

[<ch7](#) [toc](#) [ch9>](#)

Chapter 8

PATHS, PATH PRODUCTS, AND REGULAR EXPRESSIONS

1. SYNOPSIS

Path expressions are introduced as an algebraic representations of sets of paths in a graph. With suitable arithmetic laws (BRZO62A, BRZO62B, BRZO63, MCNA60, PRAT83) and weights, path expressions are converted into algebraic functions or **regular expressions** that can be used to examine structural properties of flowgraphs, such as the number of paths, processing time, or whether a data-flow anomaly can occur. These expressions are then applied to problems in test design and debugging.

2. MOTIVATION

This chapter and its continuation, [Chapter 12](#), are the two most abstract chapters in this book; but that doesn't mean that they're difficult. Considering the generally pragmatic orientation of this book, some motivation for this abstraction is warranted. I could be high-handed and patronizing and say: "Trust me, it's good for you," but you're too intelligent to let me get away with that. This stuff *is* good for you and I do want you to trust me about it; but I would rather you take the effort needed to master the subject and then to see all the nice ways it can be applied. So here's some motivation for you.

1. I introduced flowgraphs as an abstract representation of programs. I assume that by now you appreciate their utility and want to know how playing with flowgraphs is really done.
2. Almost any question you have about programs can be cast into an equivalent question about an appropriate flowgraph. This chapter tells you how to do that.
3. The concepts discussed in this chapter will help you better understand and exploit syntax testing ([Chapter 9](#)) and state testing ([Chapter 11](#)), both of which yield applicable test methods.
4. EE's have been using flowgraphs to design and analyze circuits for more than 50 years (MAYE72) and logic designers for more than 30 years. Get on the bandwagon.
5. Most software development, testing, and debugging tools use flowgraph analysis techniques (or should). You can better exploit your tools if you understand them.
6. If you're a test tool builder, I don't know how you can get along without these concepts.

3. PATH PRODUCTS AND PATH EXPRESSIONS

3.1. Overview

Our flowgraphs ([Chapter 3](#)) denoted only control-flow connectivity; that is, links had no property other than the fact that they connected nodes. In [Chapter 5](#), on data-flow testing, we expanded the notation to add **link weights**; that is, every link was annotated with a letter or sequence of letters that denoted the sequence of data-flow actions on that link. We'll now generalize the idea of link weights so that they can denote almost anything of interest that can be derived from the program's structure.

The simplest weight we can give to a link is a name. Using link names as weights, we then convert the graphical flowgraph into an equivalent algebraic-like expression which denotes the set of all possible paths (finite or infinite) from entry to exit for that flowgraph. The same basic algorithm is then used in subsequent sections with different kinds of weights to do data-flow anomaly detection, timing analyses, and to solve various debugging and testing problems.

3.2. Basic Concepts

Every link of a graph can be given a name; the link name will be denoted by lowercase italic letters. In tracing a path or path segment through a flowgraph, you traverse a succession of link names. The name of the path or path segment that corresponds to those links is expressed naturally by concatenating those link names. If you traverse links *a*, *b*, *c*, and *d* along some path, the name for that path segment is *abcd*. This path name is also called a **path product**. [Figure 8.1](#) shows some examples.

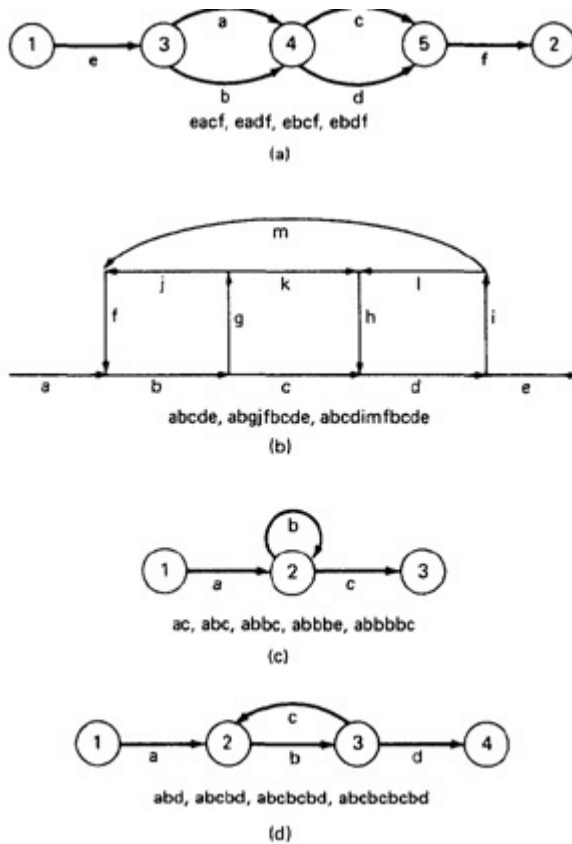


Figure 8.1. Examples of Paths.

Consider a pair of nodes in a graph and the set of paths between those nodes. Denote that set of paths by uppercase letters such as X or Y. The members of that set can be listed as follows for [Figure 8.1c](#):

ac, abc, abbc, abbbc, . . .

Alternatively, that same set of paths can be denoted by:

ac + abc + abbc + abbbc + . . .

The “+” sign is understood to mean “or.” That is, between the two nodes of interest, paths *ac*, or *abc*, or *abbc*, and so on can be taken. Any expression that consists of path names and “ORs” and which denotes a set of paths (not necessarily the set of all paths) between two nodes is called a **path expression**.

3.3. Path Products

The name of a path that consists of two successive path segments is conveniently expressed by the concatenation or **path product** of the segment names. For example, if X and Y are defined as

$$X = abcde$$

$$Y = fghij$$

then the path corresponding to X followed by Y is denoted by

$$XY = abcdefghij$$

Similarly,

$$YX = fghijabcde$$

$$aX = aabcde$$

$$Xa = abcdea$$

$$XaX = abcdeaabcde$$

If X and Y represent sets of paths or path expressions, their product represents the set of paths that can be obtained by following every element of X by any element of Y in all possible ways. For example,

$$X = abc + def + ghi$$

$$Y = uvw + z$$

Then

$$XY = abcuvw + defuvw + ghiuvw + abcz + defz + ghiz$$

If a link or segment name is repeated, that fact is denoted by an exponent. The exponent's value denotes the number of repetitions:

$$a^1 = a; a^2 = aa; a^3 = aaa; a^n = aaaa \dots n \text{ times.}$$

Similarly, if

$$X = abcde$$

then

$$X^1 = abcde$$

$$X^2 = abcdeabcde = (abcde)^2$$

$$X^3 = abcdeabcdeabcde = (abcde)^2 abcde$$

$$= abcde(abcde)^2 = (abcde)^3$$

The path product is not commutative (that is, XY does not necessarily equal YX), but expressions derived from it may be commutative. The path product is associative, but expressions derived from it may not be; that is,

$$\text{Rule 1: } A(BC) = (AB)C = ABC$$

where A, B, and C are path names, sets of path names, or path expressions.

The zeroth power of a link name, path product, or path expression is also needed for completeness. It is

denoted by the numeral “1” and denotes the “path” whose length is zero—that is, the path that doesn’t have any links.* 1 is a multiplicative identity element; that is, it has the same properties as the number 1 in ordinary arithmetic.

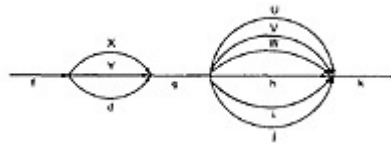
*The first edition and the literature on regular expressions use the Greek letter lambda (λ) for this. I’ve switched to an ordinary “1” to simplify the notation and to make it more familiar.

$$a^0 = 1$$

$$X^0 = 1$$

3.4. Path Sums

The “+” sign was used to denote the fact that path names were part of the same set of paths. Consider the set of paths that can lie between two arbitrary nodes of a graph. Even though these paths can traverse intermediate nodes, they can be thought of as “parallel” paths between the two nodes. The **path sum** denotes paths in parallel between two nodes. Links a and b in [Figure 8.1a](#) are parallel paths and are denoted by $a + b$. Similarly, links c and d are parallel paths between the next two nodes and are denoted by $c + d$. The set of all paths between nodes 1 and 2 of [Figure 8.1a](#) can be thought of as a set of parallel paths between nodes 1 and 2 and can be denoted by $ea cf + eadf + ebcf + ebdf$. If X and Y are sets of paths that lie between the same pair of nodes, then $X + Y$ denotes the union of those sets of paths. As an example,



The first set of parallel paths is denoted by $X + Y + d$ and the second set by $U + V + W + h + i + j$. The set of all paths in this flowgraph is

$$f(X + Y + d)g(U + V + W + h + i + j)k$$

Keep in mind in the above example that the uppercase letters can represent individual segment names ($pqrst$, say) or sets of segment names, such as $pqrst + pqrst + pqrst + \dots$. Because the path sum is a set union operation, it is clearly commutative and associative; that is,

$$\text{Rule 2: } X + Y = Y + X$$

$$\text{Rule 3: } (X + Y) + Z = X + (Y + Z) = X + Y + Z$$

3.5 Distributive Laws

The product and sum operations are distributive, and the ordinary rules of multiplication apply; that is,

$$\text{Rule 4: } A(B + C) = AB + AC \text{ and } (B + C)D = BC + BD$$

Applying these rules to [Figure 8.1a](#) yields

$$\begin{aligned} e(a + b)(c + d)f &= e(ac + ad + bc + bd)f \\ &= eacf + eadf + ebcf + e bdf \end{aligned}$$

for the set of all paths from node 1 to node 2.

3.6. Absorption Rule

If X and Y denote the same set of paths, then the union of these sets is unchanged; consequently,

Rule 5: $X + X = X$ (absorption rule)

Similarly, if a set consists of path names and a member of that set is added to it, the “new” name, which is already in that set of names, contributes nothing and can be ignored. For example, if

$$X = a + aa + abc + abcd + def$$

then

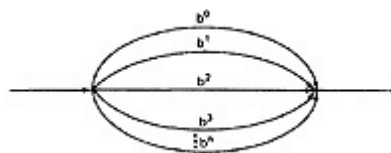
$$X + a = X + aa = X + abc = X + abcd = X + def = X$$

It follows that any arbitrary sum of identical path expressions reduces to the same path expression.

3.7. Loops

Loops can be understood as an infinite set of parallel paths. Say that the loop consists of a single link b . Then the set of all paths through that loop point is

$$b^0 + b^1 + b^2 + b^3 + b^4 + b^5 + \dots$$



This potentially infinite sum is denoted by b^* for an individual link and by X^* when X is a path expression. If the loop must be taken at least once, then it is denoted by a^+ or X^+ . The path expressions for [Figure 8.1c](#) and 8.1d, respectively, as expressed by this notation, are

$$ab^*c = ac + abc + abbc + abbbc + \dots$$

and

$$a(bc)^*bd = abd + abcbcd + abcbcbcd + a(bc)^3bd + \dots$$

Evidently

$$aa^* = a^*a = a^+$$

and

$$XX^* = X^*X = X^+$$

It is sometimes convenient to denote the fact that a loop cannot be taken more than a certain, say n , number of times. A bar is used under the exponent to denote that fact as follows:

$$X^{\bar{n}} = X^0 + X^1 + X^2 + X^3 + \dots + X^{n-1} + X^n$$

The following rules can be derived from the previous rules:

Rule 6: $X^{\bar{n}} + X^{\bar{m}} = X^{\bar{n}}$ if n is bigger than m

$= X^{\bar{m}}$ if m is bigger than n

Rule 7: $X^{\bar{n}}X^{\bar{m}} = X^{\overline{n+m}}$

Rule 8: $X^{\bar{n}}X^* = X^*X^{\bar{n}} = X^*$

Rule 9: $X^{\bar{n}}X^+ = X^+X^{\bar{n}} = X^+$

Rule 10: $X^*X^+ = X^+X^* = X^+$

3.8. Identity Elements

Returning to the meaning of terms such as a^0 or X^0 , which denote the path whose length is zero, the following rules, previously used without explanation, apply:

Rule 11: $1 + 1 = 1$

Rule 12: $1X = X1 = X$ Following or preceding a set of paths by a path of zero length doesn't change the set.

Rule 13: $1^n = 1^{\bar{n}} = 1^* = 1^+ = 1$ No matter how often you traverse a path of zero length, it is still a path of zero length.

Rule 14: $1^+ + 1 = 1^* = 1$

The final notation needed is the empty set or the set that contains no paths, not even the zero-length path 1. The null set of paths is denoted by the numeral 0.* Zero, in this algebra, plays the same role as zero does in ordinary arithmetic. That is, it obeys the following rules:

* As with the unit element λ , the first edition and regular expression literature use the Greek letter phi (Φ) for the null set.

Rule 15: $X + 0 = 0 + X = X$

Rule 16: $X0 = 0X = 0$ If you block the paths of a graph fore or aft by a graph that has no paths, there

won't be any paths.

Rule 17: $0^* = 1 + 0 + 02 + \dots = 1$

The meaning and behavior of zero and one (the identity elements) given above apply only to path names. Other applications have different identity elements with different properties.

4. A REDUCTION PROCEDURE

4.1. Overview

This section presents a reduction procedure for converting a flowgraph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that flowgraph. The procedure is a node-by-node removal algorithm. You follow these steps, which initialize the process:

1. Combine all serial links by multiplying their path expressions.
2. Combine all parallel links by adding their path expressions.
3. Remove all self-loops (from any node to itself) by replacing them with a link of the form X^* , where X is the path expression of the link in that loop.

The remaining steps are in the algorithm's loop:

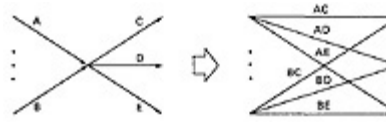
4. Select any node for removal other than the initial or final node. Replace it with a set of equivalent links whose path expressions correspond to all the ways you can form a product of the set of inlinks with the set of outlinks of that node.
5. Combine any remaining serial links by multiplying their path expressions.
6. Combine all parallel links by adding their path expressions.
7. Remove all self-loops as in step 3.
8. Does the graph consist of a single link between the entry node and the exit node? If yes, then the path expression for that link is a path expression for the original flowgraph; otherwise, return to step 4.

Each step will be illustrated and explained in further detail in the next sections. Note the use of the phrase “*a* path expression,” rather than “*the* path expression.” A flowgraph can have many equivalent path expressions between a given pair of nodes; that is, there are many different ways to generate the set of all paths between two nodes without affecting the content of that set. The appearance of the path expression depends, in general, on the order in which nodes are removed.

4.2. Cross-Term Step (Step 4)

The cross-term step* is the fundamental step of the reduction algorithm. It removes a node, thereby reducing the number of nodes by one. Successive applications of this step eventually get you down to one entry and one exit node. The following diagram shows the situation at an arbitrary node that has been selected for removal:

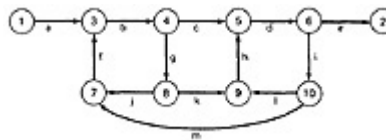
* Also known as the star-mesh transformation in signal flowgraph literature. See BRZO63, MAYE72.



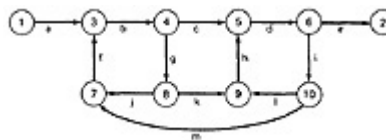
The rationale for this step is intuitively obvious. Whatever the set of paths represented by, say, A and C, there can be no other paths for the combination other than those represented by AC. Similarly, the removal of the node results in the AD, AE, BC, BD, and BE path expressions or path sets. If the path expressions are path names, it is clear that the resulting path names will be the names obtained by traversing the pair of links. If the path expressions denote sets of paths or path sums, using the definition of multiplication and the distributive rule produces every combination of incoming and outgoing path segments, as in

$$(a + b)(c + d) = ac + ad + bc + bd$$

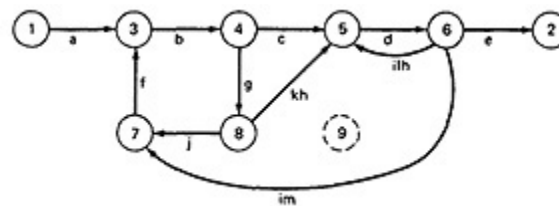
Applying this step to the graph of [Figure 8.1b](#), we remove several nodes in order; that is,



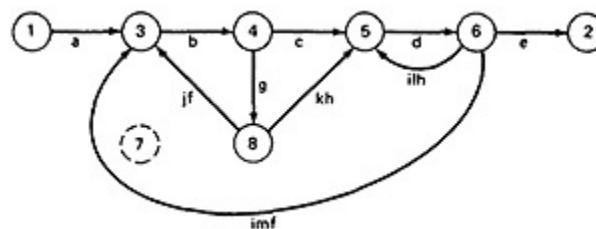
Remove node 10 by applying step 4 and combine by step 5 to yield



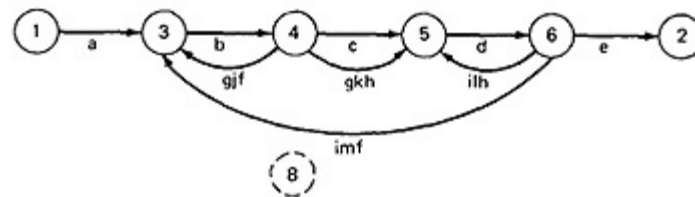
Remove node 9 by applying steps 4 and 5 to yield



Remove node 7 by steps 4 and 5, as follows:

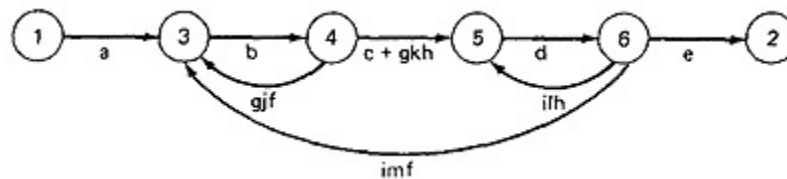


Remove node 8 by steps 4 and 5, to obtain



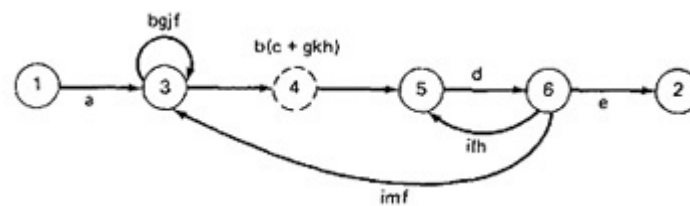
4.3. Parallel Term (Step 6)

Removal of node 8 above led to a pair of parallel links between nodes 4 and 5. Combine them to create a path expression for an equivalent link whose path expression is $c + gkh$; that is,

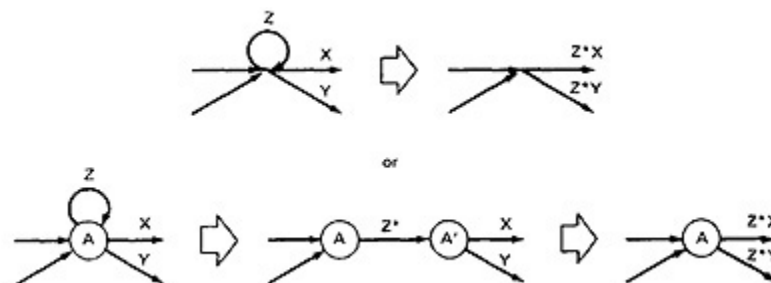


4.4. Loop Term (Step 7)

Removing node 4 leads to a loop term. The graph has now been replaced with the following equivalent, simpler graph:

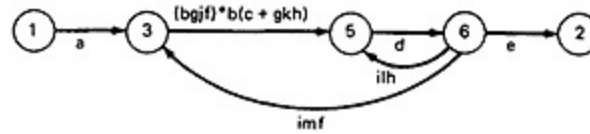


There are two ways of looking at the loop-removal operation:

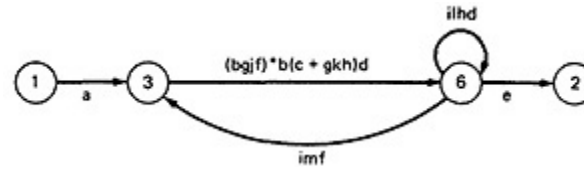


In the first way, we remove the self-loop and then multiply all outgoing links by Z^* . The second way shows things in more detail. We split the node into two equivalent nodes, call them A and A' and put in a link between them whose path expression is Z^* . Then we remove node A' using steps 4 and 5 to yield outgoing links whose path expressions are Z^*X and Z^*Y .

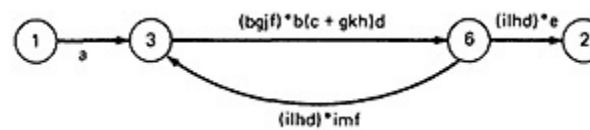
Continue the process by applying the loop-removal step, as follows:



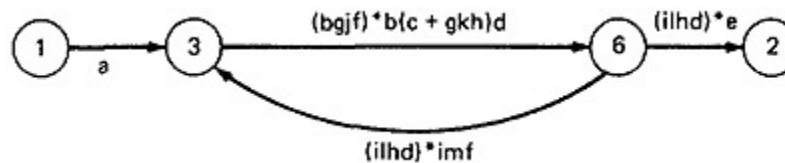
Removing node 5 produces



Remove the loop at node 6 to yield



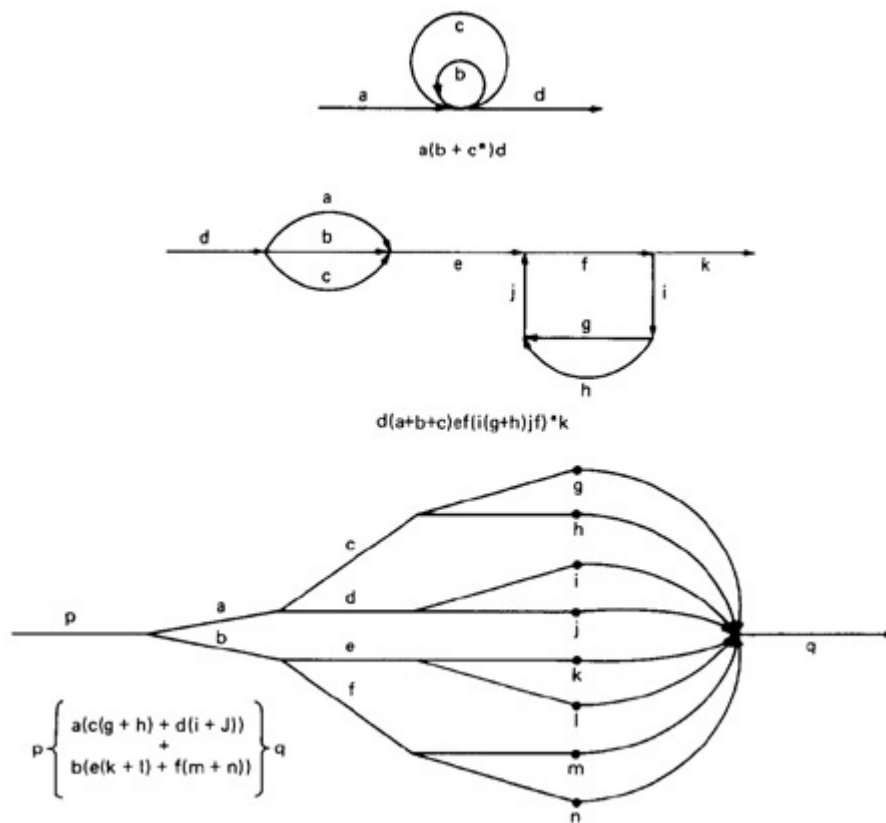
Remove node 3 to yield



Removing the loop and then node 6 results in the following ugly expression:

$a(bgjf)^*b(c + gkh)d((ilhd)^*imf(bgjf)^*b(c + gkh)d)^*(ilhd)^*e$

Figure 8.2. Some Graphs and Their Path Expressions.



We shouldn't blame the expression for being ugly because it was, after all, derived from an ugly, unstructured monster of a graph. With structured code, the path expressions are tamer. [Figure 8.2](#) shows examples.

4.5. Comments, Identities, and Node-Removal Order

I said earlier that the order in which the operations are done affects the appearance of the path expressions. Such appearances of differences also result in identities that can sometimes be used to simplify path expressions.

- I1: $(A + B)^* = (A^* + B^*)^*$
- I2: $(A^*B^*)^*$
- I3: $= (A^*B)^*A^*$
- I4: $= (B^*A)^*B^*$
- I5: $= (A^*B + A)^*$
- I6: $= (B^*A + B)^*$
- I7: $(A + B + C + \dots)^* = (A^* + B^* + C^* + \dots)^*$
- I8: $= (A^*B^*C^* \dots)^*$

These can be derived by considering different orders of node removals and then applying the series-parallel-loop rules. Each change in order can produce a different appearance for the path expression and therefore a path expression identity. Don't make the mistake of applying these identities to finite exponents or $+$. These identities hold only because they denote infinite sets of paths. These identities are not very important anyhow, because we will rarely deal with path expressions as such but rather with other kinds of expressions derived from the path expressions by using link weights and link arithmetics.

As an example of misapplying the identities, consider:

$$(A + B)^2 \neq (A^2 + B^2)^2 \neq (A^2B^2)^2$$

If A consists of the single link a and B is link b , the three expressions correspond to the following sets of paths:

$$(A + B)^2 = aa + ab + bb + ba$$

$$(A^2 + B^2)^2 = (a^4 + a^2b^2 + b^2a^2 + b^4)$$

$$(A^2B^2)^2 = a^2b^2a^2b^2 = (a^2b^2)^2$$

This algorithm can be used to find the path expression between any two nodes in a graph, including a node and itself. It is not restricted to finding the path expression between the entry and the exit of a flowgraph, although that might be the most common and most useful application. The method is tedious and cumbersome, what with having to constantly redraw the graph. In [Chapter 12](#) I'll present a more powerful version of the same algorithm that can find the path expression between every pair of nodes with less work than this graphical method requires.

5. APPLICATIONS

5.1. General

The previous sections of this chapter are more abstract than I and most readers are apt to like. They are, I admit, remote from the substantive problems of testing and test design. The purpose of all that abstraction was to present one very generalized concept—the path expression and one very generalized way of getting it, the node-removal algorithm, so that the same concepts would not have to be discussed over and over again as one variation on a theme after another. Every application follows this common pattern:

1. Convert the program or graph into a path expression.
2. Identify a property of interest and derive an appropriate set of “arithmetic” rules that characterizes the property.
3. Replace the link names by the link weights (remember them?) for the property of interest. The path expression has now been converted to an expression in some algebra, such as ordinary algebra, regular expressions, or boolean algebra. This algebraic expression summarizes the property of interest over the set of all paths.
4. Simplify or evaluate the resulting “algebraic” expression to answer the question you asked.

If it seems that the above algorithm requires you to invent a new form of arithmetic for each application, that's true, but it's far less formidable than it seems. In practice you don't do it as outlined above. You substitute the weights (the properties associated with the links) first and simplify as you develop the path expression, using the right kind of arithmetic as you remove the nodes. This is apt to be hazy in the abstract, so let's get to the first application.

5.2. How Many Paths in a Flowgraph?

5.2.1. The Question

The question is not simple. Here are some ways you could ask it:

1. What is the maximum number of different paths possible?
2. What is the fewest number of paths possible?
3. How many different paths are there really?
4. What is the average number of paths?

In all that follows, by “path” I mean paths from the entrance to the exit of a single-entry/single-exit routine.* The first question has a straightforward answer and constitutes the first application. The second question concerns the fewest number of paths and is inherently difficult. No satisfactory algorithm exists; but I’ll present an approximate solution for nicely structured flowgraphs. If we know both of these numbers (maximum and minimum number of possible paths) we have a good idea of how complete our testing is. Suppose that the minimum number of possible paths is 15 and the maximum is 144 and that we had planned only 12 test cases. The discrepancy should warn us to look for incomplete coverage. Consider two routines with comparable structure and comparable testing requirements—one with a maximum path count of 1000 and the other with a maximum path count of 100. In both cases, say that coverage was achievable with 10 paths. We should have more confidence in the routine with a lower maximum path count, because the 10 paths is closer to satisfying the all-paths testing strategy for the 100-path routine than it is for the 1000-path routine.

*Not a fundamental requirement. See [Chapter 12](#) for paths from any node to any other node, including multi-entry and multi-exit routines.

Determining the actual number of different paths is an inherently difficult problem because there could be unachievable paths resulting from correlated and dependent predicates. In fact, it is generally unsolvable by static analysis. If all the predicates are uncorrelated and independent, not only does the flowgraph have no loops, but the actual, minimum, and maximum numbers of paths are the same.

Asking for “the average number of paths” is meaningless. The questions one should ask are questions such as: What is the mean path length over all paths? What is the mean processing time considering all paths? What is the most likely path? Such questions involve a notion of probability. A model for that is also provided as an application in the next section. It should not require deep analysis to find the “typical” or normal path. It should be obvious: the one that runs straight through the program. That should be tested, of course, and tested first, even though it will probably be the least revealing path of all.

5.2.2. Maximum Path Count Arithmetic

Label each link with a link weight that corresponds to the number of paths that that link represents. Typically, that’s one to begin with; but if the link represented a subroutine call, say, and you wanted to consider the paths through the subroutine in the path count, then you would put that number on the link. Also mark each loop with the maximum number of times that the loop can be taken. If the answer is infinite, you might as well stop the analysis because it’s clear that the maximum number of paths will be infinite. There are three cases of interest: parallel links, serial links, and loops. In what follows, A and B are path expressions and W_A and W_B are algebraic expressions in the weights.

CASE	PATH EXPRESSION	WEIGHT EXPRESSION
	$A + B$	

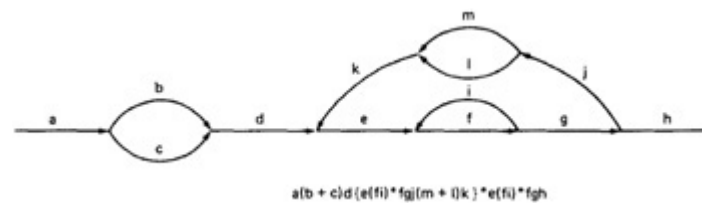
PARALLELS*		$W_A + W_B$
SERIES	AB	$W_A W_B$
LOOP	A^z	$\sum_{j=0}^n W_A^j$

* Adjustments maybe needed to avoid overcounting if both sets contain the zero-length path “1.” If X and Y have W_X and W_Y paths, respectively, then $W_{X+Y} = W_X + W_Y - 1$. Otherwise the zero-length path would be counted twice.

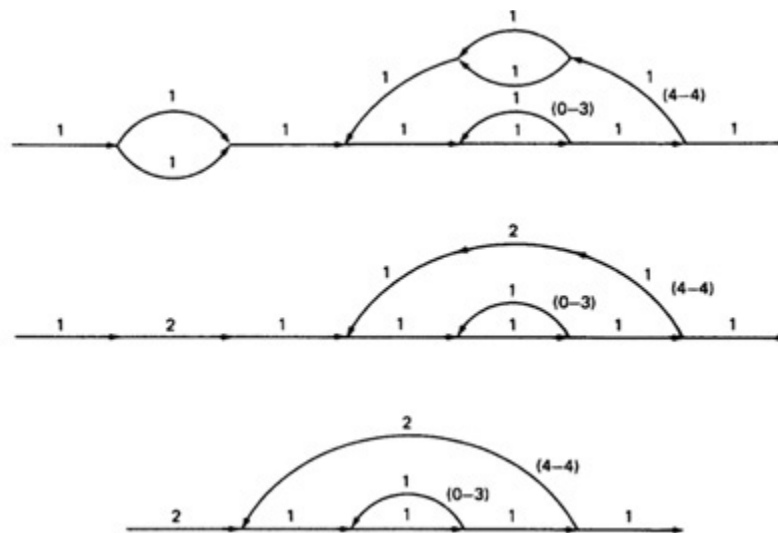
The arithmetic is ordinary algebra. This is an upper bound for the number of paths because the model can include unachievable paths. The rationale behind the parallel rule is simple. The path expressions denote the paths in a set of paths corresponding to that expression. The weight is the number of paths in each set. Assuming that the path expressions were derived in the usual way, they would have no paths in common, and consequently the sum of the paths for the union of the sets is at most the sum of the number of paths in each set. The series rule is explained by noting that each term of the path expression (say the first one A) will be combined with each term of the second expression B, in all possible ways. If there are W_A paths in A and W_B paths in B, then there can be at most $W_A W_B$ paths in the combination. The loop rule follows from the combination of the series and parallel rules, taking into account going through zero, once, twice, and so on. If you know for a fact that the minimum number of times through the loop is not zero but some other number, say j , then you do the summation from j to n rather than from 0 to n .

5.2.3. A Concrete Example

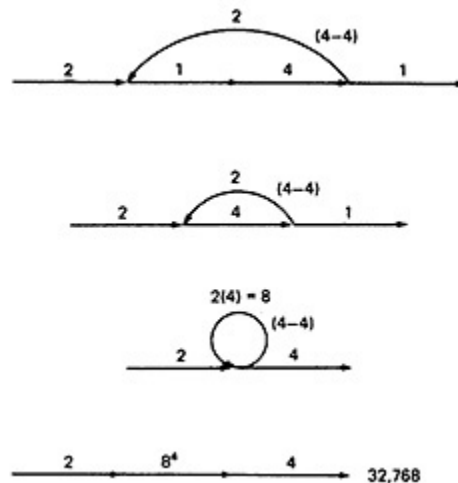
Here is a reasonably well-structured program. Its path expression, with a little work, is:



Each link represents a single link and consequently is given a weight of “1” to start. Let’s say that the outer loop will be taken exactly four times and the inner loop can be taken zero to three times. The steps in the reduction are as follows:



For the inner loop,



Alternatively, you could have substituted a “1” for each link in the path expression and then simplified, as follows:

$$1(1 + 1)1(1(1 \times 1)^3 1 \times 1 \times 1(1 + 1)1)^4 1(1 \times 1)^3 1 \times 1 \times 1$$

$$= 2(1^3 1 \times (2))^4 1^3$$

but

$$1^3 = 1 + 1^1 + 1^2 + 1^3 = 4$$

$$= 2(4 \times 2)^4 \times 4 = 2 \times 8^4 \times 4$$

$$= 32,768$$

This is the same result we got graphically. Reviewing the steps in the reduction, we:

1. Annotated the flowgraph by replacing each link name with the maximum number of paths through that link (1) and also noted the number of possibilities for looping. The inner loop was indicated by the range (0-3) as specified, and the outer loop by the range (4-4).

2. Combined the first pair of parallels outside of the loop and also the pair corresponding to the IF-THEN-ELSE construct in the outer loop. Both yielded two possibilities.
3. Multiplied things out and removed nodes to clear the clutter.
4. Took care of the inner loop: there were four possibilities, leading to the four values. Then we multiplied by the link weight following (originally link *g*) whose weight was also 1.
5. Got rid of link *e*.
6. Used the cross-term to create the self-loop with a weight of $8 = 2 \times 4$ and passed the other 4 through.

We have a test designer's bug. I've contradicted myself. I said that the outer loop would be taken exactly four times. That doesn't mean it will be taken *zero* or four times. Consequently, there is a superfluous "4" on the outlink in Step 6. Therefore the maximum number of different paths is 8192 rather than 32,768.

5.3. Approximate Minimum Number of Paths

5.3.1. Structured Code

The node-by-node reduction procedure can also be used as a test for structured code. Structured code can be defined in several different ways that do not involve ad hoc rules such as not using GOTOs. A graph-based definition by Hecht (HECH77B) is as follows:

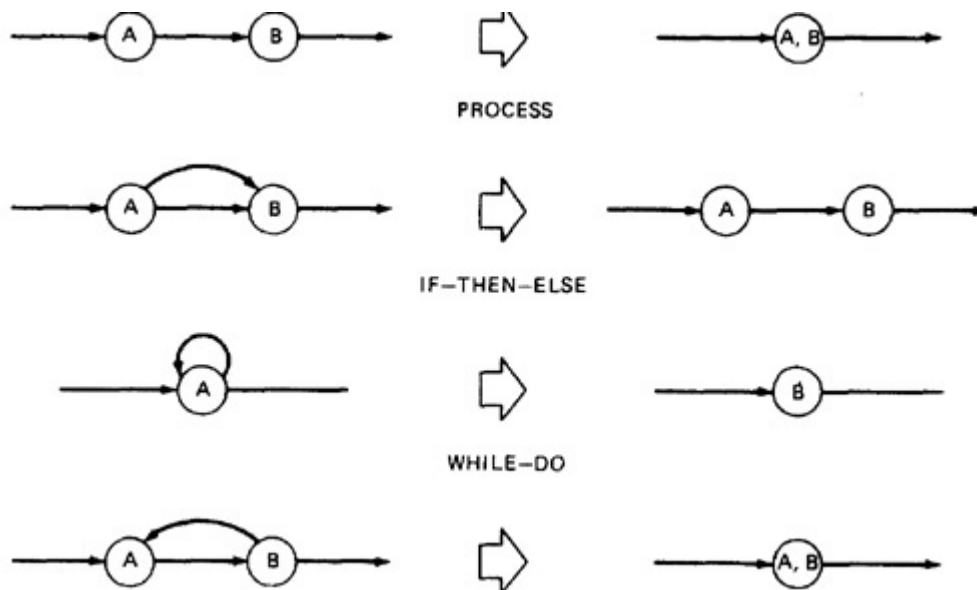


Figure 8.3. Structured-Flowgraph Transformations.

A **structured flowgraph** is one that can be reduced to a single link by successive application of the transformations of [Figure 8.3](#).

Note that the cross-term transformation is missing. An alternate characterization by McCabe (MCCA76) states that

Flowgraphs that do not contain one or more of the graphs shown in [Figure 8.4](#) as subgraphs are **structured**.

5.3.2. Lower Path Count Arithmetic

A lower bound on the number of paths in a routine can be approximated for structured flowgraphs. It is not a true lower bound because, again, unachievable paths could reduce the actual number of paths to a lower number yet. The appropriate arithmetic is as follows:

CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	$A + B$	$W_A + W_B$
SERIES	AB	$\text{MAX}(W_A, W_B)$
LOOP	A^L	$1, W_1$

The parallel case is the same as before. The values of the weights are the number of members in a set of paths. There could be an error here because both sets could contain the zero-length path, but because of the way the loop expression is defined, this cannot happen. The series case is explained by noting that each term in the first set will combine with at least one term in the second set. The minimum number of combinations must be the greater of the number of possibilities in the first set and the second set. The loop case requires that you use the minimum number of loops—possibly zero. Loops are always problematic. If the loop can be bypassed, then you can ignore the term in the loop. I don't think that this is a meaningful lower bound, because why is there a loop if it's not to be taken? By using a value of 1, we are asserting that we'll count the number of paths under the assumption that the loop will be taken once. Because in creating the self-loop we used the cross-term expression, there will be a contribution to the links following the loop, which will take things into account.

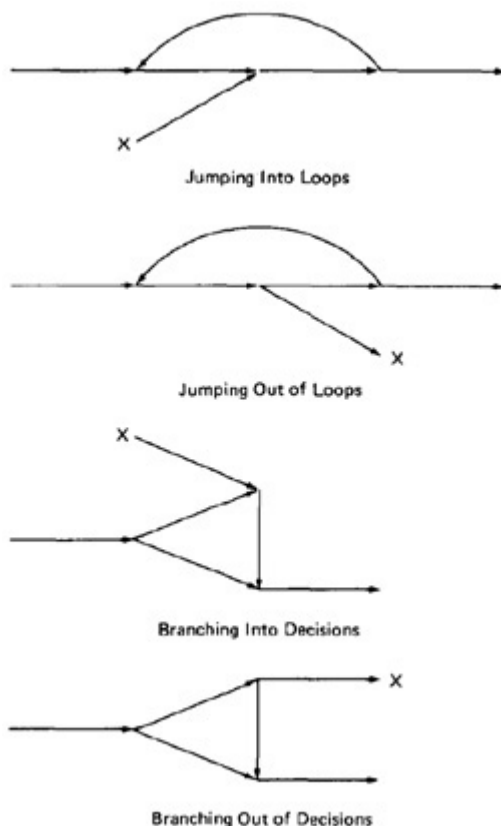
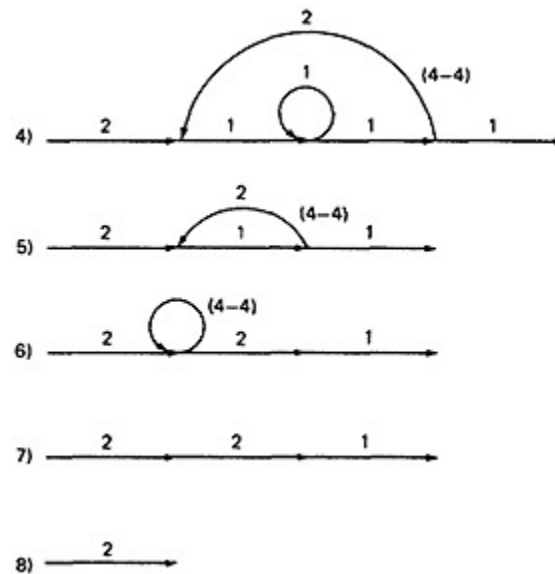


Figure 8.4. Unstructured Subgraphs.

Alternatively, you could get a higher lower bound by arguing that if the loop were to be taken once, then the path count should be multiplied by the loop weight. This however, would be equivalent to saying that the loop was assumed to be taken both zero and once because, again, the cross-term that created the self-loop was multiplied by the series term. Generally, if you ask for a minimum number of paths, it's more likely that the minimum is to be taken under the assumption that the routine will loop once—because it is consistent with coverage.

Applying this arithmetic to the earlier example gives us the identical steps until step 3, where we pick it up:



If you go back to the original graph on page 260 you'll see that it takes at least two paths to cover and that it can be done in two paths. The reason for restricting the algorithm to structured graphs is that for unstructured graphs the result can depend on the order in which nodes are removed. Structured or not, it's worth calculating this value to see if you have at least as many paths as the minimum number of paths calculated this way. If you have fewer paths in your test plan than this minimum you probably haven't covered. It's another check.

5.4. The Probability of Getting There

5.4.1. The Problem

I suggested in [Chapter 3](#) that, if anything, path selection should be biased toward the low- rather than the high-probability paths. This raises an interesting question: What is the probability of being at a certain point in a routine? This question can be answered under suitable assumptions, primarily that all probabilities involved are independent, which is to say that all decisions are independent and uncorrelated. This restriction can be removed, but the method is beyond the scope of this book. We use the same algorithm as before—node-by-node removal of uninteresting nodes.

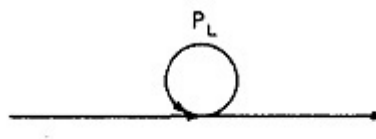
5.4.2. Weights, Notation, Arithmetic

Probabilities can come into the act only at decisions (including decisions associated with loops). Annotate each outlink with a weight equal to the probability of going in that direction. Evidently, the sum of the outlink probabilities must equal 1. For a simple loop, if the loop will be taken a mean of N times, the looping probability is $N/(N+1)$ and the probability of not looping is $1/(N+1)$. A link that is

not part of a decision node has a probability of 1. The arithmetic rules are those of ordinary arithmetic.

CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	$A + B$	$P_A + P_B$
SERIES	AB	$P_A P_B$
LOOP	A^*	$P_A / (1 - P_L)$

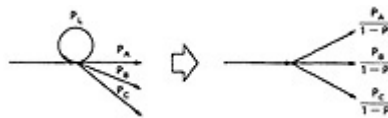
In this table, P_A is the probability of the link leaving the loop and P_L is the probability of looping. The rules are those of ordinary probability theory. If you can do something either from column A with a probability of P_A or from column B with a probability P_B , then the probability that you do either is $P_A + P_B$. For the series case, if you must do both things, and their probabilities are independent (as assumed), then the probability that you do both is the product of their probabilities.



$$P_A = 1 - P_L$$

$$P_{\text{NEW}} = \frac{P_A}{1 - P_L} = \frac{1 - P_L}{1 - P_L} = 1$$

A loop node has a looping probability of P_L and a probability of not looping of P_A , which is obviously equal to $1 - P_L$. Following the rule, all we've done is replace the outgoing probability with 1—so why the complicated rule? After a few steps in which you've removed nodes, combined parallel terms, removed loops and the like, you might find something like this:



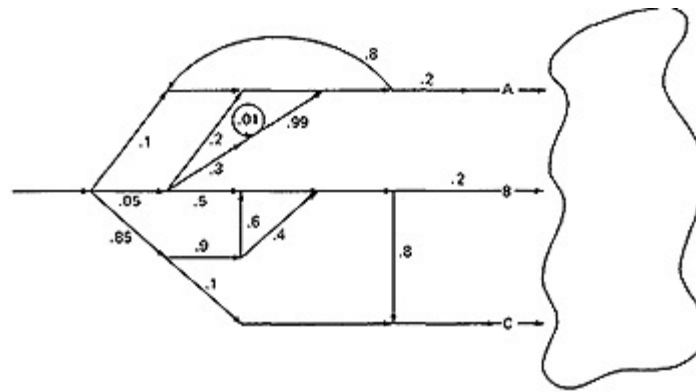
because $P_L + P_A + P_B + P_C = 1$, $1 - P_L = P_A + P_B + P_C$, and

$$\frac{P_A}{1 - P_L} + \frac{P_B}{1 - P_L} + \frac{P_C}{1 - P_L} = \frac{P_A + P_B + P_C}{1 - P_L} = 1$$

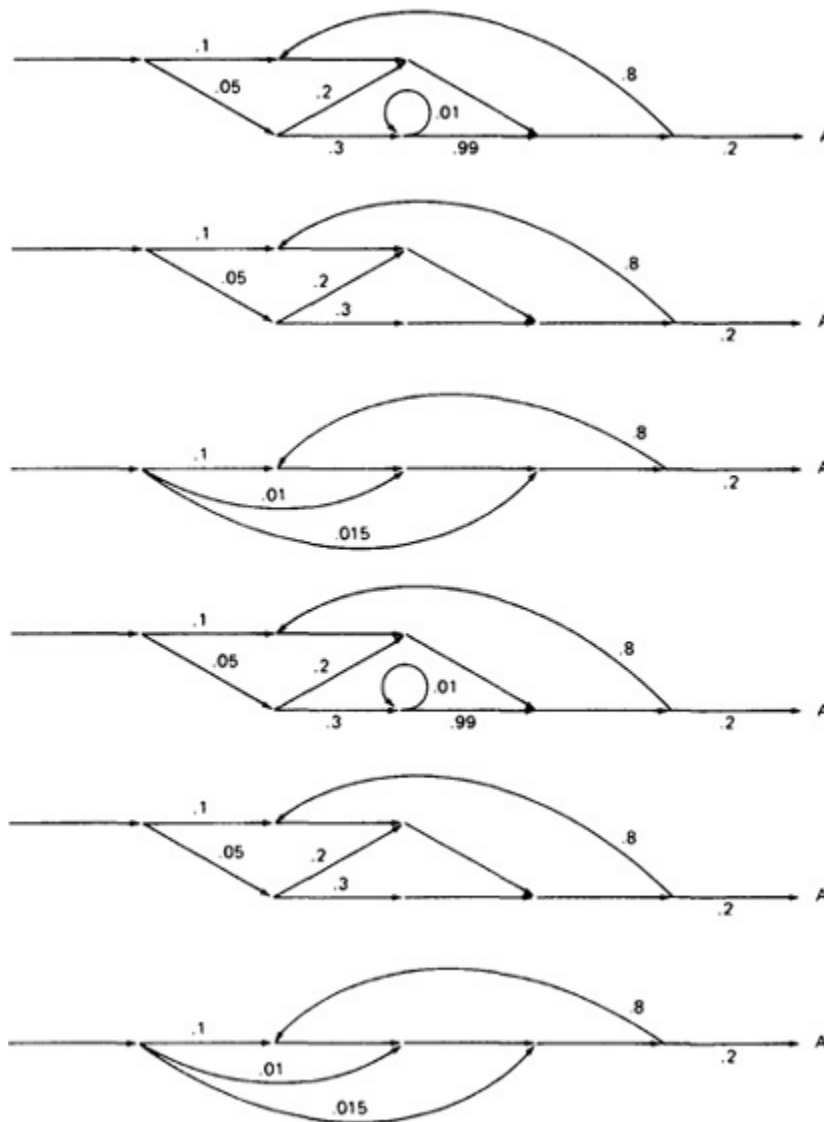
which is what we've postulated for any decision. In other words, division by $1 - P_L$ renormalizes the outlink probabilities so that their sum equals unity after the loop is removed.

5.4.3. Example

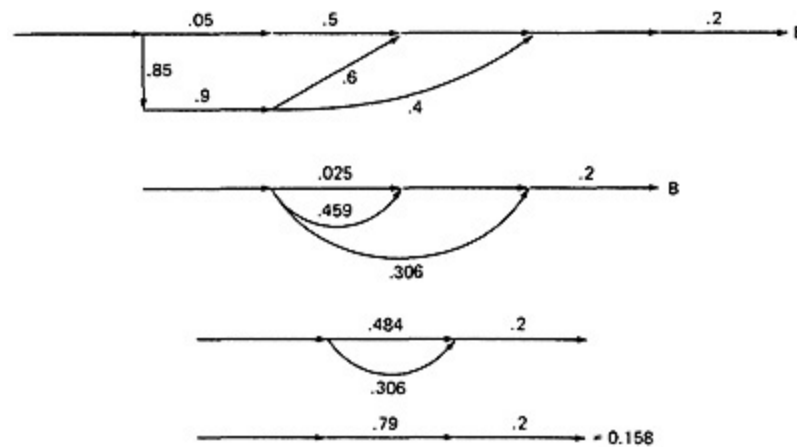
Here is a complicated bit of logic. We want to know the probability associated with cases A, B, and C.



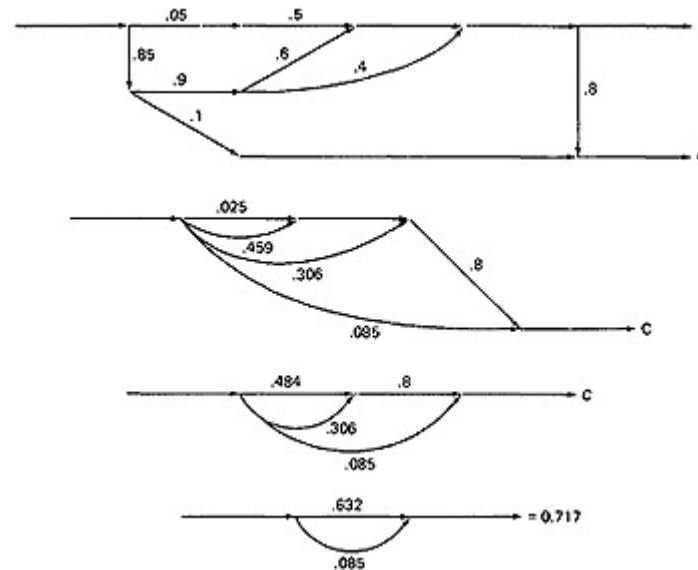
I'll do this in three parts, starting with case A. Note that the sum of the probabilities at each decision node is equal to 1. Start by throwing away anything that isn't on the way to case A, and then apply the reduction procedure. To avoid clutter, we usually leave out probabilities equal to 1—they're understood.



Case B is simpler:



Case C is similar and should yield a probability of $1 - 0.125 - 0.158 = 0.717$:



This checks. It's a good idea when doing this sort of thing to calculate all the probabilities and to verify that the sum of the routine's exit probabilities does equal 1. If it doesn't, then you've made calculation error or, more likely, you've left out some branching probability. Calculating the probability of reaching a point in a routine is not completely trivial, as you can see. If the logic is convoluted, simplistic methods of estimating can be very far off. It's better to analyze it.

How about path probabilities? That's easy. Just trace the path of interest and multiply the probabilities as you go. Alternatively, write down the path name and do the indicated arithmetic operation. Say that a path consisted of links *a*, *b*, *c*, *d*, *e*, and the associated probabilities were .2, .5, 1., .01, and 1 respectively. Path *abcbcbcddeabdddea* would have a probability of 5×10^{-10} . Long paths are usually improbable. Covering with short, simple paths is usually covering with high-probability paths. If you're going to make an argument related to testing based on probabilities, be prepared to evaluate those probabilities. If someone refutes a test based on probabilities, be prepared to demand an evaluation thereof rather than a haphazard guess.

Another good practice is to calculate the sum of the probabilities of the paths in the test set. A routine could have millions of paths but can be covered by ten or twenty. Calculate the sum of the probabilities for those paths and compare the sum with unity. Given two proposed sets of tests, the set whose path

probability sum is greater provides a statistically more complete test than the set whose path probability sum is smaller. Be careful how you apply this because:

1. Getting the probabilities can be very difficult.
2. Correlated and dependent predicates do not follow the simple rules of multiplication and addition. A comparison made in such cases could be misleading.
3. The probabilities of the covering path set tend to be small. Don't expect to compare test plan A with a probability sum of 0.9 to test plan B with a sum of 0.8—it's more likely to be a comparison of 10^{-3} to 10^{-5} .

5.5. The Mean Processing Time of a Routine

5.5.1. The Problem

Given the execution time of all statements or instructions for every link in a flowgraph and the probability for each direction for all decisions, find the mean processing time for the routine as a whole. Under suitable assumptions, specifically that the decisions are uncorrelated and independent, the following algorithm gives you the results. In practice, getting the probabilities is half the work. Data dependencies and correlated decisions can be handled with modifications of the algorithm. Furthermore, the standard deviation and higher moments can also be calculated. For more information, see BEIZ78.

5.5.2. The Arithmetic

The model has *two* weights associated with every link: the processing time for that link, denoted by T , and the probability of that link. The rules for the probabilities are identical to those discussed in Section 5.4. The rules for the mean processing times are:

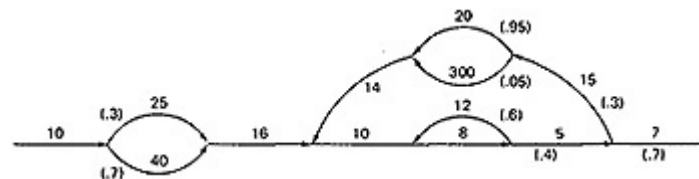
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	$A + B$	$T_{A+B} = (P_A T_A + P_B T_B) / (P_A + P_B)$ $P_{A+B} = P_A + P_B$
SERIES	AB	$T_{AB} = T_A + T_B$ $P_{AB} = P_A P_B$
LOOP	A^*	$T_A = T_A + T_L P_L / (1 - P_L)$ $P_A = P_A / (1 - P_L)$

The parallel term is the mean of the processing time over all the parallel links. Because the first node could have links that do not terminate on the second node of the parallels, we must divide by the sum of the parallel probabilities to get the proper mean. The serial term is intuitively obvious as the sum of the two processing times. The probability portion of the loop term is the same as before. The processing-time component consists of two parts. The part that was on the link leaving the loop (T_A) and the contribution of the loop. The loop contribution is most easily understood by substituting the value $N/(N + 1)$ for the probability of looping, under the assumption that the routine was expected to loop N times. The $P_L/(1 - P_L)$ term then reduces to just N . If the routine is expected to loop N times, and each time

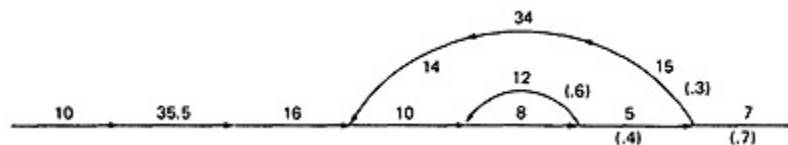
takes T_L seconds, and thereafter it does T_A seconds' worth of work, then the mean time through the entire process is $T_A + NT_L$.

5.5.3. An Example

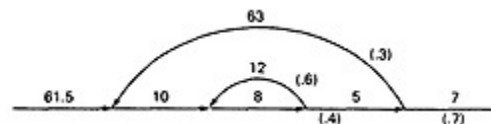
As an example, we'll use our old standby now annotated with branch probabilities, loop probabilities, and processing times for each link. The probabilities are given in parentheses. The node-removal order will be as in the previous use of this graph. It helps to remove nodes from the inside of loops to the outside.



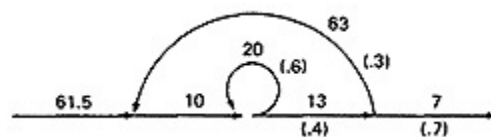
1. Start with the original flowgraph annotated with probabilities and processing time in microseconds or in instructions or whatever else is convenient.



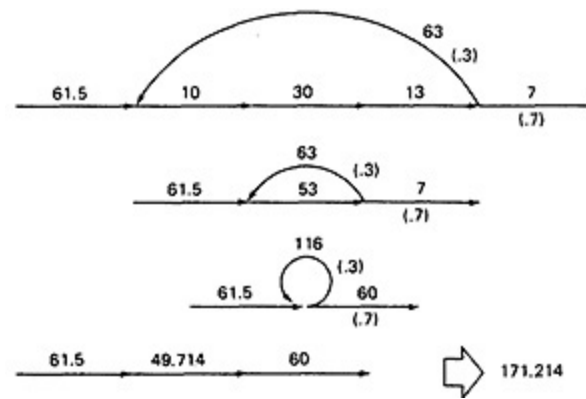
2. Combine the parallel links of the outer loop. The result is just the mean of the processing times for the links because there aren't any other links leaving the first node. Also combine the pair of links at the beginning of the flowgraph.



3. Combine as many serial links as you can.



4. Use the cross-term step to eliminate a node and to create the inner self-loop.



5.6. Push/Pop, Get/Return

5.6.1. The Problem

This model can be used to answer several different questions that can turn up in debugging. It can also help decide which test cases to design. The question is: given a pair of complementary operations such as PUSH (the stack) and POP (the stack), considering the set of all possible paths through the routine, what is the net effect of the routine? PUSH or POP? How many times? Under what conditions? Here are some other examples of complementary operations to which this model applies:

GET/RETURN a resource block.
 OPEN/CLOSE a file.
 START/STOP a device or process.

5.6.2. Push/Pop Arithmetic

CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	$A + B$	$W_A + W_B$
SERIES	AB	$W_A W_B$
LOOPS	A^*	W_A^*

An arithmetic table is needed to interpret the weight addition and multiplication operations. Typically, the exponent for loops will be the normal exponent. As before, we must be careful with loops: if a specific number of loops will be taken, the nonlooping term is not multiplied for the links following the loop. The numeral 1 is used to indicate that nothing of interest (neither PUSH nor POP) occurs on a given link. “H” denotes PUSH and “P” denotes POP. The operations are commutative, associative, and distributive.

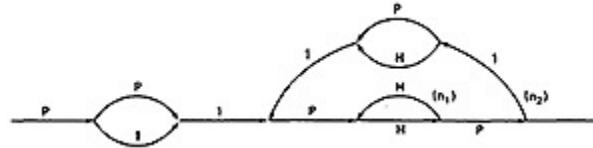
PUSH/POP MULTIPLICATION TABLE

X	H PUSH	P POP	1 NONE
H	H^2	1	H
P	1	P^2	P
1	H	P	1

PUSH/POP ADDITION TABLE

*	H PUSH	P POP	1 NONE
H	H	$P + H$	$H + 1$
P	$P + H$	P	$P + 1$
1	$H + 1$	$P + 1$	1

Example:



$$P(P + 1)1\{P(HH)^{n_1}HP1(P + H)1\}^{n_2}P(HH)^{n_1}HPH$$

Simplifying by using the arithmetic tables,

$$(P^2 + P)\{P(HH)^{n_1}(P + H)\}^{n_1}(HH)^{n_1} \\ (P^2 + P)\{H^{2n_1}(P^2 + 1)\}^{n_2}H^{2n_1}$$

The circumstances under which the stack will be pushed, popped, or left alone by the routine can now be determined. [Table 8.1](#) shows several combinations of values for the two looping terms— n_1 is the number of times the inner loop will be taken and n_2 the number of times the outer loop will be taken.

These expressions state that the stack will be popped only if the inner loop is not taken. The stack will be left alone only if the inner loop is iterated once, but it may also be pushed. For all other values of the inner loop, the stack will only be pushed.

Exactly the same arithmetic tables are used for GET/RETURN a buffer block or resource, or, in fact, for any pair of complementary operations in which the total number of operations in either direction is cumulative. As another example, consider INCREMENT/DECREMENT of a counter. The question of interest would be: For various loop counts, considering the set of all possible paths, is the net result a positive or negative count and what are the values? The arithmetic tables for GET/RETURN are:

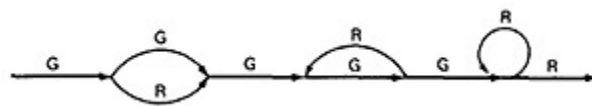
X	G	R	1
G	G^2	1	G
R	1	R^2	R
1	G	R	1

+	G	R	1
G	G	$G + R$	$G + 1$
R	$G + R$	R	$R + 1$
1	$G + 1$	$R + 1$	1

M_1	M_2	PUSH/POP
0	0	$p + p^2$
0	1	$p + p^2 + p^3 + p^4$
0	2	$\sum_{i=1}^6 p^i$
0	3	$\sum_{i=1}^8 p^i$
1	0	$1 + H$
1	1	$\sum_{i=0}^3 H^i$
1	2	$\sum_{i=0}^5 H^i$
1	3	$\sum_{i=0}^7 H^i$
2	0	$H^2 + H^3$
2	1	$\sum_{i=4}^7 H^i$
2	2	$\sum_{i=6}^{11} H^i$
2	3	$\sum_{i=8}^{15} H^i$

Table 8.1. Result of the PUSH/POP Graph Analysis

Example:



$$\begin{aligned}
 & G(G + R)G(GR)^*GGR^*R \\
 &= G(G + R)G^3R^*R \\
 &= (G + R)G^3R^* \\
 &= (G^4 + G^2)R^*
 \end{aligned}$$

This expression specifies the conditions under which the resources will be balanced on leaving the routine. If the upper branch is taken at the first decision, the second loop must be taken four times. If the lower branch is taken at the first decision, the second loop must be taken twice. For any other values, the routine will not balance. Therefore, the first loop does not have to be instrumented to verify this behavior because its impact should be nil. The first decision and the second loop should be instrumented.

5.7. Limitations and Solutions

The main limitation to these applications is the problem of unachievable paths. The node-by-node reduction procedure, and most graph-theory-based algorithms work well when all paths are possible, but may provide misleading results when some paths are unachievable. The approach to handling unachievable paths (for any application) is to partition the graph into subgraphs so that all paths in each of the subgraphs are achievable. The resulting subgraphs may overlap, because one path may be common to several different subgraphs. Each predicate's truth-functional value potentially splits the graph into two subgraphs. For n predicates, there could be as many as 2^n subgraphs. Here's the algorithm for one predicate:

1. Set the value of the predicate to TRUE and strike out all FALSE links for that predicate.
2. Discard any node, other than an entry or exit node, that has no incoming links. Discard all links that leave such nodes. If there is no exit node, the routine probably has a bug because there is a predicate value that forces an endless loop or the equivalent.
3. Repeat step 2 until there are no more links or nodes to discard. The resulting graph is the subgraph corresponding to a TRUE predicate value.
4. Change "TRUE" to "FALSE" in the above steps and repeat. The resulting graph is the subgraph that corresponds to a FALSE predicate value.

Only correlated predicates whose values exclude paths should be included in this analysis—not all predicates that may control the program flow. You can usually pick out the subgraphs by inspection because only one or two predicates, each of which appears in only a few places, cause the unachievable paths. If it isn't that simple, the routine is probably more complicated than it should be. A side benefit of the partitioning is that it may suggest a simpler, cleaner, and easier-to-test routine. The cost of this simplicity is probably a modest increase in the number of statements, which is partially paid for by an improvement in running time. Ask yourself why the routine is hard to analyze and why a formal analysis was required to see how various sets of paths were mutually exclusive. More often it's because the routine's a murky pit than because it's deep and elegant.

6. REGULAR EXPRESSIONS AND FLOW-ANOMALY DETECTION

6.1. The Problem

The generic flow-anomaly detection problem (note: not just data-flow anomalies, but *any* flow anomaly) is that of looking for a specific sequence of options considering all possible paths through a routine. Let's the operations are SET and RESET, denoted by s and r respectively, and we want to know if there is a SET followed immediately a SET or a RESET followed immediately by a RESET (i, an ss or an rr sequence). Unlike the previous examples, we'll not take advantage of a possible arithmetic over the various operations because we are interested in knowing whether a specific sequence occurred, not what the net effect of the routine is. Here are some more application examples:

1. A file can be opened (o), closed (c), read (r), or written (w). If the file is read or written to after it's been closed, the sequence is nonsensical. Therefore, cr and cw are anomalous. Similarly, if the file is read before it's been written, just after opening, we may have a bug. Therefore, or is also anomalous. Furthermore, oo and cc , though not actual bugs, are a waste of time and therefore should also be examined.
2. A tape transport can do a rewind (d), fast-forward (f), read (r), write (w), stop (p), and skip (k). There are rules concerning the use of the transport; for example, you cannot go from rewind to fast-forward without an intervening stop or from rewind or fast-forward to read or write without an intervening stop. The following sequences are anomalous: df , dr , dw , fd , and fr . Does the flowgraph lead to anomalous sequences on any path? If so, what sequences and under what

circumstances?

3. The data-flow anomalies discussed in [Chapter 5](#) requires us to detect the dd , dk , kk , and ku sequences. Are there paths with anomalous data flows?

4. You suspect a bug that could occur only if two operations, a and b , occurred in the order aba or bab . Is there a path along which this is possible?

6.2. The Method

Annotate each link in the graph with the appropriate operator or the null operator 1. Simplify things to the extent possible, using the fact that $a + a = a$ and $1^2 = 1$. Other combinations must be handled with care, because it may not be the case that a null operation can be combined with another operation. For example, $1a$ may not be the same thing as a alone. You now have a **regular expression** that denotes all the possible sequences of operators in that graph. You can now examine that regular expression for the sequences of interest. A useful theorem by Huang (HUAN79) helps simplify things:

Let A , B , C , be nonempty sets of character sequences whose smallest string is at least one character long. Let T be a two-character string of characters. Then if T is a substring of (i.e., if T appears within) AB^nC , then T will appear in AB^2C .

As an example, let

$$A = pp$$

$$B = srr$$

$$C = rp$$

$$T = ss$$

The theorem states that ss will appear in $pp(srr)^n rp$ if it appears in $pp(srr)^2 rp$. We don't need the theorem to see that ss does not appear in the given string. However, let

$$A = p + pp + ps$$

$$B = psr + ps(r + ps)$$

$$C = rp$$

$$T = p^4$$

Is it obvious that there is a p^4 sequence in AB^nC ? The theorem states that we have only to look at

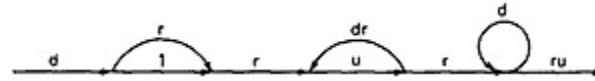
$$(p + pp + ps)[psr + ps(r + ps)]^2 rp$$

Multiplying out the expression and simplifying shows that there is no p^4 sequence.

A further point of Huang's theorem is directly useful in test design. He shows that if you substitute $1 + X^2$ for every expression of the form X^* , the paths that result from this substitution are sufficient to determine whether a given two-character sequence exists or not. Incidentally, the above observation is an informal proof of the wisdom of looping twice discussed in [Chapter 3](#). Because data-flow anomalies are represented by two-character sequences, it follows from Huang's theorem that looping twice is what you need to do to find such anomalies.

6.3. A Data-Flow Testing Example (HUAN79)

Here is a flowgraph annotated with the operators at the links that correspond to the variables of interest:



The *ku* bug is clear from the regular expression. It will occur whenever the first loop is not taken. The second part of Huang's theorem states that the following expression is sufficient to detect any two-character sequence:

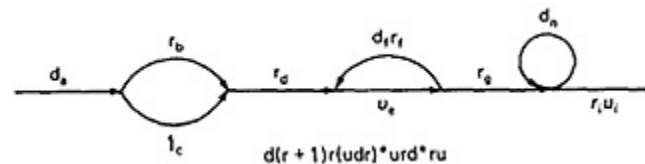
$$d(r + 1)r[1 + (udr)^2]ur(1 + d^2)ru$$

This makes the *dd* bug obvious. A *kk* bug cannot occur because there is no left-hand term that ends in *k*. Similarly, a *dk* bug cannot occur, because the only left-hand term ending in *d* is the last parenthesis, which is followed by *uk*.

$$(drr + dr)(1 + udrudr)(urru + urd^2ru)$$

There's no point in following through, because the bugs have been found. Generally, you would have to do this for every variable of interest. Also, the weights used with any link will change from variable to variable because different things are being done with each variable. For this reason, it's best to write down the path expression and to substitute the operators later. That way, you don't have to constantly redraw the graph and redo the path expression.

A better way to do this is to subscript the operator with the link name. Doing it this way, you don't lose the valuable path information. When two or more links or operators are combined, combine the link names. Here's the problem done over again with this feature added:



The path expression is

$$a(b + c)d(ef)^*egh*i$$

The regular expression is

$$d_a(r_b + 1_c)r_d(u_e d_f r_f)^*u_e r_g d_h^* r_i u_i$$

Applying Huang's theorem:

$$d_a(r_b + 1_c)r_d(1 + (u_e d_f r_f)^2)u_e r_g(1 + d_h^2)r_i u_i$$

$$(d_a r_b r_d + d_{ac} r_d)(u_e r_g + u_e d_f r_f u_e d_f r_f u_e r_g)(r_i u_i d_h^2 r_i u_i)$$

The resulting expression tells us the same thing and preserves the path names so that we can work back to see what paths are potentially responsible for the bug or, alternatively, what paths must be tested to assure detection of the problem.

Although it's a good idea to learn how to do this analysis by hand as a step in understanding the algorithms, actual application should be based on data-flow analysis tools. Unless you do a lot of practicing, you probably won't achieve enough facility with these methods to use them without tools.

6.4. Generalizations, Limitations, and Comments

Huang's theorem can be easily generalized to cover sequences of greater length than two characters. Beyond three characters, though, things get complex and this method has probably reached its utilitarian limit for manual application.

If A, B, and C are nonempty sets of strings of one or more characters, and if T is a string of k characters, and if T is a substring of AB^nC , where n is greater than or equal to k , then T is a substring of AB^kC . A sufficient test for strings of length k can be obtained by substituting P^k for every appearance of P^* (or P^n , where n is greater than or equal to k). Recall that

$$P^k = 1 + P + P^2 + P^3 + \dots + P^k$$

A warning concerning the use of regular expressions: there are almost no other useful identities beyond those shown earlier for the path expressions. There are some nice theorems for finding sequences that occur at the beginnings and ends of strings (BRZO62B) but no nice algorithms for finding strings buried in an expression. The mathematics for finding initial and terminal substrings is analogous to taking derivatives of algebraic expressions, and things get abstract fast. Because you can usually see whether a sequence starts or ends a regular expression by inspection, the additional mathematics hardly seem worthwhile. The main use of regular expressions is as a convenient notation and method of keeping track of paths and sets of paths. Once you learn it, doing the algebra and manipulating the expression is easier than tracing (and perhaps missing) paths on a flowgraph.

A final caution concerns unachievable paths. Static flow analysis methods can't determine whether a path is or is not achievable. Unless the flow analysis includes symbolic execution or similar techniques, the impact of unachievable paths will not be included in the analysis. That could lead to either optimistic or pessimistic values depending on the model and application. The flow-anomaly application, for example, doesn't tell us that there *will* be a flow anomaly—it tells us that if the path is achievable, then there will be a flow anomaly. Such analytical problems go away, of course, if you take the trouble to design routines for which all paths are achievable.

7. SUMMARY

1. A flowgraph annotated with link names for every link can be converted into a path expression that represents the set of all paths in that flowgraph. A node-by-node reduction procedure is used.
2. By substituting link weights for all links, and using the appropriate arithmetic rules, the path expression is converted into an algebraic expression that can be used to determine the minimum and maximum number of possible paths in a flowgraph, the probability that a given node will be reached, the mean processing time of a routine, and other models.
3. With different, suitable arithmetic rules, and by using complementary operators as weights for the links, the path expression can be converted into an expression that denotes, over the set of all possible paths, what the net effect of the routine is.
4. With links annotated with the appropriate weights, the path expression is converted into a regular expression that denotes the set of all operator sequences over the set of all paths in a routine. Rules for determining whether a given sequence of operations are possible are given. In other words, we have a generalized flow-anomaly detection method that'll work for data-flow anomalies or any other flow anomaly.

5. All flow analysis methods lose accuracy and utility if there are unachievable paths. Expand the accuracy and utility of your analytical tools by designs for which all paths are achievable. Such designs are *always* possible.

[<ch7](#) [toc](#) [ch9>](#)