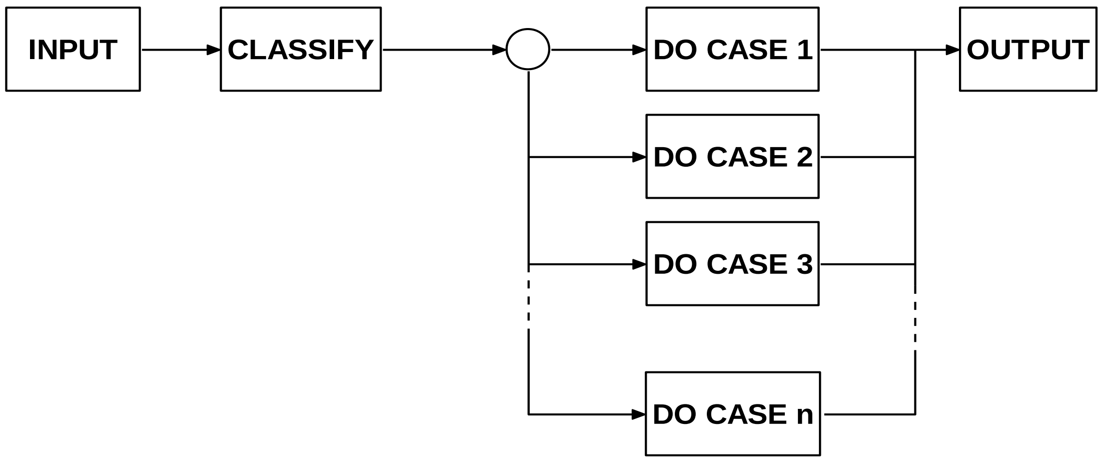
UNIT –III
DOMAIN TESTING

(1) Domains and paths:

(i) The Model:

- Domain testing can be based on specifications and/or equivalent implementation information.
- If domain testing is based on specifications, it is a functional test technique; if based on implementations, it is a structural technique.
- Domain testing is applied to one input variable or to simple combinations of two variables, based on specifications.
- The schematic representation of Domain testing is given below.



- First the different input variables are provided to a program.
- The classifier receives all input variables and divides them into different cases.
- Every case there should be at least one path to process that specified case.
- Finally output is received from this do cases..

(ii) A domain is a set:

- An input domain is a set. If the source language supports set definitions less testing is needed because the compiler (compile-time and run-time) do much of it for us.

(iii) Domains, paths and predicates:

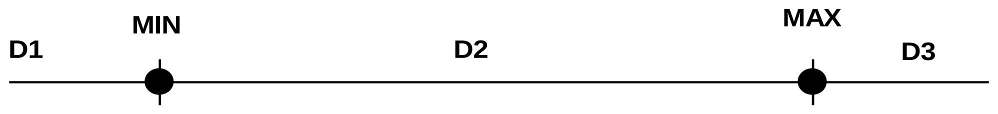
- In domain testing, predicates are assumed to be interpreted in terms of input vector variables.
- If domain testing is applied to structure (implementation), then predicate interpretation must be based on control flowgraph.
- If domain testing is applied to specifications, then predicate interpretation is based on data flowgraph.
- For every domain there is at least one path through the routine.
- There may be more than one path if the domain consists of disconnected parts.
- Unless stated otherwise, we'll assume that domains consist of a single, connected part.
- We'll also assume that the routine has no loops.
- Domains are defined by their boundaries. For every boundary there is at least one predicate.
- For example in the statement, IF X > 0 THEN ALPHA ELSE BETA we know that number greater than zero, belong to ALPHA, number smaller to zero, belong to BETA.

Review:

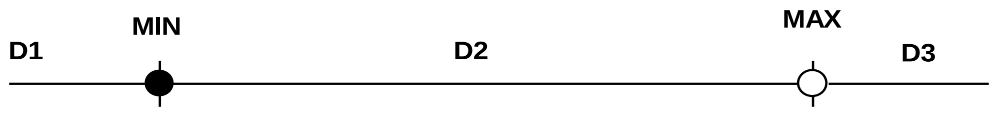
1. A domain is a loop free program.
2. For every domain there is at least one path through the routine.
3. The set of interpreted predicates defines the domain boundaries.

(iv) Domain Closure:

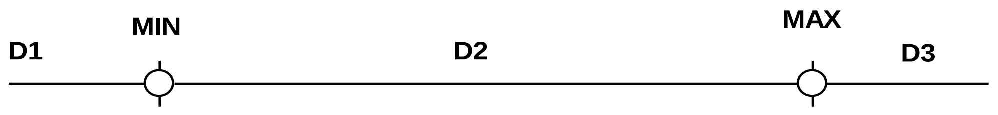
➢ To understand the domain closure, consider the following figure.



(a) Both side closed



(b) One side open



(c) Both side open

➢ If the domain boundary point belongs to the same domain then the boundary is said to close. If the domain boundary point belongs to some other domain then the boundary is said to open.

➢ In the above figure there are three domains D1, D2, D3.

➢ In **figure a** D2's boundaries are closed both at the minimum and maximum values. If D2 is closed, then the adjacent domains D1 and D3 must be open.

➢ In **figure b** D2 is closed on the minimum side and open on the maximum side, meaning that D1 is open and D3 is closed. In **figure c** D2 is open on both sides, which mean that the adjacent domains D1 and D3 must be closed.

(v) Domain Dimensionality:

➢ Depending on the input variables, the domains can be classified as number line domains, planer domains or solid domains.

➢ That is for one input variable the value of the domain is on the number line, for two variables the resultant is planer and for three variables the domain is solid.

➢ One important thing here is to note that we need not worry about the domains dimensionality with the number of predicates. Because there might be one or more boundary predicates.

(vi) The Bug Assumptions:

➢ The bug assumption for domain testing is that processing is okay but the domain definition is wrong.

➢ An incorrectly implemented domain means that boundaries are wrong, which mean that control-flow predicates are wrong.

➢ The following are some of the bugs that give to domain errors.

(a) Double-Zero Representation:

❖ Boundary errors for negative zero occur frequently in computers or programming languages where positive and negative zeros are treated differently.

(b) Floating-Point Zero Check:

- A floating-point number can equal to zero only if the previous definition of that number is set it to zero or if it is subtracted from itself, multiplied by zero.
- Floating-point zero checks should always be done about a small interval.

(c) Contradictory Domains:

- Here at least two assumed distinct domains overlap.

(d) Ambiguous Domains:

- These are missing domain, incomplete domain.

(e) Over specified Domains:

- The domain can be overloaded with so many conditions.

(f) Boundary Errors:

- This error occurs when the boundary is shifted or when the boundary is tilted or missed.

(g) Closure Reversal

- This bug occurs when we have selected the wrong predicate such as x>=0 is written as x<=0.

(h) Faulty Logic:

- This bug occurs when there are incorrect manipulations, calculations or simplifications in a domain.

(vii) Restrictions:

(a) General

- Domain testing has restrictions. i.e. we cannot use domain testing if they are violated.
- In testing there is no invalid test, only unproductive test.

(b) Coincidental Correctness

- Coincidental correctness is assumed not to occur.
- Domain testing is not good for which outcome is correct for the wrong reason.
- One important point to be noted here is that, domain testing does not support Boolean outcomes (TRUE/FALSE).
- If suppose the outputs are some discrete values, then there are some chances of coincidental correctness.

(c) Representative Outcome

- Domain testing is an example of partition testing.
- Partition testing divide the program's input space into domains.
- If the selected input is shown to be correct by a test, then processing is correct, and inputs within that domain are expected to be correct.
- Most test techniques, functional or structural fall under partition testing and therefore make this representative outcome assumption.

(d) Simple Domain Boundaries and Compound Predicates

- Each boundary is defined by a simple predicate rather than by a compound predicate.
- Compound predicates in which each part of the predicate specifies a different boundary are not a problem: for example, x >= 0 .AND. x < 17, just specifies two domain boundaries by one compound predicate.

(e) Functional Homogeneity of Bugs

- Whatever the bug is, it will not change the functional form of the boundary predicate.

(f) Linear Vector Space

- A linear predicate is defined by a linear inequality using only the simple relational operators >, >=, =, <=, <>, and <.
- Example $x^2 + y^2 > a^2$.

(g) Loop-free Software

- Loops (indefinite loops) are problematic for domain testing.

- ❖ If a loop is an overall control loop on transactions, say, there's no problem.
- ❖ If the loop is definite, then domain testing may be useful for the processing within the loop, and loop testing can be applied to the looping values.

(2) Nice Domains:

(i) Where Do Domains Come From?

- ➢ Domains are often created by salesmen or politicians.
- ➢ The first step in applying domain testing is to get consistent and complete domain specifications.

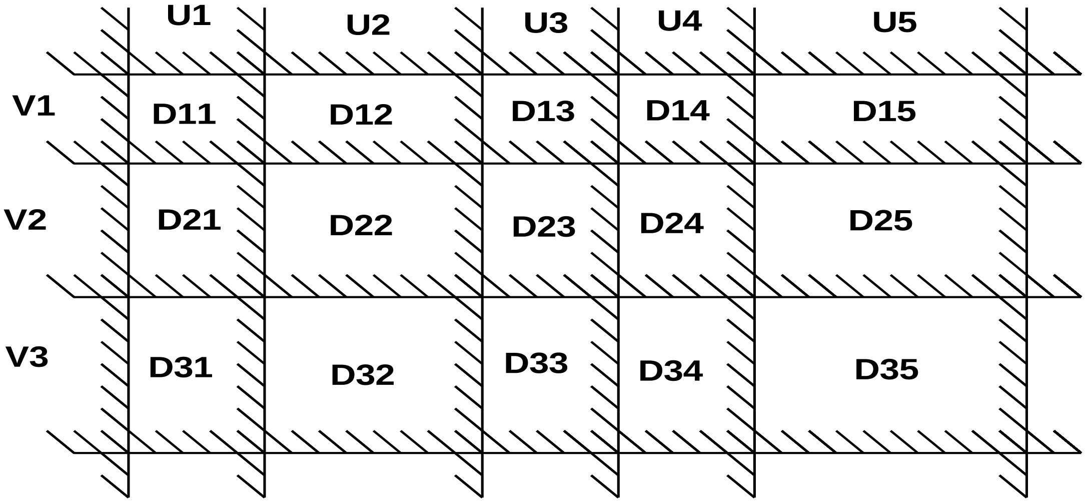
(ii) Specified versus Implemented Domains:

- ➢ Implemented domains can't be incomplete or inconsistent but specified domains can be incomplete or inconsistent.
- ➢ Incomplete means that there are input vectors for which no path is specified and inconsistent means that there are at least two contradictory specifications.

(iii) Nice Domains:

(1) General

- ❖ The representation of Nice two-dimensional domains is as follows. .



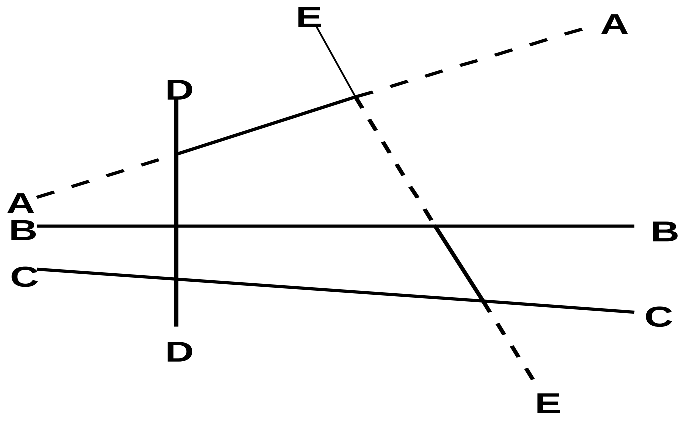
- ❖ The U and V represent boundary sets and D represents domains.
- ❖ The boundaries have several important properties. They are linear, complete, systematic, orthogonal, consistently closed, simply connected and convex.
- ❖ If domains have these properties, domain testing is very easy otherwise domain testing is tough.

(2) Linear and Nonlinear Boundaries

- ❖ Nice domain boundaries are defined by linear inequalities or equations.
- ❖ The effect on testing comes from only two points then it represents a straight line.
- ❖ If it considers three points then it represents a plane and in general it considers n + 1 points then it represents an n-dimensional hyperplane.
- ❖ Linear boundaries are more frequently used than the non-linear boundaries.
- ❖ We can linearize the non-linear boundaries by using simple transformations.

(3) Complete Boundaries

- ❖ Complete boundaries are those boundaries which do not have any gap between them.
- ❖ Nice domain boundaries are complete boundaries because they cover from plus infinity to minus infinity in all dimensions.
- ❖ Incomplete boundaries are those boundaries which consist of some gaps between them and are not covered in all dimensions.
- ❖ The following figure represents some incomplete boundaries.



- The Boundaries A and E have gaps so they are incomplete & the boundaries B, C, D are complete.
- The main advantage of a complete boundary is that it requires only one set of tests to verify the boundary

(4) Systematic Boundaries

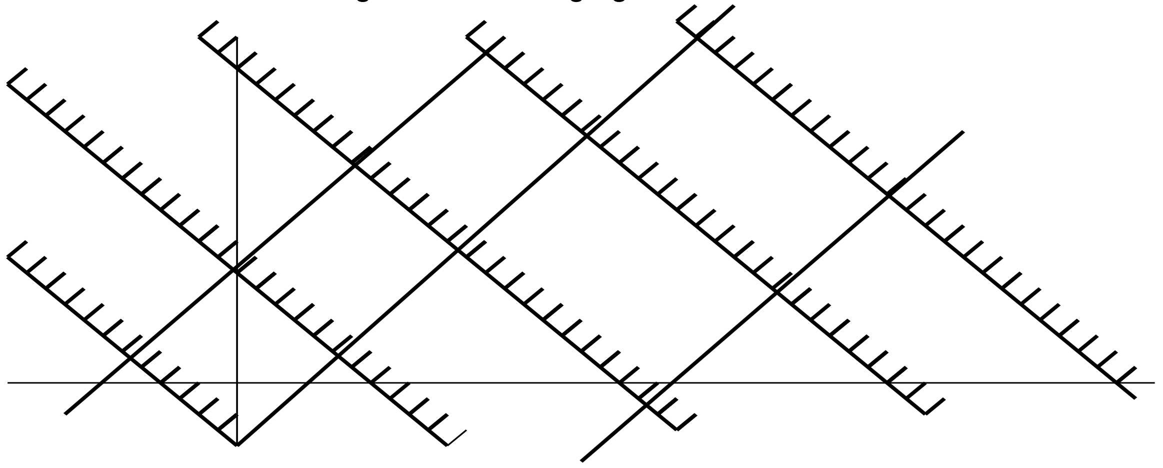
- Systematic boundaries refer to boundary inequalities with simple mathematical functions such as a constant.
- Consider the following relations,

$f_1(X) >= k_1$ or $f_1(X) >= g(1,c)$
$f_2(X) >= k_2$ $f_2(X) >= g(2,c)$
....
$f_i(X) >= k_i$ $f_i(X) >= g(i,c)$

- Where f_i is an arbitrary linear function, X is the input vector, k_i and c are constants, and $g(i,c)$ is a decent function that yields a constant, such as $k + ic$.

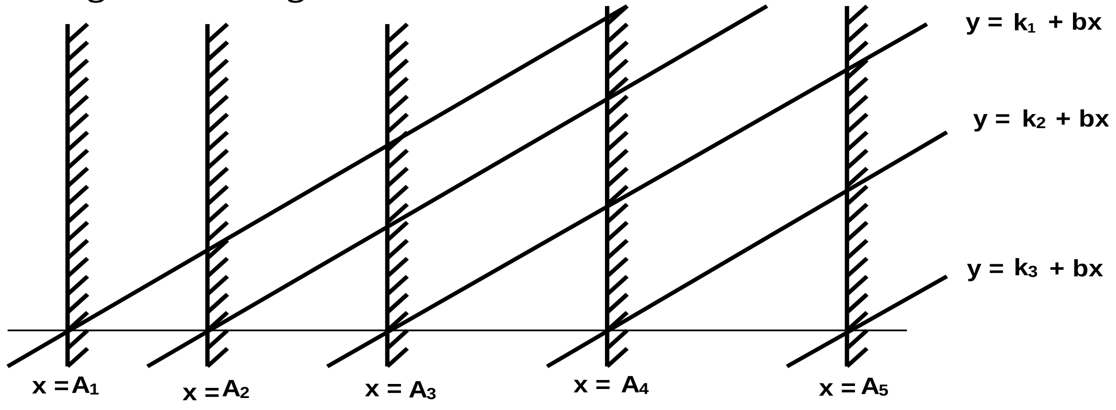
(5) Orthogonal Boundaries

- The *U* and *V* boundary sets in Nice two-dimensional domains figure are orthogonal; that is, the every boundary *V* is perpendicular to every other boundary *U*.
- If two boundary sets are orthogonal, then they can be tested independently.
- If we want to tilt the above orthogonal boundary we can do it by testing its intersection points but this can change the linear growth, O(n) into the quadratic growth O(n^2).
- If we tilt the boundaries to get the following figure then we must test the intersections.



(6) Closure Consistency

- Consistent closures are the most simple and fundamental closure.
- It gives consistent and systematic results.
- The following figure shows the boundary closures are consistent.



- In the above figure, the shading lines show one boundary and thick lines show other boundary.
- It shows Non orthogonal domain boundaries, which mean that every inequality in domain x is not perpendicular to every inequality in domain y.

(7) Convex

- A figure is said to be convex when for any two boundaries, with two points placed on them are combined by using a single line then all the points on that line are within the range of the same figure.
- Nice domains support convex property, where as dirty domains don't.

(8) Simply Connected

- Nice domains are usually simply connected because they are available at one place as a whole but not dispersed in other domains..
- Simple connectivity is a weaker requirement than convexity; if a domain is convex it is simply connected, but not vice versa.

(iv) Ugly Domains:

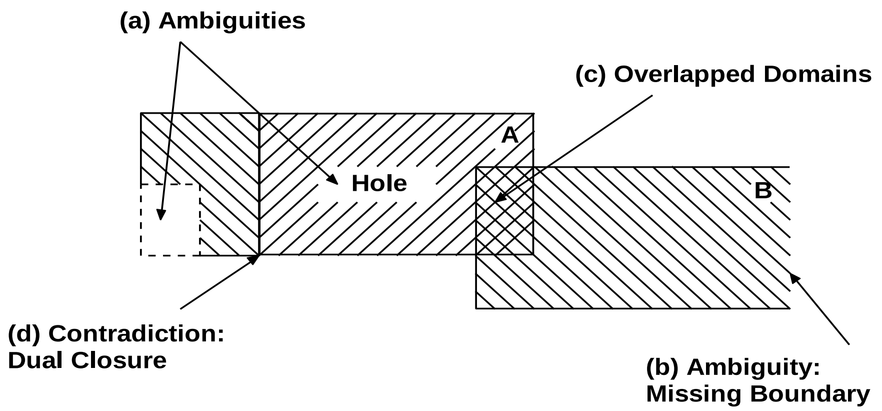
(a) General

- Some domains are born ugly. Some domains are bad specifications.
- So every simplification of ugly domains by programmers can be either good or bad.
- If the ugliness results from bad specifications and the programmer's simplification is harmless, then the programmer has made ugly good.
- But if the domain's complexity is essential such simplifications gives bugs.

(b) Nonlinear Boundaries

- Non linear boundaries are rare in ordinary programming, because there is no information on how programmers correct such boundaries.
- So if a domain boundary is non linear, then programmers make it linear.

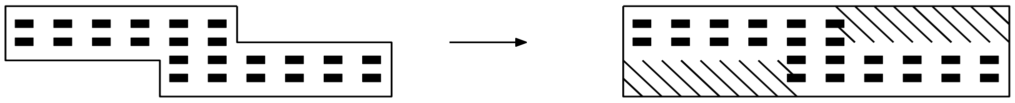
(c) Ambiguities and Contradictions:.

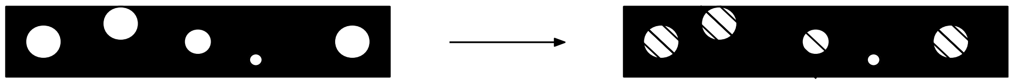

(a) Ambiguities
(c) Overlapped Domains
A
Hole
B
(d) Contradiction: Dual Closure
(b) Ambiguity: Missing Boundary

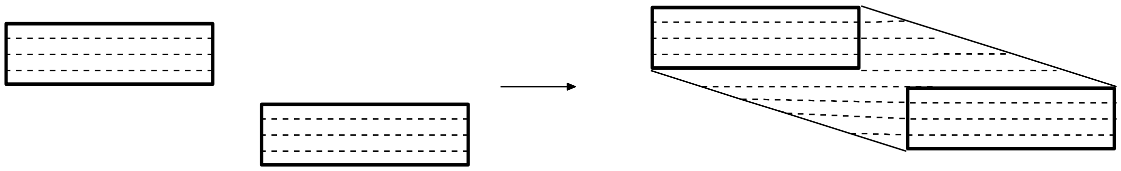
- ❖ Domain ambiguity is missing or incomplete domain boundary.
- ❖ In the above figure Domain ambiguities are holes in the A domain and missing boundary in the B domain.
- ❖ An ambiguity for one variable can be see easy.
- ❖ An ambiguity for two variables can be difficult to spot.
- ❖ An ambiguity for three or more variables impossible to spot. Hence tools are required.
- ❖ Overlapping domains and overlapping domain closure is called contradiction.
- ❖ There are two types of contradictions are possible here.
  (1) Overlapped domain specifications
  (2) Overlapped closure specifications.
- ❖ In the above figure there is overlapped domain and there is dual closure contradiction. This is actually a special kind of overlap.

(d) Simplifying the Topology

- ❖ Connecting disconnected boundary segments and extending boundaries is called simplifying the topology
- ❖ There are three generic cases of simplifying the topology.


(a) Making it convex

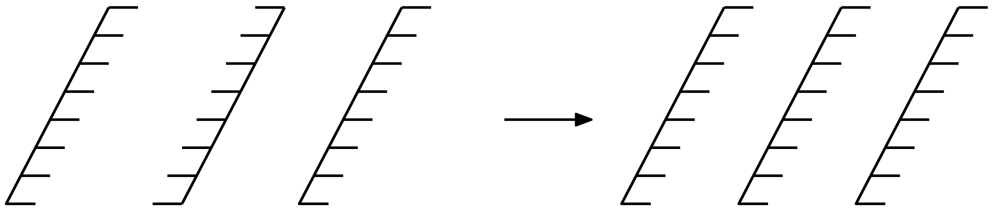

(b) Filling in the Holes

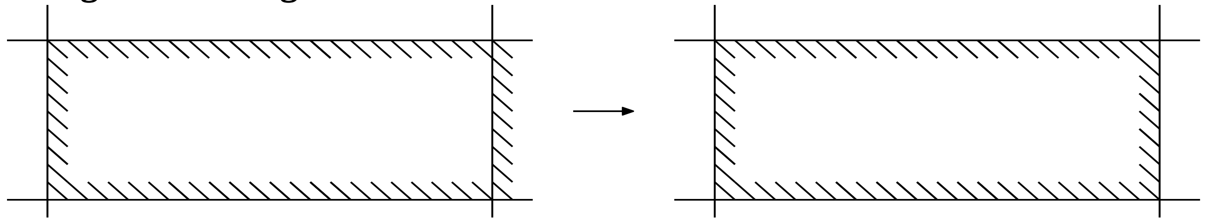

(c) Joining the Pieces

- ❖ Programmers introduce bugs and testers misdesign test cases by, smoothing out concavities, filling in holes, joining disconnected pieces.

(e) Rectifying Boundary Closures

- ❖ Different boundaries in different directions can obtain in consistent direction is called rectifying boundary closures.
- ❖ That is domain boundaries which are different directions can obtain in one direction.


(a) Consistent Direction



(b) Inclusion/Exclusion Consistency

- ❖ In the above figure the hyper plane boundary is outside that can obtain inside. This is called inclusion / exclusion consistency.

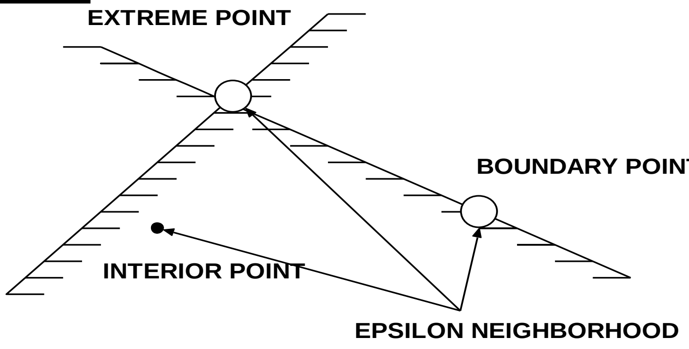
(3) Domain Testing:

(i) Overview:

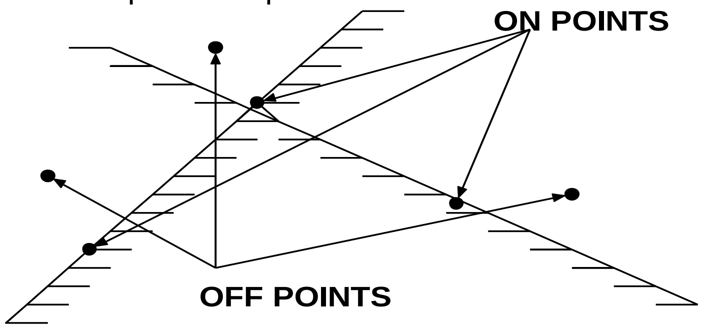
- ➢ Domains are defined by their boundaries. So domain testing concentrates test points on boundaries or near boundaries.
- ➢ Find what wrong with boundaries, and then define a test strategy.
- ➢ Because every boundary uses at least two different domains, test points used to check one domain can also be used to check adjacent domains.
- ➢ Run the tests, and determine if any boundaries are faulty.
- ➢ Run enough tests to verify every boundary of every domain.

(ii) Domain Bugs and How to Test for Them:

(a) General:

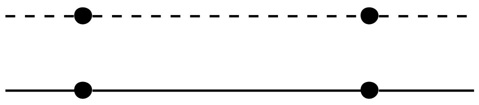


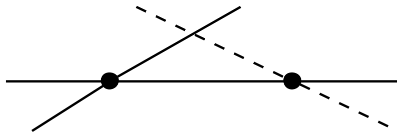
- ❖ An interior point is a point in a domain. It can be defined as a point which specifies certain distance covered by some other points in the same domain.
- ❖ This distance is known as epsilon neighborhood.
- ❖ A boundary point is on the boundary that is a point with in a specific epsilon neighborhood.
- ❖ An extreme point is a point that does not lie between any other two points.

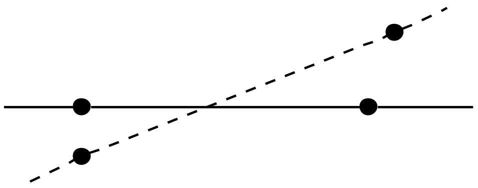


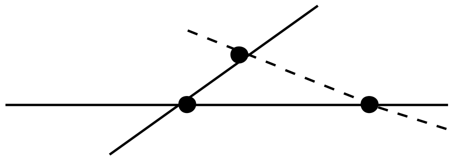
- ❖ An on point is a point on the boundary. An off point is outside the boundary.
- ❖ If the domain boundary is closed, an off point is a point near the boundary but in the adjacent domain.

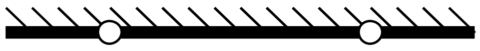
- ❖ If the domain boundary is open, an off point is a point near the boundary but in the same domain.
- ❖ Here we have to remember CLOSED OFF OUTSIDE, OPEN OFF INSIDE
- ❖ i.e. COOOOI
- ❖ The following figure shows a generic domain ways.


SHIFTED BOUNDARIES


EXTRA BOUNDARY


TILTED BOUNDARIES

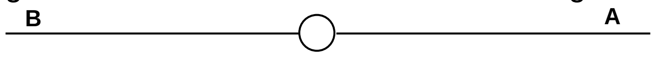

MISSING BOUNDARY


OPEN / CLOSE ERROR

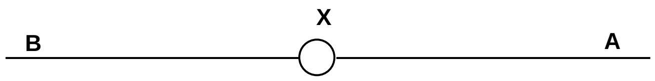
CORRECT ————
INCORRECT - - - - - - -

(b) Testing One-Dimensional Domains:

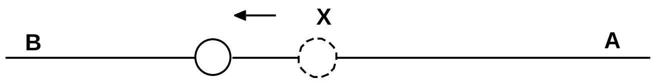
- ❖ The following figure shows one dimensional domain bugs for open boundaries.



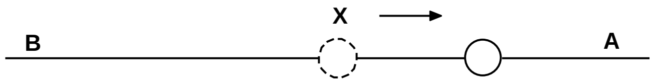
a) An Open Domain (A)



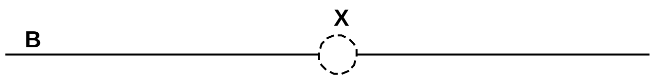
b) Closure bug



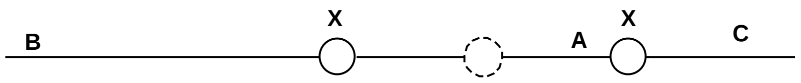
c) Boundary shifted left



c) Boundary shifted right

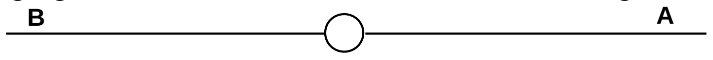


e) Missing Boundary

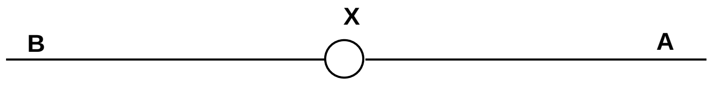


f) Extra Boundary

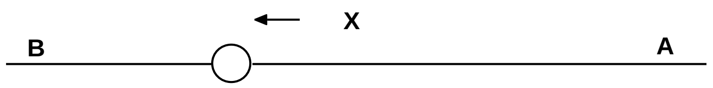
- ❖ In the above figure a) we assume that the boundary was to open for A.
- ❖ In figure b) one test point (marked X) on the boundary detects the bug.
- ❖ In figure c) a boundary shifts to left.
- ❖ In figure d) a boundary shifts to right.
- ❖ In figure e) there is a missing boundary. In figure f) there is an extra boundary.
- ❖ The following figure shows one dimensional domain bugs for closed boundaries.



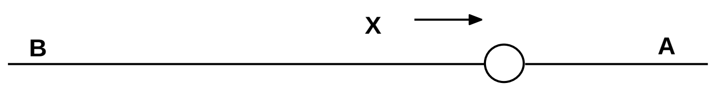
a) A closed Domain (A)



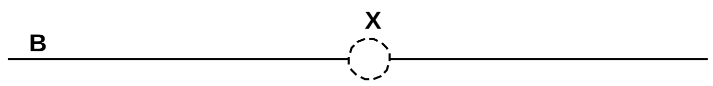
b) Closure bug



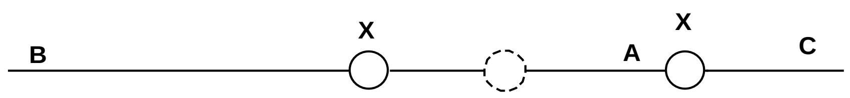
c) Boundary shifted left



c) Boundary shifted right



e) Missing Boundary



f) Extra Boundary

- ❖ In the above figure a) we assume that the boundary was to close for A.
- ❖ In figure b) one test point (marked X) on the boundary detects the bug.
- ❖ In figure c) a boundary shifts to left. In figure d) a boundary shifts to right.
- ❖ In figure e) there is a missing boundary. In figure f) there is an extra boundary.
- ❖ Only one difference from this diagram to previous diagram is here we have closed boundaries.

<u>(c) Testing Two-Dimensional Domains:</u>

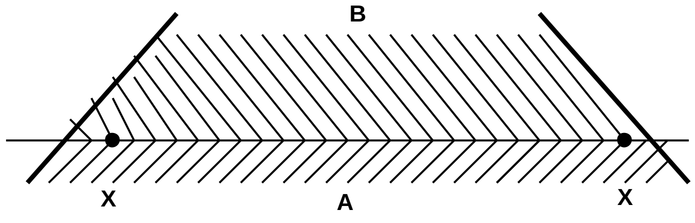
- ➢ The following figure shows domain boundary bugs for two dimensional domains.
- ➢ A and B are adjacent domains, and the boundary is closed with respect to A and the boundary is opened with respect to B.

<u>(i) Closure Bug:</u>

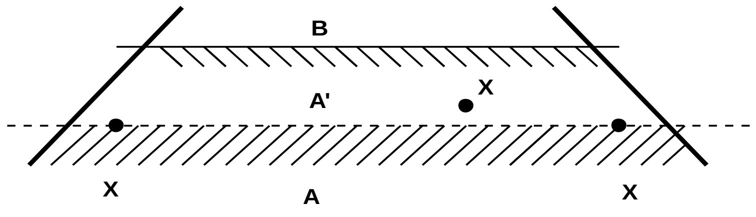
- ❖ The figure (a) shows a wrong closure, that is caused by using a wrong operator for example, x>=k was used when x > k was intended.
- ❖ The two on points detect this bug.

<u>(ii) Shifted Boundary:</u>

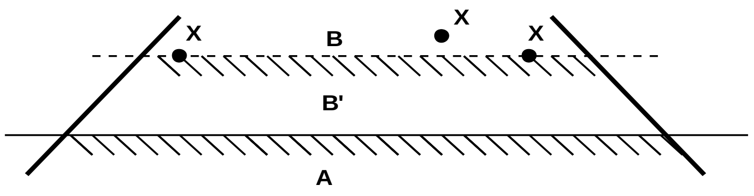
- ❖ In figure (b) the bug is shifted up, which converts part of domain B into A'.
- ❖ This is caused by incorrect constant in a predicate for example x + y >= 17 was used when x + y > = 7 was intended. Similarly figure (c) shows a shift down.



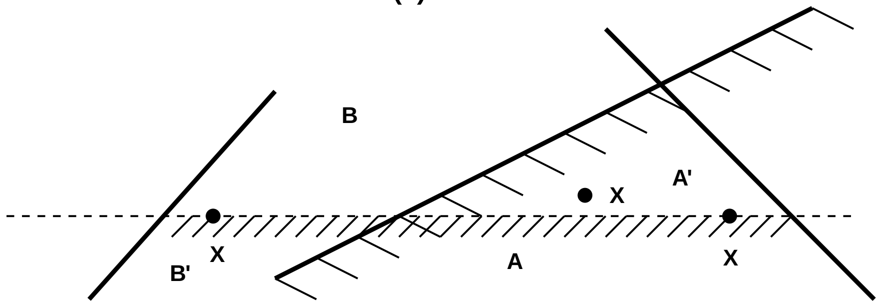
(a) Closure Bug



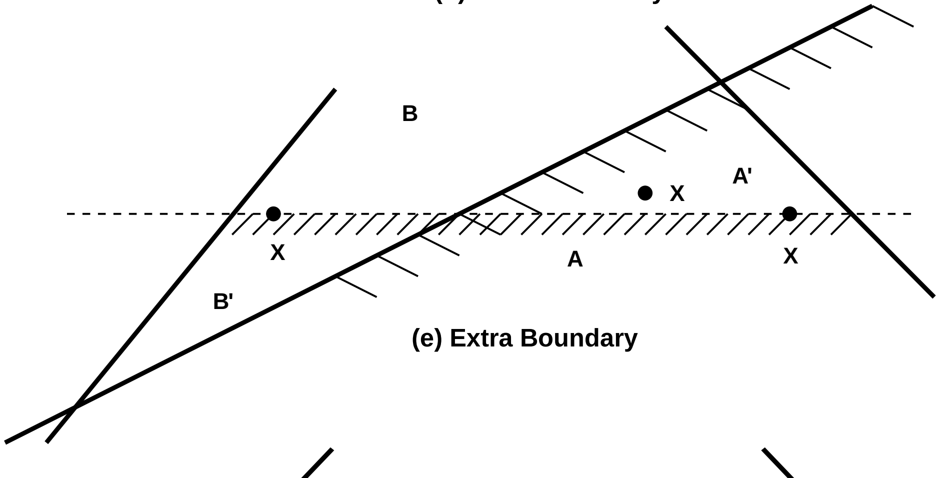
(b) Shifted Up



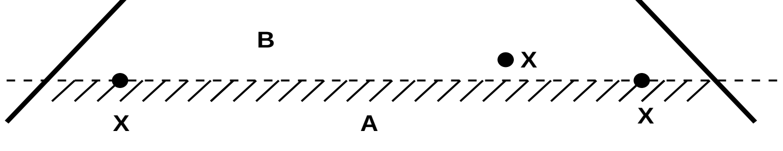
(c) Shifted Down



(d) Tilted Boundary



(e) Extra Boundary



(f) Missing Boundary

(iii) Tilted boundary:

- ❖ A tilted boundary occurs, when coefficients in the boundary inequality are wrong.
- ❖ For example we used 3x + 7y > 17 when 7x + 3y > 17 is needed.
- ❖ Figure (d) shows a tilted boundary which creates domain segments A' and B'.

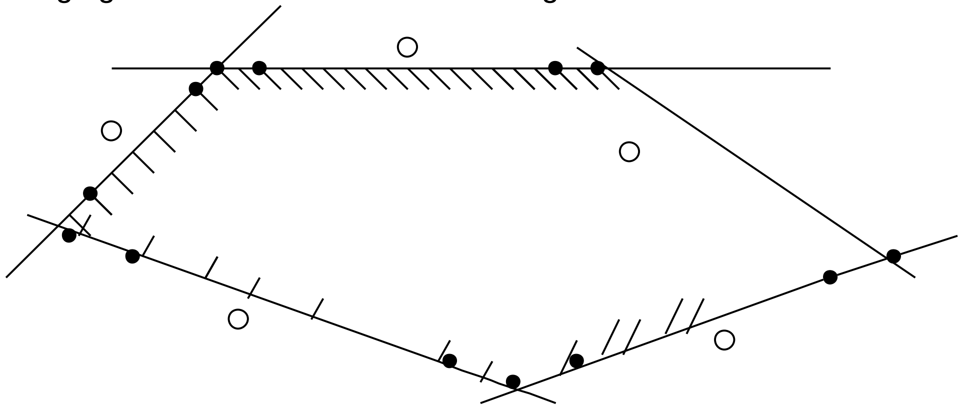
(iv) Extra Boundary:

- ❖ An extra boundary is created by an extra predicate.
- ❖ Figure (e) shows an extra boundary. The extra boundary is caught by two on points.

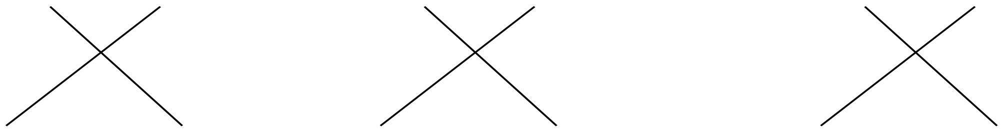
(v) Missing Boundary:

- ❖ A missing boundary is created by leaving out the predicate.
- ❖ A missing boundary shown in figure (f) is caught by two on points.

- ➢ The following figure summarizes domain testing for two dimensional domains.

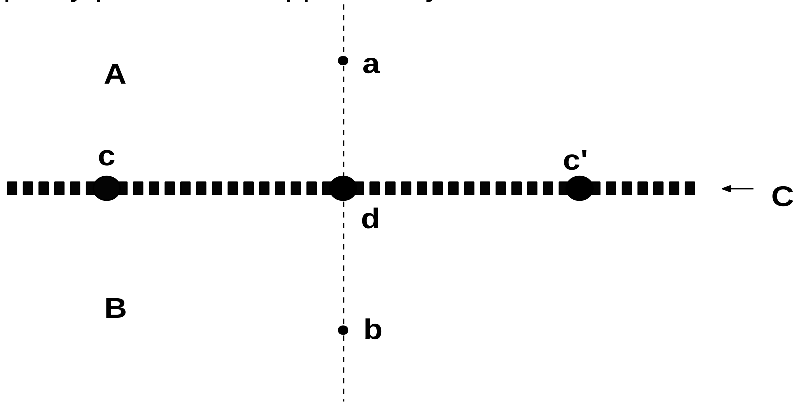


- ➢ There are two on points (closed circles) for each segment and one off point (open circle)
- ➢ Note that the selected test points are shared with adjacent domains.
- ➢ The on points for two adjacent boundary segments can also be shared.
- ➢ The shared on points is given below.



(d) Equality and Inequality Predicates:

- ❖ Equality predicates are defined by equality equation such as x + y =12.
- ❖ Equality predicates supports only few domain boundaries



- ❖ Inequality predicates are defined by inequality equation such as x + y > 12 or x + y <12
- ❖ Inequality predicates supports most of the domain boundaries.
- ❖ In domain testing, equality predicate of one dimension is a line.
- ❖ Similarly equality of two dimensions is a two dimensional domain and equality of three dimensions is a planer domain.

- Inequality predicates test points are obtained by taking adjacent domains into consideration.
- In the above figure the three domains A, B, C are planer. The domain C is a line.
- Here domain testing is done by two on points & two off points.
- That is test point b for B, and test point a for A and test points c and c' for C.

(e) Random Testing:

- Random testing is a form of functional testing that is useful when the time needed to write and run directed tests are too long.
- One of the big issues of random testing is to know when a test fails.
- When doing random testing we must ensure that they cover the specification.
- The random testing is less efficient than direct testing. But we need random test generators.

(f) Testing n-Dimensional Domains:

- If domains defined over n-dimensional input space with p-boundary segments then the domain testing gives testing n-dimensional domains.

(iii) Procedure:

- Generally domain testing can be done by hand for two dimensions.
- Without tools the strategy is practically impossible for more than two variables.

1. Identify the input variables.
2. Identify variables which appear in domain predicates.
3. Interpret all domain predicates in terms of input variables.
4. For p binary predicates there are 2^p domains.
5. Solve the inequalities to find all the extreme points of each domain.
6. Use the extreme points to solve for near by on points.

(iv) Variations, Tools, Effectiveness:

- Variations can vary the number of on and off points or the extreme points.
- The basic domain testing strategy discussed here is called the N X 1 strategy, because it uses N on points and one off point.
- In cost effectiveness of domain testing they use partition analysis, which includes domain testing, computation verification and both structural and functional information.
- Some specification tools are used in domain testing.

(4) Domains and Interface Testing:

(i) General:

- The domain testing plays a very important role in integration testing. In integration testing we can find the interfaces of different components.
- We can determine whether the components are accurate or not.

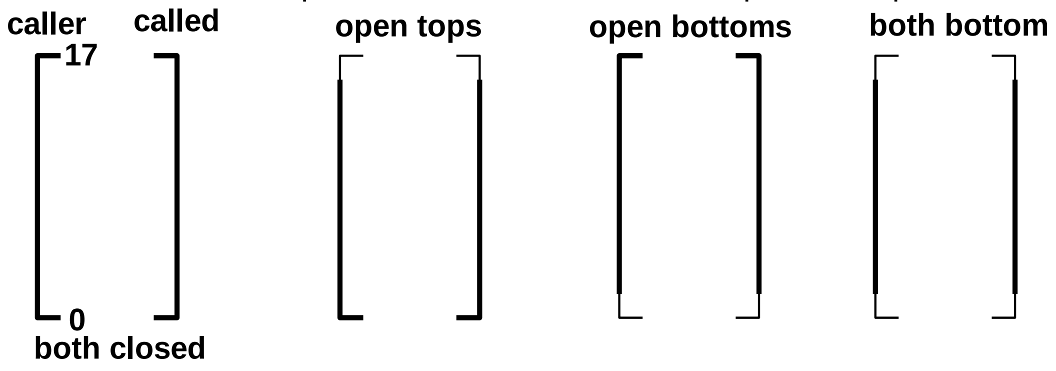
(ii) Domains and Range:

- Domains are the input values used. Range is just opposite of domains.
- i.e. Range is output obtained.
- In most testing techniques, more forces on the input values.
- This is because with the help of input values it will be easy to identify the output.
- But interface testing gives more forces on the output values.
- An interface test consists of exploring the correctness of the following mappings.

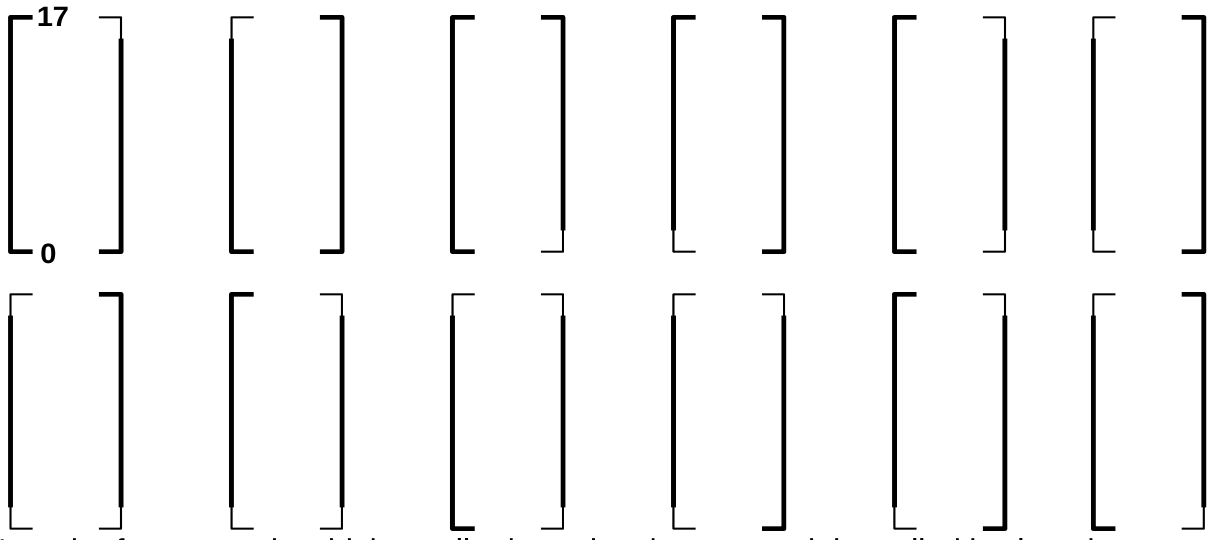
$$\begin{tikzcd}
\text{Caller domain} \arrow[r] & \text{Caller range} \\
\text{Caller range} \arrow[r] & \text{Called domain} \\
\text{Called domain} \arrow[r] & \text{Called range}
\end{tikzcd}$$

(iii) Closure Compatibility:

- Assume that the caller's range and the called domain spans the same numbers say 0 to 17
- The closure compatibility shows the four cases in which the caller's range closure and the called's domain closure can agree.
- The four cases consists of domains that are closed on top (17) & bottom (0), open top & closed bottom, closed top & open bottom and open top & bottom.
- Here the thick line represents closed and thin line represents open.



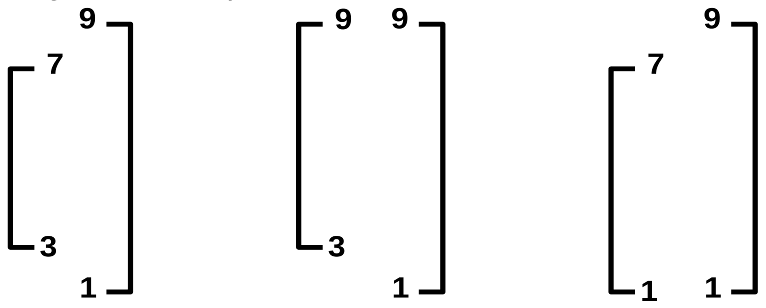
- The following figure shows the twelve different ways the caller and the called can disagree about closure. Not all of them are necessarily bugs.



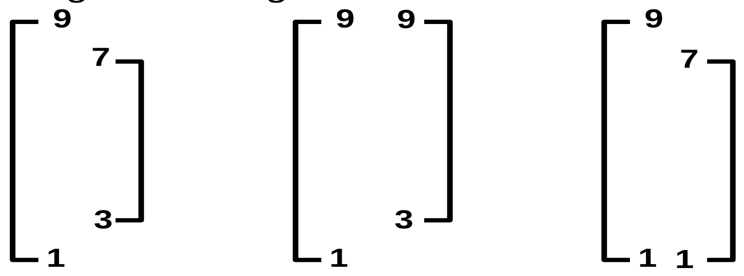
- Here the four cases in which a caller boundary is open and the called is closed are not buggy.

(iv) Span Compatibility:

- The following figure shows three possibly harmless of span incompatibilities.
- In this figure Caller span is smaller than Called.



- The range of a caller is a sub set of the called domain. That is not necessarily a bug.
- The following figure shows Called is Smaller than Caller.



(v) Interface Range/ Domain Compatibility Testing:

- The application of domain testing is also very important for interface testing because it tests the range and domain compatibilities among caller and called routines.
- It is the responsibility of the caller to provide the valid inputs to the called routine.
- After getting the valid input, the test will be done on every input variable.

(vi) Finding the values:

- Start with the called routine's domains and generate test points.
- A good component test should have included all the interesting domain-testing cases.
- Those test cases are the values for which we must find the input values of the caller.

(5) Domains and Testability:

(i) General:

- Domain testing gives orthogonal domain boundaries, consistent closure, independent boundaries, linear boundaries, and other characteristics. We know that which makes domain testing difficult. That is it consists of applying algebra to the problem.

(ii) Linearizing Transformations:

- This is used to transfer non linear boundaries to equivalent linear boundaries.
- The different methods used here are

(i)Polynomials:

- A boundary is specified by a polynomial or multinomial in several variables.
- For a polynomial each term can be replaced by a new variable.
- i.e. x, x^2, x^3, …can be replaced by $y_1 = x$, $y_2 = x^2$, $y_3 = x^3$, …
- For multinomials you add more new variables for terms such as xy, x^2y, xy^2, …
- So polynomial plays an important role in linear transformations.

(ii)Logarithmic Transforms:

- Products such as xyz can be linearized by substituting $u = \log (x)$, $v = \log (y)$, $\log (z)$.
- The original predicate $xyz > 17$ now becomes $u + v + w > 2.83$.

(iii)More general forms:

- Apart from logarithmic transform & polynomials there are general linearizable forms such as $x / (a + b)$ and ax^b. We can also linearize by using Taylor series.

(iii) Coordinate Transformations:

- The main purpose of coordinate transformation technique is to convert Parallel boundary inequalities into non parallel boundary inequalities and Non-parallel boundary inequalities into orthogonal boundary inequalities.

(iv) A Canonical Program Form:

- Testing is clearly divided into testing the predicate and coordinate transformations.
- i.e. testing the individual case selections, testing the control flow and then testing the case processing..

(v) Great Insights:

- Sometimes programmers have great insights into programming problems that result in much simpler programs than one might have expected.

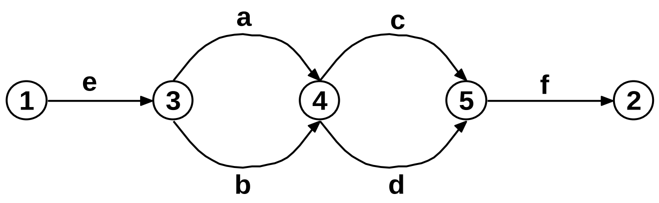
PATHS, PATH PRODUCTS AND REGULAR EXPRESSIONS

(1) Path products & path expression:

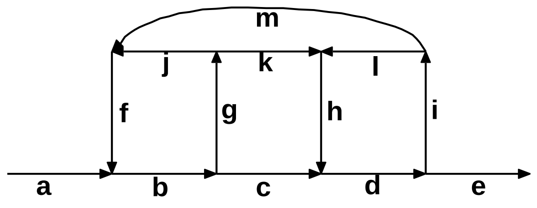
(1) Explain Paths, Path products, Path expressions, path sums and loops?

(a) Paths:

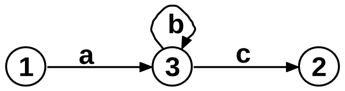
- A sequence of statements which starts at an entry and ends at an exit and passes all the decisions, junctions & processes is known as path.
- A path represents different links and we can give a simplest weight to a link is a name.
- Using link names, we can convert the graphical flowgraph into an equivalent algebraic expression.
- The link name will be denoted by lower case italic letters.
- In traversing a path, we traverse link names that give the name of the path.
- If you traverse links a, b, c, d then the name for that path is abcd.
- This path name is also called a path product. The following are some examples of paths.



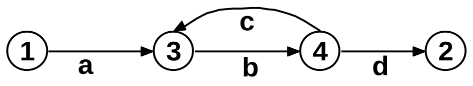
The different paths are: eacf, eadf, ebcf, ebdf



The different paths are: abcde, abgjfbcde, abcdimfbcde



The different paths are: ac, abc, abbc, abbbc, abbbbc



The different paths are: abd, abcbd, abcbcbd, abcbcbcbd

(b) Path Products:

- The concatenation of names of two consecutive path segments is called a path product.
- For example if X and Y are defined as X = abcde and Y = fghij then

  XY = abcdefghij YX = fghijabcde

  aX = aabcde Xa = abcdea XaX = abcdeaabcde .
- Another example is if X = abc + def + ghi and Y = uvw + z then

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

  If X = abcde then X^1 = abcde

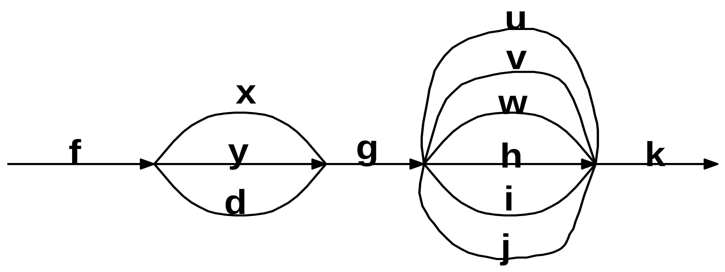
  $X^2 = (abcde)^2$ = abcdeabcde

  …
- The path product is not commutative that is XY does not necessarily equal to YX.

- The path product is associative that is (XY)Z = X(YZ).

(c) Path expression:

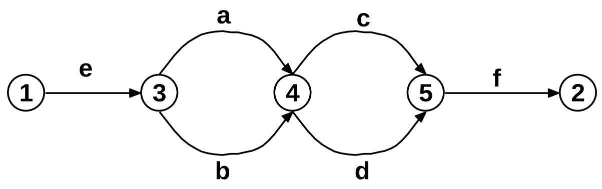
- Path expression is defined as an expression which represents set of all possible paths between an entry and exit nodes. For example:



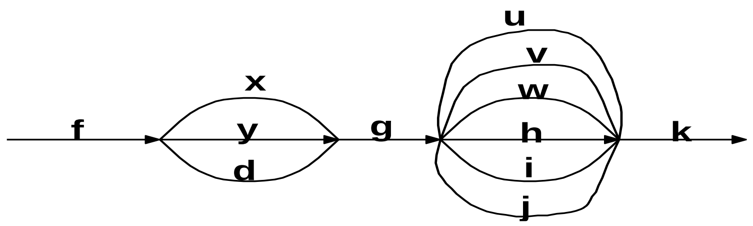
- The path expression to the above figure is: f (x + y + d) g (u + v + w + h + i + j) k

(d) Path sums:

- The path sum is the sum of all the parallel links between two nodes or sum of all parallel paths between two nodes. Path sum is denoted by '+'.
- Ex (i)



- In the above figure, links a & b are parallel, so these parallel paths are denoted by a + b.
- Similarly c and d are parallel & these parallel paths are denoted by c + d.
- The set of parallel paths between 1 and 2 nodes are eacf + eadf + ebcf + ebdf.
- Ex (ii)

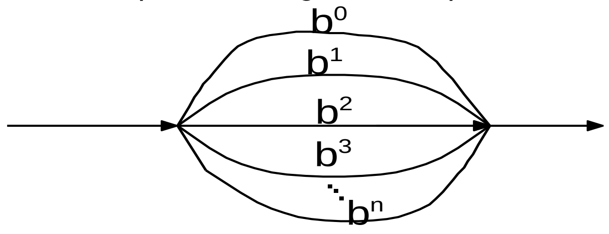


- The first set of parallel path is denoted by X + Y + d and second by u + v + w + h + i + j.
- The set of all paths in this flowgraph is f (X + Y + d) g(u + v + w + h + i + j) k
- Path sum is commutative and associative. Commutative is X + Y = Y + X

  Associative is (X+Y)+Z=X+(Y+Z)

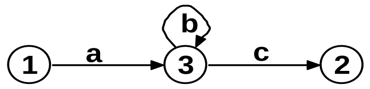
(e) Loops:

- If a single link or path expression is traversed indefinite no of times leading to infinite no of parallel paths then it is called a loop. For example the loop consists of a single link b, then the set of all paths through that loop is $b^0 + b^1 + b^2 + \dots b^n$



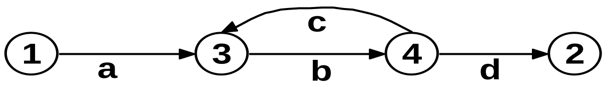
- This infinite sum is denoted by b*. So $b^* = b^0 + b^1 + b^2 + \dots b^n$.
- If the loop is taken at least once then it is denoted by b^+.

- Ex (i)



The path expression is: ab*c = $a(b^0)c + a(b^1)c + a(b^2)c + a(b^3)c+$…….
= ac + abc + a bbc + a bbbc + ……

Ex (ii)



The path expression is: a(bc)*bd = a(bc)bd + a(bc)bd + a(bc)bd + …….
=abd + abcbd + abcbcbd + …

(2) Discuss all the rules in path representation of graphs?

Rule 1:

A(BC)=(AB)C=ABC

Rule 2:

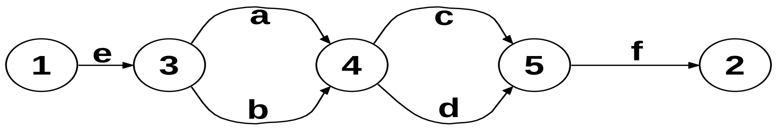
X + Y = Y + X

Rule 3:

(X + Y) + Z = X + (Y + Z) = X + Y + Z
A(BC)=(AB)C=ABC

Rule 4:

- Distributive laws are A(B+C) = AB + AC
(B + C) D = BD + CD.
- For example:



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

Rule 5:

- The absorption rule is, if X and Y denote the same set paths, then the union of these sets is not changed. Ex: X + X = X.
- Another example is: if X = a + aa + abc + abcd + def then
X + a = X + aa = X + abc = X + abcd = X + def = X

Rule 6:

$X^{\underline{n}} + X^{\underline{m}} = X^{\underline{n}}$ if n is bigger than m
$= X^{\underline{m}}$ if *m* is bigger than *n*

Rule 7:

$X^{\underline{n}}X^{\underline{m}} = X^{\underline{n+m}}$

Rule 8:

$X^{\underline{n}}X^* = X^*X^{\underline{n}} = X^*$

Rule 9:

$X^{\underline{n}}X^+ = X^+X^{\underline{n}} = X^+$

Rule 10:

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

Identity elements:(Rule 11 to Rule 17)

- a^0, X^0 denote the path whose length is zero. The rules are

Rule 11:

1 + 1 = 1

Rule 12:

$1X = X1 = X$

Rule 13:

$1^n = 1^{\underline{n}} = 1^* = 1^+ = 1$

Rule 14:

$1^+ + 1 = 1^* = 1$

Rule 15:

$X + 0 = 0 + X = X$

Rule 16:

$X0 = 0X = 0$

Rule 17:

$0^* = 1 + 0^1 + 0^2 + . . . = 1$

(2) A Reduction Procedure:

(1) Write the steps involved in Node Reduction Procedure. Illustrate all the steps with the help of neat labeled diagrams?

Node Reduction Procedure:

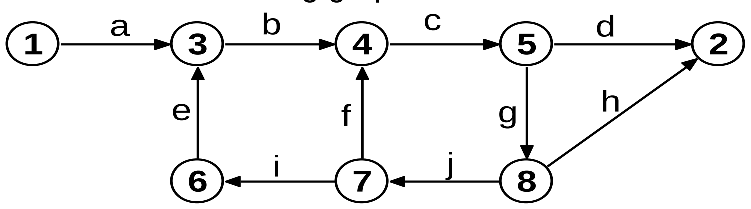
- The main aim of Node Reduction Procedure is to remove all the intermediate nodes between entry and exit nodes. This procedure is helpful in debugging process. i.e. Instead of gathering information about path expression of all the intermediate nodes for debugging; it is easy to debug only the path expression between entry and exit nodes.

Procedure:

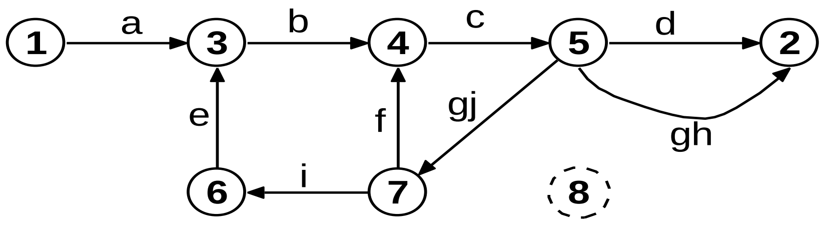
1. Combine all serial links by multiplying their path expressions.
2. Combine all parallel links by adding their path expressions.
3. Remove all self loops by replacing them with a link of the form x^*, where x is the path expression of the link in that loop.
4. Choose the node which is to be removed other than initial and final node. The path expression of the inlink and outlink of this node is multiplied and a direct link is applied with the product of path expression. This step-4 is called Cross-Term Step.
5. Combine any remaining serial links by multiplying their path expressions.
6. Combine all parallel links by adding their path expressions. This Step-6 is called Parallel Term Step.
7. Remove all self-loops as in step 3. This Step-7 is called Loop Term Step.
8. If the graph consists of a single link between the entry and the exit node, then the path expression for that link is a required path expression. Otherwise return to step 4.

Example:

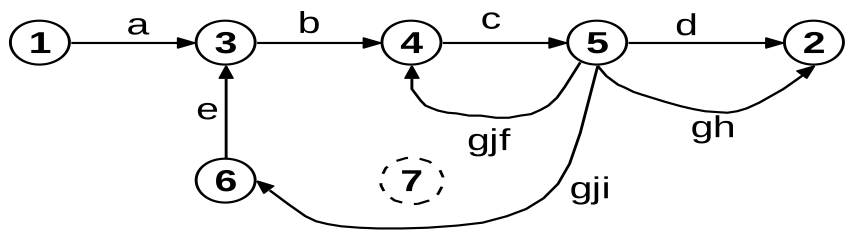
- Consider the following graph.



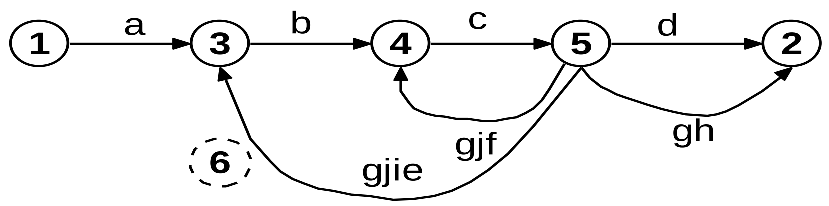
- First remove node 8 by applying step 4 (cross-term step) and combine by step 5.



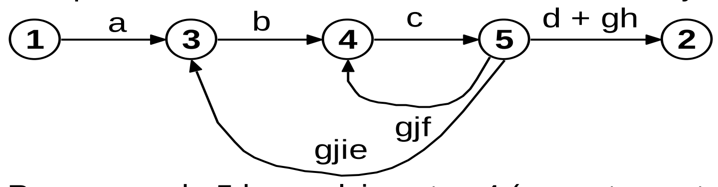
- Remove node 7 by applying step 4 (cross-term step) and combine by step 5.



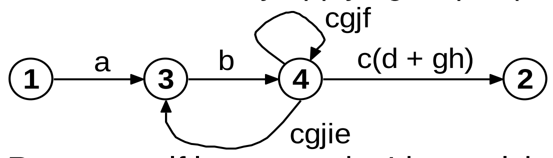
- Remove node 6 by applying step 4 (cross-term step) and combine by step 5.



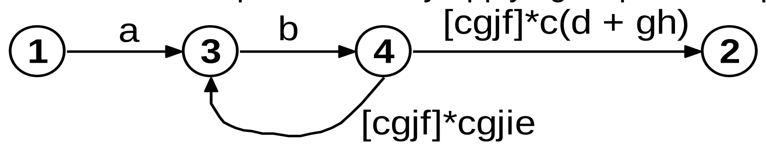
- Add parallel links between node 5 and node 2 by applying parallel term step.



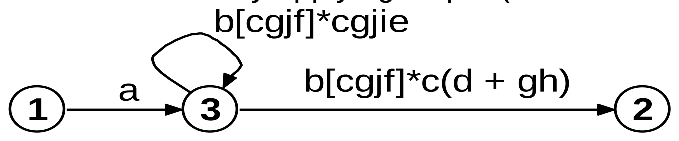
- Remove node 5 by applying step 4 (cross-term step) and combine by step 5.



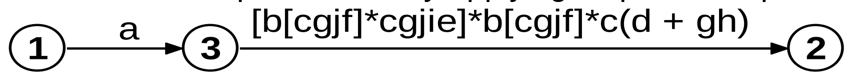
- Remove self loop at node 4 by applying loop term step.



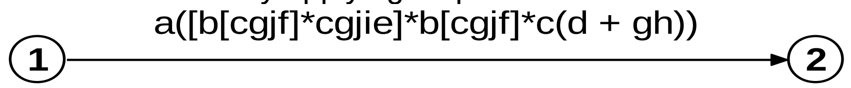
- Remove node 4by applying step 4 (cross-term step) and combine by step 5.



- Remove self loop at node 3 by applying loop term step.



- Remove node 3 by applying step 4.



<u>(3) Applications:</u>

<u>(1) How many paths in a Flowgraph:</u>

<u>Q. Explain maximum path count arithmetic of a flowgraph with an example?</u>

<u>Maximum Path Count Arithmetic:</u>

- Here each link is represented by a link weight. There are three arithmetic cases that are considered here.
- They are

- **(i) Parallel rule:**
  - Each term of the path expression A is added with each term of the path expression B if there are two path expressions A and B. So it is A+B. If there are W_A paths in A and W_B paths in B then there are $W_A + W_B$ paths in its combination.

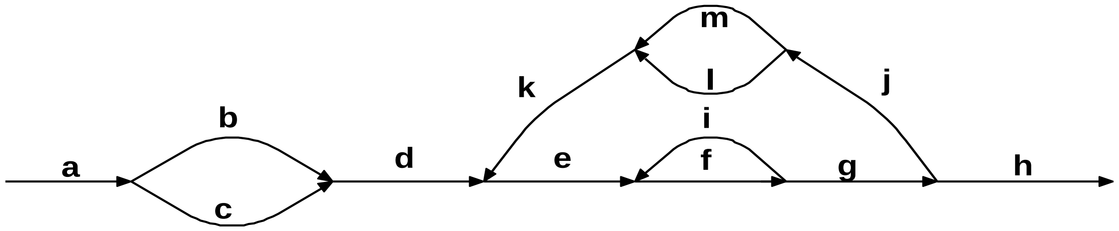
  (ii) Series rule:
  - Each term of the path expression A is multiplied with each term of the path expression B if there are two path expressions A and B. So it is AB. If there are W_A paths in A and W_B paths in B then there are $W_A W_B$ paths in its combination.

  (iii) Loop rule:
  - Loop rule is evaluated by considering number of times that the path is iterated.

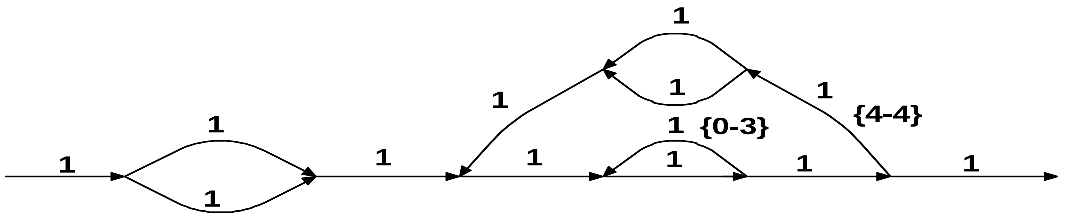
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	A + B	$W_A + W_B$
SERIES	AB	$W_A W_B$
LOOP	A^n	$\sum_{i=0}^{n} W_A^{\,i}$

Example:

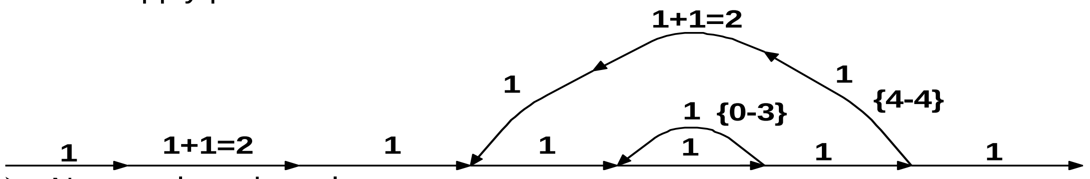
- Determine the path expression to the following figure.



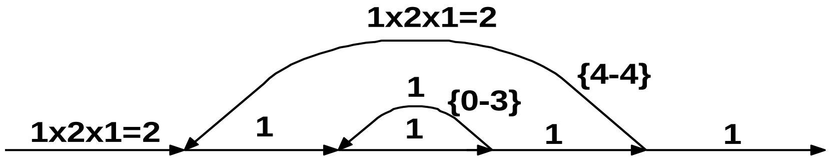
- The path expression is given by
  a(b +c) d [e(fi)*fgj(m + l)k]*e(fi)*fgh
- Let each link represents a single link and is given by a link weight 1.
- Assume that the outer loop will be taken exactly four times and the inner loop can be taken zero to three times.
- The reduction is as follows.



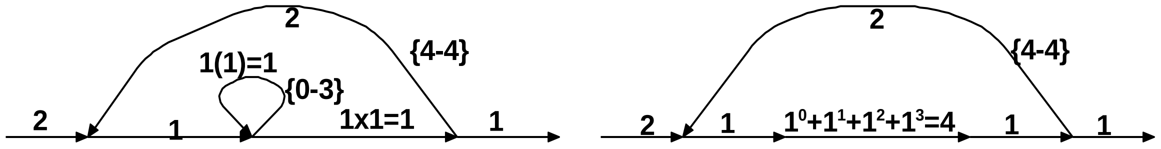
- Now apply parallel rule.



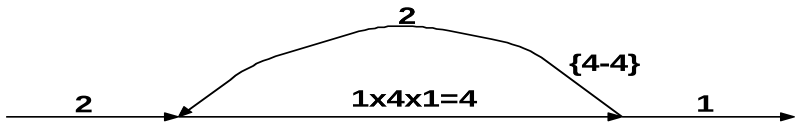
- Now apply series rule.



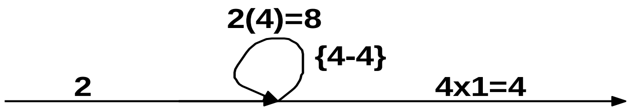
- Now create inner self loop & Apply loop rule for removing inner self loop.



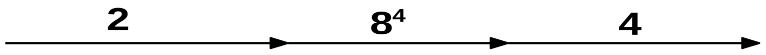
- Now apply series rule.



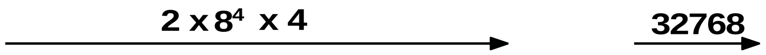
- Now create outer self loop.



- Apply loop rule to remove the self loop.



- Apply series rule.



- Alternatively we can calculate the maximum number of paths as follows.
- The path expression is given by

  a(b +c) d [e(fi)*fgj(m + l)k]*e(fi)*fgh
- In the above expression each link is substituted by 1.

  $1(1+1)1[1(1x1)^3\ 1x1x1\ (1+1)1]^4\ 1(1x1)^3\ 1x1x1$

  $=1(2)[1^3 \text{ x } 2]^4\ 1x1^3$

  $=2[4x2]^4 \text{ x } 4$ [since $1^3 = 1^0 + 1^1 + 1^2 + 1^3 = 4$]

  $=2 \text{ x } [8]^4 \text{ x } 4 = 32,768..$

(2) Approximate Minimum number of paths:

Q. Define structured code. Explain about lower path count arithmetic?

Structured code:

- A structured flowgraph is one that can be reduced to a single link by successive application of transformations.
- Based on the path expression obtained by node-by-node reduction procedure we can determine whether the given flow graph is a structured or unstructured.
- That is if the resultant expression is large and ugly then the graph is unstructured one otherwise the graph is structured one.

Lower path count arithmetic:

- The lowest number of paths in a structured flowgraph can be approximately known; it may or may not be accurate because there is every possibility of a path being unachievable which further lowers the number count.
- Here each link is represented by a link weight. Loops are always problematic.

- So it must be traversed only one time or zero times to achieve the coverage. There are three arithmetic cases here. They are.
- **(i) Parallel rule:**
  - Each term of the path expression A is added with each term of the path expression B if there are two path expressions A and B. So it is A+B. If there are W_A paths in A and W_B paths in B then there are $W_A + W_B$ paths in its combination.

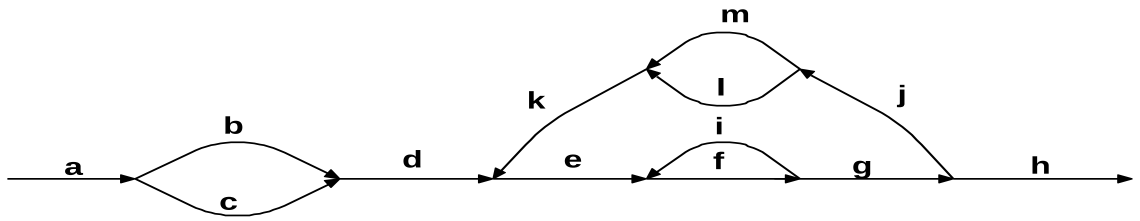
  (ii) Series rule:
  - Each term of the path expression A is multiplied with each term of the path expression B if there are two path expressions A and B. So it is AB.
  - If there are W_A paths in A and W_B paths in B then there are MAX (W_A, W_B) paths in its combination.

  (iii) Loop rule:
  - Loop rule is taken either by considering only one time that the path is iterated or zero times the path is iterated. So it gives the value 1 or its link weight.

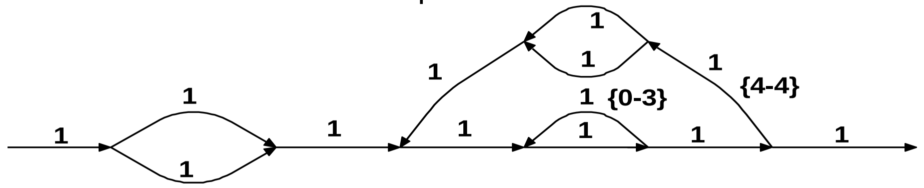
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	A + B	$W_A + W_B$
SERIES	AB	$MAX(W_A,W_B)$
LOOP	A^n	$1,W_1$

Example:

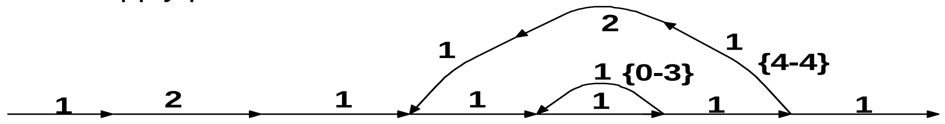
- Determine the path expression to the following figure.



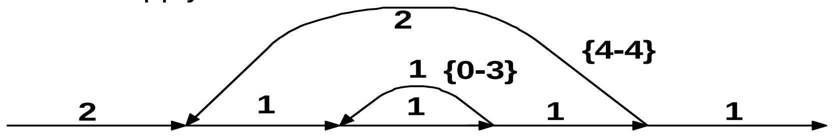
- The path expression is given by a(b +c) d [e(fi)*fgj(m + l)k]*e(fi)*fgh
- Let each link represents by a link weight 1. Assume that the outer loop will be taken exactly four times and the inner loop can be taken zero to three times. The reduction is as follows.



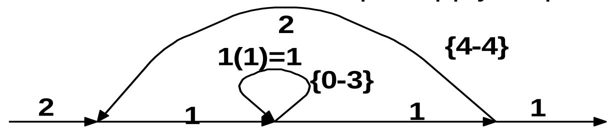
- Now apply parallel rule.

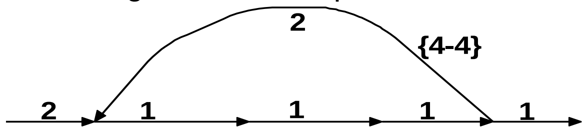


- Now apply series rule.

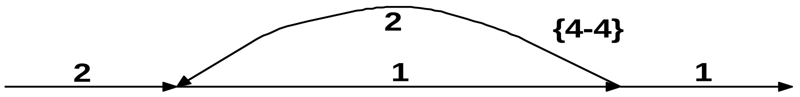


- Now create inner self loop & apply loop rule for removing inner self loop.

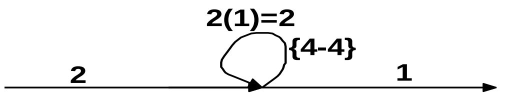




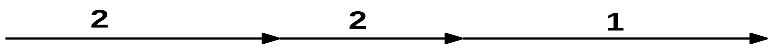
- Now apply series rule.



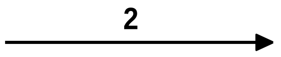
- Now create outer self loop.



- Apply loop rule to remove self loop.



- Apply series rule.



- Alternatively we can calculate the minimum number of paths as follows.
- The path expression is given bya(b +c) d [e(fi)*fgj(m + l)k]*e(fi)*fgh
- In the above expression each link is substituted by 1.

  $1(1+1)1[1(1x1)^0$ 1x1x1 $(1+1)1]^0$ $1(1x1)^0$ 1x1x1

  $=1(2)[1^0 \text{ x } 2]^0$ $1x1^0$ =2x1 = 2

(3) The probability of getting there:

Q. What is the probability of path expressions? Write arithmetic rules. Explain with an example.

Probability of path expressions:

- Specify each out link of a node equal to the probability of that link. The sum of the out link probabilities is equal to 1. For a simple loop, if the loop is taken N times then the looping probability is N/(N+1) and non looping probability is 1/(N+1).
- There are three arithmetic cases here. They are

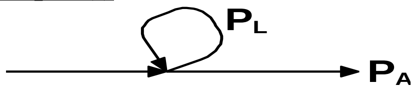
  Parallel rule:
  - Each term of the path expression A is added with each term of the path expression B if there are two path expressions A and B. So it is A+B.
  - If there is a path expression A with Probability P_A and path expression B with Probability P_B then the resultant probability is $P_A + P_B$.

  Series rule:
  - Each term of the path expression A is multiplied with each term of the path expression B if there are two path expressions A and B. So it is AB. If there is a path expression A with Probability P_A and path expression B with Probability P_B then the resultant probability is P_A P_B

  Loop rule:
  - If the probability of looping node is P_L and the probability of link leaving the loop node is P_A then $P_A + P_L=1$. So $P_A = 1- P_L$

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

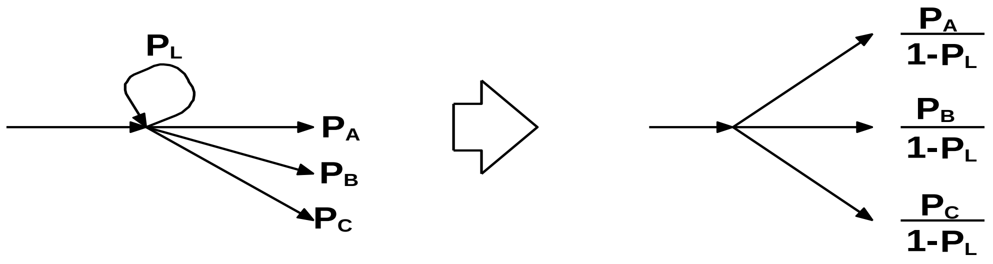
Example (i)



$P_A = 1 - P_L$

New Probability $P_{NEW} = P_A / (1-P_L) = (1-P_L) / (1-P_L) = 1$

Example (ii)



Here $P_L + P_A + P_B + P_C = 1$

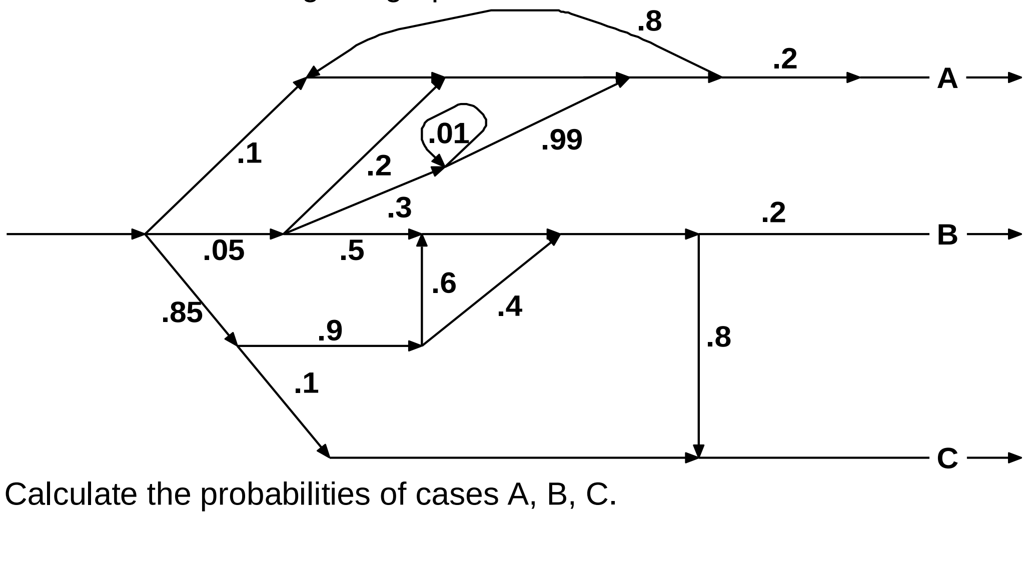
$1 - P_L = P_A + P_B + P_C$

$P_A / (1 - P_L) + P_B / (1 - P_L) + P_C / (1 - P_L) = (P_A + P_B + P_C) / (1 - P_L)$

$= (P_A + P_B + P_C) / (P_A + P_B + P_C) = 1$

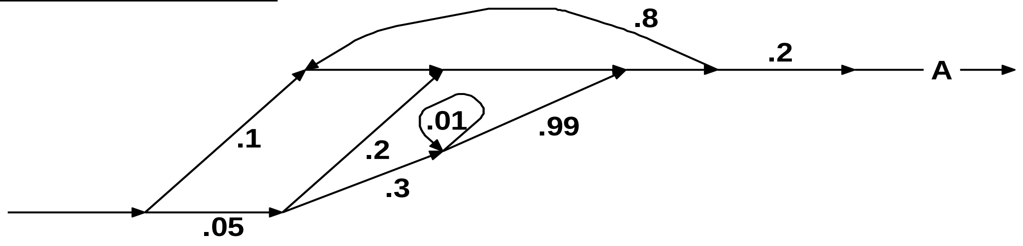
Example:

- Consider the following flowgraph.

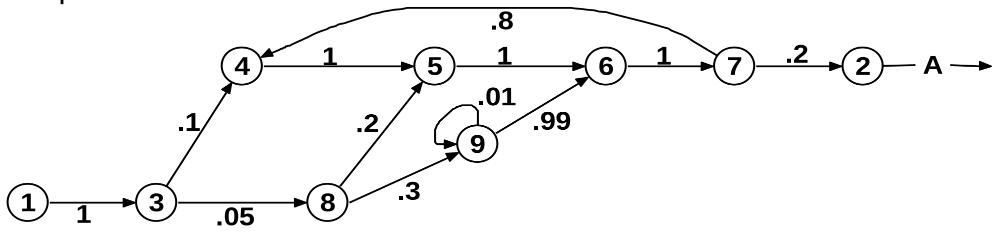


- Calculate the probabilities of cases A, B, C.

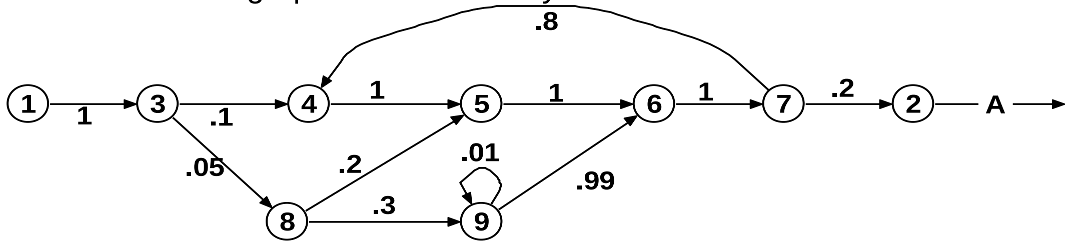
First consider case A:



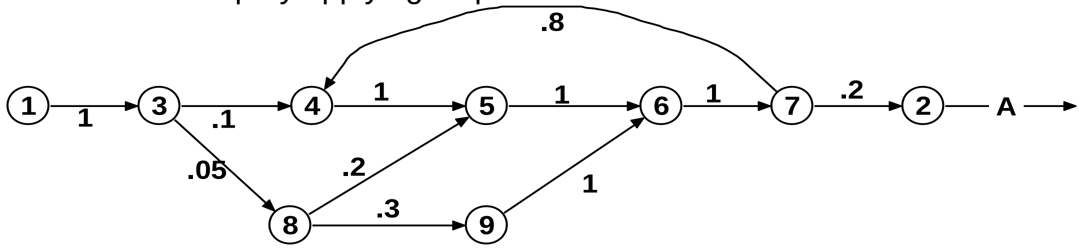
- In the above flowgraph if the link weight is not specified then it is specified by 1 and also represents its nodes as follows.



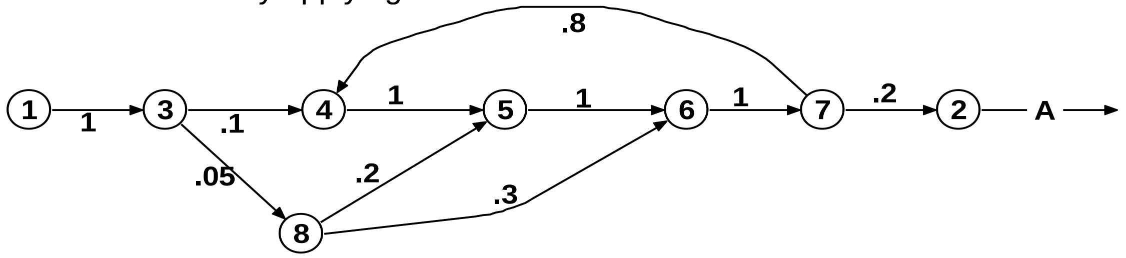
- The above flowgraph is also taken by



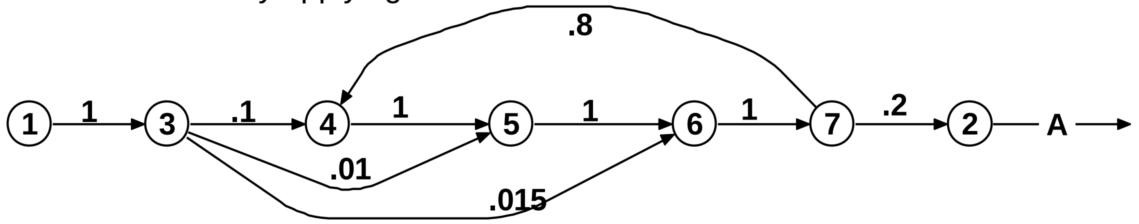
- Remove self loop by applying loop rule



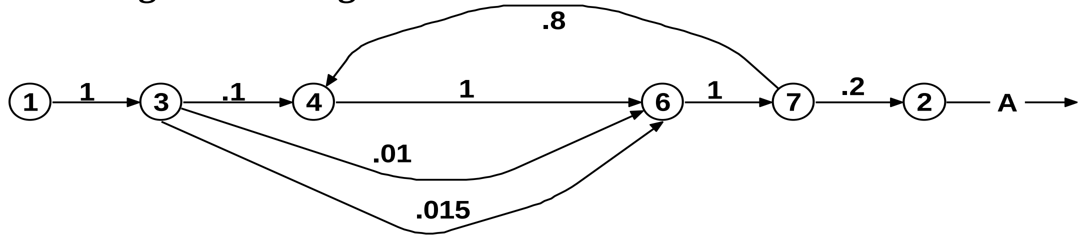
- Remove node 9 by applying series rule



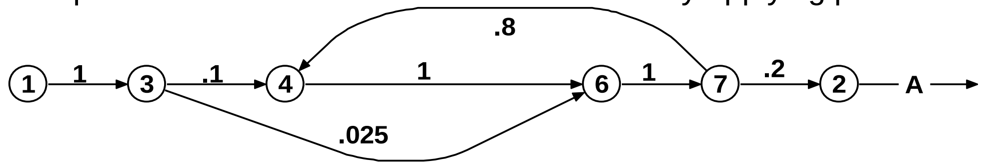
- Remove node 8 by applying series rule



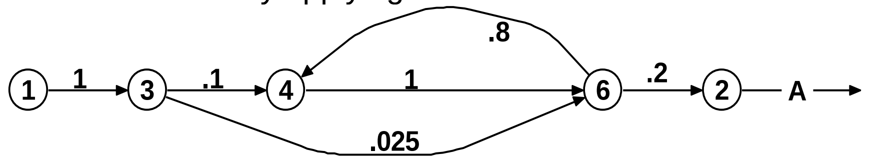
- Remove node 5 by applying series rule



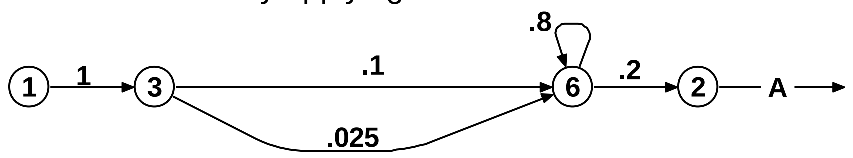
- Add parallel links between node 3 and node 6 by applying parallel rule



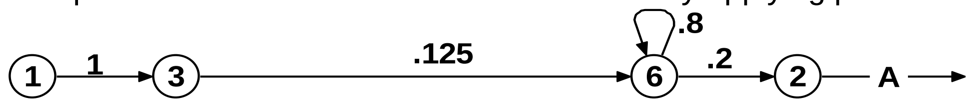
- Remove node 7 by applying series rule



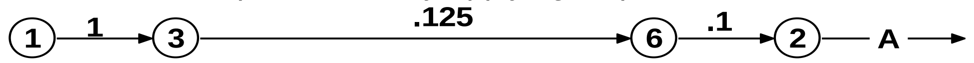
- Remove node 4 by applying series rule



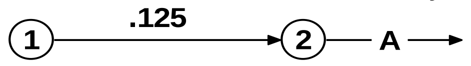
- Add parallel links between node 3 and node 6 by applying parallel rule



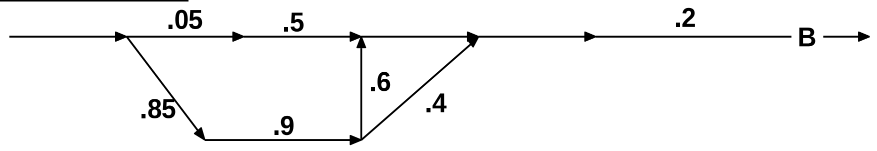
- Remove self loop at node 6 by applying loop rule



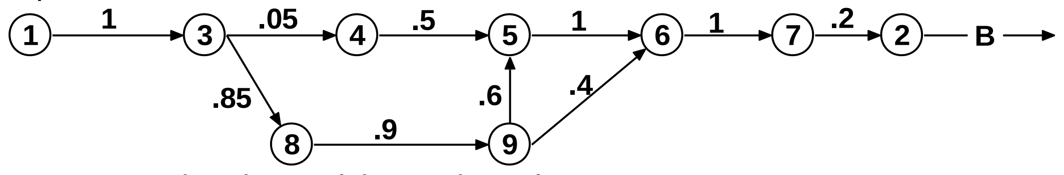
- Remove node 3 and node 6 by applying loop rule



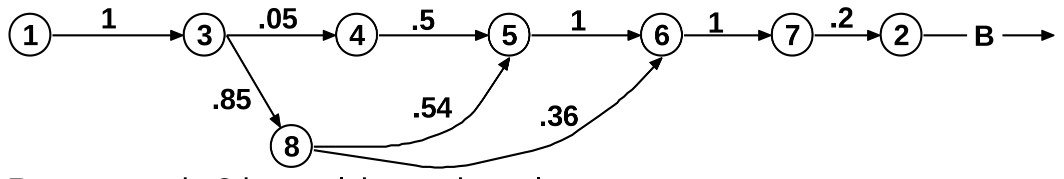
Consider case B:



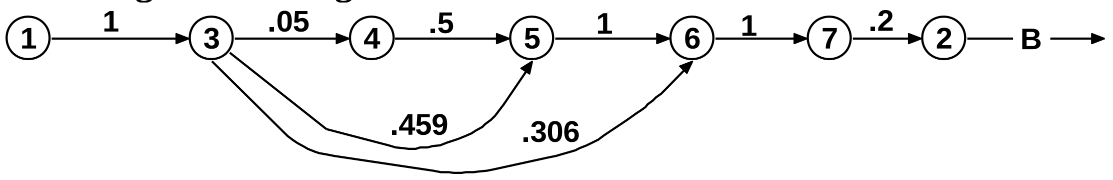
- In the above flowgraph if the link weight is not specified then it is specified by 1 and also represents its nodes as follows.



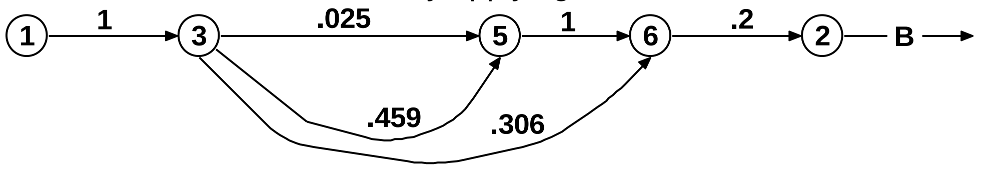
- Remove node 9 by applying series rule.



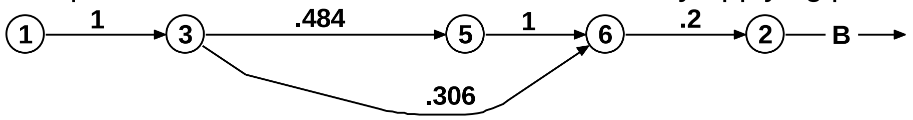
- Remove node 8 by applying series rule.



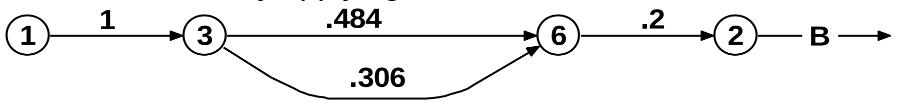
- Remove node 4 and node 7 by applying series rule.



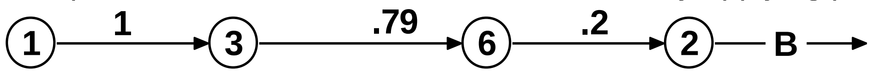
- Add parallel links between node 3 and node 5 by applying parallel rule



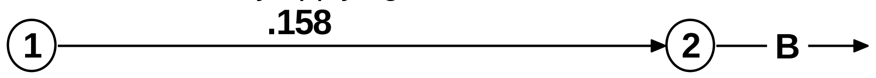
- Remove node 5 by applying series rule.



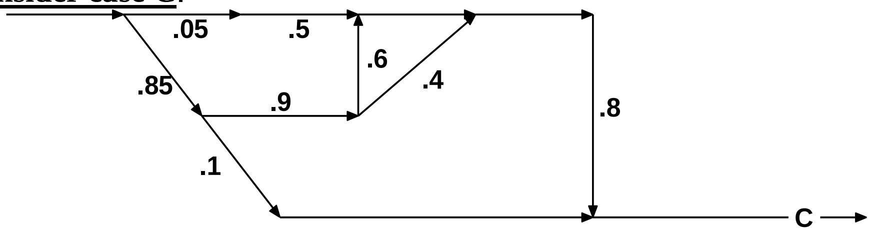
- Add parallel links between node 3 and node 5 by applying parallel rule



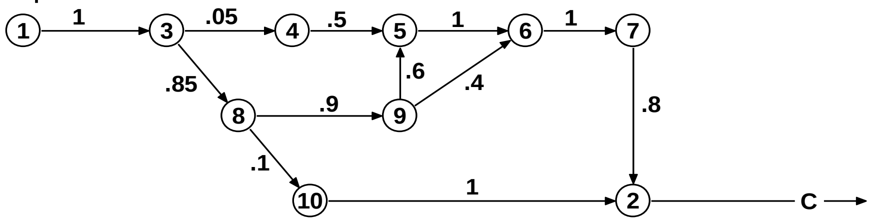
- Remove node 5 by applying series rule.



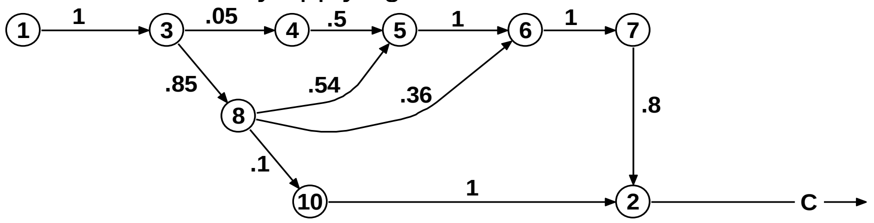
Consider case C.



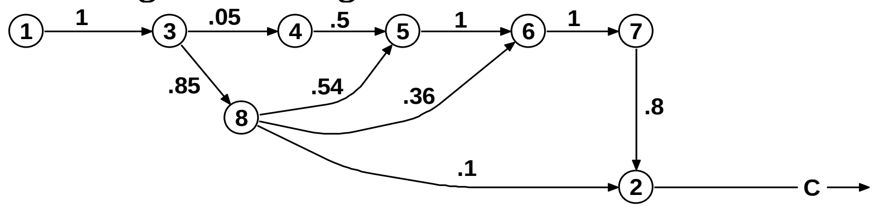
- In the above flowgraph if the link weight is not specified then it is specified by 1 and also represents its nodes as follows.



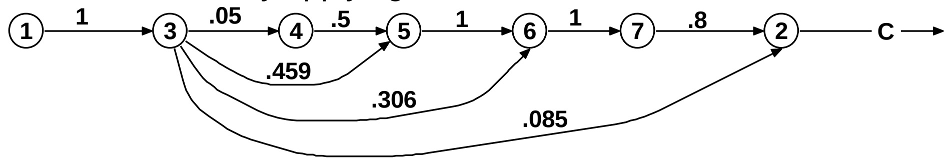
- Remove node 9 by applying series rule.



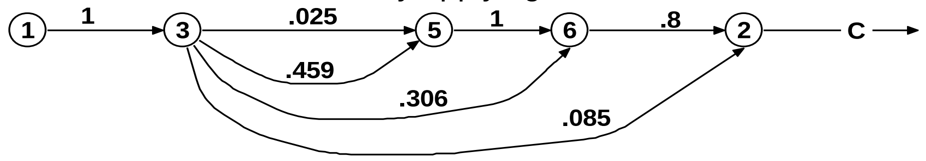
- Remove node 10 by applying series rule.



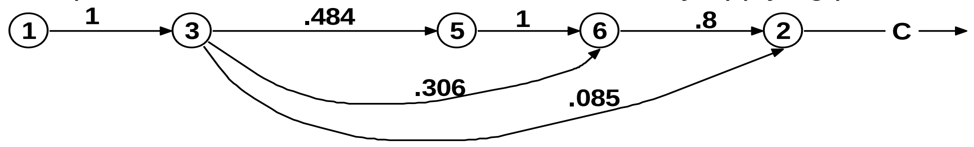
- Remove node 8 by applying series rule.



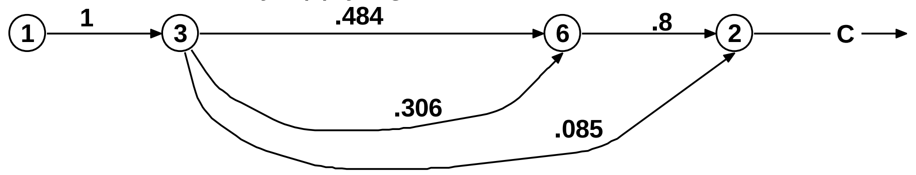
- Remove node 7 & node 4 by applying series rule.



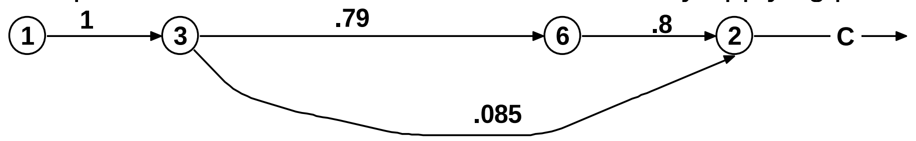
- Add parallel links between node 3 and node 5 by applying parallel rule



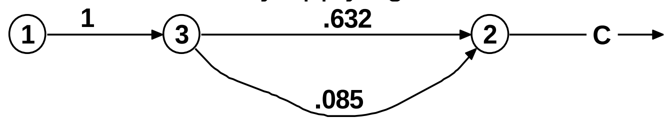
- Remove node 5 by applying series rule



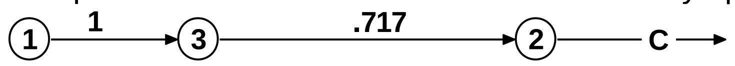
- Add parallel links between node 3 and node 6 by applying parallel rule



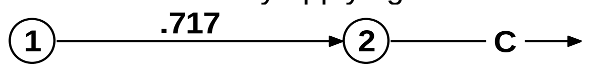
- Remove node 6 by applying series rule



- Add parallel links between node 3 and node 2 by applying parallel rule



- Remove node 3 by applying series rule



Cross check:

- Sum of case A + case B + case C = .125 + .158 + .717 =1.

(4) The mean processing time of a routine

Q. What is the mean processing time of a routine? Write arithmetic rules. Explain with an example.

Mean processing time of a routine:

- Here every link has two weights.

- One is the processing time for that link denoted by T, & other one is the probability of that link denoted by P.
- There are three arithmetic cases here.
- They are

Parallel rule:

- It is the arithmetic mean of all processing time over all parallel links.

Series rule:

- It is the sum of two processing times.

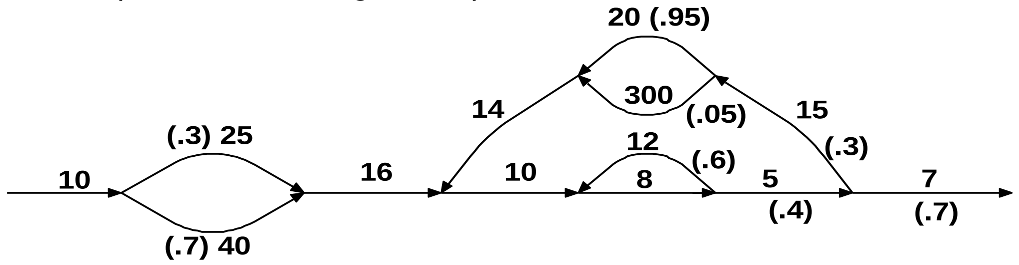
Loop rule:

- It is evaluated by considering number of times the path is iterated

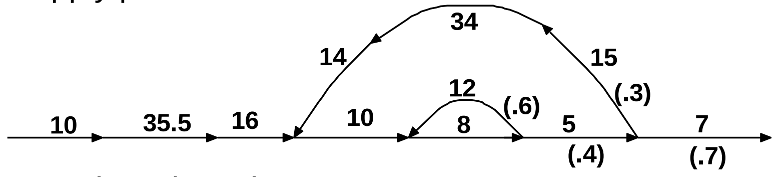
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	A + B	$T_{A+B} = (P_A T_A + P_B T_B)/(P_A + P_B)$ $P_{A+B} = P_A + P_B$
SERIES	AB	$T_{AB} = T_A + T_B$ $P_{AB} = P_A P_B$
LOOP	A^*	$T_A = (T_L P_L)/(1-P_L) + T_A$ $P_A = P_A/(1-P_L)$

Example:

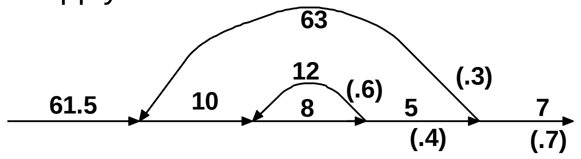
- The following figure is represented by, loop probabilities, and processing time for each link. The probabilities are given in parentheses.



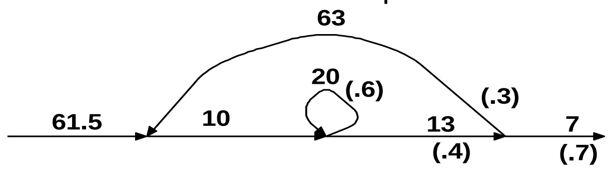
- Apply parallel rule.



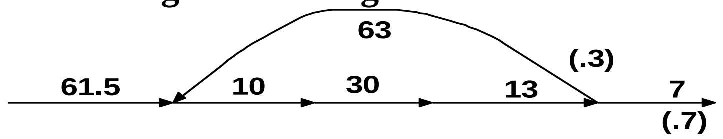
- .Apply series rule.



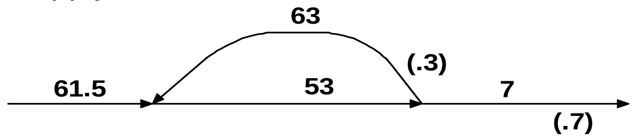
- Now create inner self loop.



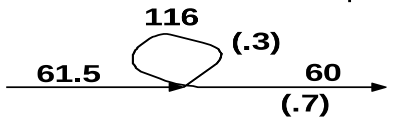
- Remove the inner self loop by applying loop rule.



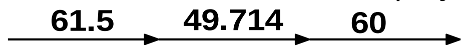
- Apply series rule.

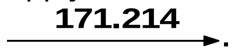


- Create the outer self loop.



- Remove the outer self loop by applying loop rule.



- Apply series rule
- 

<u>(5) Push/Pop, Get/Return</u>

<u>Q. What is Push/Pop, Get/Return? Write arithmetic rules. Explain with an example.</u>

<u>Push/Pop:</u>

- Here PUSH operation is used to insert elements into the stack. POP operation is used to remove elements from the stack.
- Apart from PUSH/POP other operations are GET/RETURN, OPEN/CLOSE and START/STOP.
- There are three arithmetic cases here.
- They are

  <u>Parallel rule:</u>

  ❖ Each term of the path expression A is added with each term of the path expression B if there are two path expressions A and B. So it is A+B. If there are W_A paths in A and W_B paths in B then there are $W_A + W_B$ paths in its combination.

  <u>Series rule:</u>

  ❖ Each term of the path expression A is multiplied with each term of the path expression B if there are two path expressions A and B. So it is AB. If there are W_A paths in A and W_B paths in B then there are $W_A W_B$ paths in its combination.

  <u>Loop rule:</u>

  ❖ It is evaluated by considering number of times the path is iterated.

CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	A + B	$W_A + W_B$
SERIES	AB	$W_A W_B$
LOOP	A^*	W^*_A

- PUSH/POP operations satisfy commutative, associative, and distributive law of addition and multiplication.
- The arithmetic tables for PUSH/POP are given by

PUSH/POP MULTIPLICATION TABLE

X	H	P	1
H	H^2	1	H
P	1	P^2	P
1	H	P	1

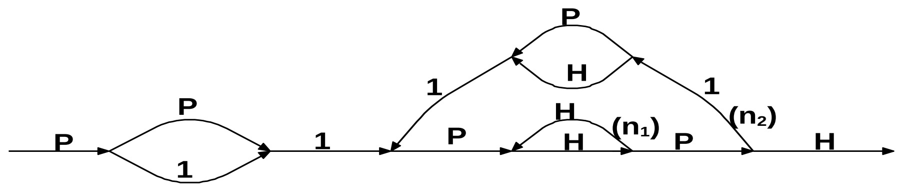
PUSH/POP ADDITION TABLE

+	H	P	1
H	H	P+H	H+1
P	P+H	P	P+1
1	H+1	P+1	1

- These tables are used to determine the weight of addition and multiplication operation.
- Here H represents the PUSH operation, P represents the POP operation and 1 represents NO operation.

Example:

- Consider the following flowgraph.



❖ Path expression for the above flowgraph is.

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

❖ Simplifying by using the arithmetic tables

PUSH/POP = $(P^2 + P)[P(HH)^{n1}(P+H)]^{n2}(HH)^{n1}$

$= (P^2+P)[H^{2n1}(P^2+1)]^{n2}H^{2n1}$

- Let us consider M_1,M_2 represents the two looping terms. i.e. M_1 represents the number of times the inner loop is considered, M_2 represents the number of times the outer loop is considered.

CASE (i)

Consider $M_1=0$, $M_2=0$ (i.e. $n_1=0$, $n_2=0$)

PUSH/POP= $(P+P^2)[H^0(P^2+1)]^0H^0 = P + P^2$

CASE (ii)

Consider $M_1=0$, $M_2=1$ (i.e. $n_1=0$, $n_2=1$)

PUSH/POP= $(P+P^2)[H^0(P^2+1)]^1H^0$

$= (P + P^2)[1+P^2] = P + P^2 + P^3 + P^4$

- For different combination of M_1, M_2 values the following table is obtained.

M_1	0	0	0	0	1	1	1	1	2	2	2	2
M_2	0	1	2	3	0	1	2	3	0	1	2	3
PUSH /POP	$P + P^2$	$P + P^2 + P^3 + P^4$	$\sum_{1}^{6} P^i$	$\sum_{1}^{8} P^i$	1+H	$\sum_{0}^{3} H^i$	$\sum_{0}^{5} H^i$	$\sum_{0}^{7} H^i$	H^2+H^3	$\sum_{4}^{7} H^i$	$\sum_{6}^{11} H^i$	$\sum_{8}^{16} H^i$

Get/Return:

- The arithmetic tables for GET/RETURN are.

GET/RETURN MULTIPLICATION TABLE

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

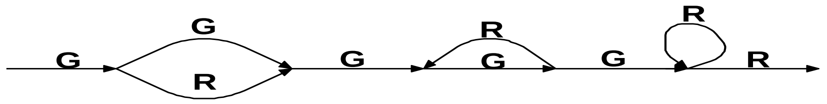
GET/RETURN ADDITION TABLE

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

- The arithmetic table for GET/RETURN is same as that of PUSH/POP.

Example:

- Consider the following flowgraph.



- Path expression for the above flowgraph is. G(G+R) G(GR)* GGR* R
- Simplifying by using the arithmetic tables

  GET/RETURN = $G(G+R)G^3 R^*R$

  $= (G+R)\, G^3 R^* = (G^4 + G^3R)\, R^* = (G^4 + G^2GR)R^* = (G^4 + G^2)R^*$

(6) Limitations and Solutions

Q. What are the limitations and solutions of the applications?

- 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 achievable, but may provide misleading results when some paths are unachievable.
- The solution to handling unachievable paths is to partition the graph into subgraphs so that all paths in each of the subgraphs are achievable. But the resulting sub graphs may overlap, because one path may be common to several different subgraphs.
- Each predicate's truth value splits the graph into two subgraphs.
- For n predicates there may be 2^n sub graphs. Here there is an 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 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 should be included in this analysis not all predicates that may control the program flow.

(4) Regular expressions and flow anomaly detection:

Q. Explain about Regular expression and Flow-Anomaly detection?

(i) The Problem:

- The generic flow-anomaly detection problem is used to search for a specific sequence of operations considering all possible paths through a routine.
- Let's say the operations are SET and RESET, denoted by *s* and *r* respectively, and we want to know if there is a SET followed immediately by a SET or a RESET followed immediately by a RESET (i.e, an *ss* or an *rr* sequence).

- Flow anomaly detection is used to know if particular sequence occurred, but not to know the total impact of the procedure.
- It is used to detect the bug sequence in the following situations.

  1. A file can be opened (*o*), closed (*c*), read (*r*), or written (*w*). If the file is read or written to after it is closed, then it is anomalous. i.e. *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.

  2. The operations performed by tape transport device are read(r), write(w), rewind (*d*), forward (*f*), skip (*k*) and stop (p). In a tape-transport device rewind and forward operations cannot be performed one after the other without performing stop operation. So the following sequences are anomalous: *df, dr, dw, fd*, and *fr*.

  3. With the help of generic flow anomaly detection, it is possible to detect the data flow bugs sequence such as *dd, dk, kk*, and *ku.*

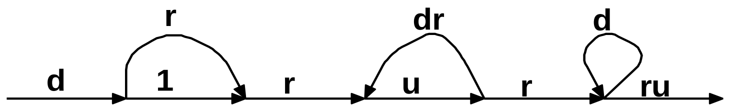
  4. A bug that occur only if two operations a and b occurred in the order aba or bab.

(ii) Huang Theorem:

- Annotate each link in the graph with the appropriate operator or the null operator 1.
- Simplify things using $a + a = a$ and $1^2 = 1$.
- The regular expression obtained should be simplified carefully, as null operations cannot be combined with other operations.
- For example, 1*a* may not be the same thing as a alone. Huang theorem is used to simplify the regular expression and to examine the specific operation sequence.
  - ❖ 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 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 if *ss* is a substring of $pp(srr)^n rp$ then ss will appear in $pp(srr)^2 rp$.
  - ❖ Similarly let A = $p + pp + ps$ B = $psr + ps(r + ps)$ C = rp T = P^4
  - ❖ If p^4 is a substring of AB^nC then p^4 will appear in AB^2C $(p + pp + ps)[psr + ps(r + ps)]^2 rp$
- Huang theorem is also useful in test design.
- Further Huang 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.
- Two character string sequences are used to represent data flow anomaly. Then using Huang's theorem these anomalous can be detected if these loop is iterated twice.

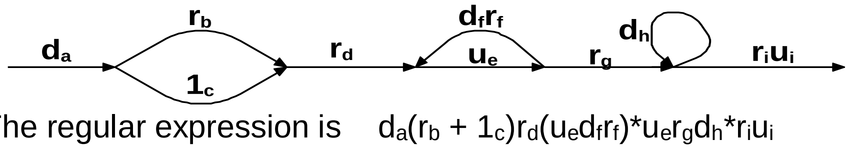
Data Flow Testing Example:

- ❖ By assigning appropriate operators on each link the following flowgraph can be used to detect different anomalies bugs.



- 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 and also a *dk* bug cannot occur.

  $(drr + dr)(1 + udrudr)(urru + urd^2ru)$
- A better way to the above is subscript the operator with the link name.



- 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:

  $d_a(r_b + 1_c)r_d(1 + (u_e d_f r_f)^2)u_e r_g(1 + d^2{}_h)r_i u_i$

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

<u>(iii) Generalizations, Limitations and comments:</u>

- Huang's theorem can be easily generalized to cover sequences of greater length than two characters. 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 $P^{\underline{k}}$ for every appearance of P*

  $P^{\underline{k}} = 1 + P + P^2 + P^3 + . . . + P^k$
- In order to find the starting and ending sequence of strings in a path expression, the mathematical approaches such as application of derivations to algebraic expression makes it easier and time consuming than the path tracing process on a flowgraph.
- Static flow analysis methods can't determine whether a path is achievable or is not achievable.
- If unachievable paths exist, then the exactness and applicability of all flow analysis methods reduces gradually. Hence achievable paths are preferred in order to overcome the problems of unachievable paths.