Low Overhead Profiling

A patchable infrastructure for execution frequency profiling

Speaker: Andrew Craik
Devin Papineau, Nicholas Coughlin
8-Nov-2017 (16th Workshop on Compiler Driven Performance)

Software Profiling

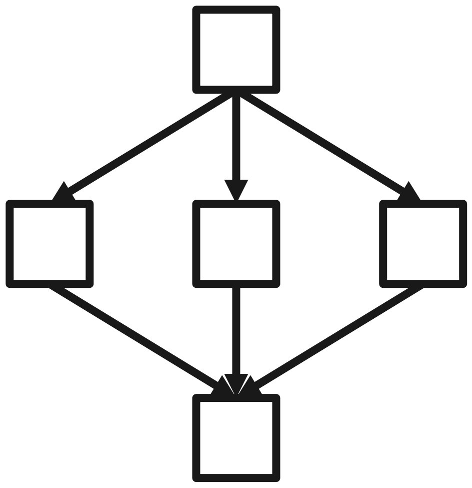
- Traditional solution to execution profiling
- Benefits: cross-platform, recycles existing compiler infrastructure
- Costs: runtime overhead, compiler complexity, code complexity
- OpenJ9 profiling story:
  - Initial profiling from bytecode interpreter (IProfiling) – full fidelity, but is shutoff once method is compiled
  - JIT profiled compilations
    - Counters placed late – avoid complexity during optimization
    - Limit optimizations to preserve basic block metadata from ilgen
    - Duplicate method body – profiled & non-profiled w/ interleaving

Hardware Profiling

- Idea: “free” hardware counters to profile running application
- Practice: Execution profiling using instruction retirement sampling
- Complications:
  - Complexity from fundamentally random nature – distilling precise data from noisy sampling is very difficult
  - Hypervisors limit access to hardware counters for security reasons
  - Sampling can inherently hide application behavior when sampling period and event period coincide closely
  - Starvation / contention caused by event buffer processing thread

Counter Placement

- Counter placement core algorithm due to:
  - Knuth & Stevenson, “Optimal measurement points for program frequency counts”, BIT13, 1973
  - Ball & Larus, “Optimally Profiling and Tracing Programs”, TOPLAS, 1994
- Formulation based on CFG edge execution frequency profiling
- Intuition:

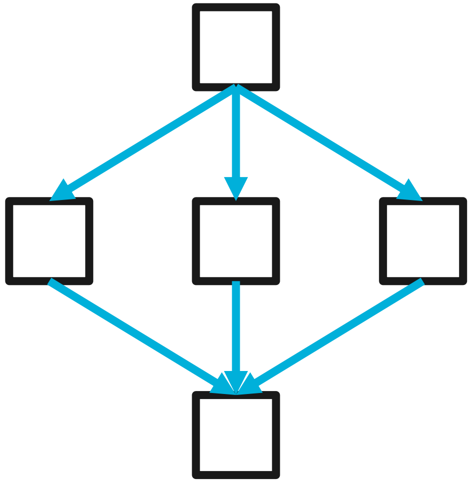


Counter Placement

- Counter placement core algorithm due to:
  – Knuth & Stevenson, "Optimal measurement points for program frequency counts", BIT13, 1973
  – Ball & Larus, "Optimally Profiling and Tracing Programs", TOPLAS, 1994

- Formulation based on CFG edge execution frequency profiling

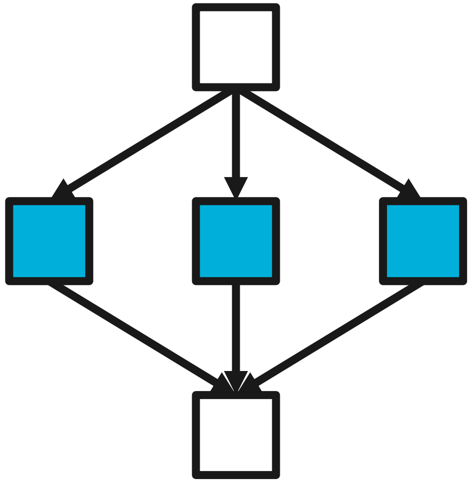
- Intuition:



Naïve – count all edges

Counter Placement

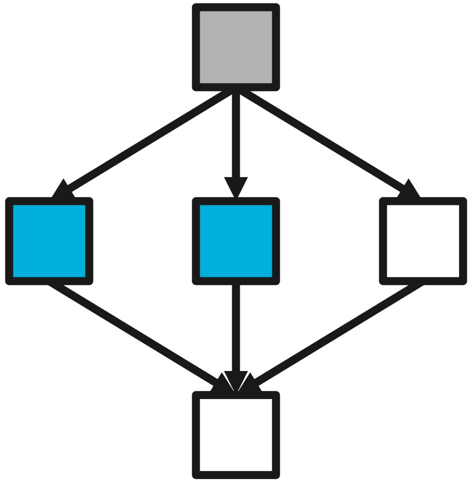
- Counter placement core algorithm due to:
  - Knuth & Stevenson, “Optimal measurement points for program frequency counts”, BIT13, 1973
  - Ball & Larus, “Optimally Profiling and Tracing Programs”, TOPLAS, 1994
- Formulation based on CFG edge execution frequency profiling
- Intuition:



Count block when edge splitting not required

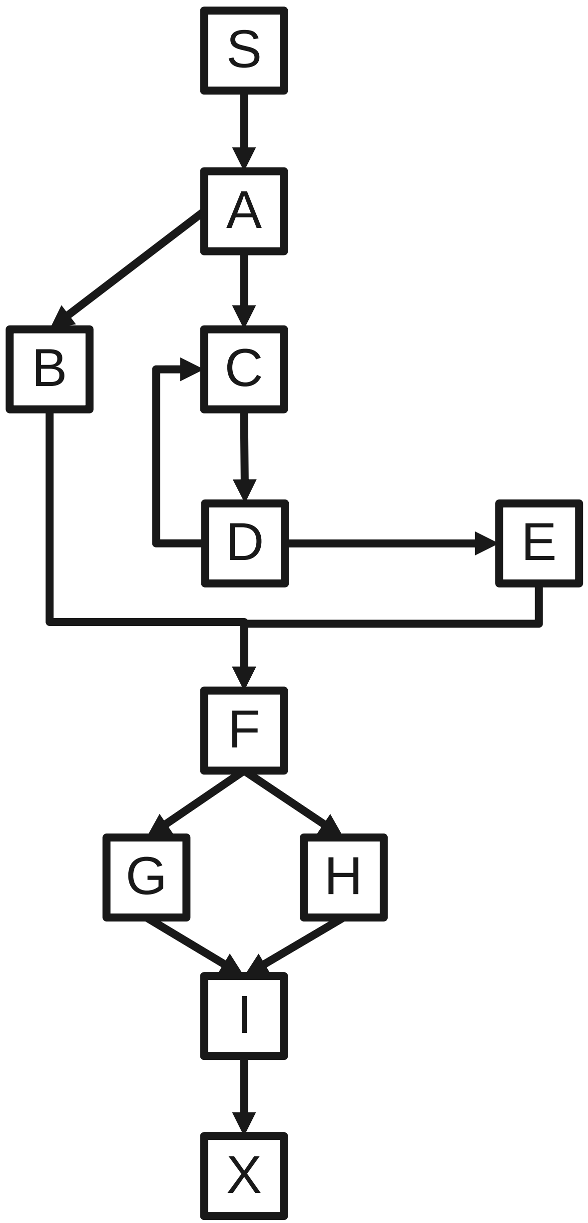
Counter Placement

- Counter placement core algorithm due to:
  - Knuth & Stevenson, "Optimal measurement points for program frequency counts", BIT13, 1973
  - Ball & Larus, "Optimally Profiling and Tracing Programs", TOPLAS, 1994
- Formulation based on CFG edge execution frequency profiling
- Intuition:

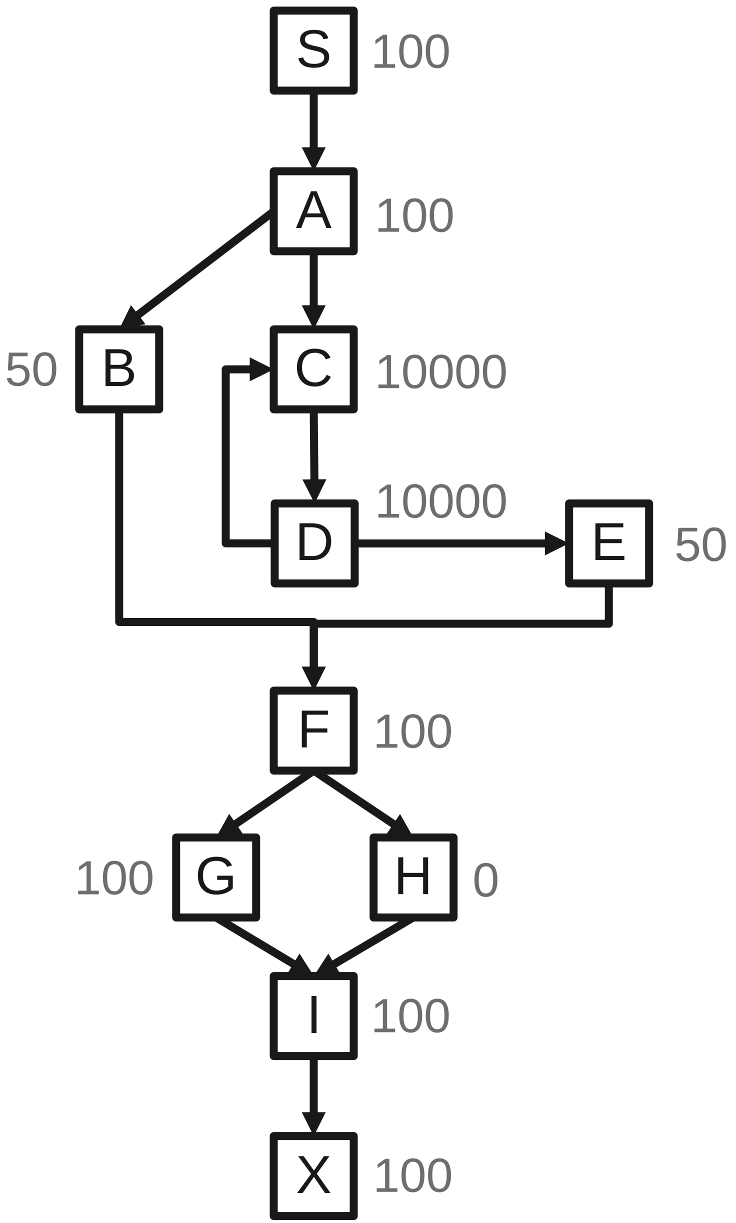


If we know the entry frequency or the exit frequency from this fragment we only need 2 counters

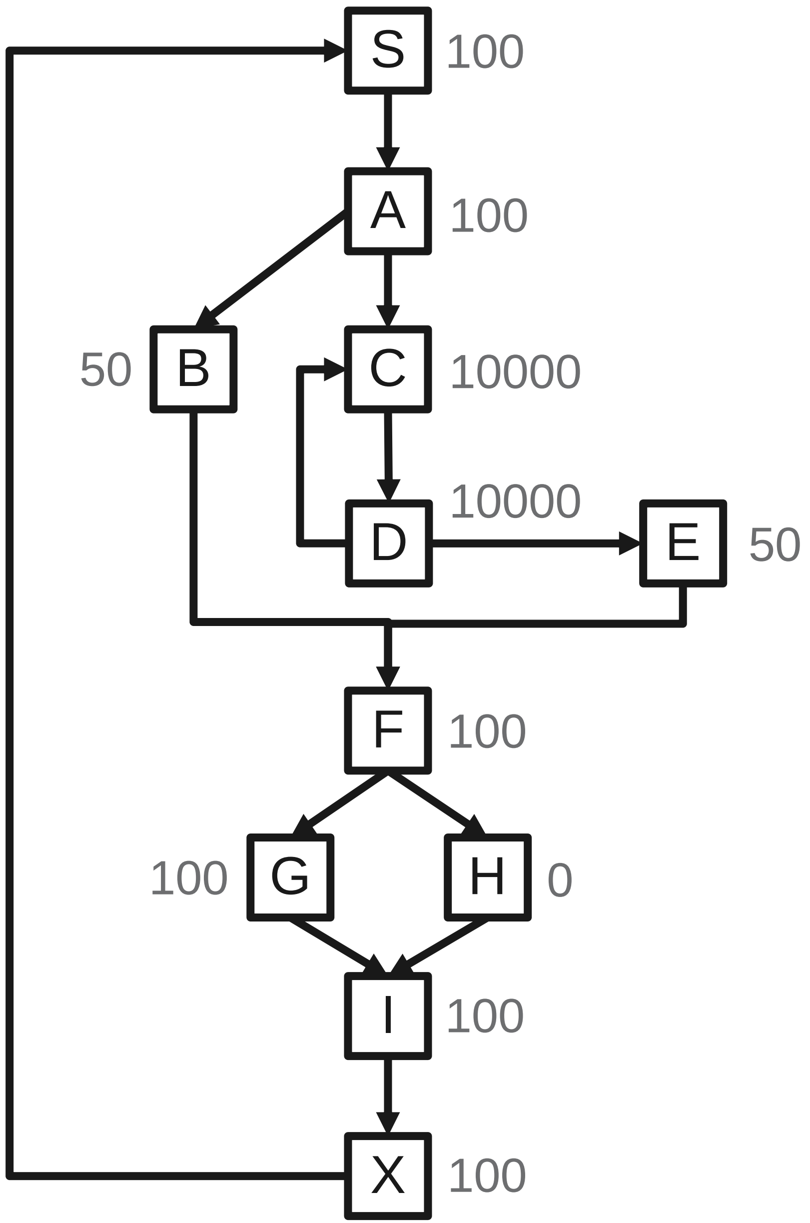
Maximum Spanning Tree Formulation



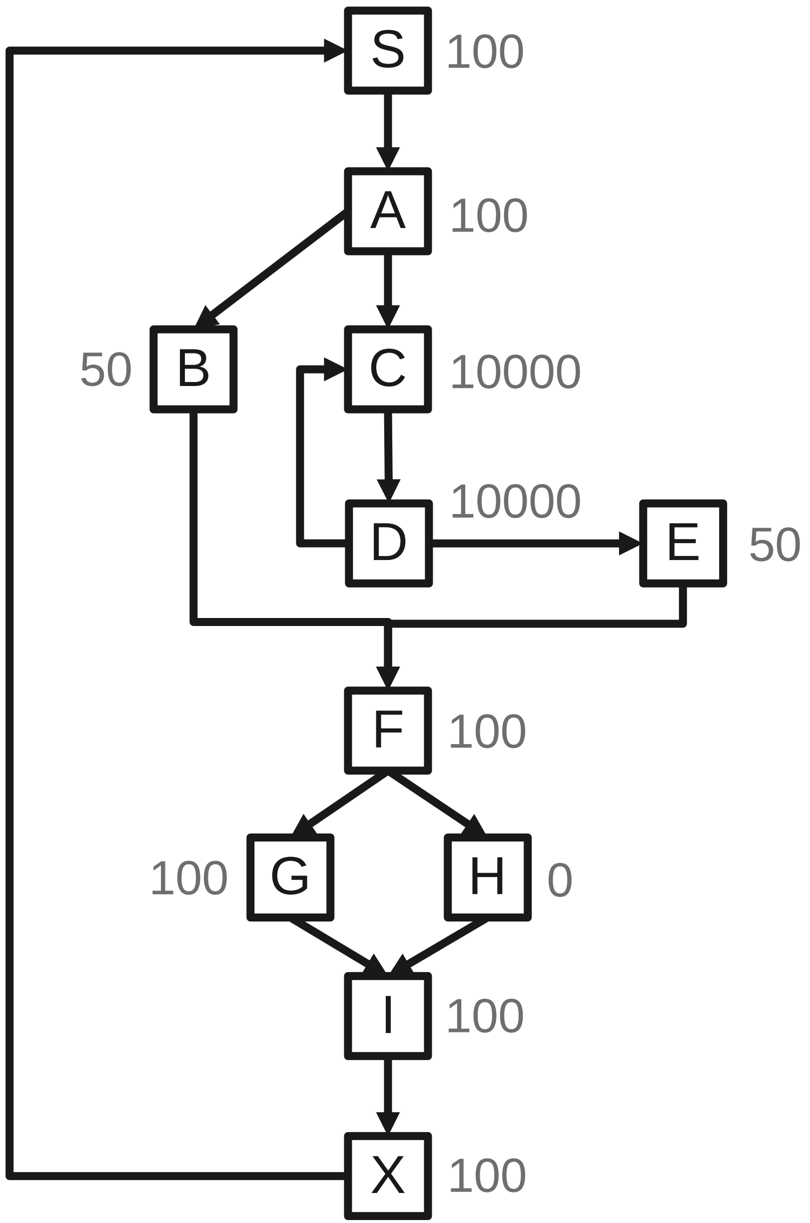
Maximum Spanning Tree Formulation



Maximum Spanning Tree Formulation

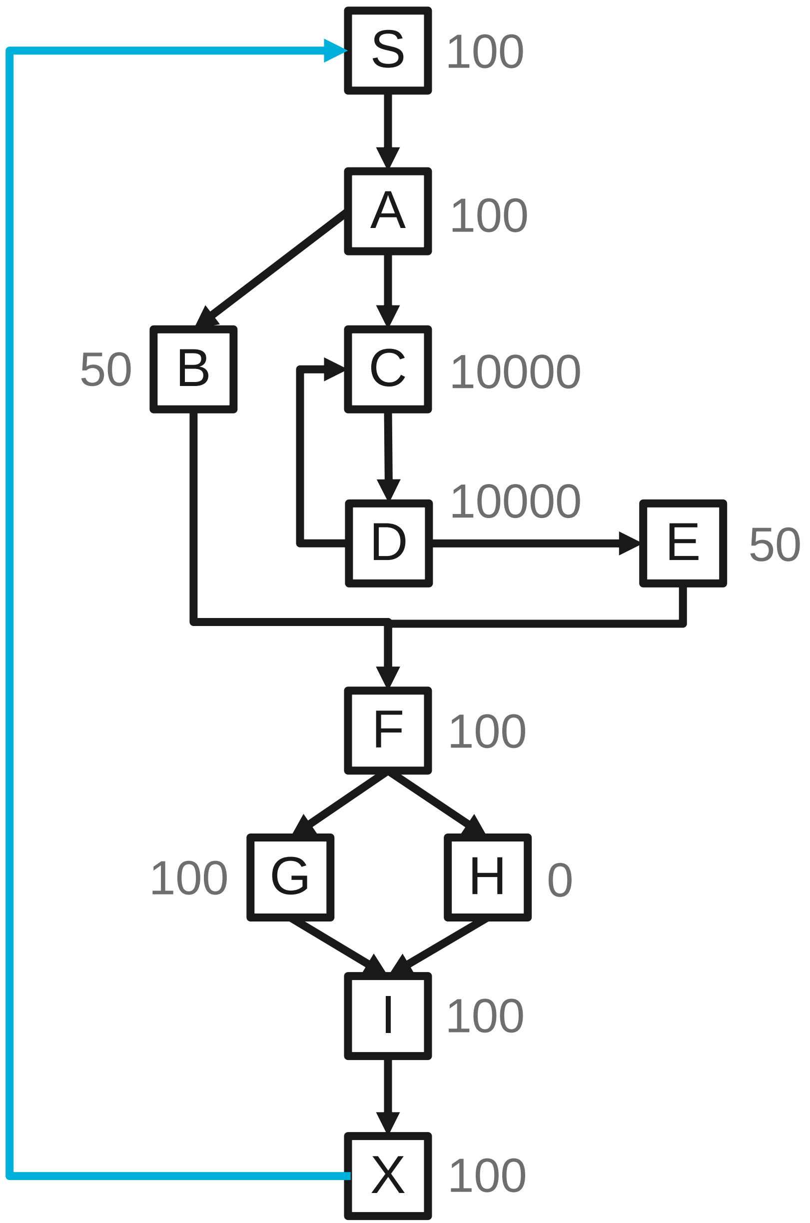


Maximum Spanning Tree Formulation



X→S	100000

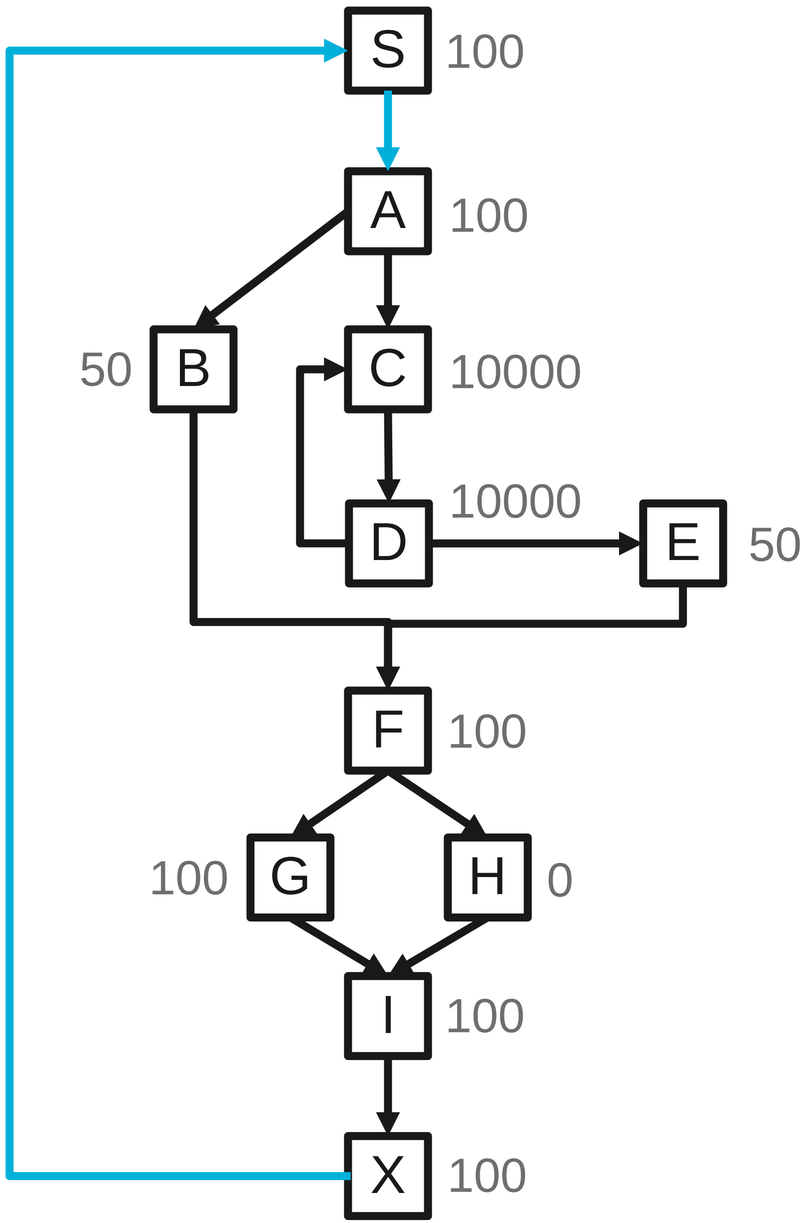
Maximum Spanning Tree Formulation



S→A	100

X→S

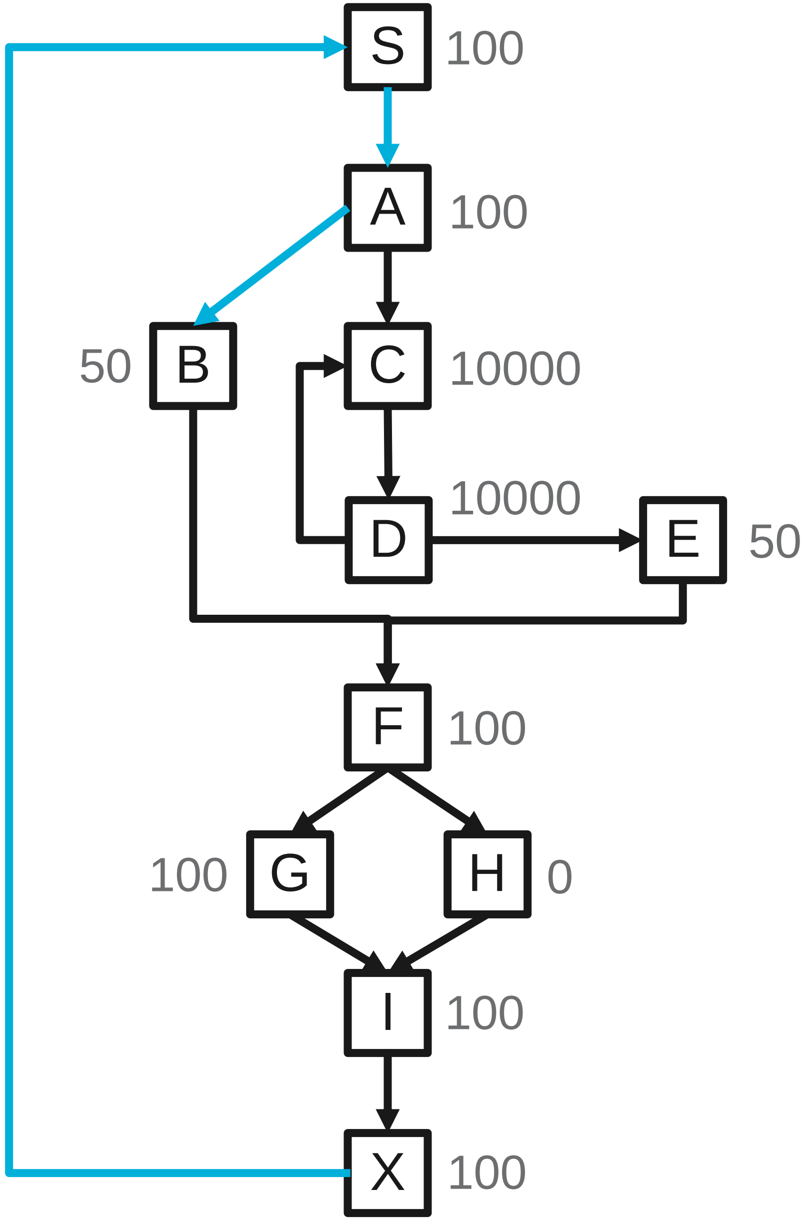
Maximum Spanning Tree Formulation



A→B	50
A→C	50

X→S
S→A

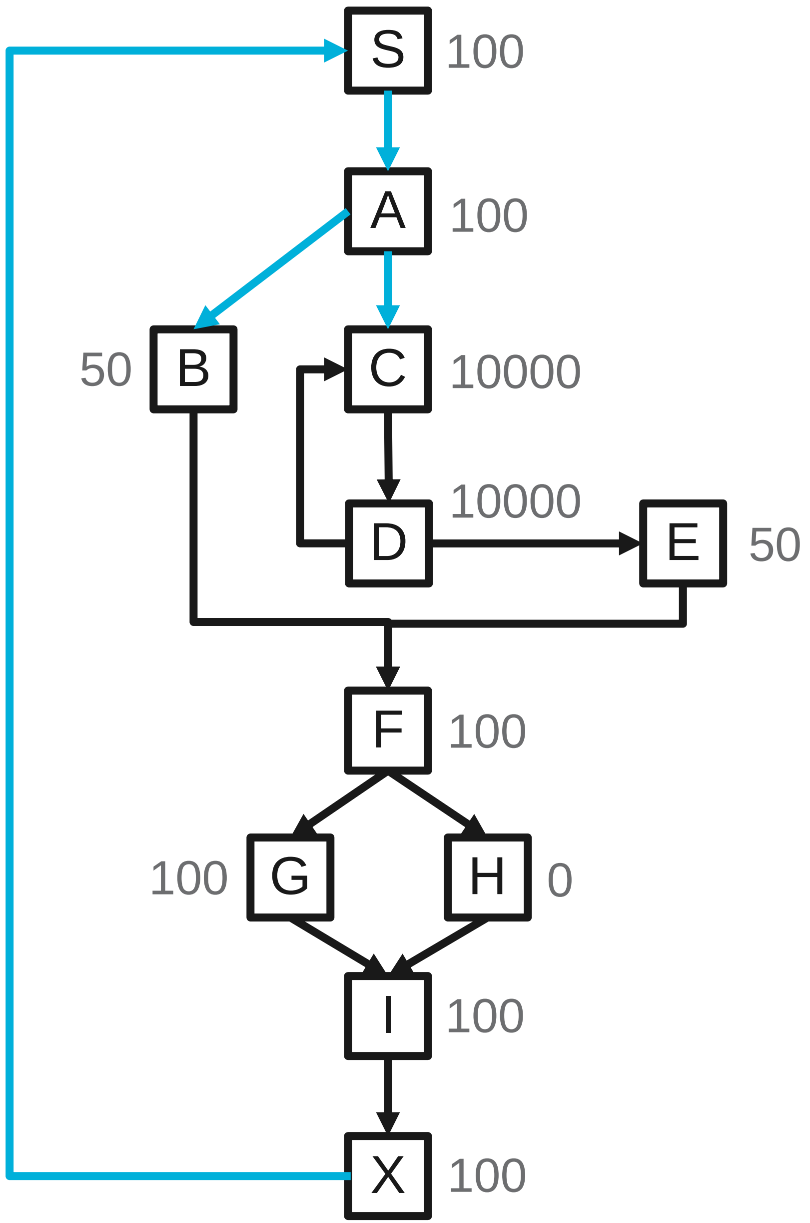
Maximum Spanning Tree Formulation



A→C	50
B→F	50

X→S
S→A
A→B

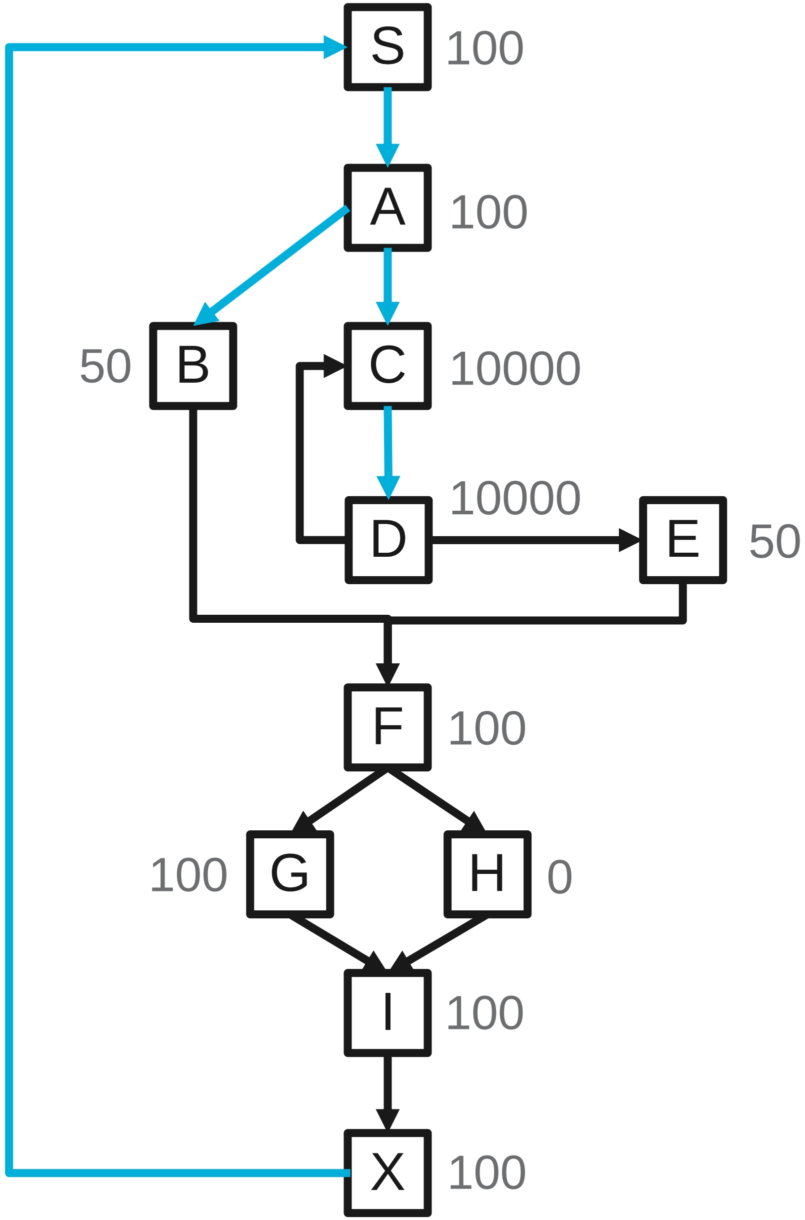
Maximum Spanning Tree Formulation



C→D	10000
B→F	50

X→S
S→A
A→B
A→C

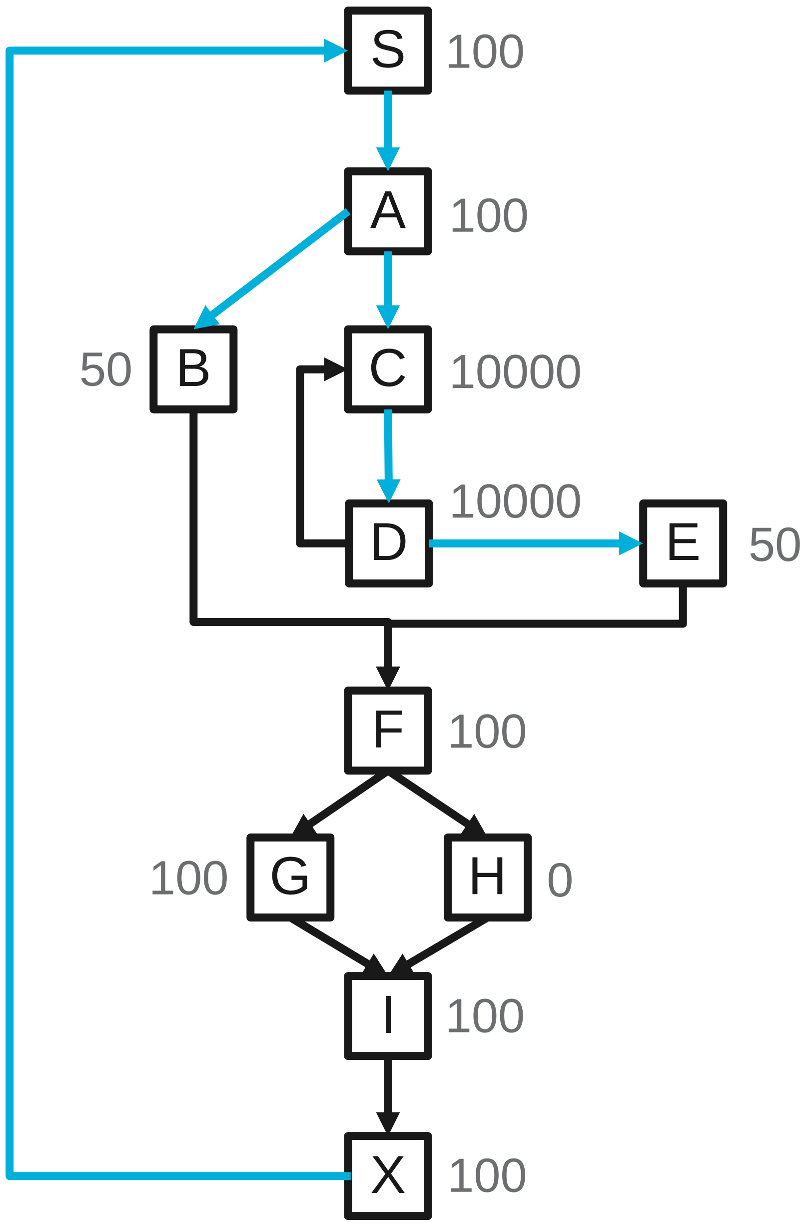
Maximum Spanning Tree Formulation



D→E	10000
B→F	50

X→S
S→A
A→B
A→C
C→D

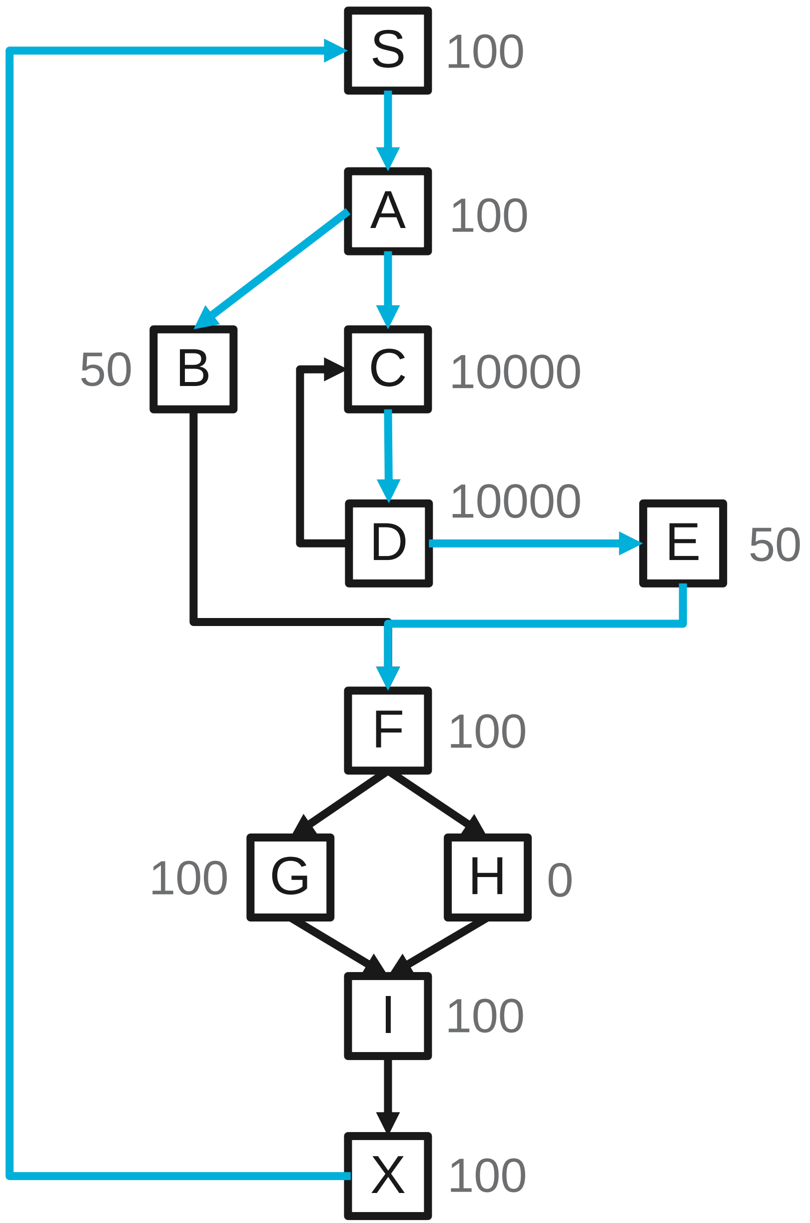
Maximum Spanning Tree Formulation



E→F	50
B→F	50

X→S
S→A
A→B
A→C
C→D

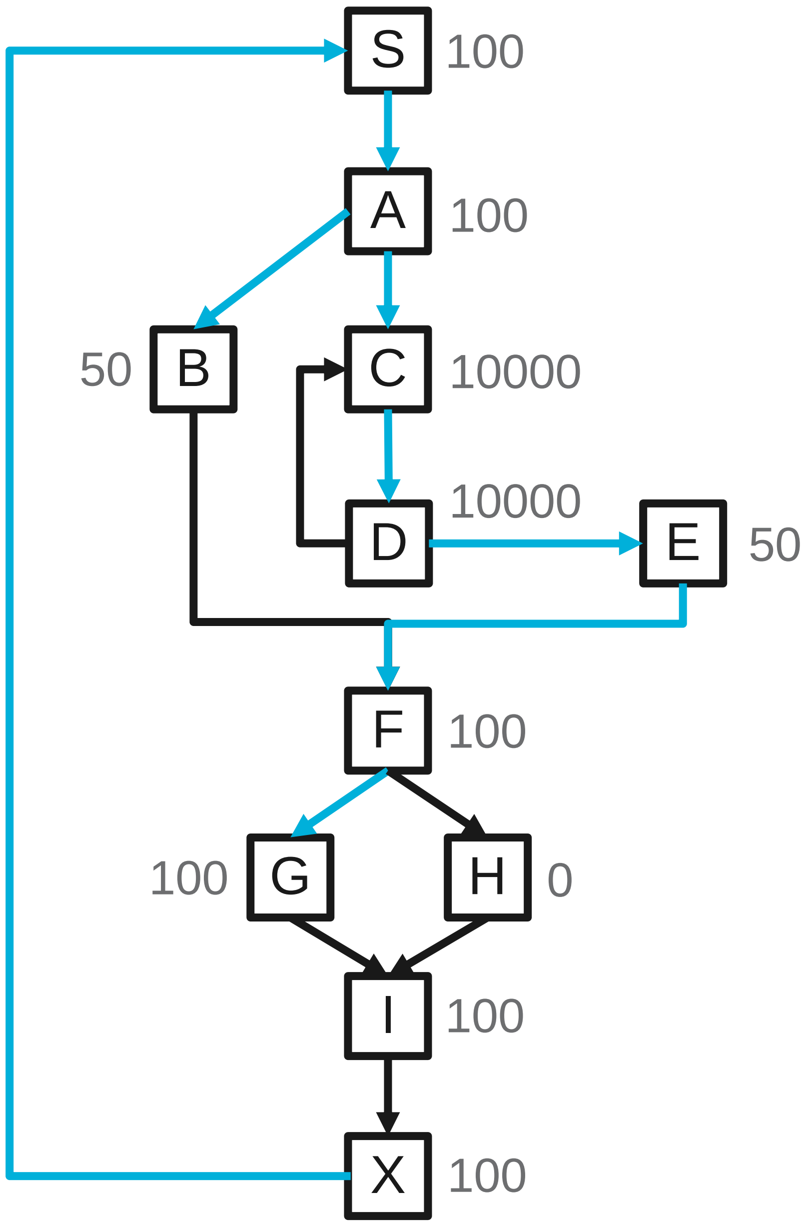
Maximum Spanning Tree Formulation



F→G	100
B→F	50
F→H	0

X→S
S→A
A→B
A→C
C→D

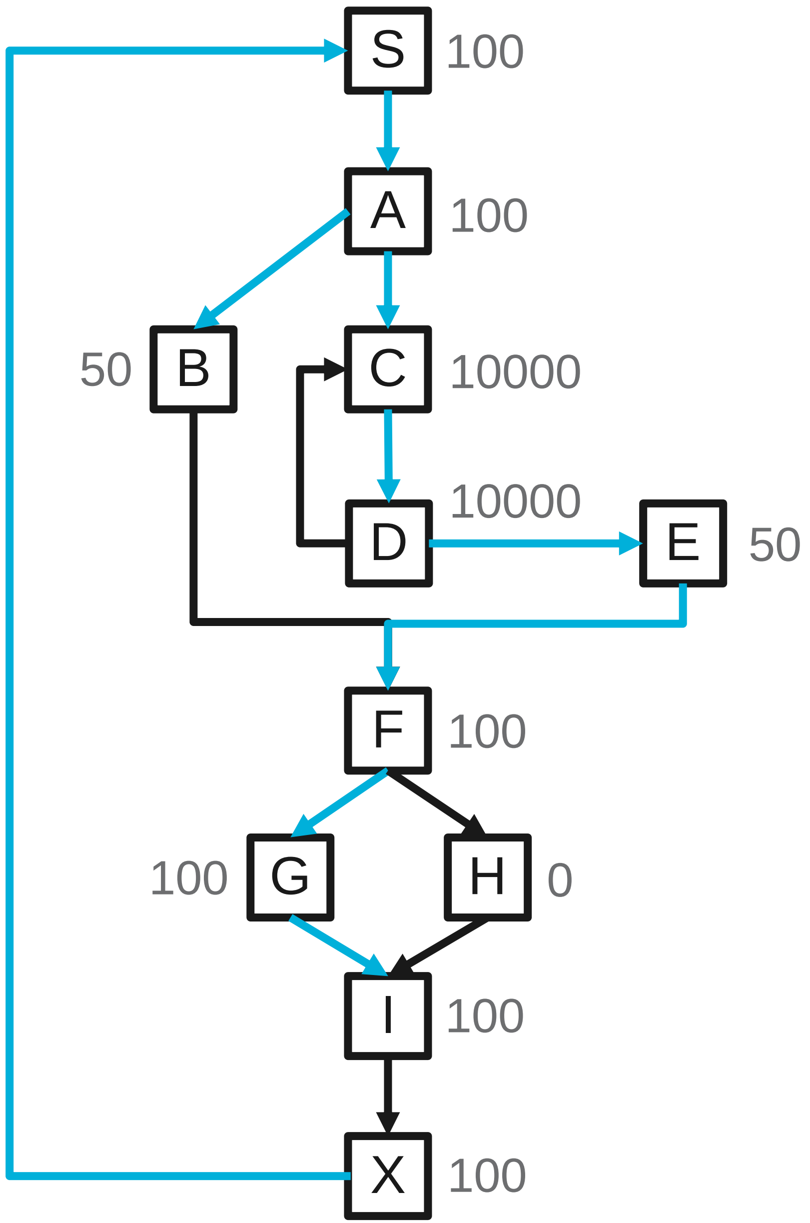
Maximum Spanning Tree Formulation



G→I	100
B→F	50
F→H	0

X→S
S→A
A→B
A→C
C→D
F→G

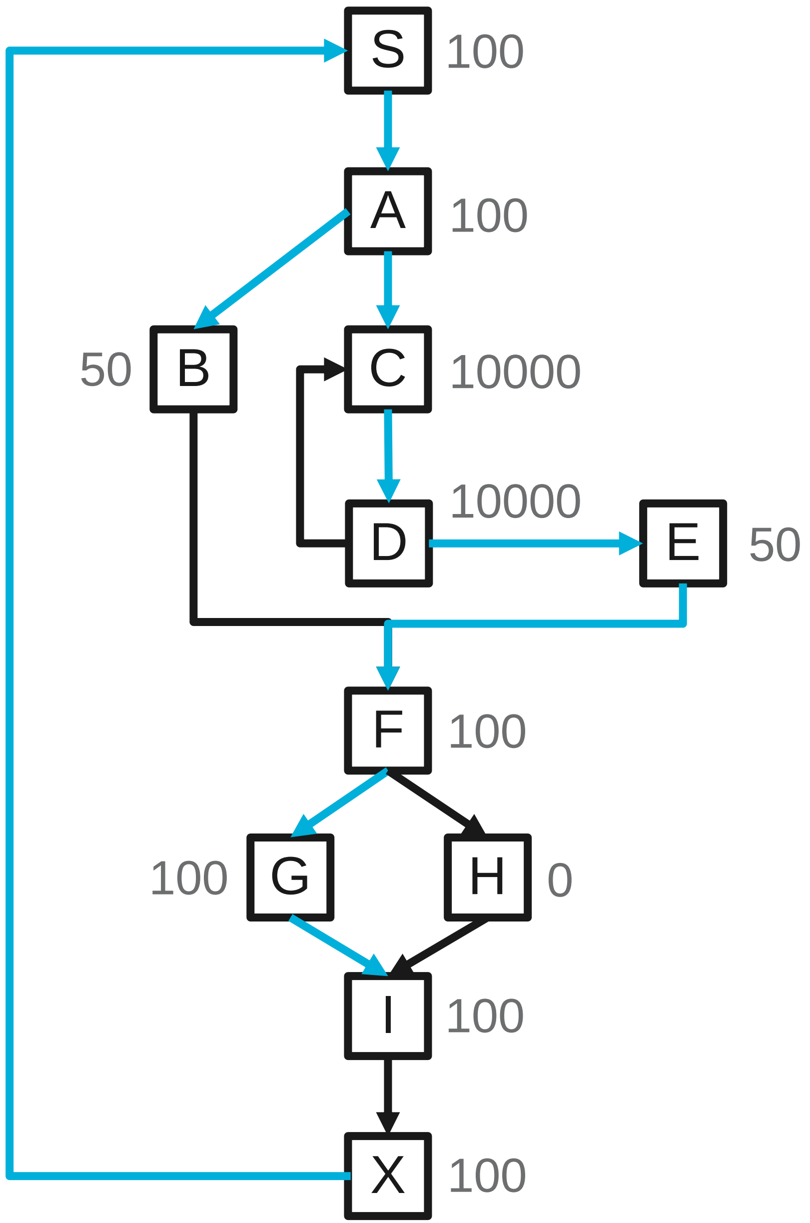
Maximum Spanning Tree Formulation



B→F	50
F→H	0

X→S
S→A
A→B
A→C
C→D
F→G
G→I

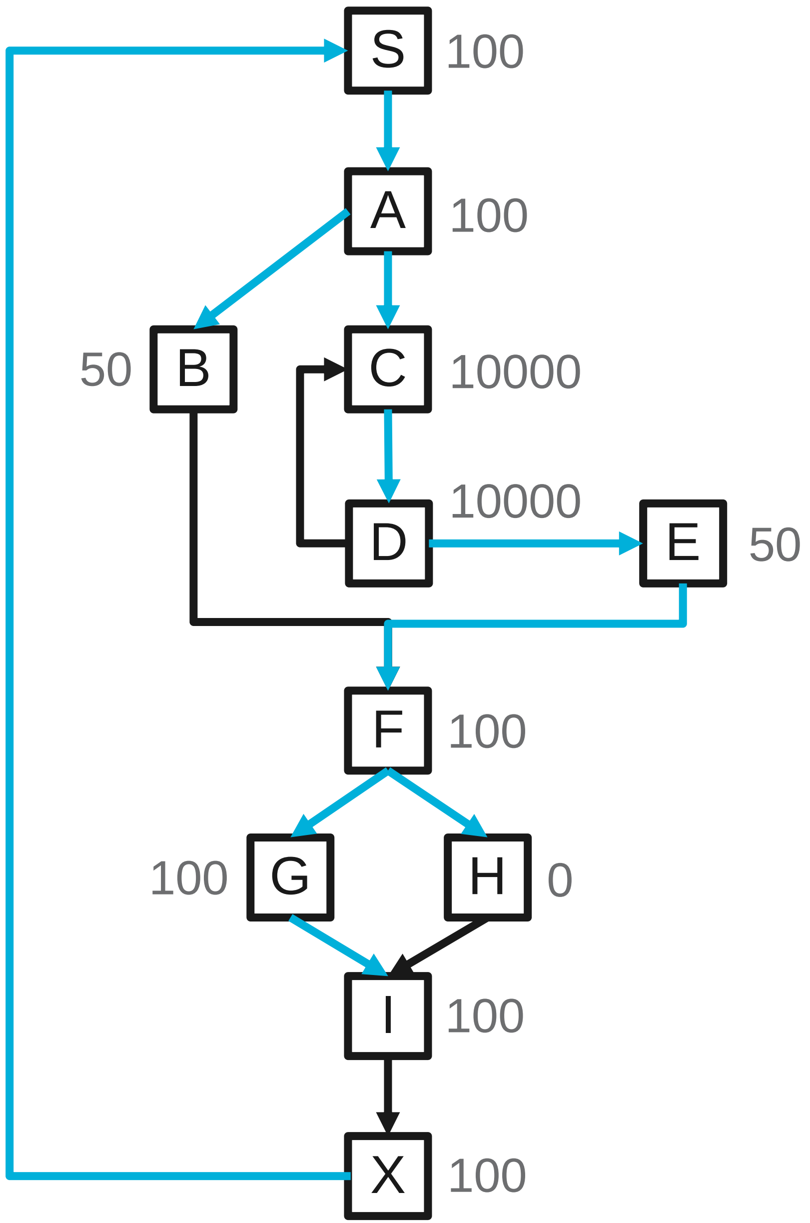
Maximum Spanning Tree Formulation



F→H	0

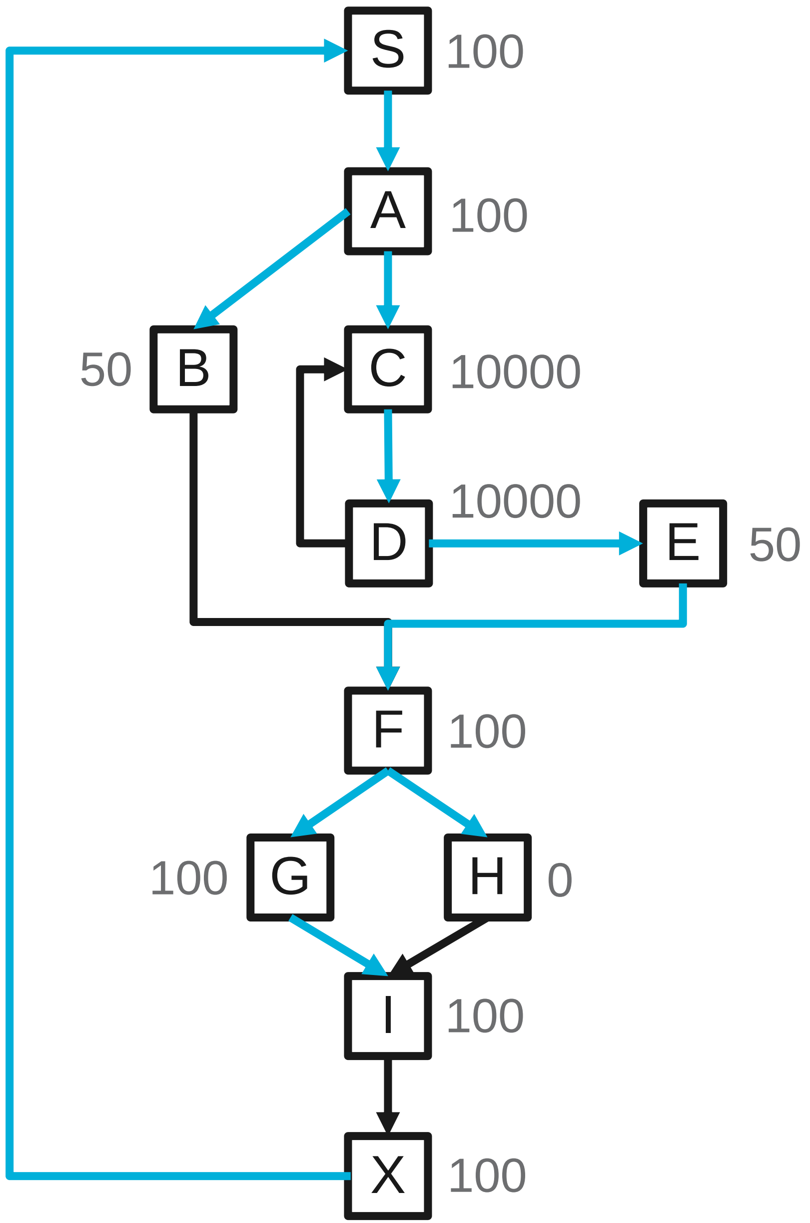
X→S
S→A
A→B
A→C
C→D
F→G
G→I

Maximum Spanning Tree Formulation



X→S
S→A
A→B
A→C
C→D
F→G
G→I
F→H

Maximum Spanning Tree Formulation



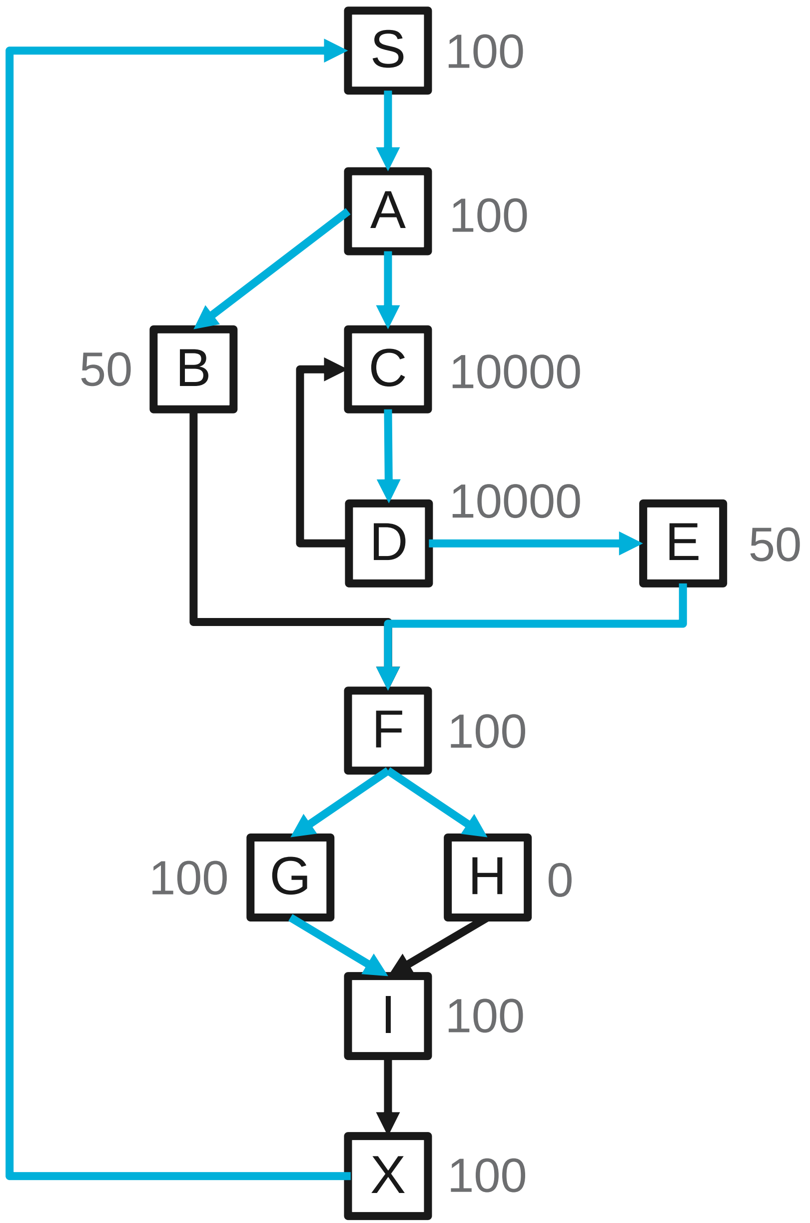
X→S
S→A
A→B
A→C
C→D
F→G
G→I
F→H

- One counter on hottest path of loop CD
- One counter on hottest path from loop to exit FGIX

Spanning Tree Placement – Benefits

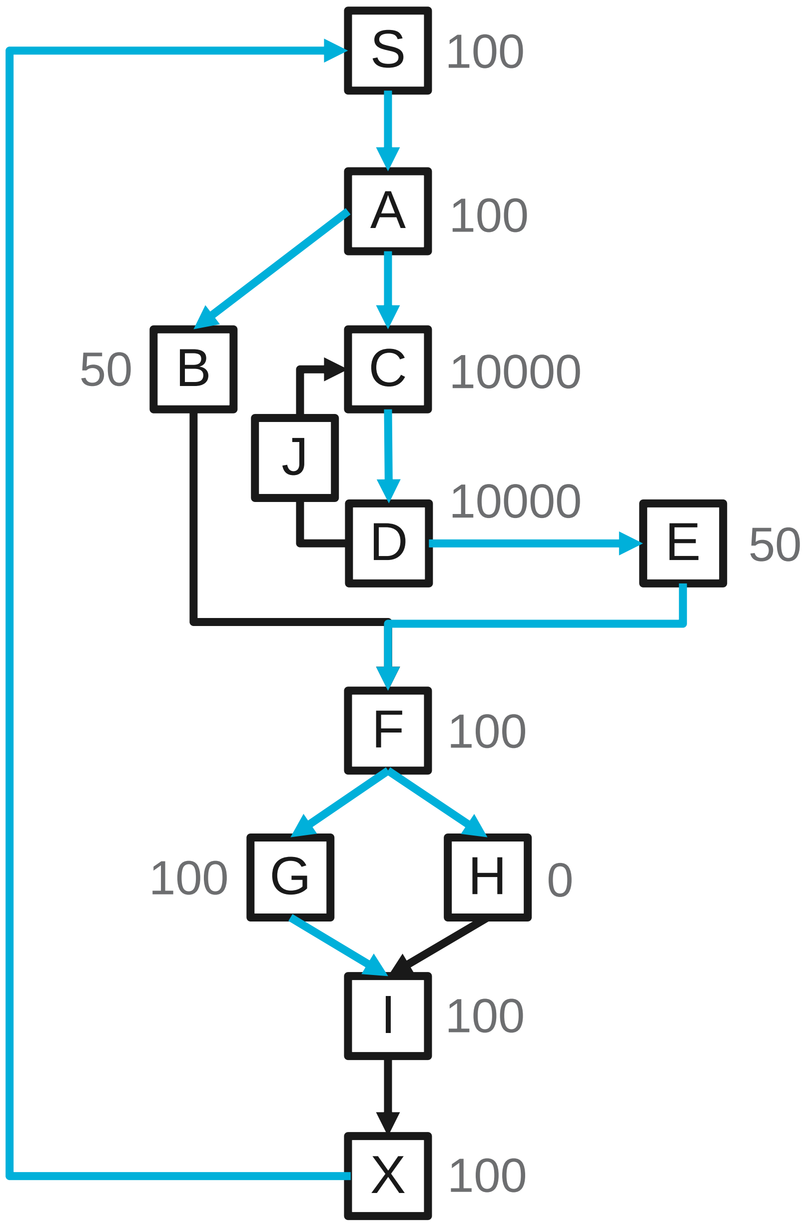
- Acyclic region – hottest path has at most one counter
- More counter updates happen on colder paths – eg the more wrong our original execution expectation (we are learning more)
- Implementation favors counters at areas of maximum CFG fanout (lower frequency edges & blocks): reduces thread contention
- Can compute frequency of any single block as a sequence of counter additions and subtractions – each counter appears at most once
- Block frequencies can be stored using a pair of bit vectors – high information density

Maximum Spanning Tree Formulation



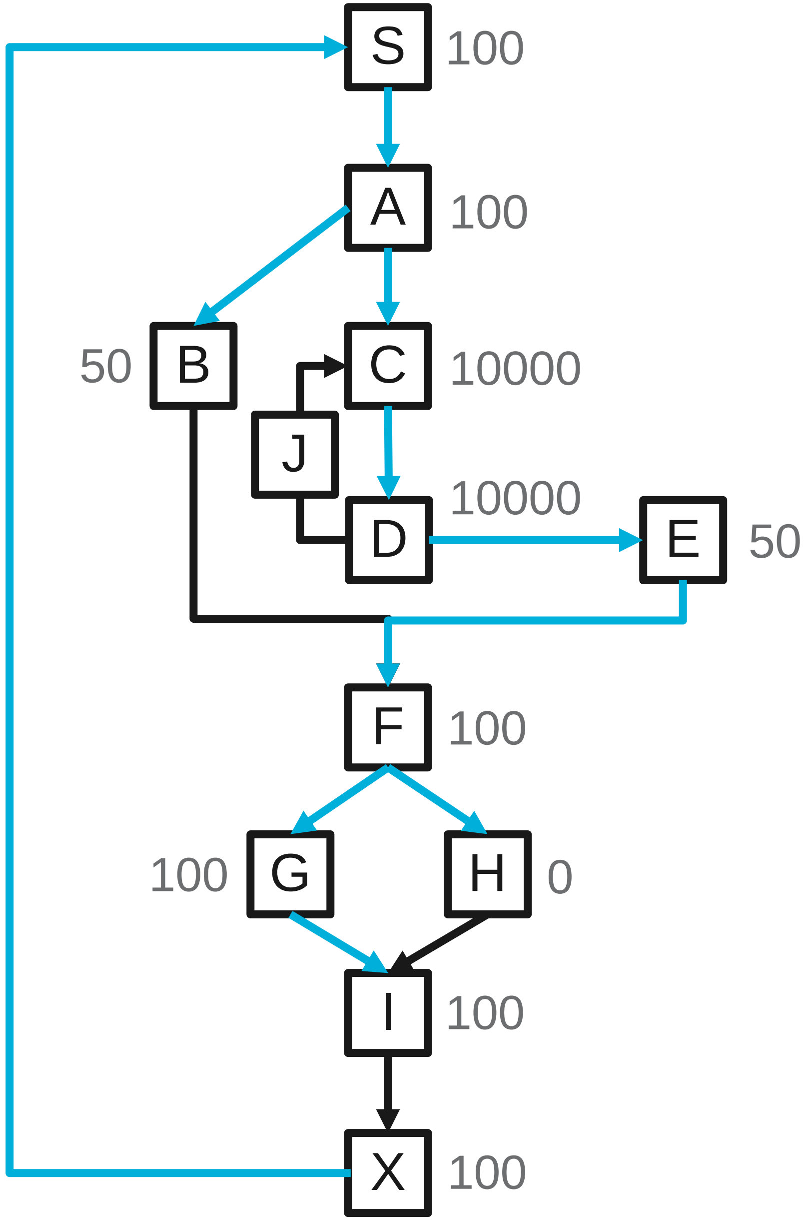
X→S
S→A
A→B
A→C
C→D
F→G
G→I
F→H

Maximum Spanning Tree Formulation



X→S
S→A
A→B
A→C
C→D
F→G
G→I
F→H

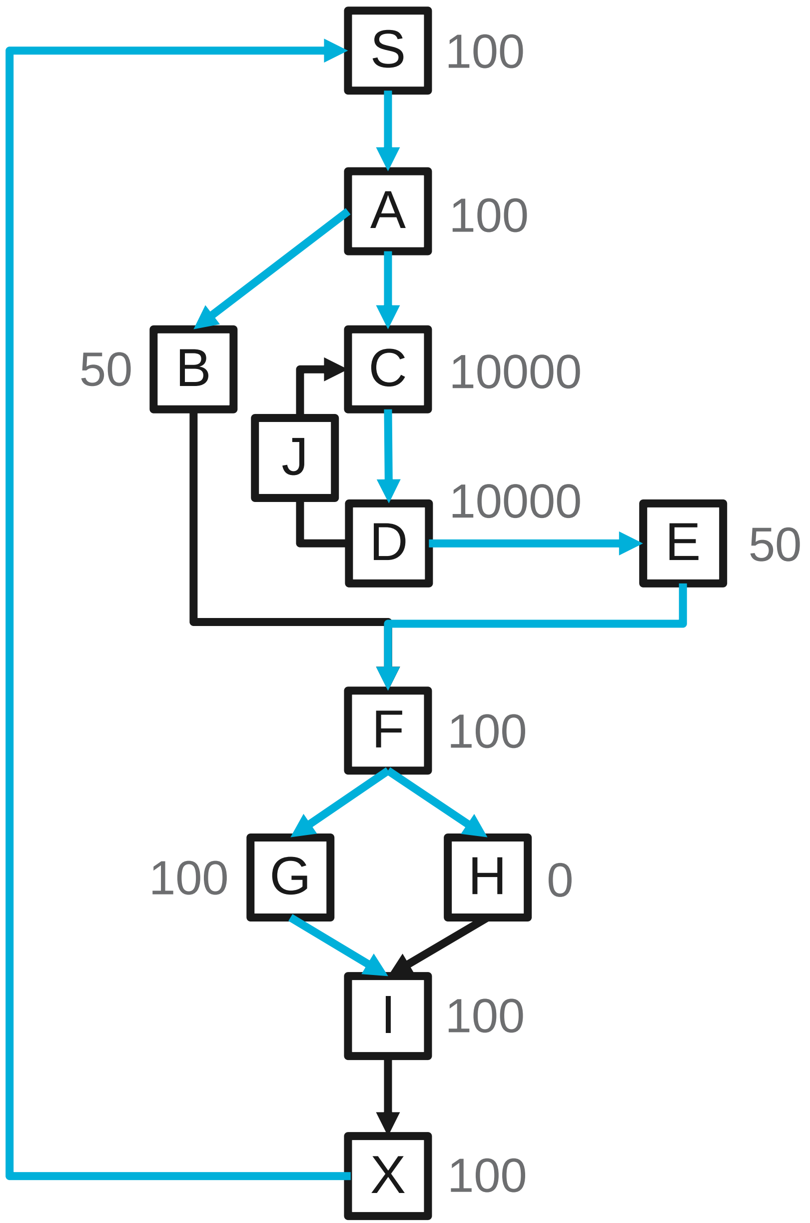
Maximum Spanning Tree Formulation



X→S
S→A
A→B
A→C
C→D
F→G
G→I
F→H

S	
A	
B	B
C	
D	
E	
F	
G	
H	H
I	I
J	J
X	

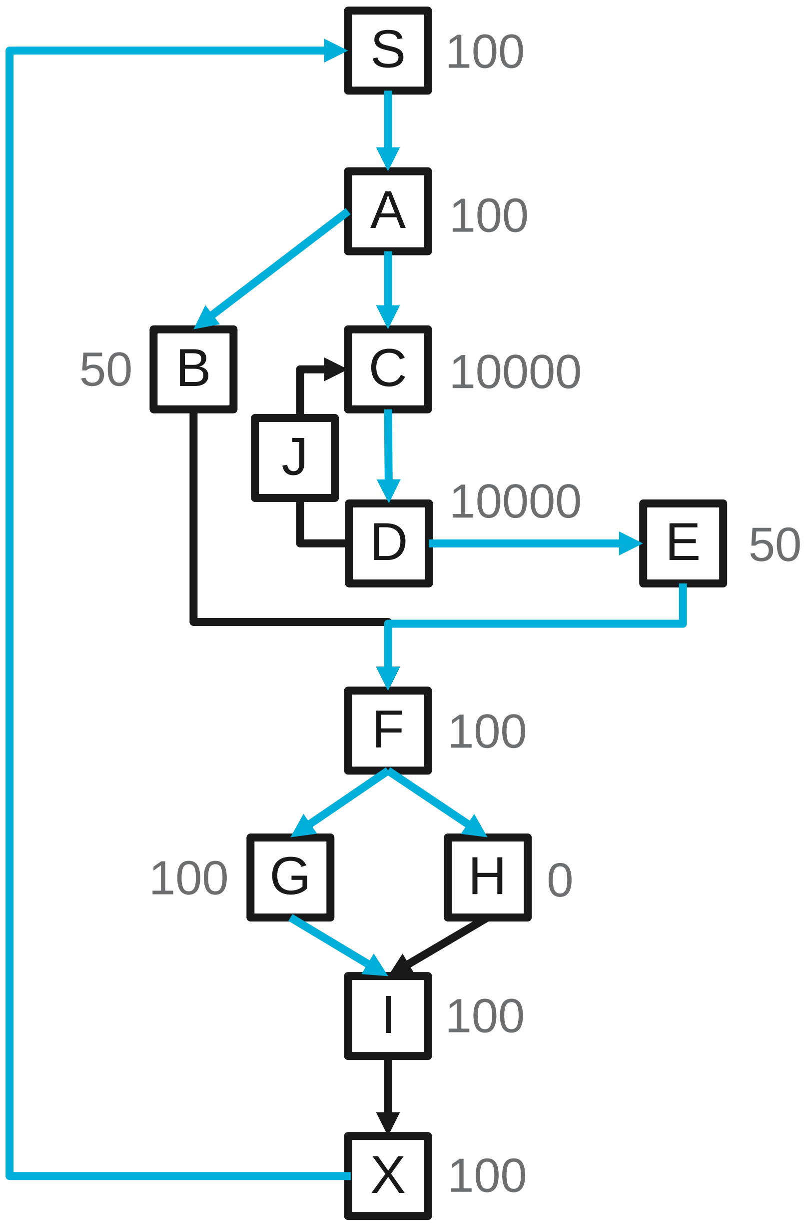
Maximum Spanning Tree Formulation



X→S
S→A
A→B
A→C
C→D
F→G
G→I
F→H

S	
A	
B	B
C	
D	
E	
F	
G	
H	H
I	I
J	J
X	I

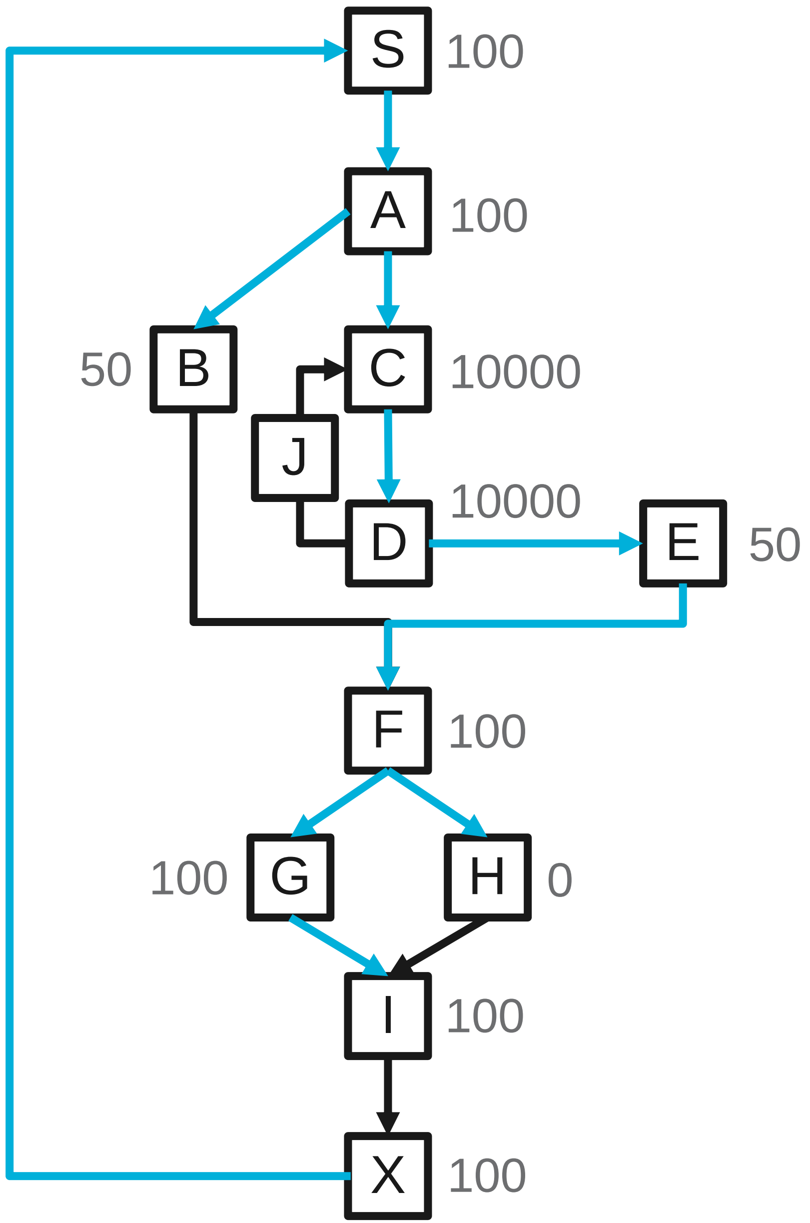
Maximum Spanning Tree Formulation



X→S
S→A
A→B
A→C
C→D
F→G
G→I
F→H

S	I
A	I
B	B
C	I-B+J
D	
E	
F	
G	
H	H
I	I
J	J
X	I

Maximum Spanning Tree Formulation



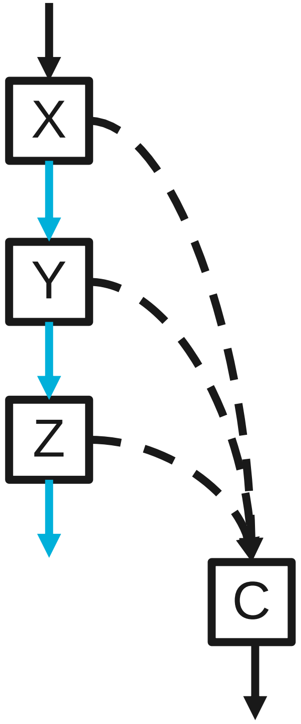
X→S
S→A
A→B
A→C
C→D
F→G
G→I
F→H

S	I
A	I
B	B
C	I-B+J
D	I-B+J
E	I-B
F	I
G	I-H
H	H
I	I
J	J
X	I

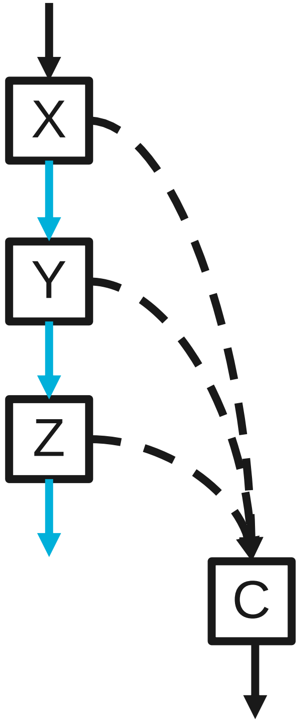
Exception Edges

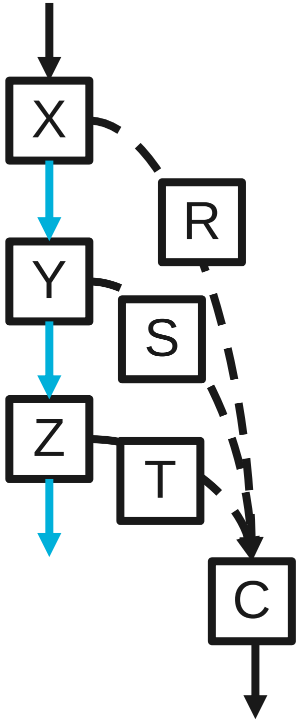
- OMR basic blocks – permit exception edge departures mid-block
- Exception edge represents non-local changes in flow-of-control
- Splitting an exception edge very hard
- Observation: exceptions are rare and usually do not occur
- Simplifying assumption: unify all counters for exception edges to a given catch block – just count the catch block
- Can detect when there is imprecision and can recompile and reprofile

Exception Edges



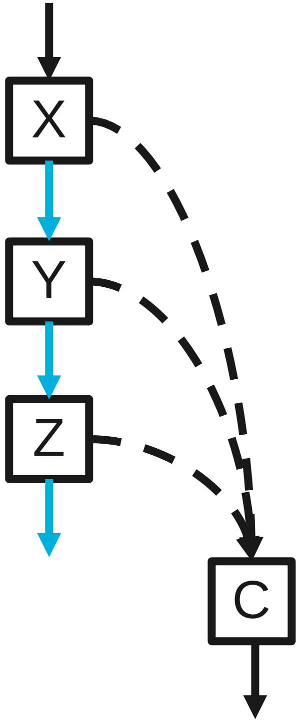
Exception Edges

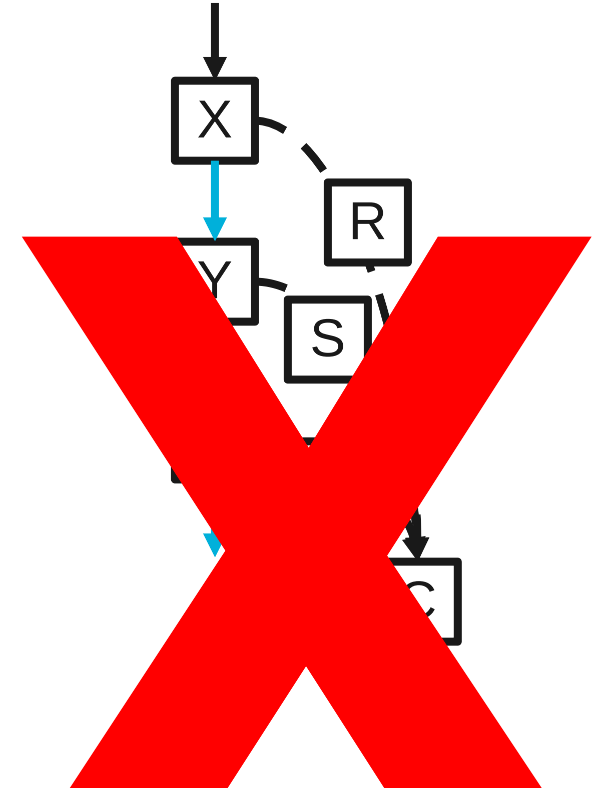




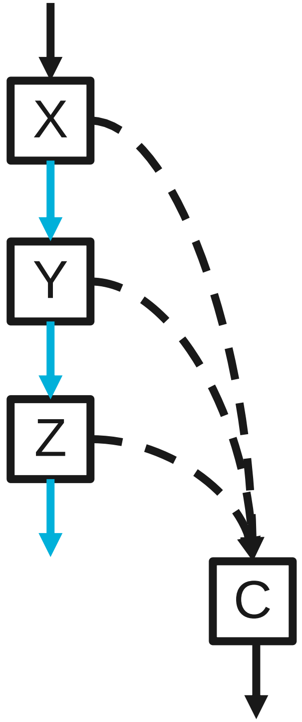
X	X
Y	X – R
Z	X – R – S
C	R + S + T

Exception Edges





[illegible]	X
Y	X – R
Z	X – R – S
C	R + S + T



X	X
Y	X – C
Z	X – C
C	C

Counter Implementation

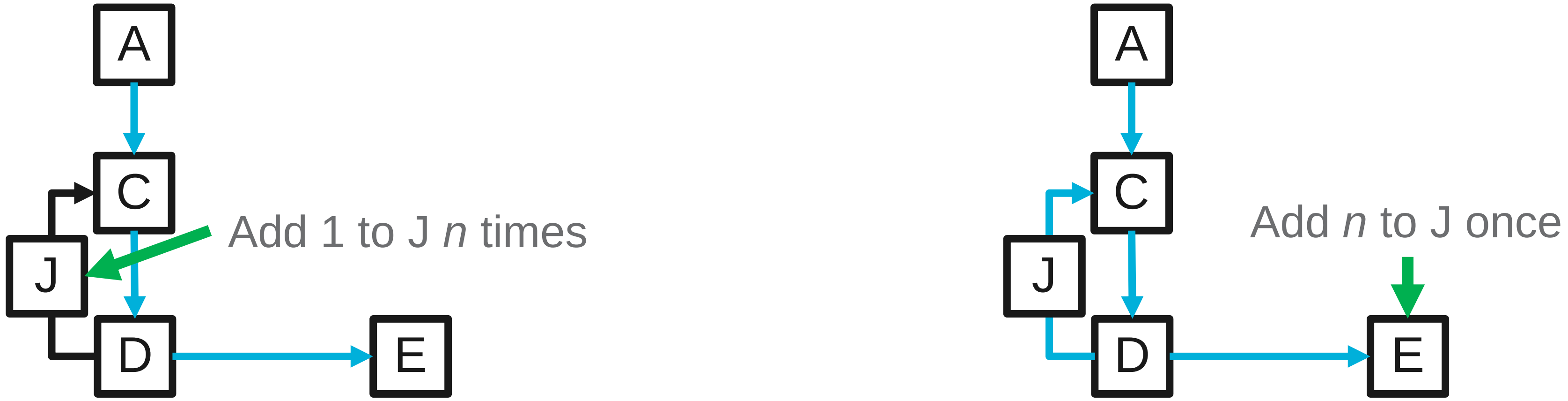
- Non-atomic updates – experiments forcing maximum contention showed deltas of at most 20% (did not change optimizer decisions)

- Instruction sequence:
  - X86: direct memory add of a constant
  - POWER: materialize address, load, add, store
  - Z Systems: materialize address to register, add to memory

- Patch in/out memory update – most expensive part
  - Store update instruction in metadata to write back in when needed
  - Align instructions for runtime patching (platform specific needs)

Counter Optimization

- Counters placed early in compilation – reflects ilgen state
- Optimization may co-locate counters – can de-duplicate if two counters always occur together
- Can optimize back-edge counting in counted loops:



- Active area of investigation to further reduce profiling overhead

Counter Consumption & Inlining

- Inlining can change between compilations
- If we inline a method body not profiled how do we set meaningful block frequencies?
- Idea: check for other compiled bodies with the same call chain otherwise use IProfiler data
- Example:
  given call sequence `operation` → `implementation` → `details`
  if we did not previously inline implementation into operation look for:
  `implementation` → `details`
  `details`

Performance Results

- DayTrader 3 – flat profile of many methods (~10000 warm compiles)
  - Counters enabled for all blocks in all methods: -10% throughput
  - Counters disabled for all blocks in all methods: <-0.5% throughput
  - Footprint increase from counter placement: <15%
- Penalties currently higher for high opt compiles with loops – counter increments can disrupt optimizations using pattern matching
- Above results do NOT include counter optimization – many examples of multiple counters incrementing in a row, no counted loop counter optimizations etc

Status

- Implementation available in a disabled-by-default state in Eclipse OpenJ9
- Working to add lightweight value profiling
- Goal: replace current profiling infrastructure & boost startup
- Hope to contribute to Eclipse OMR once complete for other runtimes

Q&A