UNIT V

State, State Graphs and Transition Testing: State Graphs, Good & Bad State Graphs, State Testing, Testability Tips.

STATE GRAPHS:

The word **"state"** is a combination of circumstances or attributes belonging for the time being to a person or thing.

- A program that detects the character sequence "ZCZC" can be in the following states:
  1. Neither ZCZC nor any part of it has been detected.
  2. Z has been detected.
  3. ZC has been detected.
  4. ZCZ has beendetected.
  5. ZCZC has beendetected.
- A moving automobile whose engine is running can have the following states with respect to its transmission:
  1. Reverse gear
  2. Neutral gear
  3. First gear
  4. Second gear
  5. Third gear
  6. Fourth gear

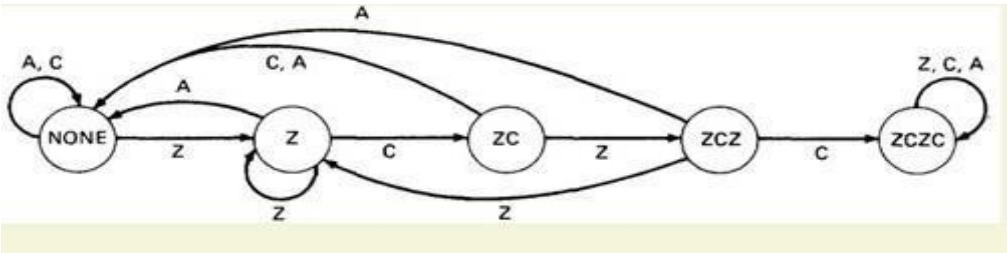


Figure: One-Time ZCZC Sequence-Detector State Graph.

Inputs and Transitions

- Whatever is being modeled is subjected to inputs. As a result of those inputs, the state changes, or is said to have made a **transition.**
- Transitions are denoted by links that join the states.
- The ZCZC detection example can have the following kinds of inputs:
  1. Z
  2. C
  3. Any character other than Z or C, which we'll denote by A
- The state graph is interpreted as follows:
  1. If the system is in the "NONE" state, any input other than a Z will keep it in that state.
  2. If a Z is received, the system transitions to the "Z" state.
  3. If the system is in the "Z" state and a Z is received, it will remain in the "Z" state. If a C is received, it will go to the "ZC" state; if any other character is received, it will go back to the "NONE" state because the sequence has been broken.
  4. A Z received in the "ZC" state progresses to the "ZCZ" state, but any other character breaks the sequence and causes a return to the "NONE" state.
  5. A C received in the "ZCZ" state completes the sequence and the system enters the "ZCZC" state. A Z breaks the sequence and causes a transition back to the "Z" state; any other character causes a return to the "NONE" state.

6. The system stays in the "ZCZC" state no matter what is received.

Outputs

- An output can be associated with any link.
- Outputs are denoted by letters or words and are separated from inputs by a slash as follows: "input/output."
- Output denotes anything of interest that's observable and is not restricted to explicit outputs by devices.
- Outputs are also link weights. If every input associated with a transition causes the same output, then denote it as: "input 1, input 2 . . . input 3/output."

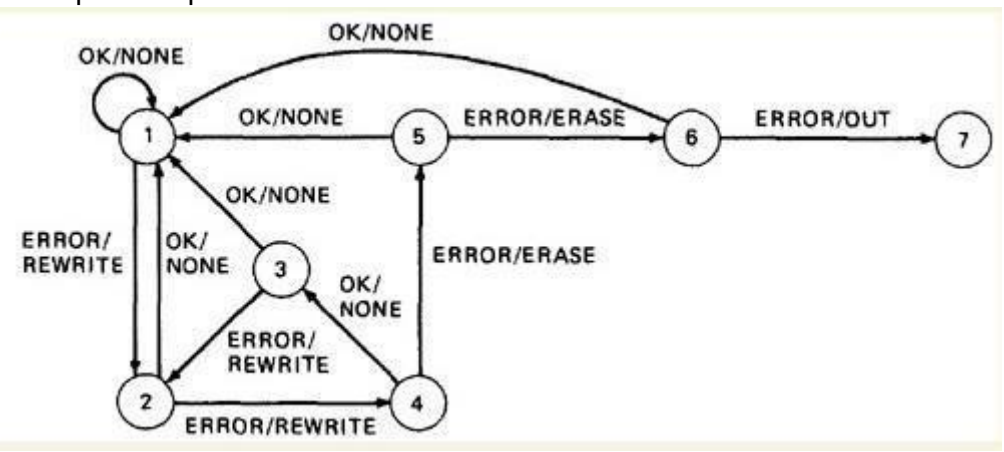


Figure: Tape Control Recovery Routine State Graph.

State Tables

- It's more convenient to represent the state graph as a table The following conventions are used:

STATE	OKAY	ERROR
1	1/NONE	2/REWRITE
2	1/NONE	4/REWRITE
3	1/NONE	2/REWRITE
4	3/NONE	5/ERASE
5	1/NONE	6/ERASE
6	1/NONE	7/OUT
7	. . .	. . .

- The **state table** or **state-transition table** specifies the states, the inputs, the transitions, and the outputs.

1. Each row of the table corresponds to a state.
2. Each column corresponds to an input condition.
3. The box at the intersection of a row and column specifies the next state (the transition) and the output, if any.

Time versus Sequence

State graphs don't represent time—they represent sequence. A transition might take microseconds or centuries;

Software Implementation

- There is rarely a direct correspondence between programs and the behavior of a process described as a state graph.
- There are four tables involved:

1. A table or process that encodes the input values into a compact list (INPUT_CODE_TABLE).
2. A table that specifies the next state for every combination of state and input code (TRANSITION_TABLE).
3. A table or case statement that specifies the output or output code, if any, associated with every state-input combination (OUTPUT_TABLE).
4. A table that stores the present state of every device or process that uses the same state table—e.g., one entry per tape transport (DEVICE_TABLE).

- The routine operates as follows, where # means concatenation:

```
BEGIN
PRESENT_STATE := DEVICE_TABLE(DEVICE_NAME)
ACCEPT INPUT_VALUE
INPUT_CODE := INPUT_CODE_TABLE(INPUT_VALUE)
POINTER := INPUT_CODE#PRESENT STATE
NEW_STATE := TRANSITION_TABLE(POINTER)
OUTPUT_CODE := OUTPUT_TABLE(POINTER)
CALL OUTPUT_HANDLER(OUTPUT_CODE)
DEVICE_TABLE(DEVICE_NAME)  :=  NEW_STATE
END
```

Points:

1. The present state is fetched from memory.
2. The present input value is fetched. If it is already numerical, it can be used directly; otherwise, it may have to be encoded into a numerical value, say by use of a case statement, a table, or some other process.
3. The present state and the input code are combined (e.g., concatenated) to yield a pointer (row and column) of the transition table and its logical image (the output table).
4. The output table, either directly or via a case statement, contains a pointer to the routine to be executed (the output) for that state-input combination. The routine is invoked.
5. The same pointer is used to fetch the new state value, which is then stored.

Input Encoding and Input Alphabet

- The input encoding could be implemented as a table lookup in a table that contained the following codes: "OTHER" = 0, "Z" = 1 and "C" = 2.
- Alternatively, we could implement it as a process: IF INPUT = "Z" THEN CODE: = 1 ELSE IF INPUT = "C" THEN CODE: = 2 ELSE CODE: = 0 ENDIF.
- Input encoding compresses the cases and therefore the state graph.
- Another advantage of input encoding is that we can run the machine from a mixture of otherwise incompatible input events, such as characters, device response codes

Output Encoding and Output Alphabet

- We might want to output a string of characters, call a subroutine, transfer control to a lower-level finite-state machine, or do nothing. Whatever we might want to do, there are only a finite number of such distinct actions, which we can encode into a convenient **output alphabet.**
- The word "output" as used in the context of finite-state machines means a "character" from the output alphabet.

State Codes and State-Symbol Products

- The term **state-symbol product** is used to mean the value obtained by any scheme used to convert the combined state and input code into a pointer to a compact table without holes.
- "States" and "state codes" in the context of finite-state machines, we mean the (possibly) hypothetical integer used to denote the state and not the actual form of the state code that could result from an encoding process.

- Similarly, "state-symbol product" means the hypothetical (or actual) concatenation used to combine the state and input codes.

GOOD & BAD STATE GRAPHS

- What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test design context. Here are some principles for judging:
  1. The total number of states is equal to the product of the possibilities of factors that make up the state.
  2. For every state and input there is exactly one transition specified to exactly one, possibly the same, state.
  3. For every transition there is one output action specified. That output could be trivial, but at least one output does something sensible.
  4. For every state there is a sequence of inputs that will drive the system back to the same state.

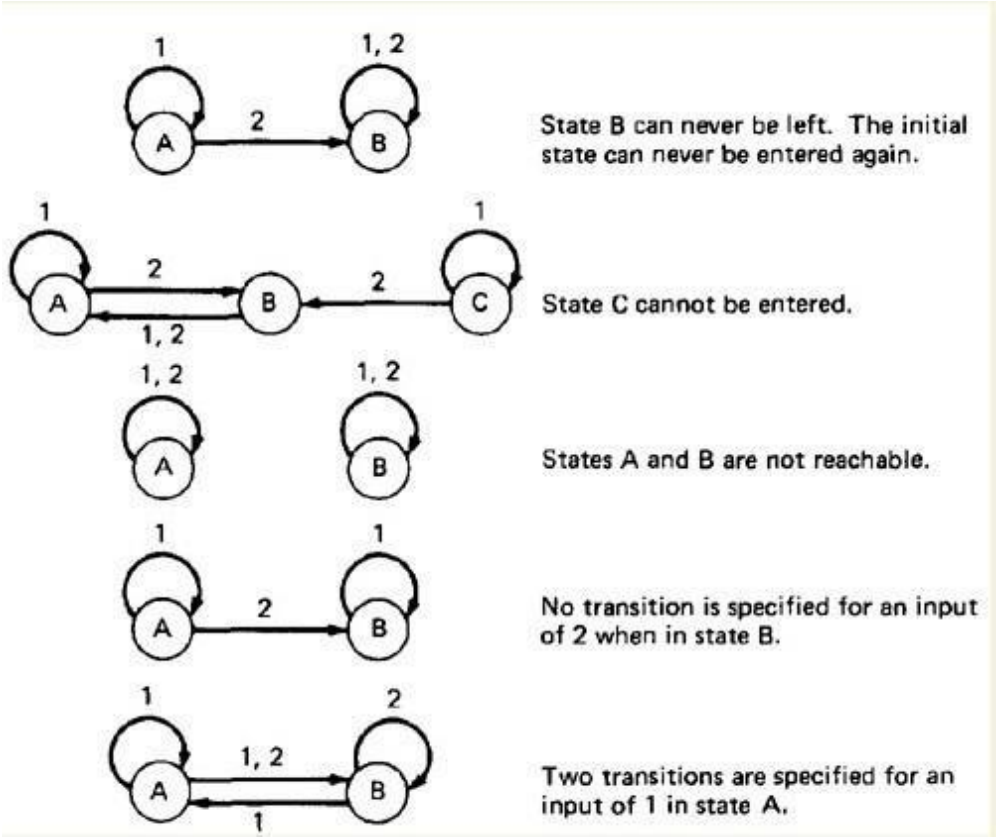


Figure: Improper State Graphs.

State Bugs

Number of States

Find the number of states as follows:

1. Identify all the component factors of the state.
2. Identify all the allowable values for each factor.
3. The number of states is the product of the number of allowable values of all the factors.

Impossible States

Some combinations of factors may appear to be impossible. Say that the factors are:

GEAR	R, N, 1, 2, 3, 4	= 6 factors
DIRECTION	Forward, reverse, stopped	= 3 factors
ENGINE	Running, stopped	= 2 factors
TRANSMISSION	Okay, broken	= 2 factors
ENGINE	Okay, broken	= 2 factors
TOTAL		= 144 states

Because the states we deal with inside computers are not the states of the real world but rather a numerical representation of real-world states, the "impossible" states can occur.

Equivalent States

Two states are **equivalent** if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. This notion can also be extended to sets of states.

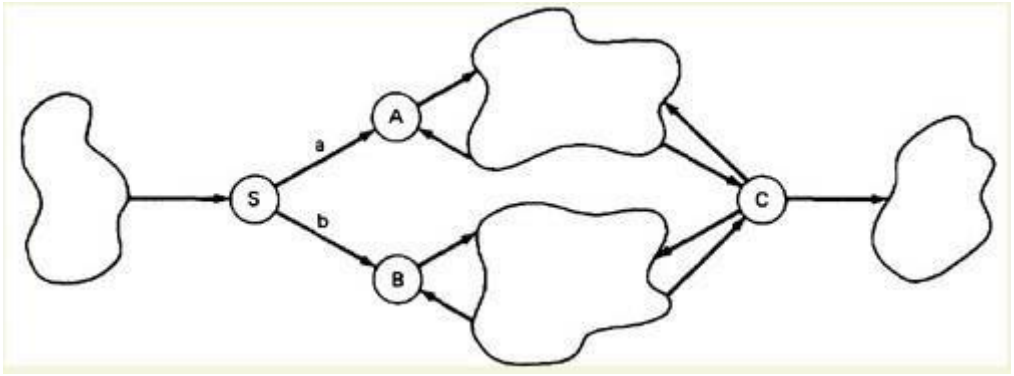


Figure: Equivalent States.

System is in state S and that an input of *a* causes a transition to state A while an input of *b* causes a transition to state B. The blobs indicate portions of the state graph whose details are unimportant.

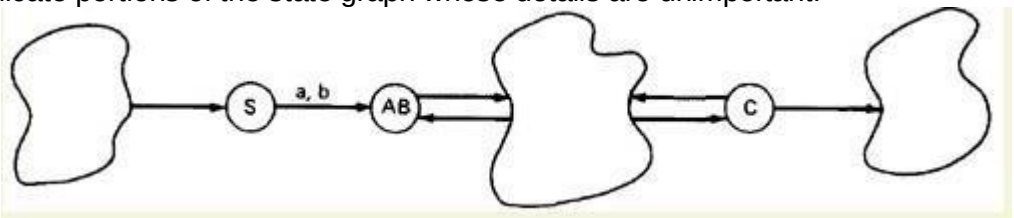


Figure: Equivalent States of above Figure Merged.

Equivalent states can be recognized by the following procedures:

1. The rows corresponding to the two states are identical with respect to input/output/next state but the name of the next state could differ.

2. There are two sets of rows which, except for the state names, have identical state graphs with respect to transitions and outputs. The two sets can be merged.

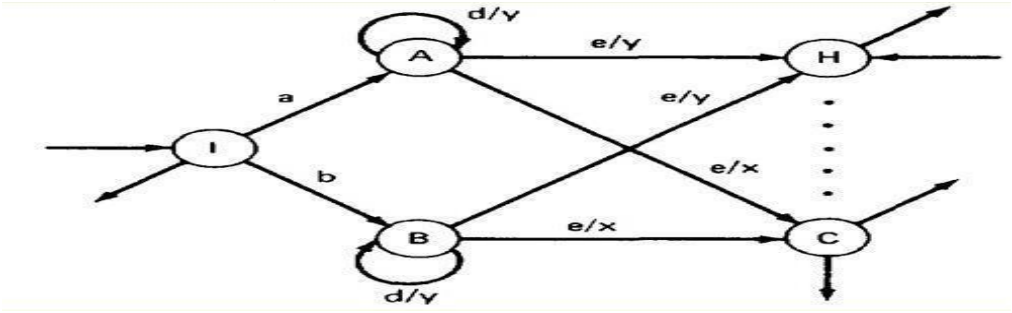


Figure: Equivalent States.

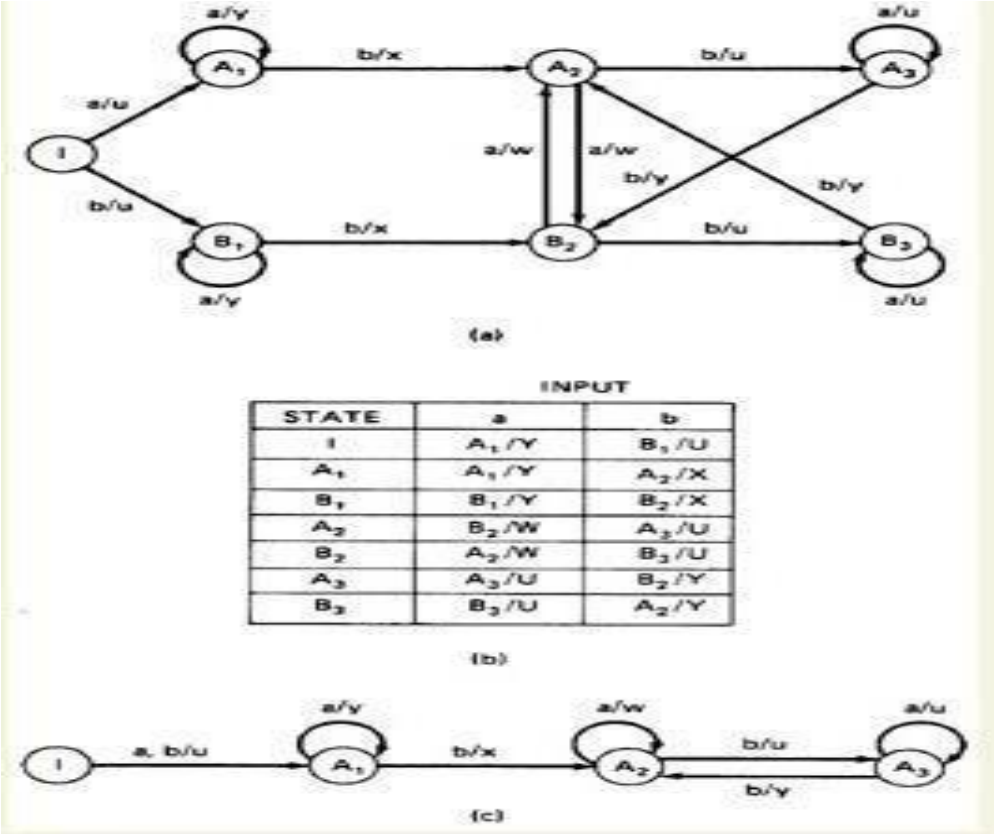


Figure: Merged Equivalent States.

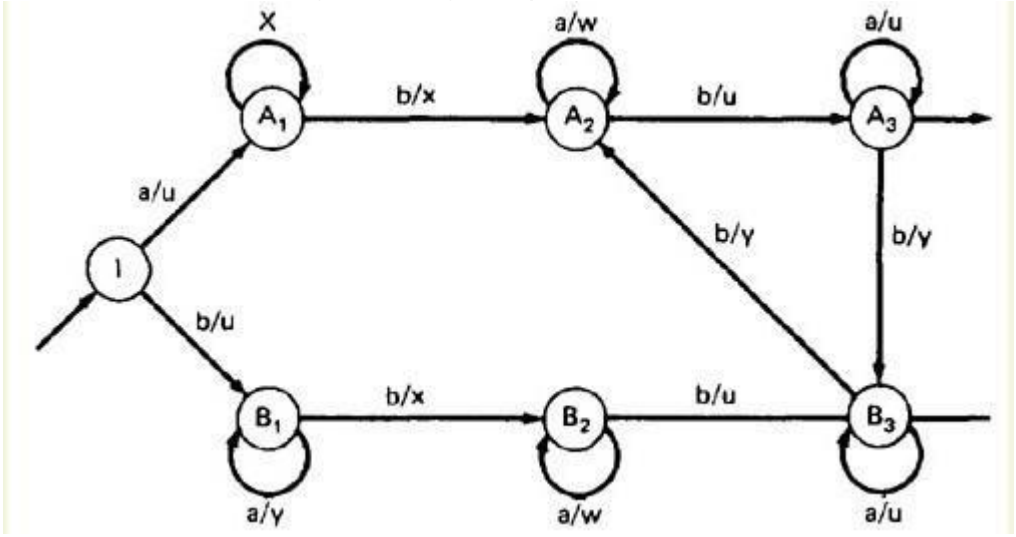


Figure: Un mergeable States.

Transition Bugs

- Every input-state combination must have a specified transition.
- If the transition is impossible, then there must be a mechanism that prevents that input from occurring in that state.
- *Exactly one transition must be specified for every combination of input and state.*
- A program can't have contradictions or ambiguities. Ambiguities are impossible because the program will do *something* (right or wrong) for every input. Even if the state does not change, by definition this is a transition to the same state. Similarly, software can't have contradictory transitions because computers can only do one thing at a time.

Rule 1: Theprogram willmaintainanerrorcounter,whichwillbeincrementedwhenever there'sanerror.

Rule 2: If there is an error, rewrite the block.

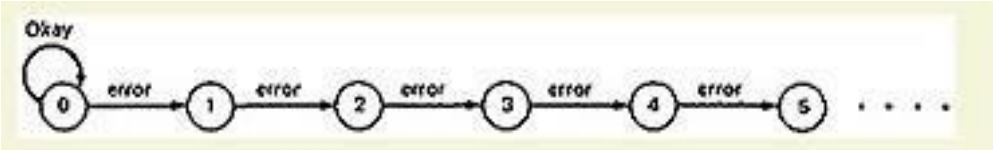
Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block.

Rule 4: Iftherehavebeenthreesuccessiveerasuresandanother erroroccurs,put theunit outofservice.

Rule 5: If the erasure was successful, return to the normal state and clear the error counter.

Rule 6: If the rewrite was unsuccessful, increment the error counter, advance the state, and try another rewrite.

Rule 7: If the rewrite was successful, decrement the error counter and return to the previous state.



STATE	OKAY	ERROR
0	0/none	1/
1		2/
2		3/
3		4/
4		5/
5		6/
6		7/
7		8/

Adding rule 2, we get

STATE	OKAY	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE
3		4/REWRITE
4		5/REWRITE
5		6/REWRITE
6		7/REWRITE
7		8/REWRITE

Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block.

INPUT		
STATE	**OKAY**	**ERROR**
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE, ERASE, REWRITE
3		4/REWRITE, ERASE, REWRITE
4		5/REWRITE, ERASE, REWRITE
5		6/REWRITE, ERASE, REWRITE
6		7/REWRITE, ERASE, REWRITE
7		8/REWRITE, ERASE, REWRITE

Rule 3, if followed blindly, causes an unnecessary rewrite. It's a minor bug, so let it go for now, but it pays to check such things. There might be an arcane security reason for rewriting, erasing, and then rewriting again.

Rule 4: If there have been three successive erasures and another error occurs, put the unit out of service.

	INPUT	
STATE	**OKAY**	**ERROR**
0	0/NONE	1/RW
1		2/RW
2		3/ER, RW
3		4/ER, RW
4		5/ER, RW
5		6/OUT
6		
7		

Rule 4 terminates our interest in this state graph so we can dispose of states beyond 6. The details of state 6 will not be covered by this specification; presumably there is a way to get back to state 0. Also, we can credit the specifier with enough intelligence not to have expected a useless rewrite and erase prior to going out of service.

Rule 5: If the erasure was successful, return to the normal state and clear the counter.

		INPUT
STATE	**OKAY**	**ERROR**
0	0/NONE	1/RW
1		2/RW
2		3/ER, RW
3	0/NONE	4/ER, RW
4	0/NONE	5/ER, RW
5	0/NONE	6/OUT
6		

Rule 6: If the rewrite was unsuccessful, increment the error counter, advance the state, and try another rewrite. Because the value of the error counter is the state, and because rules 1 and 2 specified the same action, there seems to be no point to rule 6 unless yet another rewrite was wanted. Furthermore, the order of the actions is wrong. If the state is advanced before the rewrite, we could end up in the wrong state. The proper order should have been: output = attempt-rewrite and then increment the error counter.

Rule 7: If the rewrite was successful, decrement the error counter and return to the previous state.

	INPUT	
STATE	**OKAY**	**ERROR**
0	0/NONE	1/RW
1	0/NONE	2/RW
2	1/NONE	3/ER, RW
3	0/NONE 2/NONE	4/ER, RW
4	0/NONE 3/NONE	5/ER, RW
5	0/NONE 4/NONE	6/OUT
6		

Rule 7 got ridofthe ambiguities but created contradictions. The specifier's intention was probably:

Rule 7A: If there have been no erasures and the rewrite is successful, return to the previous state.

Unreachable States

- An **unreachable state** is like unreachable code—a state that no input sequence can reach. An unreachable state is not impossible, just as unreachable code is not impossible.
- There are two possibilities: (1) There is a bug; that is, some transitions are missing. (2) The transitions are there, but you don't know about it;

Dead States

A **dead state**, (or set of dead states) is a state that once entered cannot be left.

Output Errors

The states, the transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could be incorrect. Output actions must be verified independently of states and transitions. That is, you should distinguish between a program whose state graph is correct but has the wrong output for a transition and one whose state graph is incorrect.

Encoding Bugs

- Encoding bugs for input coding, output coding, state codes, and state-symbol product formation could exist as such only in an explicit finite-state machine implementation.
- *The behavior of a finite-state machine is invariant under all encodings.*

STATE TESTING

Impact of Bugs

- A bug can manifest itself as one or more of the following symptoms:
  1. Wrong number ofstates.
  2. Wrong transition for a given state-input combination.
  3. Wrong output for a given transition.
  4. Pairs of states or sets of states that are inadvertently made equivalent (factor lost).
  5. States or sets of states that are split to create in equivalent duplicates.
  6. States or sets of states that have become dead.
  7. States or sets of states that have become unreachable.
- A path in a state graph is a succession of transitions caused by a sequence of inputs.
- The starting point of state testing is:

  1. Define a set of covering input sequences that get back to the initial state when starting from the initial state.

  2. For each step in each input sequence, define the expected next state, the expected transition, and the expected output code.
- A set of tests, then, consists of three sets of sequences:
  1. Input sequences.
  2. Corresponding transitions or next-state names.
  3. Output sequences.

TESTABILITY TIPS

Switches, Flags, and Unachievable Paths

- The functionality of switches, flags is referred to the same term state testing.
- Switches/flags are used as essential tool for state testing to cover or test the Finite State Machine in every possible state.

- A Switch/flag is set at initial process of Finite State Machine then this value is evaluated and tested.

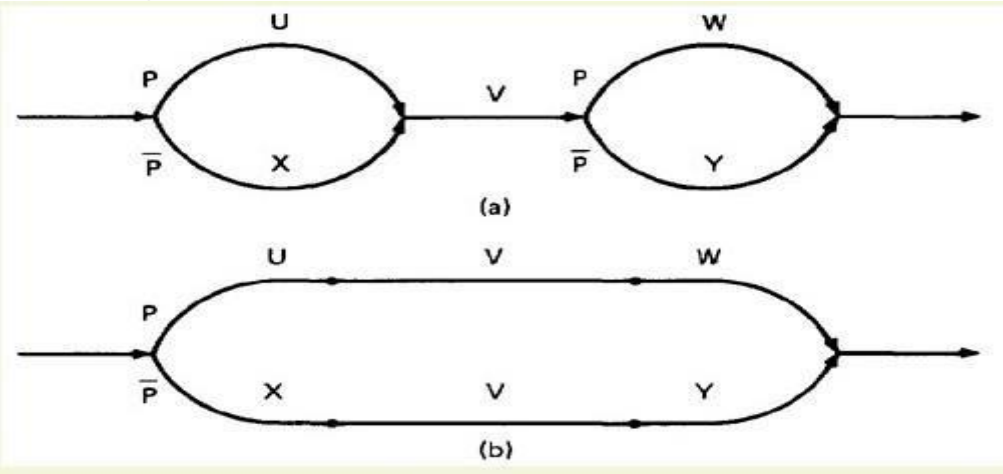


Figure: Program with One Switch Variable.

In Figure, the situation is more complicated. There are three switches this time. Again, where we go depends on the switch settings calculated earlier in the program. We can put the decision up front and branch directly, and again use subroutines to make each path explicit and do without the switches.

The advantages of this implementation are that if any of the combinations are not needed, we merely clip out that part of the decision tree:

Essential and Inessential Finite-State Behavior

- Program flags and switches are predicates deferred. There is a significant, qualitative difference between finite-state machines and combinational machines.
- A combinational machine selects paths based on the values of predicates, the predicates depend only on prior processing and the predicates' truth values will not change once they have been determined. Any path corresponds to a Boolean algebra expression over the predicates.
- Furthermore, it does not matter in which order the decisions are made. The fact that there is an ordering is a consequence of a sequential, Von Neumann computer architecture. In a parallel-data-flow machine, for example, the decisions and path selections could be made simultaneously. Sequence and finite-state behavior are in this case implementation consequences and not essential.
- The combinational machine has exactly one state and one transition back to itself for all possible inputs. The control logic of a combinational program can be described by a decision table or a decision tree.

Design Guidelines

1. Learn how it's done in hardware. I know of no books on finite-state machine design for programmers.
2. Start by designing the abstract machine. Verify that it is what you want to do.
3. Start with an explicit design—that is, input encoding, output encoding, state code assignment, transition table, output table, state storage, and how you intend to form the state-symbol product.
4. Before you start taking shortcuts, see if it really matters.
5. Take shortcuts by making things implicit only as you must to make significant reductions in time or space and only if you can show that such savings matter in the context of the whole system.
6. Consider a hierarchical design if you have more than a few dozen states.
7. Build, buy, or implement tools and languages that implement finite-state machines as software if you're doing more than a dozen states routinely.
8. Build in the means to initialize to any arbitrary state. Build in the transition verification instrumentation (the coverage analyzer). These are much easier to do with an explicit machine.

SOFTWARE TESTING

UNIT V

Graph Matrices and Application: Motivational Overview, Matrix of Graph, Relations, Power of a Matrix, Node Reduction Algorithm, Building Tools. (Student should be given an exposure to a tool like JMeter or Win-runner).

MOTIVATIONAL OVERVIEW:

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

The Problem with Pictorial Graphs

- Path tracing is not easy, and it's subject to error. You can miss a link here and there or cover some links twice.
- One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing.

The Basic Algorithms

The basic toolkit consists of:

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

MATRIX OF GRAPH

A graph matrix is a square array with one row and one column for every node in the graph. Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column.

Observe the following:

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

In general, an entry is not just a simple link name but a path expression corresponding to the paths between the pair of nodes.

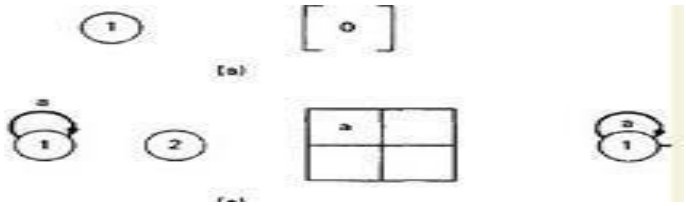


Figure: Some Graphs and Their Matrices.

A Simple Weight

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

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

A matrix with weights defined like this is called a **connection matrix**. The connection matrix is obtained by replacing each entry with I if there is a link and 0 if there isn't.

Further Notation

- To compact things, the entry corresponding to node *i* and column *j*, which is to say the link weights between nodes *i* and *j*, is denoted by a_{ij}.
- A self-loop about node *i* is denoted by a_{ii}, while the link weight for the link between nodes *j* and *i* is denoted by a_{ji}. The path segments expressed in terms of link names and, in this notation, for several paths in the graph

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

because

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

- The expression "$a_{ij}a_{jj}a_{jm}$" denotes a path from node i to j, with a self-loop at *j* and then a link from node *j* to node *m*. The expression "$a_{ij}a_{jk}a_{km}a_{mi}$" denotes a path from node *i* back to node *i* via nodes *j*, *k*, and *m*. An expression such as "$a_{ik}a_{km}a_{mj} + a_{in}a_{np}a_{pj}$" denotes a pair of paths between nodes *i* and *j*, one going via nodes *k* and *m* and the other via nodes *n* and *p*.
- This notation may seem cumbersome, but it's not intended for working with the matrix of a graph but for expressing operations on the matrix. It's a very compact notation. For example,

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

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

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

RELATIONS

- A **relation** is a property that exists between two (usually) objects of interest.
- If *a* and *b* denote objects and R is used to denote that a has the relation R to *b*:
  1. "Node *a is connected* to node *b*" or *a* R *b* where "R" means "is connected to."
  2. "*a* >= *b*" or *a* R *b* where "R" means "greater than or equal."
  3. "*a is a subset* of *b*" where the relation is "is a subset of."
- **Graph** consists of a set of abstract objects called **nodes** and a relation R between the nodes. If *a* R *b*, which is to say that a has the relation R to *b*, it is denoted by a **link** from *a* to *b*.
- In addition to the fact that the relation exists, for some relations we can associate one or more properties. These are called **link weights**.

- A **link weight** can be numerical, logical, illogical, objective, subjective, or whatever. Furthermore, there is no limit to the number and type of link weights that one may associate with a relation.
- "Is connected to" is just about the simplest relation there is: it is denoted by an unweighted link. Graphs defined over "is connected to" are called, as we said before, **connection matrices**.* For more general relations, the matrix is called a **relation matrix.**

Properties of Relations

1. **Transitive Relation**: A relation R is transitive if a R b and b R c implies a R c.
2. **Reflexive Relation**: A relation R is reflexive if, for every a, a R a. A reflexive relation is equivalent to a self-loop at every node.
3. **Symmetric Relation**: A relation R is symmetric if for every a and b, a R b implies b R a.
4. **Anti symmetric Relation:** A relation R is anti symmetric if for every *a* and *b*, if a R *b* and b R *a*, then *a* = *b*, or they are the same elements.

Equivalence Relations

An **equivalence relation** is a relation that satisfies the reflexive, transitive, and symmetric properties.

Partial Ordering Relations

- A **partial ordering relation** satisfies the reflexive, transitive, and anti symmetric properties.
- Partial ordered graphs have several important properties: they are loop-free, there is at least one maximum element, there is at least one minimum element, and if you reverse all the arrows, the resulting graph is also partly ordered.
- A **maximum element** *a* is one for which the relation *x*R*a* does not hold for any other element *x*.
- Similarly, a **minimum element** *a*, is one for which the relation *a*R*x* does not hold for any other element *x*.
- Trees are good examples of partial ordering.

THE POWERS OF A MATRIX

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

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

It states nothing more than the following:

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

Matrix Powers and Products

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

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

GiventwomatricesAandB, withentriesa_{ik}andb_{kj},respectively,theirproduct isanewmatrix C, whoseentriesare c_{ij}, where:

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

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

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

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

		a		
	d		b	
	c			f
	g	e		h

A

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

A^2

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

A^3

$A^2A = AA^2$
$A^4 = A^2A^2, (A^2)^2, A^3A, AA^3$

The Set of All Paths

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

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

This is an eloquent, but practically useless, expression. Let I be an *n* by *n* matrix, where *n* is the number of nodes. Let I's entriesconsistofmultiplicativeidentityelementsalongtheprincipaldiagonal.Forlinknames,thiscanbethenumber

"1." For other kinds of weights, it is the multiplicative identity for those weights. The above product can be re-phrased as:

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

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

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

Therefore, the original infinite sum can be replaced by

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

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

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

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

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

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

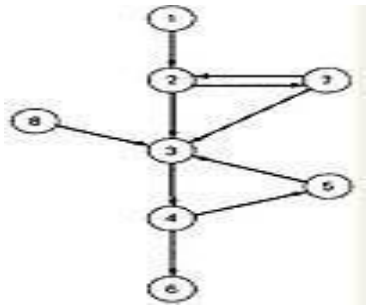
$$\sum_{i=1}^{15} A^i = A(A + I)^8(A + I)^4(A + I)^2$$

A matrix for which $A^2 = A$ is said to be **idempotent**. A matrix whose successive powers eventually yields an idempotent matrix is called an **idempotent generator**—that is, a matrix for which there is a k such that $A^{k+1} = A^k$.

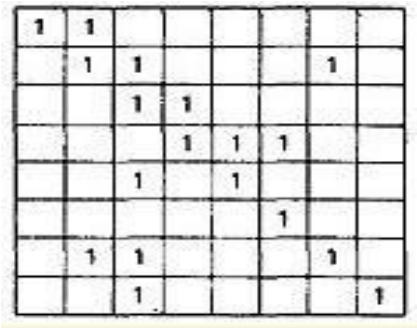
Partitioning Algorithm

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

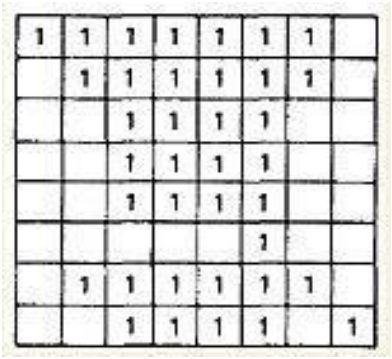
1. We might want to embed the loops within a subroutine so as to have a resulting graph which is loop-free at the top level.
2. Many graphs with loops are easy to analyze if you know where to break the loops.
3. While you and I can recognize loops, it's much harder to program a tool to do it unless you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



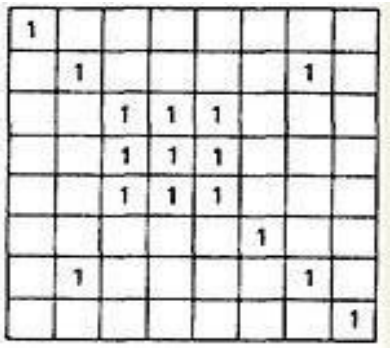
The relation matrix is



The transitive closure matrix is

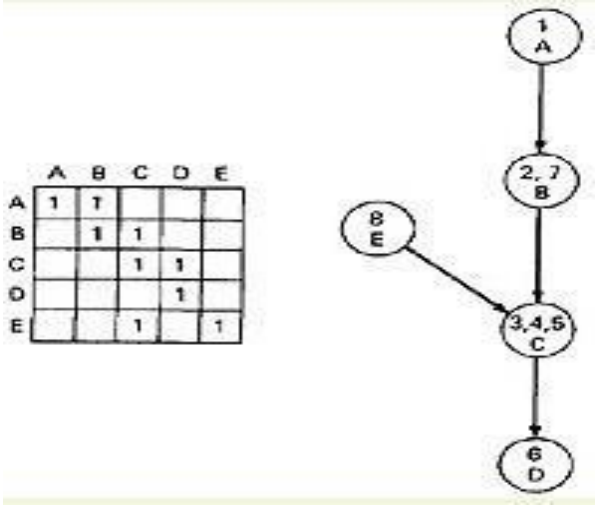


Intersection with its transpose yields



Youcanrecognizeequivalent nodes by simplypickingarow(orcolumn)andsearchingthematrix foridenticalrows. Mark the nodes that match the pattern as you go and eliminate that row. Then start again from the top with another row and another pattern. Eventually, all rows have been grouped. The algorithm leads to the following equivalent node sets:

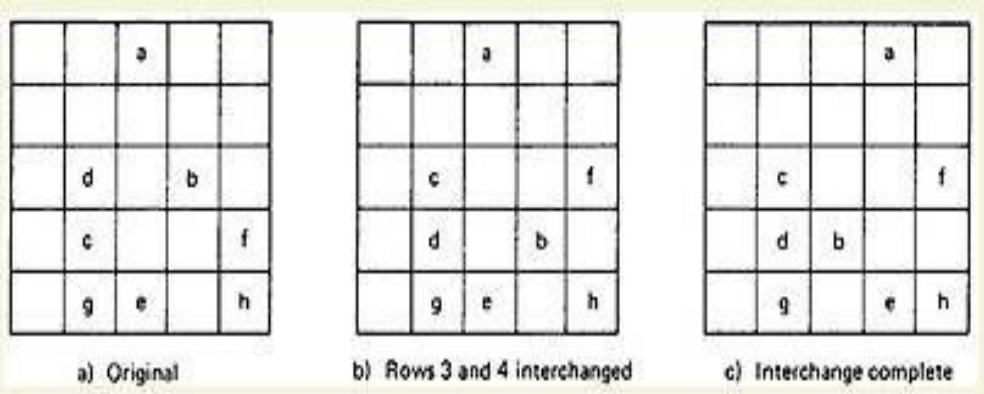
A = [1] B = [2,7] C = [3,4,5] D = [6] E = [8] whose graph is



NODE-REDUCTION ALGORITHM

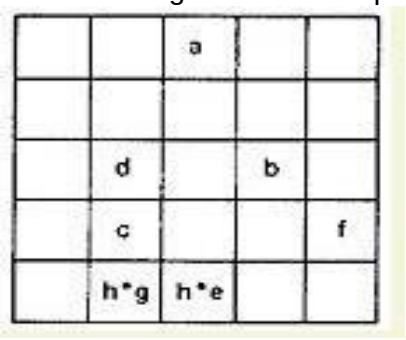
The advantage of the matrix-reduction method is that it is more methodical than the graphical method and does not entail continually redrawing the graph. It's done as follows:

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

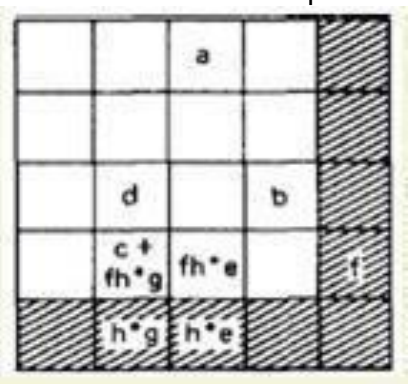


The Algorithm

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



The reduction is done one node at a time by combining the elements in the last column with the elements in the last row and putting the result into the entry at the corresponding intersection.
In the above case, the *f* in column 5 is first combined with *h*g* in column 2, and the result (*fh*g*) is added to the *c* term just above it. Similarly, the *f* is combined with *h*e* in column 3 and put into the 4,3 entry just above it.



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

		a
	d + bc + bfh*g	bfh*e

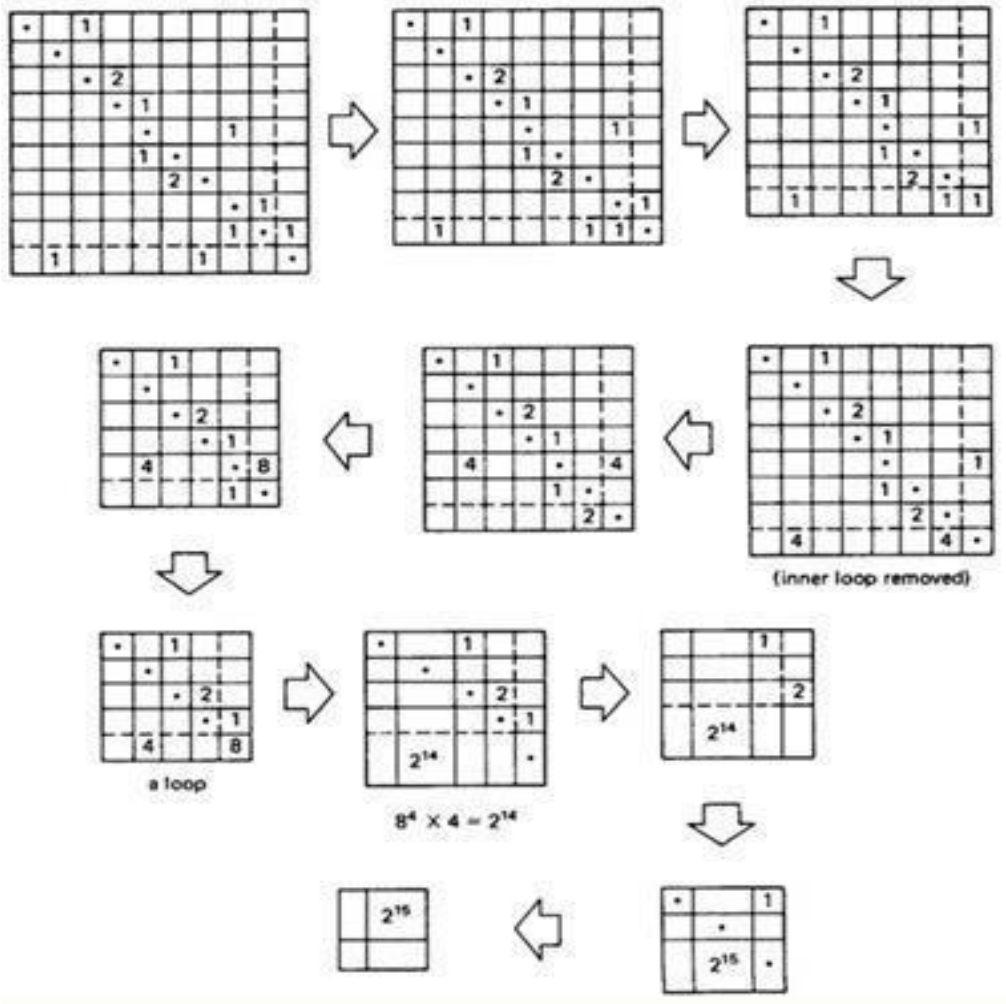
Removing the loop term yields

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

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

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

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



BUILDING TOOLS

Matrix Representation Software

We draw graphs or display them on screens as visual objects; we prove theorems and develop graph algorithms by using matrices;

Node Degree and Graph Density

The **out-degree** of a node is the number of out links it has.
The **in-degree** of a node is the number of in links it has.
The **degree** of a node is the sum of the out-degree and in-degree.

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

1. *Space*—Space grows as n^2 for the matrix representation, but for a linked list only as kn, where k is a small number such as 3 or 4.

2. *Weights*—Most weights are complicated and can have several components. That would require an additional weight matrix for each such weight.

3. *Variable-Length Weights*—If the weights are regular expressions, say, or algebraic expressions (which is what we need for a timing analyzer), then we need a two-dimensional string array, most of whose entries would be null.

4. *Processing Time*—Even though operations over null entries are fast, it still takes time to access such entries and discard them. The matrix representation forces us to spend a lot of time processing combinations of entries that we know will yield null results.

Linked-List Representation

Give every node a unique name or number. A link is a pair of node names.

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

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

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

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

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

UNIT 5 - 2 MARKS QUESTIONS WITH ANSWERS

1) Define a state.

A state is defined as: "A combination of circumstances or attributes belonging for the time being to a person or thing."

For example, a moving automobile whose engine is running can have the following states with respect to its transmission.

Reverse gear
Neutral gear
First gear
Second gear
Third gear
Fourth gear

2) Define State table.

Big state graphs are cluttered and hard to follow.

It‟s more convenient to represent the state graph as a table (the state table or State transition table) that specifies the states, the inputs, the transitions and the outputs.

The following conventions are used:

Each row of the table corresponds to a state.

Each column corresponds to an input condition.

The box at the intersection of a row and a column specifies the next state (the transition) and the output, if any.

3) Define Unreachable State.

An unreachable state is like unreachable code.

A state that no input sequence can reach.

An unreachable state is not impossible, just as unreachable code is not impossible

There may be transitions from unreachable state to other states; there usually because the State became unreachable as a result of incorrect transition.

There are two possibilities for unreachable states:

There is a bug; that is some transitions are missing.

The transitions are there, but you don‟t know about it.

4) when the two states are said to be equivalent states?

If every sequence of inputs from one state generates exactly the same sequence of outputs as the other states then these two states are known as equivalent states.

5) Advantages of state testing?

- It is useful when the error corrections are less expensive.
- It is specifically desinged for catching the deep bugs.
- It provides the biggest rewards during the design of the tests.

6) What is cyclomatic complexity?

Cyclomatic complexity is known as conditional complexity. It is a metric used for computing the complexity of source code and for counting the number of linearly independent paths that comprise the program. The general formula to compute cyclomatic complexity is

M=E-N+2P.

7) Define FSM?

A **finite-state machine** (**FSM**) or **finite-state automaton** (FSA, plural: **automata**), **finite automaton**, or simply a **state machine**, is a mathematical model of computation. .... An **FSM** is **defined** by a list of its states, its initial **state**, and the conditions for each transition.

UNIT-V

PART–A QUESTIONS(2 Marks)

1) Define a state?
2) Define State table? (Nov-2018 , June-2017)
3) Define Unreachable State? (June-2016)
4) When the two states are said to be equivalent states?
5) Advantages of state testing?
6) What is cyclomatic complexity? (Nov-2016)
7) Define FSM? (Nov-2018 , June-2018)

PART–B QUESTIONS(10 Marks)

1) Explain about State Graph of its Good and Bad with example?(Nov-2018 , June-2017)
2) Explain about Node Reduction Algorithm? (Nov-2017 , June-2018)
3) What are Graph matrices and explain their applications. (Nov-2018)