Graph Matrices and Applications: Motivational Overview

The Problem with Pictorial Graphs

Graphs were introduced as an abstraction of software structure. There are many other kinds of graphs 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.

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 was graphical and it's hard to build algorithms over visual graphs. The properties of graph matrices are fundamental to test tool building.

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 will be blind too much of testing theory, especially those parts that lead to useful algorithms.

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. 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.

The Matrix of a Graph

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. 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.

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 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.

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 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

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

Relations

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.

Properties of Relations

General

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.

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.

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.

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)

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.

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*.

The Powers of a Matrix

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.

Matrix Powers and Products

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

The indexes of the product $[C_{32}]$ 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 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: 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,

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

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.

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.

Breaking Loops and Applications

How do you find the point at which to break the loops? 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.

Node-Reduction Algorithm (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.

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.

The first step in the Node-Reduction Algorithm is the most complicated one: eliminating a node and replacing it with a set of equivalent links. 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.

Applications

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, using the appropriate arithmetic rules, but this time using matrices rather than graphs. Refer back to the corresponding examples in to follow the successive stages of the analysis.

Building Tools: Matrix Representation Software

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.

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.

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.

Linked-List Representation

Give every node a unique name or number. A link is a pair of node names. The linked list for:

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,

Matrix Operations

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 $y = y + z + w$).

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,	

Cross-Term Reduction

Select a node for reduction. 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 have 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,	

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 outlines against the list's inlinks.

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.