Instructions Scheduling

List Scheduling, Trace Scheduling, Loop Unrolling & Software Pipelining

Outline

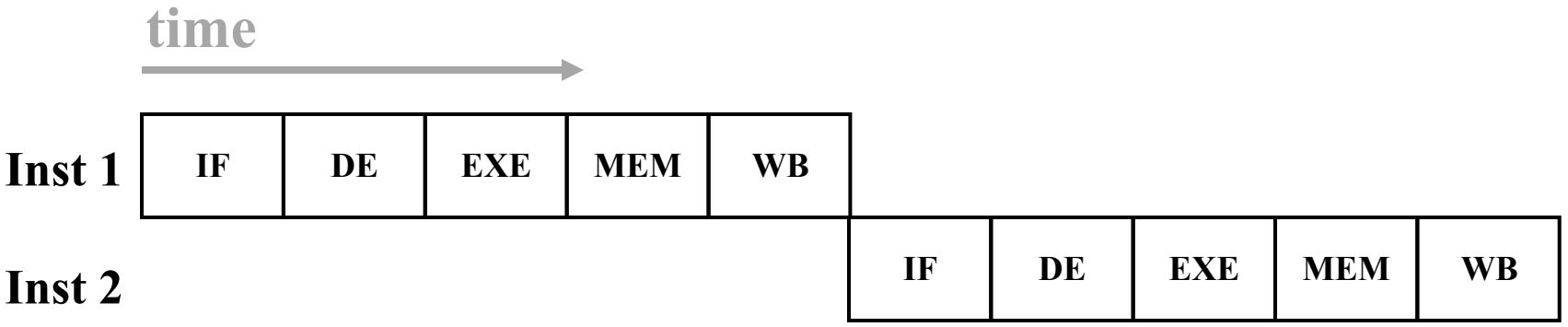
- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across Basic Blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining

Simple Execution Model

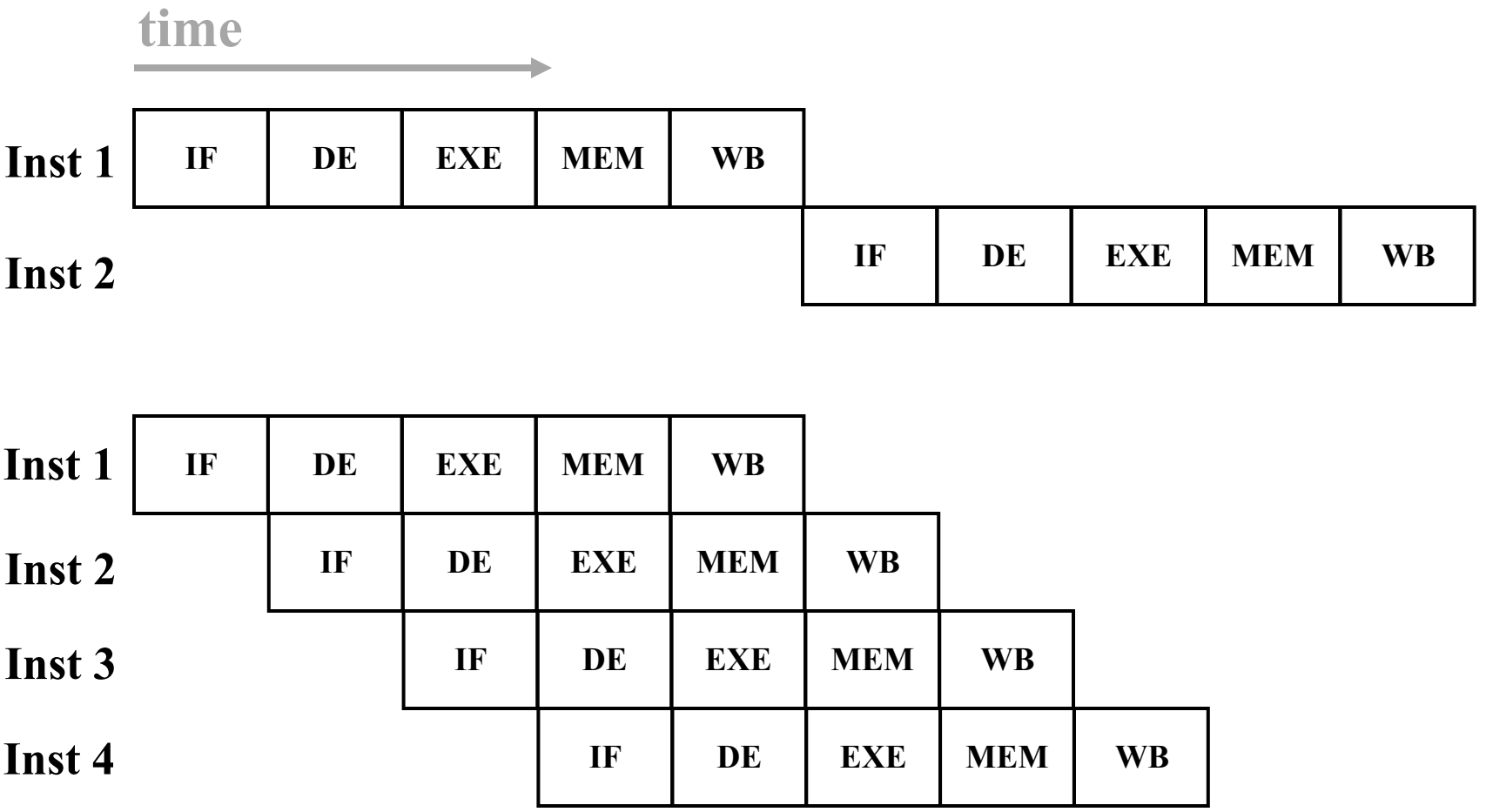
- 5 Stage pipe-line

- Fetch: Get the next instruction
- Decode: Figure-out what that instruction is
- Execute: Perform ALU operation
  - address calculation in a memory op
- Memory: Do the memory access in a Mem. Op.
- Write Back: Write results back

Simple Execution Model

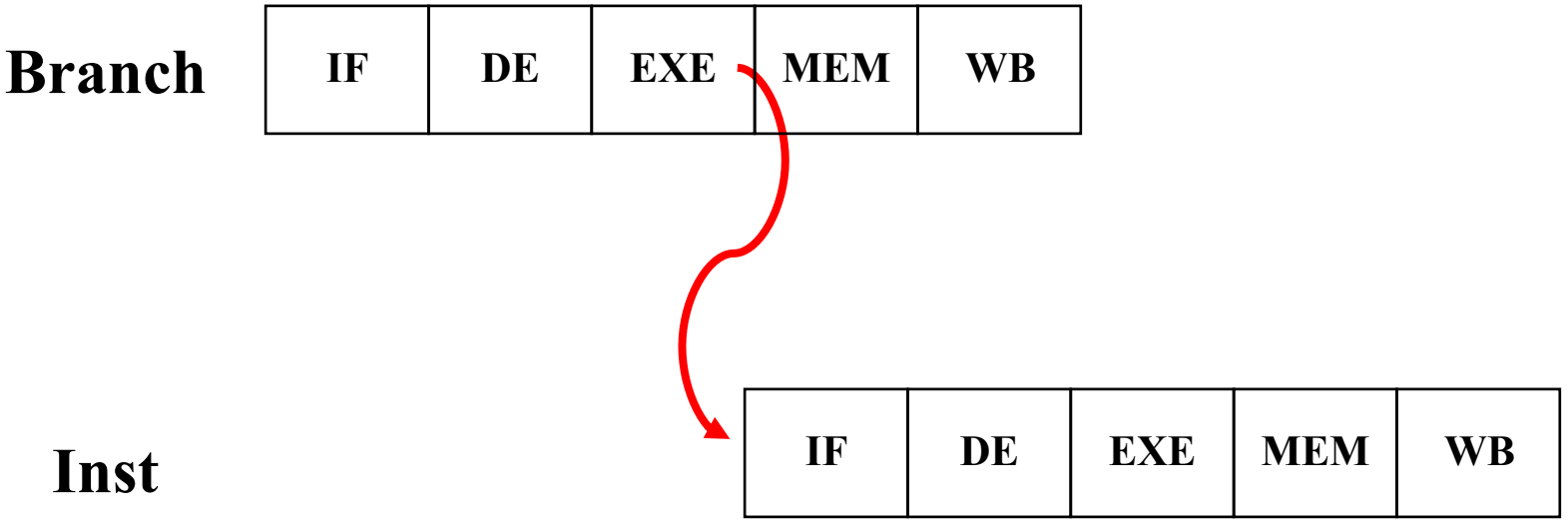


Simple Execution Model



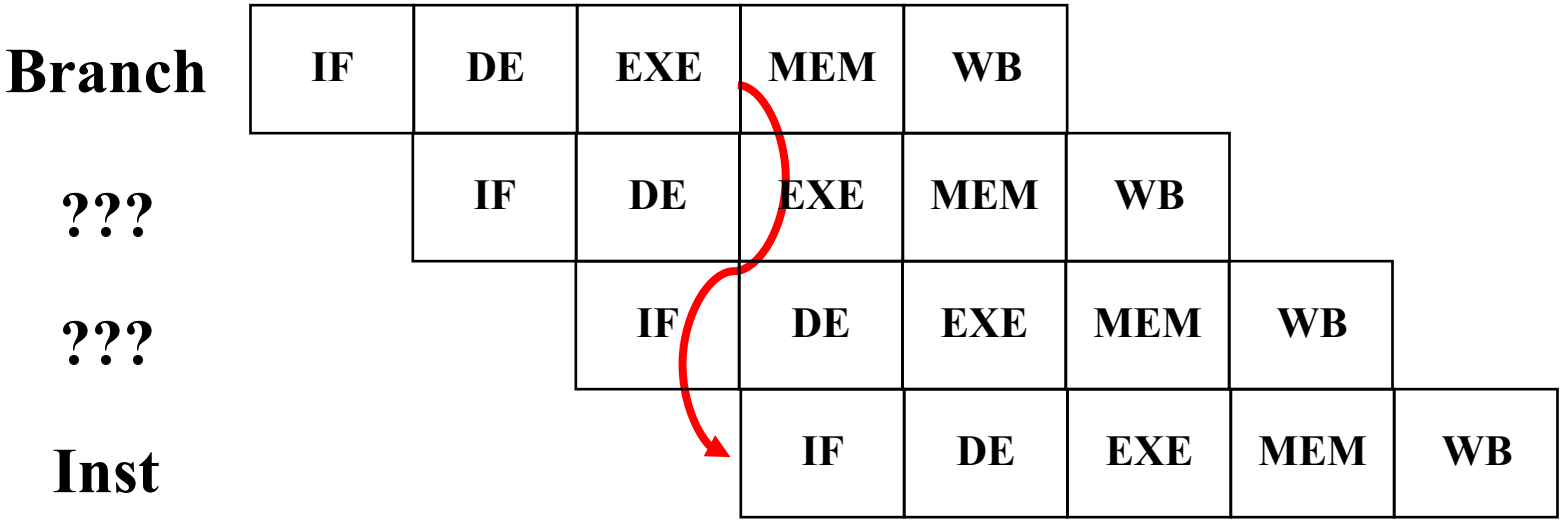
Handling Branch and Jump Instructions

- Does not know the location of the next instruction until later
  - after DE in jump instructions
  - after EXE in branch instructions



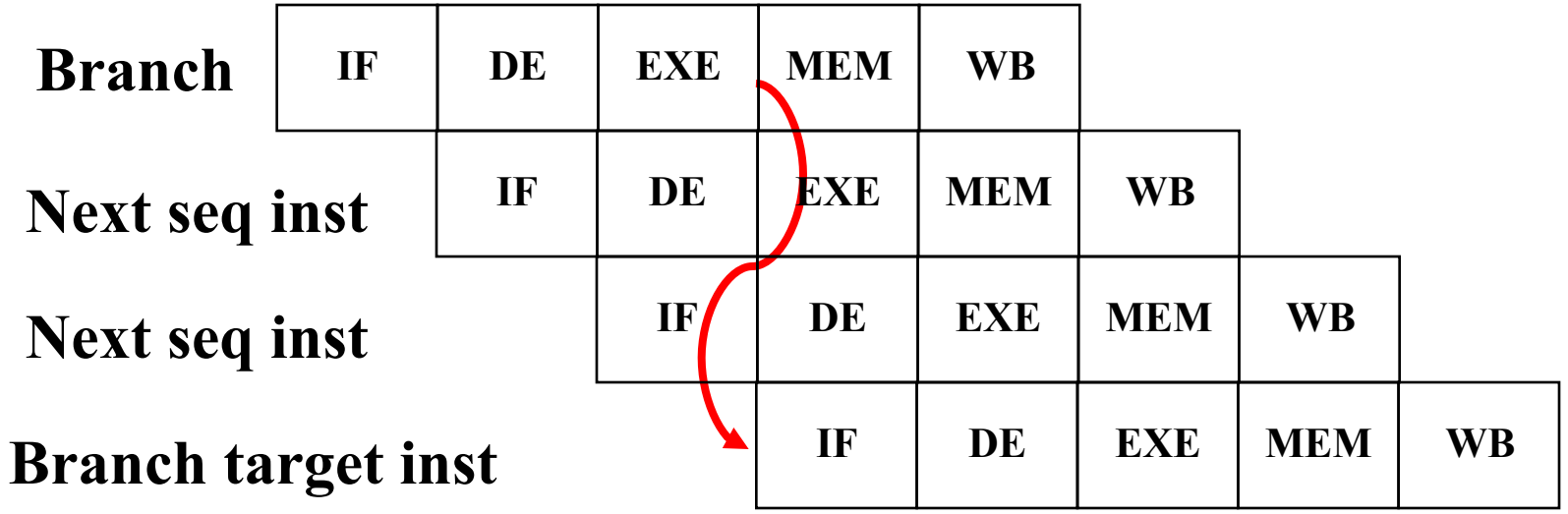
Handling Branch and Jump Instructions

- Does not know the location of the next instruction until later
  - after DE in jump instructions
  - after EXE in branch instructions
- What to do with the middle 2 instructions?



Handling Branch and Jump Instructions

- What to do with the middle 2 Instructions?
- Delay the Action of the Branch (*Delay Slots*)
  - Make branch affect only after two instructions
  - Two instructions after the branch get executed regardless of the branch



Constraints On Scheduling

- Data Dependences
  - Inherent in the code
- Control Dependences
  - Inherent in the code
- Resource Constraints

- Sometimes we can Mitigate these Issues
  - Code restructuring
  - Instruction Scheduling

Data Dependency between Instructions

- If two instructions access the same variable (i.e. the same memory location), they *can* be dependent
- Kind of Dependences
  - True: write → read
  - Anti: read → write
  - Output: write → write
  - Input: read → read

- What to do if two Instructions are Dependent?
  - The order of execution cannot be reversed
  - Reduce the possibilities for scheduling

Representing Dependences

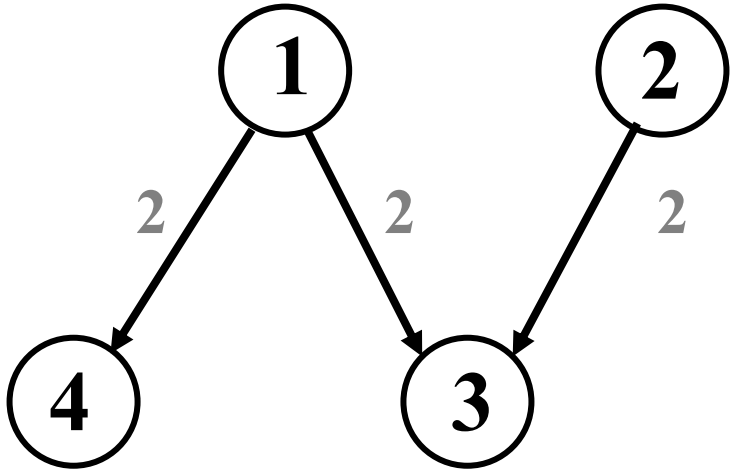
- Using a dependence DAG, one per Basic Block
- Nodes are instructions, Edges represent dependences

```
1: ld   r2, 4(r1)
2: ld   r3, 8(r1)
3: add  r4, r2, r3
4: subi r5, r2, 1
```

Representing Dependences

- Using a Dependence DAG, one per Basic Block
- Nodes are Instructions, Edges represent Dependences

```
1: ld   r2, 4(r1)
2: ld   r3, 8(r1)
3: add  r4, r2, r3
4: subi r5, r2, 1
```



- Edge is labeled with Latency:
  - v(i → j) = delay required between initiation times of i and j minus the execution time required by i (so that data can be transmitted to j)

Resource Constraints

- Modern Machines Have Many Resource Constraints
- Superscalar Architectures:
  - Can Execute few Operations Concurrently
  - But have constraints
  - Example:
    - 1 integer Operation

      ```
      ALU.op dest, src1, src2   # in 1 clock cycle
      ```

      In parallel with
    - 1 Memory Operation

      ```
      ld dst, addr              # in 2 clock cycles
      st src, addr              # in 1 clock cycle
      ```

Outline

- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across Basic Blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining

List Scheduling Algorithm

- Idea:
  - Do a Topological Sorting of the Dependence DAG
  - Consider when an instruction can be scheduled without causing a pipeline stall
  - Schedule the instruction if it causes no pipeline stall and all its predecessors are already scheduled

- Optimal List Scheduling is NP-complete
  - Use Heuristics when necessary

List Scheduling Algorithm

- Create a dependence DAG of a Basic Block
- Topological Sorting

READY List = nodes with no predecessors
loop until READY list is empty
    schedule each node in READY list when no stalling
    update READY list
end loop

Heuristics for Selection

- Heuristics for selecting from the READY list
  - Pick a node with the longest path to a leaf in the dependence graph
  - Pick a node with most immediate successors
  - Pick a node that can go to a less busy pipeline (in a superscalar)

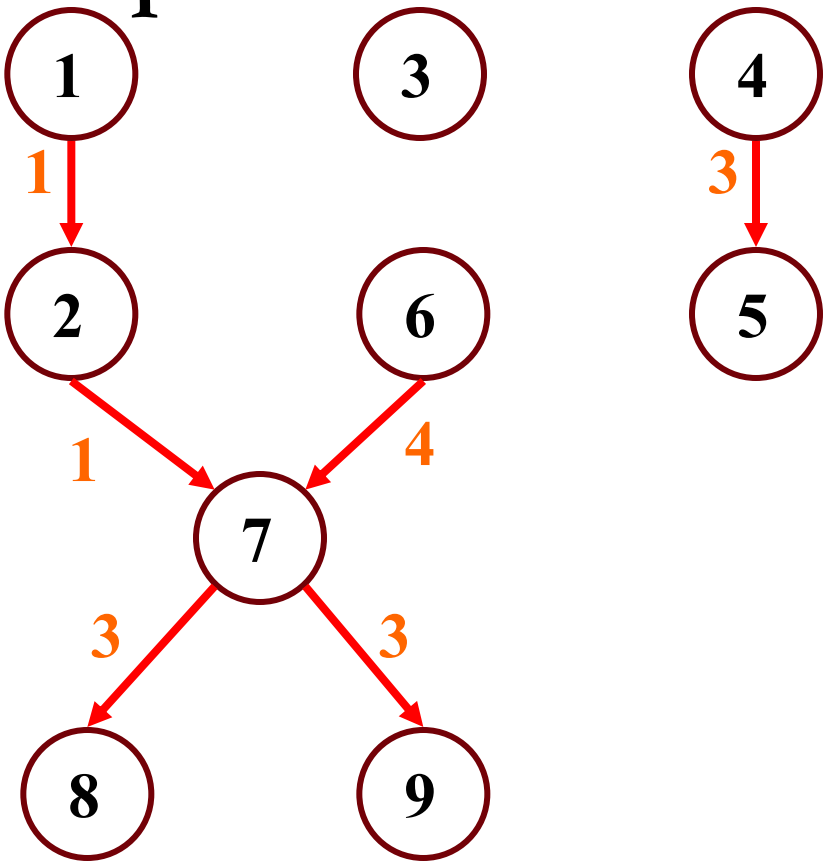
Heuristics for Selection

- Pick a node with the longest path to a leaf in the dependence graph
- Algorithm (for node x)
  - if no successors then $d_x = 0$
  - else $d_x = MAX(d_y + c_{xy})$ for all successors y of x
  - reverse breadth-first visitation order

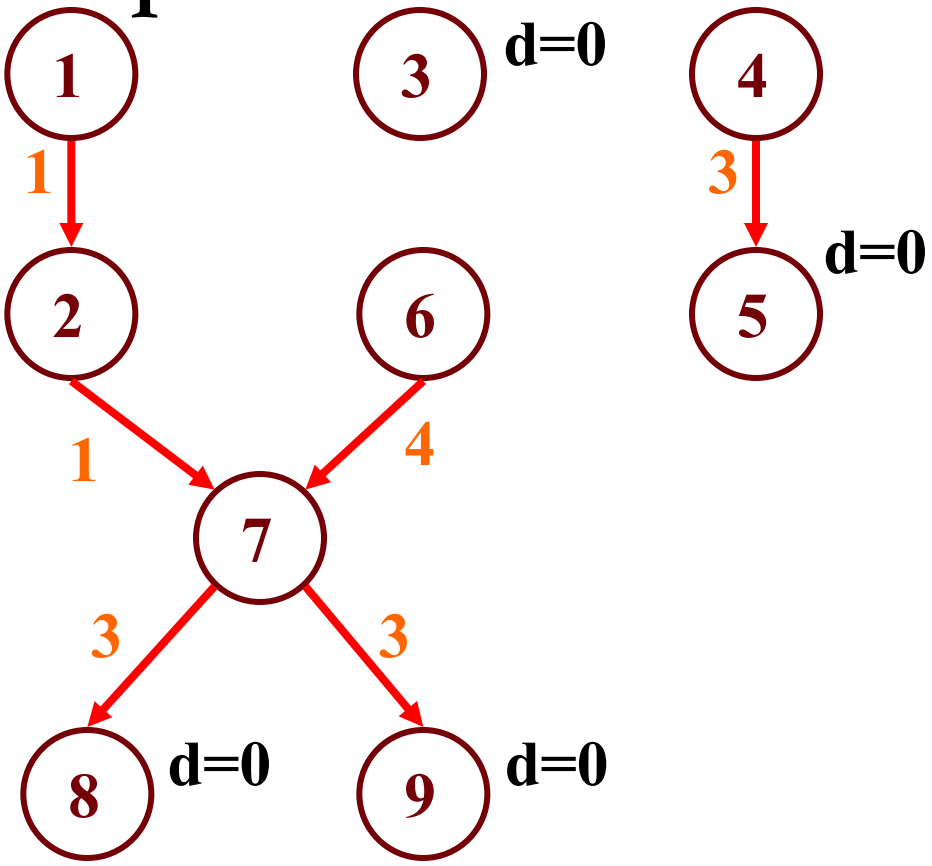
Heuristics for Selection

- Pick a node with most immediate successors
  - Rationale: Highest-degree will mean "solve" the most dependences
- Algorithm (for node x):
  - f_x = number of successors of x

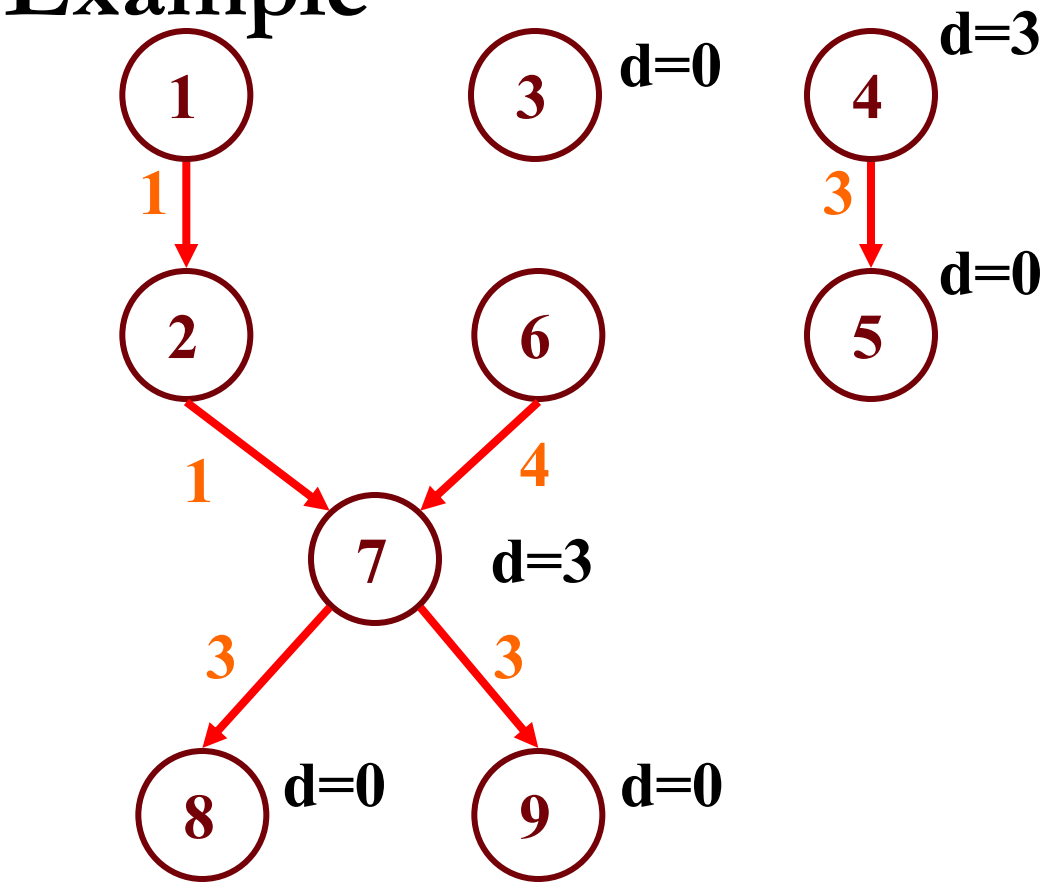
Example



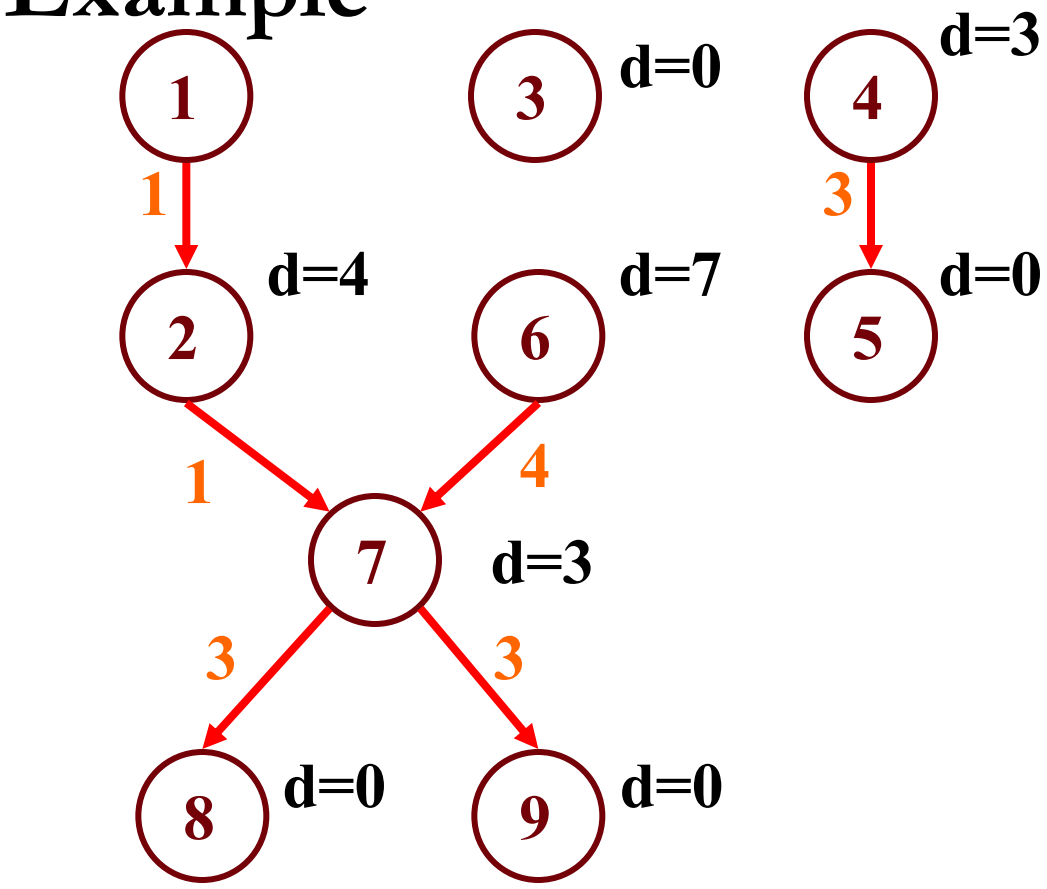
Example



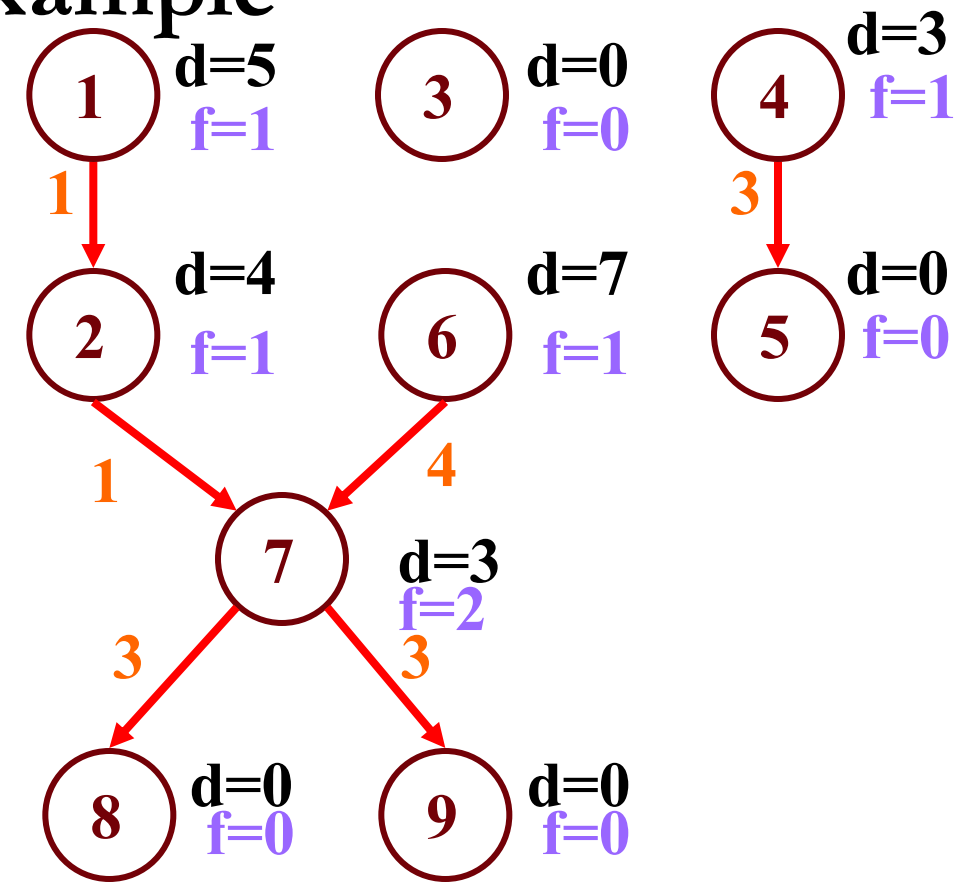
Example



Example

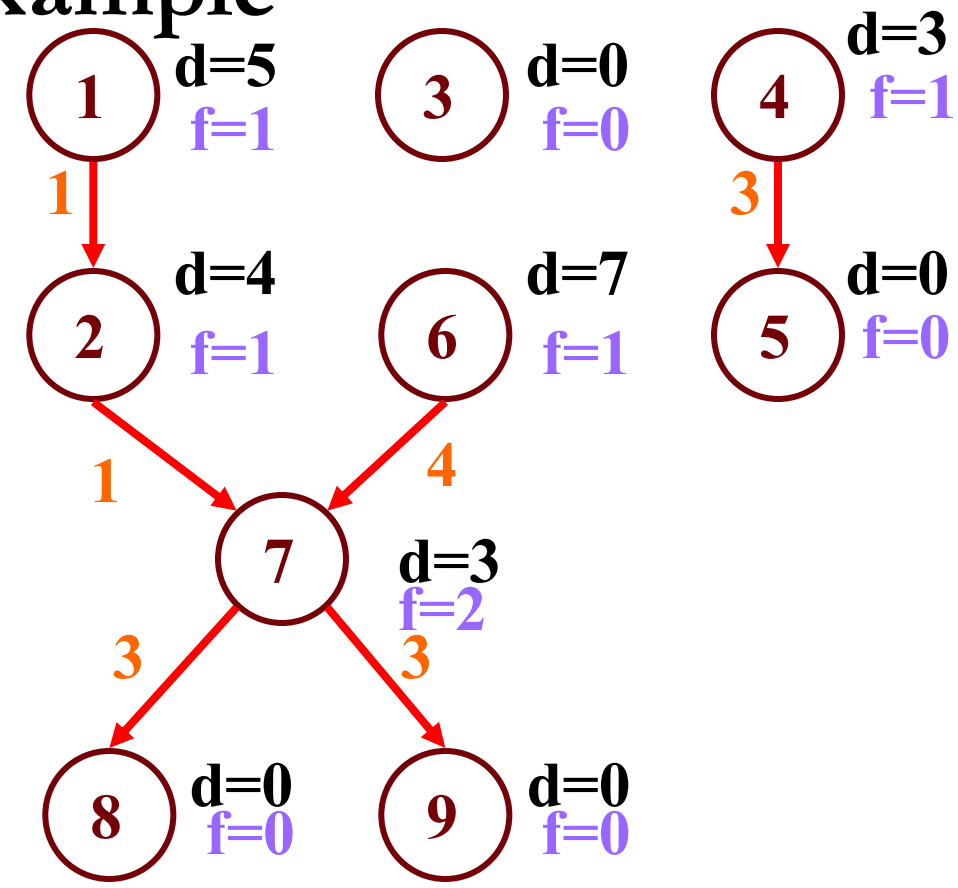


Example


1
d=5
f=1
3
d=0
f=0
4
d=3
f=1
1
3
2
d=4
f=1
6
d=7
f=1
5
d=0
f=0
1
4
7
d=3
f=2
3
3
8
d=0
f=0
9
d=0
f=0

Example

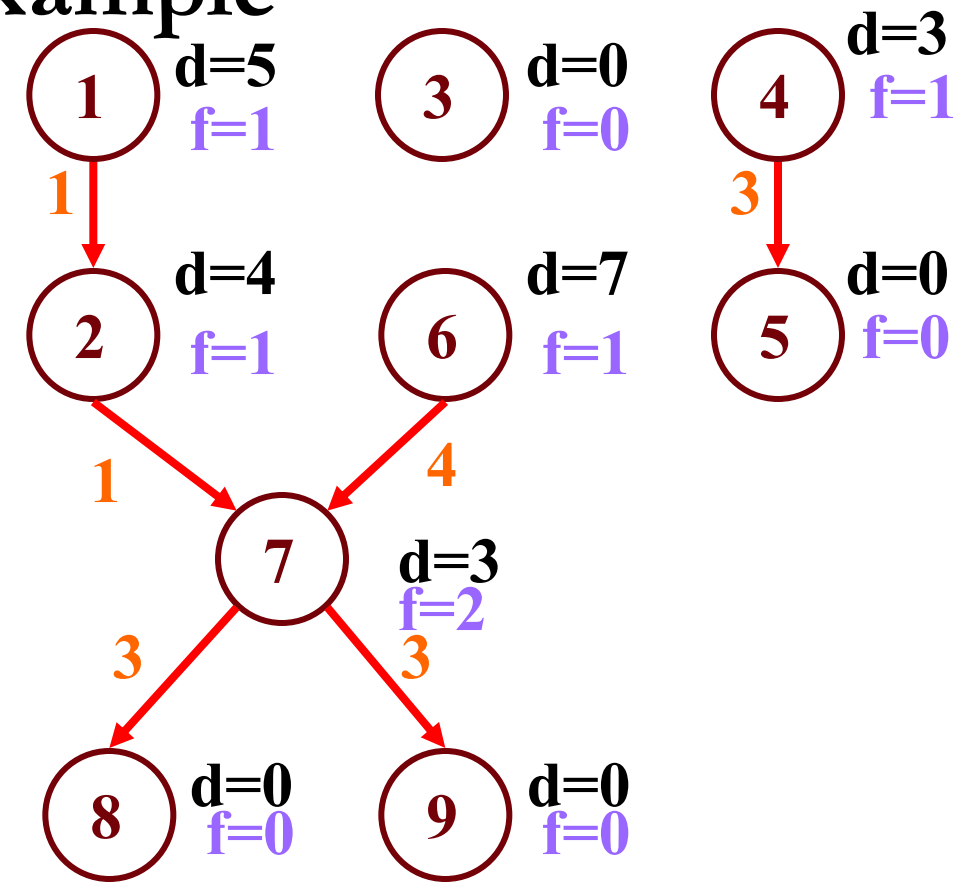
READY = { }



Example

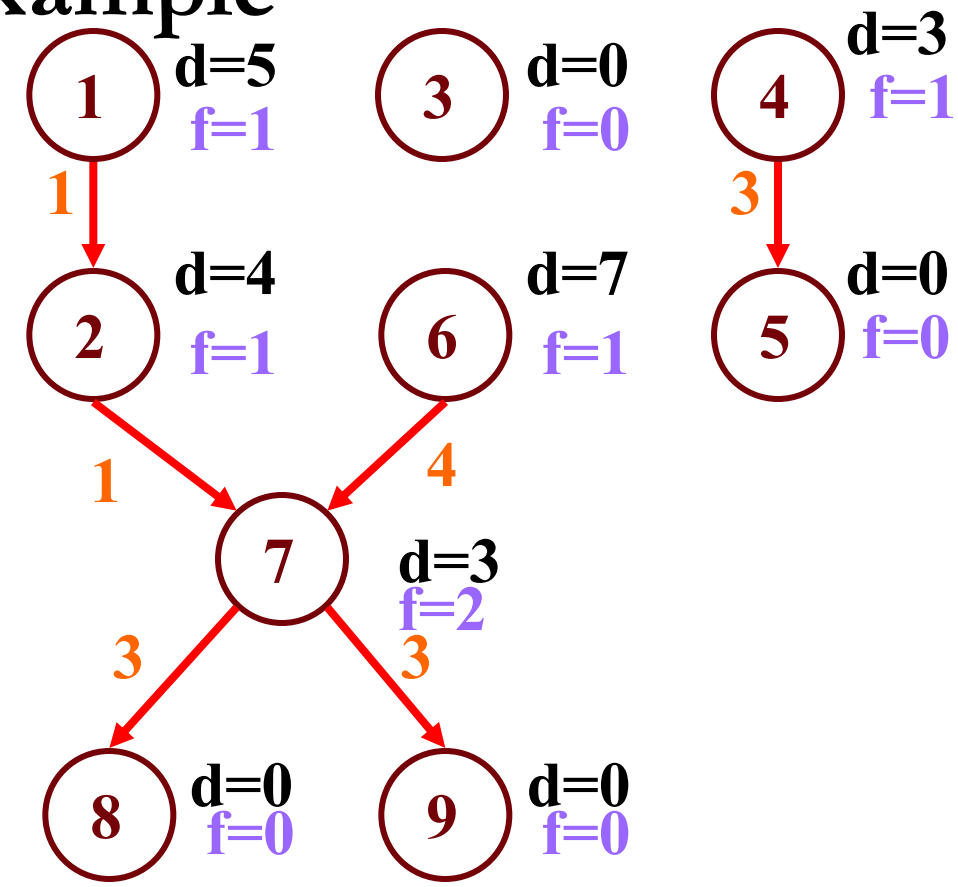
1,3,4,6

READY = { 6,1,4,3 }



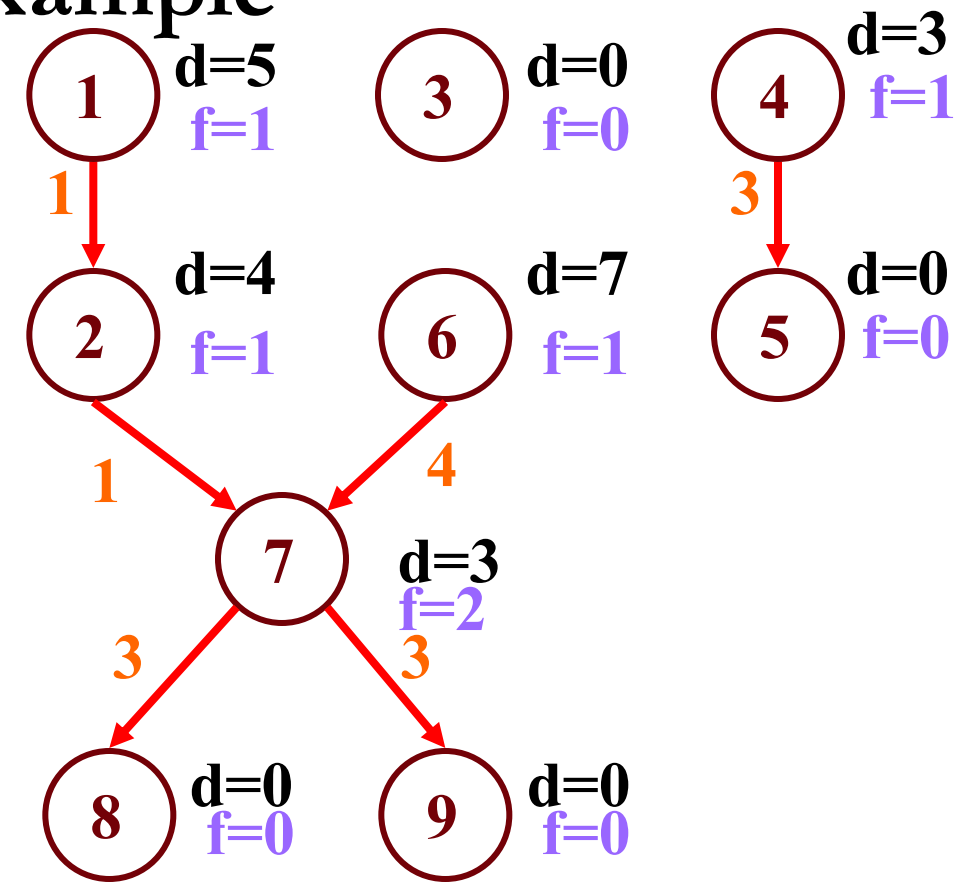
Example

READY = { 6,1,4,3 }



Example

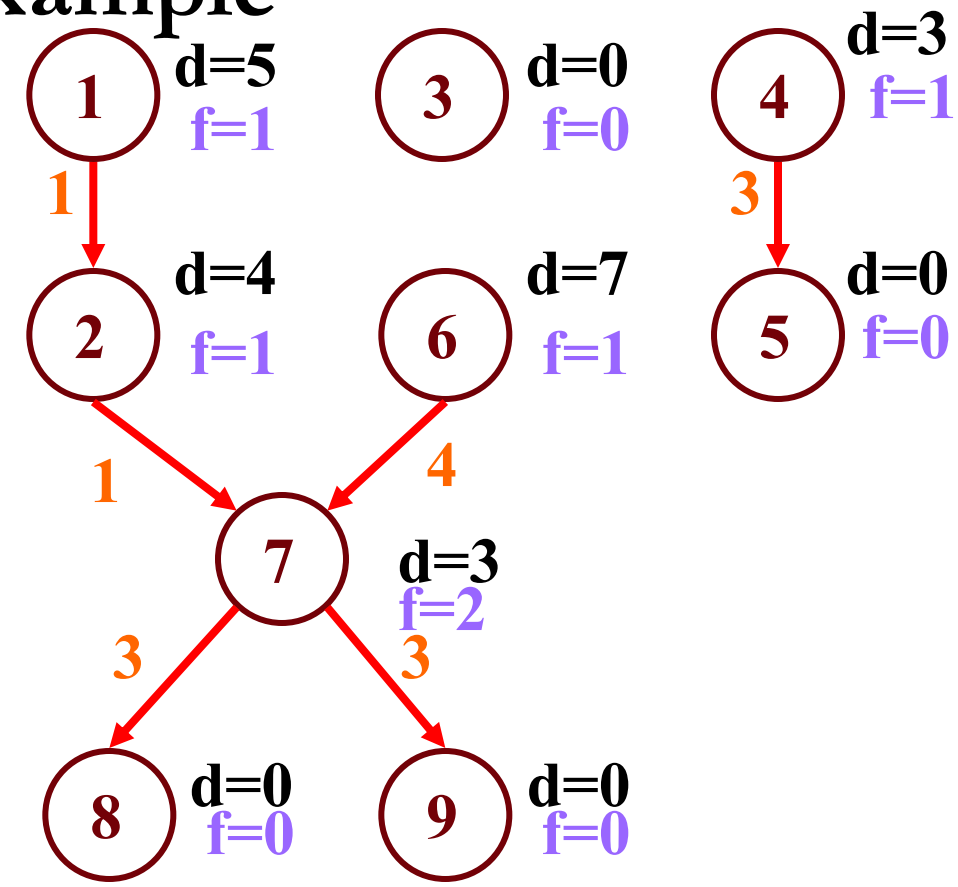
READY = { 6,1,4,3 }



6

Example

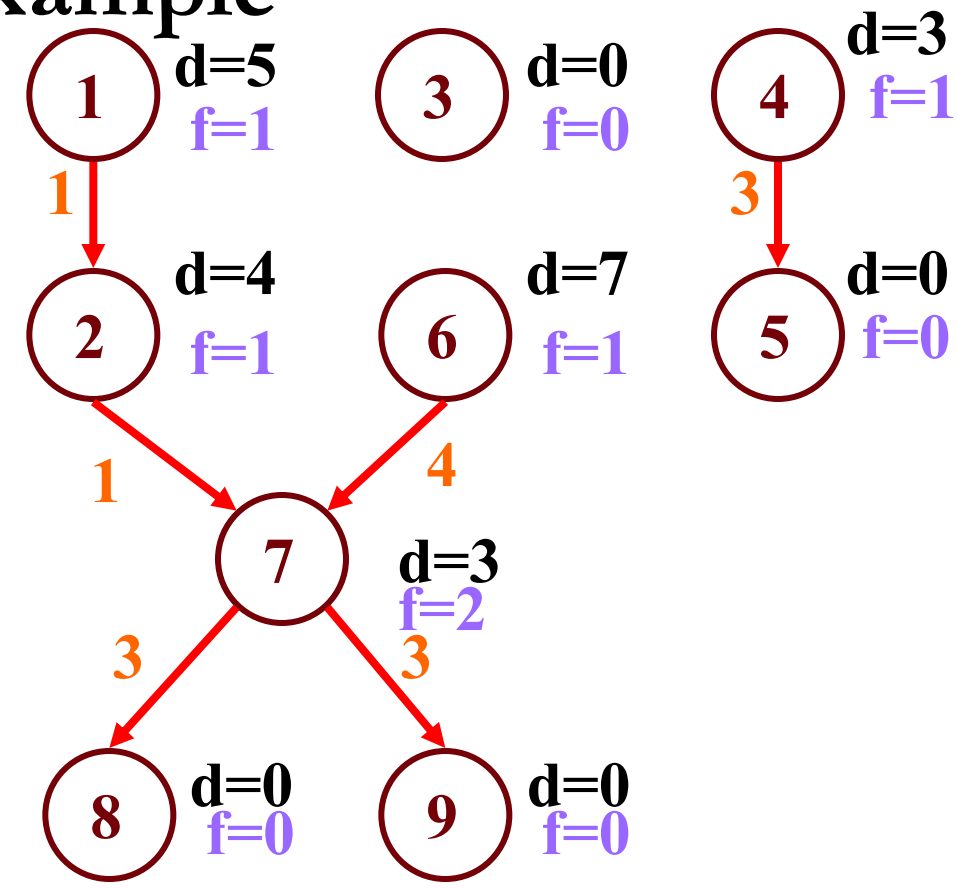
READY = { 1, 4, 3 }



6

Example

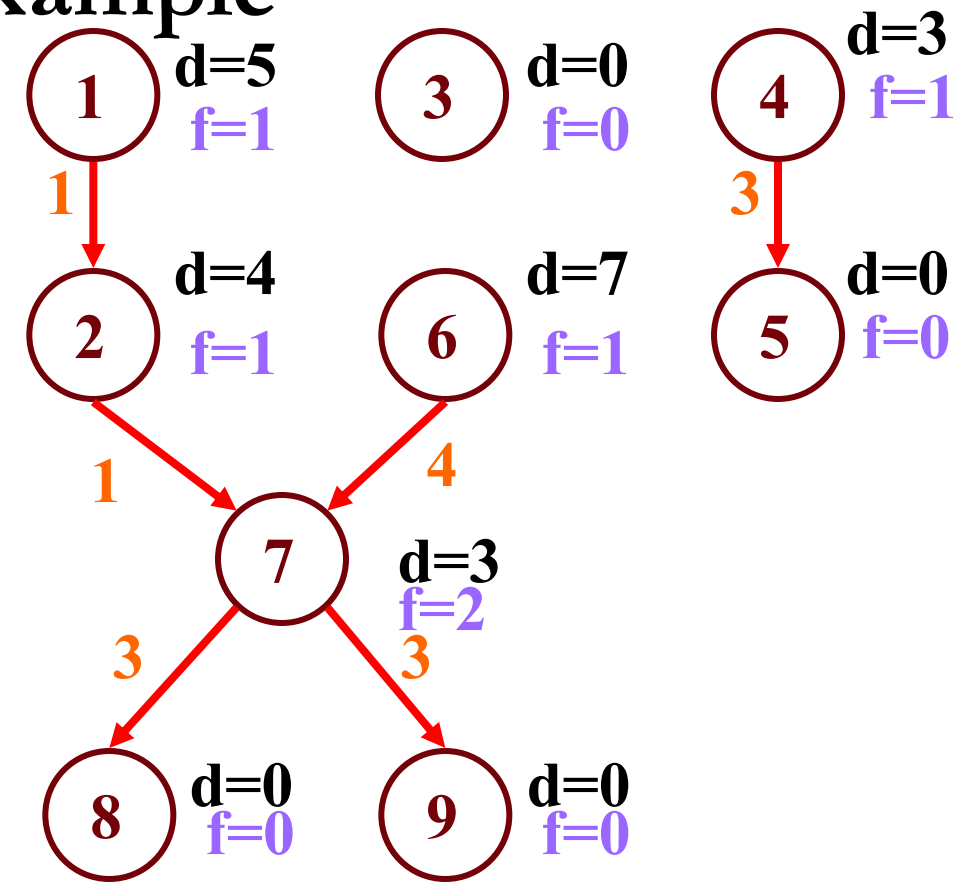
READY = { 1, 4, 3 }



6

Example

READY = { 1, 4, 3 }

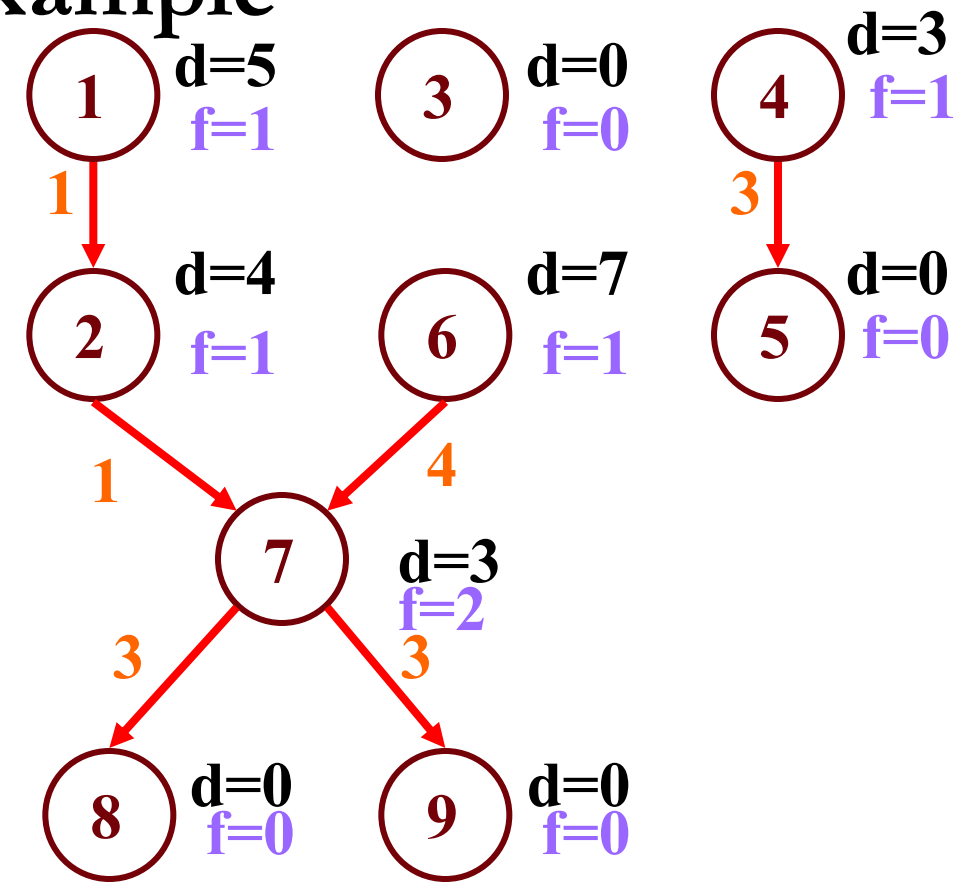


6	1

Example

2

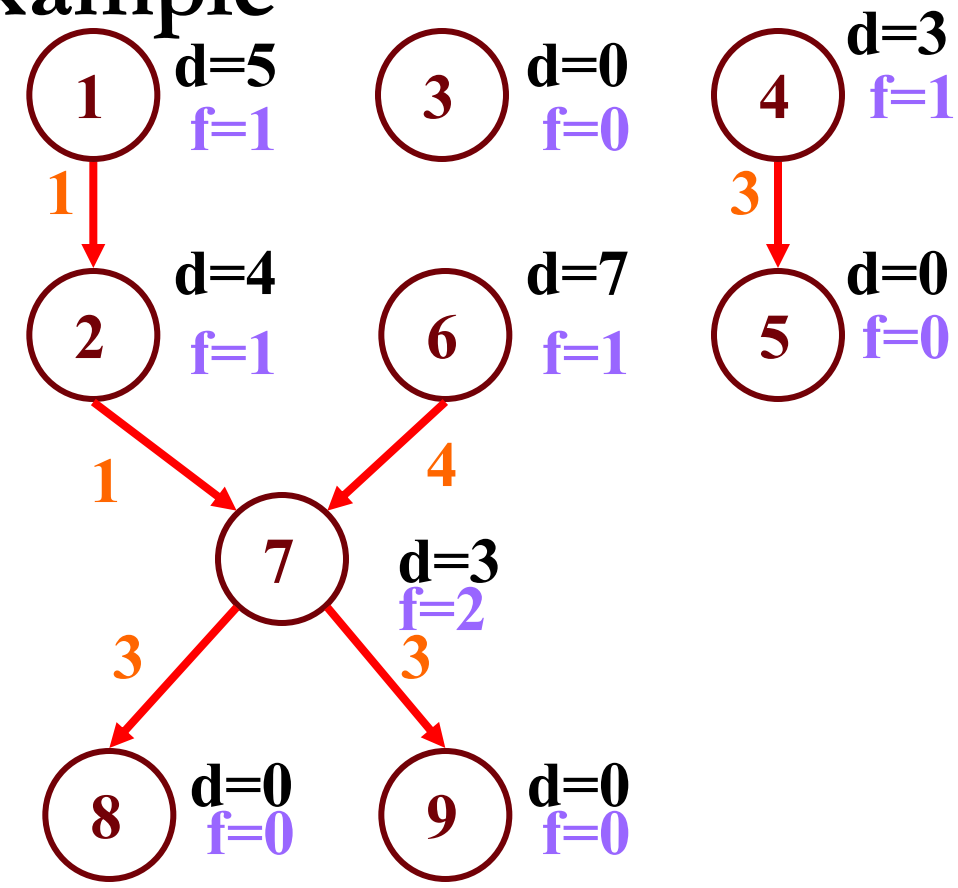
READY = { 4 , 3 }



6	1

Example

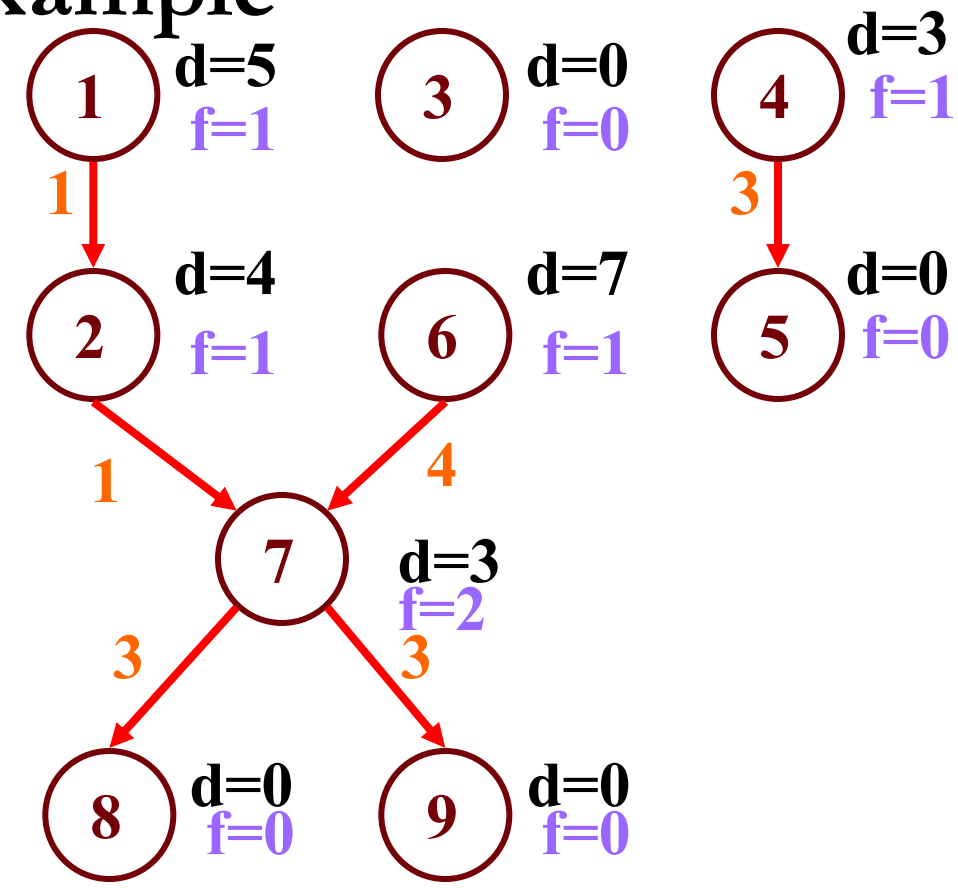
READY = { 2, 4 , 3 }



6	1

Example

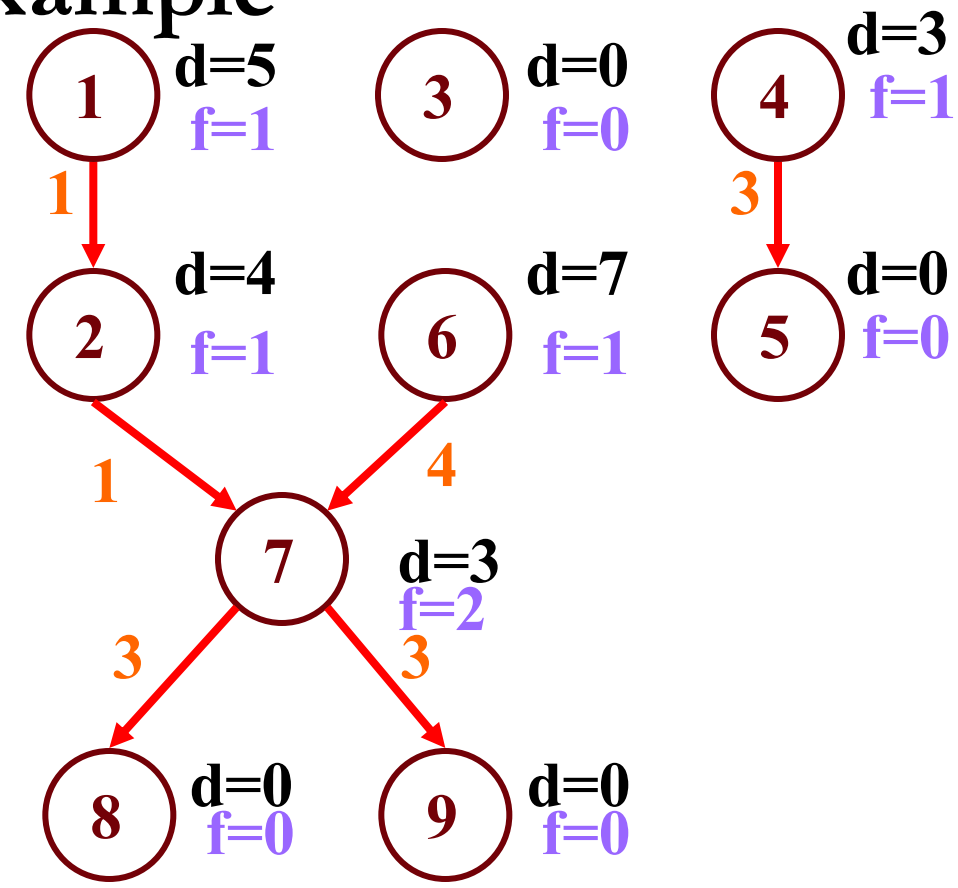
READY = { 2, 4 , 3 }

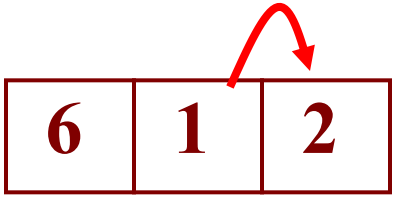


6	1

Example

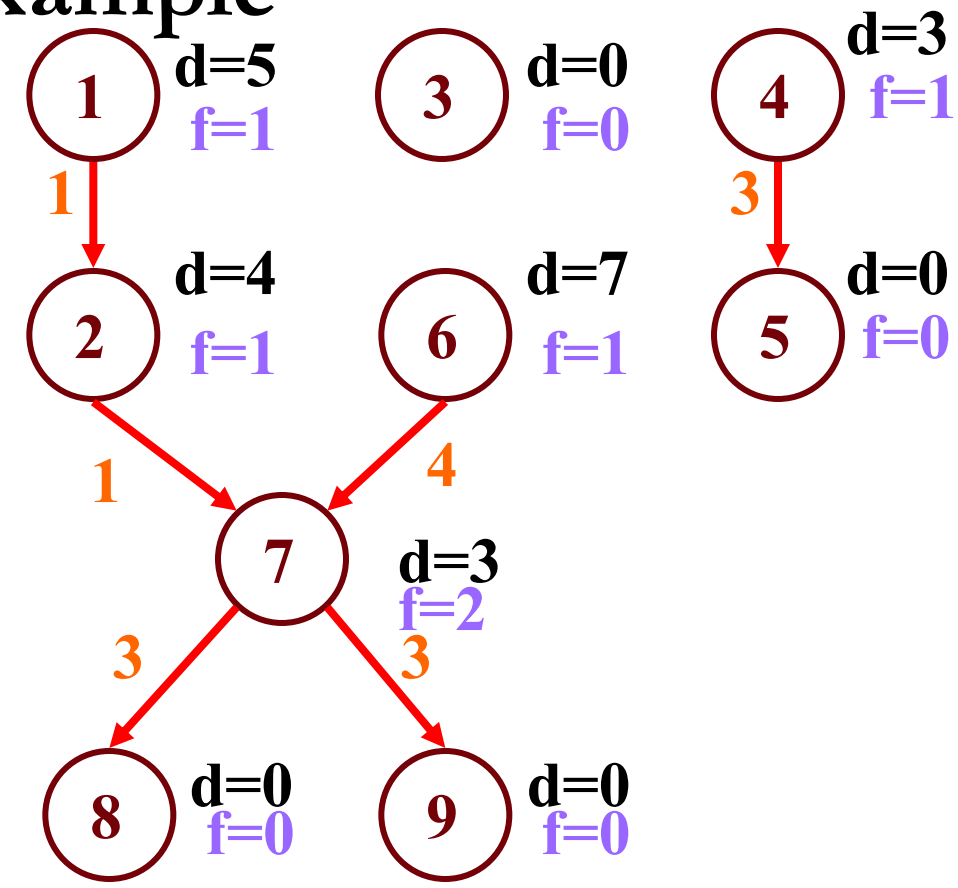
READY = { 2, 4 , 3 }

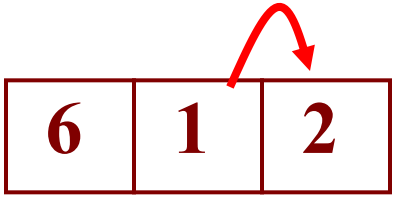




Example

READY = { 4 , 3 }

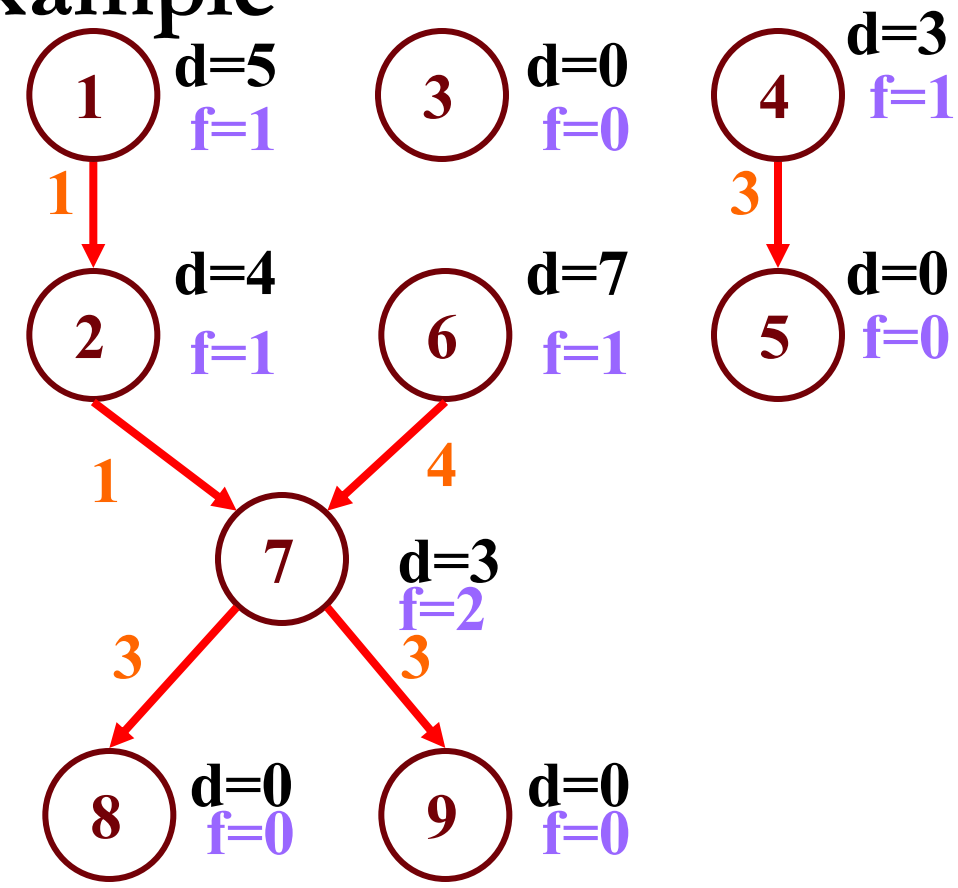




Example

7

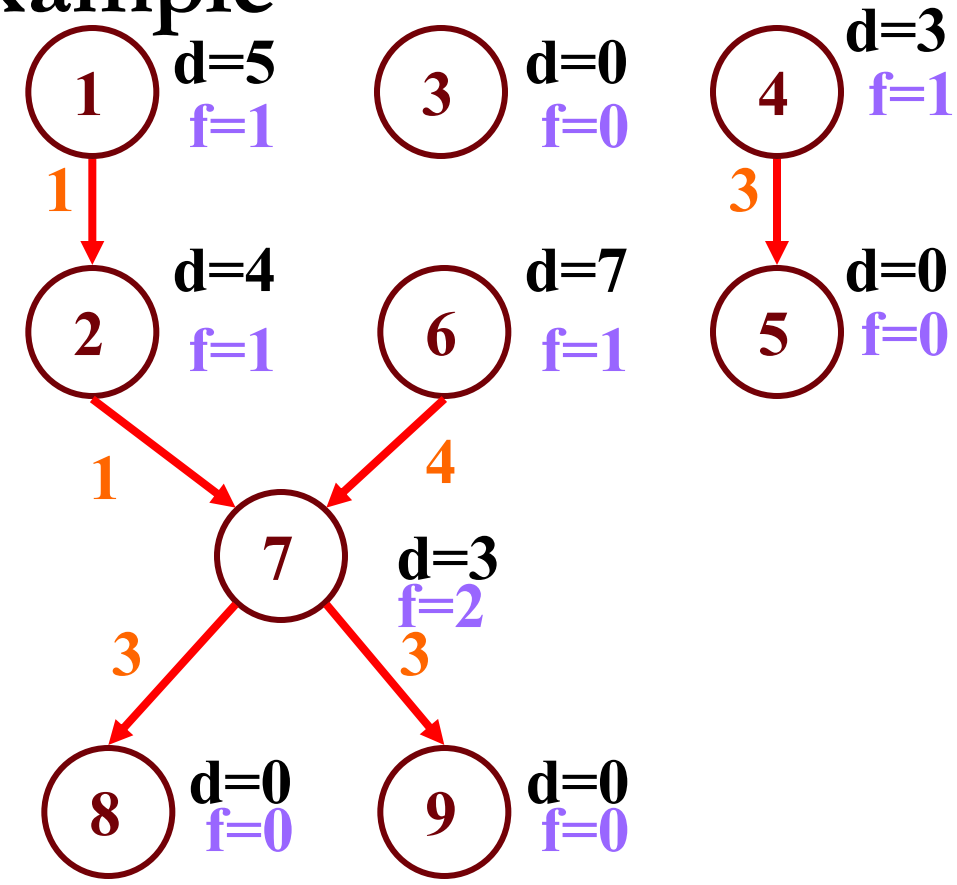
READY = { 4 , 3 }



6	1	2

Example

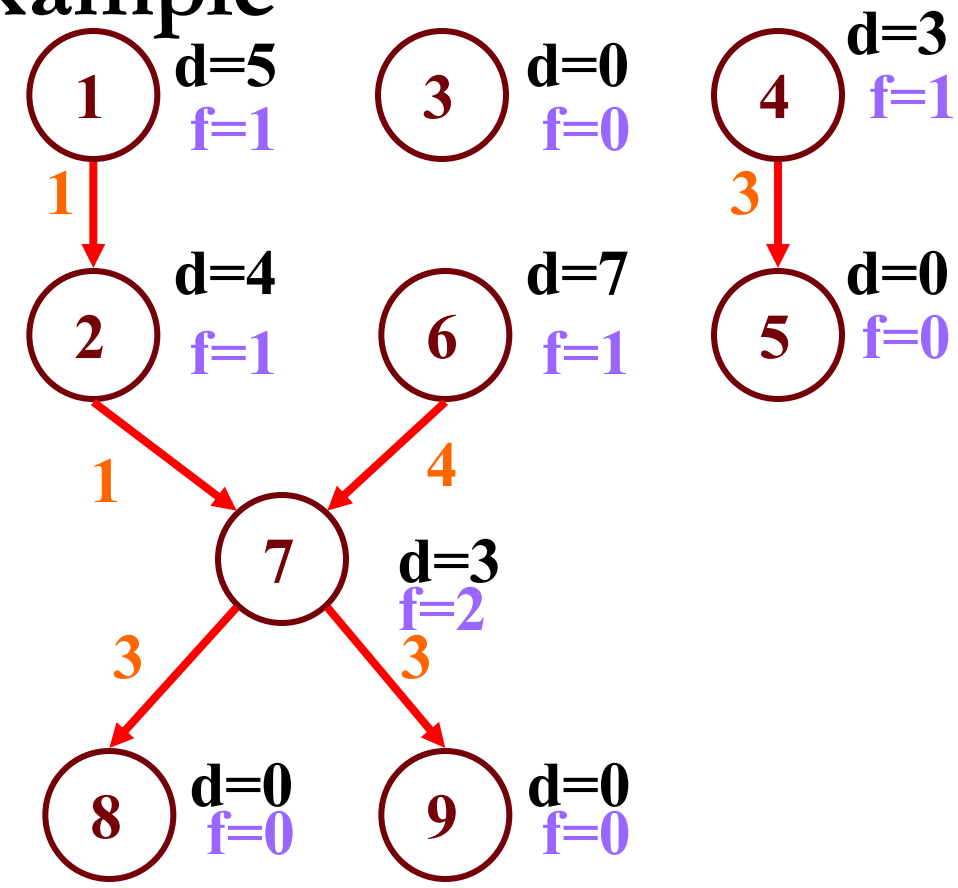
READY = { 7, 4 , 3 }



6	1	2

Example

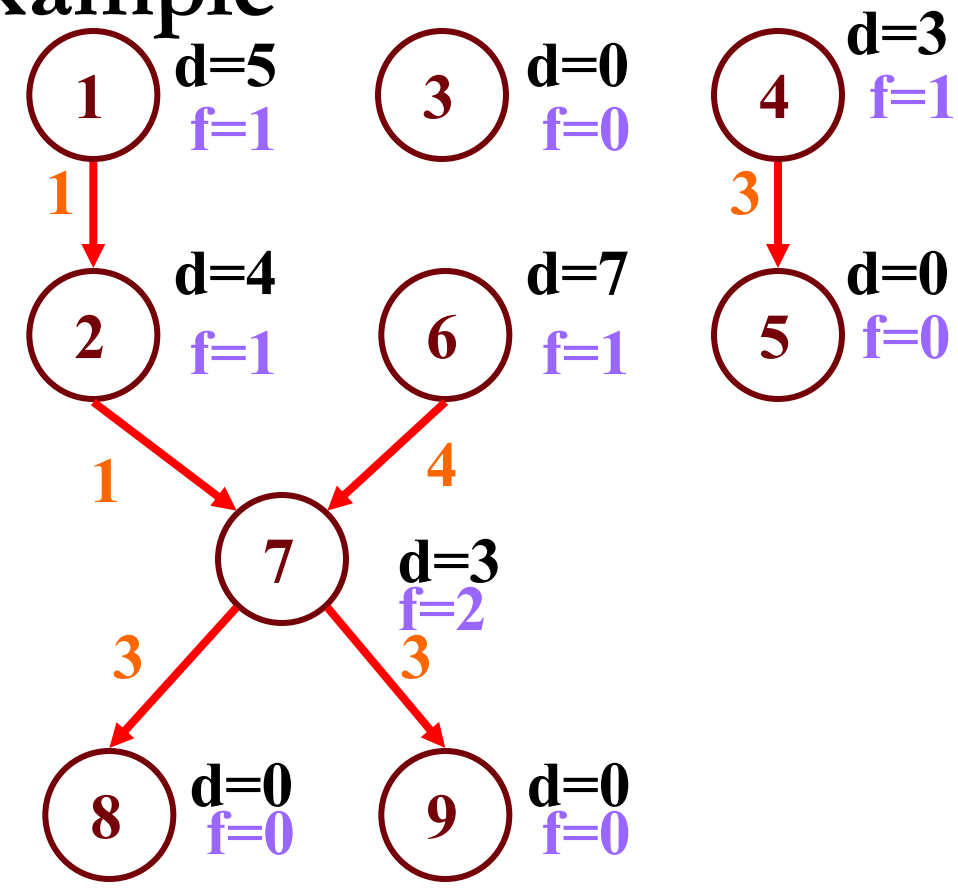
READY = { 7, 4 , 3 }

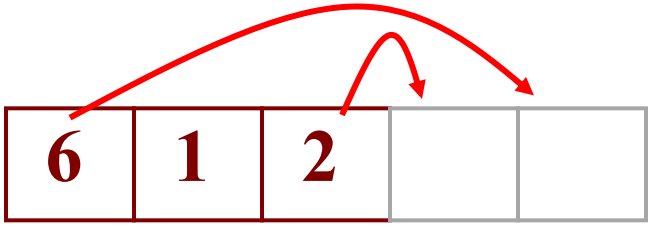


6	1	2

Example

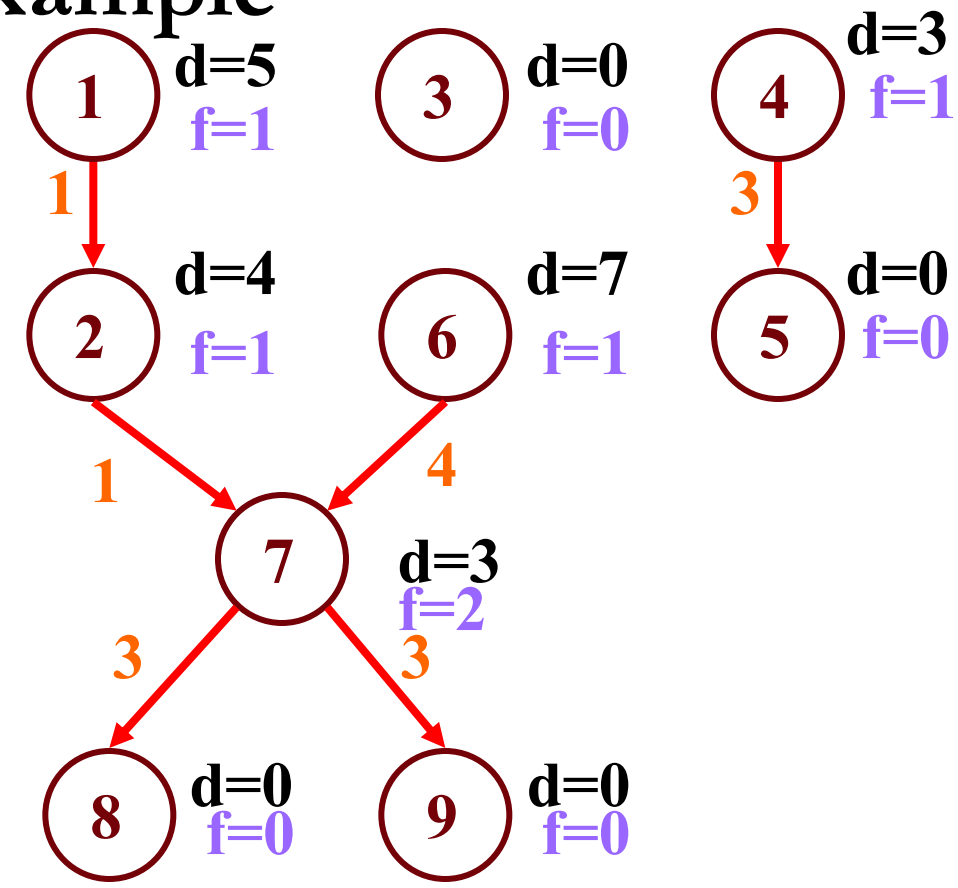
READY = { 7, 4 , 3 }





Example

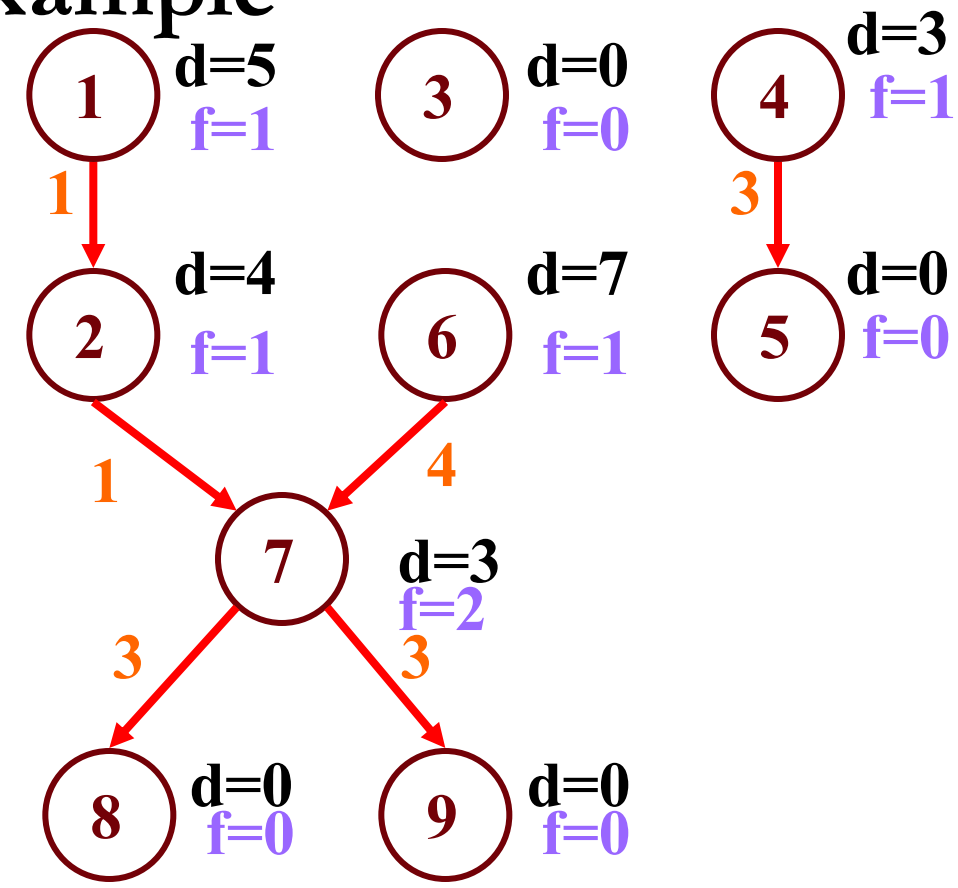
READY = { 7, 4 , 3 }



6	1	2

Example

READY = { 7, 4 , 3 }

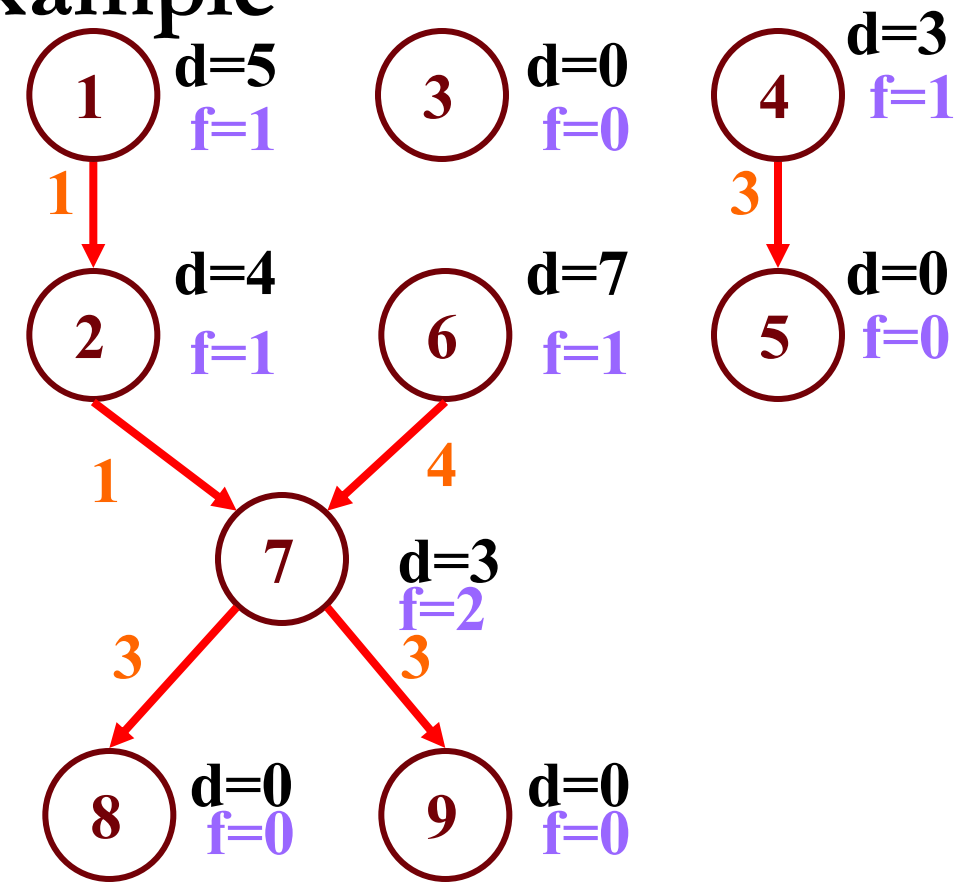


6	1	2	4

Example

5

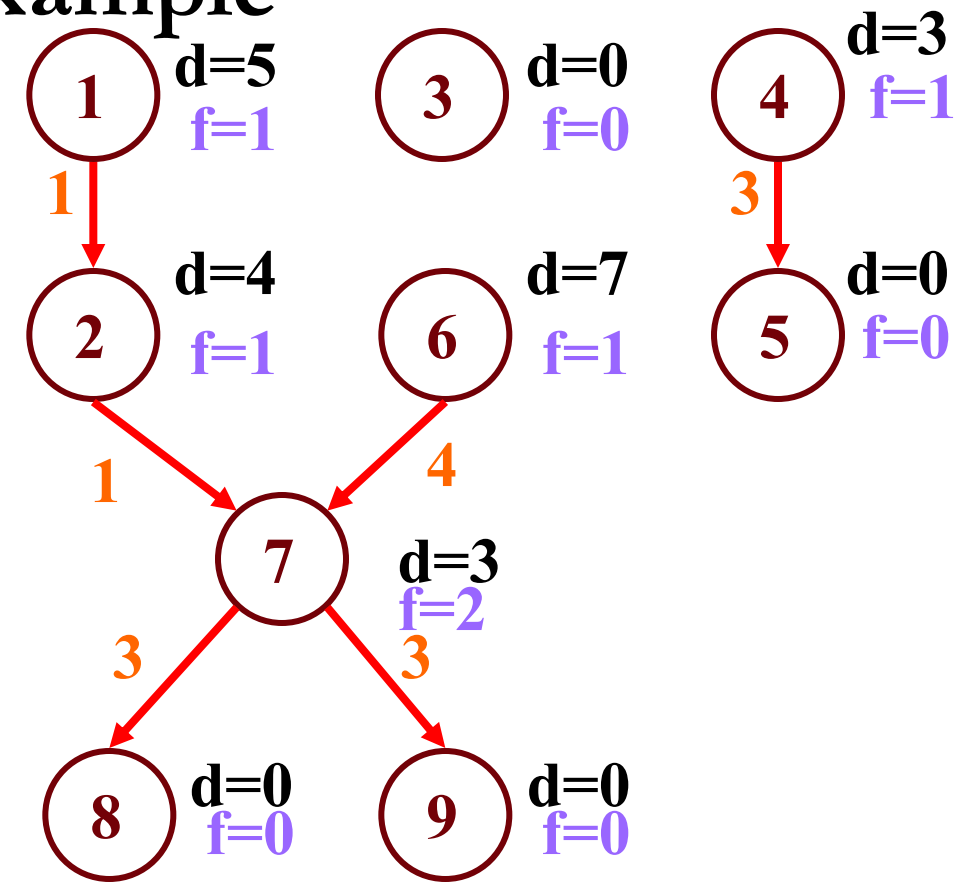
READY = { 7, 3 }



6	1	2	4

Example

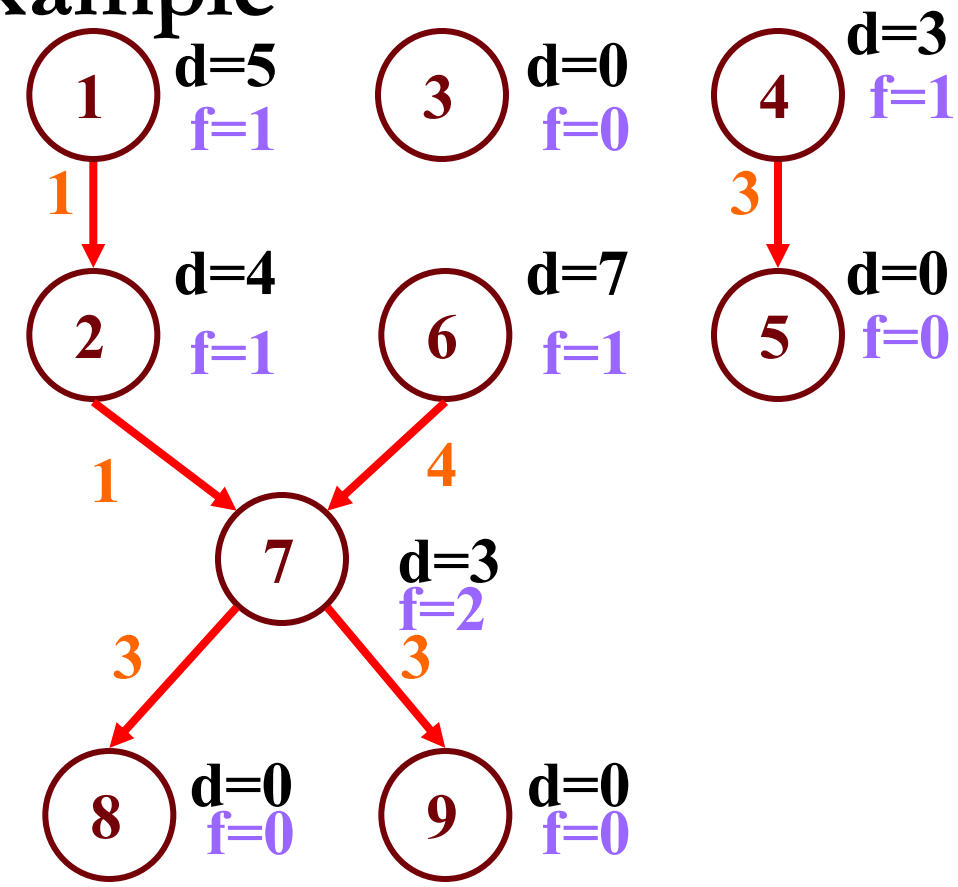
READY = { 7, 3, 5 }

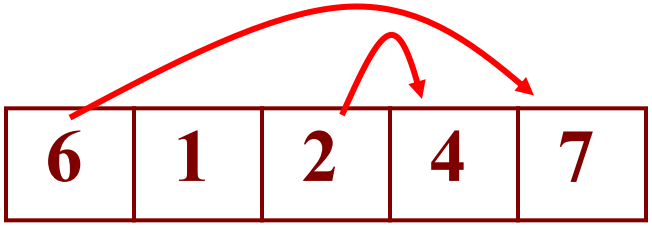


6	1	2	4

Example

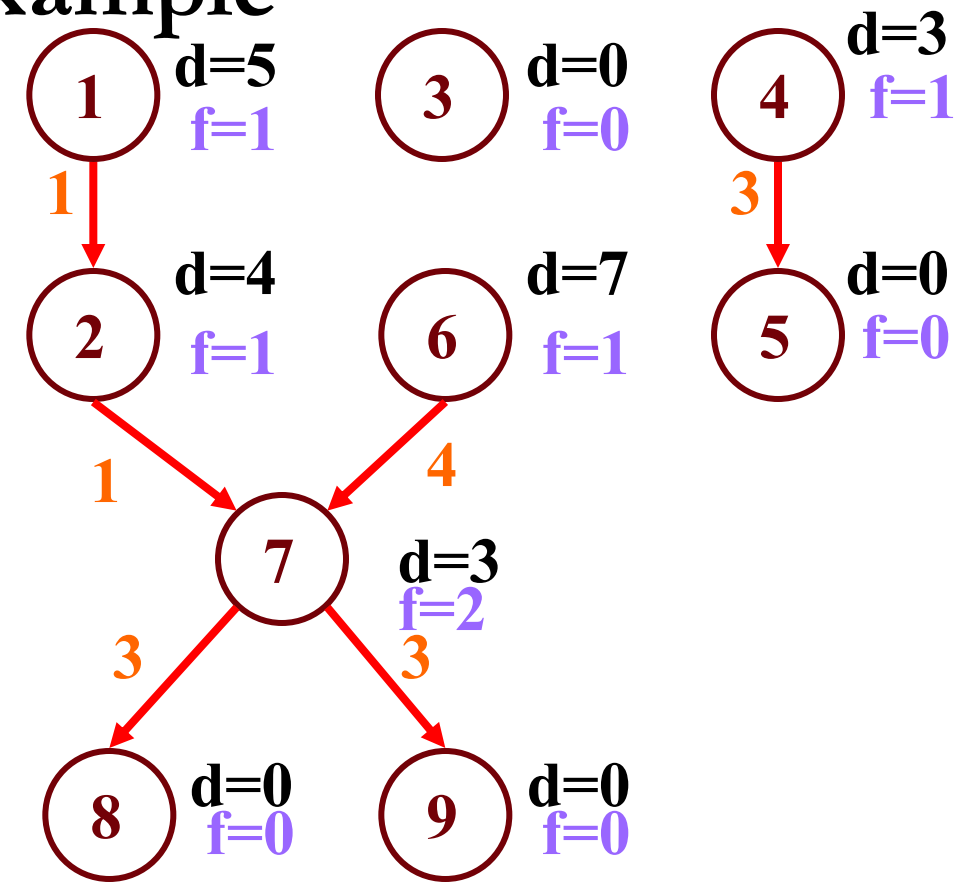
READY = { 7, 3, 5 }





Example

READY = { 3, 5 }

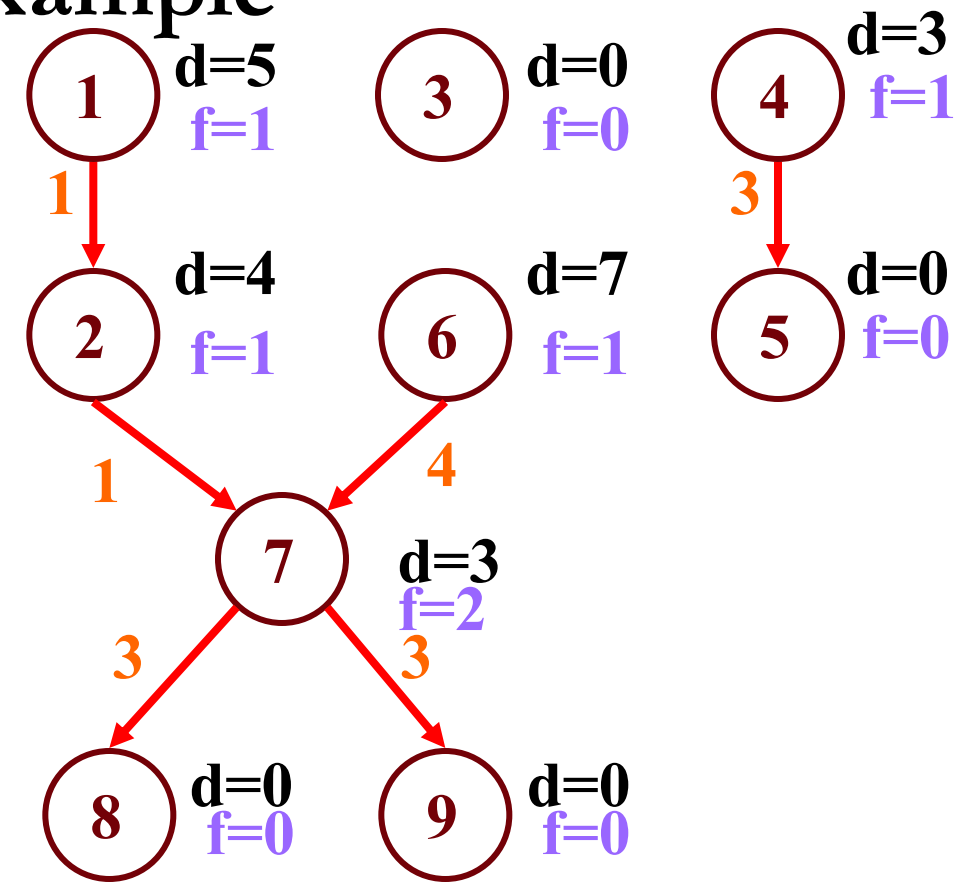


6	1	2	4	7

Example

8, 9

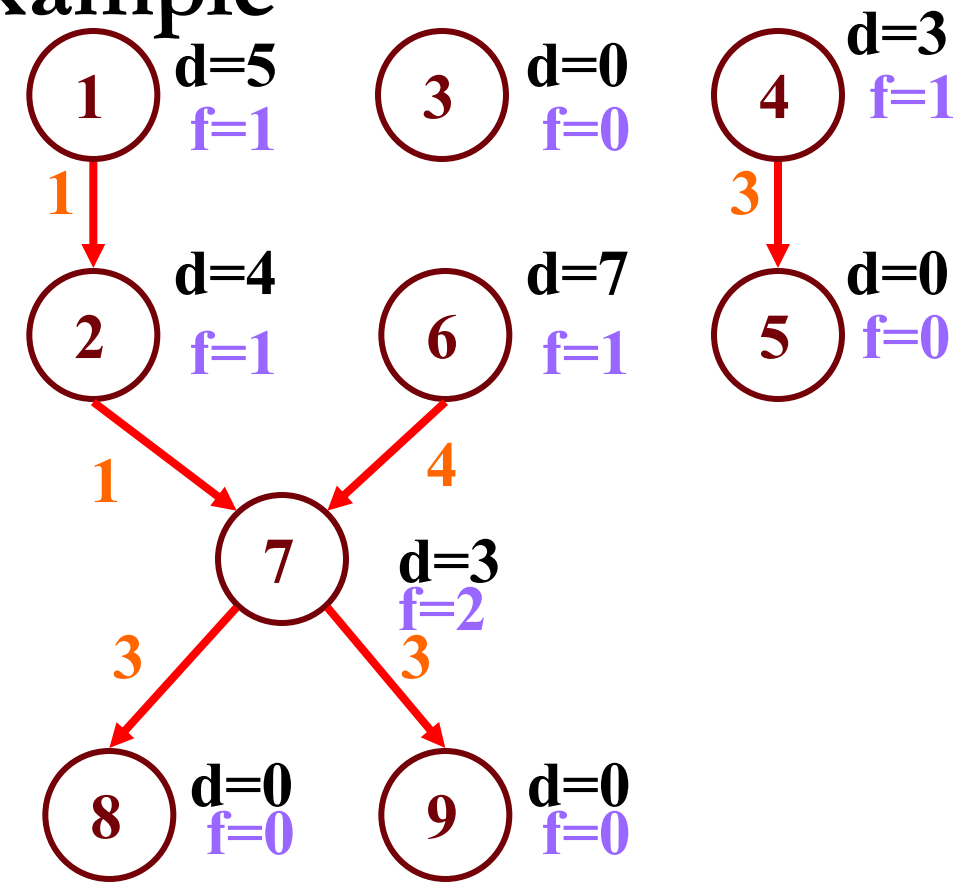
READY = { 3, 5 }



6	1	2	4	7

Example

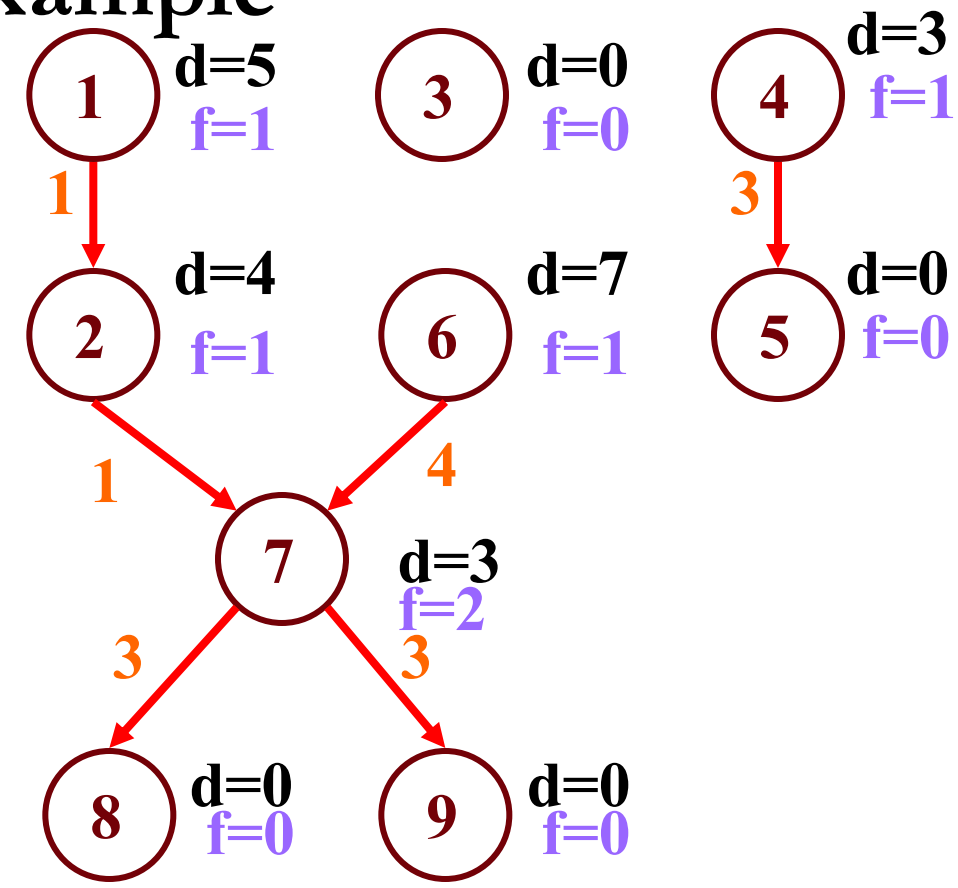
READY = {3, 5, 8, 9}



6	1	2	4	7

Example

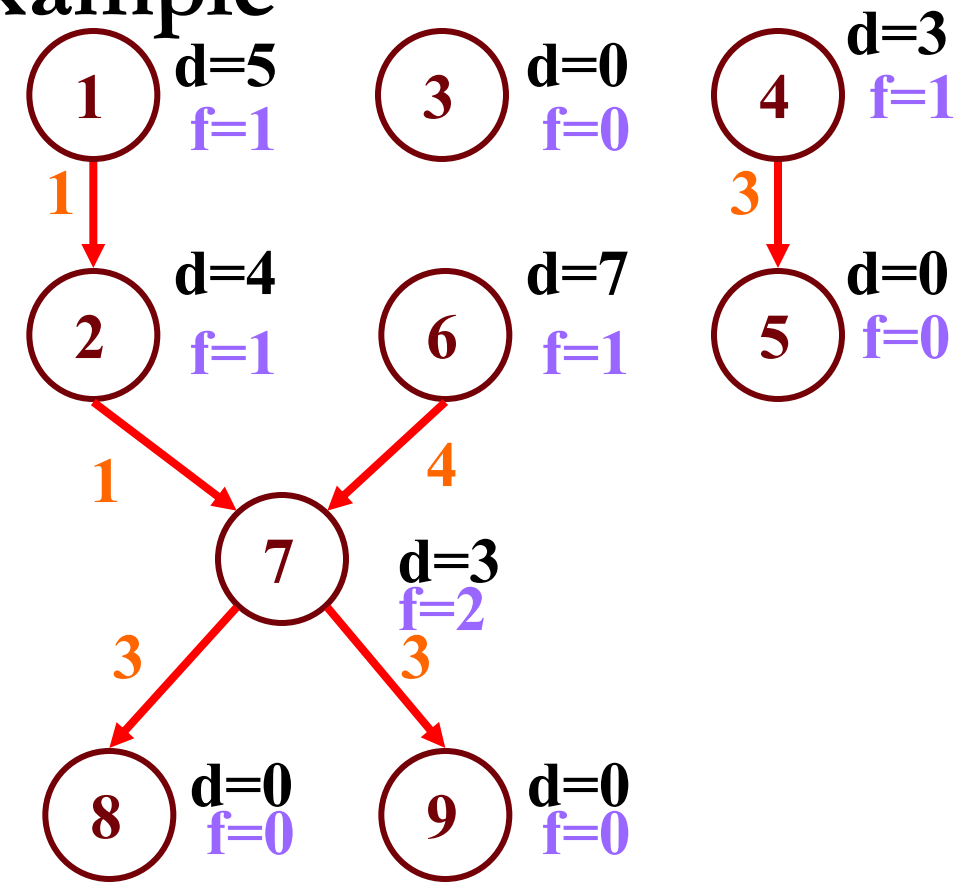
READY = {3, 5, 8, 9}



6	1	2	4	7

Example

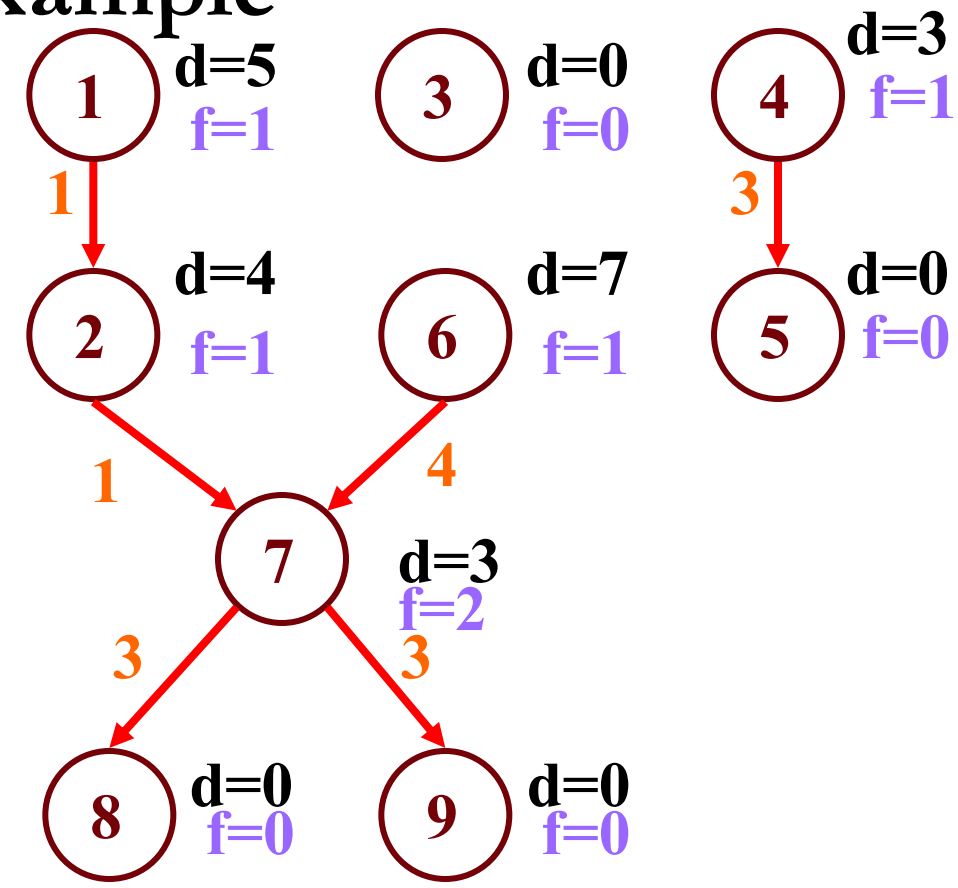
READY = {3, 5, 8, 9}



6	1	2	4	7	3

Example

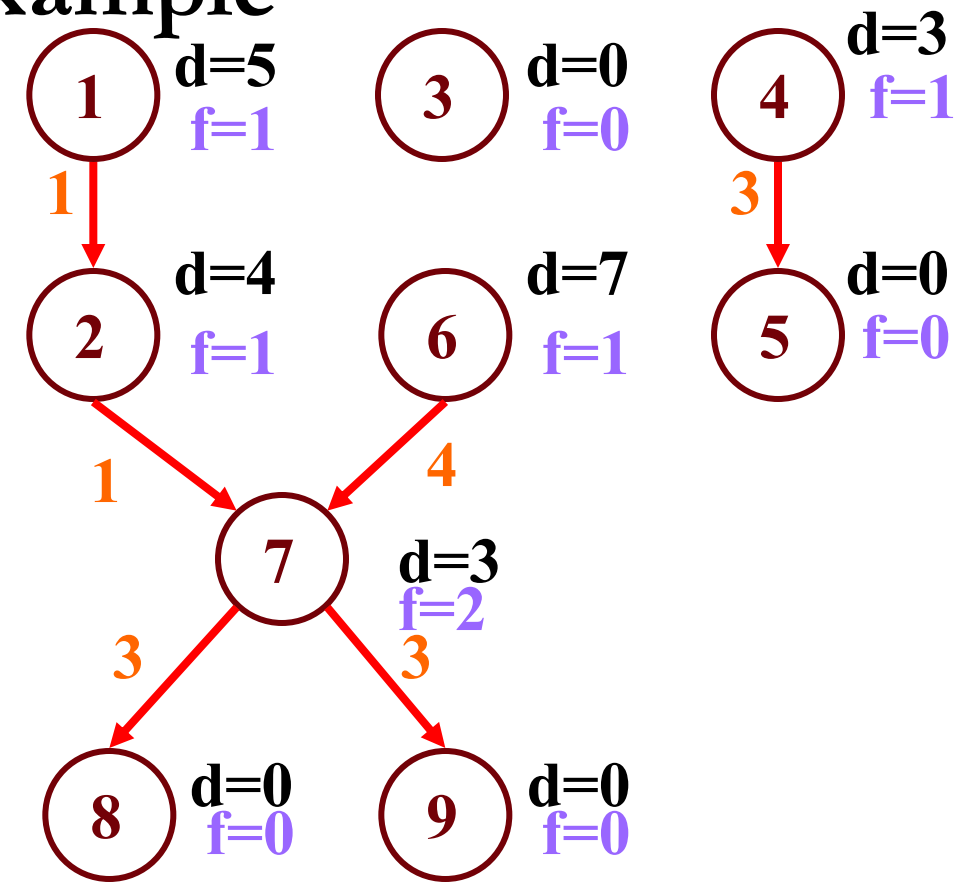
READY = { 5, 8, 9 }



6	1	2	4	7	3

Example

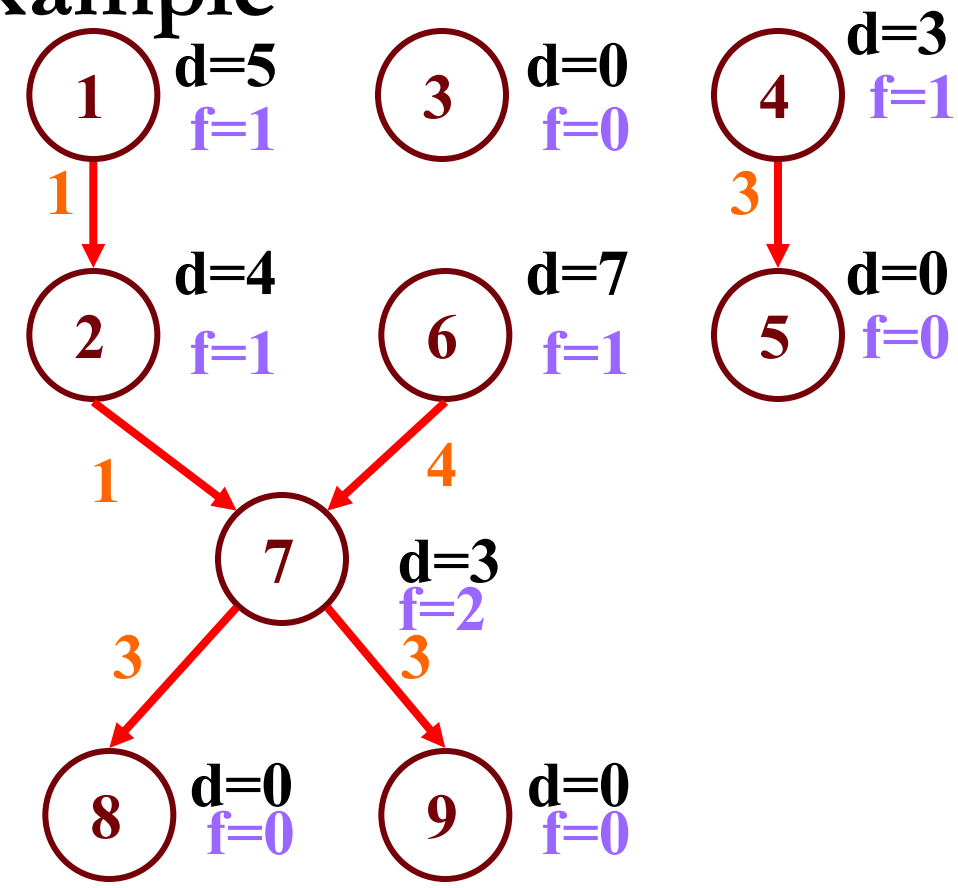
READY = { 5, 8, 9 }

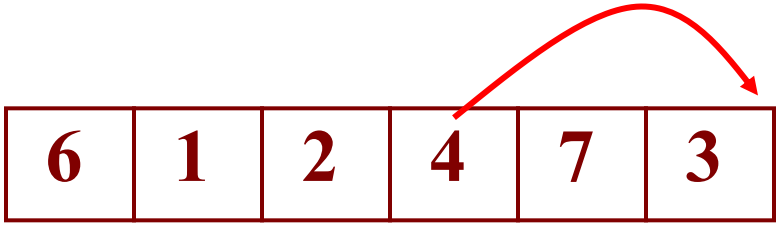


6	1	2	4	7	3

Example

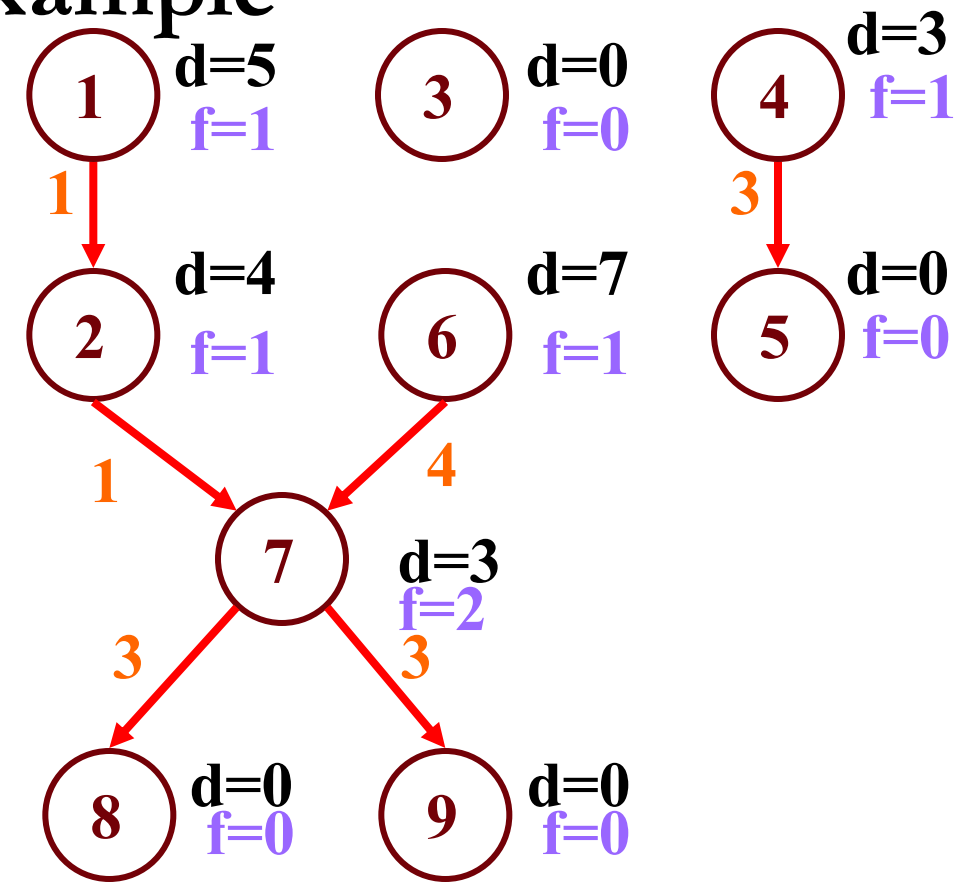
READY = { 5, 8, 9 }





Example

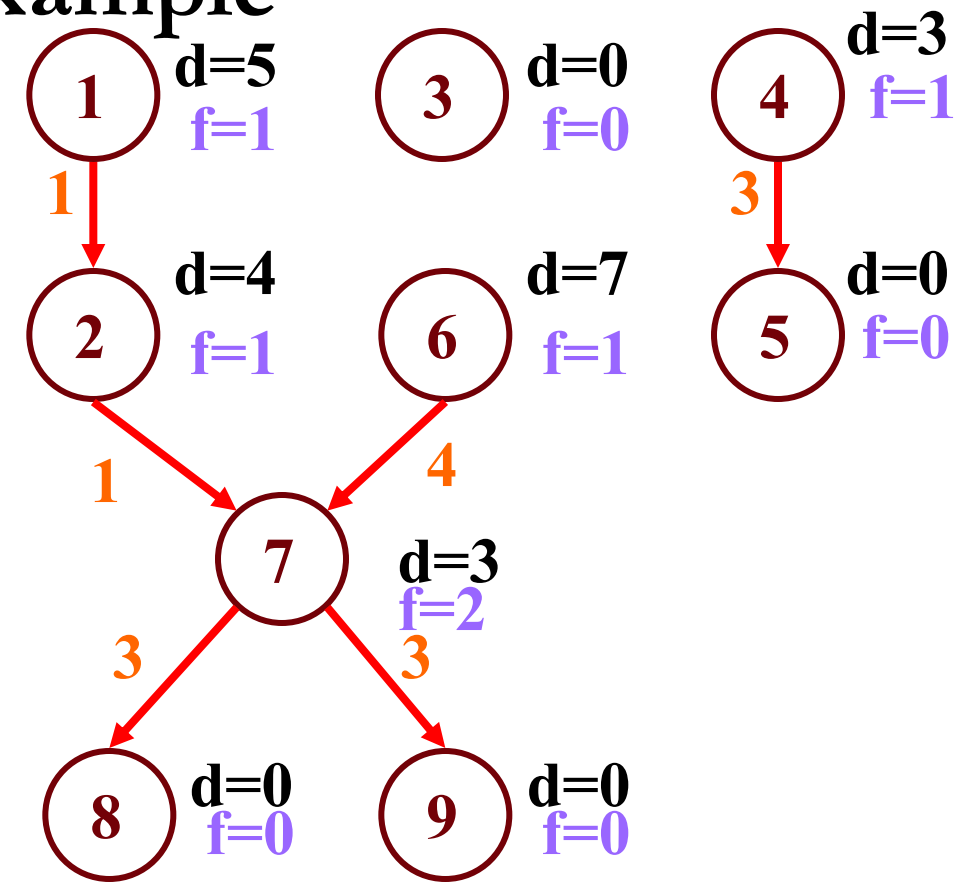
READY = { 5, 8, 9 }



6	1	2	4	7	3	5

Example

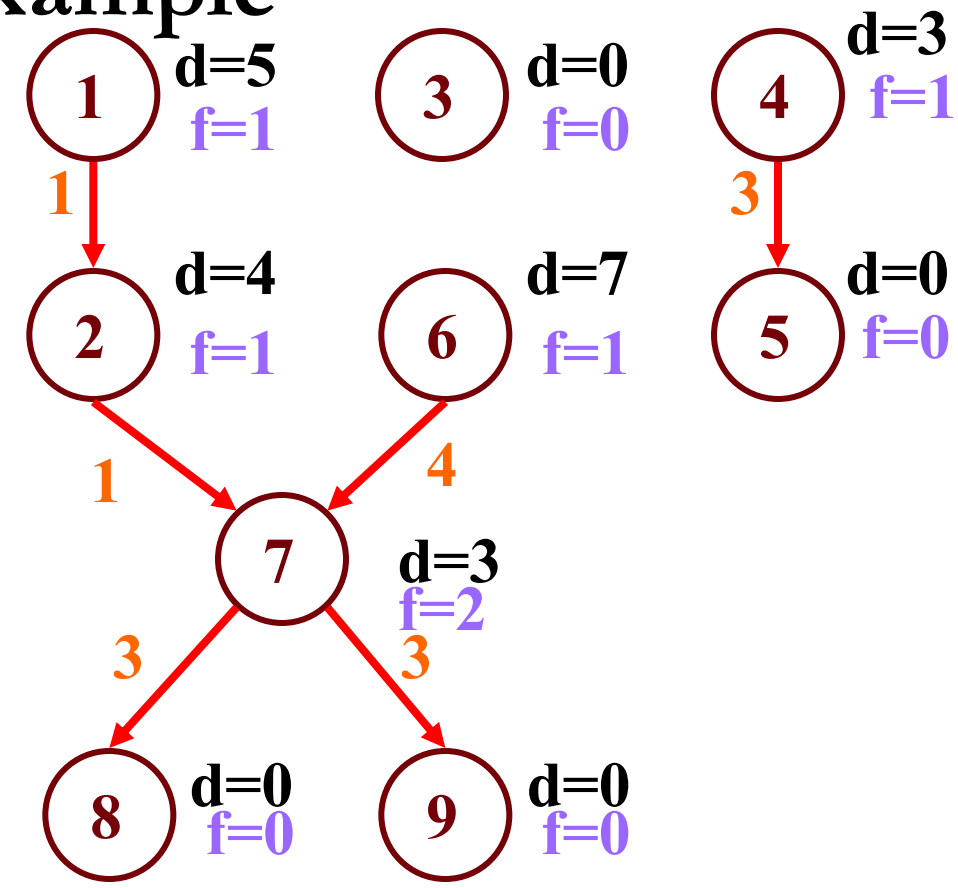
READY = { 8, 9 }



6	1	2	4	7	3	5

Example

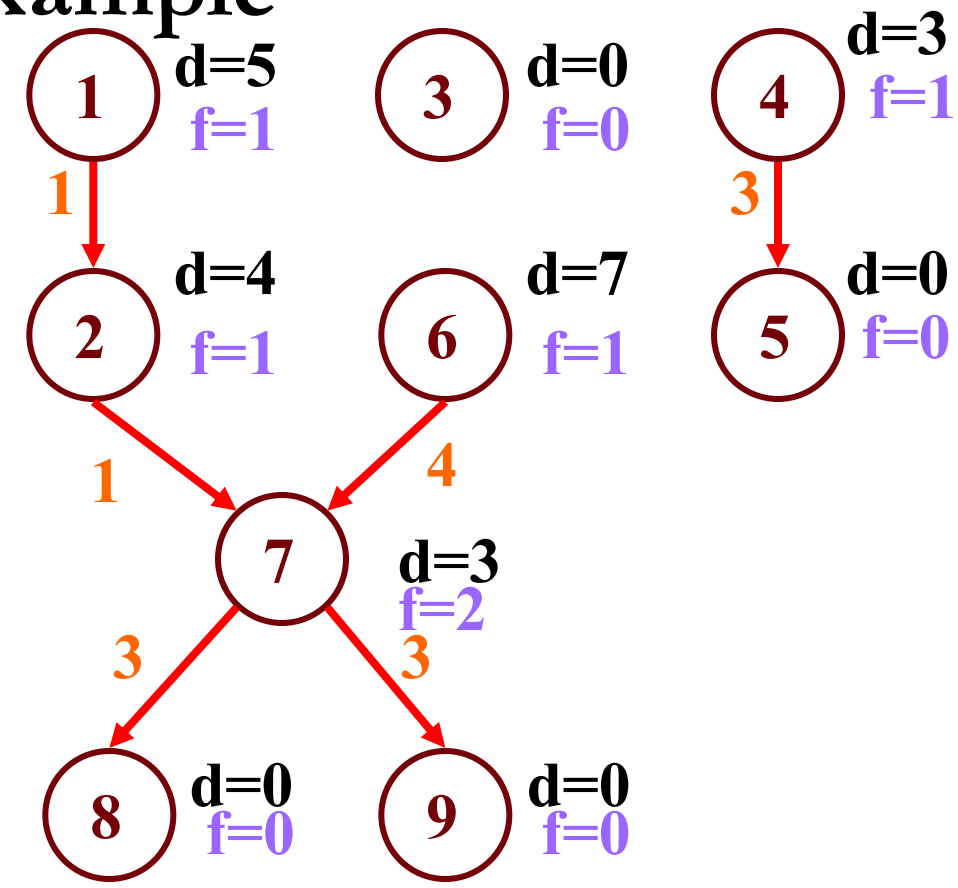
READY = { 8, 9 }



6	1	2	4	7	3	5

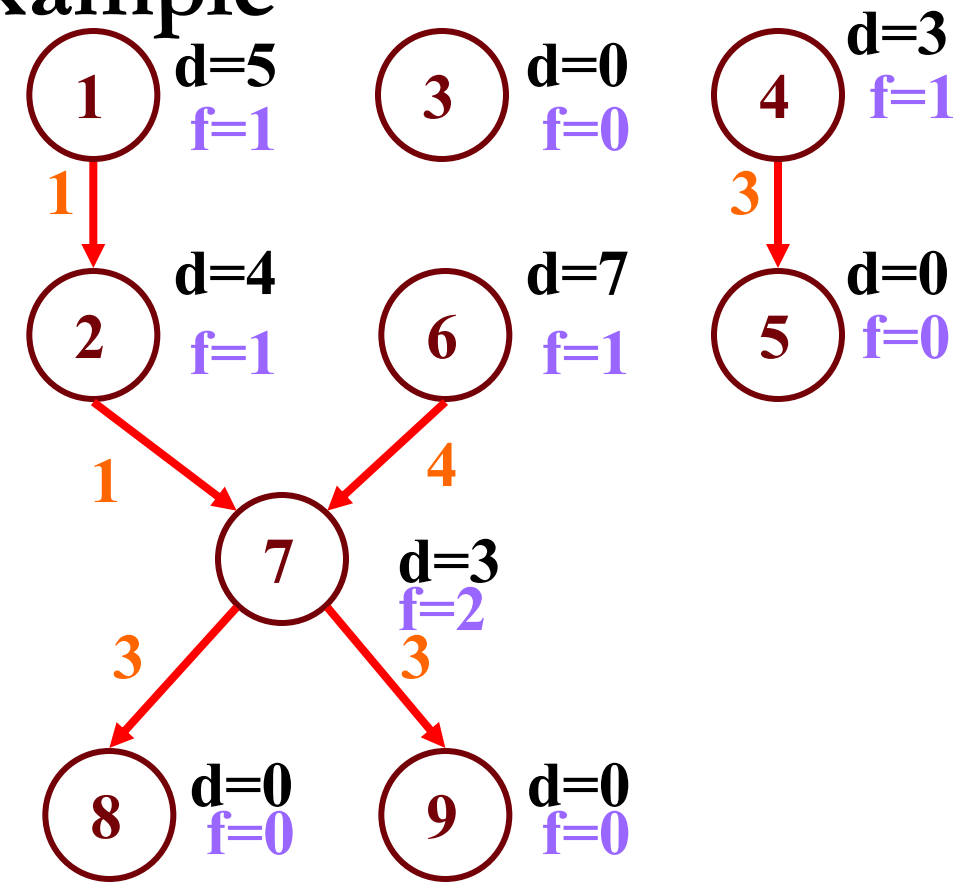
Example

READY = { 8, 9 }



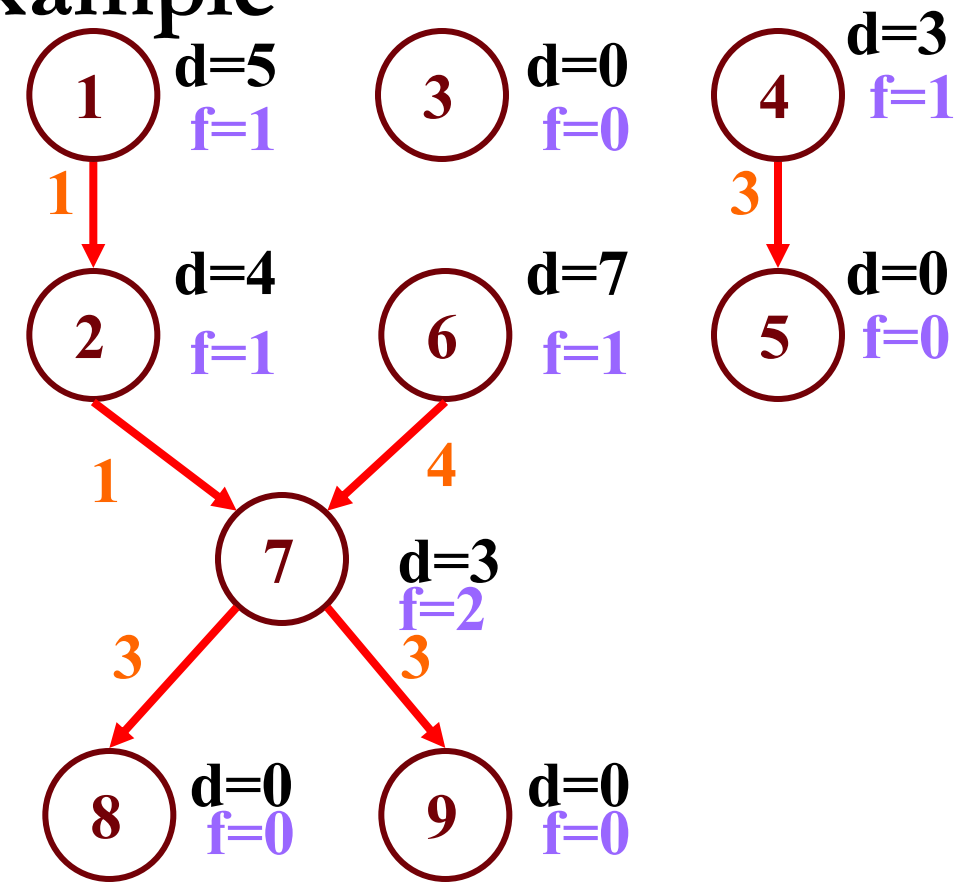
Example

READY = { 8, 9 }



Example

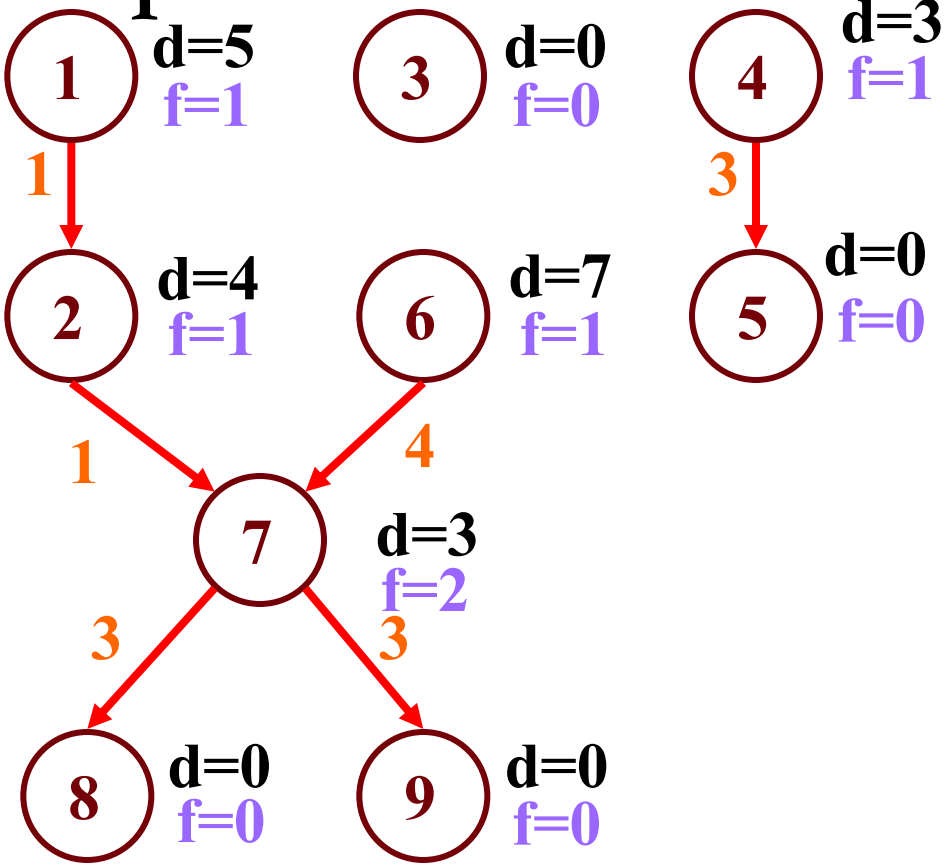
READY = { 8, 9 }



6	1	2	4	7	3	5	8

Example

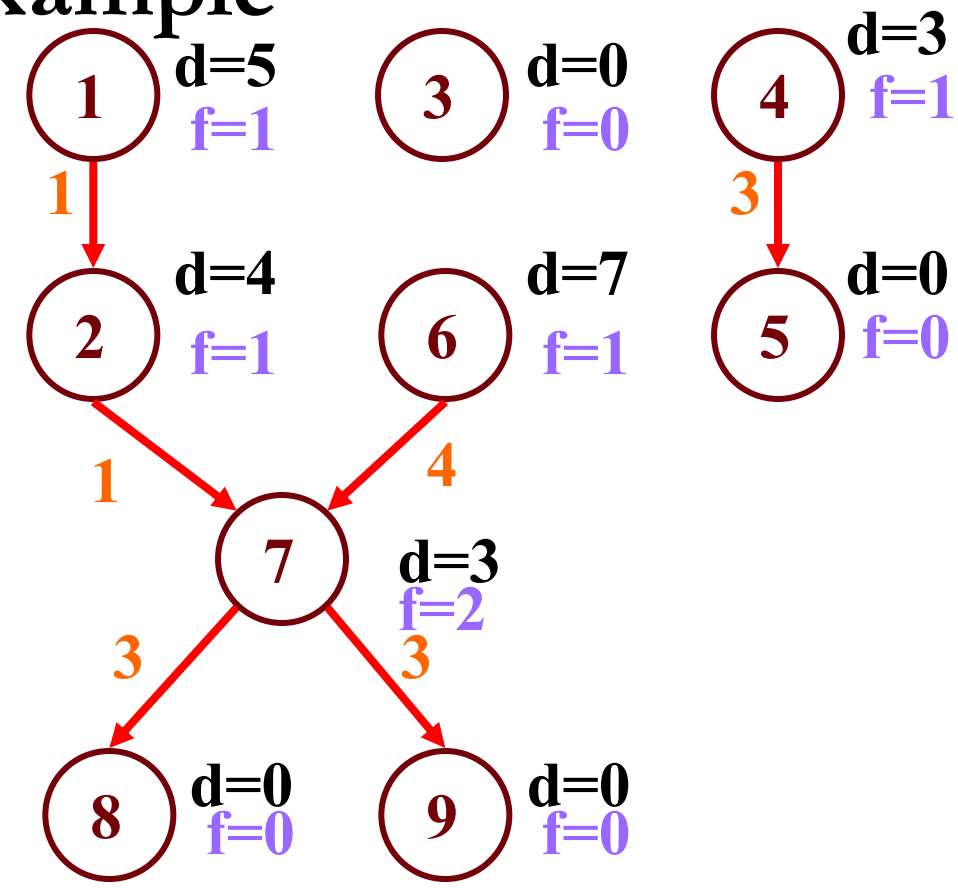
READY = { 9 }



6	1	2	4	7	3	5	8

Example

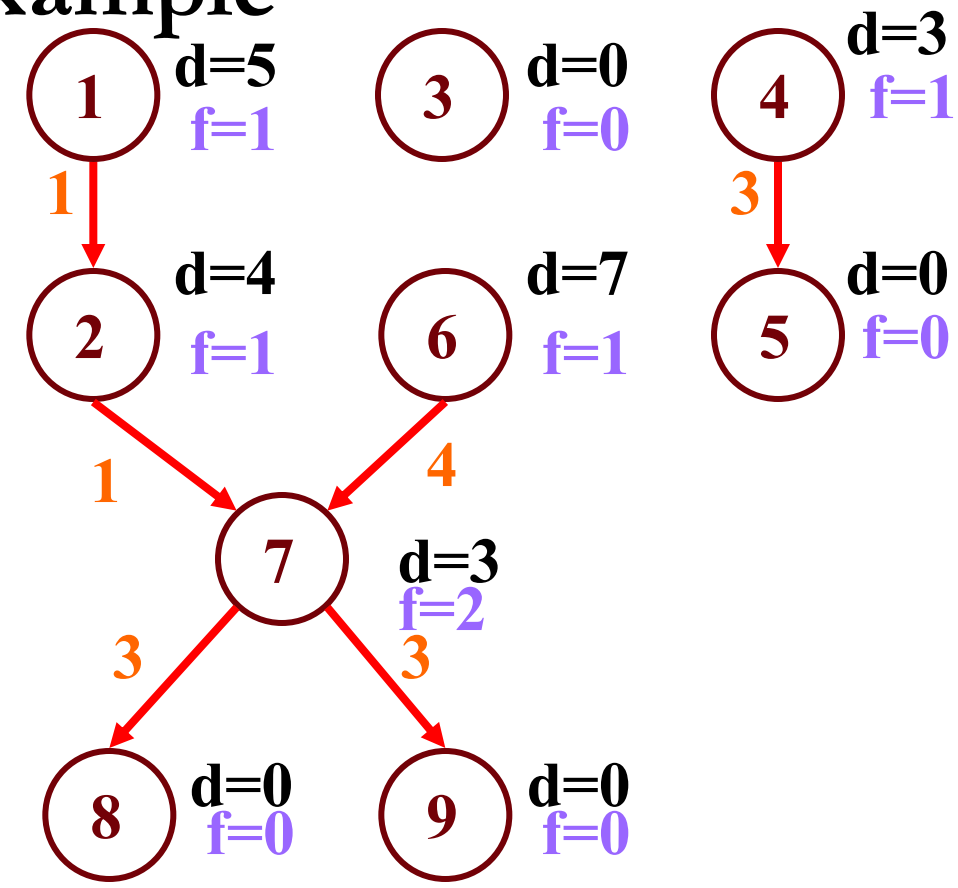
READY = { 9 }



6	1	2	4	7	3	5	8

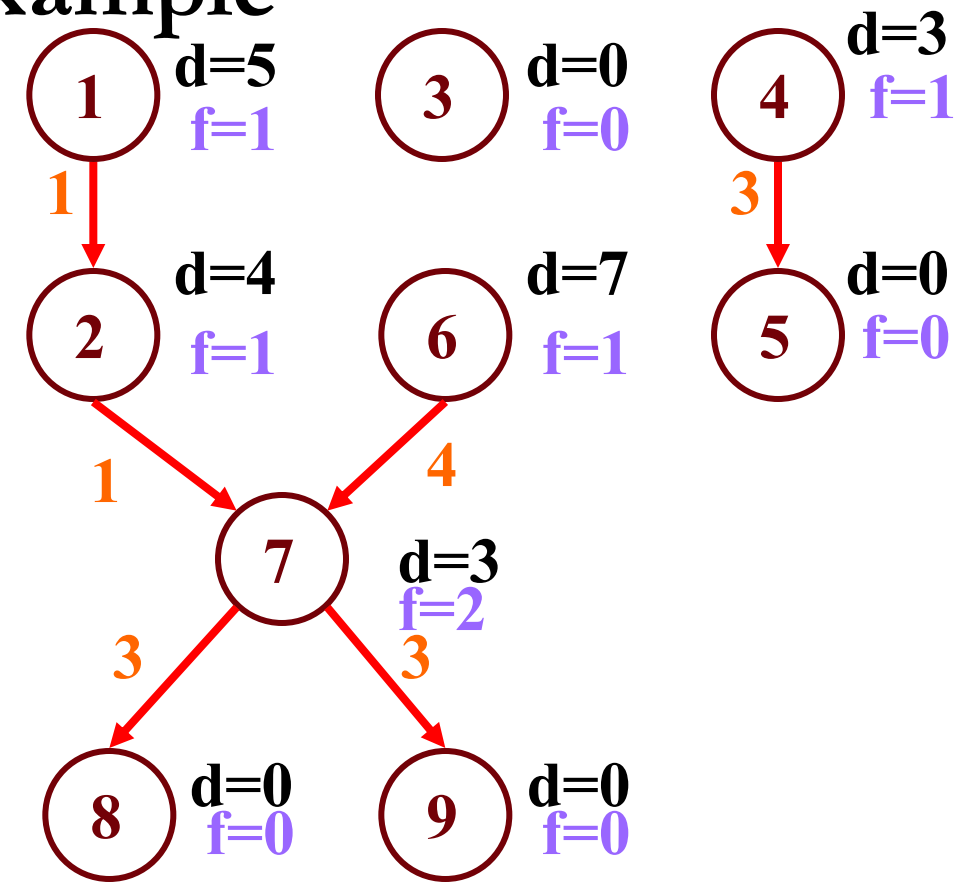
Example

READY = { 9 }



Example

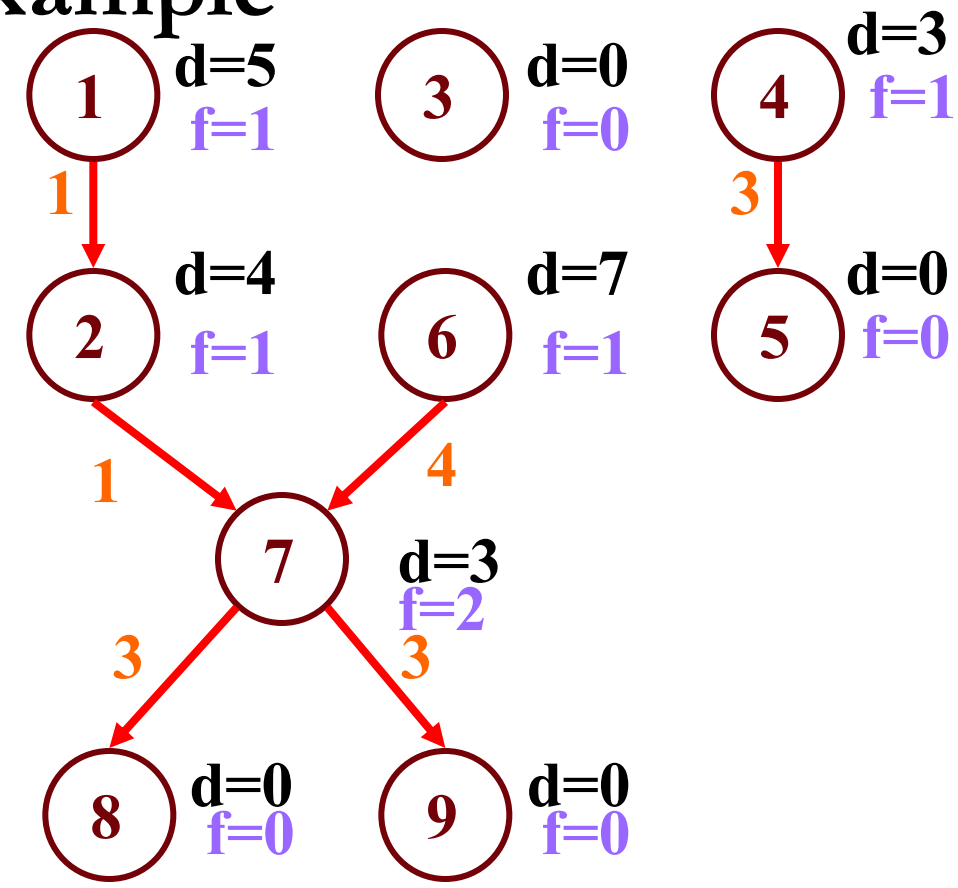
READY = { 9 }



6	1	2	4	7	3	5	8	9

Example

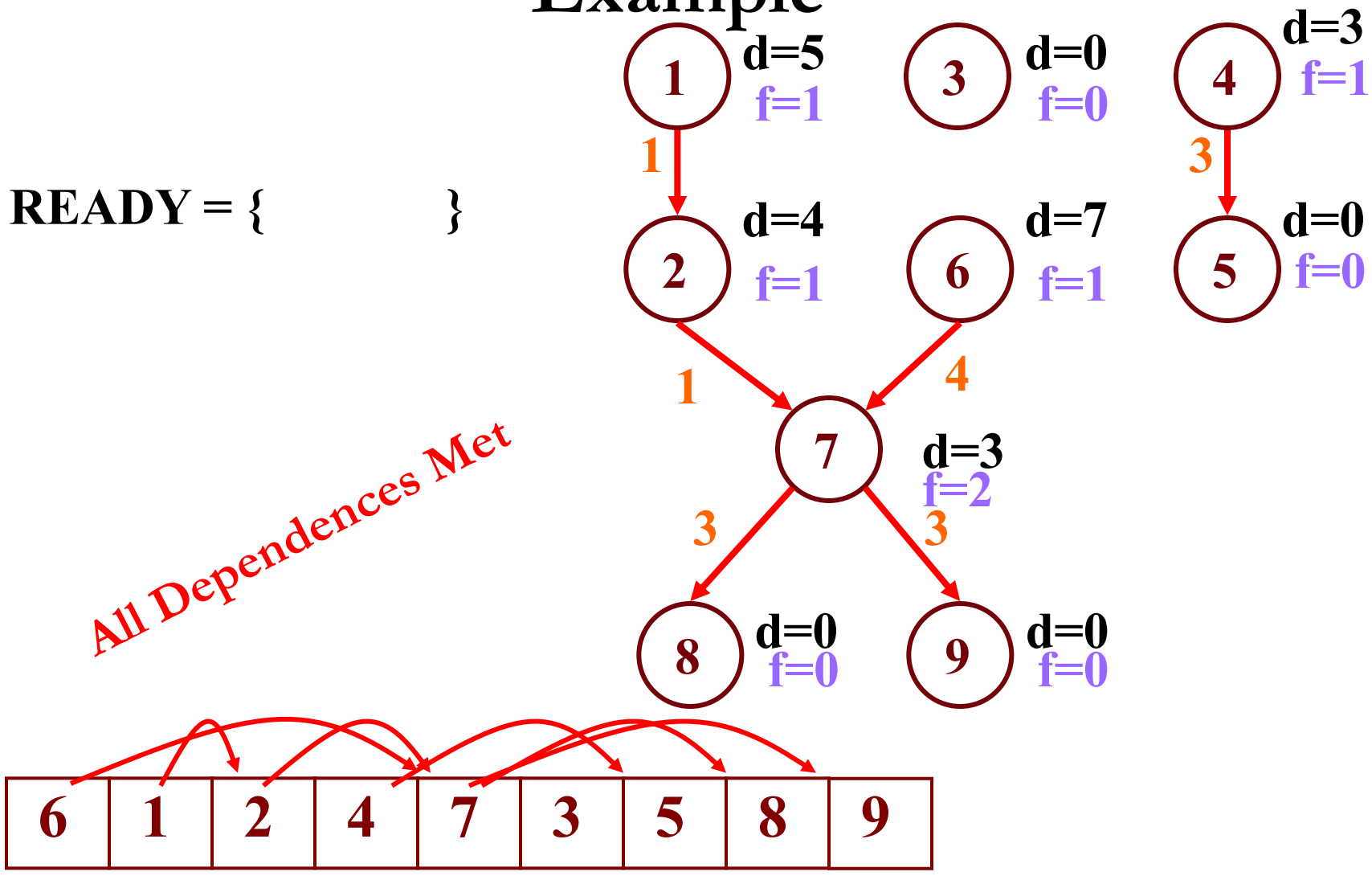
READY = { }



6	1	2	4	7	3	5	8	9

Example

READY = { }



Outline

- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across basic blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining

List Scheduling w. Resource Constraints

- Create a Dependence DAG of a Basic Block
- Topological Sort

READY = nodes with no predecessors
Loop until READY list is empty
    Let n ∈ READY list be the node with the highest priority
    Schedule n in the earliest slot
        that satisfies precedence + resource constraints
    Update READY list

Constraints of a Superscalar Processor

- Example:
  - 1 integer Operation

```
ALU.op dest, src1, src2    # in 1 clock cycle
```

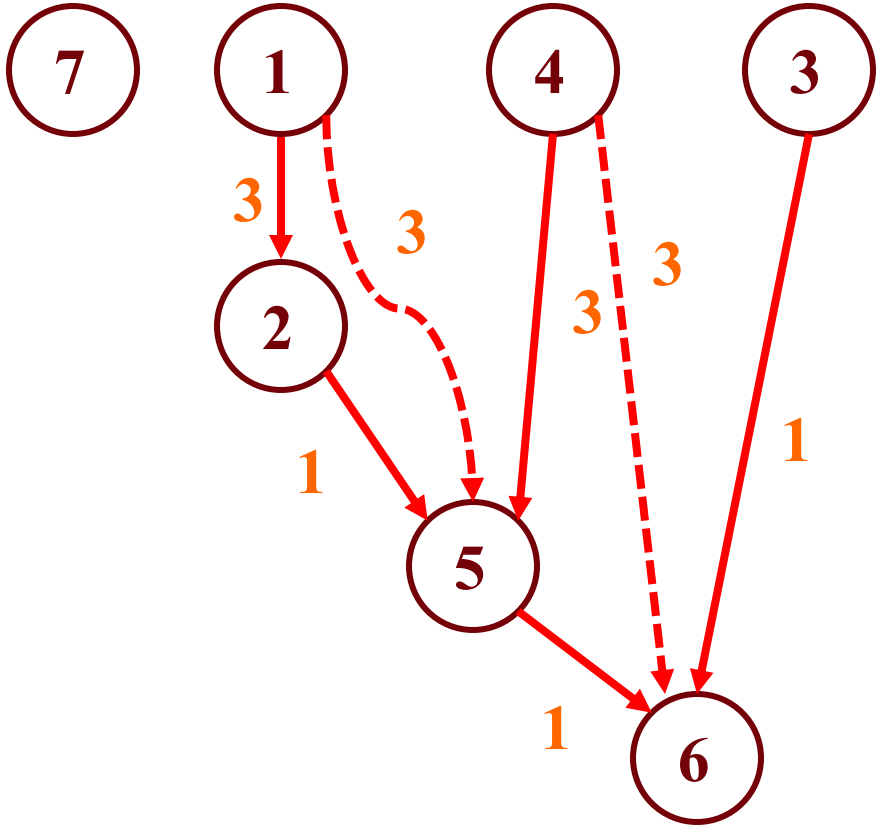
In parallel with

  - 1 Memory Operation

```
ld dst, addr               # in 2 clock cycles
st src, addr               # in 1 clock cycle
```

List Scheduling Example

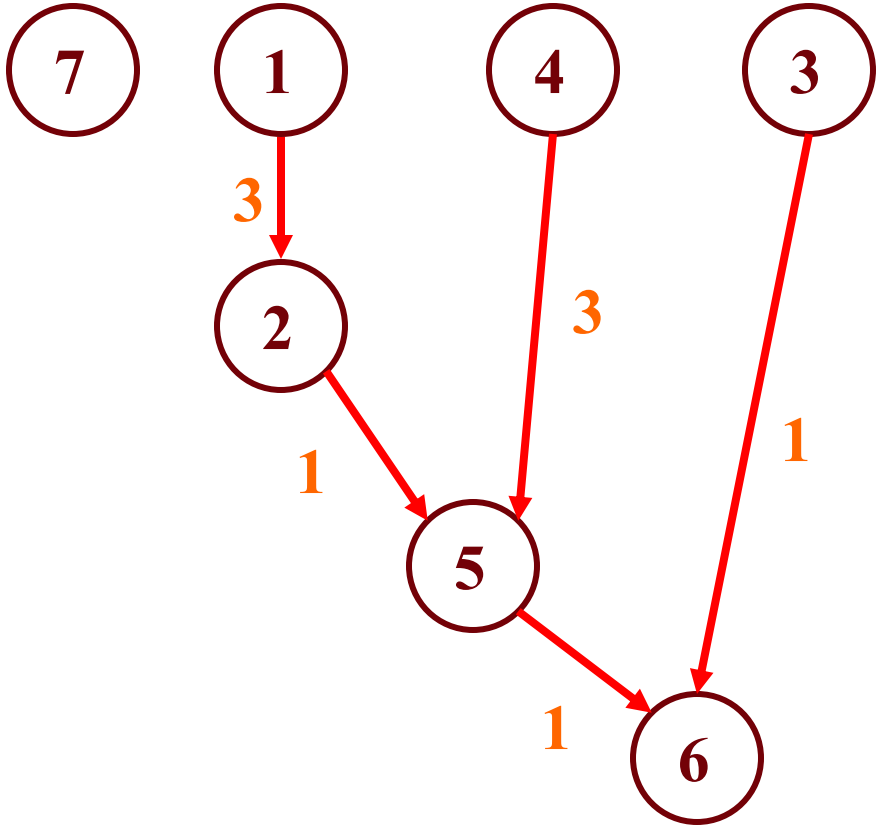
```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```



List Scheduling Example

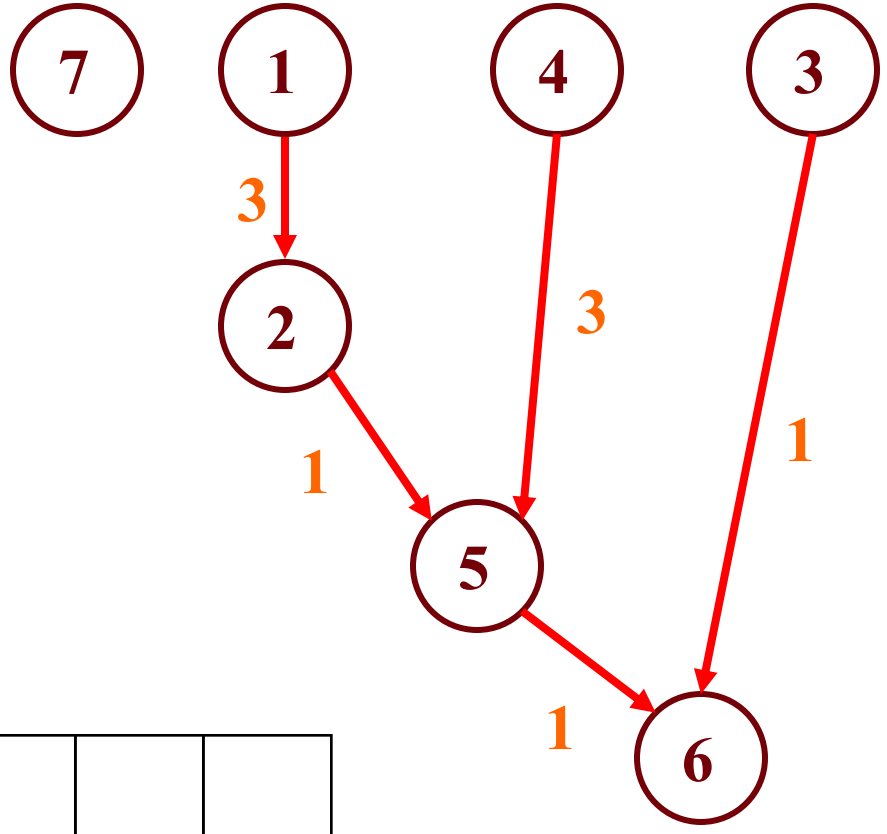
```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

READY= {1, 3, 4, 7}



List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

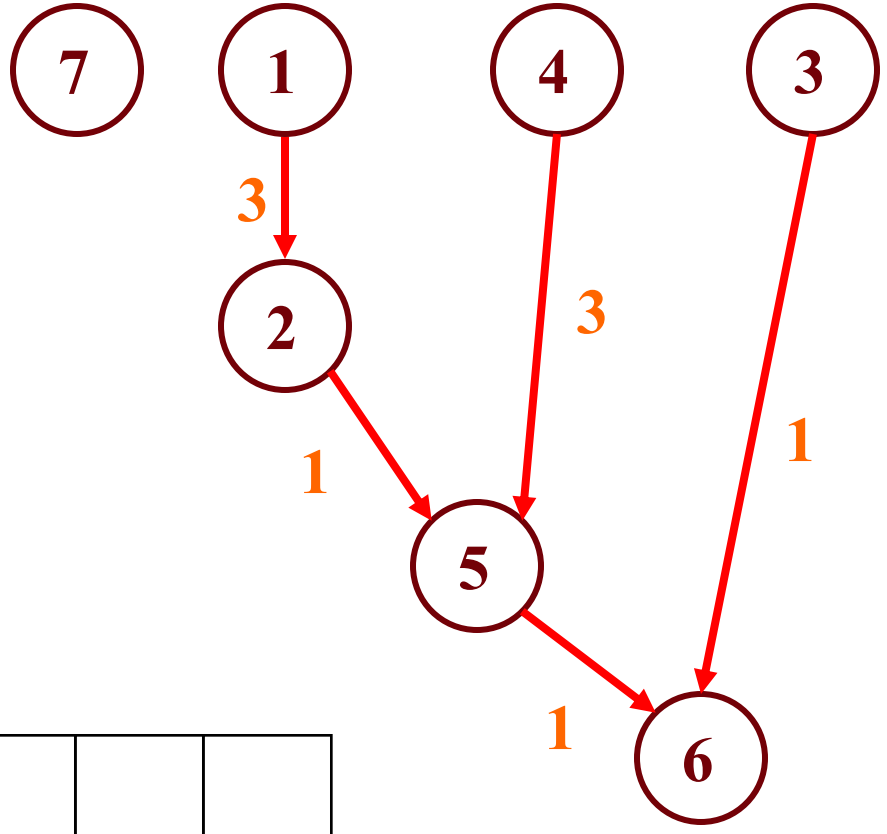


READY= {1, 3, 4, 7}

ALUop							
MEM 1							
MEM 2							

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

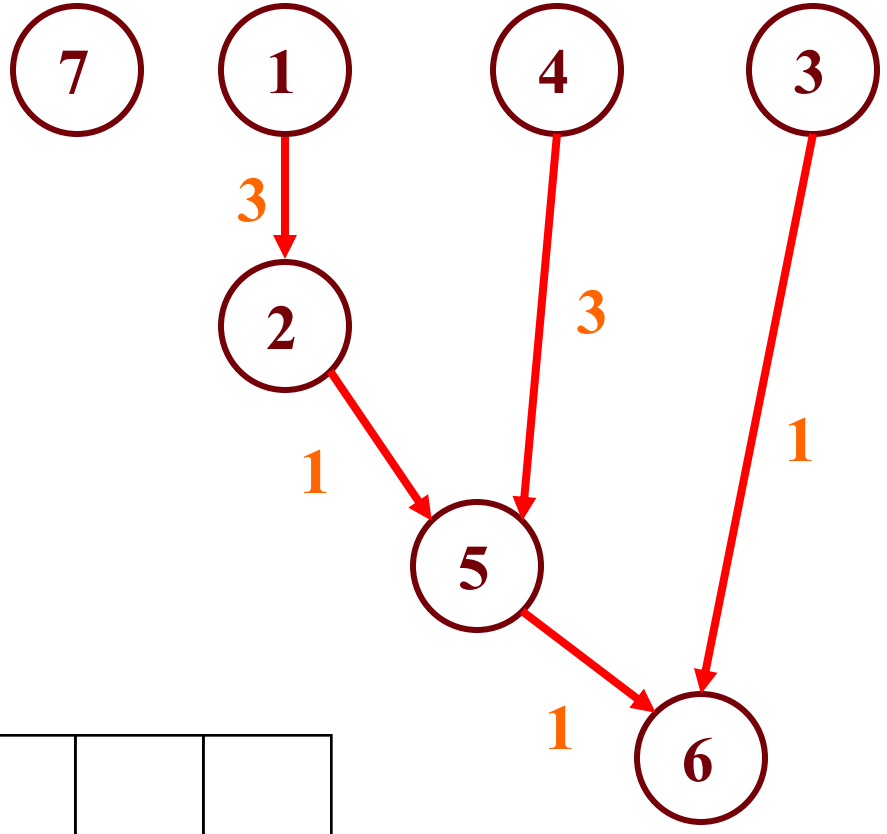


READY= {1, 3, 4, 7}

ALUop							
MEM 1	1						
MEM 2		1					

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

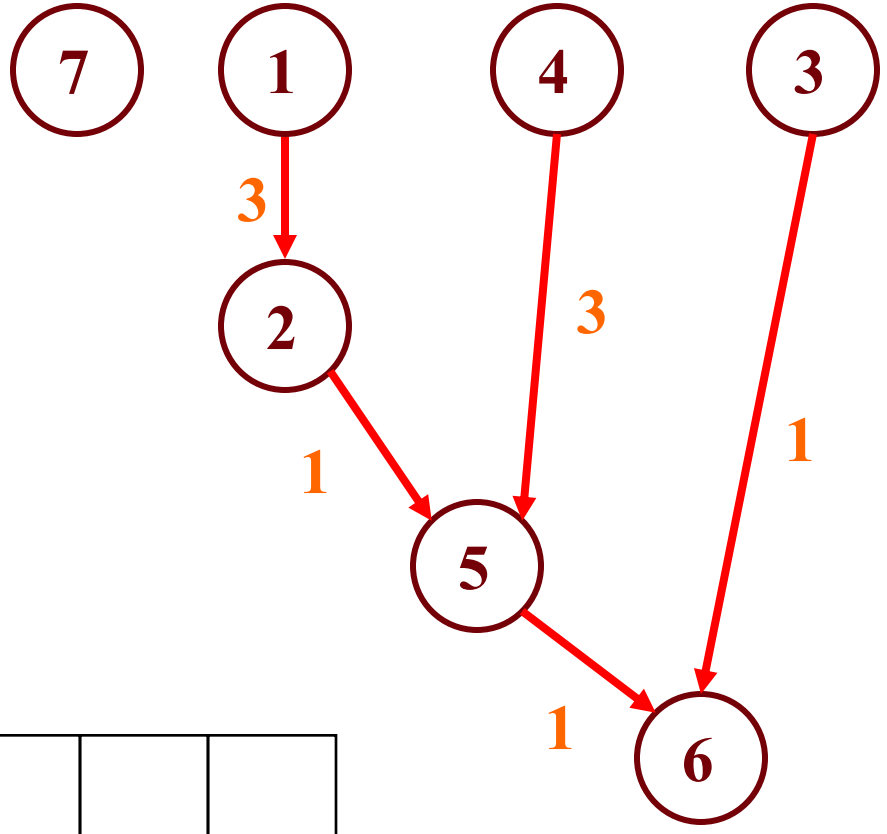


READY= {2, 3, 4, 7}

ALUop							
MEM 1	1						
MEM 2		1					

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

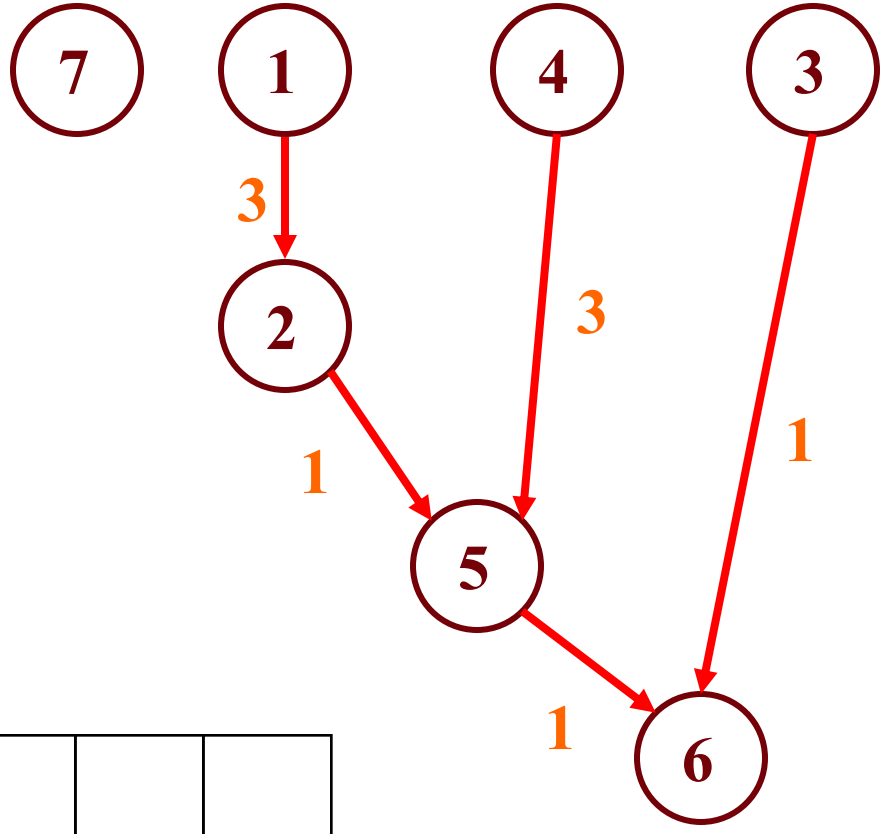


READY= {2, 3, 4, 7}

ALUop				2			
MEM 1	1						
MEM 2		1					

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

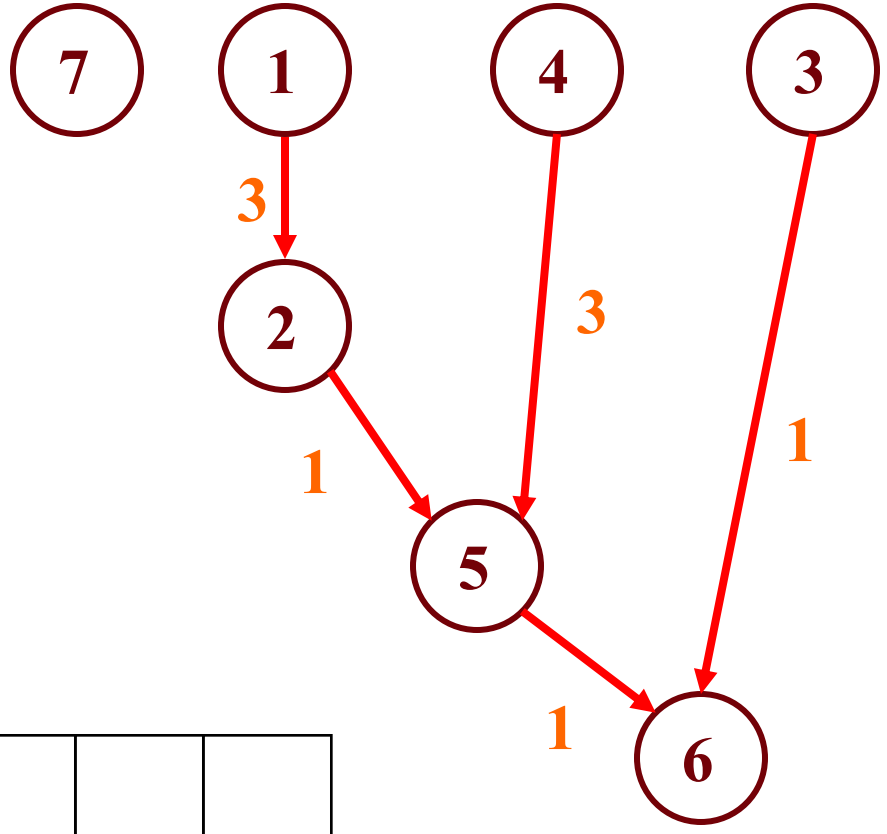


READY= {3, 4, 7}

ALUop				2			
MEM 1	1						
MEM 2		1					

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

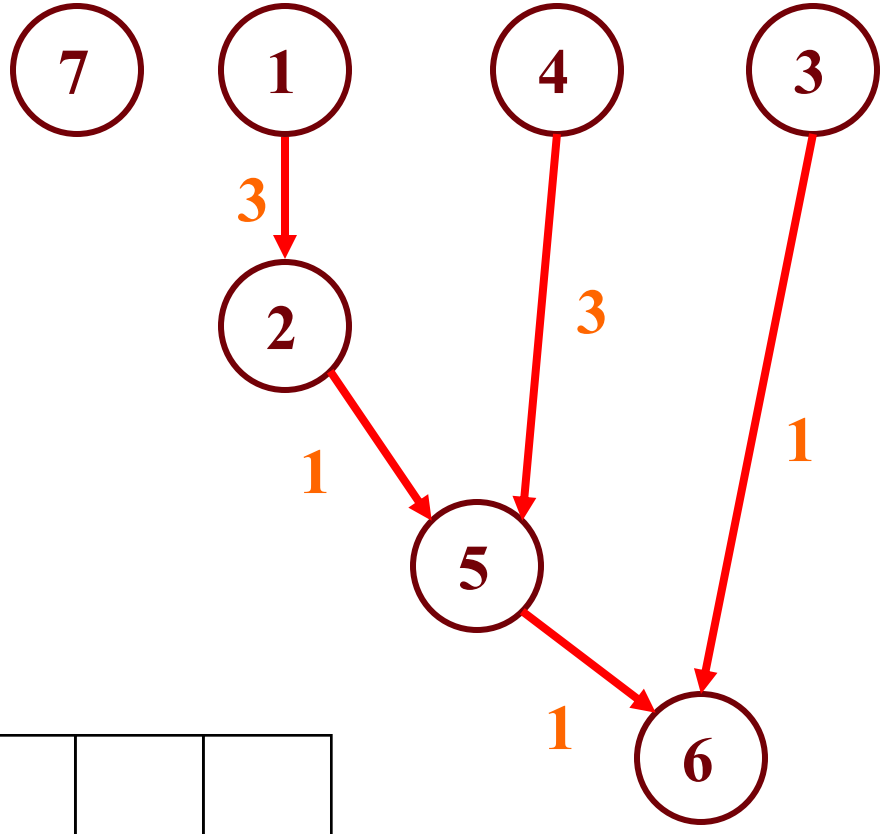


READY= {3, 4, 7}

ALUop				2			
MEM 1	1	4					
MEM 2		1	4				

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

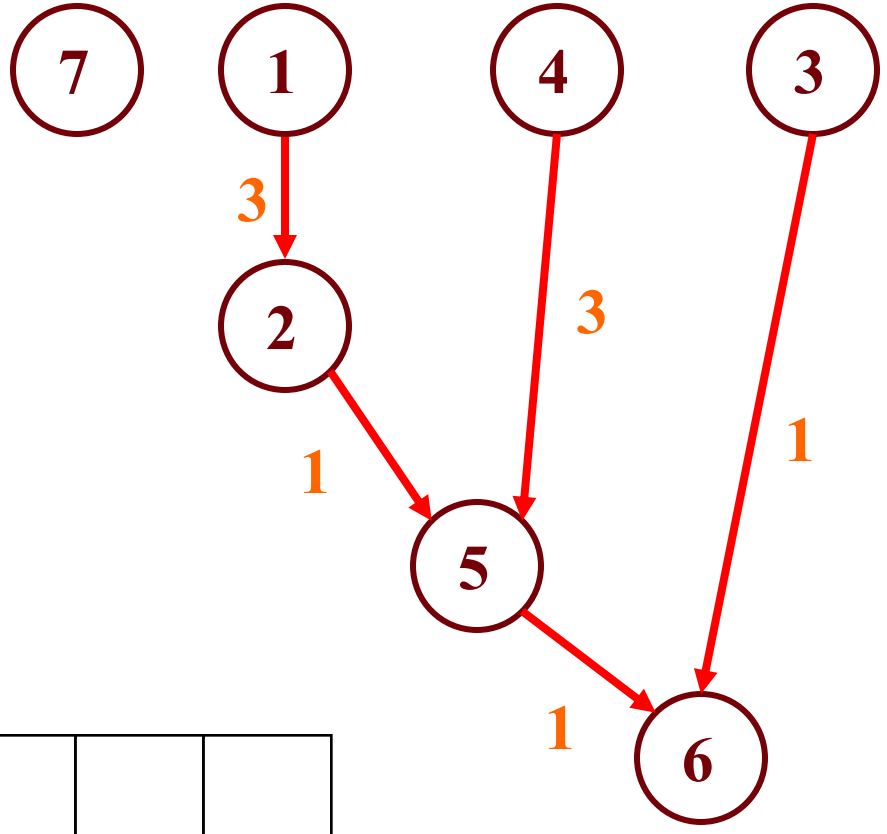


READY= {3, 5, 7}

ALUop				2			
MEM 1	1	4					
MEM 2		1	4				

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

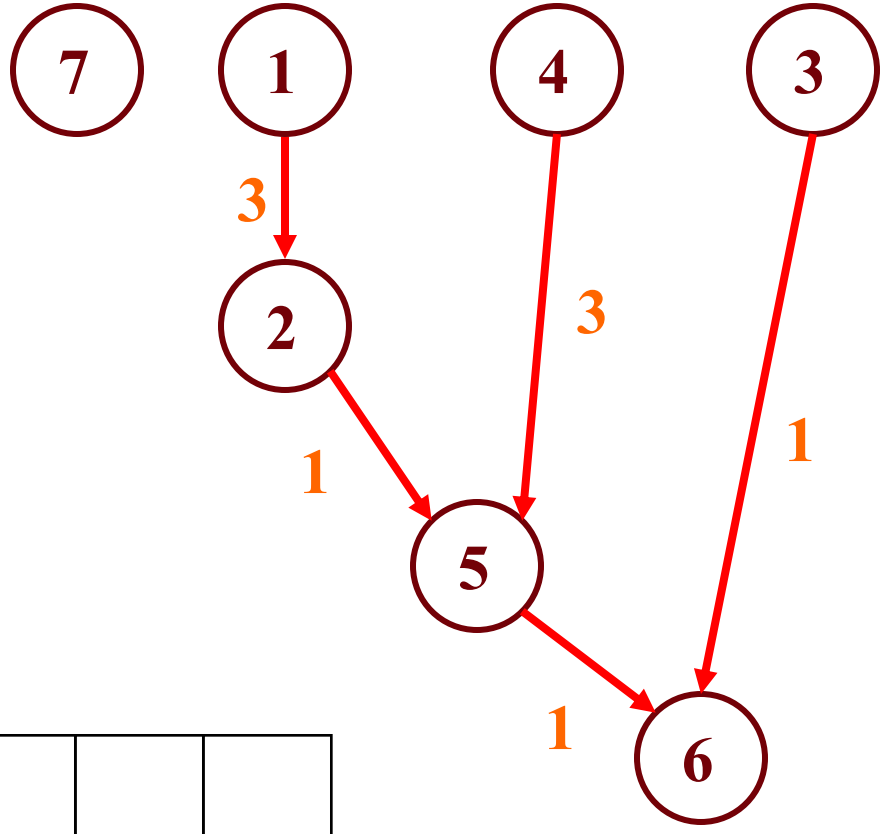


READY= {3, 5, 7}

ALUop				2			
MEM 1	1	4	3				
MEM 2		1	4				

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

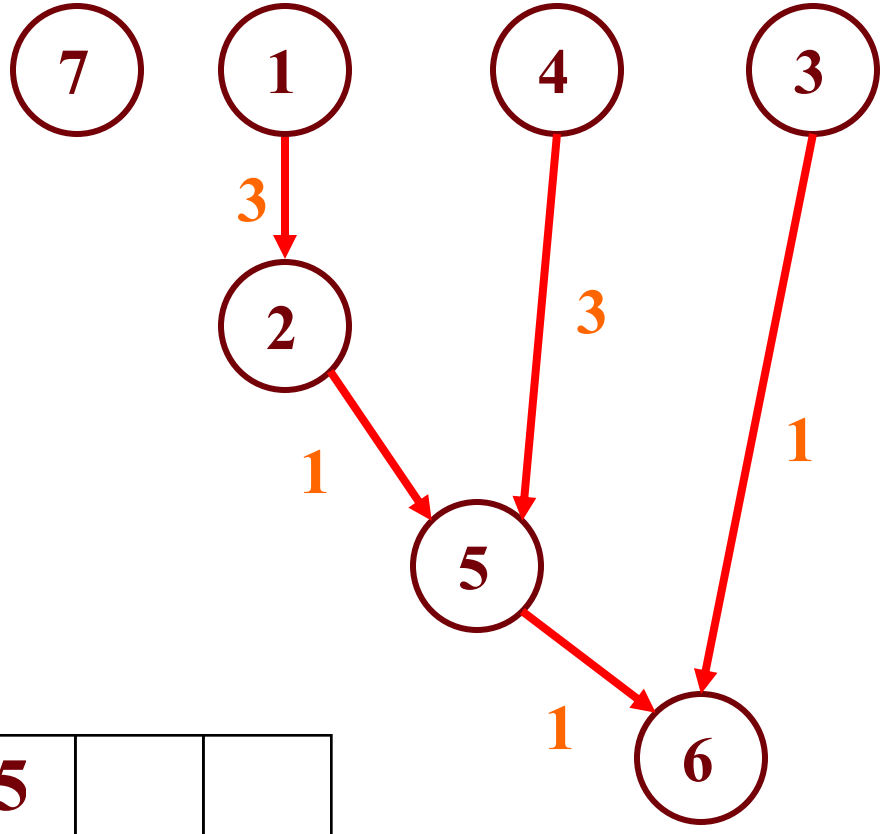


READY= {5, 7}

ALUop				2			
MEM 1	1	4	3				
MEM 2		1	4				

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

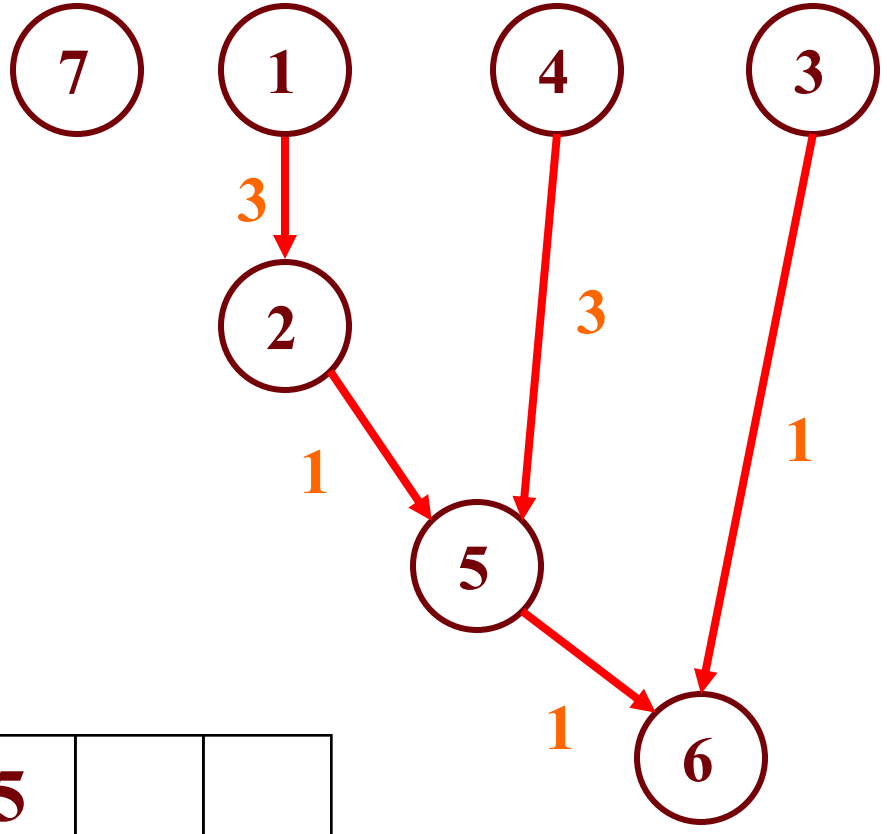


READY= {5, 7}

ALUop				2	5		
MEM 1	1	4	3				
MEM 2		1	4				

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

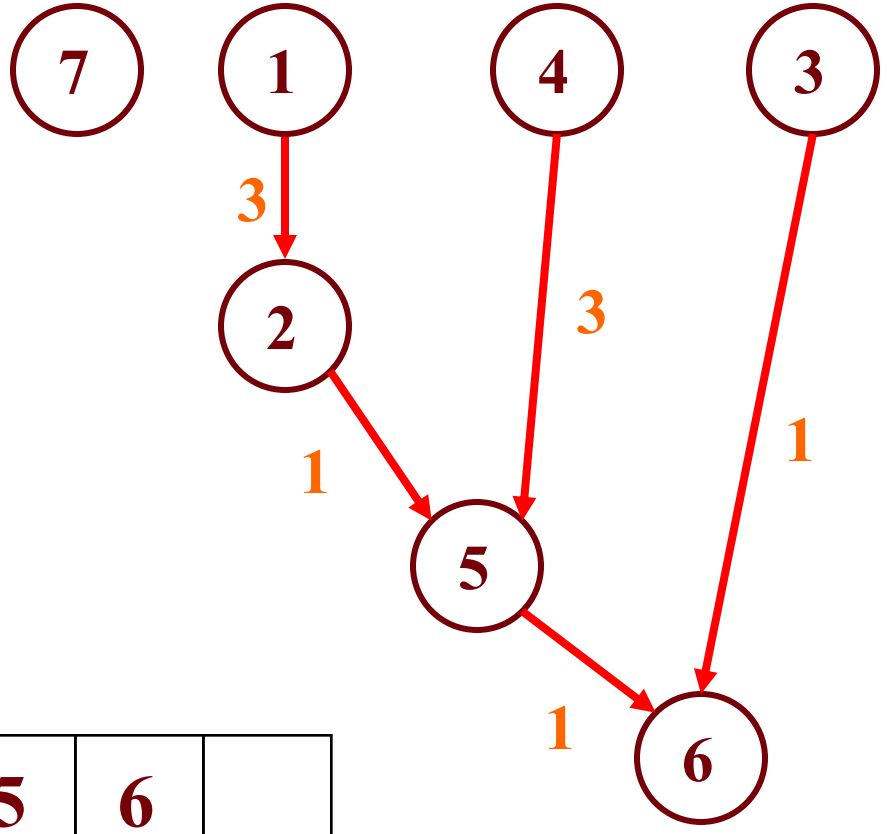


READY= {6, 7}

ALUop				2	5		
MEM 1	1	4	3				
MEM 2		1	4				

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

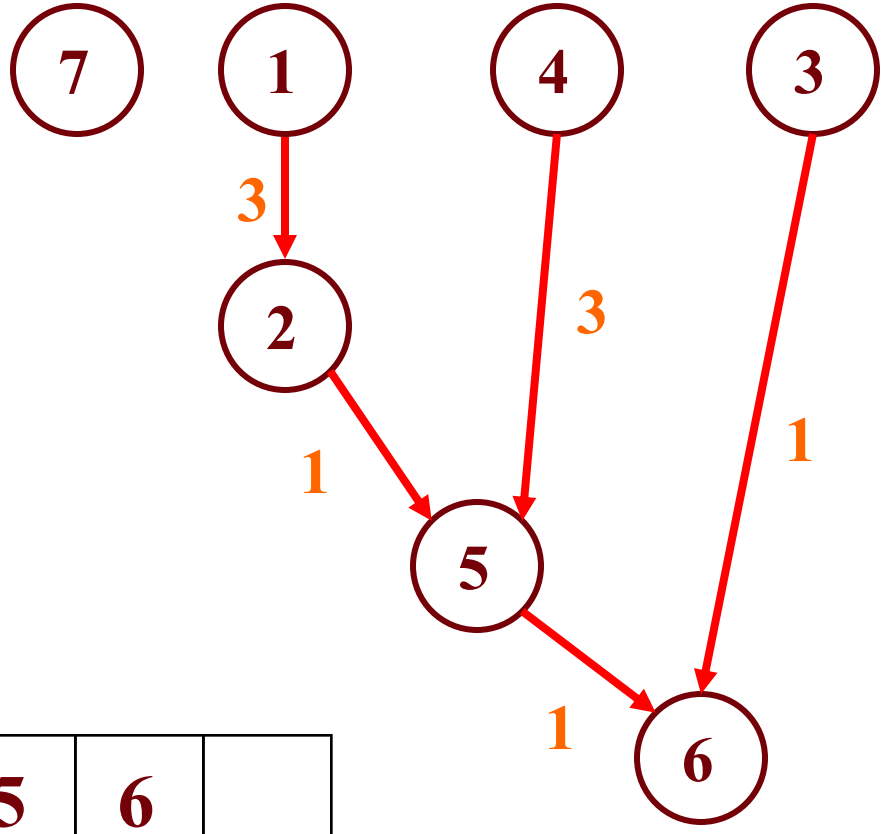


READY= {6, 7}

ALUop				2	5	6	
MEM 1	1	4	3				
MEM 2		1	4				

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

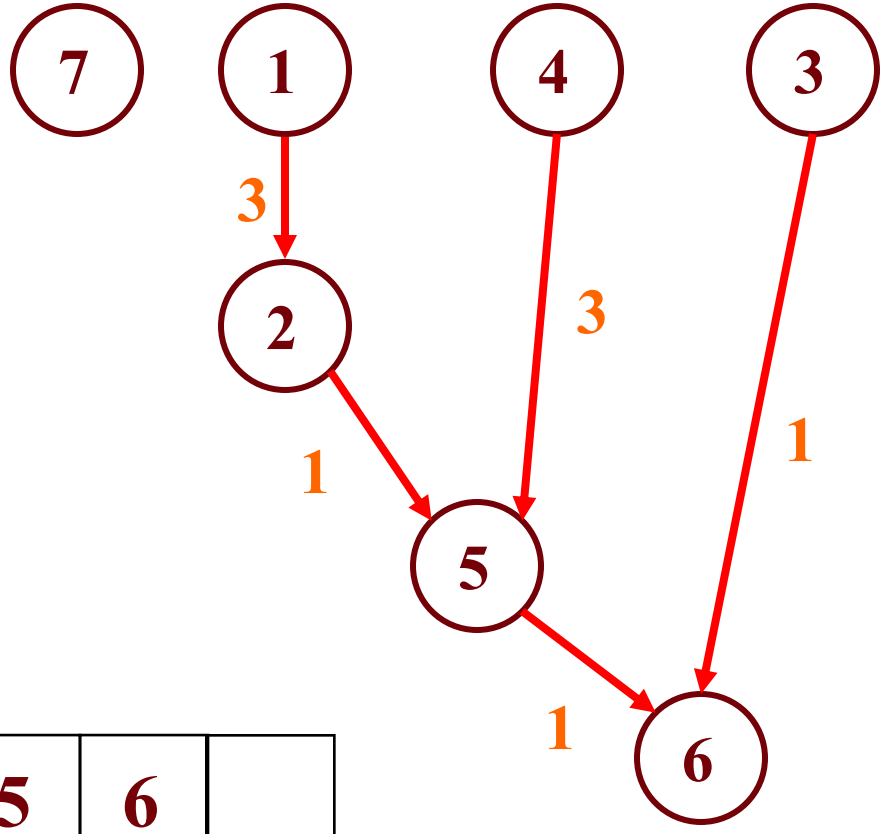


READY= {7}

ALUop				2	5	6	
MEM 1	1	4	3				
MEM 2		1	4				

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```

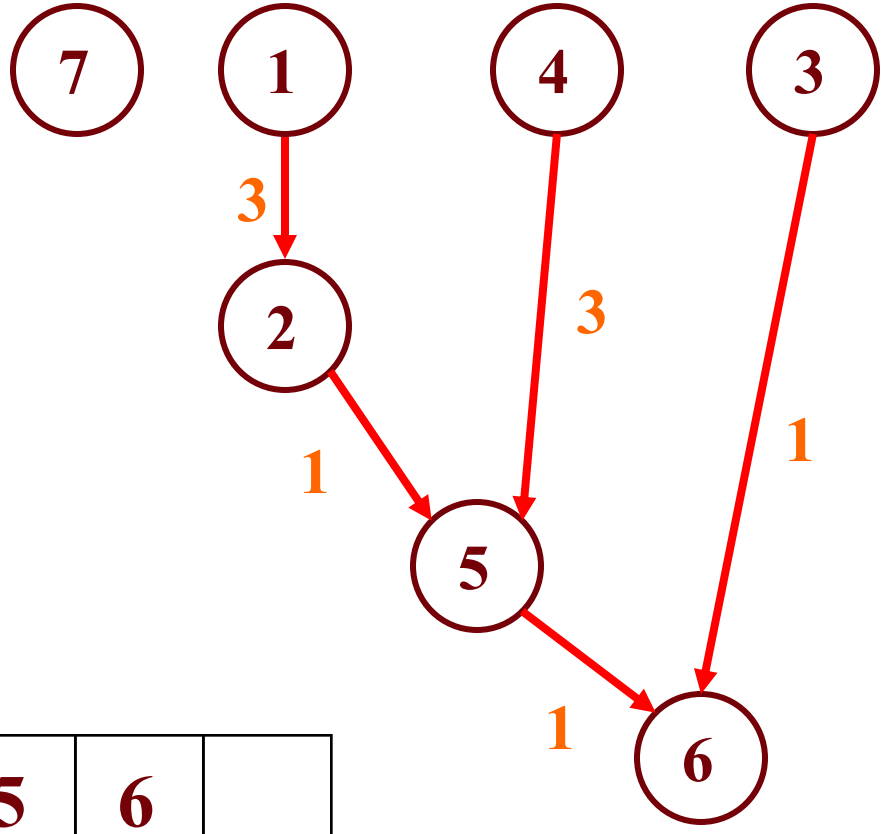


READY= {7}

ALUop				2	5	6	
MEM 1	1	4	3	7			
MEM 2		1	4				

List Scheduling Example

```
1: ld  r2, 0(r1)
2: add r2,r2,r3
3: st  r4,4(r5)
4: ld  r6,8(r1)
5: add r6,r6,r2
6: add r6,r6,r4
7: st  r7,0(r7)
```



READY= { }

ALUop				2	5	6	
MEM 1	1	4	3	7			
MEM 2		1	4				

Outline

- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across Basic Blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining

Register Allocation & Instruction Scheduling

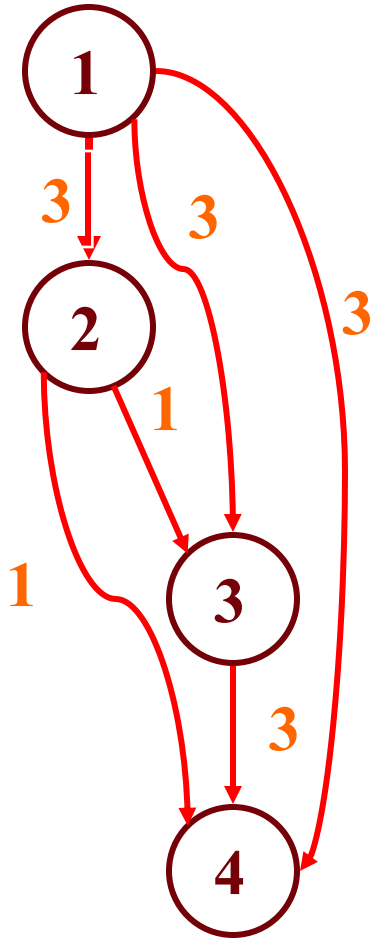
- If Register Allocation is before instruction scheduling
  - restricts the choices for scheduling

Example

```
1: ld  r2, 0(r1)
2: add r3,r3,r2
3: ld  r2,4(r5)
4: add r6,r6,r2
```

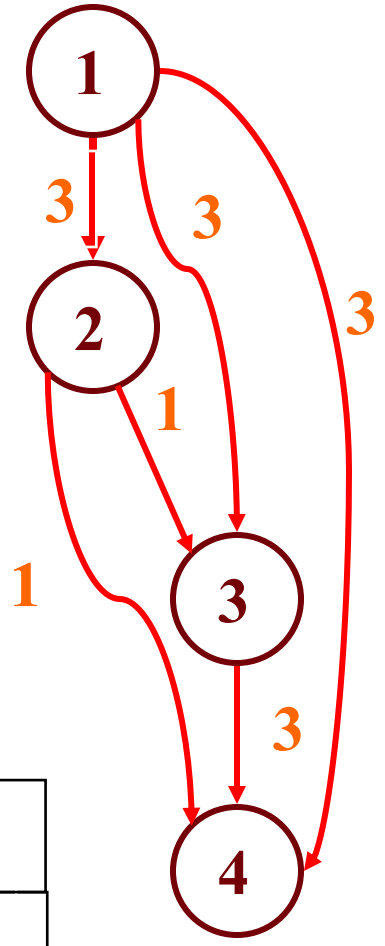
Example

```
1: ld  r2, 0(r1)
2: add r3,r3,r2
3: ld  r2,4(r5)
4: add r6,r6,r2
```



Example

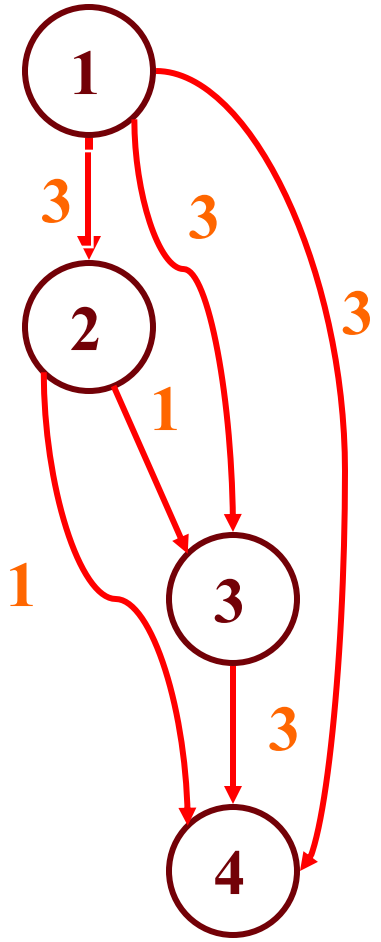
```
1: ld  r2, 0(r1)
2: add r3,r3,r2
3: ld  r2,4(r5)
4: add r6,r6,r2
```



ALUop			2			4
MEM 1	1			3		
MEM 2		1			3	

Example

```
1: ld  r2, 0(r1)
2: add r3,r3,r2
3: ld  r2,4(r5)
4: add r6,r6,r2
```



Anti-Dependence between 3 and 2.

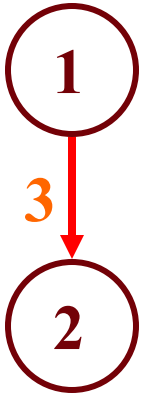
There is really no data flowing...

How to "fix" this?

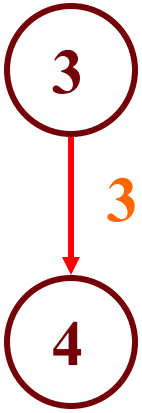
How about using a different Register?

Example

```
1: ld  r2, 0(r1)
2: add r3,r3,r2
3: ld  r4,4(r5)
4: add r6,r6,r4
```

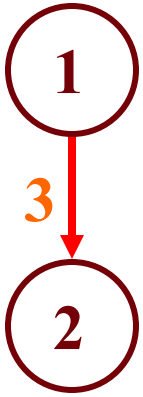


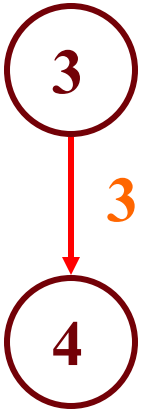
Eliminated Anti-Dependence *but* increased the number of registers *i.e.*, increased Register Pressure



Example

```
1: ld  r2, 0(r1)
2: add r3,r3,r2
3: ld  r4,4(r5)
4: add r6,r6,r4
```





ALUop			2	4
MEM 1	1	3		
MEM 2		1	3	

Register Allocation & Instruction Scheduling

- If Register Allocation is before Instruction Scheduling
  - restricts the choices for scheduling

Register Allocation & Instruction Scheduling

- If Register Allocation is before Instruction Scheduling
  - restricts the choices for scheduling
- If Instruction Scheduling before Register Allocation
  - Register allocation may spill registers
  - Will change the carefully done schedule!!!

Outline

- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across Basic Blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining

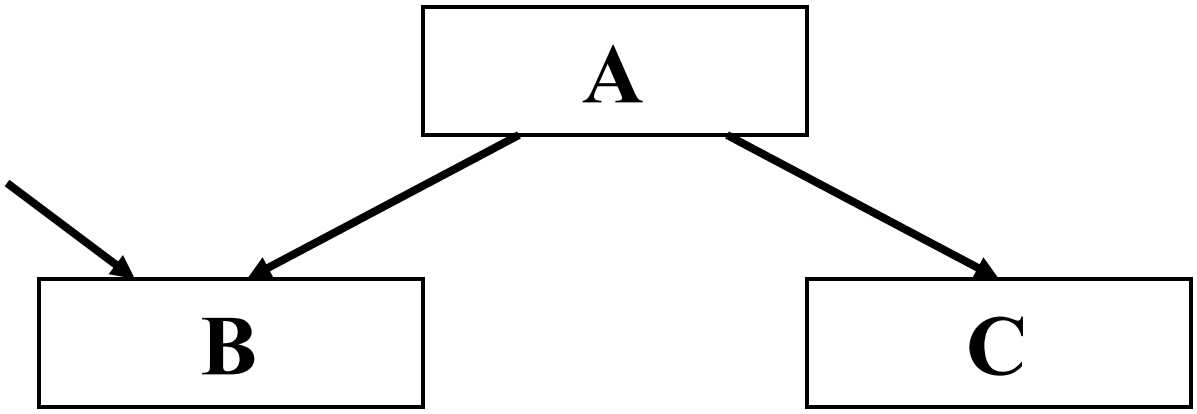
Scheduling across Basic Blocks

- Number of Instructions in a Basic Block is small
  - Cannot keep a multiple units with long pipelines busy by just scheduling within a basic block

- Need to handle Control Dependence
  - Scheduling constraints across basic blocks
  - Scheduling policy

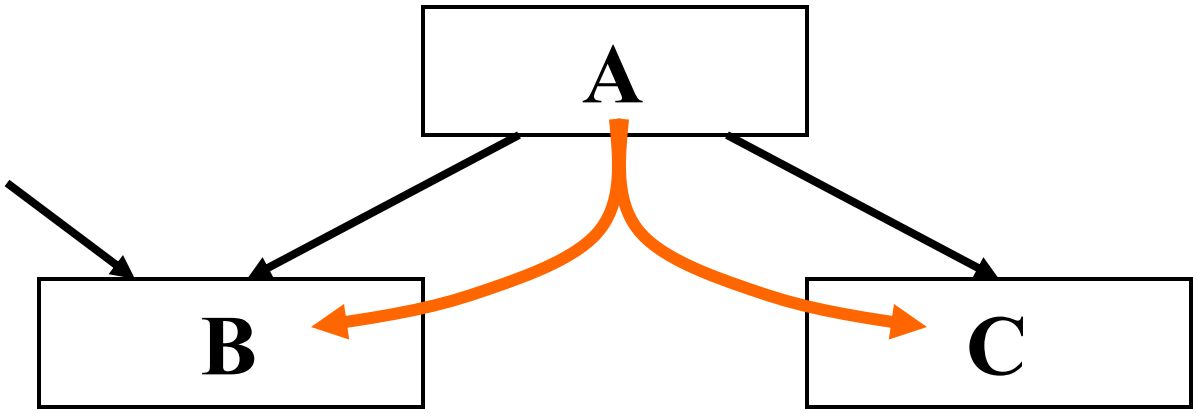
Moving across Basic Blocks

- Downward to adjacent Basic Block



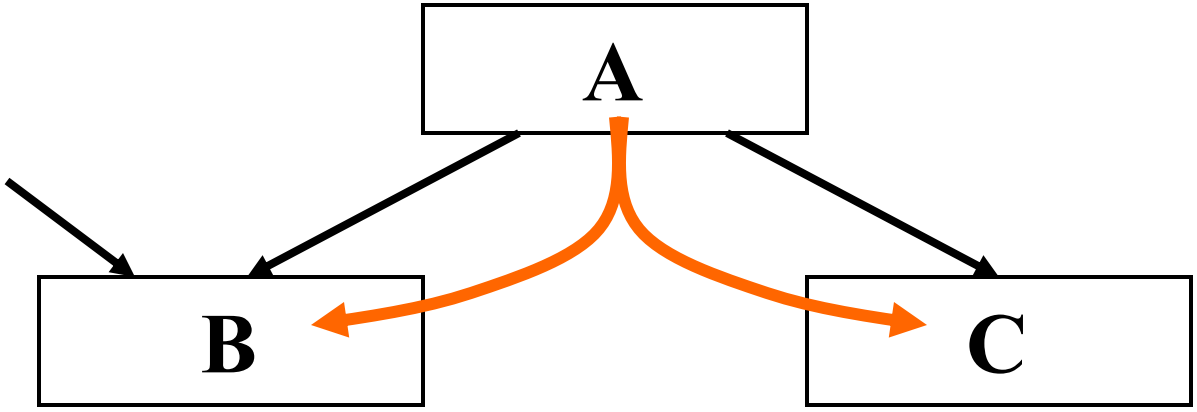
Moving across Basic Blocks

- Downward to adjacent Basic Block



Moving across Basic Blocks

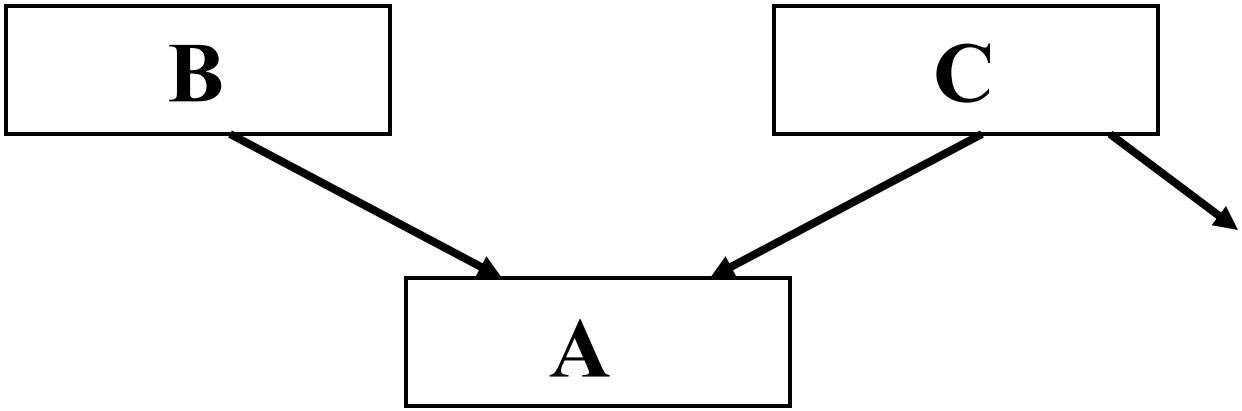
- Downward to adjacent Basic Block



- A path to B that does not execute A?

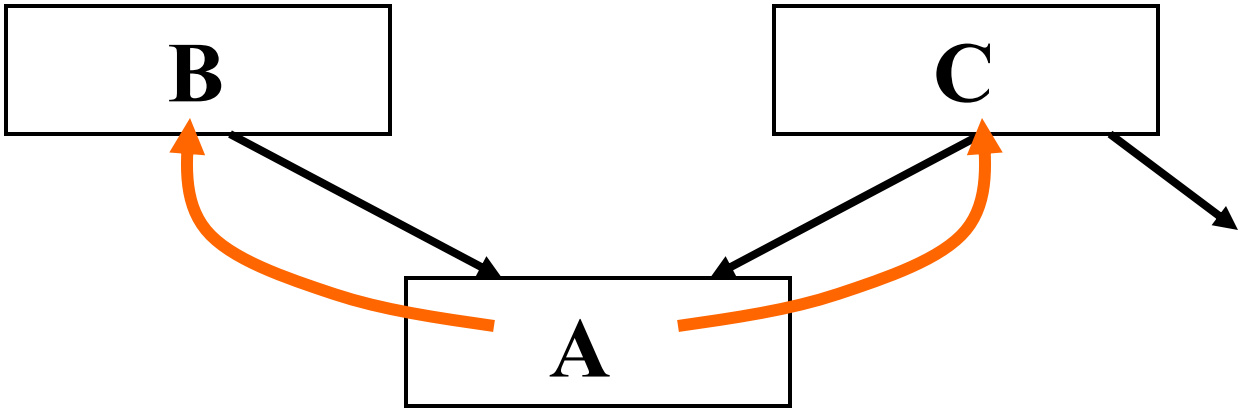
Moving across Basic Blocks

- Upward to adjacent Basic Block



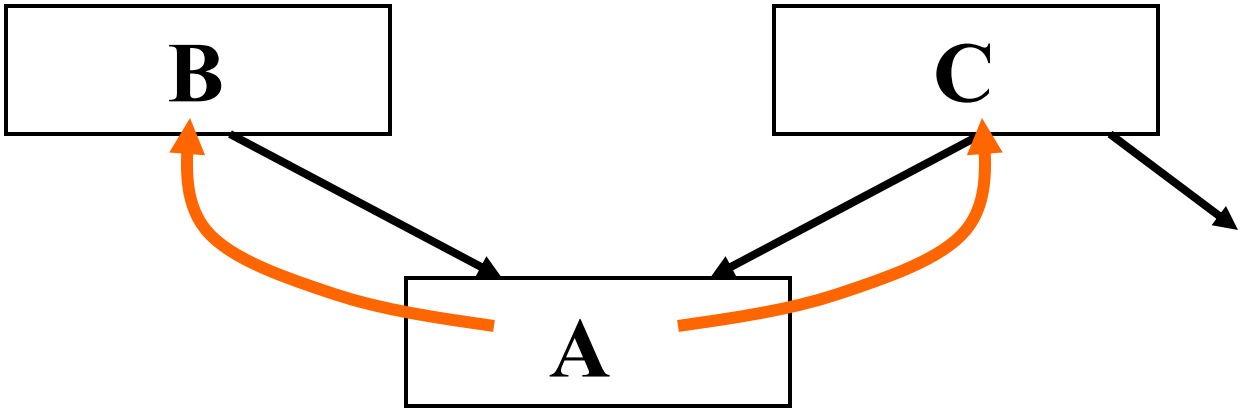
Moving across Basic Blocks

- Upward to adjacent Basic Block



Moving across Basic Blocks

- Upward to adjacent Basic Block



- A path from C that does not reach A?

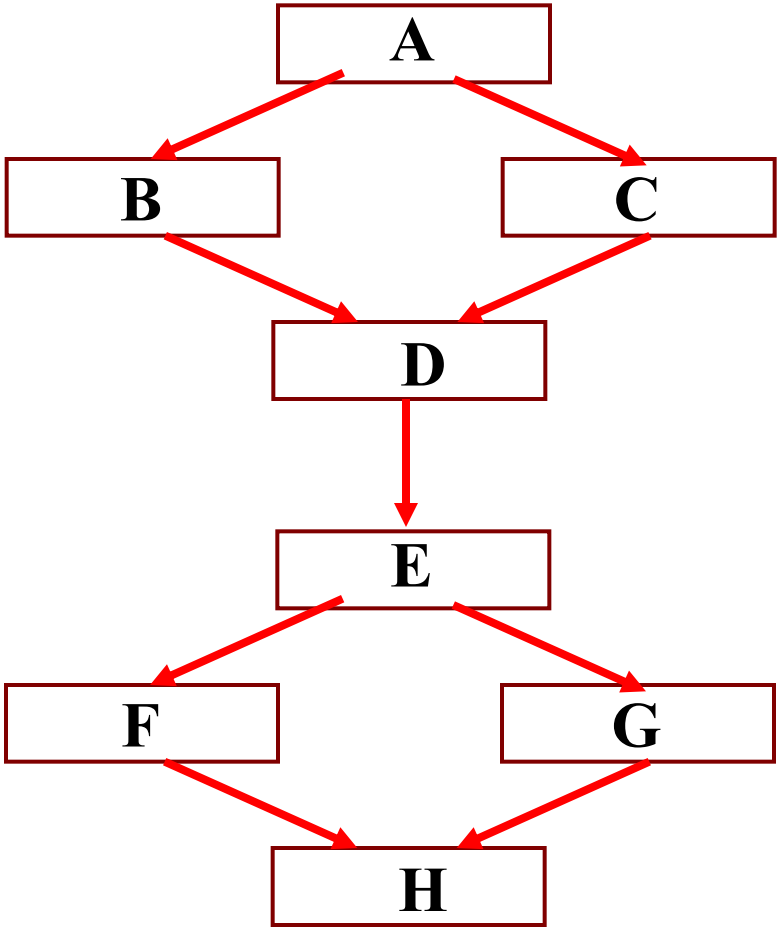
Outline

- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across Basic Blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining

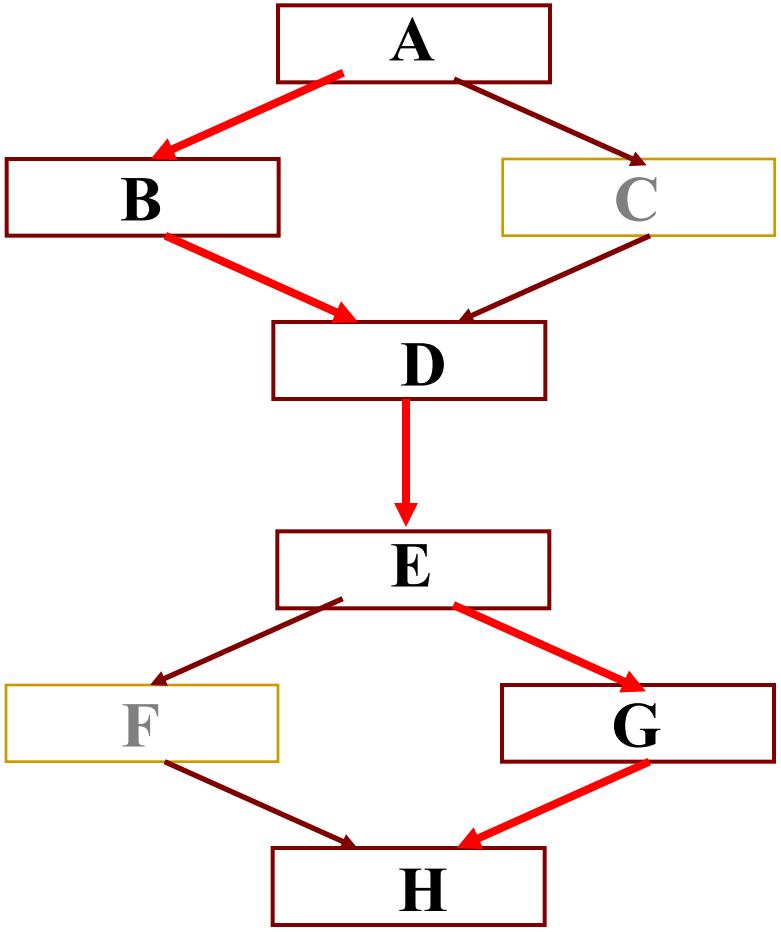
Trace Scheduling

- Find the most common *Trace* of Basic Blocks
  - Use profiling information
- Combine the Basic Blocks in the trace and schedule them as one Block
- Create *clean-up* code if the execution goes off-trace

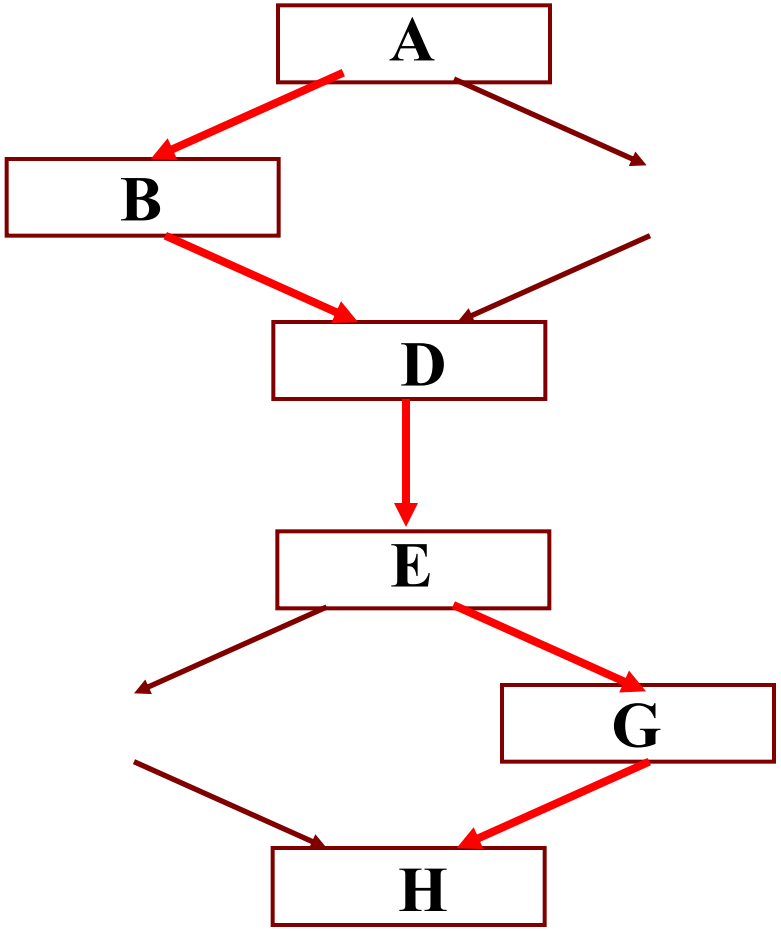
Trace Scheduling



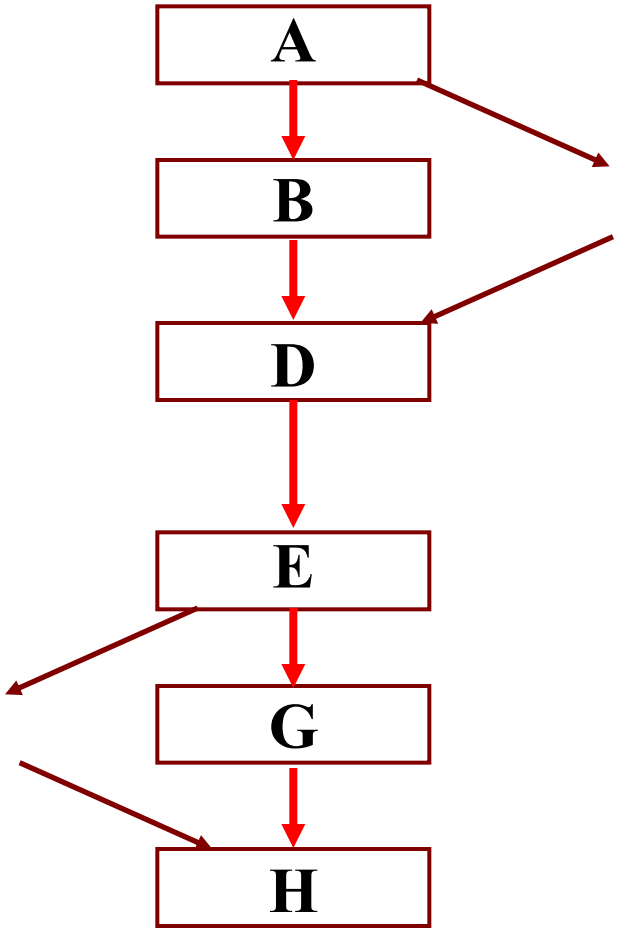
Trace Scheduling



Trace Scheduling

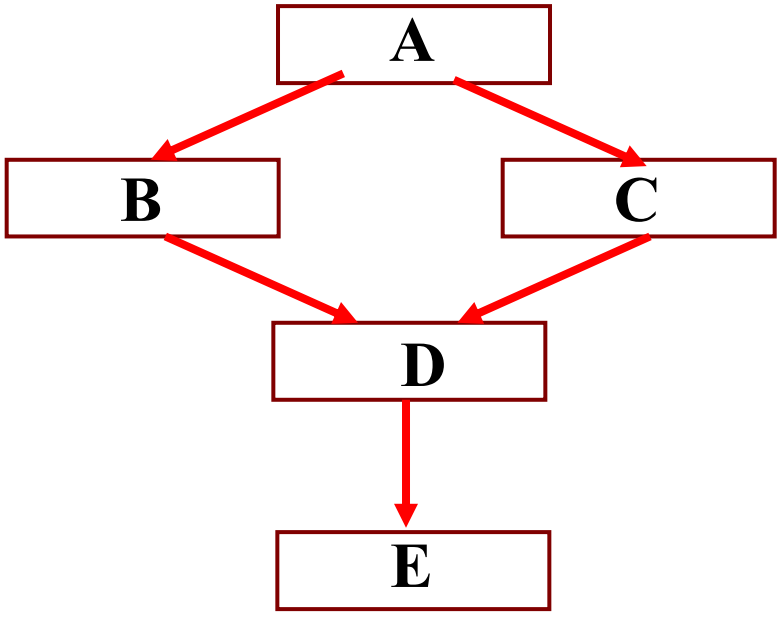


Trace Scheduling



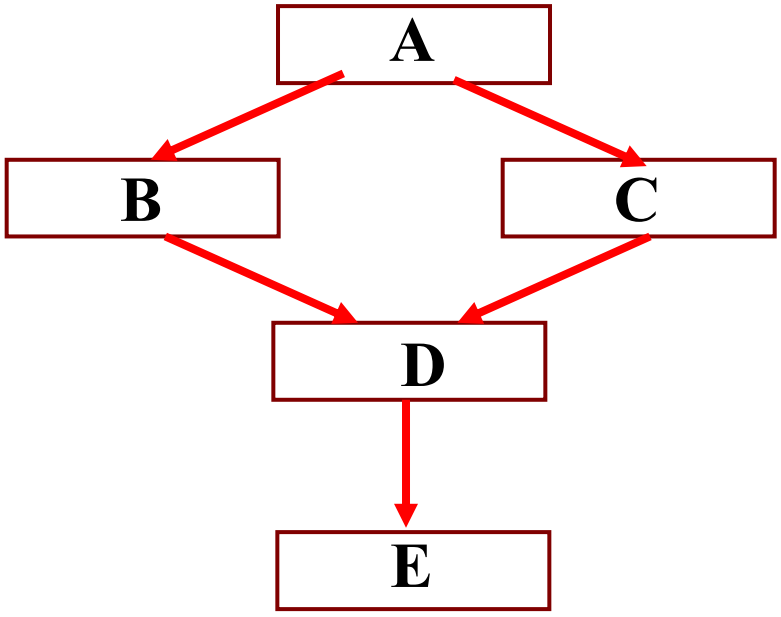
Large Basic Blocks via Code Duplication

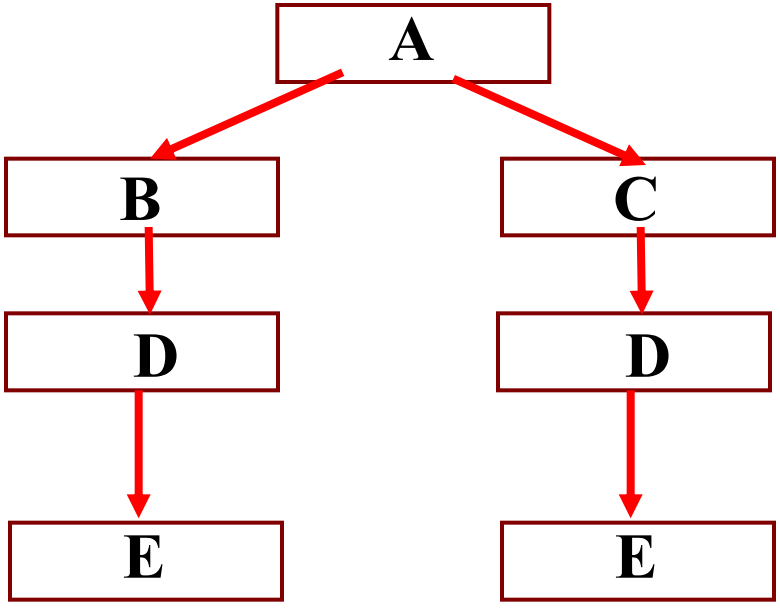
- Creating large extended Basic Blocks by duplication



Large Basic Blocks via Code Duplication

- Creating large extended Basic Blocks by duplication
- Schedule the larger Blocks





Scheduling Loops

- Loop bodies are small
- But, lot of time is spent in loops due to large number of iterations
- Need better ways to schedule loops

Loop Example

- Machine Model
  - One load/store unit
    - load 2 cycles
    - store 2 cycles
  - Two arithmetic units
    - add 2 cycles
    - branch 2 cycles (no delay slot)
    - multiply 3 cycles
  - Both units are pipelined (initiate one op each cycle)

- Source Code

```
for i = 1 to N
   A[i] = A[i] * b
```

Loop Example

- Source Code

```
for i = 1 to N
    A[i] = A[i] * b
```

- Assembly Code

```
loop:
   ld      r6, (r2)
   mul     r6, r6, r3
   st      r6, (r2)
   add     r2, r2, 4
   ble     r2, r5, loop
```

Loop Example

- Assembly Code

```
loop:
   ld      r6, (r2)
   mul     r6, r6, r3
   st      r6, (r2)
   add     r2, r2, 4
   ble     r2, r5, loop
```

- Schedule (7 cycles per iteration) – excluding branch

ld					st							
	ld					st						
		mul					ble					
			mul					ble				
				mul								
					add							
						add						

Outline

- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across Basic Blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining

Loop Unrolling

- Unroll the Loop Body a few times
- Pros:
  - Create a much larger basic block for the body
  - Eliminate few loop bounds checks
- Cons:
  - Much larger program
  - Setup code (# of iterations < unroll factor)
  - beginning and end of the schedule can still have unused slots

Loop Example

```
loop:
      ld      r6, (r2)
      mul     r6, r6, r3
      st      r6, (r2)
      add     r2, r2, 4
      ble     r2, r5, loop
```

Loop Example

```
loop:
      ld      r6, (r2)
      mul     r6, r6, r3
      st      r6, (r2)
      add     r2, r2, 4
      ld      r6, (r2)
      mul     r6, r6, r3
      st      r6, (r2)
      add     r2, r2, 4
      ble     r2, r5, loop
```

Loop Example

```
loop:
      ld      r6, (r2)
      mul     r6, r6, r3
      st      r6, (r2)
      add     r2, r2, 4
      ld      r6, (r2)
      mul     r6, r6, r3
      st      r6, (r2)
      add     r2, r2, 4
      ble     r2, r5, loop
```

- Schedule (7 cycles per iteration)

ld					st		ld					st			
	ld					st		ld					st		
		mul							mul					ble	
			mul							mul					ble
				mul							mul				
					add							add			
						add							add		

Loop Unrolling

- Rename Registers
  - Use Different Registers in Different Loop Iterations

Loop Example

```
loop:
      ld      r6,(r2)
      mul     r6, r6, r3
      st      r6,(r2)
      add     r2, r2, 4
      ld      r6,(r2)
      mul     r6, r6, r3
      st      r6,(r2)
      add     r2, r2, 4
      ble     r2, r5, loop
```

Loop Example

```
loop:
      ld      r6,(r2)
      mul     r6, r6, r3
      st      r6,(r2)
      add     r2, r2, 4
      ld      r7,(r2)
      mul     r7, r7, r3
      st      r7,(r2)
      add     r2, r2, 4
      ble     r2, r5, loop
```

Loop Unrolling

- Rename Registers
  - Use Different Registers in Different Loop Iterations
- Eliminate Unnecessary Dependences
  - Use more registers to eliminate true, anti and output dependences
  - Eliminate dependent-chains of calculations whenever possible

Loop Example

```
loop:
      ld      r6,(r2)
      mul     r6, r6, r3
      st      r6,(r2)
      add     r2, r2, 4
      ld      r7,(r2)
      mul     r7, r7, r3
      st      r7,(r2)
      add     r2, r2, 4
      ble     r2, r5, loop
```

Loop Example

```
loop:
      ld      r6,(r1)
      mul     r6, r6, r3
      st      r6,(r1)
      add     r2, r1, 4
      ld      r7,(r2)
      mul     r7, r7, r3
      st      r7,(r2)
      add     r1, r2, 4
      ble     r1, r5, loop
```

Loop Example

```
loop:
      ld      r6,(r1)
      mul     r6, r6, r3
      st      r6,(r1)
      add     r2, r1, 4
      ld      r7,(r2)
      mul     r7, r7, r3
      st      r7,(r2)
      add     r1, r2, 4
      ble     r1, r5, loop
```

Loop Example

```
loop:
      ld      r6,(r1)
      mul     r6, r6, r3
      st      r6,(r1)
      add     r2, r1, 4
      ld      r7,(r2)
      mul     r7, r7, r3
      st      r7,(r2)
      add     r1, r1, 8
      ble     r1, r5, loop
```

Loop Example

```
loop:
      ld      r6, (r1)
      mul     r6, r6, r3
      st      r6, (r1)
      add     r2, r1, 4
      ld      r7, (r2)
      mul     r7, r7, r3
      st      r7, (r2)
      add     r1, r1, 8
      ble     r1, r5, loop
```

- Schedule (3.5 cycles per iteration)

ld		ld			st		st			
	ld		ld			st		st		
		mul		mul			ble			
			mul		mul			ble		
				mul		mul				
add					add					
	add					add				

Outline

- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across Basic Blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining

Software Pipelining

- Try to overlap Multiple Iterations so that pipeline Slots will be filled
- Find the Steady-State Window so that:
  - All the instructions of the loop body are executed
  - But from different iterations

Loop Example

- Assembly Code

```
loop:
   ld       r6, (r2)
   mul      r6, r6, r3
   st       r6, (r2)
   add      r2, r2, 4
   ble      r2, r5, loop
```

- Schedule

ld		ld1		ld2	st	ld3	st1	ld4	st2	ld5	st3	ld6
	ld		ld1		ld2	st	ld3	st1	ld4	st2	ld5	st3
		mul		mul1		mul2	ble	mul3	ble1	mul4	ble2	mul5
			mul		mul1		mul2	ble	mul3	ble1	mul4	ble2
				mul		mul1		mul2		mul3		mul4
					add		add1		add2		add3	
						add		add1		add2		add3

Loop Example

- Assembly Code

```
loop:
   ld       r6, (r2)
   mul      r6, r6, r3
   st       r6, (r2)
   add      r2, r2, 4
   ble      r2, r5, loop
```

- Schedule

ld		ld1		ld2	st	ld3	st1	ld4	st2	ld5	st3	ld6
	ld		ld1		ld2	st	ld3	st1	ld4	st2	ld5	st3
		mul		mul1		mul2	ble	mul3	ble1	mul4	ble2	mul5
			mul		mul1		mul2	ble	mul3	ble1	mul4	ble2
				mul		mul1		mul2		mul3		mul4
					add		add1		add2		add3	
						add		add1		add2		add3

Loop Example

- Assembly Code

```
loop:
   ld      r6, (r2)
   mul     r6, r6, r3
   st      r6, (r2)
   add     r2, r2, 4
   ble     r2, r5, loop
```

- Schedule (2 cycles per iteration)

ld3	st1
st	ld3
mul2	ble
	mul2
mul1	
	add1
add	

Loop Example

- 4 Overlapped Iterations
  - value of r3 and r5 don't change
  - 4 regs for &A[i] (r2)
  - each address incremented by 4*4
  - 4 regs to keep value A[i] (r6)
  - Same registers can be reused after 4 of these blocks; generate code for 4 blocks, otherwise need to move


```
loop:
    ld      r6, (r2)
    mul     r6, r6, r3
    st      r6, (r2)
    add     r2, r2, 4
    ble     r2, r5, loop
```

Software Pipelining

- Optimal use of Resources
- Need a lot of Registers
  - Values in multiple iterations need to be kept separated
- Issues with Dependences:
  - Executing a store instruction in an iteration before branch instruction is executed for a previous iteration (writing when it should not have)
  - Loads and stores are issued out-of-order (need to figure-out dependencies before doing this)
- Code Generation Issues:
  - Generate pre-amble and post-amble code
  - Multiple blocks so no register copy is needed

Summary

- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across Basic Blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining