Program Analysis

Path Profiling

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2023/2024

Talks You May Find Interesting

Jan 23, 9am, 2.013	**Copilot Workspace - From Issue Description to Pull Request with AI Assistance All The Way.** Dr. Tamas Szabo (GitHub)
Jan 29, 3:45pm, 0.108	**How Is The Sausage Made? A Whirlwind Tour of V8, Real-World JIT-Compilers, and Their Trade-Offs.** Dr. Daniel Lehmann (Google)
Feb 1, 10am, 2.013	**Code Search and Comprehension.** Prof. Dr. Kathryn T. Stolee (North Carolina State University)
Feb 1, 11am, 2.013	**Software Supply Chains: Open Research Topics.** Prof. Dr. Georgios Gousios (TU Delft and Endor Labs)

Warm-up Quiz

What does this Python code print?

```
def f(x):
    if x == 3:
        return ["hi"]
    else:
        for i in range(x):
            yield i

print(list(f(3)))
```

Nothing `[]` `["hi"]` `[0, 1, 2]`

Warm-up Quiz

What does this Python code print?

```
def f(x):
    if x == 3:
        return ["hi"]
    else:
        for i in range(x):
            yield i

print(list(f(3)))
```

Nothing `[]` `["hi"]` `[0, 1, 2]`

Warm-up Quiz

What does this Python code print?

```
def f(x):
    if x == 3:
        return ["hi"]
    else:
        for i in range(x):
            yield i

print(list(f(3)))
```

`yield` turns the function into a generator

Nothing   **[ ]**   `["hi"]`   `[0, 1, 2]`

Warm-up Quiz

What does this Python code print?

```
def f(x):
    if x == 3:
        return ["hi"]
    else:
        for i in range(x):
            yield i

print(list(f(3)))
```

Returning from a generator raises `StopIteration("hi")`

`yield` turns the function into a generator

Nothing | `[]` | `["hi"]` | `[0, 1, 2]`

Outline

1. **Motivation and Challenges**
2. **Ball-Larus Algorithm for DAGs**
3. **Generalization and Applications**

Mostly based on this paper:

- *Efficient path profiling*, Ball and Larus, MICRO 1996

Other reading material:

- *Whole program paths* , Larus, PLDI 1999
- *HOLMES: Effective statistical debugging via efficient path profiling*, Chilimbi et al., ICSE 2009

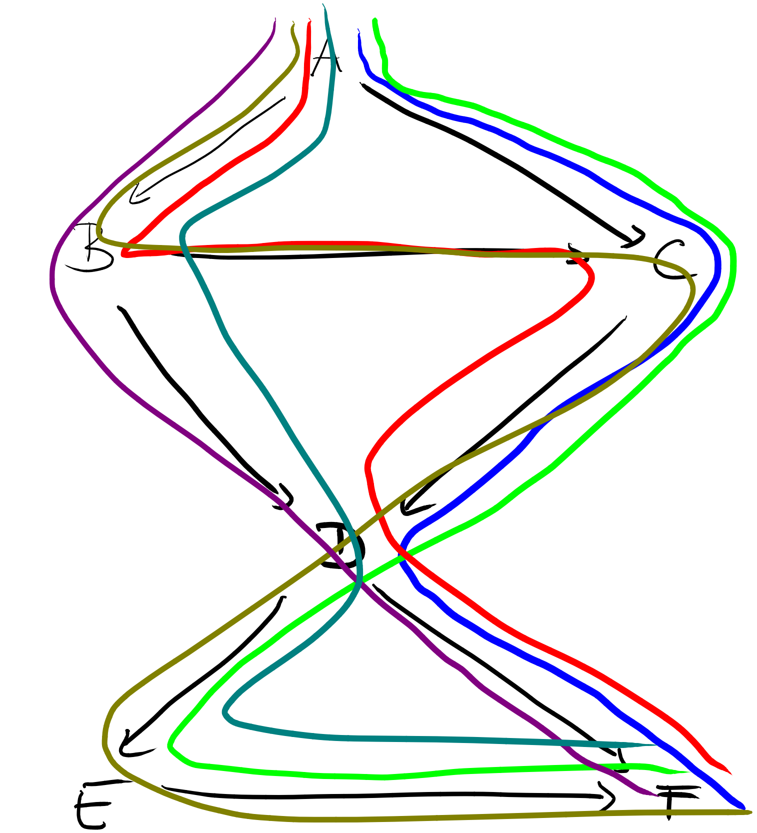
Path Profiling

- **Goal: Count how often a path through a function is executed**
- **Interesting for various applications**
  - Profile-directed compiler optimizations
  - Performance tuning: Which paths are worth optimizing?
  - Test coverage: Which paths are not yet tested?

Challenges

- **Runtime overhead**
  - Limit slowdown of program
- **Accuracy**
  - Ideally: Precise profiles (no heuristics, no approximations)
- **Infinitely many paths**
  - Cycles in control flow graph

Running Example



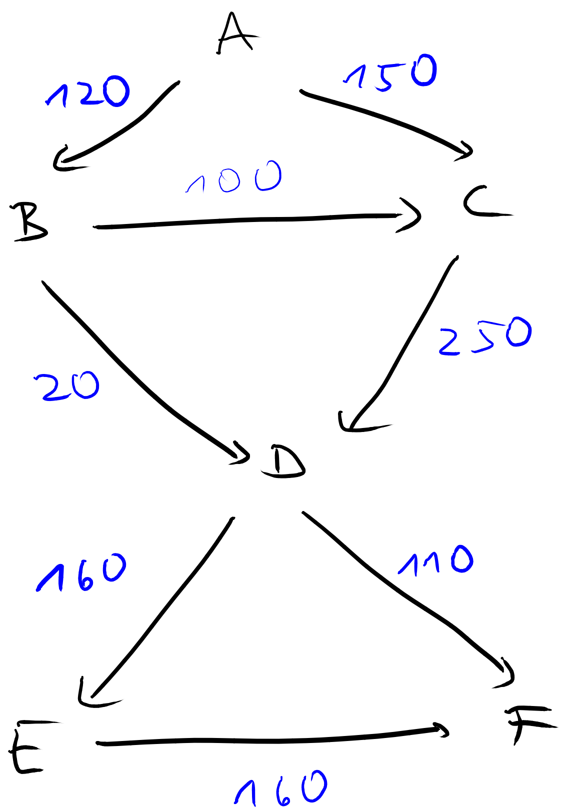
	Path	Frequency
0	A C D F	
1	A C D E F	
2	A B C D F	
3	A B C D E F	
4	A B D F	
5	A B D E F	2

Edge Profiling

Naive approach: Edge profiling

- Instrument each branching point
- Count how often each CFG edge is executed
- Estimate most frequent path: Always follow most frequent edge

Example: Edge Profiling



Frequency of execution

Most frequent path?

ACDEF

Really? Two possible path profiles

Path	Profile 1	Profile 2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20

Edge Profiling

Naive approach: Edge profiling

- **Instrument each branching point**
- **Count how often each CFG edge is executed**
- **Estimate most frequent path: Always follow most frequent edge**

Edge Profiling

Naive approach: Edge profiling

- Instrument each branching point
- Count how often each CFG edge is executed
- Estimate most frequent path: Always follow most frequent edge

Fails to uniquely identify most frequent path

Outline

1. **Motivation and Challenges**
2. **Ball-Larus Algorithm for DAGs** ←
3. **Generalization and Applications**

Mostly based on this paper:

- *Efficient path profiling*, Ball and Larus, MICRO 1996

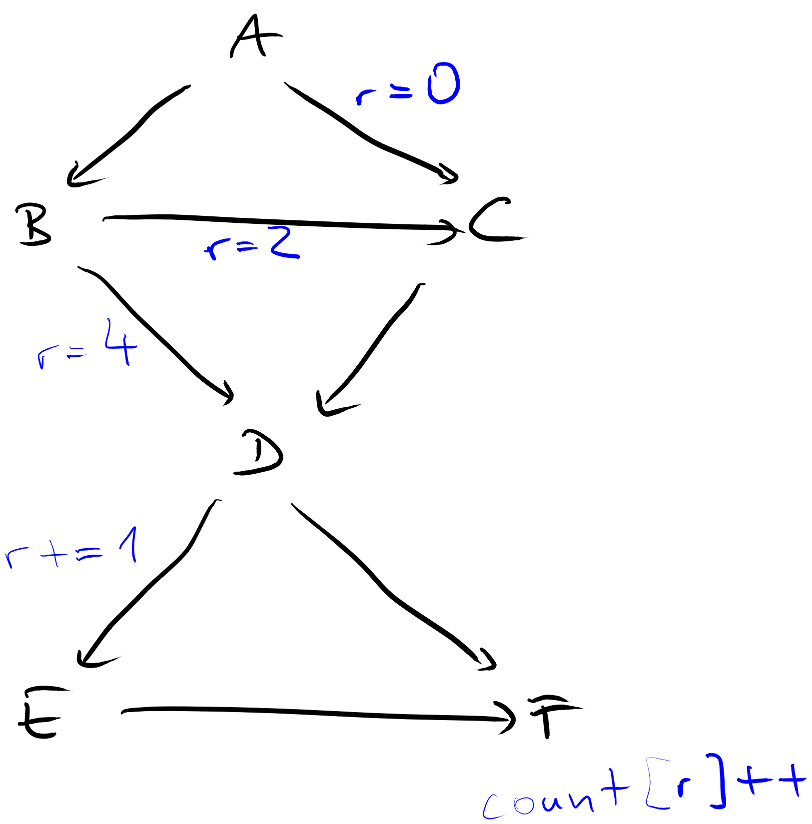
Other reading material:

- *Whole program paths* , Larus, PLDI 1999
- *HOLMES: Effective statistical debugging via efficient path profiling*, Chilimbi et al., ICSE 2009

Ball-Larus Algorithm

- **Assign a number to each path**
- **Compute path number by incrementing a counter at branching points**
- **Properties of path encoding**
  - Precise: A single unique encoding for each path
  - Minimal: Instruments subset of edges with minimal cost

Example: Path Encoding



Instrumentation:

- state/counter: r
- array of counts: count
  - ↳ indices = paths

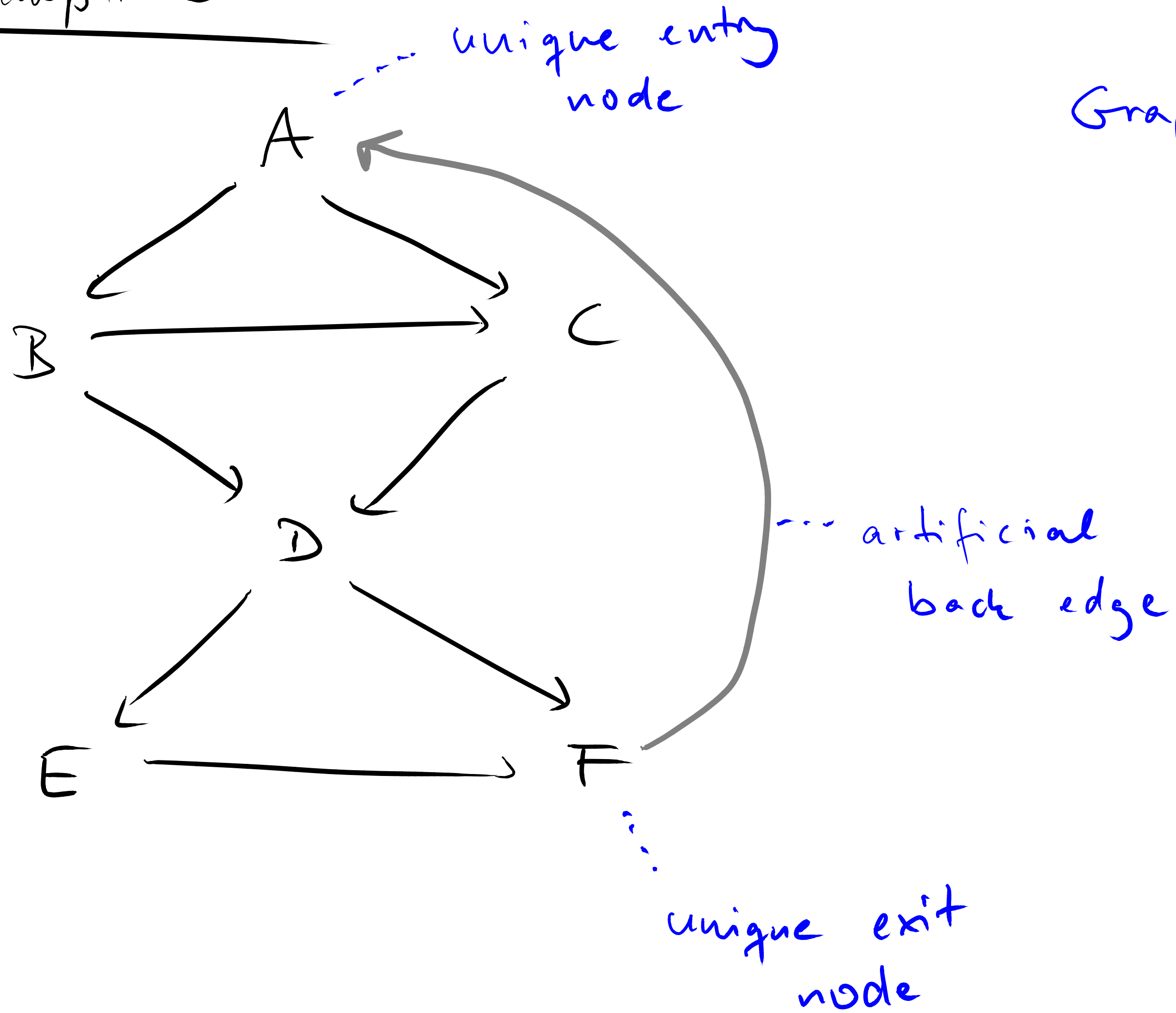
Path	Encoding
ACDF	0
ACDEF	1
ABCDF	2
ABCDEF	3
ABDF	4
ABDEF	5

Algorithm for DAGs

Assumptions

- Control flow graph is a directed acyclic graph (DAG)
- n paths (numbered 0 to $n-1$)
- Graph has unique entry and exit nodes
- Artificial back edge from exit to entry

Assumptions



Graph is acyclic ✓

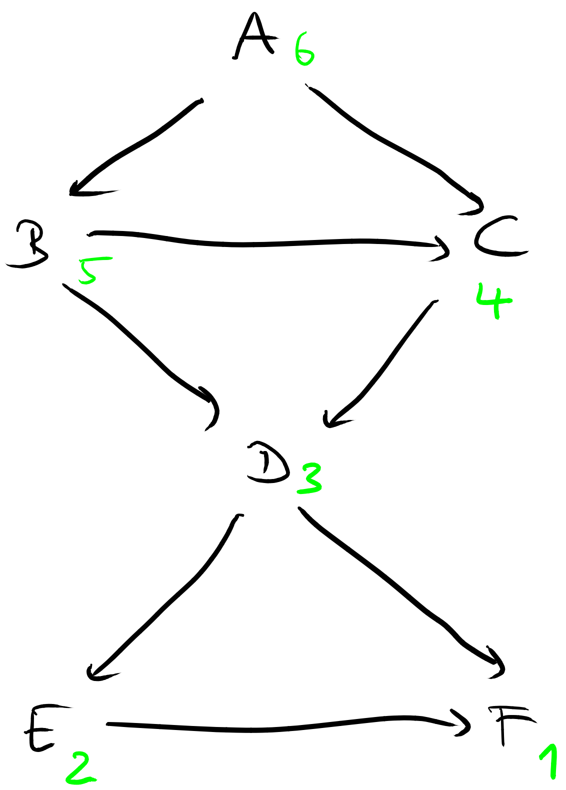
Algorithm: Overview

- **Step 1: Assign integers to edges**
  - Goal: Sum along a path yields unique number for path
  - Enough to achieve ”precise” goal
- **Step 2: Assign increment operations to edges**
  - Goal: Minimize additions along edges
  - Instrument subset of all edges
  - Assumes to know/estimate how frequent edges are executed

Representing Paths with Sums

- **Associate with each node a value: $NumPaths(n)$ = number of paths from n to exit**
- **Computing $NumPaths$**
  - Visit nodes in reverse topological order
  - If n is leaf node: $NumPaths(n) = 1$
  - Else: $NumPaths(n) =$ sum of $NumPaths$ of destination of outgoing edges

Example: NumPaths



Reverse topological order:

Successor of n visited before n

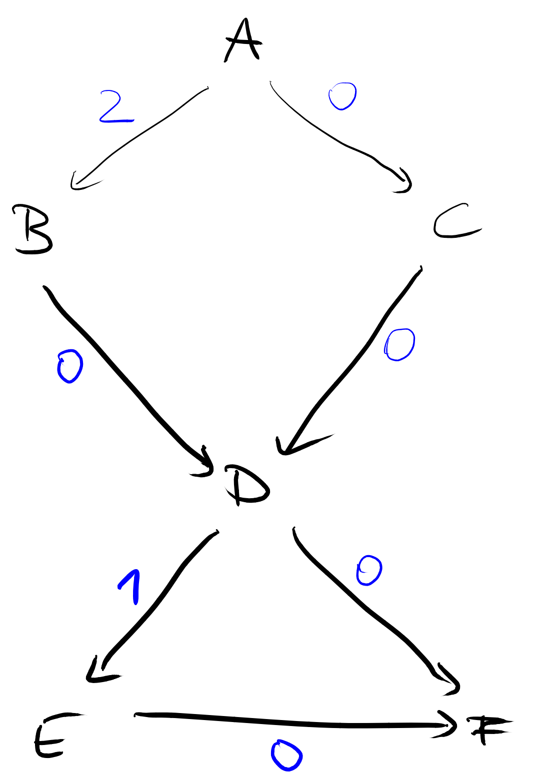
Node n	NumPaths (n)
F	1
E	1
D	1+1 = 2
C	2
B	2 + 2 = 4
A	4 + 2 = 6

Representing Paths with Sums (2)

For each node in reverse topological order:

- **If n is leaf node:**
  $NumPaths(n) = 1$
- **Else:**
  - $NumPaths(n) = 0$
  - For each edge $n \rightarrow m$:
    - $Val(n \rightarrow m) = NumPaths(n)$
    - $NumPaths(n) \mathrel{+}= NumPaths(m)$

Quiz: Values for Edges



n	NumPaths (n)
F	1
E	1
D	2
C	2
B	2
A	4

Path encoding of ABDEF: 3

Algorithm: Overview

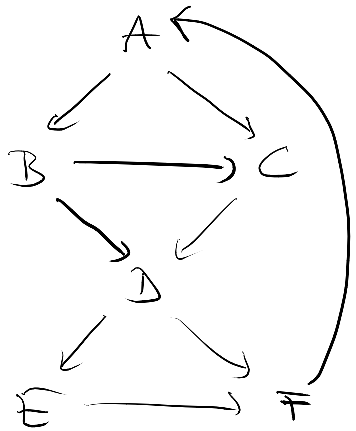
- **Step 1: Assign integers to edges**
  - Goal: Sum along a path yields unique number for path
  - Enough to achieve ”precise” goal
- **Step 2: Assign increment operations to edges**
  - Goal: Minimize additions along edges
  - Instrument subset of all edges
  - Assumes to know/estimate how frequent edges are executed

Spanning Tree

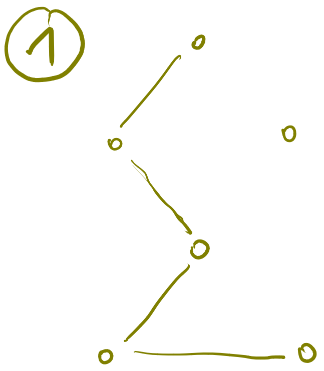
- **Given: Graph G**
- **Spanning tree T:**
  Undirected subgraph of G that is a tree and that contains all nodes of G
- **Chord edges: Edges in G but not in T**

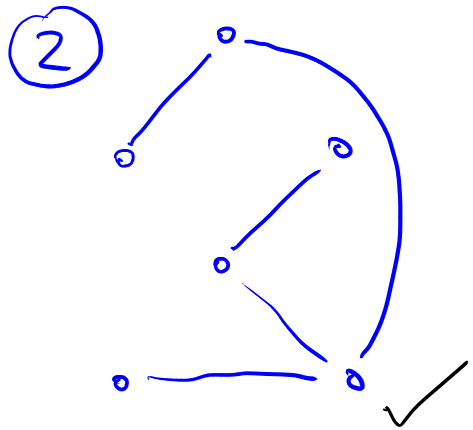
Example: Spanning Tree

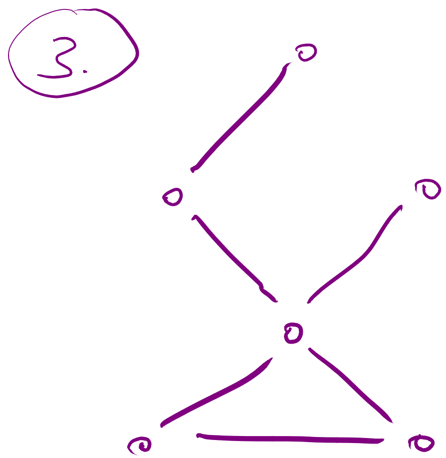
G:

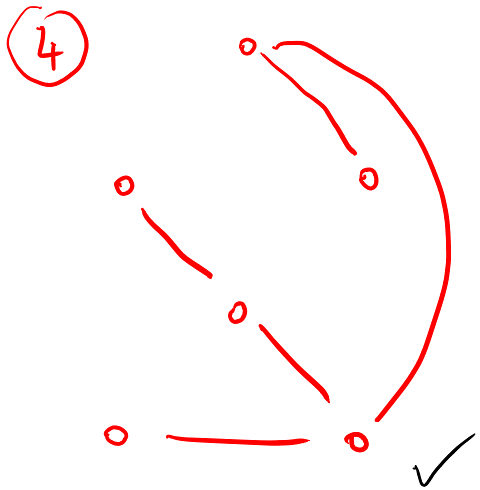


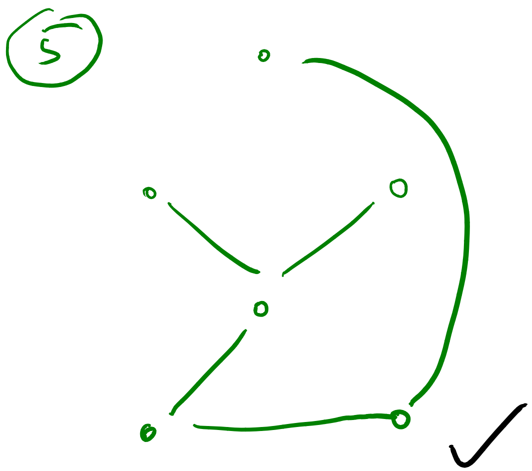
Which of these are spanning trees of G?









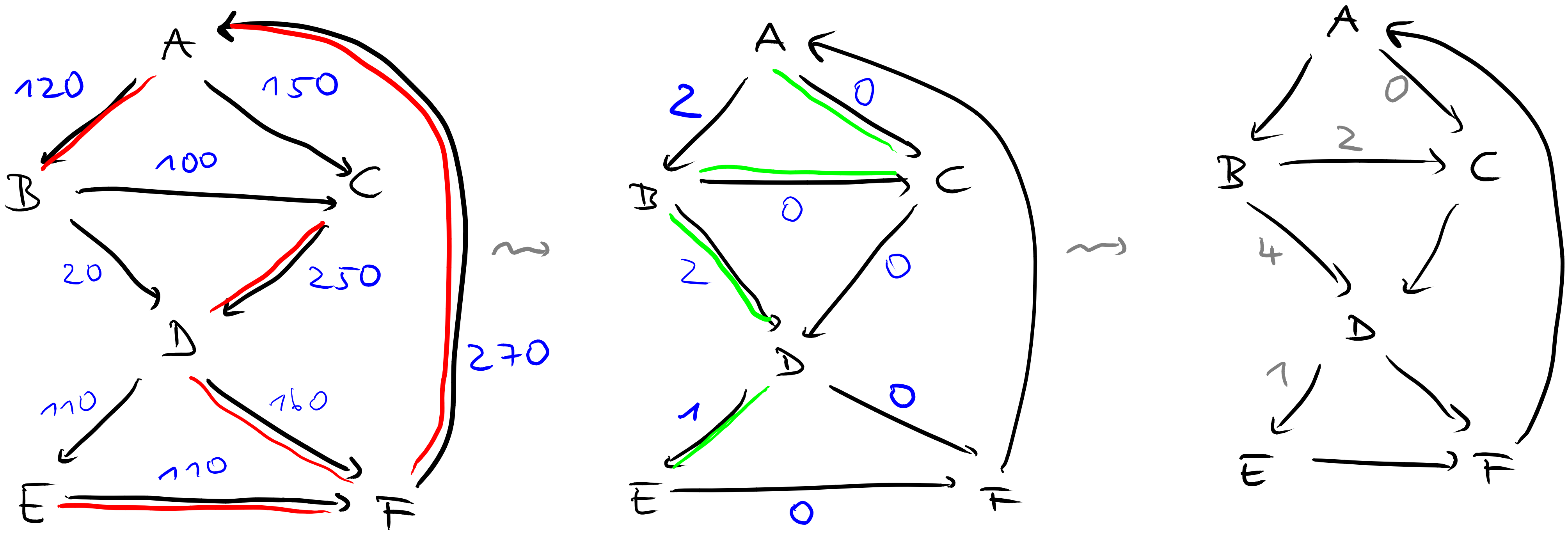


Increments for Edges

Goal: Increment sum at subset of edges

- **Choose spanning tree with maximum edge cost**
  - Cost of individual edges is assumed to be known
- **Compute increments at the chords of the spanning tree**
  - Based on existing event counting algorithm

Example: Increments for Edges



edge cost
most expensive spanning tree

chord edges
non-minimal increments

minimal increments
→ path encoding

Instrumentation

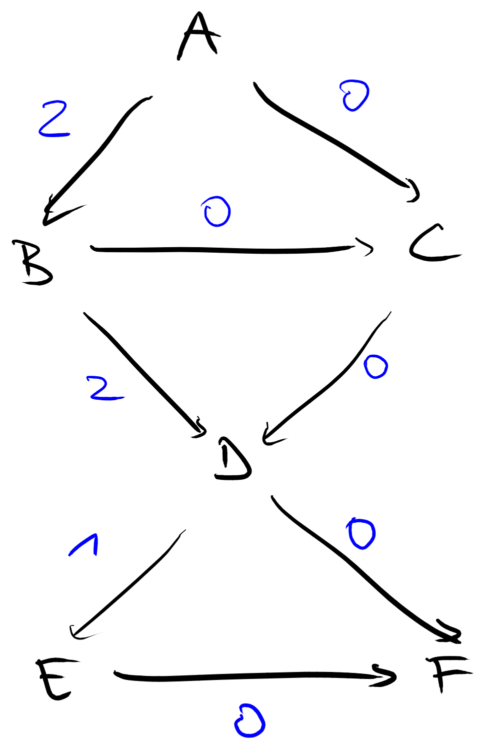
- **Basic idea**
  - Initialize sum at entry: `r=0`
  - Increment at edges: `r+=..`
  - At exit, increment counter for path: `count[r]++`
- **Optimization**
  - Initialize with incremented value, if first chord edge on path: `r=..`
  - Increment sum and counter for path, if last chord edge on path: `count[r+..]++`

Regenerating the Path

Knowing the sum r, how to determine the path?

- Use edge values from step 1 (”non-minimal increments”)
- Start at entry with $R = r$
- At branches, use edge with largest value v that is smaller than or equal to R and set $R -= v$

Example: Regenerate the Path



$r = 4$

A B D F

$R = \cancel{4}\ \cancel{2}\ 0$

$r = 1$

A C D E F

$R = \cancel{1}\ 0$

Outline

1. **Motivation and Challenges**
2. **Ball-Larus algorithm for DAGs**
3. **Generalization and Applications** ←

Mostly based on this paper:

- *Efficient path profiling*, Ball and Larus, MICRO 1996

Other reading material:

- *Whole program paths* , Larus, PLDI 1999
- *HOLMES: Effective statistical debugging via efficient path profiling*, Chilimbi et al., ICSE 2009

Generalizing to Cyclic CFGs

- **For each backedge $n \to m$, add dummy edges**
  - $Entry \to m$
  - $n \to Exit$
- **Remove backedges and add DAG-based increments**
- **In addition, add instrumentation to each backedge**
  - `count[r]++; r=0`

Generalizing to Cyclic CFGs (2)

- **Leads to four kinds of paths**
  - From entry to exit
  - From entry to backedge
  - From end of backedge to beginning of (possibly another) backedge
  - From end of backedge to exit
- **Full path information can be constructed from these four kinds**

Applications

- **Performance optimization**
  - Frequent path should get most attention by optimizer
- **Statistical debugging**
  - Paths correlated with failure are more likely to contain the bug
- **Energy analysis**
  - Warn developers about paths and statements associated with high power consumption