<ch11 toc ch13>

Chapter 12
GRAPH MATRICES AND APPLICATIONS

1. SYNOPSIS

Graph matrices are introduced as another representation for graphs; some useful tools resulting therefrom are examined. Matrix operations, relations, node-reduction algorithm revisited, equivalence class partitions.

2. MOTIVATIONAL OVERVIEW

2.1. The Problem with Pictorial Graphs

Graphs were introduced as an abstraction of software structure early in this book and used throughout. Yet another graph that modeled software behavior was introduced in Chapter 11. There are many other kinds of graphs, not discussed in this book, that are useful in software testing. Whenever a graph is used as a model, sooner or later we trace paths through it—to find a set of covering paths, a set of values that will sensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define a domain, whether the routine pushes or pops, or whether a state is reachable or not. Even algebraic representations such as BNF and regular expressions can be converted to equivalent graphs. Much of test design consists of tracing paths through a graph and most testing strategies define some kind of cover over some kind of graph.

Path tracing is not easy, and it’s subject to error. You can miss a link here and there or cover some links twice—even if you do use a marking pen to note which paths have been taken. You’re tracing a long complicated path through a routine when the telephone rings—you’ve lost your place before you’ve had a chance to mark it. I get confused tracing paths, so naturally I assume that other people also get confused.

One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods aren’t necessarily easier than path tracing, but because they’re more methodical and mechanical and don’t depend on your ability to “see” a path, they’re more reliable.

Even if you use powerful tools that do everything that can be done with graphs, and furthermore, enable you to do it graphically, it’s still a good idea to know how to do it by hand; just as having a calculator should not mean that you don’t need to know how to do arithmetic. Besides, with a little practice, you might find these methods easier and faster than doing it on the screen; moreover, you can use them on the plane or anywhere.

2.2. Tool Building

If you build test tools or want to know how they work, sooner or later you’ll be implementing or investigating analysis routines based on these methods—or you should be. Think about how a naive tool builder would go about finding a property of all paths (a possibly infinite number) versus how one might do it based on the methods of Chapter 8. But Chapter 8 was graphical and it’s hard to build algorithms over visual graphs. The properties of graph matrices are fundamental to test tool building.

2.3. Doing and Understanding Testing Theory

We talk about graphs in testing theory, but we prove theorems about graphs by proving theorems about their matrix representations. Without the conceptual apparatus of graph matrices, you'll be blind to much of testing theory, especially those parts that lead to useful algorithms.

2.4. The Basic Algorithms

This is not intended to be a survey of graph-theoretic algorithms based on the matrix representation of graphs. It's intended only to be a basic toolkit. For more on this subject, see EVEN79, MAYE72, PHIL81. The basic toolkit consists of:

1. Matrix multiplication, which is used to get the path expression from every node to every other node.
2. A partitioning algorithm for converting graphs with loops into loop-free graphs of equivalence classes.
3. A collapsing process (analogous to the determinant of a matrix), which gets the path expression from any node to any other node.

3. THE MATRIX OF A GRAPH

3.1. Basic Principles

A **graph matrix** is a square array with one row and one column for every node in the graph. Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column. The relation, for example, could be as simple as the link name, if there is a link between the nodes. Some examples of graphs and their associated matrices are shown in Figure 12.1 a through g. Observe the following:

1. The size of the matrix (i.e., the number of rows and columns) equals the number of nodes.
2. There is a place to put every possible direct connection or link between any node and any other node.
3. The entry at a row and column intersection is the link weight of the link (if any) that connects the two nodes in that direction.
4. A connection from node *i* to node *j* does not imply a connection from node *j* to node *i*. Note that in Figure 12.1h the (5,6) entry is *m*, but the (6,5) entry is *c*.
5. If there are several links between two nodes, then the entry is a sum; the "+" sign denotes parallel links as usual.

In general, an entry is not just a simple link name but a path expression corresponding to the paths between the pair of nodes. Furthermore, as with the graphs, an entry can be a link weight or an expression in link weights (see Chapter 8 for a refresher). Finally, "arithmetic operations" are the operations appropriate to the weights the links represent.

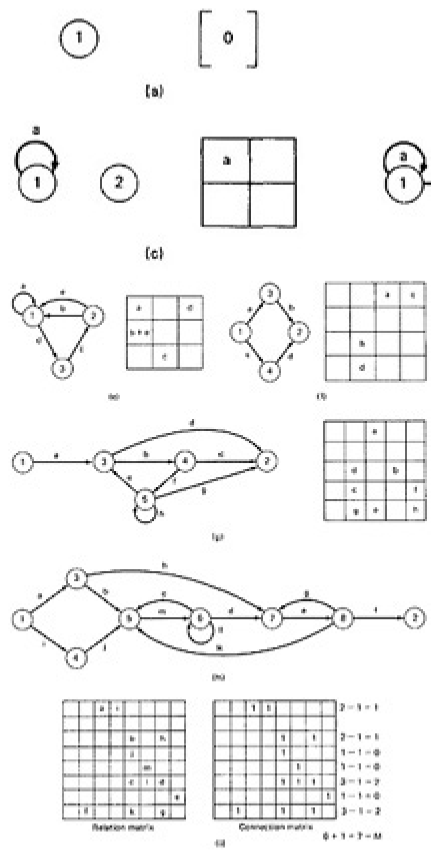


Figure 12.1. Some Graphs and Their Matrices.

3.2. A Simple Weight

The simplest weight we can use is to note that there is or isn't a connection. Let "1" mean that there is a connection and "0" that there isn't. The arithmetic rules are:

$$1 + 1 = 1, \qquad 1 + 0 = 1, \qquad 0 + 0 = 0,$$
$$1 \times 1 = 1, \qquad 1 \times 0 = 0, \qquad 0 \times 0 = 0.$$

A matrix with weights defined like this is called a **connection matrix**. The connection matrix for Figure 12.1h is obtained by replacing each entry with I if there is a link and 0 if there isn't. As usual, to reduce clutter we don't write down 0 entries. Each row of a matrix (whatever the weights) denotes the outlinks of the node corresponding to that row, and each column denotes the inlinks corresponding to that node. A branch node is a node with more than one nonzero entry in its row. A junction node is a node with more than one nonzero entry in its column. A self-loop is an entry along the diagonal. Because rows 1, 3, 6, and 8 of Figure 12.1h all have more than one entry, those nodes are branch nodes. Using the principle that a case statement is equivalent to $n - 1$ binary decisions, by subtracting 1 from the total number of entries in each row and ignoring rows with no entries (such as node 2), we obtain the equivalent number of decisions for each row. Adding these values and then adding I to the sum yields the graph's cyclomatic complexity.

3.3. Further Notation

Talking about the "entry at row 6, column 7" is wordy. To compact things, the entry corresponding to node i and column$j$, which is to say the link weights between nodes i and j, is denoted by a_{ij}. A self-loop about node *i* is denoted by a_{ii}, while the link weight for the link between nodes *j* and *i* is denoted by a_{ji}. The path segments expressed in terms of link names and, in this notation, for several paths in the graph of Figure 12.1h are:

$$abmd = a_{13}a_{35}a_{56}a_{67};$$
$$degef = a_{67}a_{78}a_{87}a_{78}a_{82};$$
$$ahekmlld = a_{13}a_{37}a_{78}a_{85}a_{56}a_{66}a_{66}a_{67};$$

because

$$a_{13} = a,\ a_{35} = b,\ a_{56} = m,\ a_{66} = l,\ a_{67} = d, \text{ etc.}$$

The expression "$a_{ij}a_{jj}a_{jm}$" denotes a path from node ${}_i$ to ${}_j$, with a self-loop at *j* and then a link from node *j* to node *m*. The expression "$a_{ij}a_{jk}a_{km}a_{mi}$" denotes a path from node *i* back to node *i* via nodes *j*, *k*, and *m*. An expression such as "$a_{ik}a_{km}a_{mj} + a_{in}a_{np}a_{pj}$" denotes a pair of paths between nodes *i* and *j*, one going via nodes *k* and *m* and the other via nodes *n* and *p*.

This notation may seem cumbersome, but it's not intended for working with the matrix of a graph but for expressing operations on the matrix. It's a very compact notation. For example,

$$\sum_{k=1}^{n} a_{ik}a_{kk}a_{kj}$$

denotes the set of all possible paths between nodes *i* and *j* via one intermediate node. But because "*i*" and "*j*" denote any node, this expression is the set of all possible paths between any two nodes via one intermediate node.

The **transpose** of a matrix is the matrix with rows and columns interchanged. It is denoted by a superscript letter "T," as in A^T. If $C = A^T$ then $c_{ij} = a_{ji}$. The **intersection** of two matrices of the same size, denoted by A#B is a matrix obtained by an element-by-element multiplication operation on the entries. For example, C = A#B means $c_{ij} = a_{ij}\#b_{ij}$. The multiplication operation is usually boolean AND or set intersection. Similarly, the **union** of two matrices is defined as the element-by-element addition operation such as a boolean OR or set union.

4. RELATIONS

4.1. General

This isn't a section on aunts and uncles but on abstract relations that can exist between abstract objects, although family and personal relations can also be modeled by abstract relations, if you want to. A **relation** is a property that exists between two (usually) objects of interest. We've had many examples of relations in this book. Here's a sample, where *a* and *b* denote objects and R is used to denote that a has the relation R to *b*:

1. "Node *a is connected* to node *b*" or *a*R*b* where "R" means "is connected to."
2. "*a* >= *b*" or *a*R*b* where "R" means "greater than or equal."
3. "*a is a subset* of *b*" where the relation is "is a subset of."

4. "It takes 20 microseconds of processing time to get from node *a* to node *b*." The relation is expressed by the number 20.
5. "Data object X is defined at program node *a* and used at program node *b*." The relation between nodes *a* and *b* is that there is a *du* chain between them.

Let's now redefine what we mean by a graph.

graph consists of a set of abstract objects called **nodes** and a relation R between the nodes. If *a*R*b*, which is to say that a has the relation R to *b*, it is denoted by a **link** from *a* to *b*. In addition to the fact that the relation exists, for some relations we can associate one or more properties. These are called **link weights**. A link weight can be numerical, logical, illogical, objective, subjective, or whatever. Furthermore, there is no limit to the number and type of link weights that one may associate with a relation.

"Is connected to" is just about the simplest relation there is: it is denoted by an unweighted link. Graphs defined over "is connected to" are called, as we said before, **connection matrices**.* For more general relations, the matrix is called a **relation matrix**.

* Also "adjacency matrix"; see EVEN79.

4.2. Properties of Relations

4.2.1. General

The least that we can ask of relations is that there be an algorithm by which we can determine whether or not the relation exists between two nodes. If that's all we ask, then our relation arithmetic is too weak to be useful. The following sections concern some properties of relations that have been found to be useful. Any given relation may or may not have these properties, in almost any combination.

4.2.2. Transitive Relations

A relation R is **transitive** if *a*R*b* and *b*R*c* implies *a*R*c*. Most relations used in testing are transitive. Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the boss of. Examples of **intransitive** relations include: is acquainted with, is a friend of, is a neighbor of, is lied to, has a du chain between.

4.2.3. Reflexive Relations

A relation R is **reflexive** if, for every *a*, *a*R*a*. A reflexive relation is equivalent to a self-loop at every node. Examples of reflexive relations include: equals, is acquainted with (except, perhaps, for amnesiacs), is a relative of. Examples of **irreflexive relations** include: not equals, is a friend of (unfortunately), is on top of, is under.

4.2.4. Symmetric Relations

A relation R is **symmetric** if for every *a* and *b*, *a*R*b* implies *b*R*a*. A symmetric relation means that if there is a link from *a* to *b* then there is also a link from *b* to *a*; which furthermore means that we can do away with arrows and replace the pair of links with a single **undirected** link. A graph whose relations are not symmetric is called a **directed graph** because we must use arrows to denote the relation's direction. A graph over a symmetric relation is called an **undirected graph**.* The matrix of an undirected graph is symmetric ($a_{ij} = a_{ji}$ for all i, j).

* Strictly speaking, we should distinguish between undirected graphs (no arrows) and bidirected graphs (arrow in both directions); but in the context of testing applications, it doesn't matter.

Examples of symmetric relations: is a relative of, equals, is alongside of, shares a room with, is married (usually), is brother of, is similar (in most uses of the word), OR, AND, EXOR. Examples of **asymmetric** relations: is the boss of, is the husband of, is greater than, controls, dominates, can be reached from.

4.2.5. Antisymmetric Relations

A relation R is **antisymmetric** if for every *a* and *b*, if *a*R*b* and *b*R*a*, then *a* = *b*, or they are the same elements.

Examples of antisymmetric relations: is greater than or equal to, is a subset of, time. Examples of **nonantisymmetric** relations: is connected to, can be reached from, is greater than, is a relative of, is a friend of.

4.3. Equivalence Relations

An **equivalence relation** is a relation that satisfies the reflexive, transitive, and symmetric properties. Numerical equality is the most familiar example of an equivalence relation. If a set of objects satisfy an equivalence relation, we say that they form an **equivalence class** over that relation. The importance of equivalence classes and relations is that any member of the equivalence class is, with respect to the relation, equivalent to any other member of that class. The idea behind **partition-testing strategies** such as domain testing and path testing, is that we can partition the input space into equivalence classes. If we can do that, then testing any member of the equivalence class is as effective as testing them all. When we say in path testing that it is sufficient to test one set of input values for each member of a branch-covering set of paths, we are asserting that the set of all input values for each path (e.g., the path's domain) is an equivalence class with respect to the relation that defines branch-testing paths. If we furthermore (incorrectly) assert that a strategy such as branch testing is sufficient, we are asserting that satisfying the branch-testing relation implies that all other possible equivalence relations will also be satisfied—that, of course, is nonsense.

4.4. Partial Ordering Relations

A **partial ordering relation** satisfies the reflexive, transitive, and antisymmetric properties. Partial ordered graphs have several important properties: they are loop-free, there is at least one maximum element, there is at least one minimum element, and if you reverse all the arrows, the resulting graph is also partly ordered. A **maximum element** *a* is one for which the relation *x*R*a* does not hold for any other element *x*. Similarly, a **minimum element** *a*, is one for which the relation *a*R*x* does not hold for any other element *x*. Trees are good examples of partial ordering. The importance of partial ordering is that while strict ordering (as for numbers) is rare with graphs, partial ordering is common. Loop-free graphs are partly ordered. We have many examples of useful partly ordered graphs: call trees, most data structures, an integration plan. Also, whereas the general control-flow or data-flow graph is not always partly ordered, we've seen that by restricting our attention to partly ordered graphs we can sometimes get new, useful strategies. Also, it is often possible to remove the loops from a graph that isn't partly ordered to obtain another graph that is.

5. THE POWERS OF A MATRIX

5.1. Principles

Each entry in the graph's matrix (that is, each link) expresses a relation between the pair of nodes that corresponds to that entry. It is a direct relation, but we are usually interested in indirect relations that exist by virtue of intervening nodes between the two nodes of interest. Squaring the matrix (using suitable arithmetic for the weights) yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that the relation is transitive. The square of the matrix represents all path segments two links long. Similarly, the third power represents all path segments three links long. And the *k*th power of the matrix represents all path segments *k* links long. Because a matrix has at most *n* nodes, and no path can be more than *n* – 1 links long without incorporating some path segment already accounted for, it is generally not necessary to go beyond the *n* – 1 power of the matrix. As usual, concatenation of links or the weights of links is represented by multiplication, and parallel links or path expressions by addition.

Let A be a matrix whose entries are a_{ij}. The set of all paths between any node *i* and any other node*j* (possibly*i*itself), via all possible intermediate nodes, is given by

$$a_{ij} + \sum_{k=1}^{n} a_{ik}a_{kj} + \sum_{k=1}^{n}\sum_{m=1}^{n} a_{ik}a_{km}a_{mj} + \sum_{k}^{n}\sum_{m}^{n}\sum_{l}^{n} a_{ik}a_{km}a_{ml}a_{lj}$$
$$+ \cdots \sum_{k=1}^{n}\sum_{m=1}^{n}\sum_{l=1}^{n} \cdots \sum_{p=1}^{n} a_{ik}a_{km}a_{ml} \cdots a_{qp}a_{pj}$$

As formidable as this expression might appear, it states nothing more than the following:

1. Consider the relation between every node and its neighbor.
2. Extend that relation by considering each neighbor as an intermediate node.
3. Extend further by considering each neighbor's neighbor as an intermediate node.
4. Continue until the longest possible nonrepeating path has been established.
5. Do this for every pair of nodes in the graph.

5.2. Matrix Powers and Products

Given a matrix whose entries are a_{ij}, the square of that matrix is obtained by replacing every entry with

$$a_{ij} = \sum_{k=1}^{n} a_{ik}a_{kj}$$

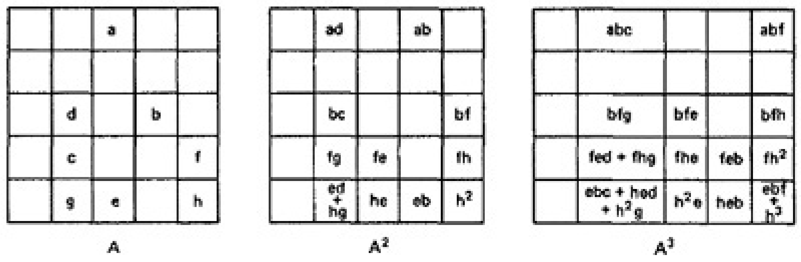
More generally, given two matrices A and B, with entries a_{ik} and b_{kj}, respectively, their product is a new matrix C, whose entries are c_{ij}, where:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{bmatrix} a_{11}a_{12}a_{13}a_{14} \\ a_{21}a_{22}a_{23}a_{24} \\ a_{31}a_{32}a_{33}a_{34} \\ a_{41}a_{42}a_{43}a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11}b_{12}b_{13}b_{14} \\ b_{21}b_{22}b_{23}b_{24} \\ b_{31}b_{32}b_{33}b_{34} \\ b_{41}b_{42}b_{43}b_{44} \end{bmatrix} = \begin{bmatrix} c_{11}c_{12}c_{13}c_{14} \\ c_{21}c_{22}c_{23}c_{24} \\ c_{31}c_{32}c_{33}c_{34} \\ c_{41}c_{42}c_{43}c_{44} \end{bmatrix}$$

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42} \\ c_{13} &= a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} + a_{14}b_{43} \\ &\cdots \\ &\cdots \\ &\cdots \\ c_{32} &= a_{31}b_{12} + a_{32}b_{22} .+ a_{33}b_{32} + a_{34}b_{42} \\ &\cdots \\ &\cdots \\ &\cdots \\ c_{44} &= a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44} \end{aligned}$$

The indexes of the product [e.g., (3,2) in C_{32}] identify, respectively, the row of the first matrix and the column of the second matrix that will be combined to yield the entry for that product in the product matrix. The C_{32} entry is obtained by combining, element by element, the entries in the third row of the A matrix with the corresponding elements in the second column of the B matrix. I use two hands. My left hand points and traces across the row while the right points down the column of B. It's like patting your head with one hand and rubbing your stomach with the other at the same time: it takes practice to get the hang of it. Applying this to the matrix of Figure 12.1g yields



		a		
	d		b	
	c			f
	g	e		h

A

	ad		ab	
	bc			bf
	fg	fe		fh
	ed + hg	hc	eb	h^2

A^2

	abc			abf
	bfg	bfe		bfh
	fed + fhg	fhe	feb	fh^2
	ebc + hed + h^2g	h^2e	heb	ebf + h^3

A^3

$A^2A = AA^2$; that is, matrix multiplication is associative (for most interesting relations) if the underlying relation arithmetic is associative. Therefore, you can get A^4 in any of the following ways: A^2A^2, $(A^2)^2$, A^3A, AA^3. However, because multiplication is not necessarily commutative, you must remember to put the contribution of the left-hand matrix in front of the contribution of the right-hand matrix and not inadvertently reverse the order. The loop terms are important. These are the terms that appear along the **principal diagonal** (the one that slants down to the right). The initial matrix had a self-loop about node 5, link *h*. No new loop is revealed with paths of length 2, but the cube of the matrix shows additional loops about nodes 3 (*bfe*), 4 (*feb*), and 5 (*ebf*). It's clear that these are the same loop around the three nodes.

If instead of link names you use some other relation and associated weight, as in Chapter 8, and use the appropriate arithmetic rules, the matrix displays the property corresponding to that relation. Successive powers of the matrix display the property when considering paths of length exactly 2, exactly 3, and so on. The methods of Chapter 8 and the applications discussed there carry over almost unchanged into equivalent matrix methods.

5.3. The Set of All Paths

Our main objective is to use matrix operations to obtain the set of all paths between all nodes or, equivalently, a property (described by link weights) over the set of all paths from every node to every other node, using the appropriate arithmetic rules for such weights. The set of all paths between all nodes is easily expressed in terms of matrix operations. It's given by the following infinite series of matrix powers:

$$\sum_{i=1}^{\infty} A^i = A + A^2 + A^3 \cdots A^{\infty}$$

This is an eloquent, but practically useless, expression. Let I be an *n* by *n* matrix, where *n* is the number of nodes. Let I's entries consist of multiplicative identity elements along the principal diagonal. For link names, this can be the number "1." For other kinds of weights, it is the multiplicative identity for those weights. The above product can be re-phrased as:

$$A(I + A + A^2 + A^3 + A^4 \ldots A^{\infty})$$

But often for relations, $A + A = A$, $(A + I)^2 = A^2 + A + A + I\ A^2 + A + I$. Furthermore, for any finite n,

$$(A + I)^n = I + A + A^2 + A^3 \ldots A^n$$

Therefore, the original infinite sum can be replaced by

$$\sum_{i=1}^{\infty} A^i = A(A + I)^{\infty}$$

This is an improvement, because in the original expression we had both infinite products and infinite sums, and now we have only one infinite product to contend with. The above is valid whether or not there are loops. If we restrict our interest for the moment to paths of length $n – 1$, where *n* is the number of nodes, the set of all such paths is given by

$$\sum_{i=1}^{n-1} A^i = A(A + I)^{n-2}$$

This is an interesting set of paths because, with *n* nodes, no path can exceed $n – 1$ nodes without incorporating some path segment that is already incorporated in some other path or path segment. Finding the set of all such paths is somewhat easier because it is not necessary to do all the intermediate products explicitly. The following algorithm is effective:

1. Express $n – 2$ as a binary number.
2. Take successive squares of (A + I), leading to $(A + I)^2$, $(A + I)^4$, $(A + 1)^8$, and so on.
3. Keep only those binary powers of (A + 1) that correspond to a 1 value in the binary representation of $n – 2$.
4. The set of all paths of length $n – 1$ or less is obtained as the product of the matrices you got in step 3 with the original matrix.

As an example, let the graph have 16 nodes. We want the set of all paths of length less than or equal to 15. The binary representation of $n – 2$ (14) is $2^3 + 2^2 + 2$. Consequently, the set of paths is given by

$$\sum_{i=1}^{15} A^i = A(A + I)^8(A + I)^4(A + I)^2$$

This required one multiplication to get the square, squaring that to get the fourth power, and squaring again to get the eighth power, then three more multiplications to get the sum, for a total of six matrix multiplications without additions, compared to fourteen multiplications and additions if gotten directly.

A matrix for which $A^2 = A$ is said to be **idempotent**. A matrix whose successive powers eventually yields an idempotent matrix is called an **idempotent generator**—that is, a matrix for which there is a k such that $A^{k+1} = A^k$. The point about idempotent generator matrices is that we can get properties over all paths by successive squaring. A graph matrix of the form (A + I) over a transitive relation is an idempotent generator; therefore, anything of interest can be obtained by even simpler means than the binary method discussed above. For example, the relation "connected" does not change once we reach A^{n-1} because no connection can take more than $n - 1$ links and, once connected, nodes cannot be disconnected. Thus, if we wanted to know which nodes of an n-node graph were connected to which, by whatever paths, we have only to calculate: A^2, $A^2A^2 = A^4 \ldots A^P$, where p is the next power of 2 greater than or equal to n. We can do this because the relation "is connected to" is reflexive and transitive. The fact that it is reflexive means that every node has the equivalent of a self-loop, and the matrix is therefore an idempotent generator. If a relation is transitive but not reflexive, we can augment it as we did above, by adding the unit matrix to it, thereby making it reflexive. That is, although the relation defined over A is not reflexive, A + I is. A + I is an idempotent generator, and therefore there's nothing new to learn for powers greater than $n - 1$, the length of the longest nonrepeating path through the graph. The nth power of a matrix A + I over a transitive relation is called the **transitive closure** of the matrix.

5.4. Loops

Every loop forces us into a potentially infinite sum of matrix powers. The way to handle loops is similar to what we did for regular expressions. Every loop shows up as a term in the diagonal of some power of the matrix—the power at which the loop finally closes—or, equivalently, the length of the loop. The impact of the loop can be obtained by preceding every element in the row of the node at which the loop occurs by the path expression of the loop term starred and then deleting the loop term. For example, using the matrix for the graph of Figure 12.1e, we obtain the following succession of powers for A + I:

The first matrix (A + I) had a self-loop about node 5 link h. Moving link h out to the other entries in the row, leaving the "1" entry at the (5,5) position, yielded the h^*g and the h^*e entries at (5,2) and (5,3) respectively. No new loops were closed for the second power. The third-power matrix has a loop about node 3, whose expression is bfh^*e. Consequently, all other entries in that row are premultiplied by $(bfh^*e)^*$, to yield $(bfh^*e)^*(d + bc + bfh^*g)$ for (3,2), $(bfh^*e)^*b$ for (3,4), and $(bfh^*e)^*bf$ for (3,5). Similarly, the fh^*eb term in the (4,4) entry is removed by multiplying every other nonzero term in the fourth row by $(fh^*eb)^*$, and the elements in the fifth row is multiplied by $(h^*ebf)^*$ to get rid of the loop.

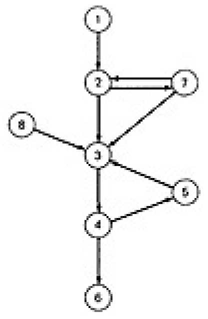
Applying this method of characterizing all possible paths is straightforward. The above operations are interpreted in terms of the arithmetic appropriate to the weights used. Note, however, that if you are working with predicates and you want the logical function (predicate function, truth-value function) between every node and every other node, this may lead to loops in the logical functions. The specific "arithmetic" for handling predicate loops has not been discussed in this book. The information can be found in any good text on switching and automata theory, such as MILL66. Code that leads to predicate loops is not very nice, not well structured, hard to understand, and harder to test—and anyone who codes that way deserves the analytical difficulties arising therefrom. Predicate loops come about from declared or undeclared program switches and/or unstructured loop constructs. This means that the routine's code remembers. If you didn't realize that you put such a loop in, you probably didn't intend to. If you did intend it, you should have expected the loop.

5.5. Partitioning Algorithm (BEIZ71, SOHO84)

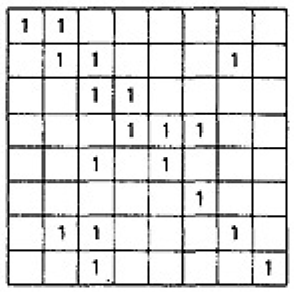
Consider any graph over a transitive relation. The graph may have loops. We would like to partition the graph by grouping nodes in such a way that every loop is contained within one group or another. Such a graph is partly ordered. There are many used for an algorithm that does that:

1. We might want to embed the loops within a subroutine so as to have a resulting graph which is loop-free at the top level.
2. Many graphs with loops are easy to analyze if you know where to break the loops.
3. While you and I can recognize loops, it's much harder to program a tool to do it unless you have a solid algorithm on which to base the tool.

The way to do this is straightforward. Calculate the following matrix: $(A + I)^n \# (A + I)^{nT}$. This groups the nodes into strongly connected sets of nodes such that the sets are partly ordered. Furthermore, every such set is an equivalence class so that any one node in it represents the set. Now consider all the places in this book where we said "except for graphs with loops" or "assume a loop-free graph" or words to that effect. If you can bury the loop in a real subroutine, you can as easily bury it in a conceptual subroutine. Do the analysis over the partly ordered graph obtained by the partitioning algorithm and treat each loop-connected node set as if it is a subroutine to be examined in detail later. For each such component, break the loop and repeat the process. You now have a divide-and-conquer approach for handling loops. Here's an example, worked with an arbitrary graph:

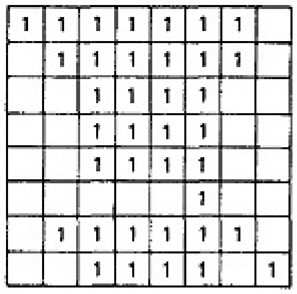


The relation matrix is



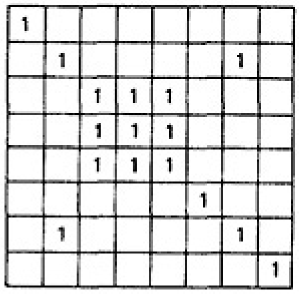
1	1						
	1	1				1	
		1	1				
			1	1	1		
		1		1			
					1		
	1	1				1	
		1					1

The transitive closure matrix is



1	1	1	1	1	1	1	
	1	1	1	1	1	1	
		1	1	1	1		
		1	1	1	1		
		1	1	1	1		
					1		
	1	1	1	1	1	1	
		1	1	1	1		1

Intersection with its transpose yields

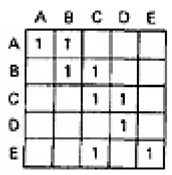


1							
	1					1	
		1	1	1			
		1	1	1			
		1	1	1			
					1		
	1					1	
							1

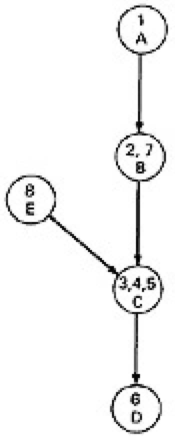
You can recognize equivalent nodes by simply picking a row (or column) and searching the matrix for identical rows. Mark the nodes that match the pattern as you go and eliminate that row. Then start again from the top with another row and another pattern. Eventually, all rows have been grouped. The algorithm leads to the following equivalent node sets:

A = [1]
B = [2,7]
C = [3,4,5]
D = [6]
E = [8]

whose graph is



	A	B	C	D	E
A	1	1			
B		1	1		
C			1	1	
D				1	
E			1		1



5.6. Breaking Loops And Applications

And how do you find the point at which to break the loops, you ask? Easy. Consider the matrix of a strongly connected subgraph. If there are entries on the principal diagonal, then start by breaking the loop for those links. Now consider successive powers of the matrix. At some power or another, a loop is manifested as an entry on the principal diagonal. Furthermore, the regular expression over the link names that appears in the diagonal entry tells you all the places you can or must break the loop. Another way is to apply the node-reduction algorithm (see below), which will also display the loops and therefore the desired break points.

The divide-and-conquer, or rather partition-and-conquer, properties of the equivalence partitioning algorithm is a basis for implementing tools. The problem with most algorithms is that they are computationally intensive and require of the order of n^2 or n^3 arithmetic operations, where n is the number of nodes. Even with fast, cheap computers it's hard to keep up with such growth laws. The key to solving big problems (hundreds of nodes) is to partition them into a hierarchy of smaller problems. If you can go far enough, you can achieve processing of the order of n, which is fine. The partition algorithm makes graphs into trees, which are relatively easy to handle.

6. NODE-REDUCTION ALGORITHM

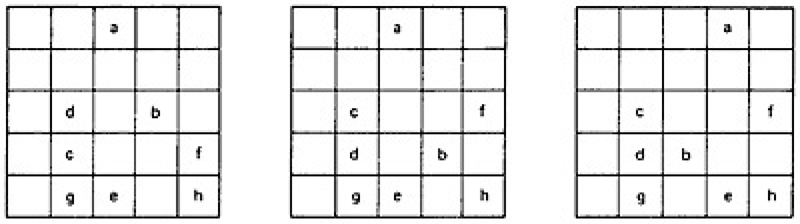
6.1. General

The matrix powers usually tell us more than we want to know about most graphs. In the context of testing, we're usually interested in establishing a relation between two nodes — typically the entry and exit nodes—rather than between every node and every other node. In a debugging context it is unlikely that we would want to know the path expression between every node and every other node; there also, it is the path expression or some other related expression between a specific pair of nodes that is sought: for example, "How did I get *here* from *there*?" The method of this section is a matrix equivalence to the node-by-node reduction procedure of Chapter 8. The advantage of the matrix-reduction method is that it is more methodical than the graphical method of Chapter 8 and does not entail continually redrawing the graph. It's done as follows:

1. Select a node for removal; replace the node by equivalent links that bypass that node and add those links to the links they parallel.
2. Combine the parallel terms and simplify as you can.
3. Observe loop terms and adjust the outlinks of every node that had a self-loop to account for the effect of the loop.
4. The result is a matrix whose size has been reduced by 1. Continue until only the two nodes of interest exist.

6.2. Some Matrix Properties

If you numbered the nodes of a graph from 1 to n, you would not expect that the behavior of the graph or the program that it represents would change if you happened to number the nodes differently. Node numbering is arbitrary and cannot affect anything. The equivalent to renumbering the nodes of a graph is to interchange the rows and columns of the corresponding matrix. Say that you wanted to change the names of nodes i and j to j and i, respectively. You would do this on the graph by erasing the names and rewriting them. To interchange node names in the matrix, you must interchange both the corresponding rows and the corresponding columns. Interchanging the names of nodes 3 and 4 in the graph of Figure 12.1g results in the following:



		a		
	d		b	
	c			f
	g	e		h

a) Original

		a		
	c			f
	d		b	
	g	e		h

b) Rows 3 and 4 interchanged

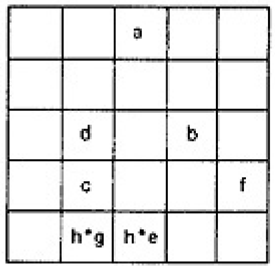
			a	
	c			f
	d	b		
	g		e	h

c) Interchange complete

If you redraw the graph based on c, you will see that it is identical to the original except that node 3's name has been changed to 4, and node 4's name to 3.

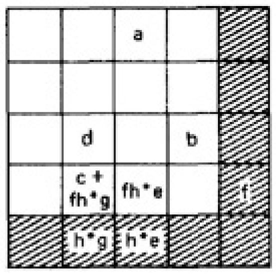
6.3. The Algorithm

The first step is the most complicated one: eliminating a node and replacing it with a set of equivalent links. Using the example of Figure 12.1g, we must first remove the self-loop at node 5. This produces the following matrix:



		a		
	d		b	
	c			f
	h*g	h*e		

The reduction is done one node at a time by combining the elements in the last column with the elements in the last row and putting the result into the entry at the corresponding intersection. In the above case, the *f* in column 5 is first combined with *h*g* in column 2, and the result (*fh*g*) is added to the *c* term just above it. Similarly, the *f* is combined with *h*e* in column 3 and put into the 4,3 entry just above it. The justification for this operation is that the column entry specifies the links entering the node, whereas the row specifies the links leaving the node. Combining every column entry with the corresponding row entries for that node produces exactly the same result as the node-elimination step in the graphical-reduction procedure. What we did was: $a_{45}a_{52} = a_{42}$ or $f \times h^*g = a_{52}$, but because there was already a *c* term there, we have effectively created a parallel link in the (5,2) position leading to the complete term of $c + fh^*g$. The matrix resulting from this step is



		a		
	d		b	
	c + fh*g	fh*e		f
	h*g	h*e		

If any loop terms had occurred at this point, they would have been taken care of by eliminating the loop term and premultiplying every term in that row by the loop term starred. There are no loop terms at this point. The next node to be removed is node 4. The *b* term in the (3,4) position will combine with the (4,2) and (4,3) terms to yield a (3,2) and a (3,3) term, respectively. Carrying this out and discarding the unnecessary rows and columns yields

		a
	d + bc + bfh*g	bfh*e

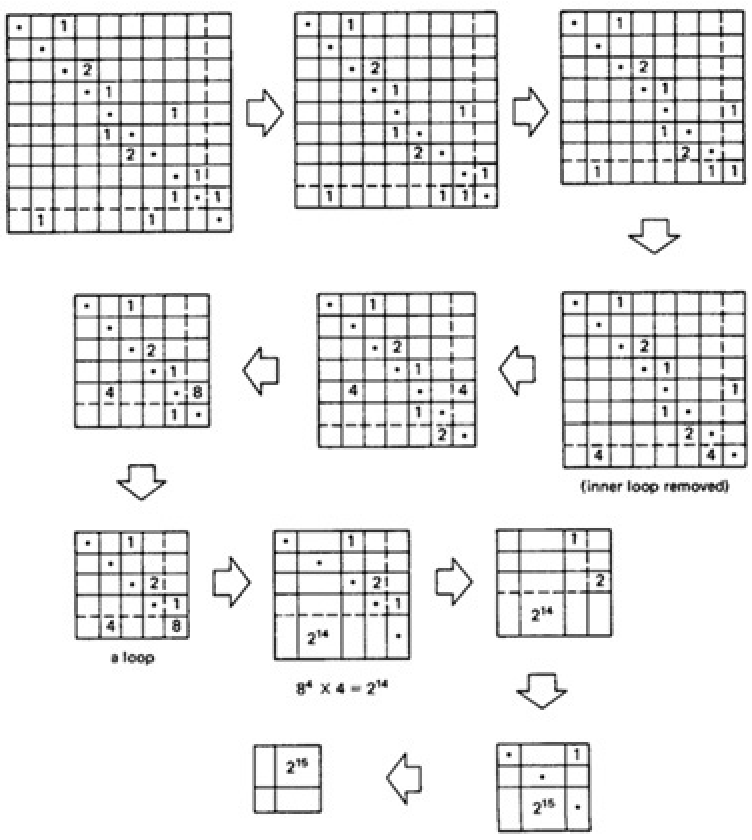
Removing the loop term yields

		a
	(bfh*e)* X (d + bc + bfh*g)	

There is only one node to remove now, node 3. This will result in a term in the (1,2) entry whose value is

$$a(bfh^*e)^*(d + bc + bfh^*g)$$

This is the path expression from node 1 to node 2. Stare at this one for awhile before you object to the (*bfh***e*)* term that multiplies the *d*; any fool can see the direct path via *d* from node 1 to the exit, but you could miss the fact that the routine could circulate around nodes 3, 4, and 5 before it finally took the *d* link to node 2.



6.4. Applications

6.4.1. General

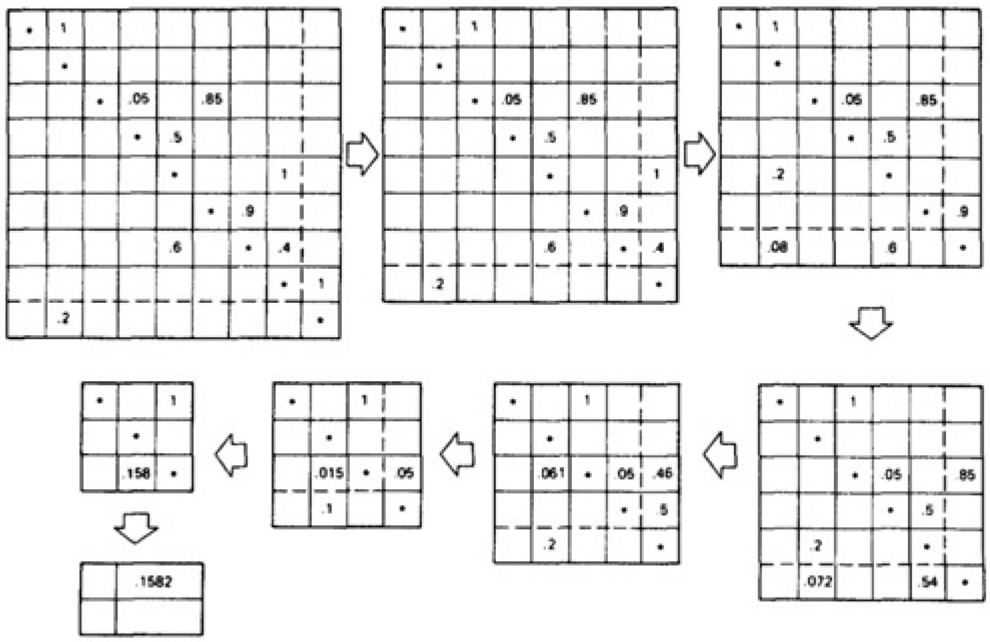
The path expression is usually the most difficult and complicated to get. The arithmetic rules for most applications are simpler. In this section we'll redo applications from Chapter 8, using the appropriate arithmetic rules, but this time using matrices rather than graphs. Refer back to the corresponding examples in Chapter 8 to follow the successive stages of the analysis.

6.4.2. Maximum Number of Paths

The matrix corresponding to the graph on page 261 is on the opposite page. The successive steps are shown. Recall that the inner loop about nodes 8 and 9 was to be taken from zero to three times, while the outer loop about nodes 5 and 10 was to be taken exactly four times. This will affect the way the diagonal loop terms are handled.

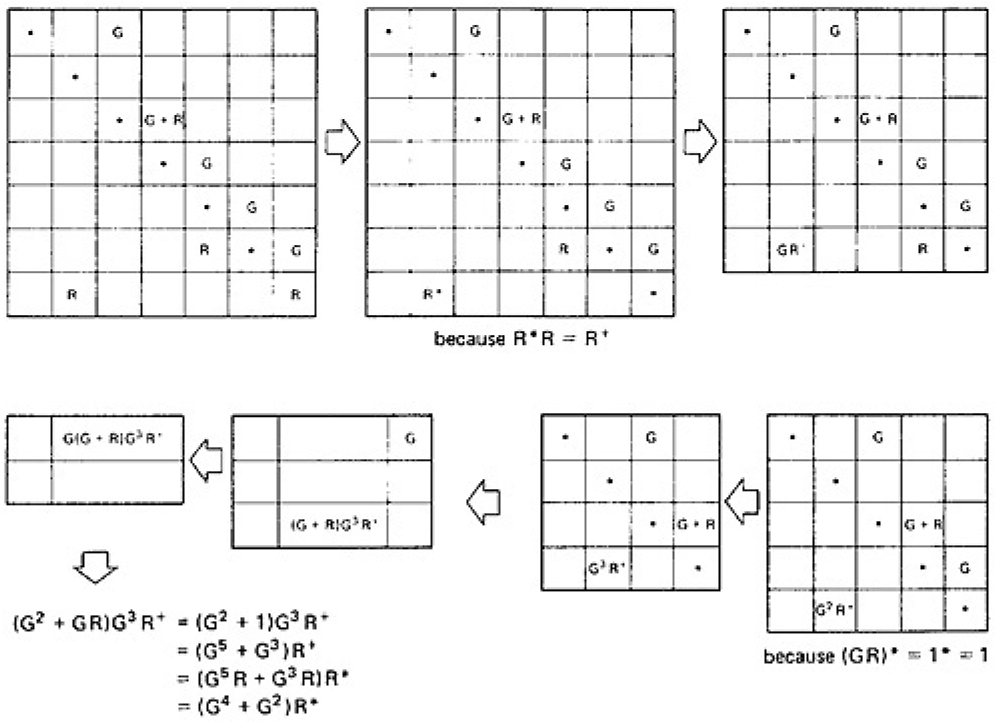
6.4.3. The Probability of Getting There

A matrix representation for the probability problem on page 268 is



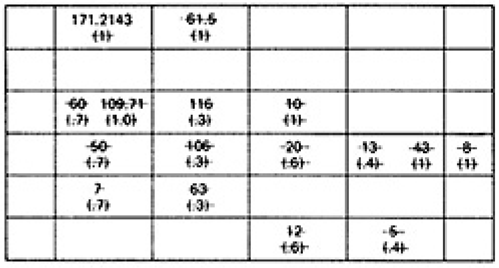
6.4.4. Get/Return Problem

The GET/RETURN problem on page 276 has the following matrix reduction:



6.5. Some Hints

Redrawing the matrix over and over again is as bad as redrawing the graph to which it corresponds. You actually do the work in place. Things get more complicated, and expressions get bigger as you progress, so you make the low-numbered boxes larger than the boxes corresponding to higher-numbered nodes, because those are the ones that are going to be removed first. Mark the diagonal lightly so that you can easily see the loops. With these points in mind, the work sheet for the timing analysis graph on page 272 looks like this:



	171.2143 ~~(1)~~	~~61.5~~ ~~(1)~~			
	~~60~~ ~~109.71~~ ~~(.7)~~ ~~(1.0)~~	~~116~~ ~~(.3)~~	~~10~~ ~~(1)~~		
	~~50~~ ~~(.7)~~	~~106~~ ~~(.3)~~	~~20~~ ~~(.6)~~	~~13~~ ~~43~~ ~~(.4)~~ ~~(1)~~	~~8~~ ~~(1)~~
	~~7~~ ~~(.7)~~	~~63~~ ~~(.3)~~			
			~~12~~ ~~(.6)~~	~~5~~ ~~(.4)~~	

7. BUILDING TOOLS

7.1. Matrix Representation Software

7.1.1. Overview

We draw graphs or display them on screens as visual objects; we prove theorems and develop graph algorithms by using matrices; and when we want to process graphs in a computer, because we're building tools, we represent them as linked lists. We use linked lists because graph matrices are usually very sparse; that is, the rows and columns are mostly empty.

7.1.2. Node Degree and Graph Density

The **out-degree** of a node is the number of outlinks it has. The **in-degree** of a node is the number of inlinks it has. The **degree** of a node is the sum of the out-degree and in-degree. The average degree of a node (the mean over all nodes) for a typical graph defined over software is between 3 and 4. The degree of a simple branch is 3, as is the degree of a simple junction. The degree of a loop, if wholly contained in one statement, is only 4. A mean node degree of 5 or 6 say, would be a very busy flowgraph indeed.

7.1.3. What's Wrong with Arrays?

We can represent the matrix as a two-dimensional array for small graphs with simple weights, but this is not convenient for larger graphs because:

1. *Space*—Space grows as n^2 for the matrix representation, but for a linked list only as kn, where k is a small number such as 3 or 4.
2. *Weights*—Most weights are complicated and can have several components. That would require an additional weight matrix for each such weight.
3. *Variable-Length Weights*—If the weights are regular expressions, say, or algebraic expressions (which is what we need for a timing analyzer), then we need a two-dimensional string array, most of whose entries would be null.
4. *Processing Time*—Even though operations over null entries are fast, it still takes time to access such entries and discard them. The matrix representation forces us to spend a lot of time processing combinations of entries that we know will yield null results.

The matrix representation is useful in building prototype tools based on untried algorithms. It's a lot easier to implement algorithms by using direct matrix representations, especially if you have matrix manipulation subroutines or library functions. Matrices are reasonable to about 20–30 nodes, which is good enough for testing most prototype tools.

7.1.4. Linked-List Representation

Give every node a unique name or number. A link is a pair of node names. The linked list for Figure 12.1g on page 400 is:

1,3;*a*
2,
3,2;*d*
3,4;*b*
4,2;*c*
4,5;*f*
5,2;*g*
5,3;*e*
5,5;*h*

Note that I've put the list entries in lexicographic order. The link names will usually be pointers to entries in a string array Where the actual link weight expressions are stored. If the weights are fixed length then they can be associated directly with the links in a parallel, fixed entry-length array. Let's clarify the notation a bit by using node names and pointers.

List Entry	*Content*
1	node1,3;*a*
2	node2,exit

3	node3,2;*d*
	,4;*b*
4	node4,2;*c*
	,5;*f*
5	node5,2;*g*
	,3;*e*
	,5;*h*

The node names appear only once, at the first link entry. Also, instead of naming the other end of the link, we have just the pointer to the list position in which that node starts. Finally, it is also very useful to have back pointers for the inlinks. Doing this we get

List Entry	*Content*
1	node1,3;*a*
2	node2,exit
	3,
	4,
	5,
3	node3,2;*d*
	,4;*b*
	1,
	5,
4	node4,2;*c*
	,5;*f*
5	3,
	node5,2;*g*
	,3;*e*
	,5;*h*
	4,
	5,

It's important to keep the lists sorted in lexicographic ordering with the following priorities: node names or pointers, outlink names or pointers, inlink names or pointers. Because the various operations will result in nearly sorted lists, a sort algorithm, such as stringsort, that's optimum for such lists is an essential subroutine.

7.2. Matrix Operations

7.2.1. Parallel Reduction

This is the easiest operation. Parallel links after sorting are adjacent entries with the same pair of node names. For example:

node 17,21;*x*
,44;*y*

,44;*z*
,44;*w*

We have three parallel links from node 17 to node 44. We fetch the weight expressions using the *y*, *z*, and *w* pointers and we obtain a new link that is their sum:

node17,21;*x*
,44;*y* (where *y* = *y* + *z* + *w*).

7.2.2. Loop Reduction

Loop reduction is almost as easy. A loop term is spotted as a self-link. The effect of the loop must be applied to all the outlinks of the node. Scan the link list for the node to find the loop(s). Apply the loop calculation to every outlink, except another loop. Remove that loop. Repeat for all loops about that node. Repeat for all nodes. For example removing node 5's loop:

List Entry	*Content*	*Content After*
5	node5,2;g →	node5,2;*h***g*
	,3;*e* →	,3;*h***e*
	,5;*h* →	
	4,	4,
	5,	

7.2.3. Cross-Term Reduction

Select a node for reduction (see Section 7.3 below for strategies). The cross-term step requires that you combine every inlink to the node with every outlink from that node. The outlinks are associated with the node you've selected. The inlinks are obtained by using the back pointers. The new links created by removing the node will be associated with the nodes of the inlinks. Say that the node to be removed was node 4.

List Entry	*Content Before*	
2	node2,exit	node2,exit
	3,	3,
	4,	4,
	5,	5,
3	node3,2;*d*	node3,2;*d*
	,4;*b*	,2;*bc*
		,5;*bf*
	1,	1,
	5,	5,
4	node4,2;*c*	
	,5;*f*	
	3,	
5	node5,2;*h***g*	node5,2;*h***g*

,3;*h***e* ,3;*h***e*
4,

As implemented, you can remove several nodes in one pass if you do careful bookkeeping and keep your pointers straight. The links created by node removal are stored in a separate list which is then sorted and thereafter merged into the master list.

7.2.4 Addition, Multiplication, and Other Operations

Addition of two matrices is straightforward. If you keep the lists sorted, then simply merge the lists and combine parallel entries.

Multiplication is more complicated but also straightforward. You have to beat the node's outlinks against the list's inlinks. It can be done in place, but it's easier to create a new list. Again, the list will be in sorted order and you use parallel combination to do the addition and to compact the list.

Transposition is done by reversing the pointer directions, resulting in a list that is not correctly sorted. Sorting that list provides the transpose. All other matrix operations can be easily implemented by sorting, merging, and combining parallels.

7.3. Node-Reduction Optimization

The optimum order for node reduction is to do lowest-degree nodes first. The idea is to get the lists as short as possible as quickly as possible. Nodes of degree 3 (one in and two out or two in and one out) reduce the total link count by one link when removed. A degree-4 node keeps the link count the same, and all higher-degree nodes increase the link count. Although this is not guaranteed, by picking the lowest-degree node available for reduction you can almost prevent unlimited list growth. Because processing is dominated by list length rather than by the number of nodes on the list, this strategy is effective. For large graphs with 500 or more nodes and an average degree of 6 or 7, the difference between not optimizing the node-reduction order and optimizing it was about 50: 1 in processing time.

8. Summary

1. Working with pictorial graphs is tedious and a waste of time. Graph matrices are used to organize the work.
2. The graph matrix is the tool of choice for proving things about graphs and for developing algorithms.
3. As implemented in tools, graph matrices are usually represented as linked lists.
4. Most testing problems can be recast into an equivalent problem about some graph whose links have one or more weights and for which there is a problem—specific arithmetic over the link weights. The link-weighted graph is represented by a relation matrix.
5. Relations as abstract operators are well understood and have interesting properties which can be exploited to create efficient algorithms. Properties of interest include transitivity, reflexivity, symmetry, asymmetry, and antisymmetry. These properties in various combinations define ordering, partial ordering, and equivalence relations.
6. The powers of a relation matrix define relations spanning one, two, three, and up to the maximum number of links that can be included in a path. The powers of the matrix are the primary tool for finding properties that relate any node to any other node.
7. The transitive closure of the matrix is used to define equivalence classes and to convert an arbitrary graph into a partly ordered graph.

8. The node reduction algorithm first presented in Chapter 8 is redefined in terms of matrix operations.
9. There is an old and copious literature on graph matrices and associated algorithms. Serious tool builders should learn that literature lest they waste time reinventing ancient algorithms or on marginally useful heuristics.

<ch11 toc ch13>