```
procedure WALKSAT
Input: A network $\mathcal{R} = (X, D, C)$, number of flips MAX_FLIPS, MAX_TRIES,
    probability $p$.
Output: "True," and a solution, if the problem is consistent, "false," and an
    inconsistent best assignment, otherwise.
  1. for i = 1 to MAX_TRIES do
  2. start with a random initial assignment $\bar{a}$.
  3. Compare best assignment with $\bar{a}$ and retain the best.
  4. for i = 1 to MAX_FLIPS
     • if $\bar{a}$ is a solution, return true and $\bar{a}$.
     • else,
        i.  pick a violated constraint $C$, randomly
        ii. choose with probability $p$ a variable-value pair $\langle x,a' \rangle$ for $x \in scope(C)$, or, with probability $1 - p$, choose a variable-value pair $\langle x,a' \rangle$ that
            minimizes the number of new constraints that break when the value
            of $x$ is changed to $a'$ (minus 1 if the current constraint is satisfied).
        iii. Change $x$'s value to $a'$.
  5. endfor
  6. return false and the best current assignment.
```

Figure 7.2 Algorithm WALKSAT.

with probability 1. Figure 7.2 presents a simple version of WALKSAT. WALKSAT has been shown to be highly effective on a range of problem domains, such as hard random k-SAT problems, logistics planning formulas, graph coloring, and circuit synthesis formulas.

EXAMPLE 7.2 Following Example 7.1, with the initial assignment of value 1 to all the variables, we will first select an unsatisfied clause, such as $(\neg B \vee \neg C)$, and then select a variable. If we try to minimize the number of additional constraints that would be broken, we will select B and flip its value. Subsequently, the only unsatisfied clause is $\neg C$, which is selected and flipped. •

Simulated Annealing

A famous stochastic local search method is *simulated annealing*. Simulated annealing uses a noise model inspired by statistical mechanics. At each step, the algorithm picks a variable and a value, and then computes δ, the change in the cost function when the value of the variable is changed to the value picked. If this change

improves or has no impact on the cost function, we make the change. Otherwise, we make the change with probability $e^{-\delta/T}$, where T is a parameter called *temperature*. The temperature can be held constant, or slowly reduced from a high temperature to a near zero temperature according to some "cooling schedule." This algorithm converges to the exact solution if the temperature T is reduced gradually.

7.2.2 Properties of Local Search

The most notable property of local search is that it terminates at local minima. When randomness is introduced, the performance of local search can be illuminated using the theory of random walks. Consider a satisfiable 2-SAT formula having N variables and let $\bar{a}$ denote a satisfying assignment. The random walk procedure starts with a random assignment $\bar{a}'$. On the average this truth assignment will differ from $\bar{a}$ on $N/2$ propositional variables. Consider the unsatisfied clauses in the formula. Since the clause is unsatisfied, $\bar{a}'$ assigns negative values to both of its literals while $\bar{a}$ assigns "true" to at least one of these literals (since $\bar{a}$ satisfies every clause). Now, randomly select a variable in this clause and flip its value in $\bar{a}'$. Since there are only two literals, there is at least a 50% chance of selecting the variable corresponding to a literal set to true in $\bar{a}$. Therefore, with at least a 50% chance, the distance between the new assignment and $\bar{a}$ will be reduced by 1 and with less than a 50% chance we will increase the distance. Consider a general random walk that starts at a given location and takes L steps, either one step to the left or to the right with probability 0.5. It can be shown that after L^2 steps, such a walk will on the average travel a distance of L units from its starting point. Given that our random walk started at a distance of $N/2$ from the satisfying truth assignment $\bar{a}$, after an order of N^2 steps, the walk will hit a satisfying assignment, with probability going to 1.

The analysis for 2-SAT breaks down for k-SAT with $k \geq 3$ (see Exercise 2). In fact, it can be shown that a random walk strategy will take an exponential number of flips on 3-SAT formulas, in the worst case. Indeed, some local search methods such as WALKSAT can be shown to eventually hit a satisfying assignment with probability 1 if such an assignment exists. Others, such as GSAT, WalkSAT/TABU, Novelty, and R-Novelty (Hoos and Stutzle 1999), cannot.

A related property is that SLS algorithms are *anytime*—the longer they run, the better the solution they may produce (i.e., a solution that satisfies more and more constraints). Unlike complete algorithms, however, local search algorithms cannot prove inconsistency. When the problem does not have a solution, an SLS algorithm would run forever. In practice, a time limit is set, after which a failure is reported.

We can analyze the complexity of a local search step. In general, we can maintain information from one flip to the next and recompute only the changes caused by the previous flip. This computation is normally linear in the neighborhood size of each variable in the network.

$r, b\}$

istency.

ion of this
consistent
thm's first
uality con-
g changes.
the other
to process
f we apply
t between
cessed just
•

though we can
on the relation
exploiting such
ble (Mohr and
It is an optimal
und; namely, it

and PC-4 have
nd PC-2 can be
C-4 (which was
k)) even in the

an alternative

DEFINITION 3.4 **(path-consistent constraint)**

A constraint R_{ij} is path-consistent, relative to the path of length m through the nodes $(i = i_0, i_1, \ldots, i_m = j)$, 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 $(a_i = a_{i_0}, a_{i_1}) \in R_{i_0 i_1}$, $(a_{i_0}, a_{i_1}) \in R_{i_0 i_1}, \ldots$, and $(a_{i_{m-1}}, a_{i_m} = a_j) \in R_{i_{m-1} i_m}$. •

For a constraint graph that is complete, the two definitions can be shown to be the same. However, if we require path-consistency in Definition 3.4 to hold relative to only real paths in the constraint graph, then the two above-referenced definitions are not the same (see Exercise 15).

3.4 Higher Levels of *i*-Consistency

Arc- and path-consistency algorithms process subnetworks of size 2 and 3, respectively. We're now ready to generalize the concept of local consistency to subnetworks of size i. In this case nonbinary constraints also come into play.

DEFINITION 3.5 **(*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 $a \in D_y$, such that (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*. •

Globally consistent networks are characterized by 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.

EXAMPLE 3.8

Consider the constraint network for the 4-queens problem. We see that the network is 2-consistent since, given that we have placed a single queen on the board, we can always place a second queen on any remaining column such that the two queens do not attack each other. This network is not 3-consistent, however; given the consistent placement of two queens shown in Figure 3.13(a), 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 (Figure 3.13(b)). •

$(\langle x, red\rangle, \langle y, blue\rangle)$, no color for z will be consistent with both $\langle x, red\rangle$ and $\langle y, blue\rangle$. Indeed, we can infer something about 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 violated here is called *path-consistency*, requiring consistency relative to paths of length 3 (the path from x to y that goes through z). It is interesting to note that in the minimal network path-consistency 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 of path-consistency involves only binary constraints, it is relevant only to the binary constraints of the network. An extension of the definition to nonbinary constraints is presented later; it constitutes only a minor change that is relevant when the network also has ternary constraints.

DEFINITION 3.3 **(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 $(\langle x_i, a_i\rangle, \langle x_j, a_j\rangle)$ there is a value $a_k \in D_k$ such that the assignment $(\langle x_i, a_i\rangle, \langle x_k, a_k\rangle)$ is consistent and $(\langle x_k, a_k\rangle, \langle x_j, a_j\rangle)$ is consistent. Alternatively, a binary constraint R_{ij} is path-consistent relative to x_k iff for every pair $(a_i, a_j) \in R_{ij}$, where a_i and a_j are from their respective domains, there is a value $a_k \in D_k$ such that $(a_i, a_k) \in R_{ik}$ and $(a_k, a_j) \in R_{kj}$. A subnetwork over three variables $\{x_i, x_j, x_k\}$ is path-consistent iff for any permutation of (i, j, k), R_{ij} is path-consistent relative to x_k. A network is path-consistent iff for every R_{ij} (including universal binary 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 the connected pair can be extended to a triangle with a third value as is shown in Figure 3.8(b). When a three-variable subnetwork is not path-consistent, we can enforce path-consistency by making the necessary inferences. So, if in the input specification there is no constraint between x and y (meaning that everything is allowed), we can deduce a new constraint by the length-3 path from x to y through z. In this case, the necessary inferences are adding binary constraints or tightening binary constraints (deleting tuples from the relation). In the coloring example (see the matching diagram in Figure 3.8(a)), if we attempt to make R_{xy} path-consistent relative to z, this implies emptying the constraint R_{xy} and declaring inconsistency.

Alternatively, consider again the network of three variables $\{x, y, z\}$ where all variables have the same domain and the equality constraints $R_{xz}: x = z$ and

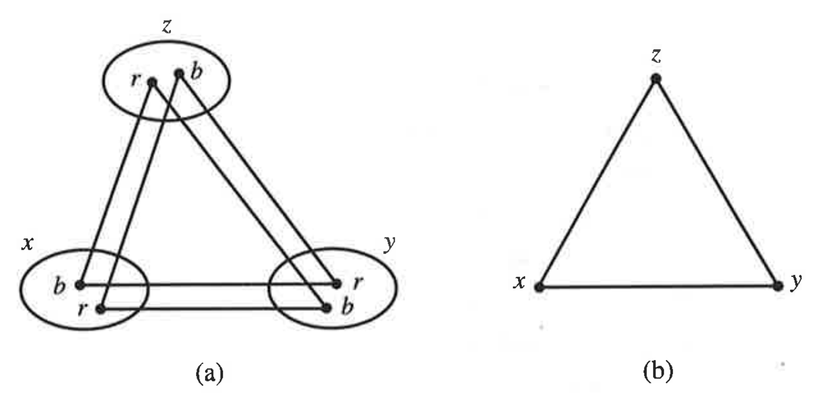

Figure 3.8 (a) The matching diagram of a two-value graph-coloring problem. (b) Graphical picture of path-consistency using the matching diagram.

```
REVISE-3((x, y), z)
Input: A three-variable subnetwork over (x, y, z), $R_{xy}$, $R_{yz}$, $R_{xz}$.
Output: Revised $R_{xy}$ path-consistent with z.
1. for each pair (a, b) ∈ $R_{xy}$
2.     if no value c ∈ $D_z$ exists such that (a, c) ∈ $R_{xz}$ and (b, c) ∈ $R_{yz}$
3.         then delete (a, b) from $R_{xy}$.
4.     endif
5. endfor
```

Figure 3.9 REVISE-3.

$R_{yz}: y = z$. To make this network path-consistent, we should infer and add to the network the constraint $R_{xy}: x = y$.

To use an analog of REVISE for arc-consistency, we define a procedure REVISE-3$((x,y),z)$ (Figure 3.9). This procedure takes a pair of variables (x,y) and their constraint that we wish to modify, R_{xy} (and which can also be the universal constraint), and a third variable, z, and returns the loosest constraint R'_{xy} that satisfies path-consistency. REVISE-3 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$((x,y),z)$ can be expressed succinctly as the composition

$$R_{xy} \leftarrow R_{xy} \cap \pi_{xy}(R_{xz} \bowtie D_z \bowtie R_{zy}) \tag{3.1b}$$

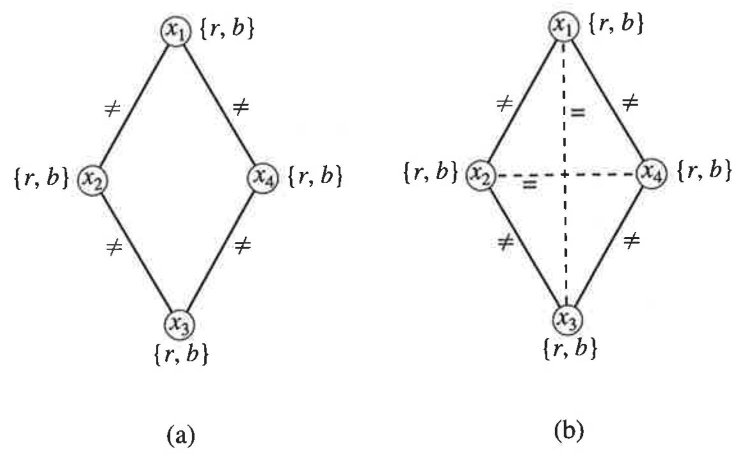

Figure 3.12 A graph-coloring graph (a) before path-consistency and (b) after path-consistency.

constraint $x_2 = x_4$. The path-consistent constraint network version of this example is depicted in Figure 3.12(b). If we generate a path-consistent network by applying PC-1 to the original network, the algorithm's first cycle applies REVISE-3 to four triplets, generating the two equality constraints. A full cycle will then be executed to verify that nothing changes. This verification requires a second processing of each triplet. On the other hand, if we enforce path-consistency by PC-2, we may be able to process each triplet only once, assuming the right ordering is picked. If we apply REVISE-3 first to (x_1, x_3, x_2), that is, to the universal constraint between x_1 and x_3, and then to (x_2, x_4, x_1), each triplet would be processed just once. •

Like its arc-consistent counterpart (AC-3), PC-2 is not optimal, although we can devise an optimal algorithm, akin to AC-4. It would require 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 (Mohr and Henderson 1986),and its complexity bound is $O(n^3k^3)$ or $O(n^3tk)$. It is an optimal algorithm, since even verifying path-consistency has that lower bound; namely, it is $\Omega(n^3k^3)$.

Regarding best-case performance, we observe that PC-1, PC-2, and PC-4 have properties that parallel those of arc-consistency. Algorithms PC-1 and PC-2 can be as good as $O(n^3 \cdot t)$ and $O(n^3 \cdot k^2)$, respectively, while algorithm PC-4 (which was not presented explicitly) requires an order of $O(n^3k^3)$ (or $O(n^3 \cdot t \cdot k)$) even in the best case because of its initialization (see Exercise 14).

Let's conclude our introduction to path-consistency by giving an alternative definition that may explain the origin of the term.

DEFINITION 3.4 **(path-consistent constraint)**

A constraint R_{ij} is path-consistent, relative to the path of length m through the nodes $(i = i_0, i_1, \ldots, i_m = j)$, 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 $(a_i = a_{i_0}, a_{i_1}) \in R_{i_0 i_1}$, $(a_{i_0}, a_{i_1}) \in R_{i_0 i_1}, \ldots$, and $(a_{i_{m-1}}, a_{i_m} = a_j) \in R_{i_{m-1} i_m}$. •

For a constraint graph that is complete, the two definitions can be shown to be the same. However, if we require path-consistency in Definition 3.4 to hold relative to only real paths in the constraint graph, then the two above-referenced definitions are not the same (see Exercise 15).

3.4 Higher Levels of *i*-Consistency

Arc- and path-consistency algorithms process subnetworks of size 2 and 3, respectively. We're now ready to generalize the concept of local consistency to subnetworks of size i. In this case nonbinary constraints also come into play.

DEFINITION 3.5 **(*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 $a \in D_y$, such that (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*. •

Globally consistent networks are characterized by 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.

EXAMPLE 3.8 Consider the constraint network for the 4-queens problem. We see that the network is 2-consistent since, given that we have placed a single queen on the board, we can always place a second queen on any remaining column such that the two queens do not attack each other. This network is not 3-consistent, however; given the consistent placement of two queens shown in Figure 3.13(a), 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 (Figure 3.13(b)). •

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. •

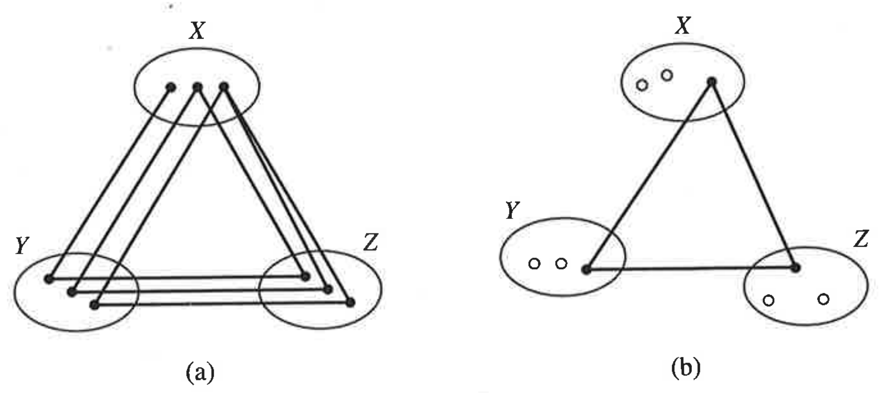

Figure 3.3 (a) Matching diagram describing a network of constraints that is not arc-consistent. (b) An arc-consistent equivalent network.

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).

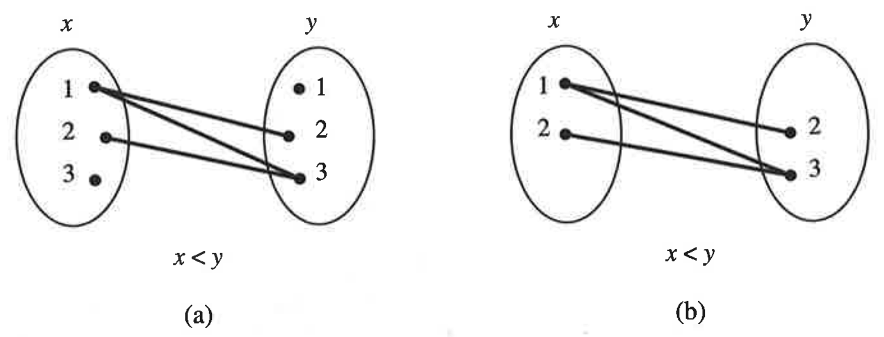

Figure 3.1 A matching diagram describing the arc-consistency of two variables x and y: (a) The variables are not arc-consistent. (b) The domains have been reduced, and the variables are now arc-consistent.

```
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.
1. for each $a_i \in D_i$
2.     if there is no $a_j \in D_j$ such that $(a_i, a_j) \in R_{ij}$
3.         then delete $a_i$ from $D_i$
4.     endif
5. 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. •