UNIT IV

PATHS, PATH PRODUCTS AND REGULAR EXPRESSIONS

Paths,Path products and Regular expressions:- path products &pathexpression,reduction procedure, applications, regular expressions & flow anomaly detection.
Logic Based Testing:-overview,decision tables,pathexpressions,kv charts, specifications.

PATH PRODUCTS AND PATH EXPRESSION:

MOTIVATION:

Flow graphs are being an abstract representation of programs.
Any question about a program can be cast into an equivalent question about an appropriate flowgraph.
Most software development, testing and debugging tools use flow graphs analysis techniques.

PATH PRODUCTS:

Normally flow graphs used to denote only control flow connectivity.
The simplest weight we can give to a link is a name.
Using link names as weights, we then convert the graphical flow graph into an equivalent algebraic like expressions which denotes the set of all possible paths from entry to exit for the flow graph.
Every link of a graph can be given a name.
The link name will be denoted by lower case italic letters In tracing a path or path segment through a flow graph, 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.
For example, 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 5.1 shows some examples:

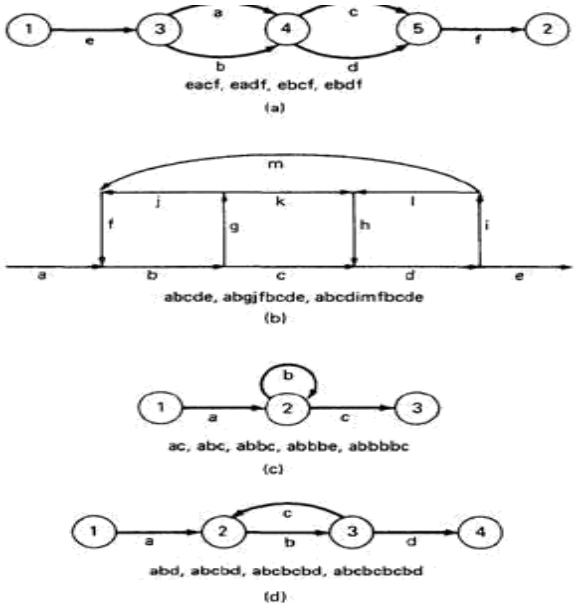


Figure 5.1: Examples of paths.

PATH EXPRESSION:

Consider a pair of nodes in a graph and the set of paths between those node. Denote that set of paths by Upper case letter such as X,Y. From Figure 5.1c, the members of the path set can be listed as follows:

ac, abc, abbc, abbbc, abbbbc.....

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

ac+abc+abbc+abbbc+abbbbc+.....

The + sign is understood to mean "or" 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 "OR"s and which denotes a set of paths between two nodes is called a "**Path Expression".**

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

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:
- $a^1 = a$; $a^2 = aa$; $a^3 = aaa$; $a^n = aaaa \ldots n$ times. Similarly, if X = abcde then

$X^1 = abcde$

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

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

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

The path product is not commutative (that is XY!=YX).

- The path product is Associative.

RULE 1: A(BC)=(AB)C=ABC

where A,B,C are path names, set 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.

$a^0 = 1$

$X^0=1$

PATH SUMS:

The "+" sign was used to denote the fact that path names were part of the same set of paths.

The "PATH SUM" denotes paths in parallel between nodes.

Links a and b in Figure 5.1a 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 can be thought of as a set of parallel paths and denoted by eacf+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 set of paths. For example, in Figure 5.2:

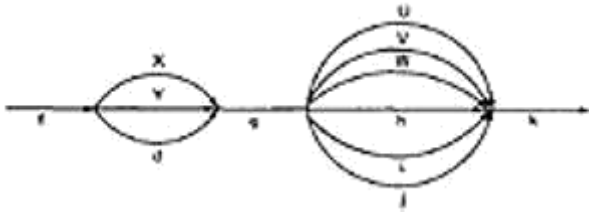


Figure 5.2: Examples of path sums.

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

The path is a set union operation, it is clearly Commutative and Associative.

RULE 2: X+Y=Y+X

RULE 3: (X+Y)+Z=X+(Y+Z)=X+Y+Z

DISTRIBUTIVE LAWS:

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

Applying these rules to the below Figure 5.1a yields

e(a+b)(c+d)f=e(ac+ad+bc+bd)f = eacf+eadf+ebcf+ebdf

ABSORPTION RULE:

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

If a set consists of paths 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.

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 b0+b1+b2+b3+b4+b5+.....

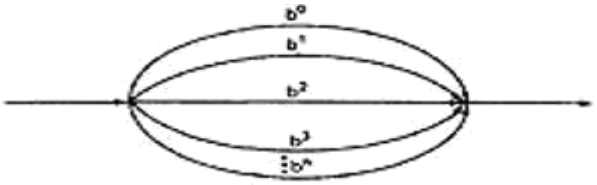


Figure 5.3: Examples of path loops.

This potentially infinite sum is denoted by b* for an individual link and by X*

Figure 5.4: Another example of path loops.

The path expression for the above figure is denoted by the notation: ab*c=ac+abc+abbc+abbbc+.....

Evidently,

aa*=a*a=a+ and XX*=X*X=X+

It is more 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 the fact as follows: $X^{\underline{n}} = X^0+X^1+X^2+X^3+X^4+X^5+ \ldots\ldots\ldots\ldots +X^n$

RULES 6 - 16:

The following rules can be derived from the previous rules:

- RULE 6: $X^n + X^{\underline{m}} = X^n$ if n>m

RULE 6: $X^{\underline{n}} + X^{\underline{m}} = X^{\underline{m}}$ if m>n

RULE 7: $X^nX^m = X^{\underline{n+m}}$

RULE 8: $X^nX^* = X^*X^n = X^*$

RULE 9: $X^nX^+ = X^+X^n = X^+$

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

RULE11:1+1=1

RULE 12: 1X = X1 = X

Following or preceding a set of paths by a path of zero length does not change the set.

RULE 13: $1^n = 1^{\underline{n}} = 1^* = 1^+ = 1$

No matter how often you traverse a path of zero length,It is a path of zero length. RULE 14: $1^+ +1 = 1^* = 1$

The null set of paths is denoted by the numeral 0. it obeys the following rules: RULE 15: X+0=0+X=X

RULE 16: 0X=X0=0

If you block the paths of a graph for or aft by a graph that has no paths , there won't be any paths.

REDUCTION PROCEDURE:

REDUCTION PROCEDURE ALGORITHM:

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.

The steps in Reduction Algorithm are as follows:

Combine all serial links by multiplying their path expressions.

Combine all parallel links by adding their path expressions.

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.

STEPS 4 - 8 ARE IN THE ALGORIHTM'S LOOP:

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.

Combine any remaining serial links by multiplying their path expressions.

Combine all parallel links by adding their path expressions.

Remove all self-loops as in step 3.

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.

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.

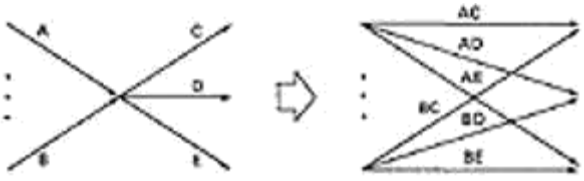
The appearance of the path expression depends, in general, on the order in which nodes are removed.

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:

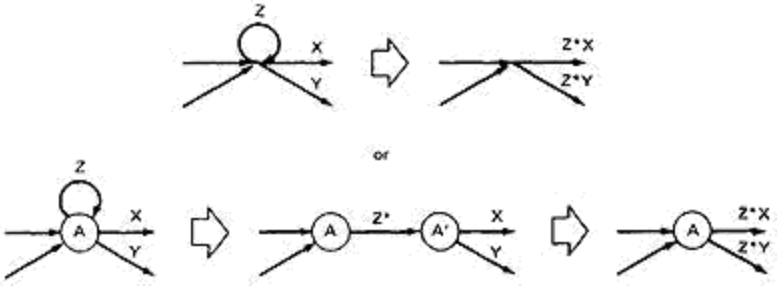


From the above diagram, one can infer:

- (a + b)(c + d + e) = ac + ad + + ae + bc + bd + be

LOOP REMOVAL OPERATIONS:

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*.

In the second way, 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.

A REDUCTION PROCEDURE - EXAMPLE:

Let us see by applying this algorithm to the following graph where we remove several nodes in order; that is

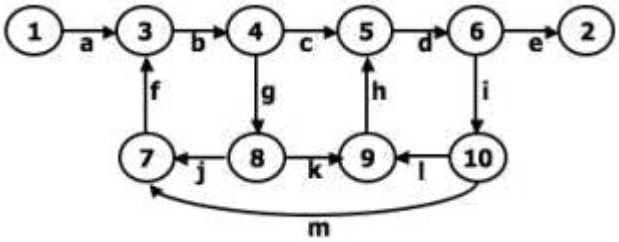
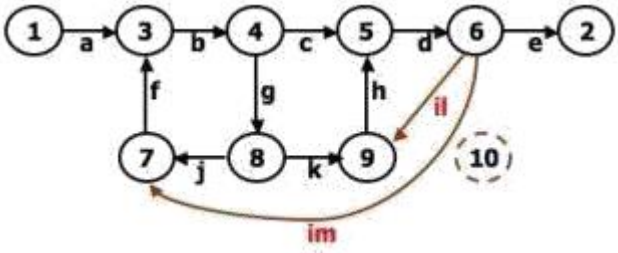
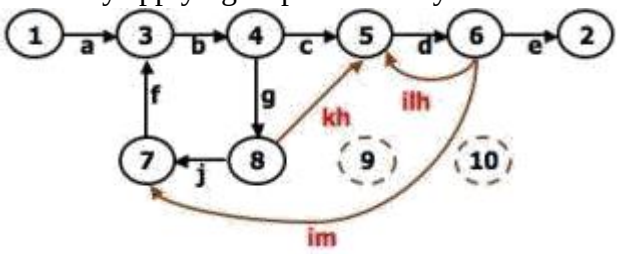


Figure 5.5: Example Flowgraph for demonstrating reduction procedure.

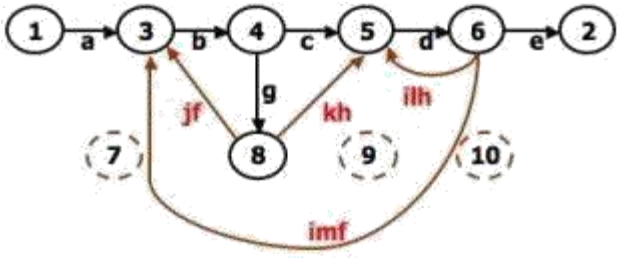
Remove node 10 by applying step 4 and combine by step 5 to yield



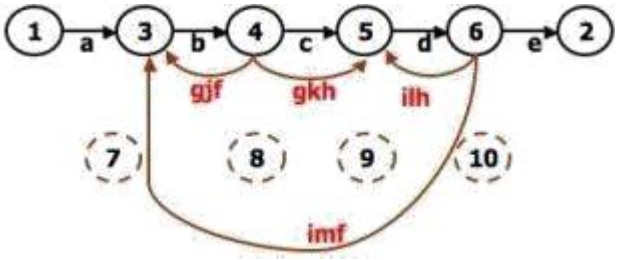
Remove node 9 by applying step4 and 5 to yield



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

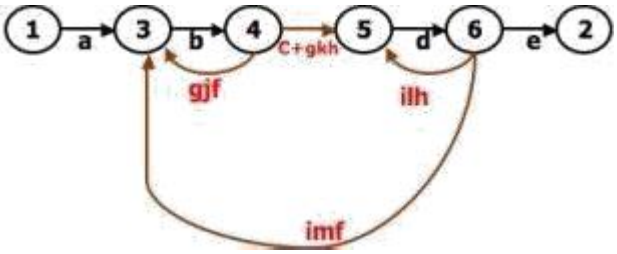


Remove node 8 by steps 4 and 5, to obtain:



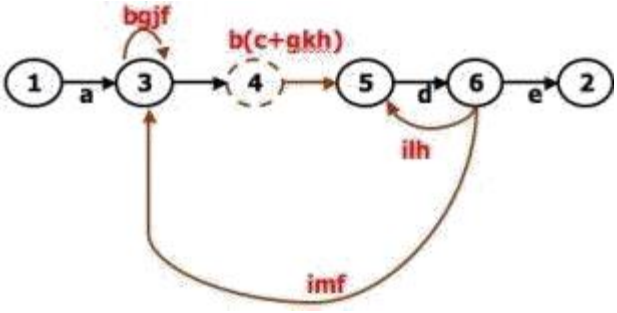
PARALLEL TERM (STEP 6):

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

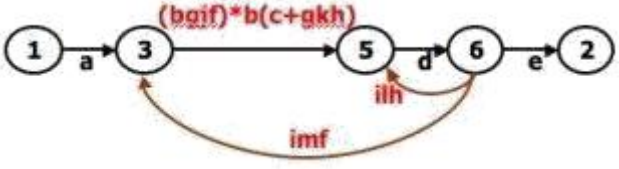


LOOP TERM (STEP 7):

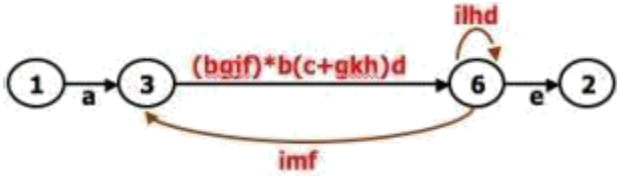
Removing node 4 leads to a loop term. The graph has now been replaced with the following equivalent simpler graph:



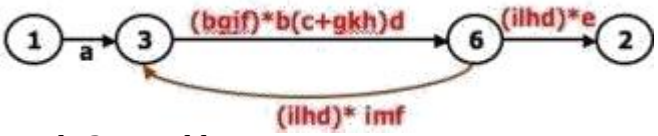
- Continue the process by applying the loop-removal step as follows:



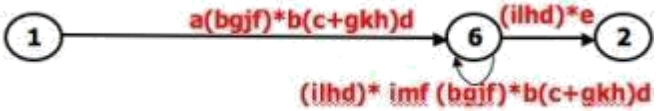
Removing node 5 produces:



Remove the loop at node 6 to yield:



Remove node 3 to yield



Removing the loop and then node 6 result in the following expression:

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

You can practice by applying the algorithm on the following flowgraphs and generate their respective path expressions:

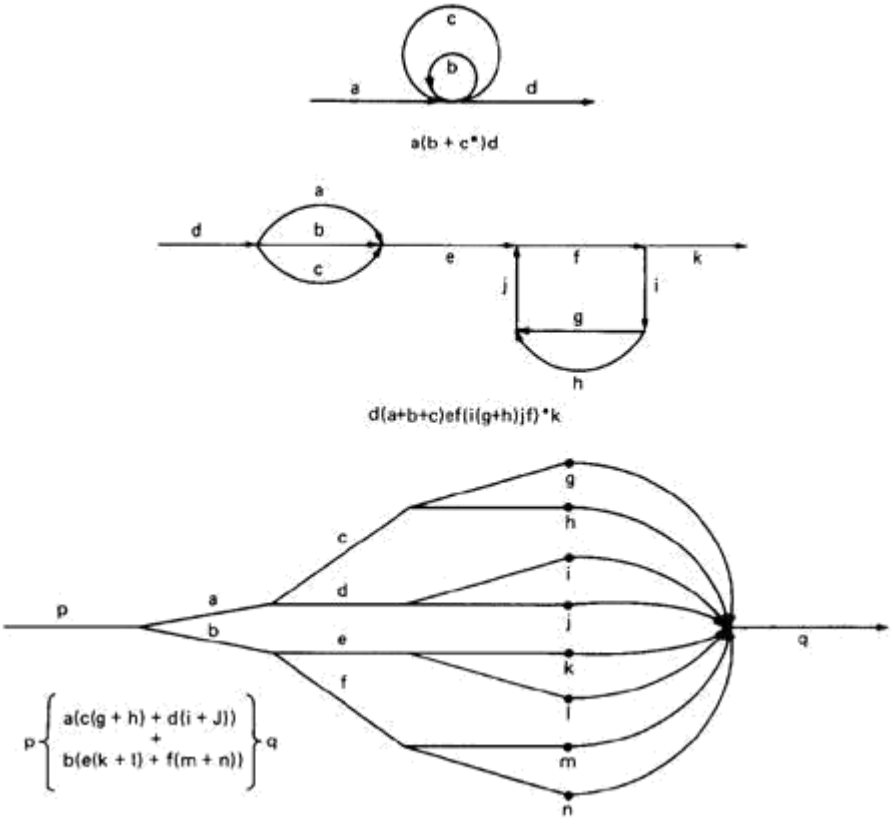


Figure 5.6: Some graphs and their path expressions.

APPLICATIONS:

The purpose of the node removal algorithm is to present one very generalized concept-the path expression and way of getting it.

Every application follows this common pattern:

Convert the program or graph into a path expression.

Identify a property of interest and derive an appropriate set of "arithmetic" rules that characterizes the property.

Replace the link names by the link weights 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.

Simplify or evaluate the resulting "algebraic" expression to answer the question you asked.

HOW MANY PATHS IN A FLOW GRAPH ?

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

What is the maximum number of different paths possible?

What is the fewest number of paths possible?

How many different paths are there really?

What is the average number of paths?

Determining the actual number of different paths is an inherently difficult problem because there could be unachievable paths resulting from correlated and dependent predicates.

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.

Asking for "the average number of paths" is meaningless.

MAXIMUM PATH COUNT ARITHMETIC:

Label each link with a link weight that corresponds to the number of paths that link represents.

Also mark each loop with the maximum number of times that loop can be taken. If the answer is infinite, you might as well stop the analysis because it is clear that the maximum number of paths will be infinite.

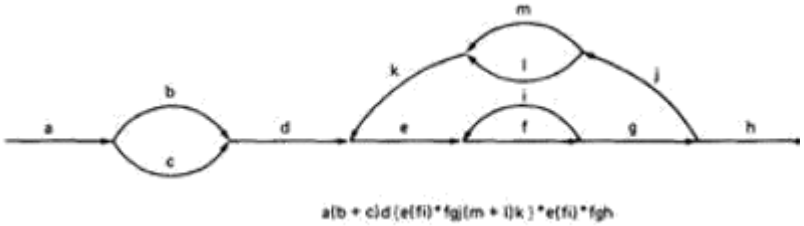
There are three cases of interest: parallel links, serial links, and loops.

Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	W_AW_B
Loop	A^n	$\sum_{j=0}^{n} W_A^j$

This arithmetic is an ordinary algebra. The weight is the number of paths in each set.

EXAMPLE:

The following is a reasonably well-structured program.



a(b + c)d {e(fi)*fgj(m + l)k }*e(fi)*fgh

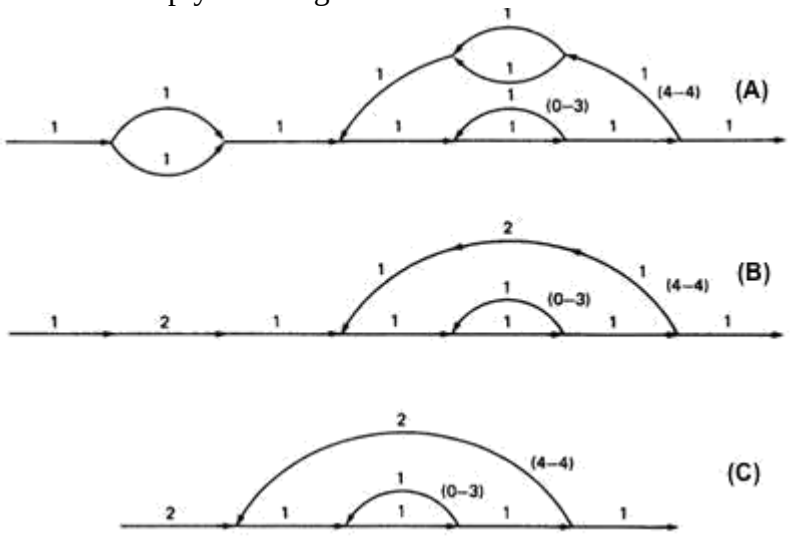
Each link represents a single link and consequently is given a weight of "1" to start. Let's say the outer loop will be taken exactly four times and inner Loop Can be taken zero or three times Its path expression, with a little work, is:

Path expression: a(b+c)d{e(fi)*fgj(m+l)k}*e(fi)*fgh

A: The flow graph should be annotated by replacing the link name with the maximum of paths through that link (1) and also note the number of times for looping.

B: Combine the first pair of parallel loops outside the loop and also the pair in the outer loop.

C: Multiply the things out and remove nodes to clear the clutter.



For the Inner Loop:

D:Calculate the total weight of inner loop, which can execute a min. of 0 times and max. of 3 times. So, it inner loop can be evaluated as follows:

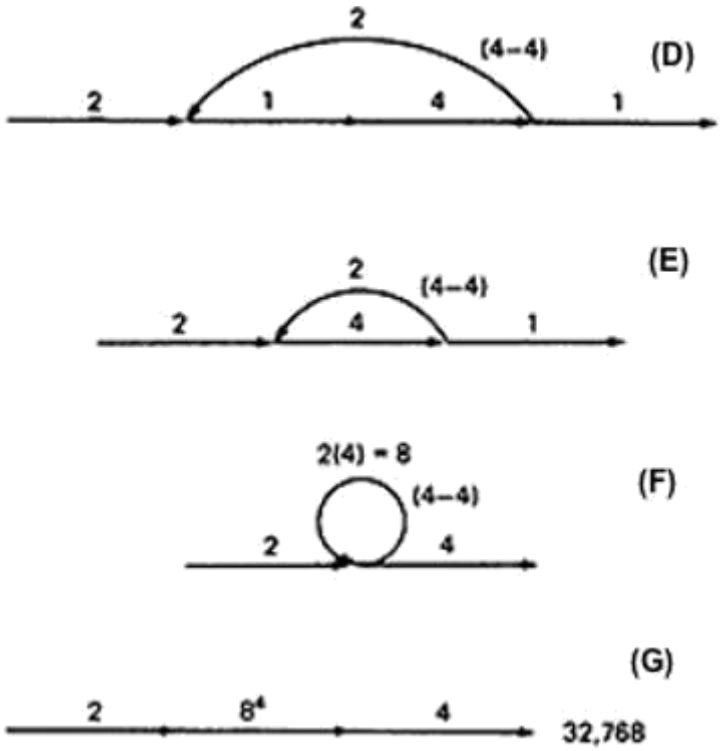
$1^{\underline{3}}=1^0+1^1+1^2+1^3=1+1+1+1=4$

E: Multiply the link weights inside the loop: 1 X 4 = 4

F: Evaluate the loop by multiplying the link wieghts: 2 X 4 = 8.

G: Simpifying the loop further results in the total maximum number of paths in the flowgraph:

$2 \text{ X } 8^4 \text{ X } 2 = 32{,}768.$



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

a(b+c)d{e(fi)*fgj(m+l)k}*e(fi)*fgh

$1(1 + 1)1(1(1 \times 1)^{\overline{3}}1 \times 1 \times 1(1 + 1)1)^{\overline{4}}1(1 \times 1)^{\overline{3}}1 \times 1 \times 1$

$2(1^{\overline{3}}1 \times (2))^{\overline{4}}1^{\overline{3}}$

$2(4 \times 2)^{\overline{4}} \times 4$

$2 \times 8^{\overline{4}} \times 4 = 32{,}768$

This is the same result we got graphically.Actually, the outer loop should 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 the last step. Therefore the maximum number of different paths is 8192 rather than 32,768.

STRUCTURED FLOWGRAPH:

Structured code can be defined in several different ways that do not involve ad-hoc rules such as not using GOTOs.

A structured flowgraph is one that can be reduced to a single link by successive application of the transformations of Figure 5.7.

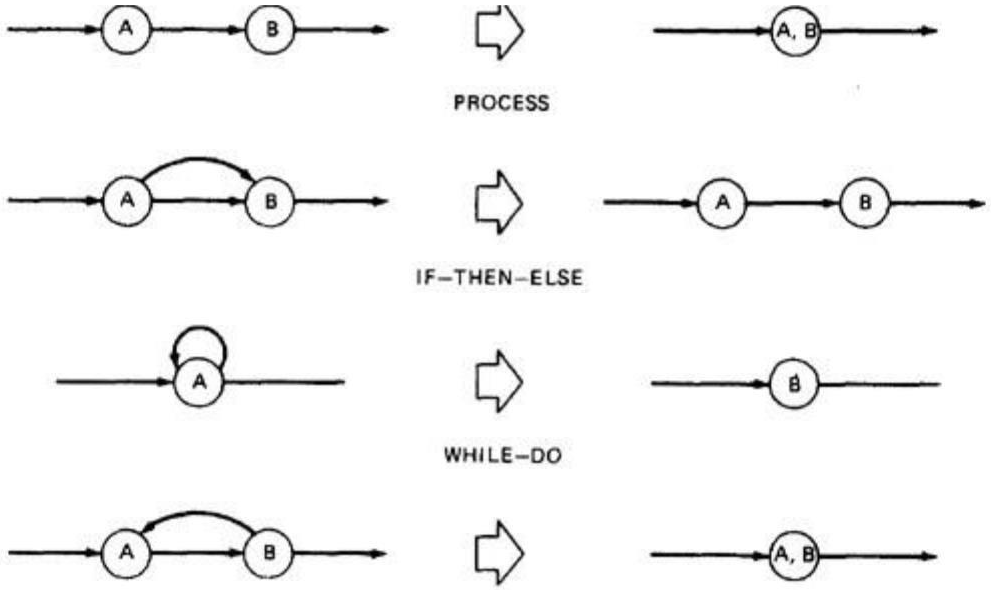


Figure 5.7: Structured Flowgraph Transformations.

The node-by-node reduction procedure can also be used as a test for structured code.Flow graphs that DO NOT contain one or more of the graphs shown below (Figure 5.8) as subgraphs are structured.

Jumping into loops
Jumping out of loops
Branching into decisions
Branching out of decisions

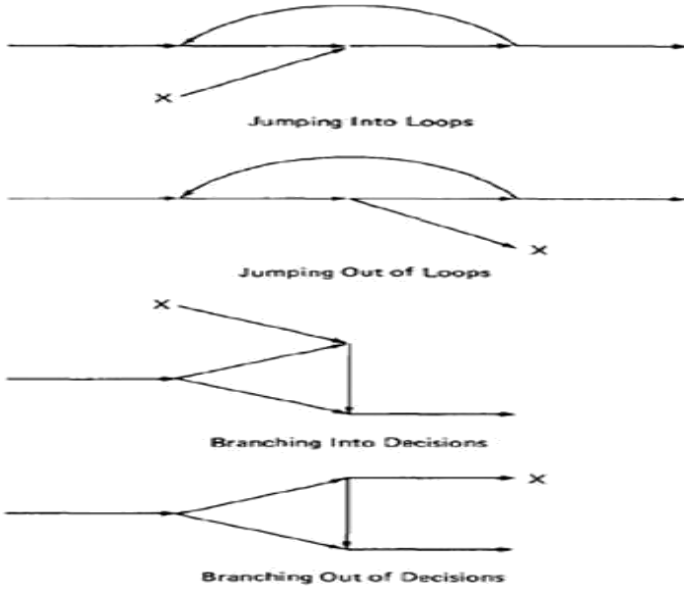


Figure 5.8: Un-structured sub-graphs.

LOWER PATH COUNT ARITHMETIC:

A lower bound on the number of paths in a routine can be approximated for structured flow graphs.

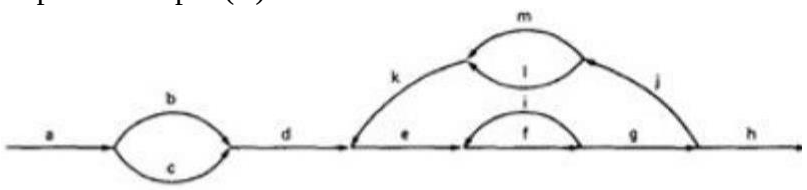
The arithmetic is as follows:

Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	$\max(W_A,W_B)$
Loop	A^n	1, W_1

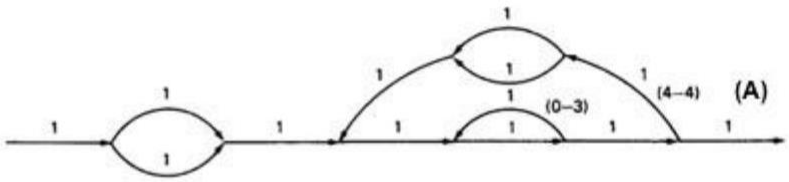
The values of the weights are the number of members in a set of paths.

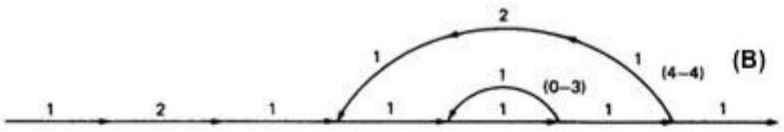
EXAMPLE:

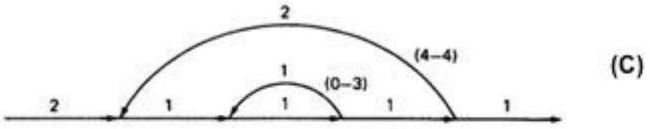
Applying the arithmetic to the earlier example gives us the identical steps unitl step 3 (C) as below:



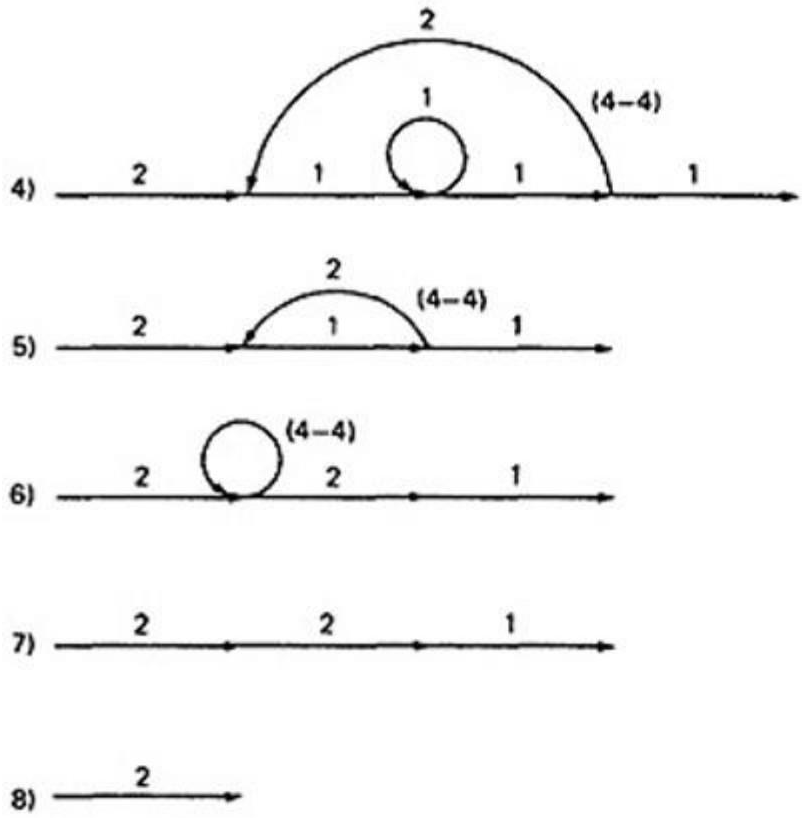
a(b + c)d (e(fi)*fgj(m + i)k)*e(fi)*fgh







- From Step 4, the it would be different from the previous example:



If you observe the original graph, it takes at least two paths to cover and that it can be done in two paths.
If you have fewer paths in your test plan than this minimum you probably haven't covered. It's another check.

CALCULATING THE PROBABILITY:
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.
We use the same algorithm as before: node-by-node removal of uninteresting nodes.

Weights, Notations and Arithmetic:

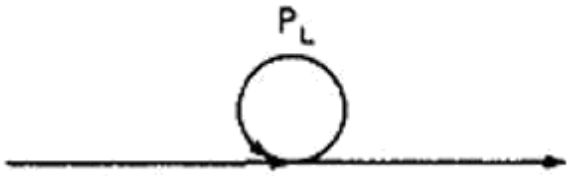
Probabilities can come into the act only at decisions (including decisions associated with loops).
Annotate each outlink with a weight equal to the probability of going in that direction.
Evidently, the sum of the outlink probabilities must equal 1
For a simple loop, if the loop will be taken a mean 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_AP_B
Loop	A*	$P_A / (1-P_L)$

In this table, in case of a loop, 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.

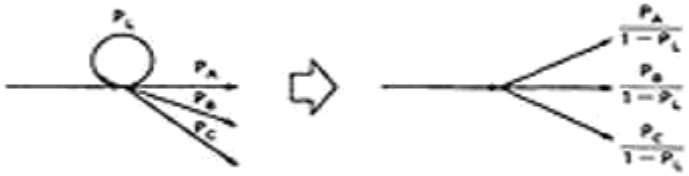
For example, a loop node has a looping probability of P_L and a probability of not looping of P_A, which is obviously equal to I - PL.



$$P_A = 1 - P_L$$

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

Following the above 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:



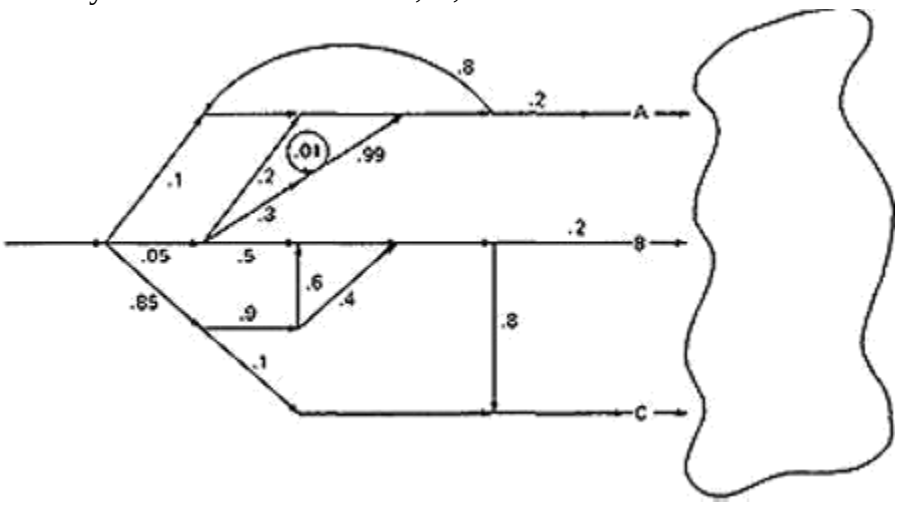
because $P_L + P_A + P_B + P_C = 1$, $1 - P_L = P_A + P_B + P_C$, and

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

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.

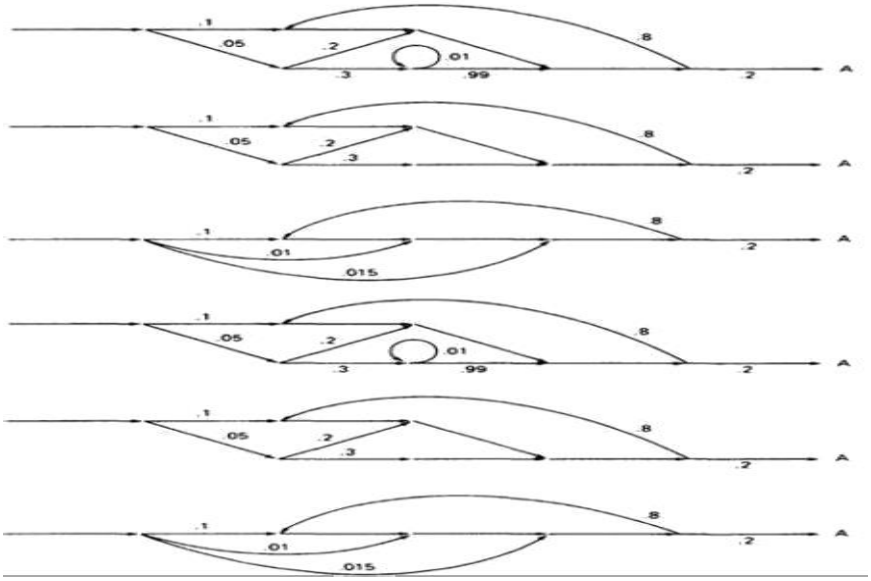
EXAMPLE:

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

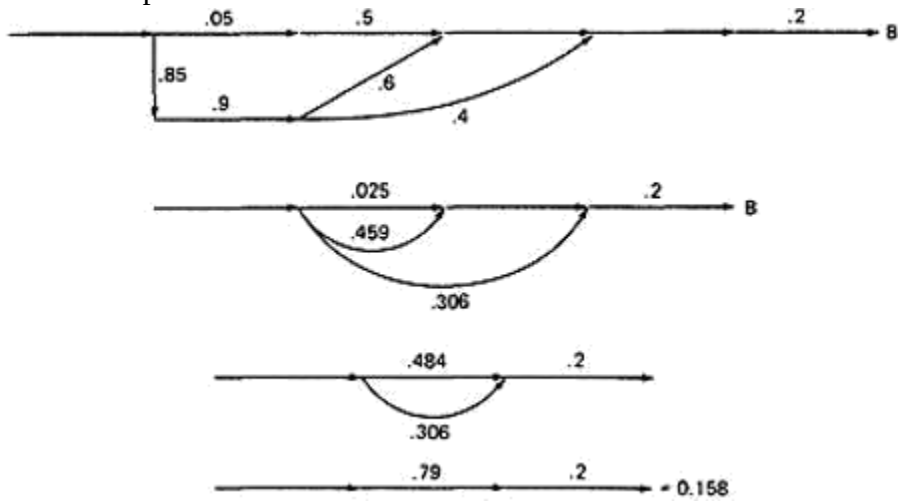


Let us do this in three parts, starting with case A. Note that the sum of the probabilities at each decision node is equal to 1. Start by throwing away anything that isn't on the way to case A, and then apply the reduction procedure. To avoid clutter, we usually leave out probabilities equal to 1.

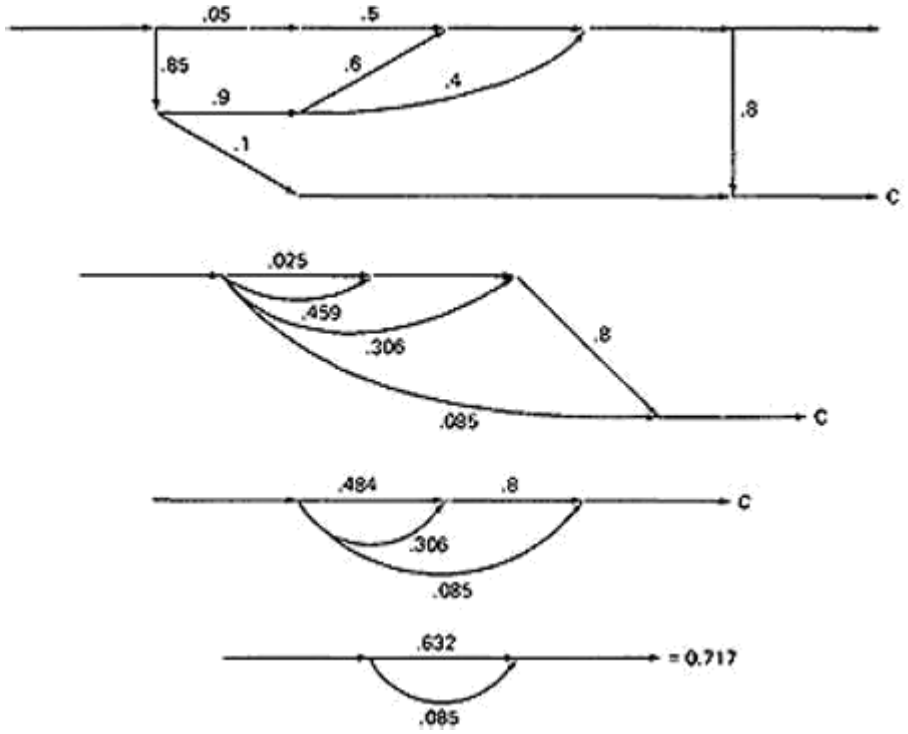
CASE A:



Case B is simpler:



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



These 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 bra 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 I respectively. Path *abcbcbcdeabddea* would have a probability of 5×10^{-10}.
Long paths are usually improbable.

MEAN PROCESSING TIME OF A ROUTINE:

Given the execution time of all statements or instructions for every link in a flowgraph and the probability for each direction for all decisions are to find the mean processing time for the routine as a whole.

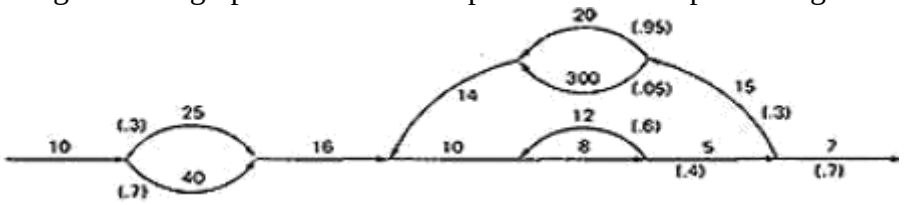
The model has two weights associated with every link: the processing time for that link, denoted by **T**, and the probability of that link **P**.

The arithmetic rules for calculating the mean time:

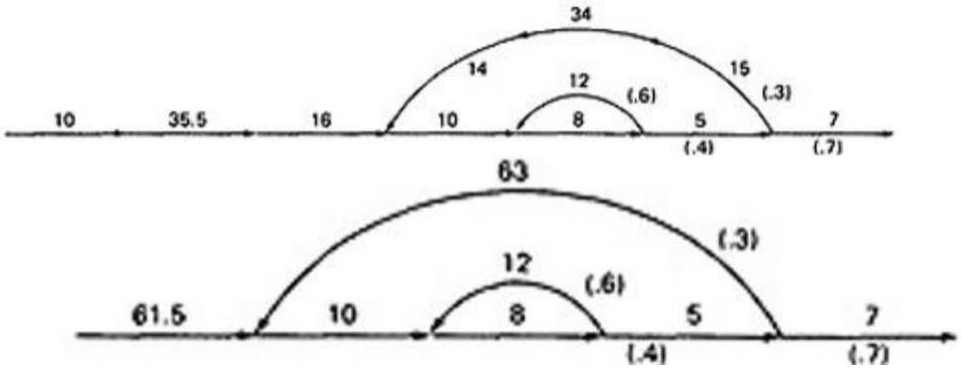
Case	Path expression	Weight expression
Parallel	A+B	$T_{A+B}=(P_AT_A+P_BT_B)/(P_A+P_B)$ $P_{A+B}= P_A+P_B$
Series	AB	$T_{AB}=T_A+T_B$ $P_{AB} =P_AP_B$
Loop	A^n	$T_A= T_A+ T_LP_L/(1-P_L)$ $P_A =P_A/(1-P_L)$

EXAMPLE:

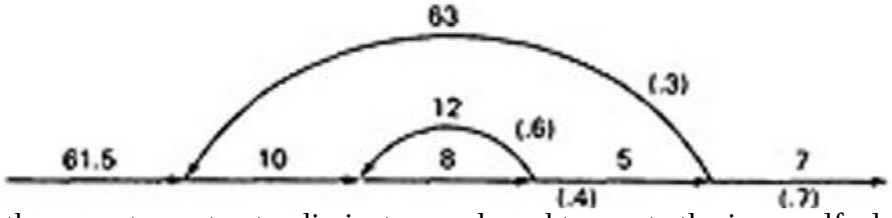
1. Start with the original flow graph annotated with probabilities and processing time.



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 flow graph.

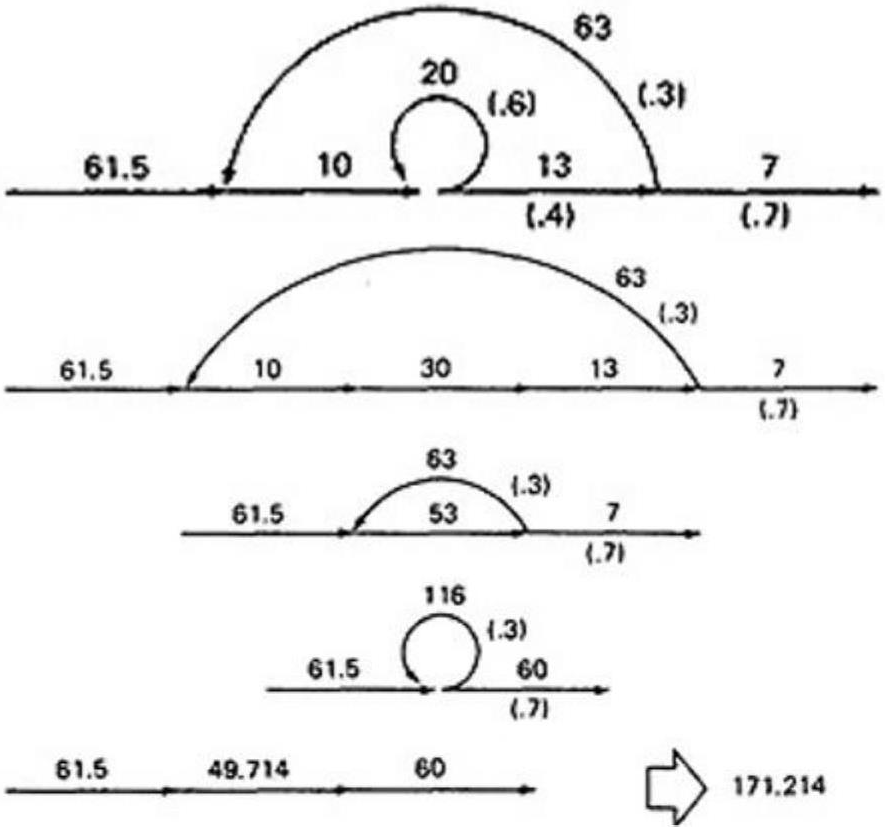


3. Combine as many serial links as you can.



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

5. Finally, you can get the mean processing time, by using the arithmetic rules as follows:



PUSH/POP, GET/RETURN:

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.

EXAMPLE 1 (PUSH / POP):

- Here is the Push/Pop Arithmetic:

Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	W_AW_B
Loop	A^*	W_A^*

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.

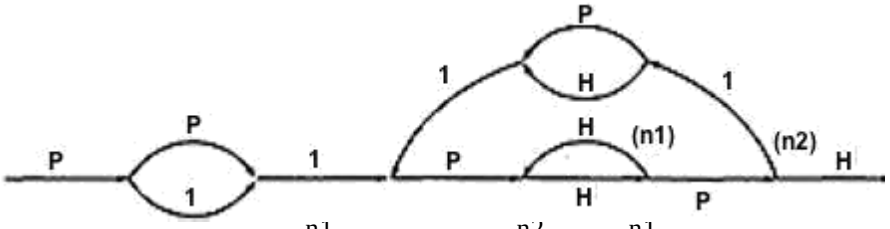
PUSH/POP MULTIPLICATION TABLE

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

Consider the following flow graph:



$P(P + 1)1\{P(HH)^{n1}HP1(P + H)1\}^{n2}P(HH)^{n1}HPH$

Simplifying by using the arithmetic tables,

$(P^2 + P)\{P(HH)^{n1}(P + H)\}^{n1}(HH)^{n1}$

$(P^2 + P)\{H^{2n1}(P^2 + 1)\}^{n2}H^{2n1}$

Below Table 5.9 shows several combinations of values for the two looping terms - M1 is the number of times the inner loop will be taken and M2 the number of times the outer loop will be taken.

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

Figure 5.9: Result of the PUSH / POP Graph Analysis.

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.

EXAMPLE 2 (GET / RETURN):

Exactly the same arithmetic tables used for previous example 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.
The arithmetic tables for GET/RETURN are:

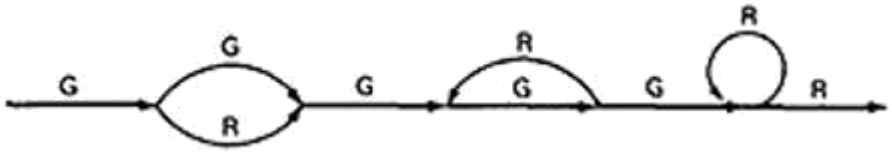
Multiplication Table

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

Addition Table

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

"G" denotes GET and "R" denotes RETURN.
Consider the following flowgraph:



$G(G + R)G(GR)^*GGR^*R$
$G(G + R)G^3R^*R$
$(G + R)G^3R^*$
$(G^4 + G^2)R^*$
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.

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.

REGULAR EXPRESSIONS AND FLOW ANOMALY DETECTION:

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 the operations be 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 (an *ss* or an *rr* sequence).

Some more application examples:

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.

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?

The data-flow anomalies discussed in Unit 4 requires us to detect the *dd*, *dk*, *kk*, and *ku* sequences. Are there paths with anomalous data flows?

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 12 = 1.

- 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.
- **EXAMPLE:** 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. (**HUANG's Theorem**)

  As an example,

  let
  - A = *pp*
  - B = *srr*
  - C = *rp*
  - T = *ss*

The theorem states that *ss* will appear in $pp(srr)^nrp$ if it appears in $pp(srr)^2rp$.

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

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

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

Incidentally, the above observation is an informal proof of the wisdom of looping twice discussed in Unit 2. Because data-flow anomalies are represented by two-character sequences, it follows the above theorem that looping twice is what you need to do to find such anomalies.

LIMITATIONS:

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.

There are some nice theorems for finding sequences that occur at the beginnings and ends of strings but no nice algorithms for finding strings buried in an expression.

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.

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.

UNIT IV(Part-II)
LOGIC BASED TESTING

OVERVIEW OF LOGIC BASED TESTING:

INTRODUCTION:

The functional requirements of many programs can be specified by **decision tables**, which provide a useful basis for program and test design.

Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using **Karnaugh-Veitch charts**.

"Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques.

Boolean algebra is to logic as arithmetic is to mathematics. Without it, the tester or programmer is cut off from many test and design techniques and tools that incorporate those techniques.

Logic has been, for several decades, the primary tool of hardware logic designers.

- Many test methods developed for hardware logic can be adapted to software logic testing. Because hardware testing automation is 10 to 15 years ahead of software testing automation, hardware testing methods and its associated theory is a fertile ground for software testing methods.
- As programming and test techniques have improved, the bugs have shifted closer to the process front end, to requirements and their specifications. These bugs range from 8% to 30% of the total and because they're first-in and last-out, they're the costliest of all.
- The trouble with specifications is that they're hard to express.
- Boolean algebra (also known as the sentential calculus) is the most basic of all logic systems.
- Higher-order logic systems are needed and used for formal specifications.
- Much of logical analysis can be and is embedded in tools. But these tools incorporate methods to simplify, transform, and check specifications, and the methods are to a large extent based on boolean algebra.

KNOWLEDGE BASED SYSTEM:

The **knowledge-based system** (also expert system, or "artificial intelligence" system) has become the programming construct of choice for many applications that were once considered very difficult.

Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database. The data can then be queried and interacted with to provide solutions to problems in that domain.

One implementation of knowledge-based systems is to incorporate the expert's knowledge into a set of rules. The user can then provide data and ask questions based on that data.

The user's data is processed through the rule base to yield conclusions (tentative or definite) and requests for more data. The processing is done by a program called the **inference engine**.

Understanding knowledge-based systems and their validation problems requires an understanding of formal logic.

Decision tables are extensively used in business data processing; Decision-table preprocessors as extensions to COBOL are in common use; boolean algebra is embedded in the implementation of these processors.

Although programmed tools are nice to have, most of the benefits of boolean algebra can be reaped by wholly manual means if you have the right conceptual tool: the Karnaugh-Veitch diagram is that conceptual tool.

DECISION TABLES:

Figure 6.1 is a limited - entry decision table. It consists of four areas called the condition stub, the condition entry, the action stub, and the action entry.

Each column of the table is a rule that specifies the conditions under which the actions named in the action stub will take place.

The condition stub is a list of names of conditions.

CONDITION ENTRY

		RULE 1	RULE 2	RULE 3	RULE 4
CONDITION STUB	CONDITION 1	YES	YES	NO	NO
	CONDITION 2	YES	I	NO	I
	CONDITION 3	NO	YES	NO	I
	CONDITION 4	NO	YES	NO	YES
ACTION STUB	ACTION 1	YES	YES	NO	NO
	ACTION 2	NO	NO	YES	NO
	ACTION 3	NO	NO	NO	YES

ACTION ENTRY

Figure 6.1 : Examples of Decision Table.

A more general decision table can be as below:

Printer troubleshooter

		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Figure 6.2 : Another Examples of Decision Table.

A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "I" means that the condition plays no part in the rule, or it is immaterial to that rule.

The action stub names the actions the routine will take or initiate if the rule is satisfied.

If the action entry is "YES", the action will take place; if "NO", the action will not take place.

The table in Figure 6.1 can be translated as follows:

Action 1 will take place if conditions 1 and 2 are met and if conditions 3 and 4 are not met (rule or if conditions 1, 3, and 4 are met (rule 2).

"Condition" is another word for predicate.

Decision-table uses "condition" and "satisfied" or "met". Let us use "predicate" and TRUE / FALSE.

Now the above translations become:

Action 1 will be taken if predicates 1 and 2 are true and if predicates 3 and 4 are false (rule 1), or if predicates 1, 3, and 4 are true (rule 2).

Action 2 will be taken if the predicates are all false, (rule 3).

Action 3 will take place if predicate 1 is false and predicate 4 is true (rule 4).

In addition to the stated rules, we also need a **Default Rule** that specifies the default action to be taken when all other rules fail. The default rules for Table in Figure 6.1 is shown in Figure 6.3

	Rule 5	Rule 6	Rule 7	Rule 8
CONDITION 1	I	NO	YES	YES
CONDITION 2	I	YES	I	NO
CONDITION 3	YES	I	NO	NO
CONDITION 4	NO	NO	YES	I
DEFAULT ACTION	YES	YES	YES	YES

Figure 6.3 : The default rules of Table in Figure 6.1

DECISION-TABLE PROCESSORS:

Decision tables can be automatically translated into code and, as such, are a higher-order language

If the rule is satisfied, the corresponding action takes place

Otherwise, rule 2 is tried. This process continues until either a satisfied rule results in an action or no rule is satisfied and the default action is taken

Decision tables have become a useful tool in the programmers kit, in business data processing.

DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:

The specification is given as a decision table or can be easily converted into one.
The order in which the predicates are evaluated does not affect interpretation of the rules or the resulting action - i.e., an arbitrary permutation of the predicate order will not, or should not, affect which action takes place.
The order in which the rules are evaluated does not affect the resulting action - i.e., an arbitrary permutation of rules will not, or should not, affect which action takes place.
Once a rule is satisfied and an action selected, no other rule need be examined.
If several actions can result from satisfying a rule, the order in which the actions are executed doesn't matter.

DECISION-TABLES AND STRUCTURE:

Decision tables can also be used to examine a program's structure.
- Figure 6.4 shows a program segment that consists of a decision tree.
- These decisions, in various combinations, can lead to actions 1, 2, or 3.

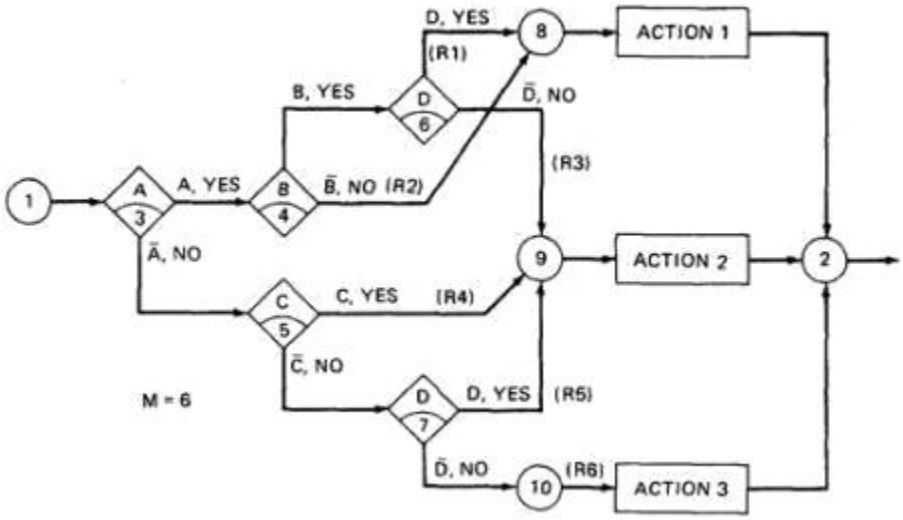


Figure 6.4 : A Sample Program

If the decision appears on a path, put in a YES or NO as appropriate. If the decision does not appear on the path, put in an I, Rule 1 does not contain decision C, therefore its entries are: YES, YES, I, YES.

- The corresponding decision table is shown in Table 6.1

	RULE 1	RULE 2 I	ULE 3 F	ULE 4 R	ULE 5 R	JLE 6
CONDITION A **CONDITION B** **CONDITION C** **CONDITION D**	YES YES I YES	YES NO I I	YES YES I NO	NO I YES I	NO I NO YES	NO I NO NO
ACTION 1 **ACTION 2** **ACTION 3**	YES NO NO	YES NO NO	NO YES NO	NO YES NO	NO YES NO	NO NO YES

Table 6.1: Decision Table corresponding to Figure 6.4

As an example, expanding the immaterial cases results as below:

	RULE 1	RULE 2
CONDITION 1	YES	YES
CONDITION 2	I	NO
CONDITION 3	YES	I
CONDITION 4	NO	NO
ACTION 1	YES	NO
ACTION 2	NO	YES

RULE 1.1	RULE 1.2	RULE 2.1	RULE 2.2
YES	YES	YES	YES
YES	NO	NO	NO
YES	YES	YES	NO
NO	NO	NO	NO
YES	YES	NO	NO
NO	NO	YES	YES

Table 6.2: Expansion of Table 6.1

Similalrly, If we expand the immaterial cases for the above Table 6.1, it results in Table 6.2 as below:

	R 1	RULE 2	R 3	RULE 4	R 5	R 6
CONDITION A	YY	YYYY	YY	NNNN	NN	NN
CONDITION B	YY	NNNN	YY	YYNN	NY	YN
CONDITION C	YN	NNYY	YN	YYYY	NN	NN
CONDITION D	YY	YNNY	NN	NYYN	YY	NN

Sixteen cases are represented in Table 6.1, and no case appears twice. Consequently, the flowgraph appears to be complete and consistent.

As a first check, before you look for all sixteen combinations, count the number of Y's and N's in each row. They should be equal. We can find the bug that way.

ANOTHER EXAMPLE - A TROUBLE SOME PROGRAM:

Consider the following specification whose putative flowgraph is shown in Figure 6.5:

> If condition A is met, do process A1 no matter what other actions are taken or what other conditions are met.

If condition B is met, do process A2 no matter what other actions are taken or what other conditions are met.
If condition C is met, do process A3 no matter what other actions are taken or what other conditions are met.
If none of the conditions is met, then do processes A1, A2, and A3.
When more than one process is done, process A1 must be done first, then A2, and then A3. The only permissible cases are: (A1), (A2), (A3), (A1,A3), (A2,A3) and (A1,A2,A3).

Figure 6.5 shows a sample program with a bug.

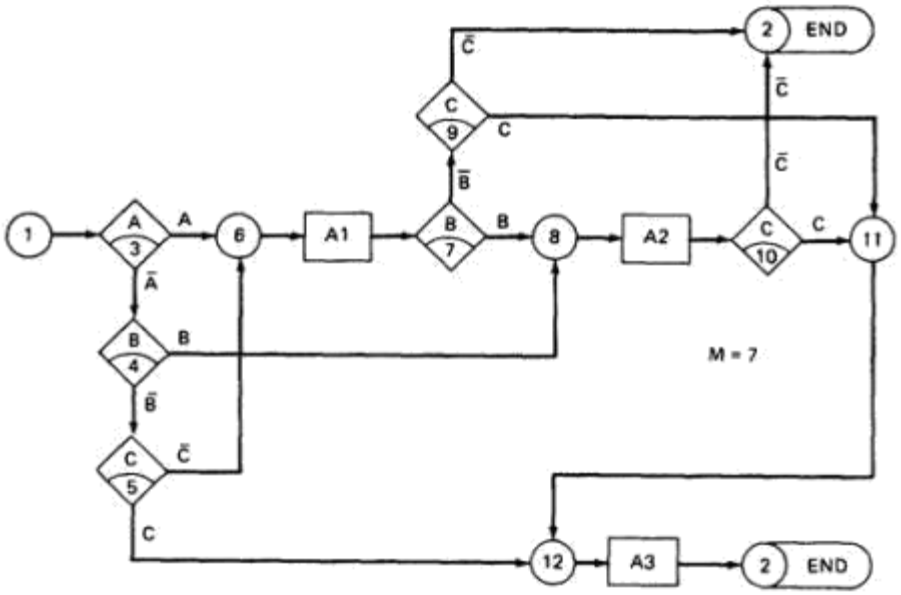


Figure 6.5 : A Troublesome Program

The programmer tried to force all three processes to be executed for the $\bar{A}\bar{B}\bar{C}$ cases but forgot that the B and C predicates would be done again, thereby bypassing processes A2 and A3.
Table 6.3 shows the conversion of this flow graph into a decision table after expansion.

	RULES							
	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}B\bar{C}$	$\bar{A}BC$	$A\bar{B}\bar{C}$	$A\bar{B}C$	$AB\bar{C}$	ABC
CONDITION A	NO	NO	NO	NO	YES	YES	YES	YES
CONDITION B	NO	NO	YES	YES	YES	YES	NO	NO
CONDITION C	NO	YES	YES	NO	NO	YES	YES	NO
ACTION 1	YES	NO	NO	NO	YES	YES	YES	YES
ACTION 2	YES	NO	YES	YES	YES	YES	NO	NO
ACTION 3	YES	YES	YES	NO	NO	YES	YES	NO

Table 6.3: Decision Table for Figure 6.5

PATH EXPRESSIONS:

- **GENERAL:**

  Logic-based testing is structural testing when it's applied to structure (e.g., control flow graph of an implementation); it's functional testing when it's applied to a specification.

  - In logic-based testing we focus on the truth values of control flow predicates.

A **predicate** is implemented as a process whose outcome is a truth-functional value.

For our purpose, logic-based testing is restricted to binary predicates.

We start by generating path expressions by path tracing as in Unit V, but this time, our purpose is to convert the path expressions into boolean algebra, using the predicates' truth values (e.g., A and $\overline{A}$) as weights.

- **BOOLEAN ALGEBRA:**

STEPS:

Label each decision with an uppercase letter that represents the truth value of the predicate. The YES or TRUE branch is labeled with a letter (say A) and the NO or FALSE branch with the same letter overscored (say $\overline{A}$).

The truth value of a path is the product of the individual labels. Concatenation or products mean "AND". For example, the straight-through path of Figure 6.5, which goes via nodes 3, 6, 7, 8, 10, 11, 12, and 2, has a truth value of ABC. The path via nodes 3, 6, 7, 9 and 2 has a value of $AB\overline{C}$.

If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR".

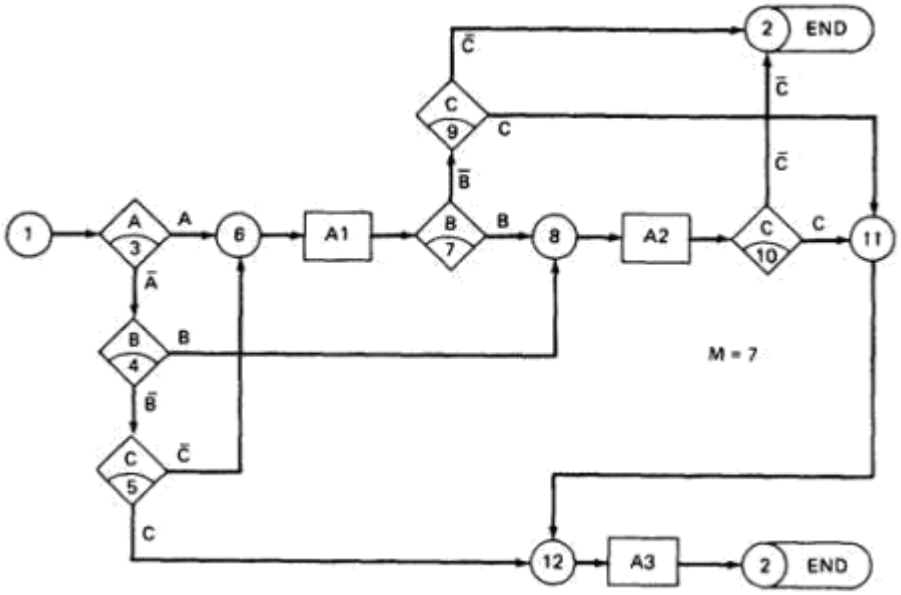


Figure 6.5: A Troublesome Program

Using this convention, the truth-functional values for several of the nodes can be expressed in terms of segments from previous nodes. Use the node name to identify the point.

$$N6 = A + \overline{A}\,\overline{B}\,\overline{C}$$
$$N8 = (N6)B + \overline{A}B = AB + \overline{A}\,\overline{B}\,\overline{C}B + \overline{A}B$$
$$N11 = (N8)C + (N6)\overline{B}C$$
$$N12 = N11 + \overline{A}\,\overline{B}C$$
$$N2 = N12 + (N8)\overline{C} + (N6)\overline{B}\,\overline{C}$$

There are only two numbers in boolean algebra: zero (0) and one (1). One means "always true" and zero means "always false".

RULES OF BOOLEAN ALGEBRA:

Boolean algebra has three operators: X (AND), + (OR) and $\bar{A}$ (NOT)

X : meaning AND. Also called multiplication. A statement such as AB (A X B) means "A and B are both true". This symbol is usually left out as in ordinary algebra.

+ : meaning OR. "A + B" means "either A is true or B is true or both".

$\bar{A}$ meaning NOT. Also negation or complementation. This is read as either "not A" or "A bar". The entire expression under the bar is negated.

The following are the laws of boolean algebra:

1. $A + A$	$= A$	If something is true, saying it
$\bar{A} + \bar{A}$	$= \bar{A}$	twice doesn't make it truer, ditto for falsehoods.
2. $A + 1$	$= 1$	If something is always true, then "either A or true or both" must also be universally true.
3. $A + 0$	$= A$	
4. $A + B$	$= B + A$	Commutative law.
5. $A + \bar{A}$	$= 1$	If either A is true or not-A is true, then the statement is always true.
6. AA	$= A$	
$\bar{A}\bar{A}$	$= \bar{A}$	
7. $A \times 1$	$= A$	
8. $A \times 0$	$= 0$	
9. AB	$= BA$	
10. $A\bar{A}$	$= 0$	A statement can't be simultaneously true and false.
11. $\bar{\bar{A}}$	$= A$	"You ain't not going" means you are. How about, "I ain't not never going to get this nohow."?
12. $\bar{0}$	$= 1$	
13. $\bar{1}$	$= 0$	
14. $\overline{A + B}$	$= \bar{A}\bar{B}$	Called "De Morgan's theorem or law."
15. $\overline{AB}$	$= \bar{A} + \bar{B}$	
16. $A(B + C)$	$= AB + AC$	Distributive law.
17. $(AB)C$	$= A(BC)$	Multiplication is associative.
18. $(A + B) + C$	$= A + (B + C)$	So is addition.
19. $A + \bar{A}B$	$= A + B$	Absorptive law.
20. $A + AB$	$= A$	

In all of the above, a letter can represent a single sentence or an entire boolean algebra expression.

Individual letters in a boolean algebra expression are called **Literals** (e.g. A,B) The product of several literals is called a **product term** (e.g., ABC, DE).

An arbitrary boolean expression that has been multiplied out so that it consists of the sum of products (e.g., ABC + DEF + GH) is said to be in **sum-of-products form**.

The result of simplifications (using the rules above) is again in the sum of product form and each product term in such a simplified version is called a **prime implicant**. For example, ABC + AB DEF reduce by rule 20 to AB + DEF; that is, AB and DEF are prime implicants. The path expressions of Figure 6.5 can now be simplified by applying the rules. The following are the laws of boolean algebra:

$$
\begin{aligned}
N6 &= A + \overline{A}\,\overline{B}\overline{C} \\
&= A + \overline{B}\overline{C} && \text{: Use rule 19, with "B" = } \overline{B}\overline{C}. \\
N8 &= (N6)B + \overline{A}B \\
&= (A + \overline{B}\overline{C})B + \overline{A}B && \text{: Substitution.} \\
&= AB + \overline{B}\overline{C}B + \overline{A}B && \text{: Rule 16 (distributive law).} \\
&= AB + B\overline{B}C + \overline{A}B && \text{: Rule 9 (commutative multiplication).} \\
&= AB + 0C + \overline{A}B && \text{: Rule 10.} \\
&= AB + 0 + \overline{A}B && \text{: Rule 8.} \\
&= AB + \overline{A}B && \text{: Rule 3.} \\
&= (A + \overline{A})B && \text{: Rule 16 (distributive law).} \\
&= 1 \times B && \text{: Rule 5.} \\
&= B && \text{: Rules 7, 9.}
\end{aligned}
$$

Similarly,

$$
\begin{aligned}
N11 &= (N8)C + (N6)\overline{B}C \\
&= BC + (A + \overline{B}\overline{C})\overline{B}C && \text{: Substitution.} \\
&= BC + A\overline{B}C && \text{: Rules 16, 9, 10, 8, 3.} \\
&= C(B + \overline{B}A) && \text{: Rules 9, 16.} \\
&= C(B + A) && \text{: Rule 19.} \\
&= AC + BC && \text{: Rules 16, 9, 9, 4.} \\
N12 &= N11 + \overline{A}\,\overline{B}C \\
&= AC + BC + \overline{A}\,\overline{B}C \\
&= C(B + \overline{A}\,\overline{B}) + AC \\
&= C(\overline{A} + B) + AC \\
&= C\overline{A} + AC + BC \\
&= C + BC \\
&= C \\
N2 &= N12 + (N8)\overline{C} + (N6)\overline{B}\,\overline{C} \\
&= C + B\overline{C} + (A + \overline{B}\overline{C})\overline{B}\,\overline{C} \\
&= C + B\overline{C} + \overline{B}\,\overline{C} \\
&= C + \overline{C}(B + \overline{B}) \\
&= C + \overline{C} \\
&= 1
\end{aligned}
$$

The deviation from the specification is now clear. The functions should have been:

$$
\begin{aligned}
N6 &= A + \overline{A}\,\overline{B}\overline{C} = A + \overline{B}\overline{C} && \text{: correct.} \\
N8 &= B + \overline{A}\,\overline{B}\overline{C} = B + \overline{A}\overline{C} && \text{: wrong, was just B.} \\
N12 &= C + \overline{A}\,\overline{B}\overline{C} = C + \overline{A}\,\overline{B} && \text{: wrong, was just C.}
\end{aligned}
$$

Loops complicate things because we may have to solve a boolean equation to determine what predicate value combinations lead to where.

KV CHARTS:

- **INTRODUCTION:**

  If you had to deal with expressions in four, five, or six variables, you could get bogged down in the algebra and make as many errors in designing test cases as there are bugs in the routine you're testing.

  Karnaugh-Veitch chart reduces boolean algebraic manipulations to graphical trivia.

  Beyond six variables these diagrams get cumbersome and may not be effective.

- **SINGLE VARIABLE:**
  - Figure 6.6 shows all the boolean functions of a single variable and their equivalent representation as a KV chart.

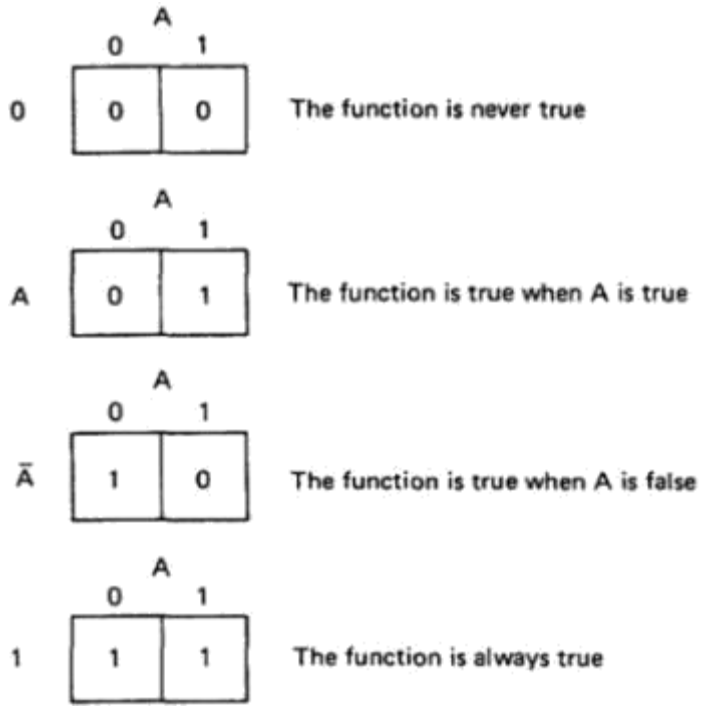


Figure 6.6 : KV Charts for Functions of a Single Variable.

The charts show all possible truth values that the variable A can have. A "1" means the variable's value is "1" or TRUE. A "0" means that the variable's value is 0 or FALSE.

The entry in the box (0 or 1) specifies whether the function that the chart represents is true or false for that value of the variable.

We usually do not explicitly put in 0 entries but specify only the conditions under which the function is true.

- **TWO VARIABLES:**
  - Figure 6.7 shows eight of the sixteen possible functions of two variables.

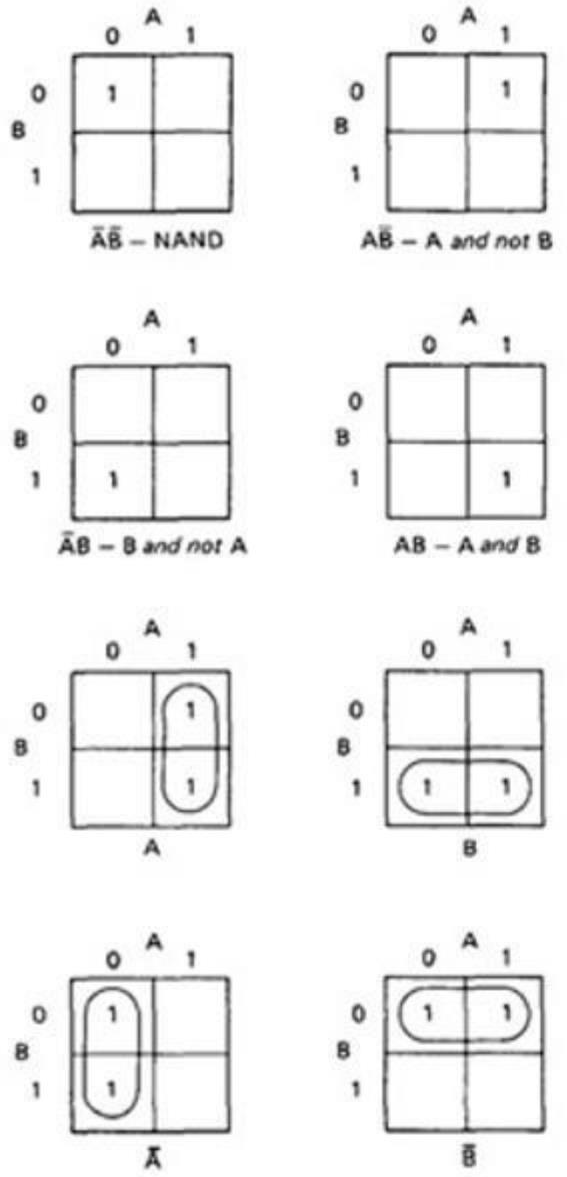


Figure 6.7: KV Charts for Functions of Two Variables.

Each box corresponds to the combination of values of the variables for the row and column of that box.

A pair may be adjacent either horizontally or vertically but not diagonally.

Any variable that changes in either the horizontal or vertical direction does not appear in the expression.

In the fifth chart, the B variable changes from 0 to 1 going down the column, and because the A variable's value for the column is 1, the chart is equivalent to a simple A.

- Figure 6.8 shows the remaining eight functions of two variables.

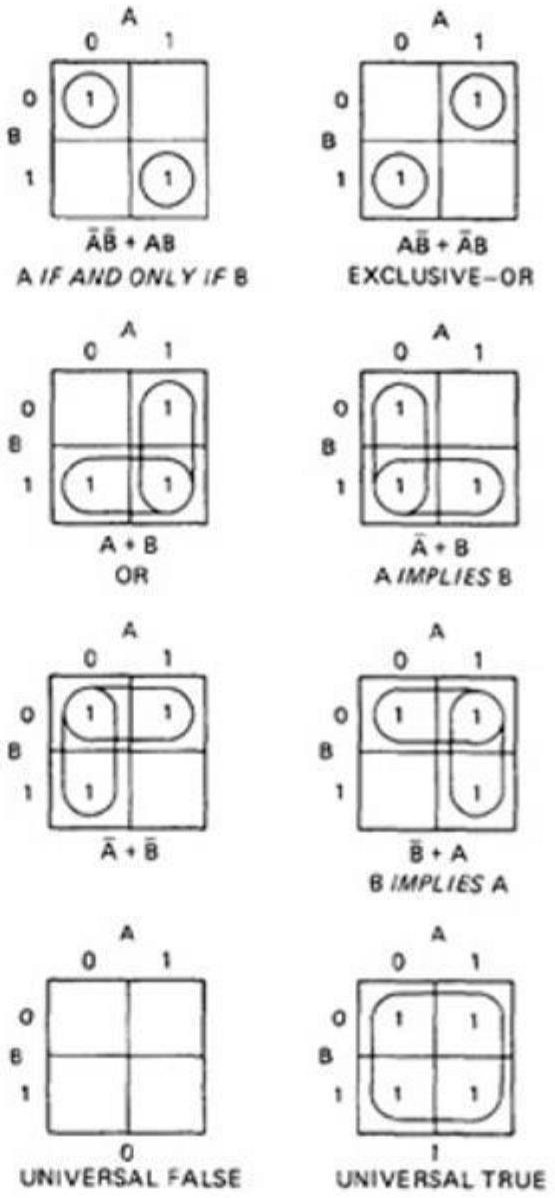


Figure 6.8: More Functions of Two Variables.

The first chart has two 1's in it, but because they are not adjacent, each must be taken separately.

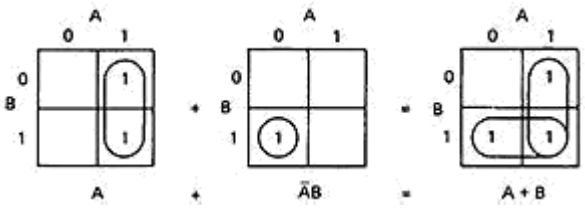
They are written using a plus sign.

It is clear now why there are sixteen functions of two variables.

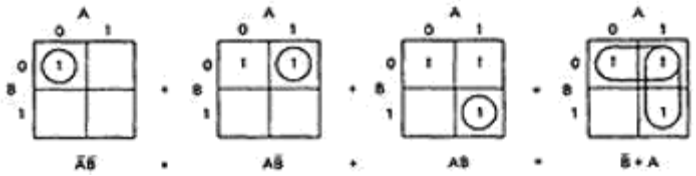
Each box in the KV chart corresponds to a combination of the variables' values. ○ That combination might or might not be in the function (i.e., the box corresponding to that combination might have a 1 or 0 entry).

- Since n variables lead to 2^n combinations of 0 and 1 for the variables, and each such combination (box) can be filled or not filled, leading to 2^{2n} ways of doing this.
- Consequently for one variable there are $2^{21} = 4$ functions, 16 functions of 2 variables, 256 functions of 3 variables, 16,384 functions of 4 variables, and so on.

Given two charts over the same variables, arranged the same way, their product is the term by term product, their sum is the term by term sum, and the negation of a chart is gotten by reversing all the 0 and 1 entries in the chart.



OR

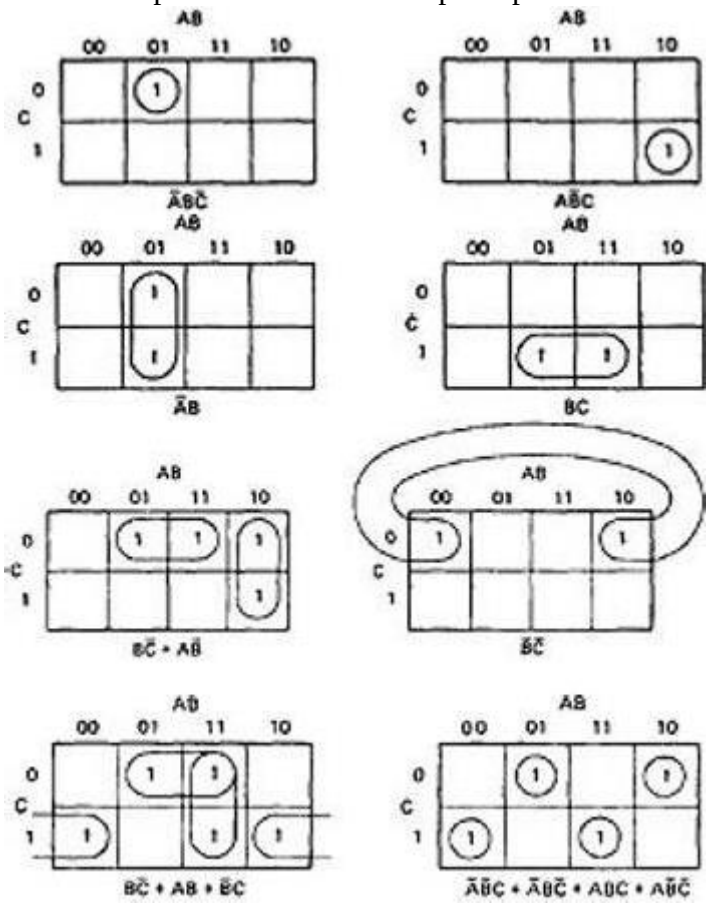


- **THREE VARIABLES:**

KV charts for three variables are shown below.

As before, each box represents an elementary term of three variables with a bar appearing or not appearing according to whether the row-column heading for that box is 0 or 1.

A three-variable chart can have groupings of 1, 2, 4, and 8 boxes. o A few examples will illustrate the principles:



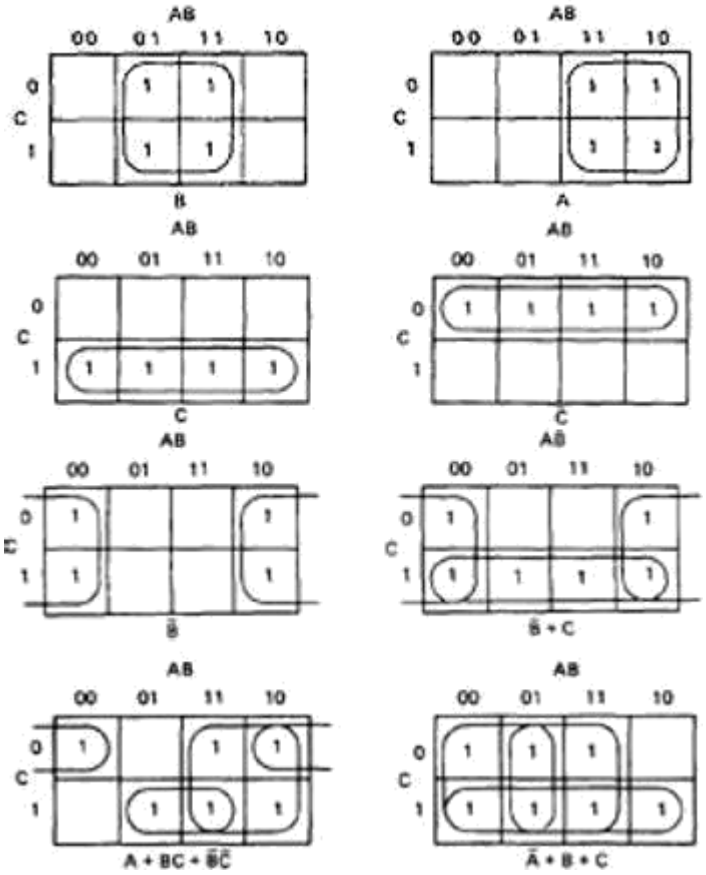


Figure 6.8: KV Charts for Functions of Three Variables.

You'll notice that there are several ways to circle the boxes into maximum-sized covering groups.