

Introduction to Software Testing

Chapter 2.7

Representing Graphs Algebraically

Paul Ammann & Jeff Offutt

www.introsoftwaretesting.com

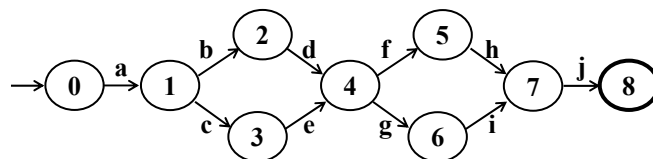
Graphs – Circles and Arrows

- It is usually easier for humans to understand graphs by viewing nodes as circles and edges as arrows
- Sometimes other forms are used to manipulate the graphs
 - Standard data structures methods are appropriate for tools
 - An algebraic representation (*regular expression*) allows certain useful mathematical operations to be performed

Representing Graphs Algebraically

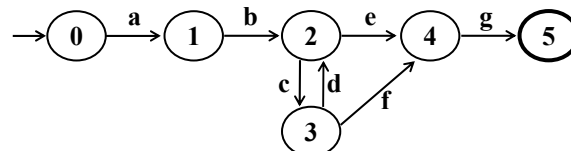
- Assign a unique label to each edge
 - Could represent semantic information, such as from an activity graph or an FSM
 - Could be arbitrary – we use lower case letters as an abstraction
- Operators:
 - Concatenation (multiplicative) : if edge a is followed by edge b , they are concatenated as “ $a * b$ ”, or more usually, “ ab ”
 - Selection (additive) : if either edge a or b can be taken, they are summed as “ $a + b$ ”
- Path Product : A sequence of edges “multiplied” together is called a product of edges, or a path product
- Path Expression : Path products and possibly ‘+’ operators
 - $A = ab + cd$
- Loops : represented as exponents

Examples



Double-diamond graph

Path Expression = $abdfhj + abdgij + acegij + acefhj$



Path Products : $A = abeg$, $B = (cd)^*$, $C = cf$

Path Expression = $ab (cd)^* (e + cf) g$

Let's Look at the Math

- **Path Products**
 - **Commutative** : Path product is **not** ($AB \neq BA$)
 - **Associative** : Path product **is** ($A(BC) = (AB)C = ABC$)
- **Path Summation**
 - **Commutative** : Summation is ($A + B = B + A$)
 - **Associative** : ($(A+B)+C = A+(B+C) = A + B + C$)
- **These are close to the “usual” arithmetic addition and multiplication, so most of the standard algebraic laws can be used**
 - In normal math, multiplication is commutative
- **(Don't worry ... we'll get back to testing soon ...)**

Algebraic Laws on Graph Expressions

- **Distributive** : $A (B + C) = AB + AC$
- **Distributive** : $(B + C) D = BD + CD$
- **Absorption rule** : $A + A = A$
- **Shortcut notation for loops**
 - At least one iteration : $A^+ = AA^*$
 - Bounds on iteration : $A^3 = A^0 + A^1 + A^2 + A^3$
 - More generally : $A^n = A^0 + A^1 + \dots + A^n$
- **Absorbing exponents**
 - $A^n + A^m = A^{\max(n,m)}$
 - $A^n A^m = A^{n+m}$
 - $A^n A^* = A^* A^n = A^*$
 - $A^n A^+ = A^+ A^n = A^+$
 - $A^* A^+ = A^+ A^* = A^+$

Identity Operators

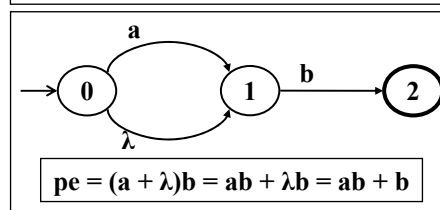
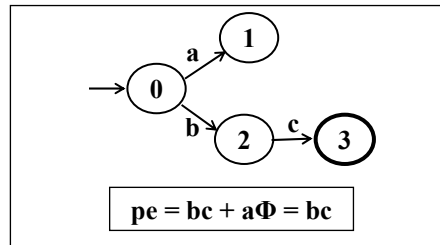
- **Multiplicative** : λ (an empty path)
- **Additive** : Φ (a null path – set of paths that has no paths)

- **Multiplicative identity laws**

- $\lambda + \lambda = \lambda$
- $\lambda A = A\lambda = A$
- $\lambda^n = \lambda \lambda = \lambda^* = \lambda^+ = \lambda$
- $\lambda^+ + \lambda = \lambda^* = \lambda$

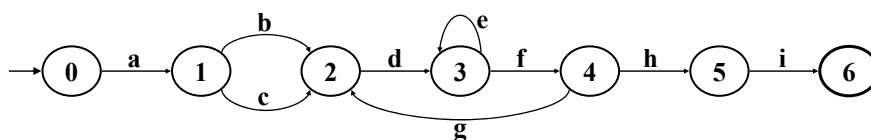
- **Additive identity laws**

- $A + \Phi = \Phi + A = A$
- $A\Phi = \Phi A = \Phi$
- $\Phi^* = \lambda + \Phi + \Phi^2 + \dots = \lambda$



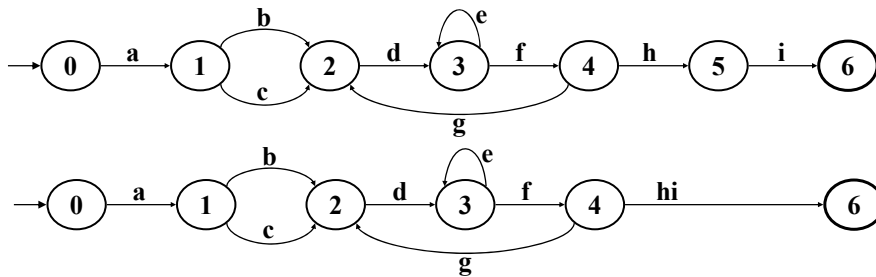
Graphs to Path Expressions

- **General algorithms for reducing finite state machines to regular expressions are widely known**
- **This lecture presents a four step, iterative, special case process**
- **The four steps are illustrated on a simple graph**



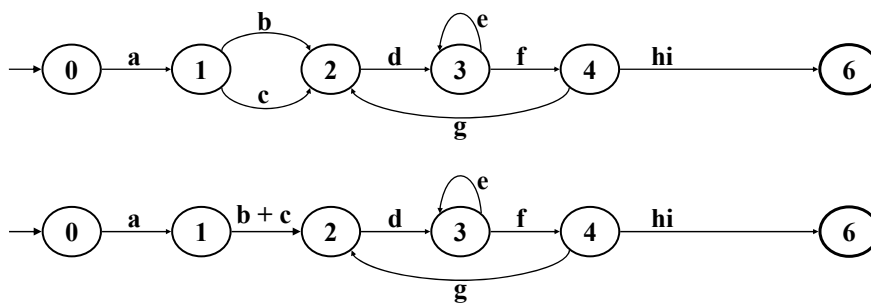
Step 1 : Sequential Edges

- Combine all sequential edges
- Multiply the edge labels
- **Precisely:** For any node that has only one incoming and one outgoing edge, eliminate the node, combine the two edges, and multiply their path expressions
- **Example:** Combine edges h and i



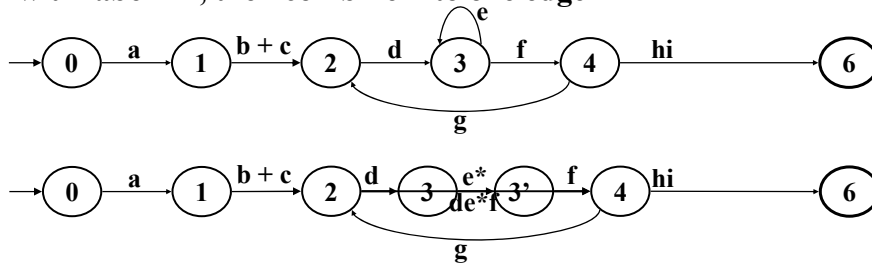
Step 2 : Parallel Edges

- Combine all parallel edges
- Add the edge labels
- **Precisely:** For any pair of edges with the same source and target nodes, combine the edges and add their path expressions
- **Example :** Combine edges b and c



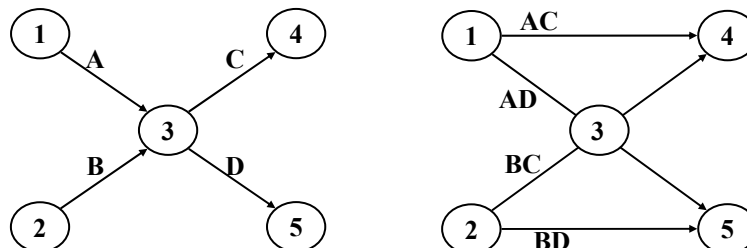
Step 3 : Self-Loops

- Combine all self-loops (loops from a node to itself)
- Add a new “dummy” node
- An incoming edge with exponent
- Merge the three resulting sequential nodes with multiplication
- **Precisely:** For any node $n1$ with a self-loop X , incoming edge A and outgoing edge B , remove X , add a new node $n1'$ and edge with label X^* , then combine into one edge AX^*B



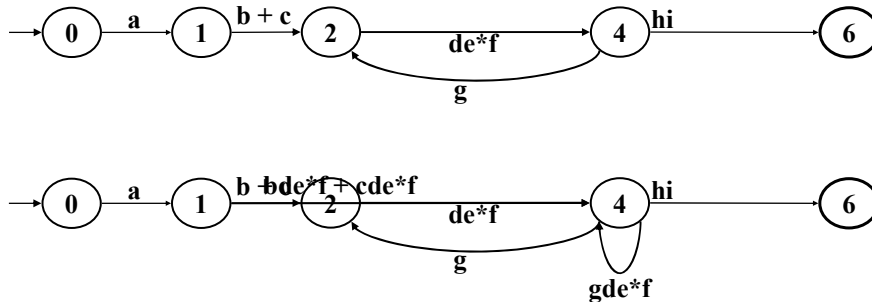
Step 4 : Remove Tester-Chosen Nodes

- Use for “other” special cases, when steps 1-3 do not apply
- Choose a node that is not initial or final
- Replace it by inserting edges from all predecessors to all successors
- Multiply path expressions from all incoming with all outgoing edges



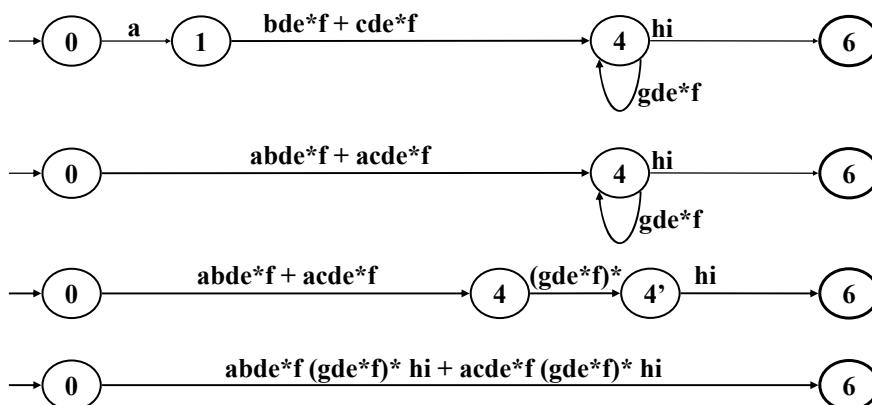
Step 4 : Eliminate Node 2

- Remove node 2
- Edges (1, 2) and (2, 4) become one edge
- Edges (4, 2) and (2, 4) become a self-loop



Repeat Steps 1 Through 4

Continue until only one edge is left ...



Applications of Path Expressions

1. Deriving test inputs
2. Counting paths in a flow graph
3. Minimum number of paths to satisfy All Edges
4. Complementary operations analysis

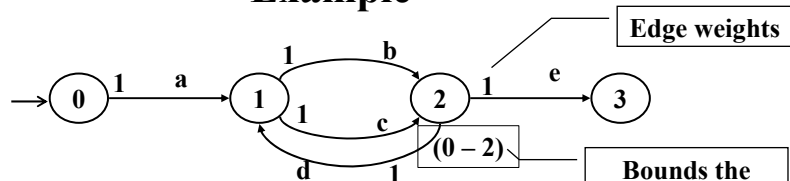
1. Deriving Test Inputs

- Very simple ... find a graph and cover it
- Cover regular expressions by covering each separate path product
- This is a form of the specified path coverage criterion
- Loops are represented by exponents – if an unbounded exponent appears, replace with a constant value based on domain knowledge of the program
- Test requirements for running example:
 - Final path expression: $abde^*f(gde^*f)^*hi + acde^*f(gde^*f)^*hi$
 - Test Requirements
 - $abde^5f(gde^5f)^5hi$
 - $acde^5f(gde^5f)^5hi$

2. Counting Paths in a Flow Graph

- It is sometimes useful to know the number of paths in a graph
- The path expressions allow this computation with straightforward arithmetic
- Cycles mean we have an infinite number of paths, so we have to make assumptions to approximate
- Put a reasonable bound on the number of iterations by replacing the '*' with an integer value
 - The bound may be a true maximum number of iterations
 - The bound may represent a tester's assumption that executing the loop 'N times' is enough

2. Counting Paths in a Flow Graph Example



$$\begin{aligned}
 pe &= a (b + c) (d (b + c))^2 e \\
 &= 1 * (1 + 1) * (1 * (1 + 1))^2 * 1 \\
 &= 1 * 2 * 2^2 * 1 \\
 &= 2 * (\sum_{i=0}^2 2^i) * 1 \\
 &= 2 * (2^0 + 2^1 + 2^2) * 1 \\
 &= 2 * (1 + 2 + 4) * 1 \\
 &= 2 * 7 * 1 \\
 &= 14
 \end{aligned}$$

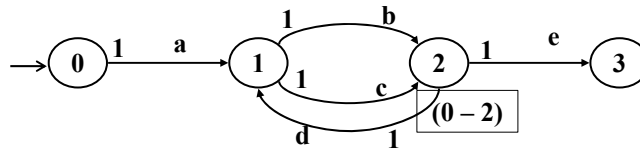
2. Counting Costs of Executing Paths Edge Weights

- It sometimes helps to have the number of paths include a measure of cost for executing each path
 - Number of paths in a function call
 - Expensive operations
 - An operation our intuition tells the tester to avoid or encourage
- Edge weights default to 1 – otherwise marked by tester
- Cycle weight: the maximum number of iterations allowed
 - Only mark one edge per cycle !
- Compute path expression and substitute weights for edges
 - Path expression : $A+B$ becomes $weight(A) + weight(B)$
 - Path product : AB becomes $weight(A) * weight(B)$
 - Loop is sum of the weight of all iterations
- The result of this computation is the estimated total cost of executing all paths

3. Minimum Number of Paths to Satisfy All Edges

- If we wish to satisfy All Edges, how many paths are needed?
- How many tests will be needed?
- Similar to computing the maximum number of paths
 - Different computation
- Specifically:
 - Path expression : $A+B$ becomes $weight(A) + weight(B)$
 - Path product : AB becomes $\max(weight(A), weight(B))$
 - Loop : A^n is either 1 or $weight(A)$
 - Judgment of tester – if all paths in the loop can be taken in one test, use 1, else use $weight(A)$

3. Min Number of Paths to Satisfy All Edges Example



$$pe = a (b + c) (d (b + c))^2 e$$

Conservatively assume the same edge from 1 to 2 must be taken every iteration through the loop –use the edge weight ...

$$= 1 * 2 * (1 * (2))^2 * 1$$

$$= 1 * 2 * (1 * 2) * 1$$

$$= \max (1, 2, 1, 2, 1)$$

$$= 2$$

4. Complementary Operations Analysis

- A method for finding potential anomalies
 - Def-use anomalies (use before a def)
 - FileADT example (closing before writing)
- A pair of operations are complementary if their behaviors negate each other, or one must be done before the other
 - push & pop
 - enqueue & dequeue
 - getting memory & disposing of memory
 - open & close
- Edge weights are replaced with one of three labels
 - C – Creator operation
 - D – Destructor operation
 - I – Neither

4. Complementary Operations Analysis Arithmetic Operations

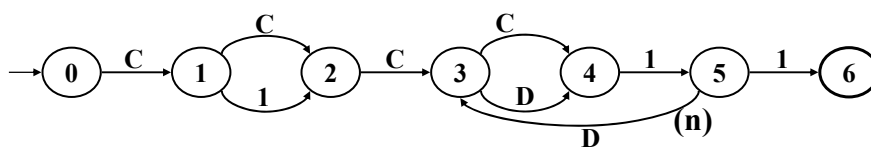
- The addition and multiplication operators are replaced with two tables

*	C	D	1
C	C ²	1	C
D	DC	D ²	D
1	C	D	1

+	C	D	1
C	C	C+D	C+1
D	D+C	D	D+1
1	C+1	D+1	1

- Not the same as usual algebra
 - $C * D = 1$, $C + C = C$, $D + D = D$
 - Multiplication is not commutative because the operations are not equivalent ... a destructor cancels a creator, but not vice versa

4. Complementary Operations Analysis Example



$$pe = C (C+1) C (C+D) 1 (D (C+D) 1)^n 1$$

$$\text{reduced } pe = (CCCC + \cancel{CCCD} + \cancel{CCC} + \cancel{CCD}) (DC + DD)^n$$

$C * D = 1$,
canceling out

$$\text{final } pe = (CCCC + CC + CCC + C) (DC + DD)^n$$

4. Using Complementary Operations Analysis

$$\text{final pe} = (\text{CCCC} + \text{CC} + \text{CCC} + \text{C}) (\text{DC} + \text{DD})^n$$

- **Ask questions:**

- Can we have more destructors than creators?

- CCCD (DD)ⁿ, n > 1
- CCCD (DD)ⁿ, n > 0
- CCC (DDDCDD)

- Can we have more creators than destructors?

- CCCC
- CCD (DC)ⁿ, for all n

- Each “yes” response represents a specification for a test that **might** cause anomalous behavior

Summary of Path Expressions

- **Having an algebraic representation of a graph can be very useful**
- **Most techniques involve some human input and subjectivity**
- **The techniques have not been sufficiently quantified**
- **We know of no commercial tools to support this analysis**