistent relative to a variable y not in S iff for every $t \in R_S$, there exists a value $a \in D_y$, s.t. (t, a) is consistent. A network is* i-consistent *iff given any consistent instantiation of any $i-1$ distinct variables, there exists an instantiation of any ith variable such that the i values taken together satisfy all of the constraints among the i variables. A network is* strongly i-consistent *iff it is j-consistent for all $j \leq i$. A strongly n-consistent network, where n is the number of variables in the network, is called* globally consistent.

Arc-, and path-consistency correspond to two- and three-consistency, respectively. Globally consistent networks have the property that any consistent instantiation of a subset of the variables can be extended to a consistent instantiation of all of the variables without encountering any dead-ends.

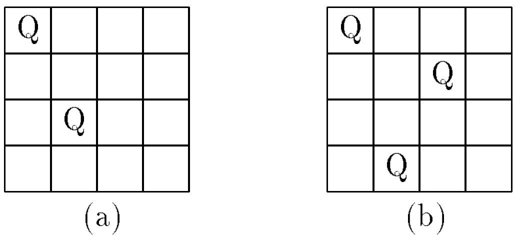

Figure 3.13: (a) Not 3-consistent; (b) Not 4-consistent

Example 3.4.2 Consider the constraint network for the 4-queens problem. It can be seen that the network is 2-consistent since, given that we have placed a single queen on

[2]restricted to binary networks

the board, we can always place a second queen on any remaining column such that the queens do not attack each other. However, the network is not 3-consistent. For example, given the consistent placement of two queens shown in Figure 3.13a, there is no way to place a queen in the third column that is consistent with the previously placed queens. Similarly the network is not 4-consistent (see Figure 3.13b). $\square$

Enforcing higher levels of consistency will require recording constraints on more than two variables. Consider first the i-level Revise rule. If a consistent instantiation of $i-1$ variables cannot be consistently extended to an i^{th} variable, we need to *rule out* that instantiation, namely, label it as a *nogood.* This means that we should record an explicit constraint over the scope of these $i-1$ variables that excludes that particular nogood assignment. As a consequence, even if the input network is binary, enforcing i-consistency may result in a non-binary network having constraints of arity $i-1$. For example, to make the 4-queen example 4-consistent we need to add, among other constraints, the ternary constraint over x_1, x_2, x_3 that forbids the tuple $(< x_1, 1 >< x_2, 4 >< x_3, 2 >)$. The Revise-i operation carries out this task and is defined in Figure 3.14.

REVISE-$i(\{x_1, x_2, ..., x_{i-1}\}, x_i)$
input: a network $\mathcal{R} = (X, D, C)$
output: a constraint R_S, $S = \{x_1, ..., x_{i-1}\}$ i-consistent relative to x_i.
1. **for** each instantiation $\bar{a}_{i-1} = (< x_1, a_1 >, < x_2, a_2 >, ..., < x_{i-1}, a_{i-1} >)$ do,
2. **if** no value of $a_i \in D_i$ exists s.t. $(\bar{a}_{i-1}, a_i)$ is consistent
   then delete $\bar{a}_{i-1}$ from R_S
   (Alternatively, let $\mathcal{S}$ be the set of all subsets of $\{x_1, ..., x_i\}$ that contain x_i
   and appear as scopes of constraints of $\mathcal{R}$, then
   $R_S \leftarrow R_S \cap \pi_S(\bowtie_{S' \subseteq \mathcal{S}} R_{S'}))$
3. **endfor**

Figure 3.14: Revise-i

Clearly the complexity of brute-force Revise-i is $O(k^i)$ if we assume that the input specification includes binary constraints only. However, when arbitrary constraints are involved, the complexity of Revise-i by itself can be as bad as $O((2k)^i)$ because there may be $O(2^i)$ constraints that need to be tested for each tuple.

Several algorithms for enforcing i-consistency have been developed. The optimal ones follow the same basic ideas of improving their performance using more involved data structure, as we have seen for arc- and path-consistency (e.g., AC-4 and PC-4). When Revise-i is incorporated into a brute-force algorithm enforcing i-consistency over a given

I-CONSISTENCY($\mathcal{R}$)
input: a network $\mathcal{R}$.
output: $\mathcal{R}'$ an i-consistent network equivalent to $\mathcal{R}$.
1. **repeat**
2. **for** every subset $S \subseteq X$ of size $i-1$, and for every x_i, do
3. let $\mathcal{S}$ be the set of all subsets of $\{x_1, ..., x_i\}$ that contain x_i and in $scheme(\mathcal{R})$
4. $R_S \leftarrow R_S \cap \pi_S(\Join_{S' \in \mathcal{S}} R_{S'})$ (this is Revise-i(S, x_i))
6. **endfor**
7. **until** no constraint is changed.

Figure 3.15: i-consistency-1

network $\mathcal{R}$, all subsets of size $i-1$ need to be modified, requiring $O(n^i(2k)^i) = O((2nk)^i)$ steps, and this should be repeated until a complete cycle produces no change. The number of such cycles equals $O(n^i k^{i-1})$, yielding overall time complexity of $O((nk)^{2i}2^i)$ and space complexity of $O(n^i k^i)$. On the other hand, a lower bound for enforcing i-consistency is $\Omega((nk)^i)$ (for more details see [36]). Indeed optimal algorithms achieving this bound analogous to AC-4 and PC-4 are available [24]. In summary,

Theorem 3.4.3 (complexity of i-consistency) *The time and space complexity of brute-force i-consistency is $O(2^i(nk)^{2i})$ and $O(n^i k^i)$, respectively. A lower bound for enforcing i-consistency is $\Omega(n^i k^i)$.* $\square$

Many additional variants of local-consistency algorithms exists in the literature and we will revisit this topic in Chapter 8 that introduces *relational consistency*. There we follow the intuition that often we want to characterize consistency level by the number of constraints involved rather than by the number of variables. One immediate approach would be to apply the variable-based consistency level to the dual problem which treats each constraint as a variable.

3.4.1 3-consistency and Path-consistency

For binary constraint networks 3-consistency is identical to path-consistency. When the networks also has ternary constraints, the definition of 3-consistency requires testing also ternary constraints.

Example 3.4.4 Suppose a constraint network involves three variables x, y, z having domains $\{0,1\}$ and a single ternary constraint $R_{xyz} = \{(0,0,0)\}$. If we apply path-consistency the algorithm will do nothing since there are no binary constraints to test, namely the network is path-consistent. However, the network is *not* 3-consistent: we can assign the values $\langle x, 1\rangle, \langle y, 1\rangle$ (since there is no constraint) while we cannot extend this assignment to z in a way that satisfies the given ternary constraints. Indeed, if we apply 3-consistency to this network we will add the constraint $R_{xy} = \{(< x, 0 >< y, 0 >)\}$ in addition to the constraint $R_x = \{(< x, 0 >)\}$. □

We can see that Revise-3 operation, as depicted in Figure 3.14 will indeed test consistency relative to ternary constraints as well.

3.4.2 Trees and bi-valued networks

As we have seen, both arc- and path-consistency can sometimes decide inconsistency and for some restricted classes of constraint problems, they are even guaranteed to solve the problem. Such *tractable* classes will be discussed at length in a later chapter. Here we will only mention two such cases.

Consider a binary constraint network whose graph is a tree, having no cycles. If arc-consistency is enforced on such a tree network and if no domain becomes empty as a result, then the problem is consistent (see exercise 11).

Next, consider the case of an arbitrary binary constraint network whose domain sizes are bounded to 2 (i.e., $k = 2$). If we apply path-consistency to such a bi-valued binary constraint network and no constraint becomes empty, then the network is guaranteed to have a solution. Moreover, a path-consistent bi-valued binary network is guaranteed to be minimal and decomposable (exercise 7).

3.5 Consistency for numeric and boolean constraints

When we process constraint systems that have a special syntactic form such as numeric constraints expressed algebraically or boolean constraints expressed by boolean expressions, the composition operation can be applied directly to the syntactic form of those expressions.

Example 3.5.1 Consider the following unary and binary numeric constraints restricted to the integers and defined over closed intervals:

$$D_x: \; x \in [1, 10], \quad D_y: \; y \in [5, 15], \quad R_{xy}: \; x + y = 10$$

Clearly x is not arc-consistent relative to y, and, vice versa, y is not arc-consistent relative to x. For example, $x = 10$ is in D_x but there is no value of y in D_y satisfying $x + y = 10$.

Enforcing arc-consistency will restrict the domain of x to $x \in [1, 5]$ and the domain of y to $[5, 9]$. In this case, composition can be expressed as linear elimination: from $x + y = 10$ and $-y \leq -5$ we infer $x \leq 5$ by summing the first two expressions.

Assume now that we also have variable z with domain $D_z : z \in [-10, 10]$ and a constraint $R_{yz} : y + z \leq 3$. Clearly x and z are not path-consistent relative to y since, for example the pair $x = 1, z = -3$ cannot be extended to a value of y. Indeed we can make x, z path-consistent relative to y by adding the constraint $x - z \geq 7$ which is obtained by summing $x + y = 10$ and $-y - z \geq -3$ (see also exercise 16). □

Example 3.5.2 Consider the two clauses,

$$(A \vee \neg B) \ and \ (B).$$

In this case, (A) is the proposition that "A is true" and $(\neg B)$ is the proposition that "B is false". In this case, B is arc-consistent relative to A, but A is not arc-consistent relative to B. If A is false, B has to be false, while it is restricted to being *true*. Arc-consistency is achieved if we add the unary constraint (A), restricting A's domain to true. In this boolean domain, the composition operation in Eq. 9.1 takes the form of *resolution*: from $(A \vee \neg B)$ and (B), we can infer (A).

If we have, in addition, the clause $B \vee C$, then making A and C path-consistent relative to B requires the clause $A \vee C$ that can be obtained by resolution over B of the two clauses $(A \vee \neg B)$ and $(B \vee C)$.

□

3.6 Summary

The chapter introduces the basic inference algorithms for constraint processing, called arc, path and i-consistency. Figure 3.16 demonstrate schematically how arc-, path-, and i-consistency are applied to one, two, or $i - 1$ variables. Arc-consistency applies to subnetworks of size 2, path-consistency to subnetworks of size 3, and i-consistency to subnetworks of size i. The form of these consistency-enforcing methods for Boolean and numeric constraints was demonstrated.

Since consistency-enforcing algorithms infer new constraints, they change the structure of the network. Arc-consistency restricts the domains of variables, path-consistency restricts and add constraints on pairs of variables, while i-consistency enforces constraints of arity $i - 1$. These changes can be reflected in the problem's constraint graph as depicted in Figure 3.17. i-bounded consistency algorithms are polynomial (exponential in i) and are not guaranteed to solve the problem. As i increase their power in deciding the problem's consistency increases as well. In Chapter 8 we will present another class of consistency algorithms that are defined by the constraints themselves.

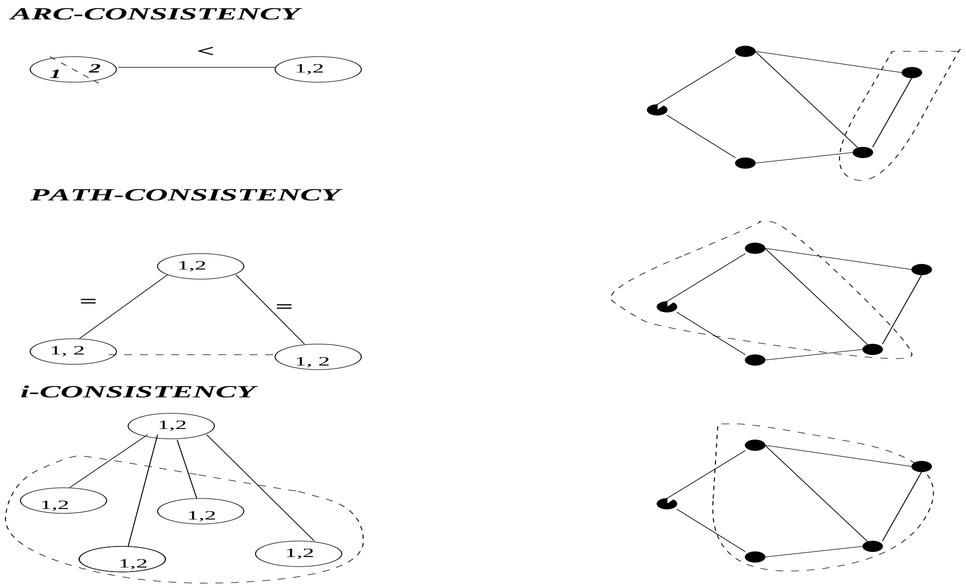

Figure 3.16: The scope of consistency enforcing: (a) arc-consistency, (b) path-consistency, (c) i-consistency

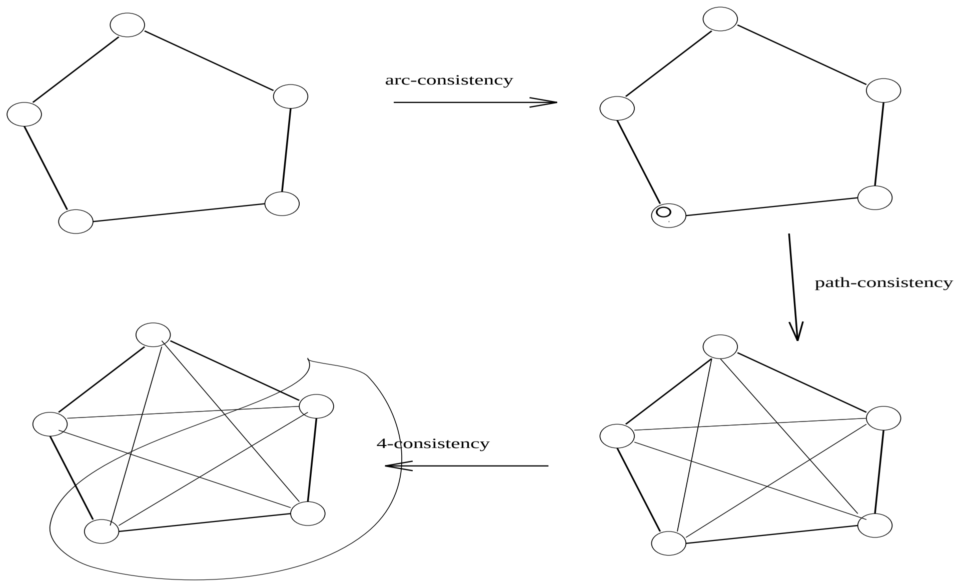

Figure 3.17: Changes in the network's graph as a result of arc-consistency, path-consistency and 4-consistency.

3.7 Chapter notes

The work on constraint propagation and consistency algorithms was initiated by Waltz [165], who introduced a constraint propagation algorithm (later called arc-consistency) for the three-dimensional cubic drawing interpretations. He demonstrated empirically that for some polyhedral figures, basic propagation algorithms are sufficient to solve these problems. This work was formalized and extended by Montanari in his paper [114], who also introduced the term "constraint networks". Montanari defined the notion of path-consistency, presented algorithms and analyses. Building on these two works, Mackworth [102] distinguished three properties that characterize local consistency of networks, *node*, *arc*, and *path consistency*, and introduced several algorithms: (AC-1, AC-2, and AC-3), for imposing arc-consistency, and PC-1, PC-2, and PC-3 for imposing path-consistency. The complexity of these algorithms were subsequently analyzed by Mackworth and Freuder [104] who also observed that arc-consistency can solve tree-like problems. Freuder [50] then generalized these notions to k-consistency. Mohr and Henderson [113] presented optimal algorithms for arc- and path- consistency, called AC-4 and PC-4, both were based on the earlier work by Laurier on a language for solving combinatorial problems [97] and these optimal algorithms were generalized for i-consistency in [24]. Subsequently several authors developed more efficient and refined arc-consistency algorithms for special classes of constraints such as monotonic and functional constraints [44, 123], continuing numbering such improved algorithms (AC-5,AC-6, AC-7, PC-5, PC-6, PC-7). In particular algorithm AC-6 [14] is able to achieve optimla worst-case complexity while avoiding the bad average case of AC-4, making it practically superior to both AC-4 and AC-3.

3.8 Exercises

1. Consider the following network having 8 variables named 1,2...,8, each having domains $\{1, 2, 3, 4\}$, whose constraints and graph are in Figure 3.18. Find an equivalent arc- and path-consistent network. Is the path-consistent network minimal? Is it backtrack-free?

2. Consider the CSP formulation of the *Zebra problem* where you have 25 variables, divided into clusters where the domains are house numbers (problem 3 in Chapter 2) see Figure 3.19. Is your CSP formulation of the Zebra problem arc-consistent? If not, describe an equivalent arc-consistent network. Is it path-consistent?

3. Prove that algorithm *AC-1* generates an equivalent arc-consistent network and when an empty domain is encountered, the network has no solution.

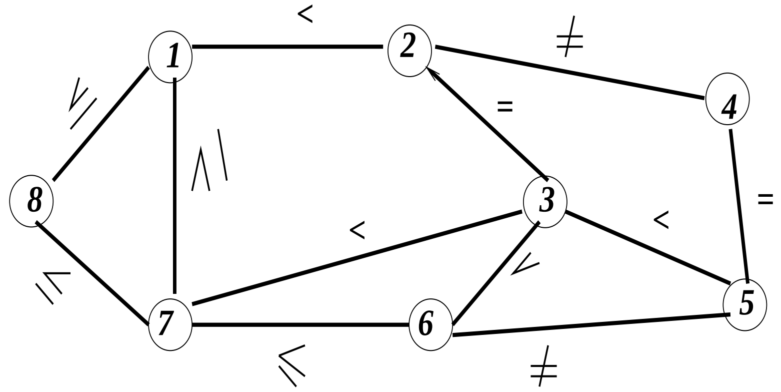

Figure 3.18: A constraint graph

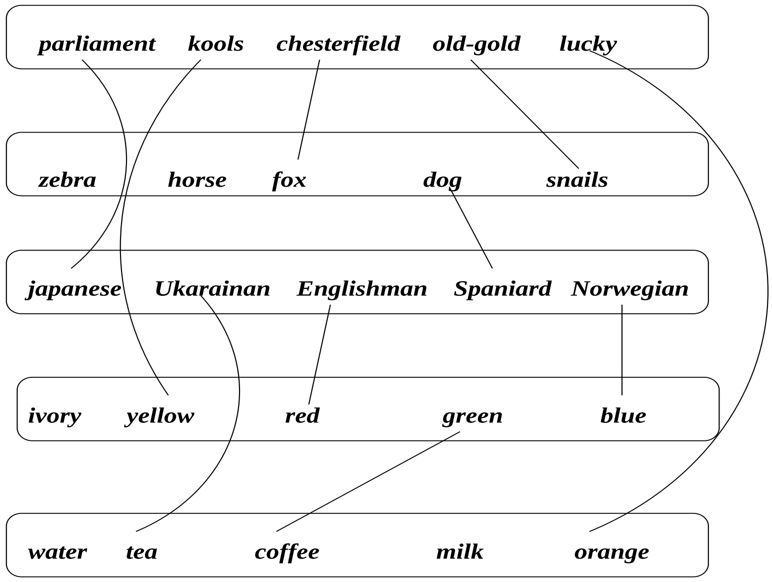

Figure 3.19: Constraint graph of the zebra problem

4. Prove that the time and space complexity of *AC-4* is $O(ek^2)$, where e is the number of constraints and k bounds the domain size.

5. Consider a binary network formulation of the crossword puzzle, and suppose that we are to fill it with the words from the following list using each word at most once:

   - HOSES, LASER, SAILS, SHEET, STEER,
   - HEEL, HIKE, KEEL, KNOT, LINE,
   - AFT, ALE, EEL, LEE, TIE.

   Is the problem arc-consistent? If not, find an equivalent arc-consistent network. Is the network path-consistent? If not enforce path-consistency.

6. Prove that a minimal network is always path-consistent: any consistent pair of values can be consistently extended by any third variable.

7. Prove that a bi-valued non-empty path-consistent binary network is consistent. Prove that it is also a minimal network.

8. Implement algorithms AC-3 and AC-4, and apply them to a set of randomly generated problems as well as to the n-queens problem. Which algorithm is better in practice?

9. Consider a graph coloring problem with 3 colors having 4 nodes x_1, x_2, x_3, x_4 where every two nodes are connected. Is the problem arc-consistent? path-consistent? 4-consistent? If not, enforce strong 4-consistency. Can you make the problem 4-consistent by adding binary constraints only?

10. Let $\mathcal{R}$ be an arbitrary 3-graph coloring problem.
    1. Discuss what would be the effect of enforcing 2- and 3-consistency on $\mathcal{R}$.
    2. What would be the effect of enforcing $2-\ ,3-\ ,4-\ ,...k-1-\ ,k-\ ,k+1-$ consistency on a k-coloring problem?

11. Prove that an arc-consistent binary network that has no cycle is consistent.

12. Derive a worst-case complexity analysis for algorithm *PC-3*.

13. Read algorithm PC-4 in [113] and show that the best-case performance of *PC-4* is $O(n^4k^3)$ or $O(n^3tk)$, where n is the number of variables, k bounds the domain sizes, and t bounds the number of tuples in a a relation.

14. Show that the two definitions of path-consistency are definition 3.3.2 and 3.3.8 are equivalent only when *all* triplets of variables are considered.

15. A graph is transitively closed iff any two nodes connected by a path are directly connected. Prove that the constraint graph of a path-consistent network is transitively closed.

16. Generate an arc and path-consistent network which is equivalent to the network in Example 3.5.1.

$$D_x : \; x \in [1, 10], \quad D_y : \; y \in [5, 15], \quad R_{xy} : \; x + y = 10$$

$$D_z : z \in [-10, 10], \quad R_{yz} : y + z \leq 3$$