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 (this follows immediately from Proposition 2.2). This property is termed *arc-consistency*, and it can be satisfied by nonminimal networks as well. Arc-consistency can be enforced on any network by an efficient computation that, because of its local and distributed character, is often called *propagation*.

The following example more clearly demonstrates the notion of arc-consistency. 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 3.1

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 $x < y$. The constraint R_{xy} is depicted in a *matching diagram*[1] in Figure 3.1(a), where the domain of each variable is an enclosed set of points, and arcs connect points that correspond to consistent pairs of values. (Note: This type of diagram should not be confused with the constraint graph of the network.) Because the value $3 \in D_x$ has no consistent matching value in D_y, we say 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. In matching diagrams, a constraint is not arc-consistent if any of its variables have *lonely* values.

Now, if 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 3.1(b). 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 **(arc-consistency)**

Given a constraint network $\mathcal{R} = (X, D, C)$, 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. •

1. Also called a *microstructure* (Jégou 1993).

Figure 3.1 A r
abl
no

Figure 3.2

PROPOSIT
3.1

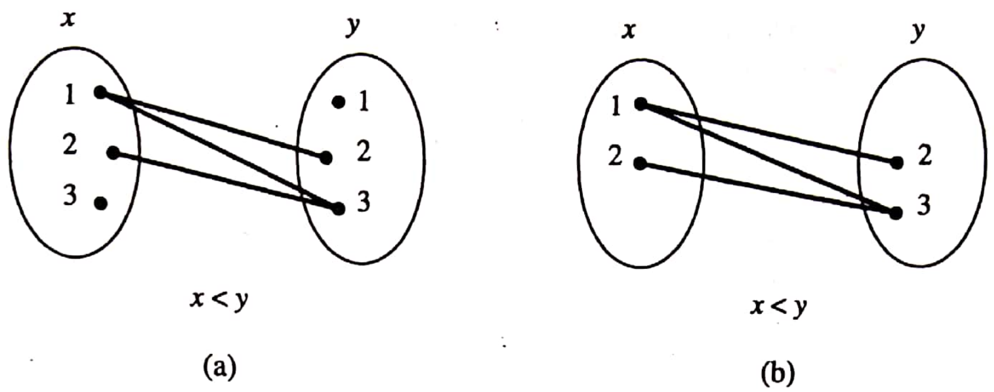

Figure 3.1 A matching diagram describing the arc-consistency of two variables x and y: (a) The variables are not arc-consistent. (b) The domains have been reduced, and the variables are now arc-consistent.

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: D_i, such that x_i is arc-consistent relative to x_j.

```
for each a_i ∈ D_i
    if there is no a_j ∈ D_j such that (a_i, a_j) ∈ R_ij
        then delete a_i from D_i
    endif
endfor
```

Figure 3.2 The REVISE procedure.

As we saw in the earlier example, we can make a binary constraint arc-consistent by shrinking the domains of the variables in its scope. If a value does not participate in a solution of a two-variable subnetwork, it will clearly not be part of a complete solution. But how do we ensure that we only eliminate values that will not affect the set of the network's solutions? The simple procedure REVISE((x_i), x_j), shown 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 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 3.1 The complexity of REVISE is $O(k^2)$, where k bounds the domain size. •

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

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

In this case, D_i stands for the one-column relation over x_i. (Consult Section 1.3 for the definitions of the 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 an entire network. Arc-consistency of a whole network is accomplished by applying the REVISE procedure to all pairs of variables, although applying the procedure just once to each pair of variables is sometimes not enough to ensure the arc-consistency of a network, as we see in the following example.

EXAMPLE 3.2 Consider now the matching diagram of the three-variable constraint network depicted in Figure 3.3(a). Without knowing the nature of the constraint between y and x, 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, to make $\{x, z\}$ arc-consistent, we must delete a value from the domain of x, which will leave y no longer arc-consistent relative to x. 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. •

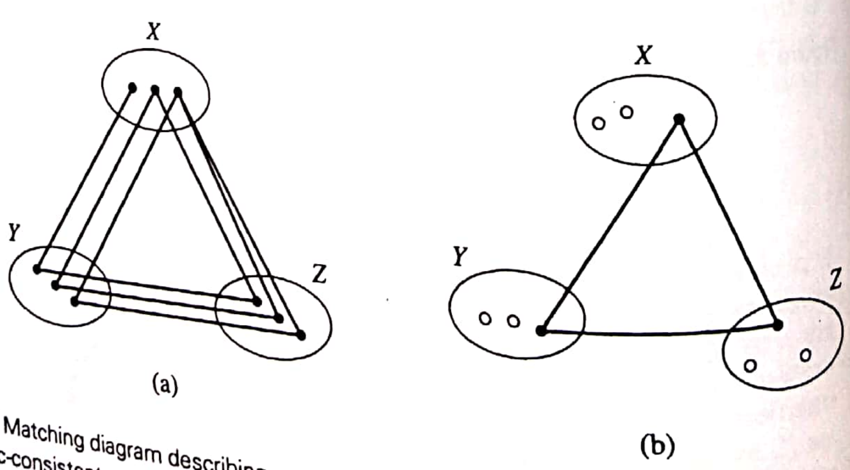

Figure 3.3 (a) Matching diagram describing a network of constraints that is not arc-consistent. (b) An arc-consistent equivalent network.

AC-1($\mathcal{R}$)
Input: A ne
Output: $\mathcal{R}'$,
1. **repeat**
2. **for**
3.
4.
5. **en**
6. **until** nc

Figure 3.4 ARC-CONSISTENC

Algorith consistency c rule to all pa that no dom the matchin
Occasio

EXAMPLE 3.3 Conside of all tl Applyi the seq tions) constra to {3}. proces netwo

As we work, and has no solu

PROPOSITION 3.2 Given bound

Proof In AC- $O(ek^2$ just o the m of O(

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

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

In this case, D_i stands for the one-column relation over x_i. (Consult Section 1.3 for the definitions of the 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 an entire network. Arc-consistency of a whole network is accomplished by applying the REVISE procedure to all pairs of variables, although applying the procedure just once to each pair of variables is sometimes not enough to ensure the arc-consistency of a network, as we see in the following example.

EXAMPLE 3.2 Consider now the matching diagram of the three-variable constraint network depicted in Figure 3.3(a). Without knowing the nature of the constraint between y and x, 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, to make $\{x, z\}$ arc-consistent, we must delete a value from the domain of x, which will leave y no longer arc-consistent relative to x. 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. ●

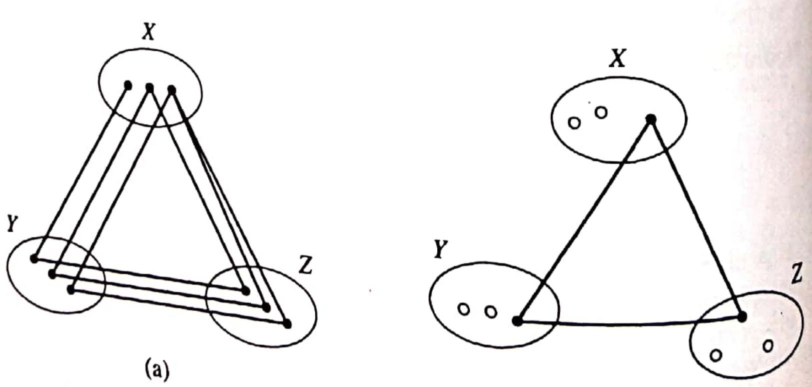

Figure 3.3 (a) Matching diagram describing a network of constraints that is not arc-consistent. (b) An arc-consistent equivalent network.

AC-1($\mathcal{R}$)
Input: A net
Output: $\mathcal{R}'$,
1. **repeat**
2. **for**
3.
4.
5. **end**
6. **until** no

Figure 3.4 ARC-CONSISTENCY

Algorithm ... consistency o... rule to all pai... that no doma... the matching ...

Occasion...

EXAMPLE 3.3 Conside... of all th... Applyin... the sequ... tions) t... constrai... to {3}. W... processi... network...

As we ... work, and w... has no solut...

PROPOSITION 3.2 Given ... bounde...

Proof In AC-1 ... $O(ek^2)$. ... just one ... the max ... of $O(n$ ...

```
AC-1(ℛ)
Input: A network of constraints ℛ = (X, D, C).
Output: ℛ', which is the largest arc-consistent network equivalent to 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 ← D_i ∩ π_i (R_ij ⋈ D_j))
4.         REVISE((x_j), x_i) (or D_j ← D_j ∩ π_j (R_ij ⋈ D_i))
5.     endfor
6. until no domain is changed
```

Figure 3.4 ARC-CONSISTENCY-1 (AC-1).

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 in the matching diagram of Figure 3.3(a) is given in Figure 3.3(b).

Occasionally, arc-consistency enforcing may discover inconsistency.

EXAMPLE 3.3 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 $x<y$, $y<z$, $z<x$. 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 further reduced to $\{2\}$ and the domain of z to $\{3\}$. When R_{zx} is processed, the domain of z becomes empty. Subsequent processing will empty the domains of y and z, and we conclude that the network is inconsistent. •

As we have seen, algorithm AC-1 generates an equivalent arc-consistent network, and when an empty domain is encountered, we conclude that the network has no solution. AC-1 has the following complexity:

PROPOSITION 3.2 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 In AC-1, one cycle through all the binary constraints (steps 2–5) takes $O(ek^2)$. Since, in the worst case, one cycle may cause the deletion of just one value from one domain, and since, overall, there are nk values, the maximum number of such cycles is nk, resulting in the overall bound of $O(n \cdot ek^3)$. •

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 ∈ R
        queue ← queue ∪ {(x_i, x_j), (x_j, x_i)}
endfor
while queue ≠ {}
        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 ← queue ∪ {(x_k, x_i), k ≠ i, k ≠ j, R_ki ∈ R}
        endif
endwhile
```

Figure 3.5 ARC-CONSISTENCY-3 (AC-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 improved version of arc-consistency establishes and maintains 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 is placed back in the queue only if the domain of its second variable is modified as a result of the processing of adjacent constraints. We name this algorithm ARC-CONSISTENCY-3 (AC-3) to conform to the usage already established in the community. A previous improvement called AC-2 provides only a minor advancement step in between these two versions. Algorithm AC-3, which uses a queue data structure, is presented in Figure 3.5.

EXAMPLE 3.4 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 Figure 3.6(a). Assume that we apply AC-3 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 the problem, 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 place (x, z) back on the queue. Processing (y, z) causes no further change, but when (x, z) is revised,

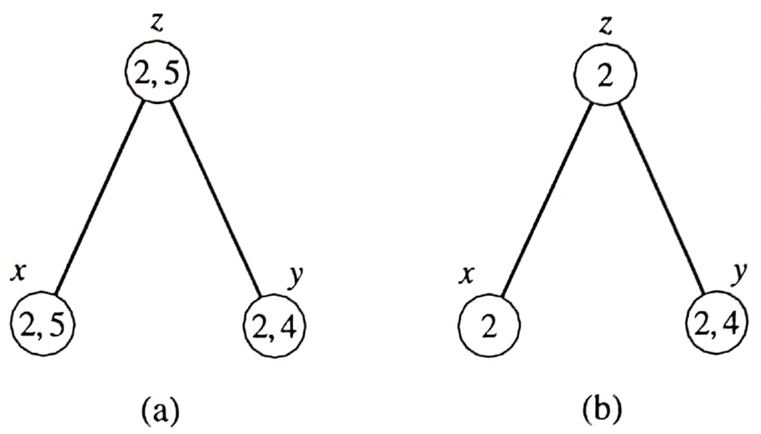

Figure 3.6 A three-variable network, with two constraints: z divides x, and z divides y (a) before and (b) after AC-3 is applied.

5 will be deleted from D_x. At this point, no further constraints will be inserted into the queue (remember, no constraint already in the queue needs to be inserted), and the algorithm terminates with the domains $D_x = \{2\}$, $D_z = \{2\}$, and $D_y = \{2, 4\}$. •

Algorithm AC-3 processes each constraint at most $2k$ times, where k bounds the domain size, since each time it is reintroduced into the queue, the domain of a variable in its scope has just been reduced 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 the following:

PROPOSITION 3.3 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 the worst case of merely verifying the arc-consistency of a network requires ek^2 operations. Indeed, algorithm ARC-CONSISTENCY-4 (AC-4) achieves this optimal performance. AC-4 does not use REVISE or the composition operator as an atomic operation. Instead, it exploits the underlying microstructure of the constraint relation and tunes its operation to that level.

AC-4 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. A value a_i is then removed from the domain D_i if it has no support from some neighboring variable. The algorithm maintains a *List* of currently unsupported variable-value pairs, a counter array $counter(x_i, a_i, x_j)$ of supports for a_i from x_j, and an array $S_{(x_j,a_j)}$ that points to all values in other variables supported by $\langle x_j, a_j \rangle$. In each step, the algorithm picks up an unsupported value from *List*, adds it to the *removed* list M, and updates all the supports of potentially affected values. Those

```
AC-4(ℛ)
Input: A network of constraints ℛ.
Output: An arc-consistent network equivalent to ℛ.
Initialization: M ← ∅
      initialize S(xi,ci), counter(i, ai, j) for all Rij
      for all counters
            if counter(xi, ai, xj) = 0 (if ⟨xi, ai⟩ is unsupported by xj)
                  then add ⟨xi, ai⟩ to List
            endif
      endfor
while List is not empty
      choose ⟨xi,ai⟩ from List, remove it, and add it to M
      for each ⟨xj,aj⟩ in S(xi,ai)
            decrement counter(xj, aj, xi)
            if counter(xj, aj, xi) = 0
                  then add ⟨xj, aj⟩ to List
            endif
      endfor
endwhile
```

Figure 3.7 ARC-CONSISTENCY-4 (AC-4).

values that became unsupported as a result are placed in *List*. If a_j is unsupported, the counters of values that it supports will be reduced. Algorithm AC-4 is given in Figure 3.7.

EXAMPLE 3.5

Consider the problem in Figure 3.6. Initializing the $S_{(x,a)}$ arrays (indicating all the variable-value pairs that each $\langle x, a\rangle$ supports), we have

$$S_{(z,2)} = \{\langle x,2\rangle, \langle y,2\rangle, \langle y,4\rangle\}, S_{(z,5)} = \{\langle x,5\rangle\}, S_{(x,2)} = \{\langle z,2\rangle\},$$
$$S_{(x,5)} = \{\langle z,5\rangle\}, S_{(y,2)} = \{\langle z,2\rangle\}, S_{(y,4)} = \{\langle z,2\rangle\}.$$

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, $List = \{\langle z,5\rangle\}$, $M = \emptyset$. Once $\langle z,5\rangle$ is removed from *List* and placed in M, the counter of $\langle x, 5\rangle$ is updated to $counter(x, 5, z) = 0$, and $\langle x, 5\rangle$ is placed in *List*. Then, $\langle x, 5\rangle$ is removed from *List* and placed in M. Since the only value it supports is $\langle z, 5\rangle$ and since $\langle z, 5\rangle$ is already in M, *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 $S_{(x_j,a_j)}$ is on the order of ek^2. We can show:

PROPOSITION 3.4 The time and space complexity of AC-4 is $O(ek^2)$.

Proof See Exercise 4. •

Since worst-case complexity often overestimates, and since average-case analysis is hard to achieve, it is sometimes useful to introduce more refined parameters into the analysis. Instead of using $O(k^2)$—which is the size of the universal relation (remember 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 is 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 may also provide insight. 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 remains at 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 AC-4 is optimal in the worst case.

3.3 Path-Consistency

As we saw in Example 3.3, arc-consistency can sometimes decide inconsistency by discovering an empty domain. However, arc-consistency is not complete for deciding consistency because it makes inferences based on a single (binary) constraint and single domain constraints.

EXAMPLE 3.6 Consider the example we mentioned at the outset having three variables x, y, z with respective domains $\{red, blue\}$ and constrained by $x \neq y$, $y \neq z$, $z \neq x$. This constraint network is arc-consistent without reducing any domains, and therefore AC enforcement will not reveal the network's inconsistency. Although the constraint R_{xy}, which is $x \neq y$, allows the assignment from *List* and placed in M, the counter of $\langle x, 5\rangle$ is updated to $counter(x, 5, z) = 0$, and $\langle x, 5\rangle$ is placed in *List*. Then, $\langle x, 5\rangle$ is removed from *List* and placed in M. Since the only value it supports is $\langle z, 5\rangle$ and since $\langle z, 5\rangle$ is already in M, *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 $S_{(x_j,a_j)}$ is on the order of ek^2. We can show:

PROPOSITION 3.4 The time and space complexity of AC-4 is $O(ek^2)$.

Proof See Exercise 4. •

Since worst-case complexity often overestimates, and since average-case analysis is hard to achieve, it is sometimes useful to introduce more refined parameters into the analysis. Instead of using $O(k^2)$—which is the size of the universal relation (remember 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 is 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 may also provide insight. 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 remains at 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 AC-4 is optimal in the worst case.

3.3 Path-Consistency

As we saw in Example 3.3, arc-consistency can sometimes decide inconsistency by discovering an empty domain. However, arc-consistency is not complete for deciding consistency because it makes inferences based on a single (binary) constraint and single domain constraints.

EXAMPLE 3.6 Consider the example we mentioned at the outset having three variables x, y, z with respective domains $\{red, blue\}$ and constrained by $x \neq y$, $y \neq z$, $z \neq x$. This constraint network is arc-consistent without reducing any domains, and therefore AC enforcement will not reveal the network's inconsistency. Although the constraint R_{xy}, which is $x \neq y$, allows the assignment