Class 18

- Questions/comments
- Discussion of academic honesty, GT Honor Code
- Efficient path profiling
- Final project presentations: Dec 1, 3; 4:35-6:45
- Assign (see Schedule for links)
  - Problem Set 7 discuss
  - Readings

Execution Tracing and Profiling

- Gathering dynamic information about programs
  - Execution coverage
  - Execution profiling
  - Execution tracing

Execution Tracing and Profiling

- Gathering dynamic information about programs
  - Execution coverage
  - Execution profiling
  - Execution tracing
- Three main alternatives
  - Debugging interfaces
  - Customized runtime systems
  - Instrumentation
    - Post-processing
    - Online processing
    - Preprocessing

Debugging Interfaces

- **Debugging Interfaces** provide hooks into the runtime system that allow for collecting various dynamic information while the program executes.
- Examples:
  - Java Platform Debugger Architecture (JPDA)
    - JVM Debugging Interface (JVMDI)
    - JVM Profiling Interface (JVMPI)
    - Java Virtual Machine Tool Interface (JVMTI) [New]
  - Valgrind
  - DynamoRIO
  - Emulators for embedded systems

Customized Runtime Systems

- Customized Runtime Systems are runtime systems modified to collect some specific dynamic information.
- Examples:
  - Jalapeño JVM

Instrumentation Tools

- Source-level
  - EDG parser (AST)
  - Customized gcc
- Binary/bytecode level
  - Vulcan
  - BCEL
  - SOOT
- Dynamic
  - Dyninst
  - PIN
  - Valgrind

Efficient Path Profiling

Profiling (recap)

- Program profiling counts occurrences of an event during a program's execution
  - Basic blocks
  - Control-flow edges
  - Acyclic path
- Application
  - Performance tuning
  - Profile-directed compilation
  - Test coverage

Goal

- Goal of paper: To efficiently collect path profiles for a DAG (i.e., acyclic-path profiling)

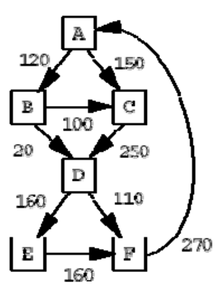
- Why not use existing techniques (existing at the time that the paper was written)?

State of the Art

- Edge profiling: 16% overhead
- Estimation of path profiles from edge profiles

  Correctly estimated only 38% of paths in SPEC benchmarks)

Explain this figure



Path	Prof1	Prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20

Acyclic-Path Profiling

- Assume for now—all paths acyclic—no loops
- Subsumes
  - basic block/statement profiling
  - edge/branch profiling
- Better approximation of intra-procedural path frequencies
- Stronger coverage criterion for white-box testing

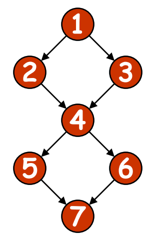
Goals for Instrumentation

- Low time and space overhead
- Minimal number of probes
- Optimal placement of the probes
- Paths represented with a simple integer value
- Compact numbering of paths

Algorithm Overview (i)

- Each potential path is represented as a state
- Upon entry all paths are possible
- Each branch taken narrows the set of possible final states
- State reached at the end of the procedure represents the path taken

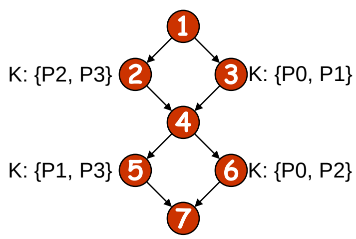
- Example:
  - P0: 1, 2, 4, 5, 7
  - P1: 1, 2, 4, 6, 7
  - P2: 1, 3, 4, 5, 7
  - P3: 1, 3, 4, 6, 7



Algorithm Overview (i)

- Each potential path is represented as a state
- Upon entry all paths are possible
- Each branch taken narrows the set of possible final states
- State reached at the end of the procedure represents the path taken

- Example:
  - P0: 1, 2, 4, 5, 7
  - P1: 1, 2, 4, 6, 7
  - P2: 1, 3, 4, 5, 7
  - P3: 1, 3, 4, 6, 7



Algorithm Overview (i)

- Each potential path is represented as a state
- Upon entry all paths are possible
- Each branch taken narrows the set of possible final states
- State reached at the end of the procedure represents the path taken

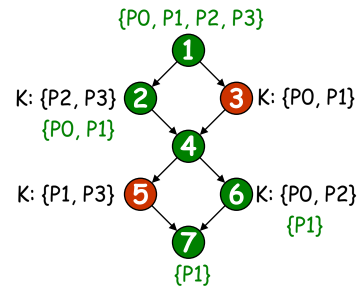
- Example:

  P0: 1, 2, 4, 5, 7

  P1: 1, 2, 4, 6, 7

  P2: 1, 3, 4, 5, 7

  P3: 1, 3, 4, 6, 7



Algorithm Overview (ii)

- Final "states" (i.e., paths) are represented by integers in [0, n-1] (n == number of paths)
- Instrumentation not at every branch
- Transitions computed by simple arithmetic operations (no tables)
- CFG transformed in acyclic CFGs (DAGs)

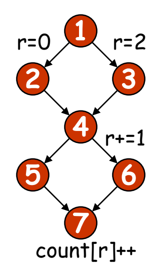
- Example:

  P0: 1, 2, 4, 5, 7

  P1: 1, 2, 4, 6, 7

  P2: 1, 3, 4, 5, 7

  P3: 1, 3, 4, 6, 7



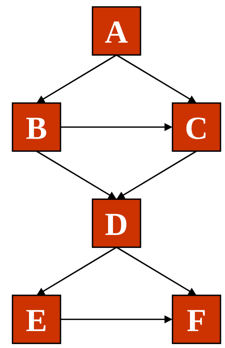
Algorithm Steps

1. Assign integer values to edges such that no two paths compute the same path sum
2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge
3. Select appropriate instrumentation
4. Derive the executed paths from the collected run-time profiles

Algorithm (Step 1 of 4)

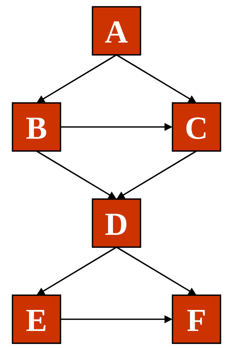
1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```



Topological, Reverse Topological Order

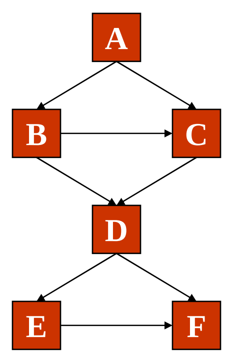
- Create a depth-first spanning tree
- Find topological and reverse topological orders
- Are there other orders based on other spanning trees?



Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

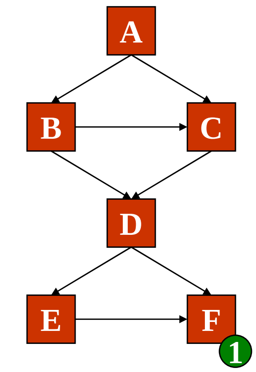
```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

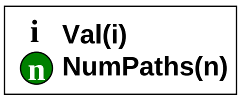


Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

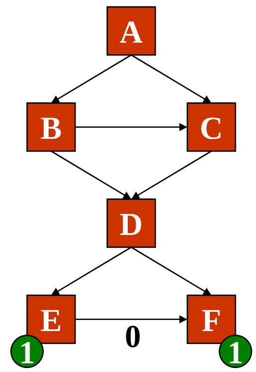


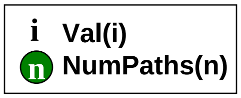


Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

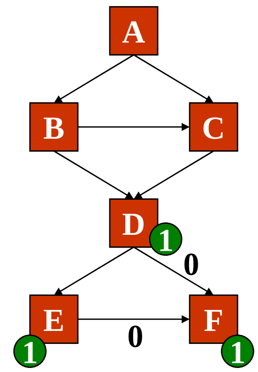





Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

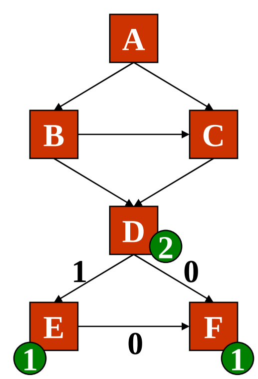


i Val(i)
n NumPaths(n)

Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

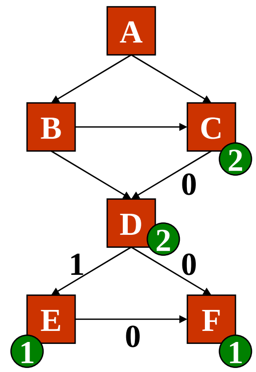


i Val(i)
n NumPaths(n)

Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

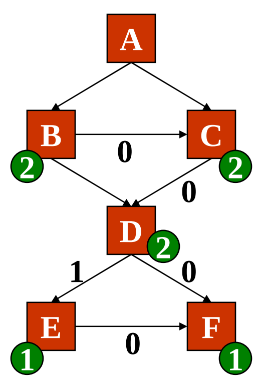


i Val(i)
n NumPaths(n)

Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

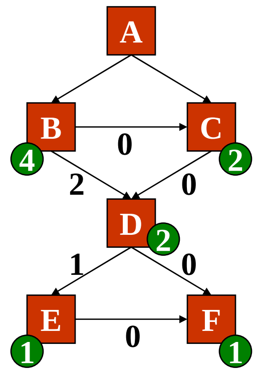


i Val(i)
n NumPaths(n)

Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

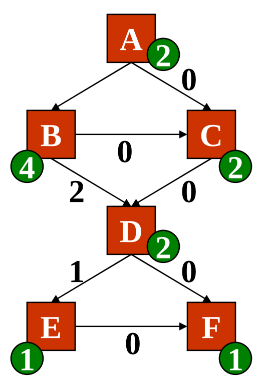


i Val(i)
n NumPaths(n)

Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

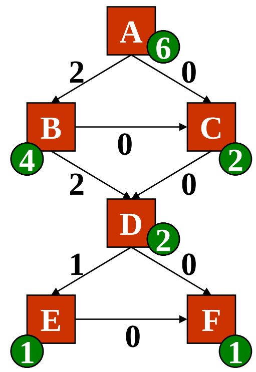


i Val(i)
n NumPaths(n)

Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```

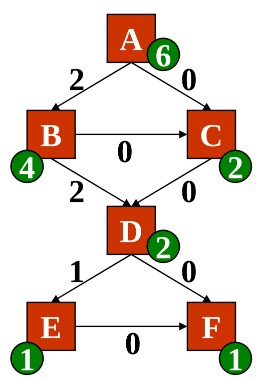


i Val(i)
n NumPaths(n)

Algorithm (Step 1 of 4)

1. Assign to each edge *e* a value *Val(e)* such that the sum along a path is unique and [0,n-1]

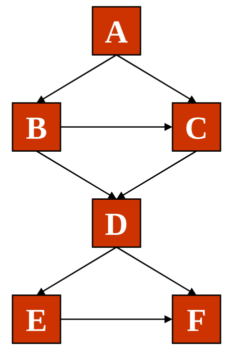
```
for each vertex v in rev. top. order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) += NumPaths(w);
    }
  }
}
```



Not necessarily the best placement

Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.



Begin
Side Discussion of Probe Placement

(

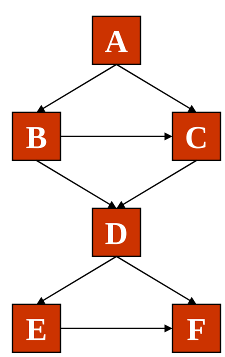
Probe placement

- Knuth published efficient algorithms for finding the minimum number of edge counters for edge profiling
- Algorithm
  - Compute spanning tree *T* of CFG; edges in the spanning tree are bidirectional
  - Chords of spanning tree are edges *E* in CFG minus edges in *T*
  - Instrumenting only the chords is sufficient to deduce execution of remaining edges

Example

Show several spanning trees for the graph, and determine where probes should be placed

(complete on board)



Optimal Placement of Probes

- Several spanning trees may be possible on a CFG
- A maximum spanning tree is a spanning tree of maximum weight on its edges (why do we want maximum spanning tree?)
- Weight is defined as the execution frequency of the edge (how can this be done?)

Edge Execution Frequency

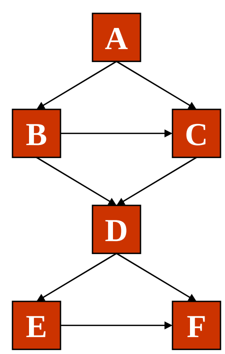
- Program can be profiled to gather edge execution frequency
- Edge execution frequency can be approximated statically—static approximation heuristic [Ball and Larus 94]
- Generally impractical for purposes of profiling paths (why?)

End
Side Discussion of Probe Placement

)

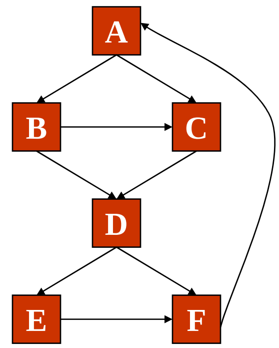
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.



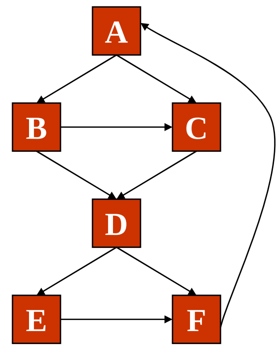
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.
   - Add edge EXIT -> ENTRY



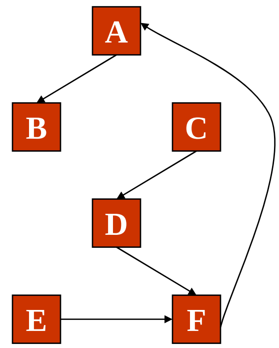
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.
   - Add edge EXIT -> ENTRY
   - Compute a maximal spanning tree (find chords)



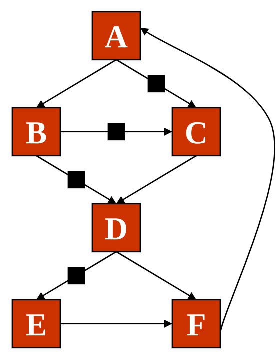
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.
   - Add edge EXIT -> ENTRY
   - Compute a maximal spanning tree (find chords)



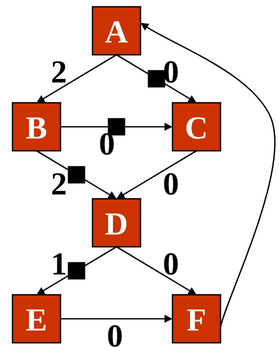
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.
   - Add edge EXIT -> ENTRY
   - Compute a maximal spanning tree (find chords)



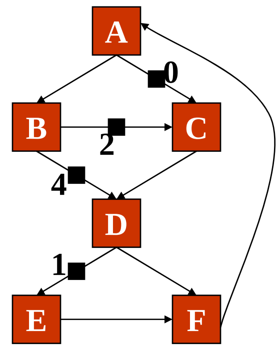
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.
   - Add edge EXIT -> ENTRY
   - Compute a maximal spanning tree (find chords)
   - Assign increments: start from Val(e) and “propagate” to chord [Ball and Larus 94]



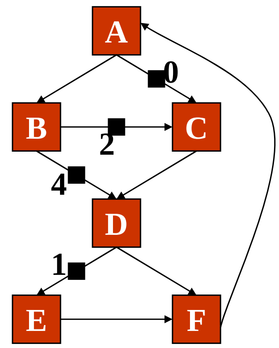
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.
   - Add edge EXIT -> ENTRY
   - Compute a maximal spanning tree (find chords)
   - Assign increments: start from Val(e) and “propagate” to chord [Ball and Larus 94]



Algorithm (Step 3 of 4)

3. Select appropriate instrumentation
   - Initialize path register (r=0)
   - Update r in chords (r += inc)
   - Increment path's counter at EXIT (count[r]++)

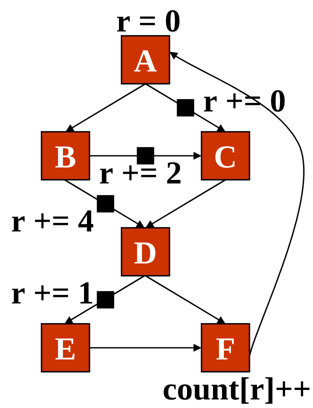


Algorithm (Step 3 of 4)

3. Select appropriate instrumentation
   - Initialize path register (r=0)
   - Update r in chords (r += inc)
   - Increment path's counter at EXIT (count[r]++)

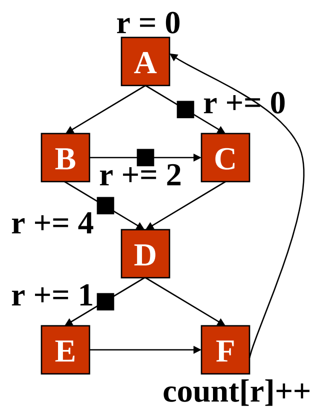
Algorithm (Step 3 of 4)

3. Select appropriate instrumentation
   - Initialize path register (r=0)
   - Update r in chords (r += inc)
   - Increment path's counter at EXIT (count[r]++)



Algorithm (Step 3 of 4)

3. Select appropriate instrumentation
   - Initialize path register (r=0)
   - Update r in chords (r += inc)
   - Increment path's counter at EXIT (count[r]++)
   - Optimize
     - Initializations (first chord on paths)
     - Path's counter increment (last chord on paths)

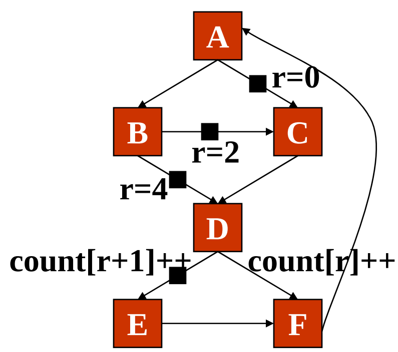


Algorithm (Step 3 of 4)

3. Select appropriate instrumentation
   - Initialize path register (r=0)
   - Update r in chords (r += inc)
   - Increment path's counter at EXIT (count[r]++)
   - Optimize
     - Initializations (first chord on paths)
     - Path's counter increment (last chord on paths)

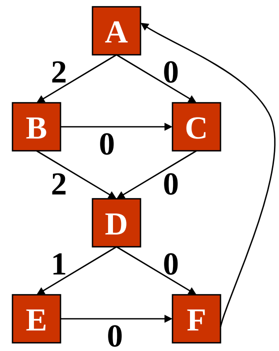
Algorithm (Step 3 of 4)

3. Select appropriate instrumentation
   - Initialize path register (r=0)
   - Update r in chords (r += inc)
   - Increment path's counter at EXIT (count[r]++)
   - Optimize
     - Initializations (first chord on paths)
     - Path's counter increment (last chord on paths)



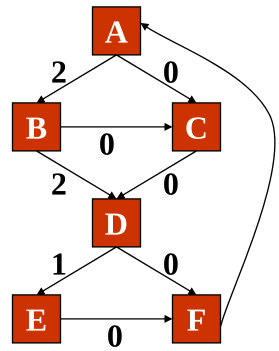
Algorithm (Step 4 of 4)

4. Regenerating a path after collecting a profile
   - Start at ENTRY
   - Let r be the path value
   - Select which edge to follow by finding the edge with the largest value Val(e) <= r
   - Traverse edge e and r = r – Val(e)



Algorithm (Step 4 of 4)

4. Regenerating a path after collecting a profile
   - Start at ENTRY
   - Let r be the path value
   - Select which edge to follow by finding the edge with the largest value Val(e) <= r
   - Traverse edge e and r = r – Val(e)



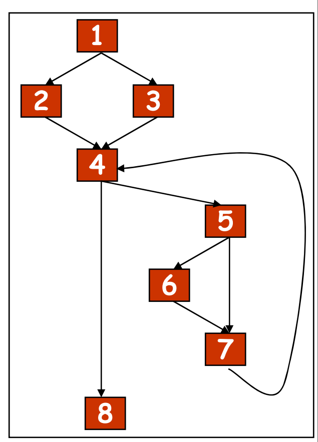
Generate path for 5
Generate path for 3

Acyclic Paths

- All paths are intra-procedural (later extension to interprocedural)
- No cycles (to avoid infinite number of paths)
- Different kinds of loops and representations of loops (we'll see in what follows)

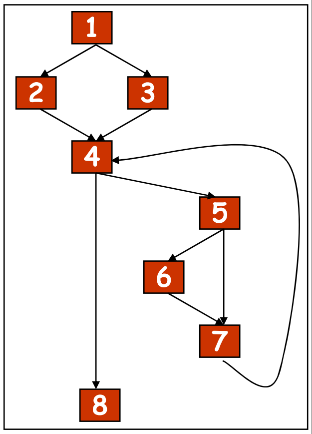
Arbitrary Control Flow (loops)

- Loop implies the presence of a back-edge
- Back-edges instrumented to increment path counter and reinitialize path register (count[r]++; r=0)



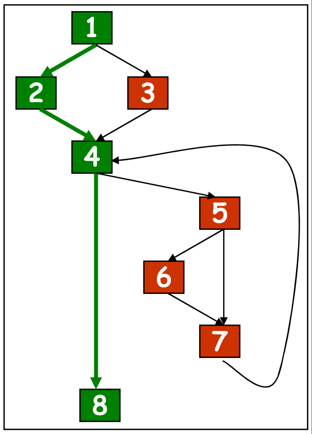
Arbitrary Control Flow (loops)

- Loop implies the presence of a back-edge
- Back-edges instrumented to increment path counter and reinitialize path register (count[r]++; r=0)
- This is not enough; with loops, 4 types of paths (v->w and x->y are back-edges)
  - ENTRY to EXIT
  - ENTRY to v (ending with execution of v->w)
  - w to x (after executing v->w and ending with the execution of x->y, v->w and x->y can be the same back-edge)
  - w to EXIT (after executing v->w)
- Need to distinguish them



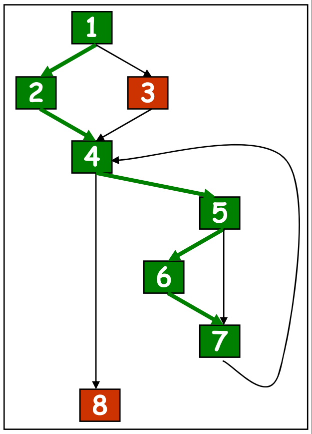
Arbitrary Control Flow (loops)

- Loop implies the presence of a back-edge
- Back-edges instrumented to increment path counter and reinitialize path register (count[r]++; r=0)
- This is not enough; with loops, 4 types of paths (v->w and x->y are back-edges)
  - <u>ENTRY to EXIT</u>
  - ENTRY to v (ending with execution of v->w)
  - w to x (after executing v->w and ending with the execution of x->y, v->w and x->y can be the same back-edge)
  - w to EXIT (after executing v->w)
- Need to distinguish them



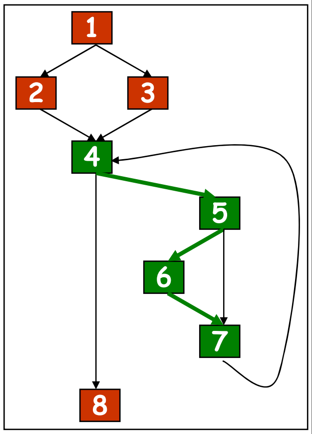
Arbitrary Control Flow (loops)

- Loop implies the presence of a back-edge
- Back-edges instrumented to increment path counter and reinitialize path register (count[r]++; r=0)
- This is not enough; with loops, 4 types of paths (v->w and x->y are back-edges)
  - ENTRY to EXIT
  - ENTRY to v (ending with execution of v->w)
  - w to x (after executing v->w and ending with the execution of x->y, v->w and x->y can be the same back-edge)
  - w to EXIT (after executing v->w)
- Need to distinguish them



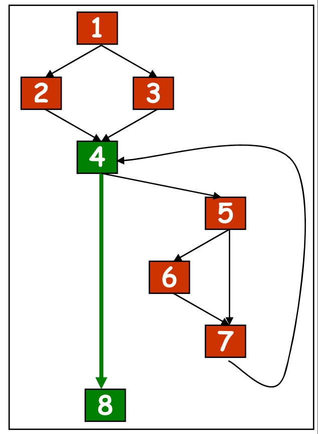
Arbitrary Control Flow (loops)

- Loop implies the presence of a back-edge
- Back-edges instrumented to increment path counter and reinitialize path register (count[r]++; r=0)
- This is not enough; with loops, 4 types of paths (v->w and x->y are back-edges)
  - ENTRY to EXIT
  - ENTRY to v (ending with execution of v->w)
  - w to x (after executing v->w and ending with the execution of x->y, v->w and x->y can be the same back-edge)
  - w to EXIT (after executing v->w)
- Need to distinguish them



Arbitrary Control Flow (loops)

- Loop implies the presence of a back-edge
- Back-edges instrumented to increment path counter and reinitialize path register (count[r]++; r=0)
- This is not enough; with loops, 4 types of paths (v->w and x->y are back-edges)
  - ENTRY to EXIT
  - ENTRY to v (ending with execution of v->w)
  - w to x (after executing v->w and ending with the execution of x->y, v->w and x->y can be the same back-edge)
  - w to EXIT (after executing v->w)
- Need to distinguish them



Convert Arbitrary CFGs to DAGs

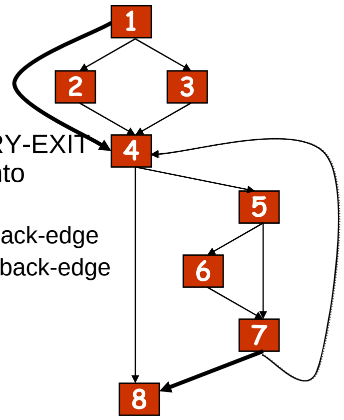
- Eliminate back-edges before computation of edge values and chord increments
- Remove a loop back-edge
- Add two edges

  (1) ENTRY -> Target of back-edge

  (2) Source of back-edge -> EXIT
- The dummy edges create extra paths ENTRY-EXIT that the value assignment algorithm takes into account
  - Edge (1) represents reinitializing along the back-edge
  - Edge (2) represents incrementing along the back-edge

Convert Arbitrary CFGs to DAGs

- Eliminate back-edges before computation of edge values and chord increments
- Remove a loop back-edge
- Add two edges
  - (1) ENTRY -> Target of back-edge
  - (2) Source of back-edge -> EXIT
- The dummy edges create extra paths ENTRY-EXIT that the value assignment algorithm takes into account
  - Edge (1) represents reinitializing along the back-edge
  - Edge (2) represents incrementing along the back-edge



Implementation

- Implemented in a tool called PP
- PP instruments SPARC binaries
- Built on top of EEL (binary instrumenter)
- Uses a register to store r
- Replaces array of counters with hash table if number of paths too large
- Plus some other optimizations

Experimental Results (i)

- Used SPEC95 benchmark programs and test suites
- Edge profiling average overhead=16.1% (2.6%-52.8%)
- Path profiling average overhead=30.9% (5.5%-96.9%)
- When hashing is used performance is hurt
- Using no hashing, overhead is comparable or lower than edge profiling

Algorithm Evolution

- Ball & Larus, "Optimally Profiling and Tracing Programs"
  - Focuses on edge and vertex profiling
  - Optimal placement of probes
- Ball, "Efficiently Counting Program Events with Support for On-line Queries"
  - Developed the technique for edge profiling with one register (instead of a counter for each edge)