

Compilerconstructie

najaar 2019

<http://www.liacs.leidenuniv.nl/~vlietrvan1/coco/>

Rudy van Vliet

kamer 140 Snellius, tel. 071-527 2876

rvvliet(at)liacs(dot)nl

college 10, vrijdag 6 december 2019

+ 'werkcollege'

Code Optimization (2)

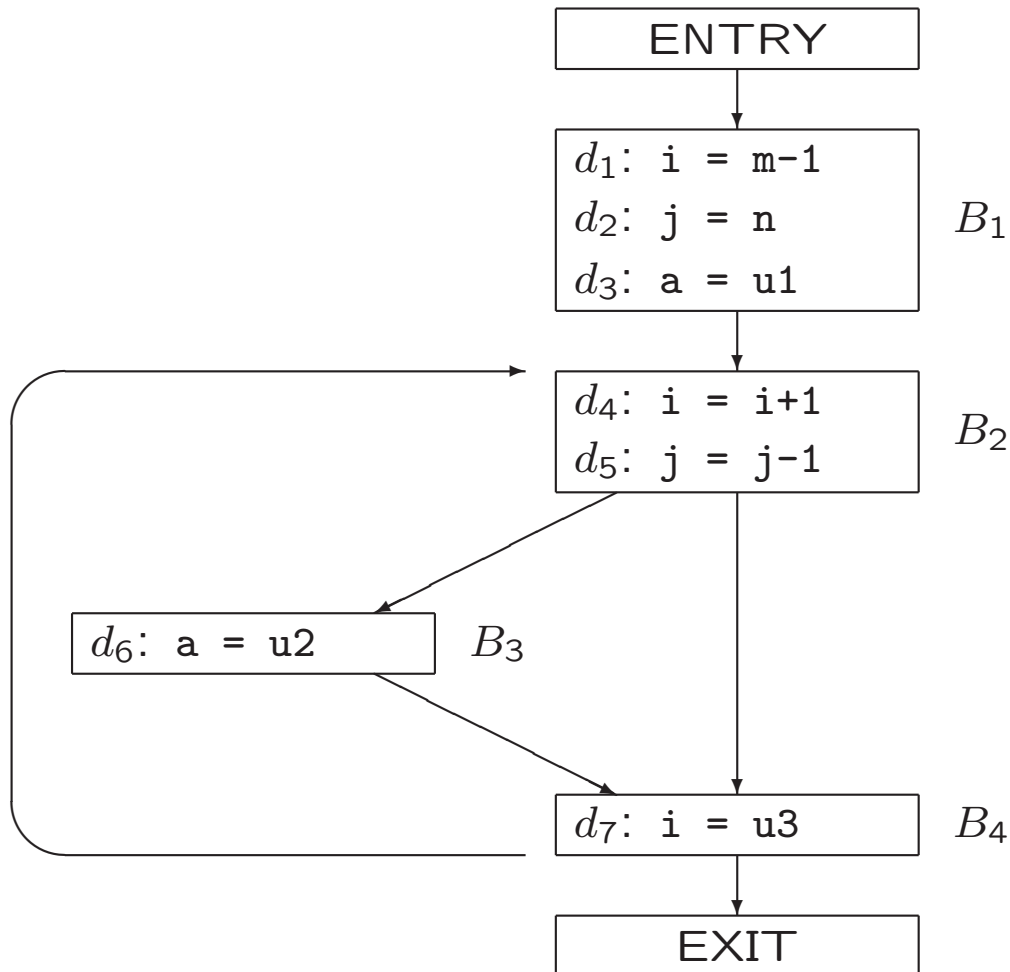
9.2 Introduction to Data-Flow Analysis

- Optimizations depend on **data-flow analysis**, e.g.,
 - Global common subexpression elimination
 - Dead-code elimination
- **Execution path** yields program state at **program point**
- Extract information from program state for data-flow analysis
- Usually infinite number of execution paths / program states
- Different analyses extract different information

Data-Flow Analysis (Examples)

- **Reaching definitions:** which definitions (assignments of values) of variable x **may** reach program point?
 - Useful for debugging:
May variable x be undefined?
 - Useful for constant folding:
Can variable x only have one, constant value at program point?

9.2.4 Computing Reaching Definitions



Reaching definitions

- Before B_1 : \emptyset
- After B_1 : $\{d_1, d_2, d_3\}$
- Before B_2 : \dots
- After B_2 : \dots

9.2.2 The Data Flow Analysis Schema

Data flow values

- $IN[s]$: before statement s
- $OUT[s]$: after statement s
- Transfer function f_s
 - forward: $OUT[s] = f_s(IN[s])$
 - backward: $IN[s] = f_s(OUT[s])$

Computing Reaching Definitions

- Effect of single definition $d : u = v \text{ op } w$:
 - $\text{OUT}[d] = \{d\} \cup (\text{IN}[d] - \dots)$

Computing Reaching Definitions

Effect of single definition $d : u = v \text{ op } w$:

- $\text{OUT}[d] = \{d\} \cup (\text{IN}[d] - \{\text{all other definitions of } u \text{ in program}\})$

- Hence,

$$\begin{aligned} f_d(x) &= \{d\} \cup (x - \{\text{all other definitions of } u \text{ in program}\}) \\ &= \text{gen}_d \cup (x - \text{kill}_d) \end{aligned}$$

where

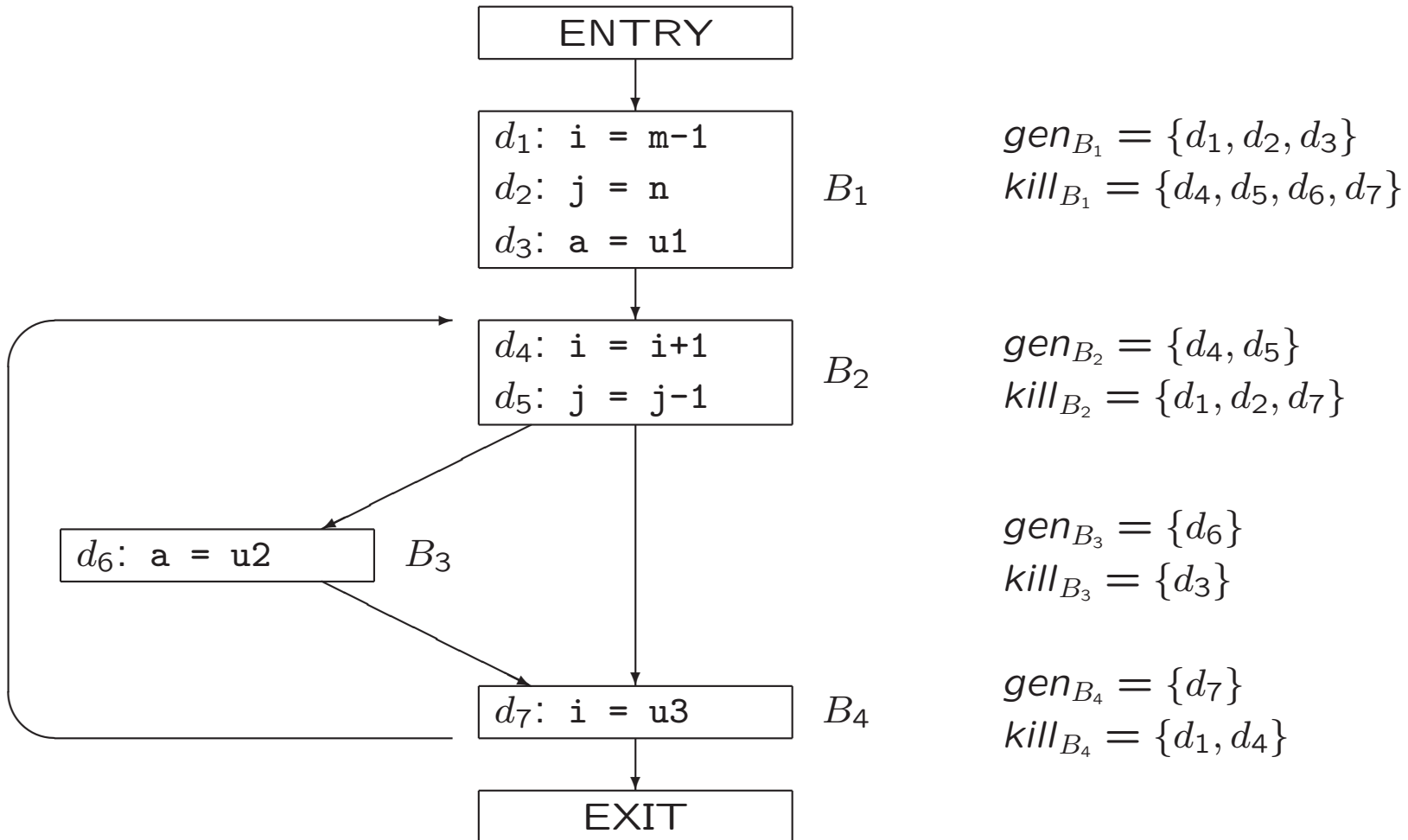
$$\begin{aligned} \text{gen}_d &= \{d\} \\ \text{kill}_d &= \{\text{all other definitions of } u \text{ in program}\} \end{aligned}$$

Computing Reaching Definitions

Effect of block B , with definitions $1, 2, \dots, n$:

$$\begin{aligned} gen_B &= \{n, n-1, \dots, 1\} - \{ \text{definitions killed afterwards} \} \\ &= gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \dots \\ kill_B &= kill_1 \cup kill_2 \cup \dots \cup kill_n \end{aligned}$$

Computing Reaching Definitions



In this case, $gen_B \dots$

Iterative Algorithm for Computing Reaching Definitions

```
OUT[ENTRY] =  $\emptyset$ 
for each basic block  $B$  other than ENTRY
    OUT[ $B$ ] =  $\emptyset$ 

while (changes to any OUT occur)
    for each basic block  $B$  other than ENTRY
    {   IN[ $B$ ] =  $\cup_{\text{predecessors } P \text{ of } B}$  OUT[ $P$ ]

        OUT[ $B$ ] =  $gen_B \cup (IN[ $B$ ] - kill_B)$ 
    }
```

Typical form of algorithm for forward data-flow analysis

\cup is **meet operator** (why \cup ?)

Example with $B = B_1, B_2, B_3, B_4, \text{EXIT} \dots$

Implementation of Iterative Algorithm for Computing Reaching Definitions...

```
OUT[ENTRY] =  $\emptyset$ 
for each basic block  $B$  other than ENTRY
    OUT[ $B$ ] =  $\emptyset$ 

while (changes to any OUT occur)
    for each basic block  $B$  other than ENTRY
    {   IN[ $B$ ] =  $\cup_{\text{predecessors } P \text{ of } B} \text{OUT}[P]$ 

        OUT[ $B$ ] =  $gen_B \cup (\text{IN}[B] - kill_B)$ 
    }
```

Implementation of Iterative Algorithm for Computing Reaching Definitions

With bit vectors

Block B	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	000 0000	001 0111	001 0111	001 0111

Iterative Algorithm for Computing Reaching Definitions

```
OUT[ENTRY] = ...  
for each basic block  $B$  other than ENTRY  
    OUT[ $B$ ] = ...  
  
while (changes to any OUT occur)  
    for each basic block  $B$  other than ENTRY  
    {   IN[ $B$ ] =  $\cup_{\text{predecessors } P \text{ of } B}$  OUT[ $P$ ]  
        OUT[ $B$ ] =  $gen_B \cup (IN[ $B$ ] - kill_B)$   
    }
```

Fixed point / steady state

Often, solution depends on initialization

9.2.5 Live-Variable Analysis

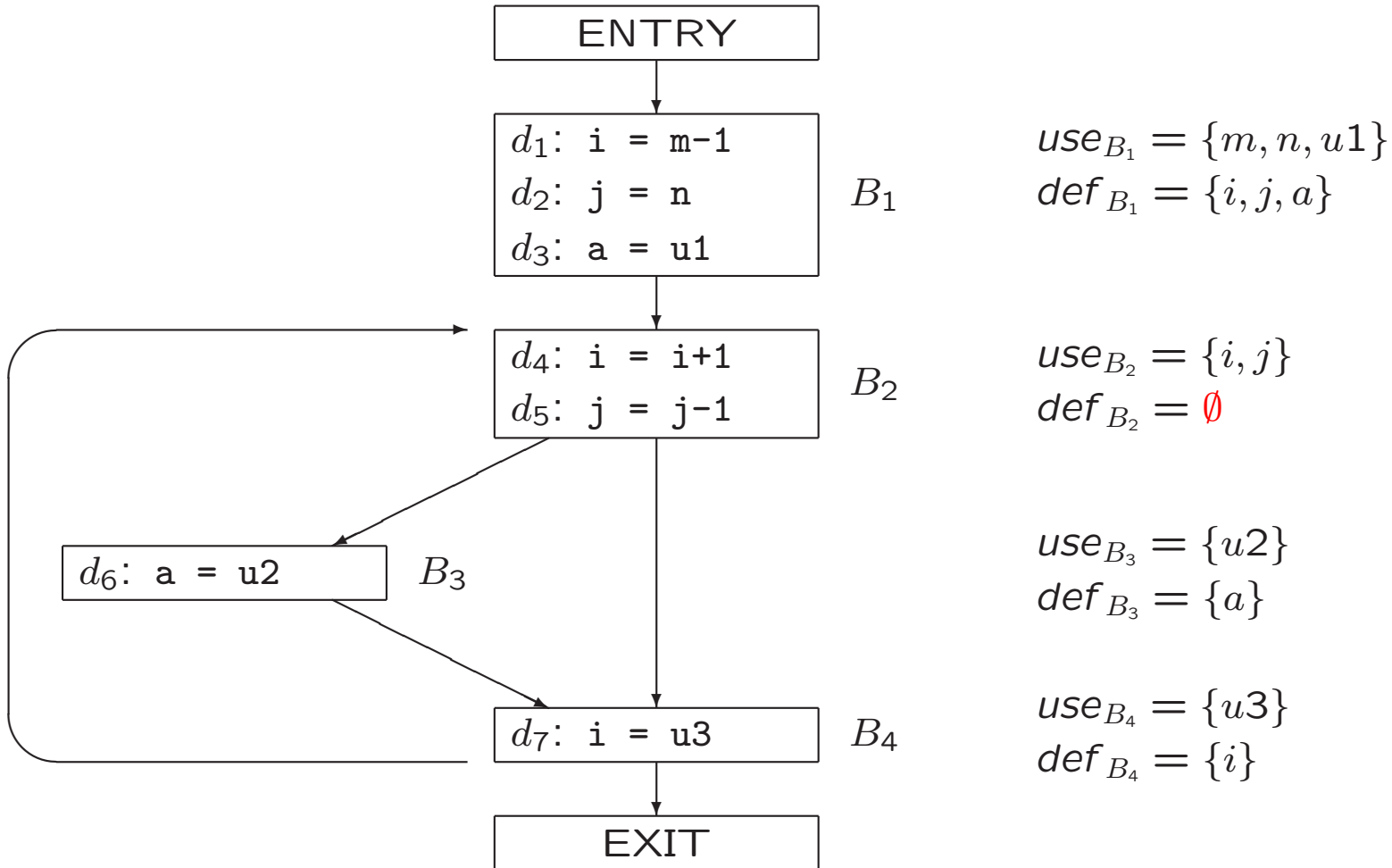
- Variable x is **live** at program point p ,
if value of x at p **could** be used later along *some* path
- Otherwise x is **dead** at p
- Information useful for register allocation (see lecture 7)
- Information about later use must be propagated backwards

Live-Variable Analysis

Effect of block B on live variables

- use_B :
variables that **may be used** in B prior to any definition in B
($\approx gen$)
- def_B :
variables that are **(definitely) defined** in B **prior to any use of that variable** in B
($\approx kill$)

Computing Live Variables



Iterative Algorithm to Compute Live Variables

$IN[EXIT] = \emptyset$

for each basic block B other than EXIT

$IN[B] = \emptyset$

while (changes to any IN occur)

for each basic block B other than EXIT

{ $OUT[B] = \cup_{\text{successors } S \text{ of } B} IN[S]$

$IN[B] = use_B \cup (OUT[B] - def_B)$

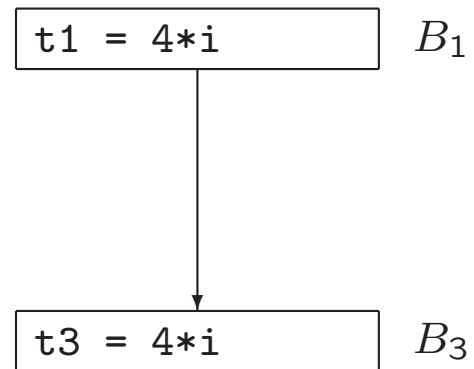
}

Typical form of algorithm for backward data-flow analysis

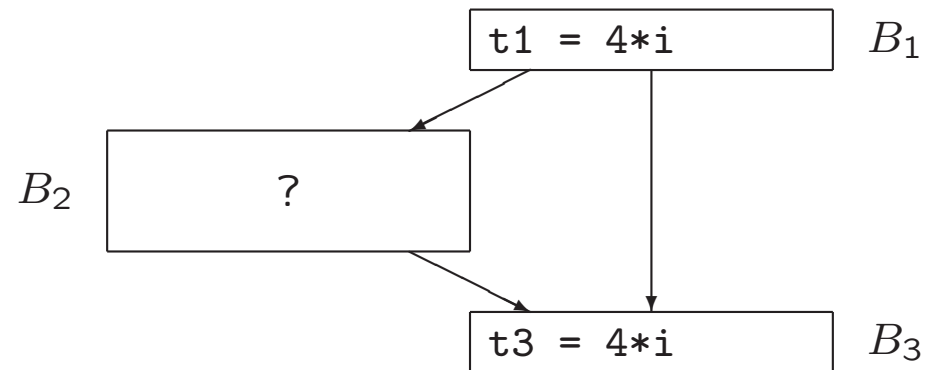
9.2.6 Available expressions

- **Is** (value of) expression $x \text{ op } y$ available?
- Useful for global common subexpression elimination
- Can be decided with data-flow analysis

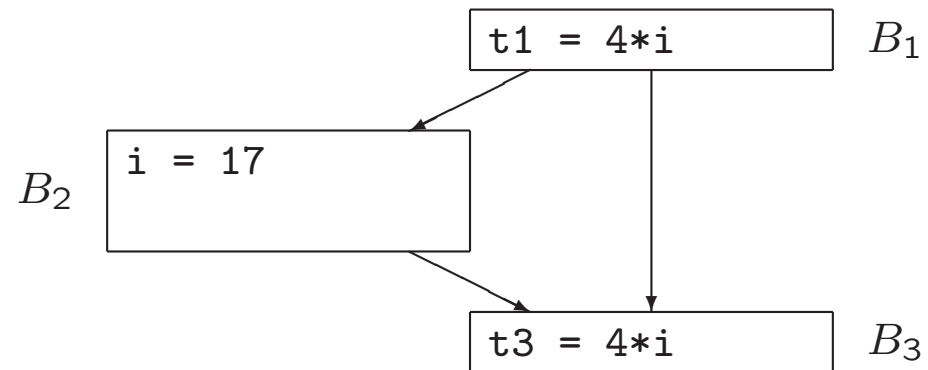
Available Expressions (Example)



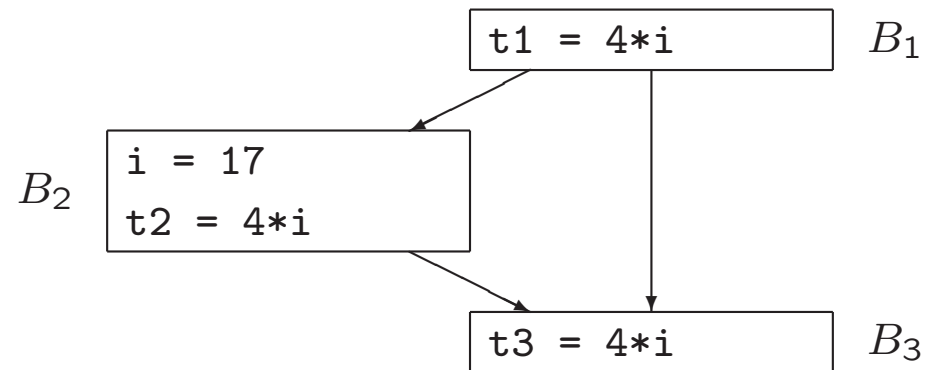
Available Expressions (Example)



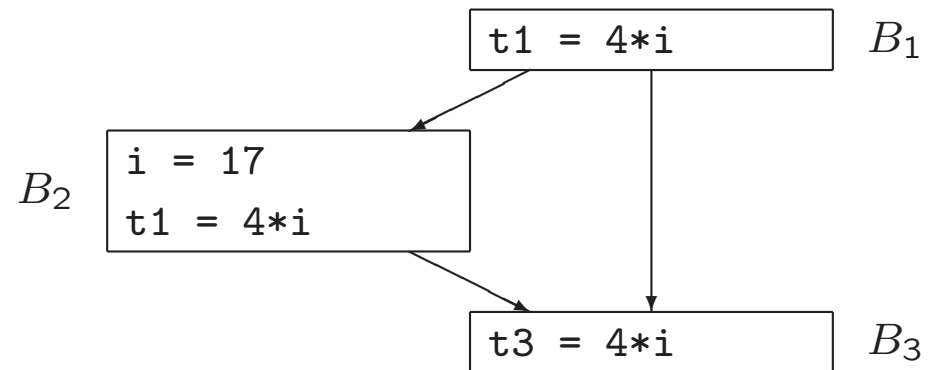
Available Expressions (Example)



Available Expressions (Example)



Available Expressions (Example)



Computing Available Expressions

Effect of block B on available expressions

- e_gen_B :
expressions $y \text{ op } z$ that are **definitely** computed in B ,
and for which y and z are not subsequently redefined
- e_kill_B :
expressions $y \text{ op } z$ for which y and/or z are **(or may be)**
defined in B ,
and that are not subsequently recomputed

Computing e_gen_B (Example)

$$S = \emptyset$$

For each statement $x = y \text{ op } z$ in block B (forwards)

- add $y \text{ op } z$ to S
- delete from S any expression involving x

Statement	Available Expressions S
	\emptyset
$a = b + c$	$\{b + c\}$
$b = a - d$	$\{a - d\}$
$c = b + c$	$\{a - d\}$
$d = a - d$	\emptyset

Computing Available Expressions

OUT[ENTRY] = \emptyset

for each basic block B other than ENTRY

 OUT[B] = U

while (changes to any OUT occur)

for each basic block B other than ENTRY

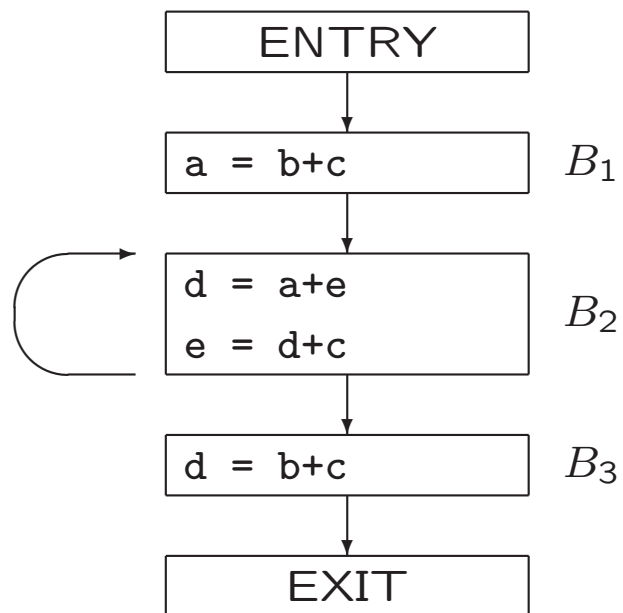
 { IN[B] = \bigcap predecessors P of B OUT[P]

 OUT[B] = $e_gen_B \cup (\text{IN}[B] - e_kill_B)$

 }

Why U ...

Available Expressions (Example)



Efficient Iterative Data-Flow Analysis

Example: computing reaching definitions

$OUT[ENTRY] = \emptyset$

for each basic block B other than ENTRY

$OUT[B] = \emptyset$

while (changes to any OUT occur)

for each basic block B other than ENTRY

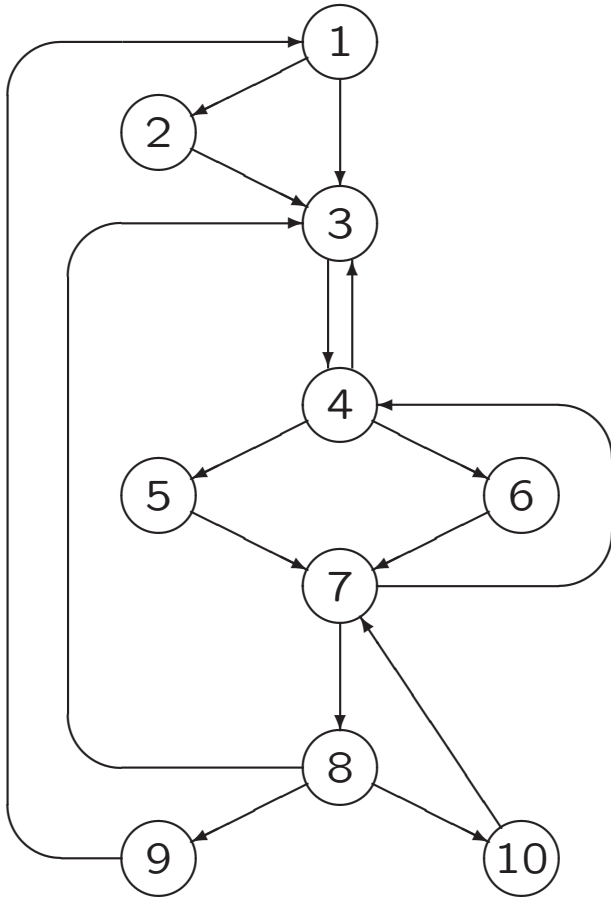
$IN[B] = \cup_{\text{predecessors } P \text{ of } B} OUT[P]$

$OUT[B] = gen_B \cup (IN[B] - kill_B)$

 }

Order of blocks in second for-loop matters

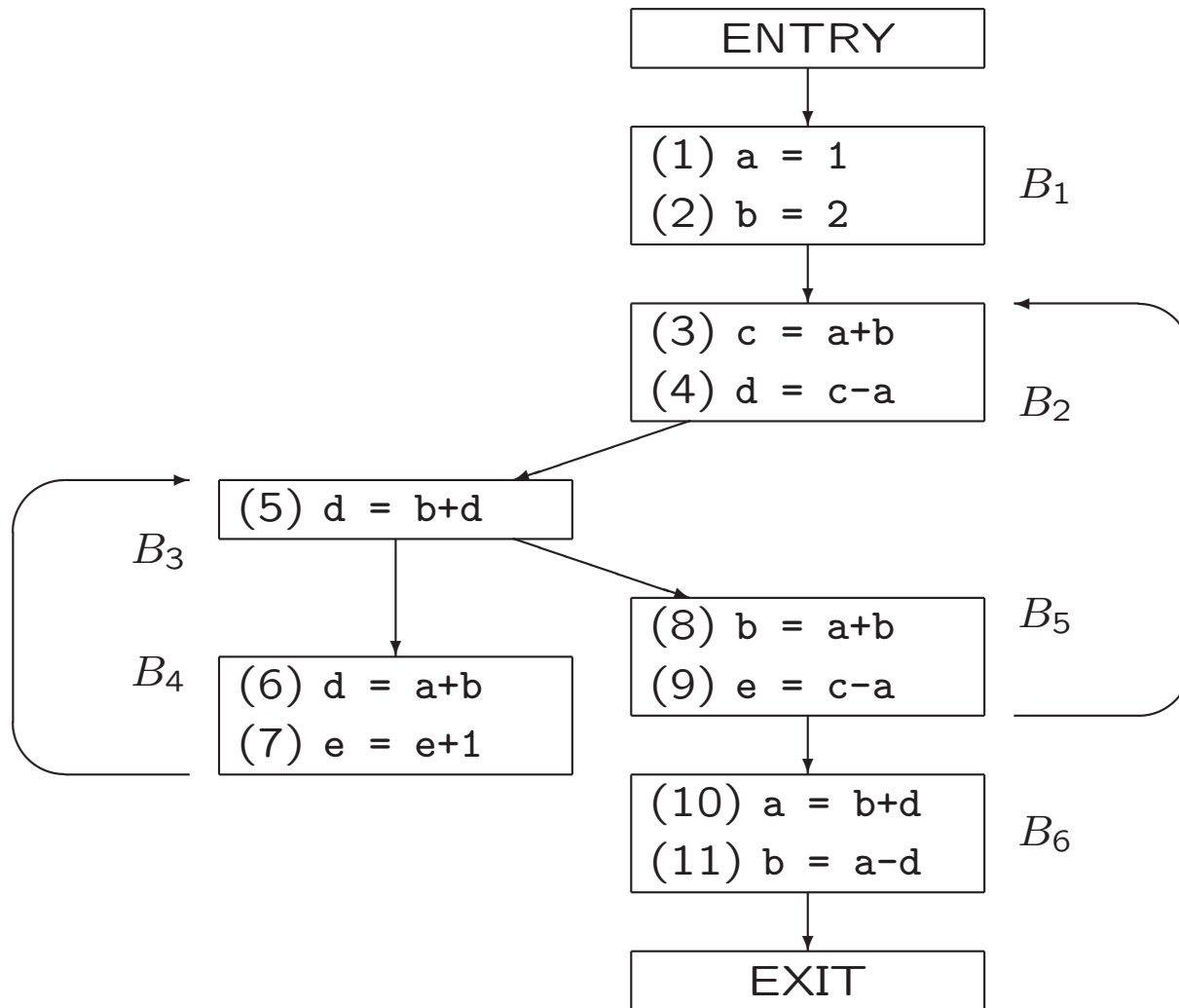
Efficient Iterative Data-Flow Analysis



Order of blocks in second for-loop matters

Exercises

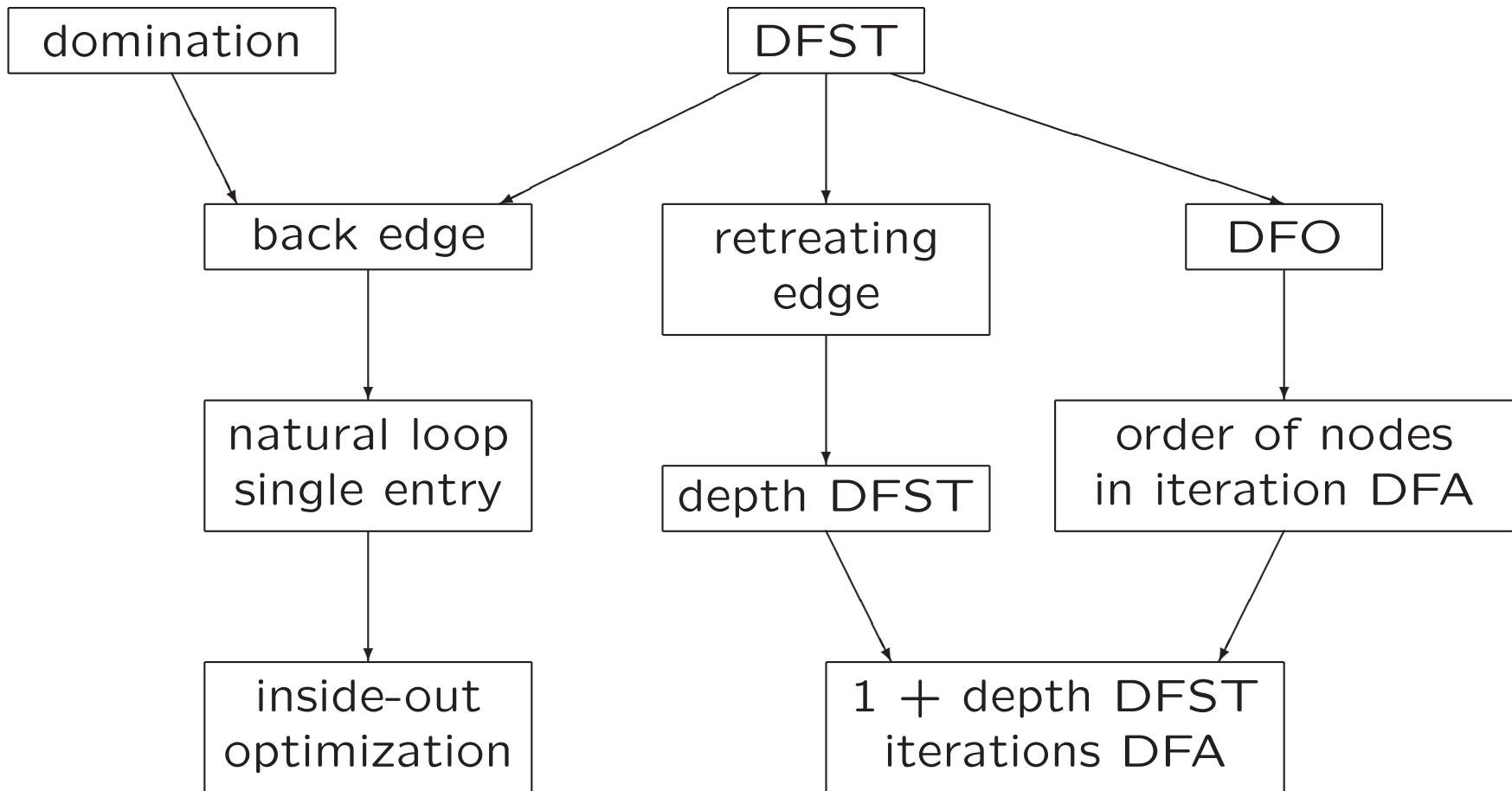
Flow Graph For Data Flow Analysis



9.6 Loops in Flow Graphs

- Optimizations of loops have significant impact
- Loops affect speed of convergence of iterative DFA
- Essential to identify loops
- Used in region based analysis (not for exam)

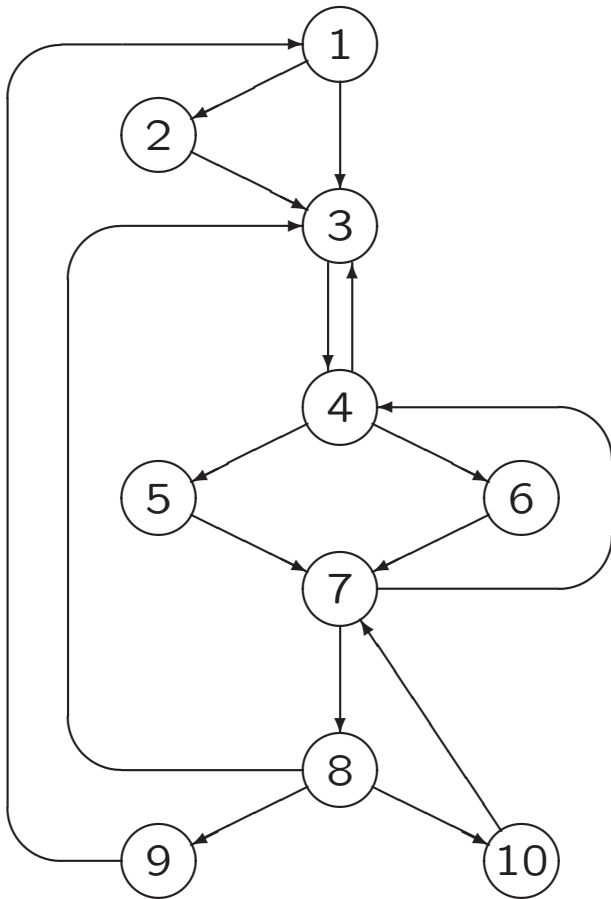
Flow Graph G



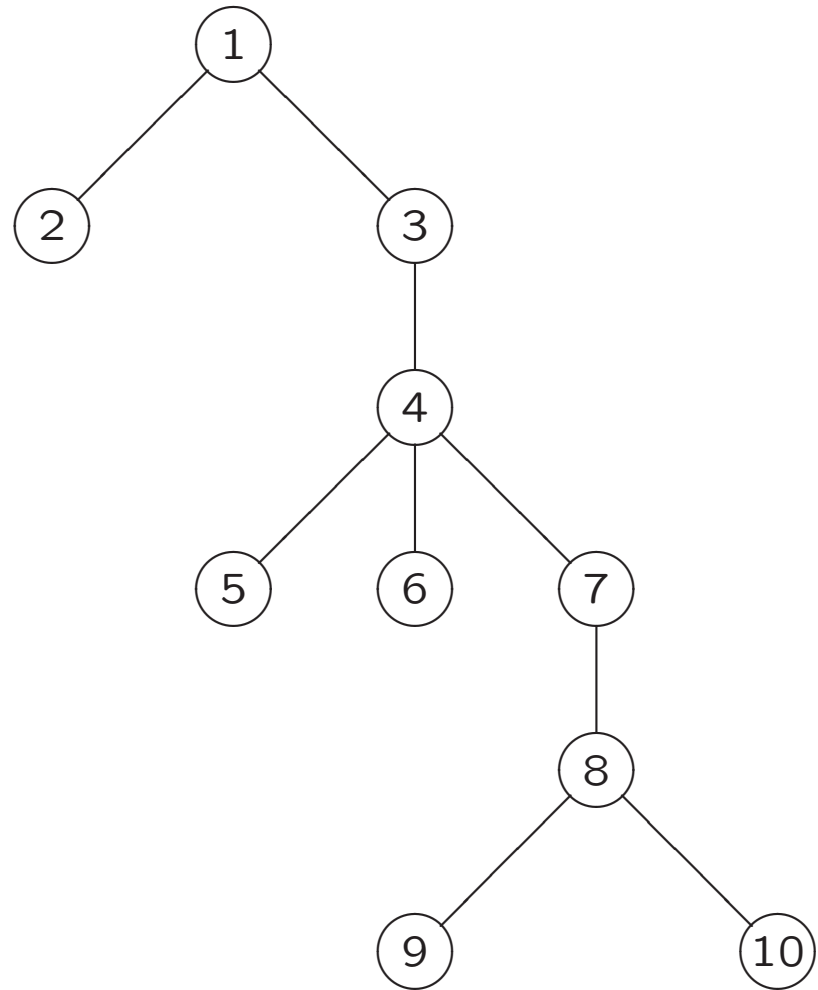
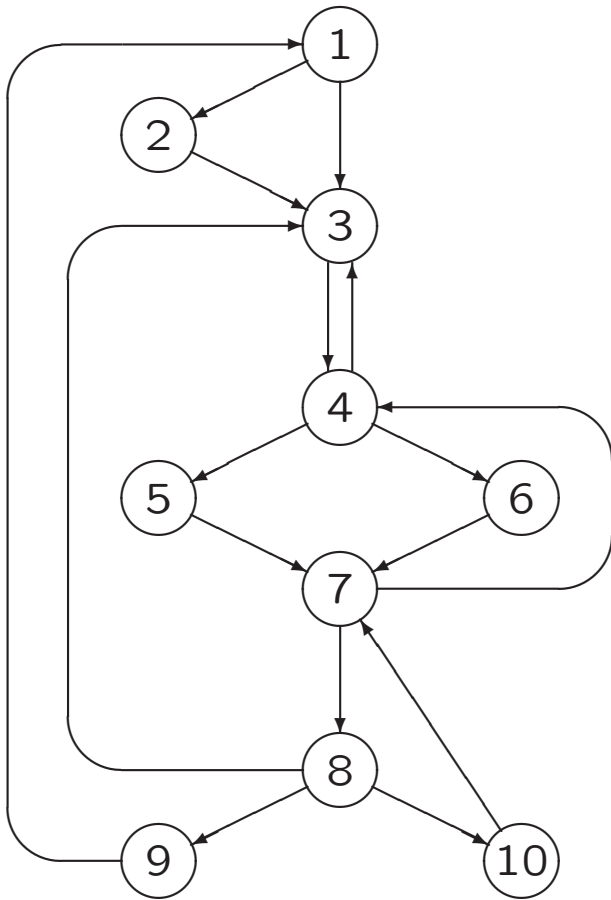
9.6.1 Dominators

- Dominators:
 - Node d dominates node n if every path from ENTRY node to n goes through d : $d \text{ dom } n$
 - Node n dominates itself
 - Loop entry dominates all nodes in loop
- Immediate dominator m of n :
last dominator on (any) path from ENTRY node to n
 - if $d \neq n$ and $d \text{ dom } n$, then $d \text{ dom } m$

Dominators (Example)



Dominator Trees (Example)



Finding Dominators

Forward data-flow analysis

N is set of all nodes

$OUT[ENTRY] = \{ENTRY\}$

for each node n other than ENTRY

$OUT[n] = N$

while (changes to any OUT occur)

for each node n other than ENTRY

{ $IN[n] = \bigcap_{\text{predecessors } m \text{ of } n} OUT[m]$

$OUT[n] = IN[n] \cup \{n\}$

}

Finding Dominators

Forward data-flow analysis

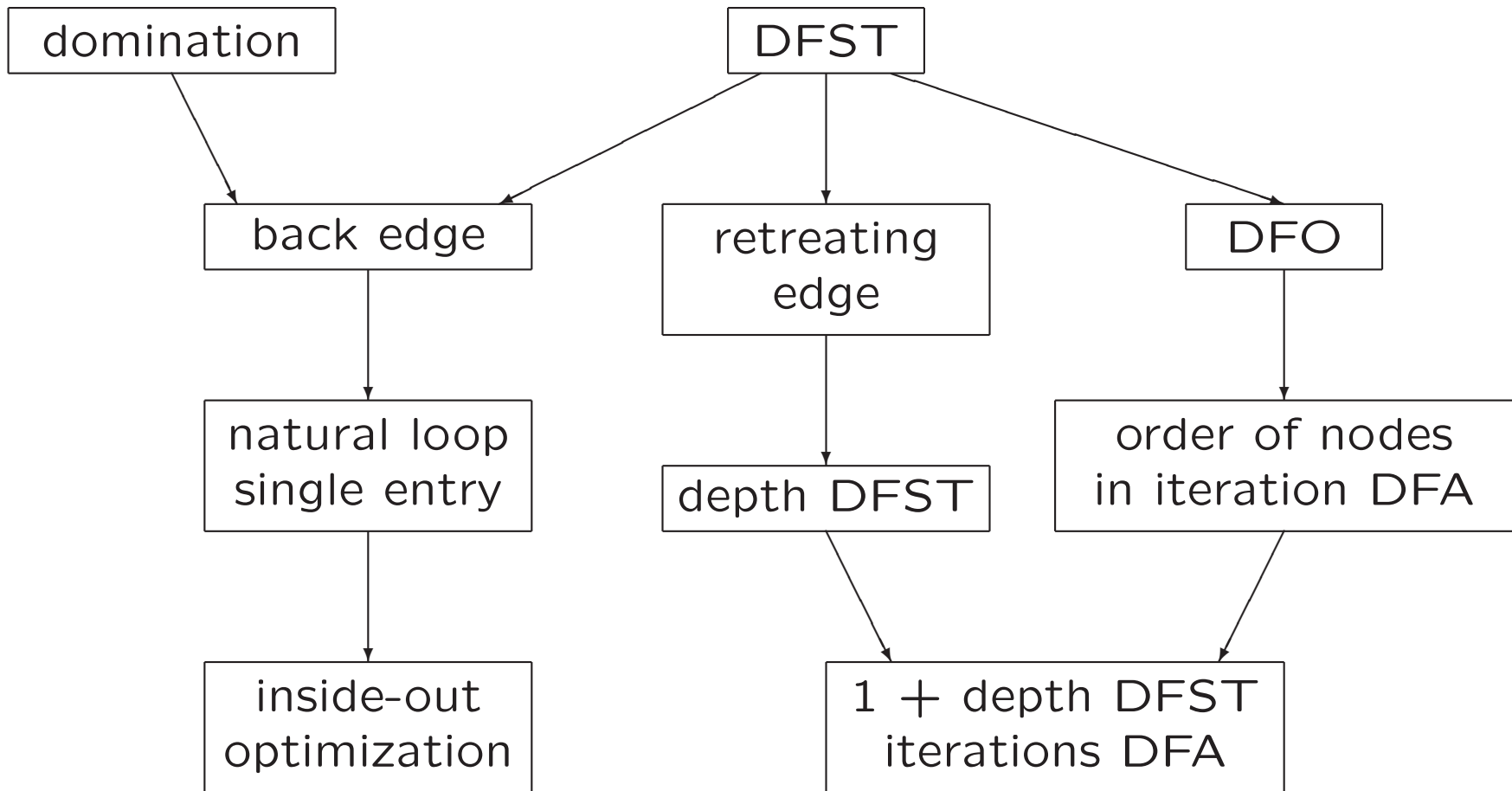
Incorrect different initialization...

```
OUT[ENTRY] = {ENTRY}
for each node  $n$  other than ENTRY
    OUT[ $n$ ] =  $\emptyset$ 

while (changes to any OUT occur)
    for each node  $n$  other than ENTRY
    {   IN[ $n$ ] =  $\cap_{\text{predecessors } m \text{ of } n}$  OUT[ $m$ ]

        OUT[ $n$ ] = IN[ $n$ ]  $\cup$  { $n$ }
    }
```

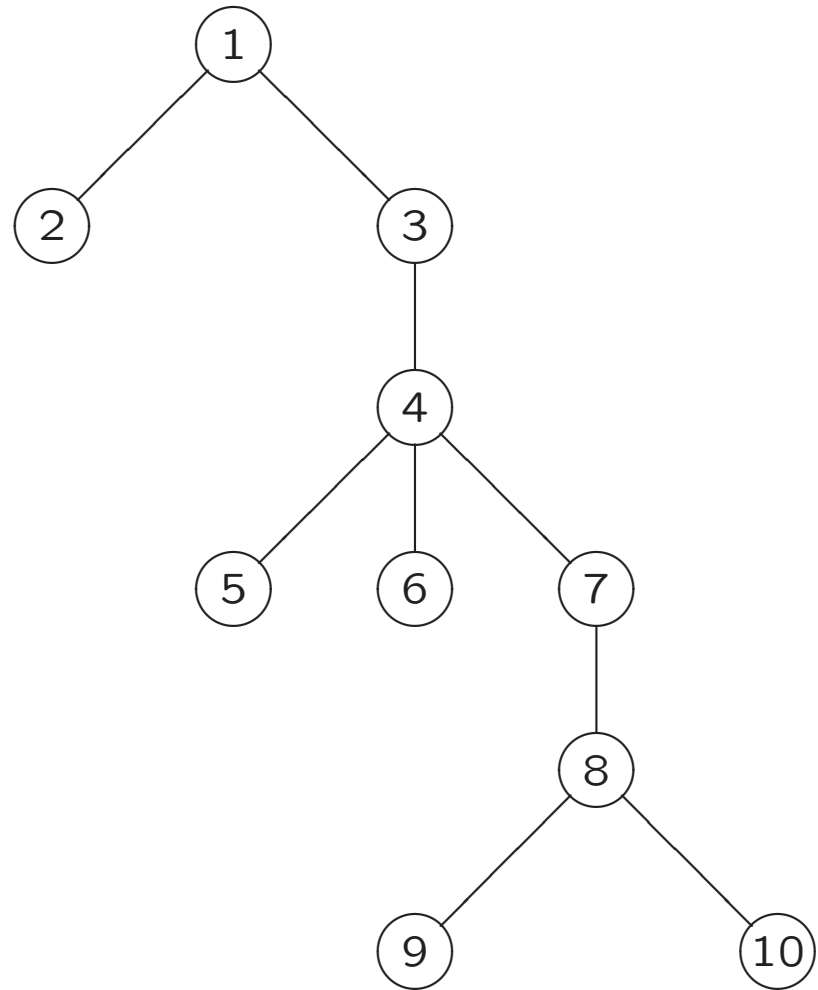
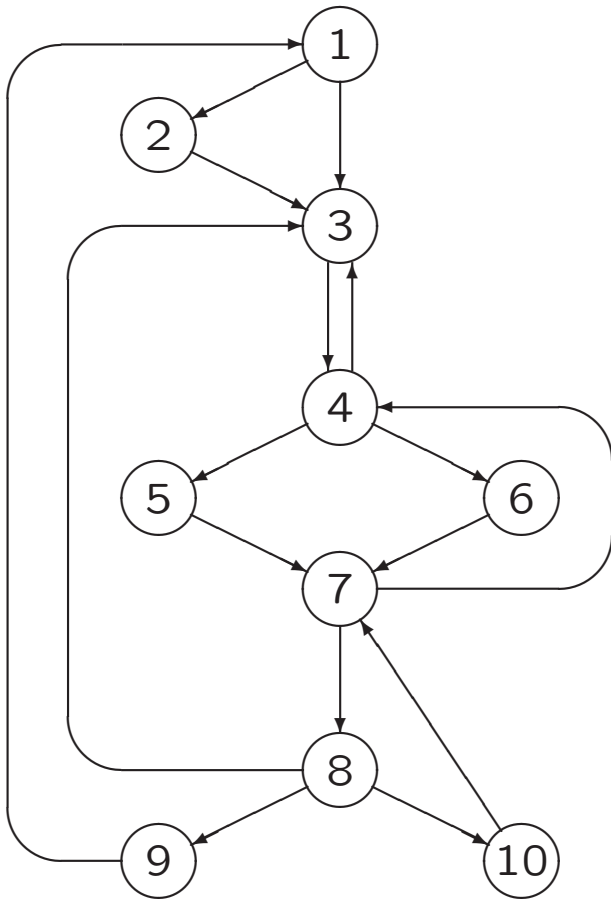
Flow Graph G



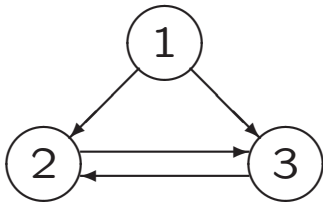
9.6.6 Natural loops

- **Back edge** $a \rightarrow b$, if b dominates a
- If loop has single-entry node, then compiler can assume certain initial conditions
- **Natural loop**
 1. Has single-entry node: **header**
 2. Has back edge to header
- Each back edge $n \rightarrow d$ determines natural loop, consisting of
 - d
 - all nodes that can reach n without going through d
- Constructing natural loop of back edge...

Dominator Trees (Example)



No Natural Loops



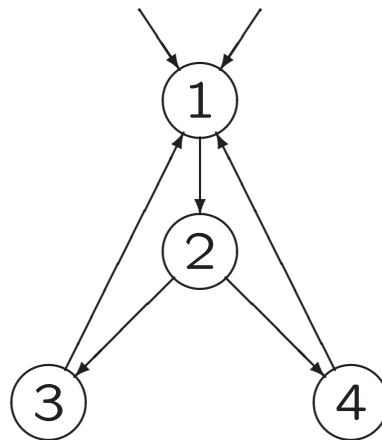
Natural Loops

- Useful property: unless two natural loops have same header
 - either they are disjoint
 - or one is nested within other

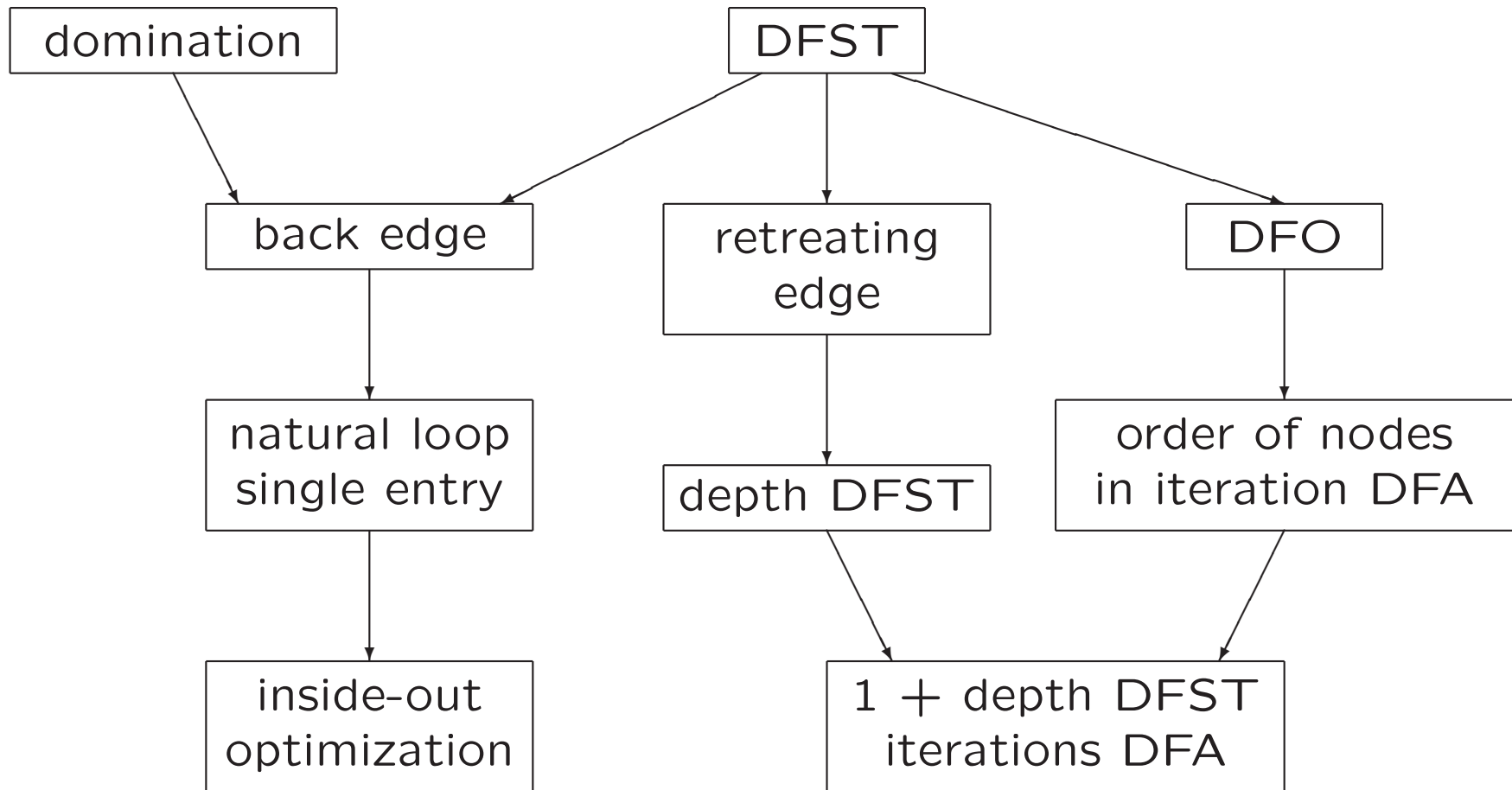
Allows for inside-out optimization

- Assumption: if necessary, combine natural loops with same header...

- Assumption: if necessary, combine natural loops with same header...



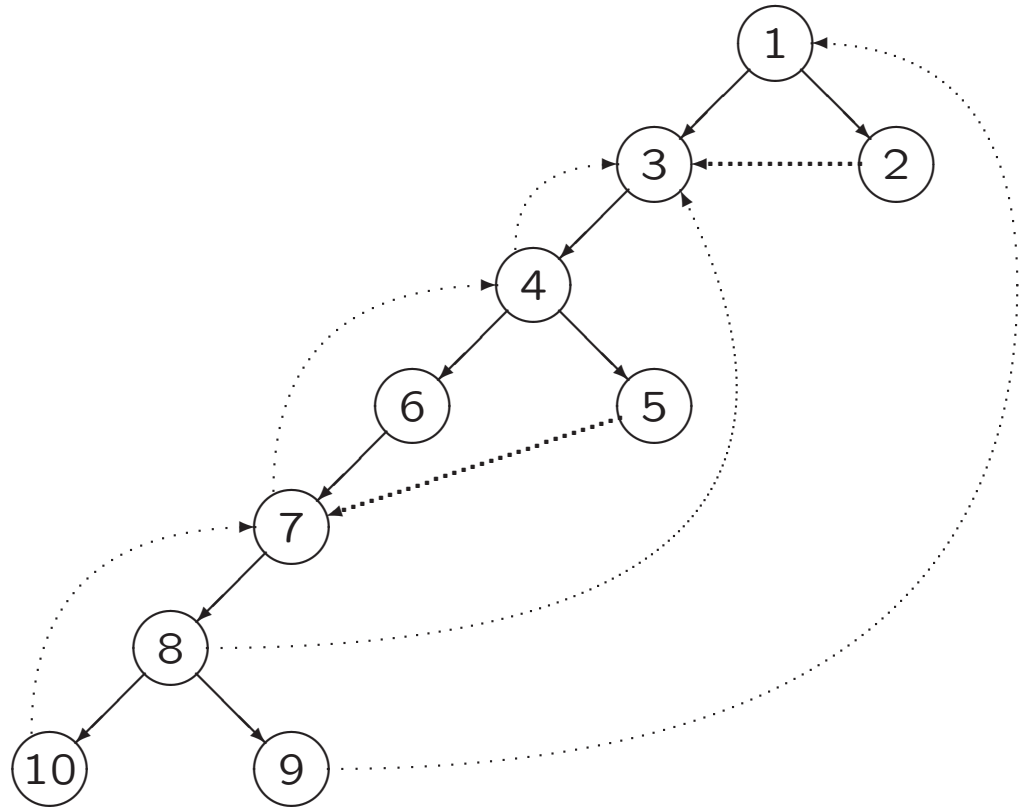
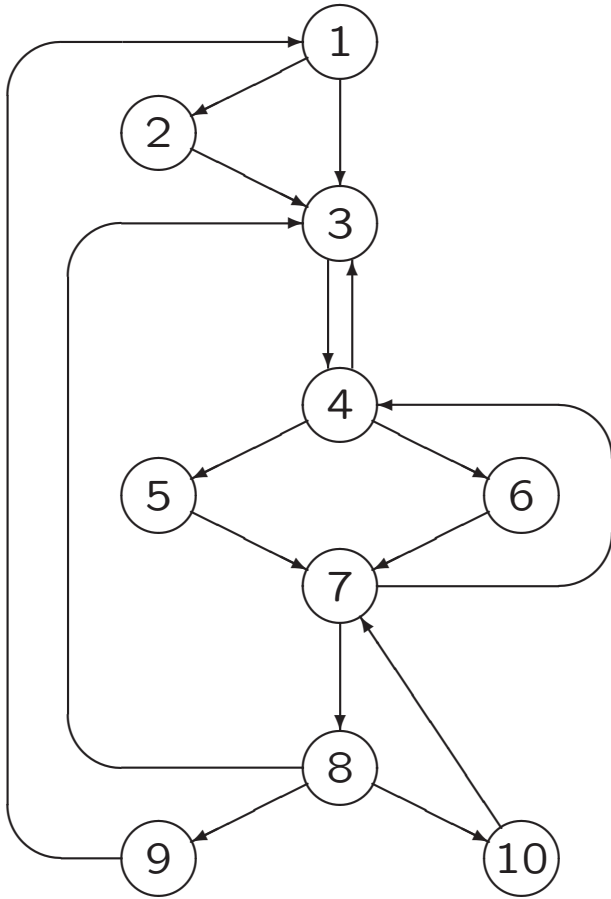
Flow Graph G



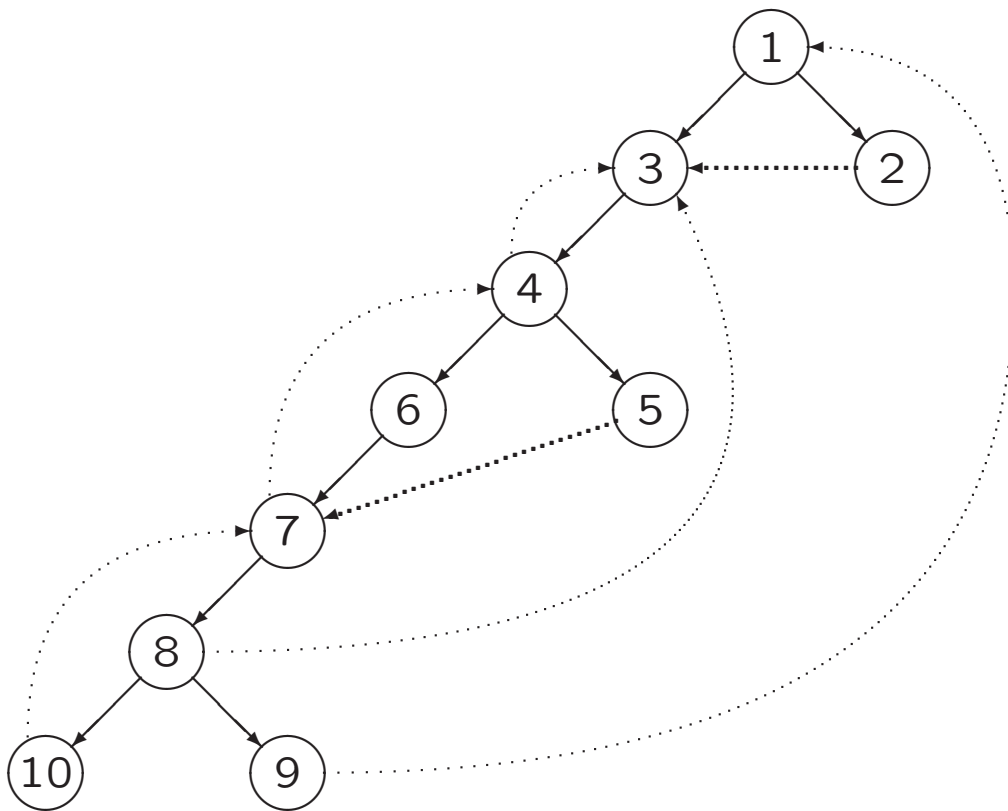
9.6.2 Depth-First Ordering

- Depth-first traversal of graph
 - Start from entry node
 - Recursively visit neighbours (in any order)
 - Hence, visit nodes far away from the entry node as quickly as it can (DF)

A Depth-First Spanning Tree



9.6.3 Edges in Depth-First Spanning Tree



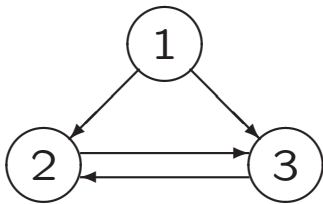
- Advancing edges
- Retreating edges
- Cross edges
- Back edge $a \rightarrow b$,
if b dominates a
(regardless of DFST)
- Each back edge is
retreating edge in DFST
- Flow graph is **reducible**,
if each retreating edge in
any DFST is back edge

9.6.3 Edges in Depth-First Spanning Tree

A different depth-first spanning tree...

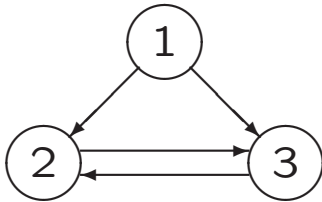
9.6.3 Edges in Depth-First Spanning Tree

Two different depth-first spanning trees...



9.6.4 Reducibility

- In practice, almost every flow graph is reducible
- Example of nonreducible flow graph (with advancing edges)

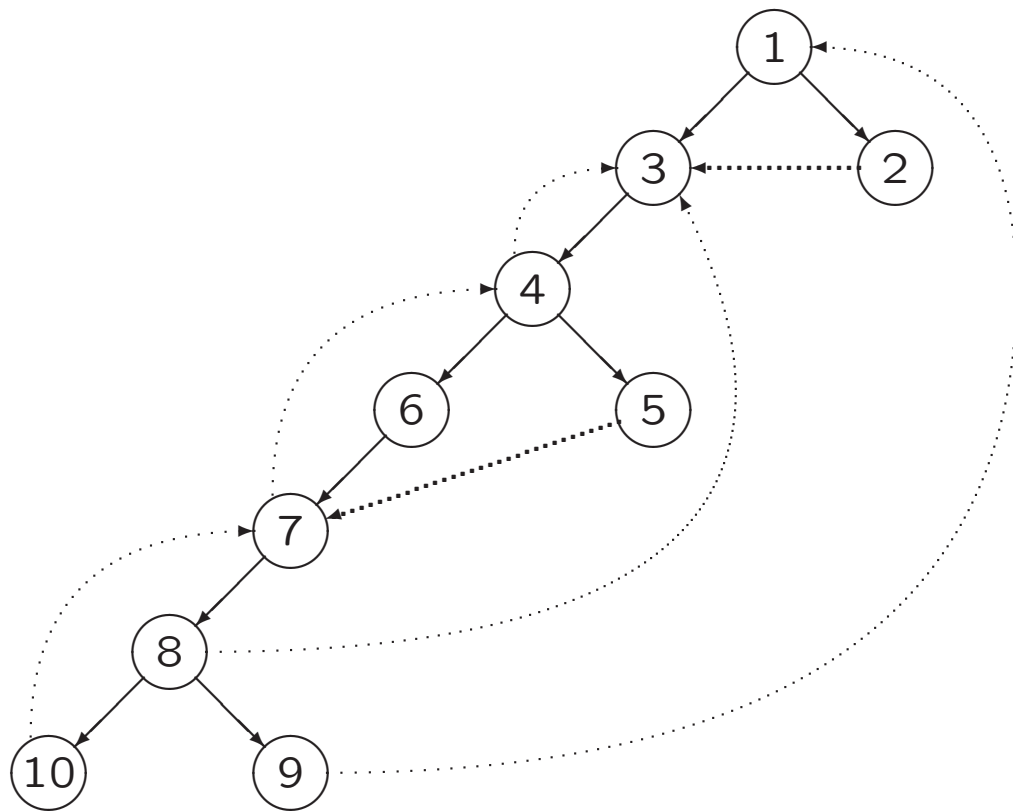


- To decide on reducibility:
 1. Remove back edges
 2. Is remaining graph acyclic?

9.6.5 Depth of Flow Graph

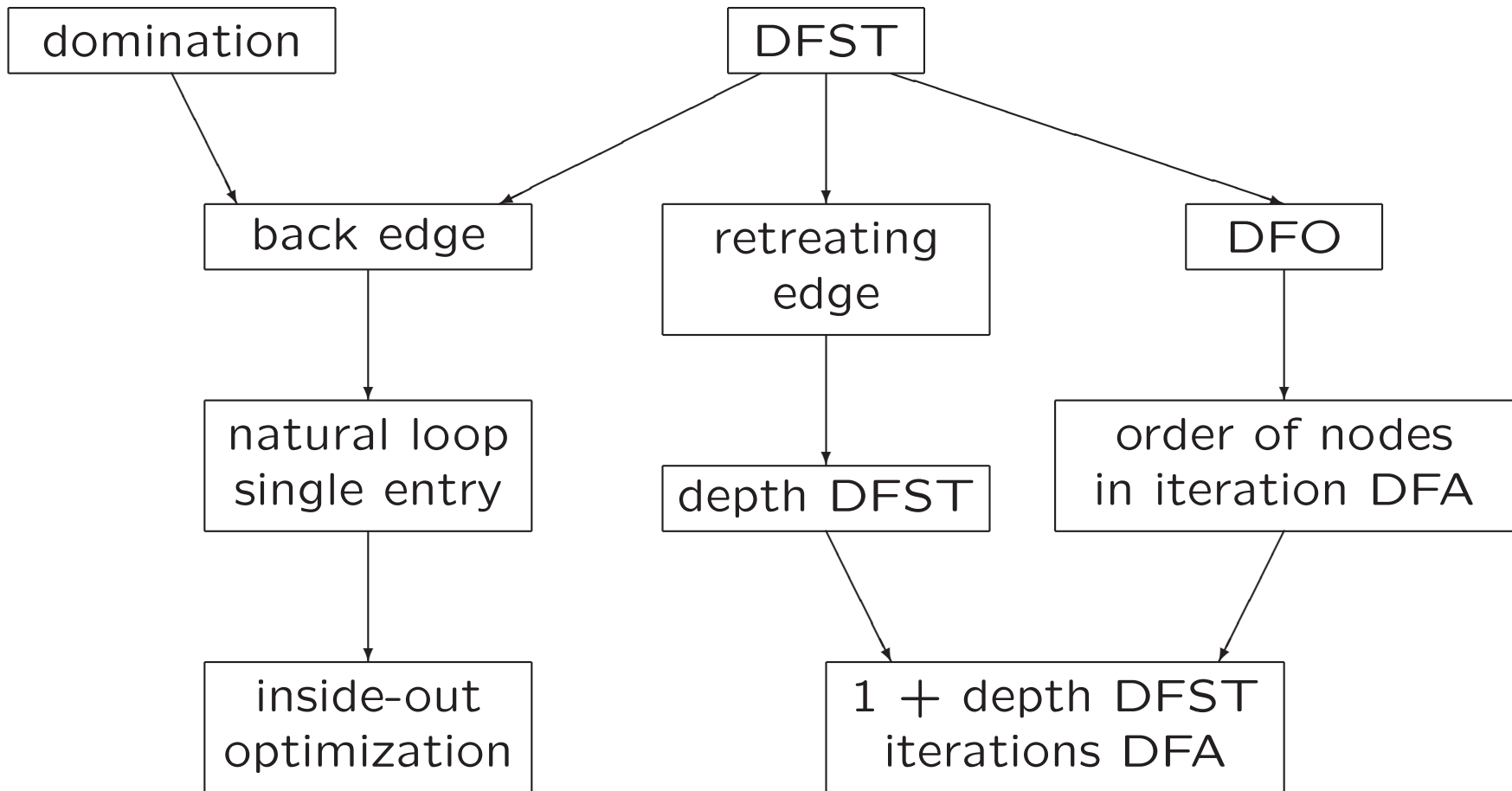
- **Depth** of DFST is largest number of retreating edges on any cycle-free path
- If flow graph is reducible, then depth is independent of DFST:
depth of flow graph
- $\text{Depth} \leq \text{depth of loop nesting in flow graph}$

Depth of Flow Graph (Example)

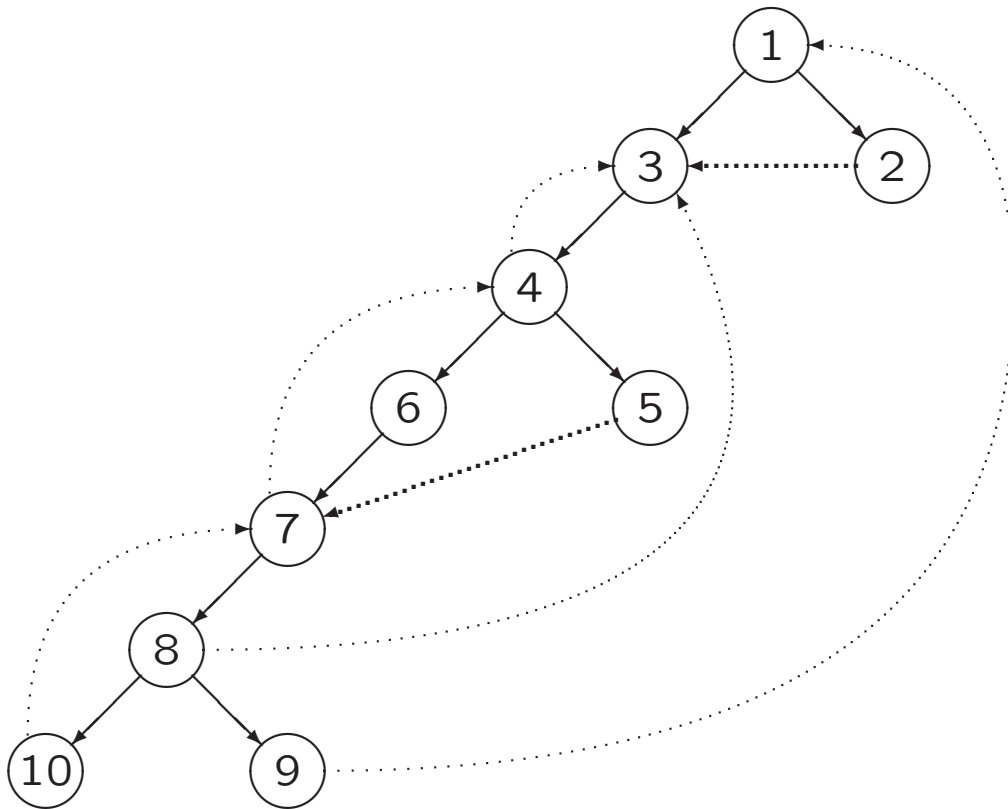


Depth is 3, because of path
 $10 \rightarrow 7 \rightarrow 4 \rightarrow 3$

Flow Graph G



9.6.2 **A** Depth-First Ordering



- **Depth-First Ordering:**
nodes in DFST in reverse of postorder
- Example:
1,2,3,4,5,6,7,8,9,10
- Edge $m \rightarrow n$ is retreating, if and only if n comes before m in depth-first ordering

9.6.7 Speed of Convergence of Iterative Data-Flow Algorithms

In data-flow analysis, can significant events be propagated to node along acyclic path?

- Yes for
 - Reaching definitions
 - Live-variable analysis
 - Available expressions
- No for
 - Copy propagation

If yes, then fast convergence possible

Efficient Iterative Data-Flow Analysis

Example: computing reaching definitions

$OUT[ENTRY] = \emptyset$

for each basic block B other than ENTRY

$OUT[B] = \emptyset$

while (changes to any OUT occur)

for each basic block B other than ENTRY

$IN[B] = \cup_{\text{predecessors } P \text{ of } B} OUT[P]$

$OUT[B] = gen_B \cup (IN[B] - kill_B)$

 }

Order of blocks in second for-loop matters

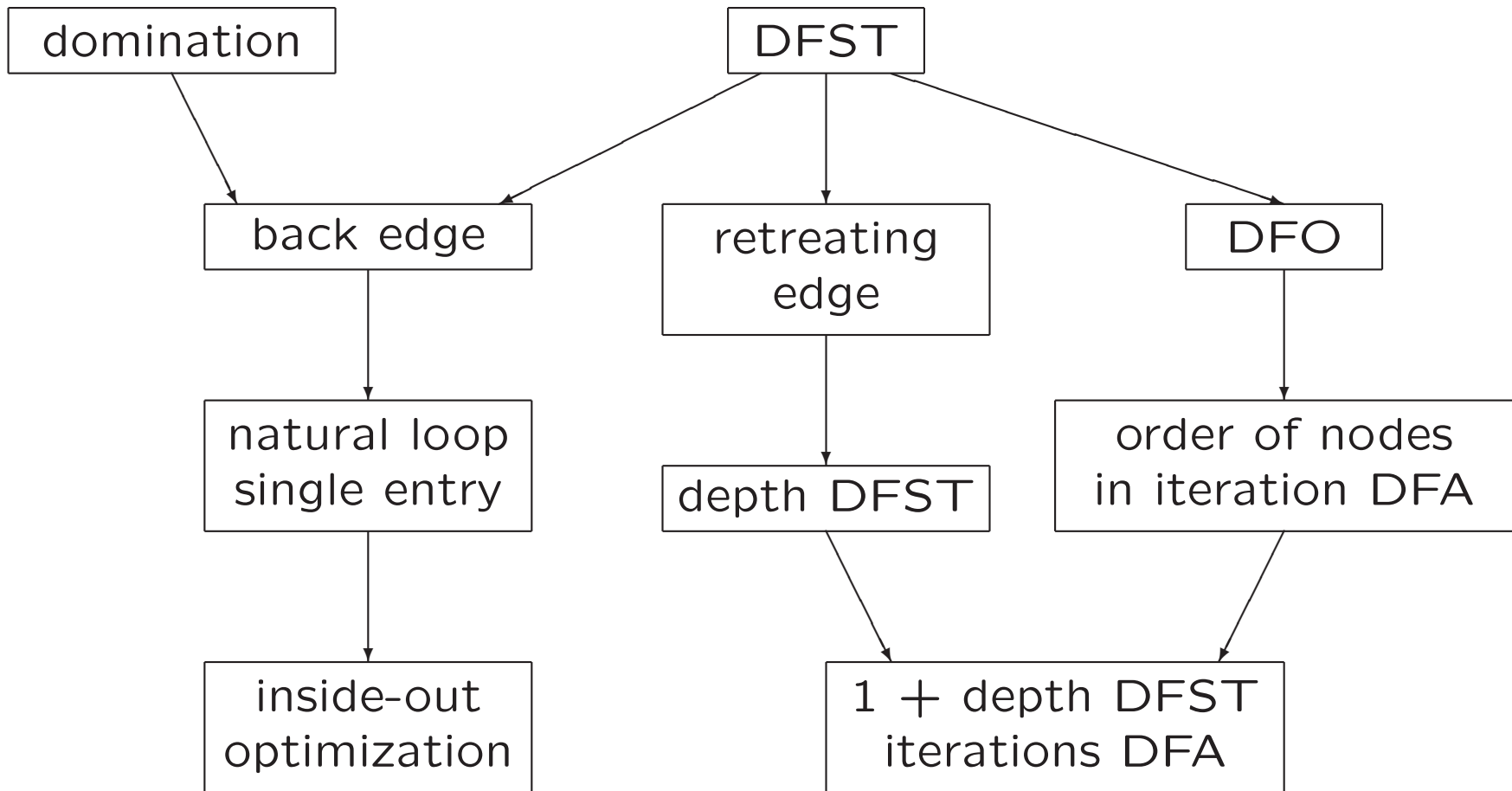
Fast Convergence

- Forward data-flow problem: visit nodes in depth-first-order
- Recall: edge $m \rightarrow n$ is retreating, if and only if n comes before m in depth-first ordering
- Example: path of propagation of definition d :

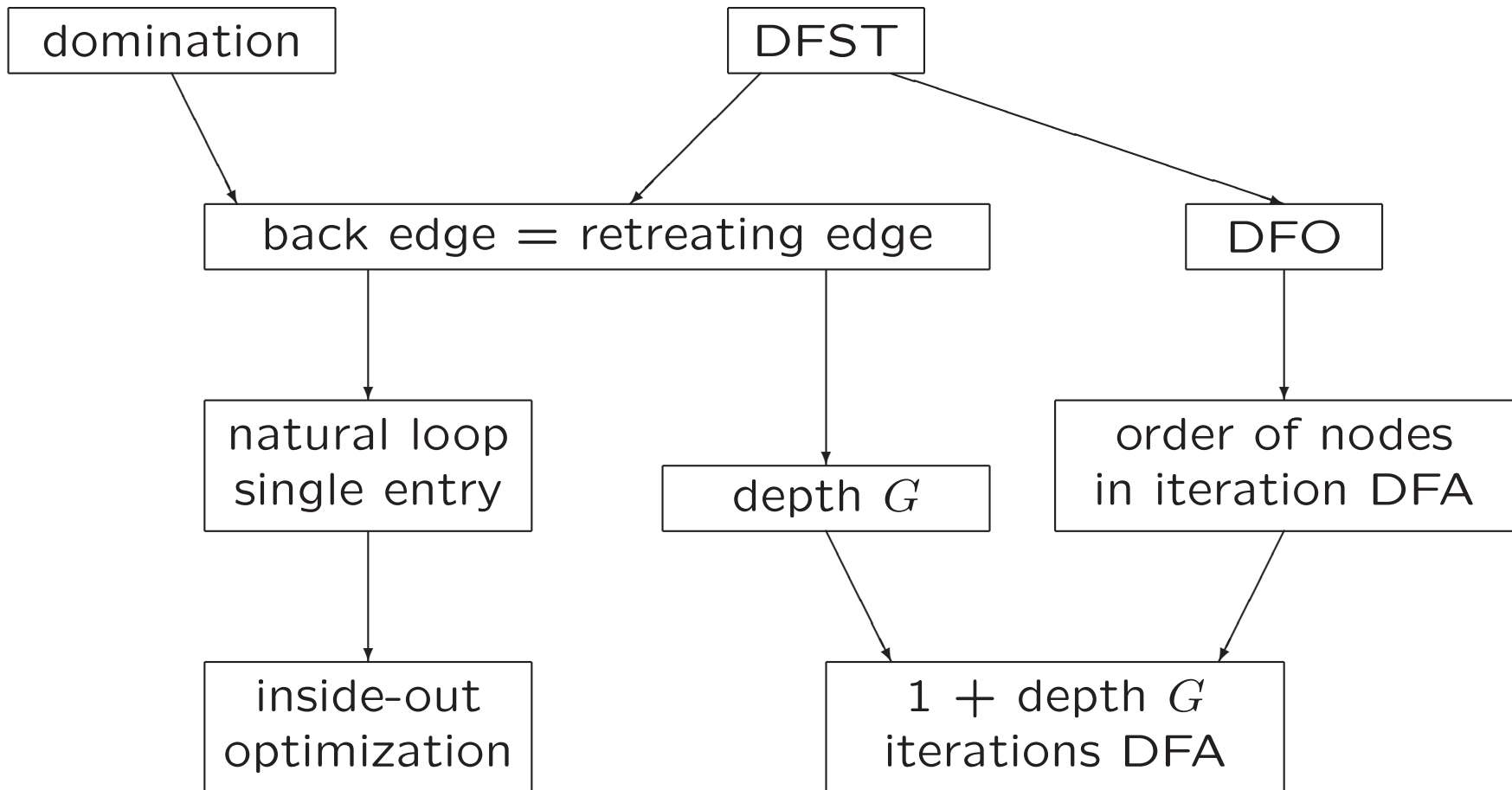
$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$

- Number of iterations: $1 + \text{depth} (+ 1)$
- Typical flow graphs have depth 2.75
- Backward data-flow problem: visit nodes in reverse of depth-first-order

Flow Graph G



Reducible Flow Graph G



En verder...

- Vanmiddag, 16.15 uur: practicum over opdracht 4
- Donderdag 12 december: inleveren opdracht 4
- Donderdag 19 december, 14.15–17.15: tentamen
- Vragenuur ?

Compiler constructie

college 10

Code Optimization

Chapters for reading:

9.2, 9.6