

COMP 250

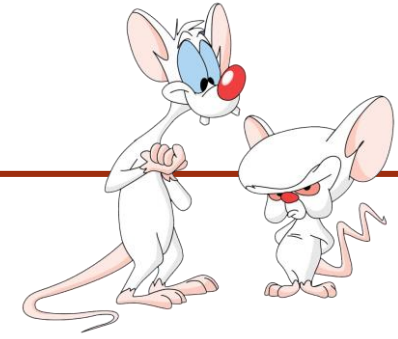
INTRODUCTION TO COMPUTER SCIENCE

35 – Graphs Traversals

Giulia Alberini, Fall 2022

Slides adapted from Michael Langer's

WHAT ARE WE GOING TO DO TODAY?

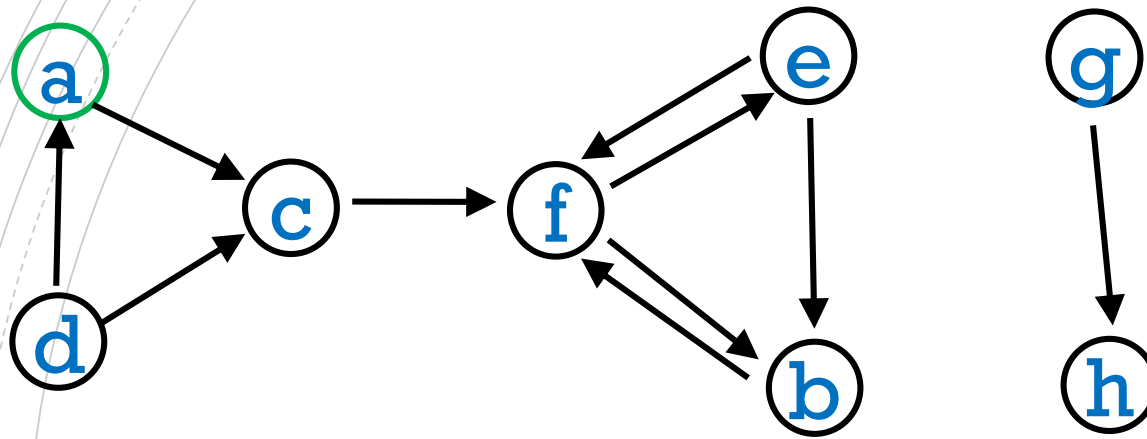


- Non-recursive graph traversal
 - depth first
 - breadth first
- Shortest path
 - Dijkstra's Algorithm

GRAPH TRAVERSAL (RECURSIVE)

```
depthFirst_Graph (v) {  
    v.visited = true  
    visit v // do something with v  
    for each w such that (v,w) is in E  
    // i.e. for each w in v.adjList  
        if !(w.visited) // avoid cycles!  
            depthFirst_Graph(w)  
}
```

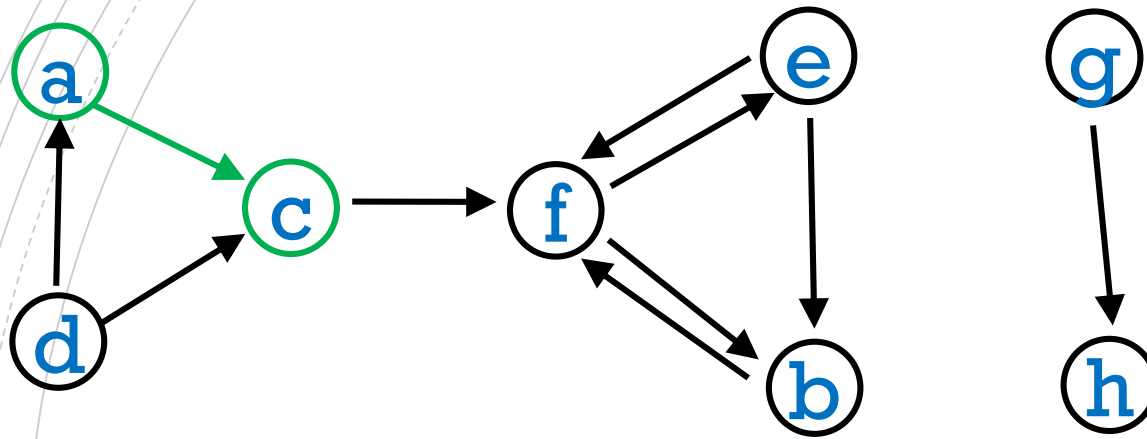
CALL STACK FOR depthFirst(a)



a

```
depthFirst_Graph (v) {  
    v.visited = true  
    for each w s.t. (v,w) is in E  
        if !(w.visited)  
            depthFirst_Graph(w)  
}
```

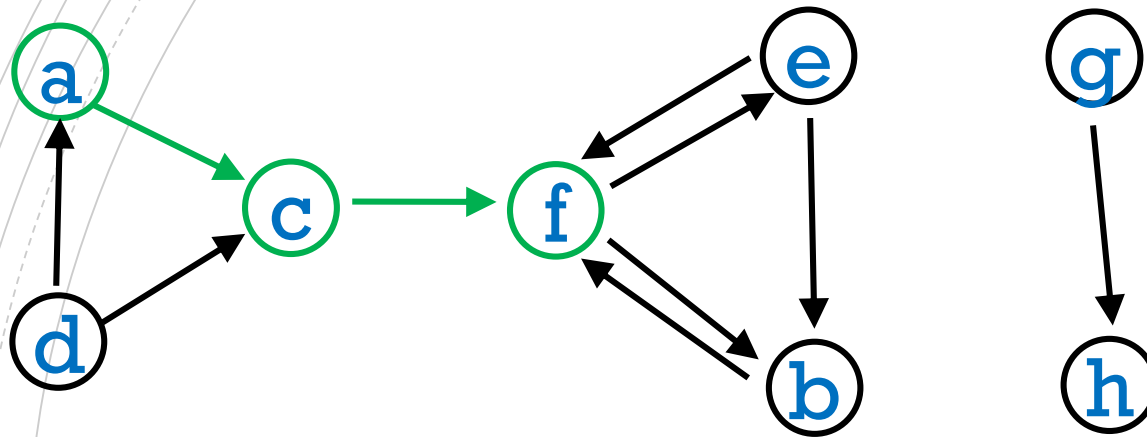
CALL STACK FOR depthFirst(a)



a c
a

```
depthFirst_Graph (v) {  
    v.visited = true  
    for each w s.t. (v,w) is in E  
        if !(w.visited)  
            depthFirst_Graph(w)  
}
```

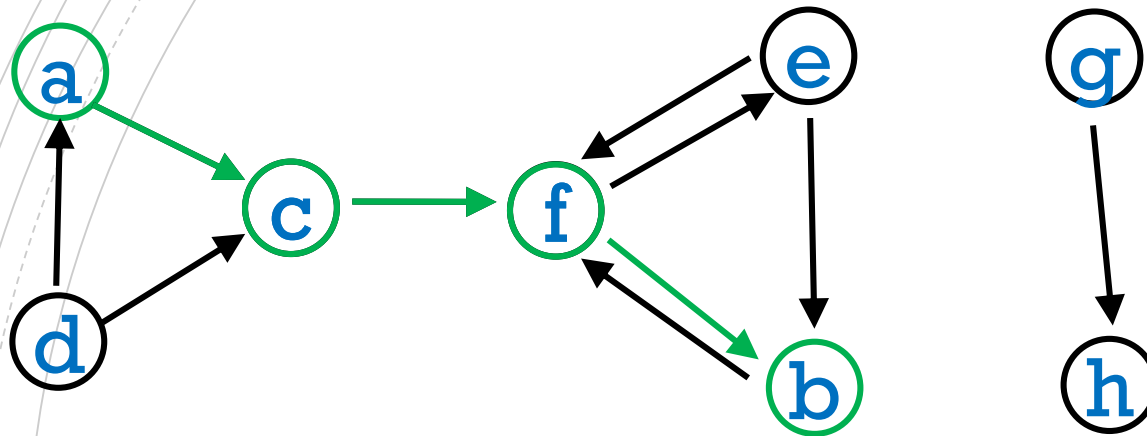
CALL STACK FOR depthFirst(a)



a c f
a c a

```
depthFirst_Graph (v) {  
    v.visited = true  
    for each w s.t. (v,w) is in E  
        if !(w.visited)  
            depthFirst_Graph(w)  
}
```

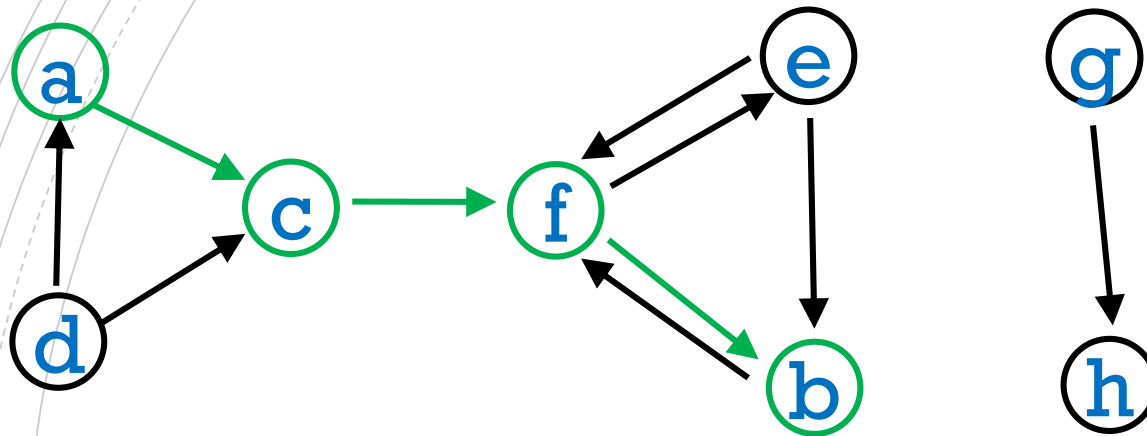
CALL STACK FOR depthFirst(a)



			b
		f	f
	c	c	c
a	a	a	a

```
depthFirst_Graph (v) {  
    v.visited = true  
    for each w s.t. (v,w) is in E  
        if !(w.visited)  
            depthFirst_Graph(w)  
}
```

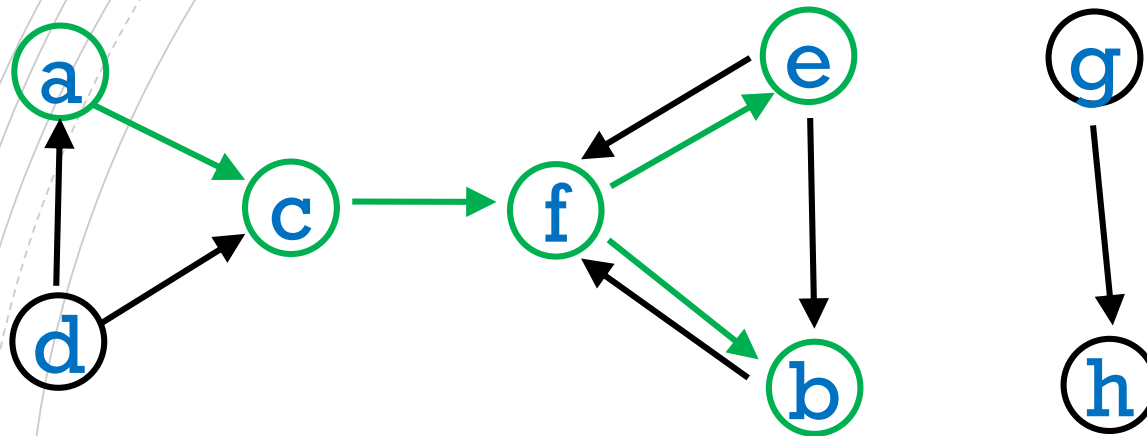
CALL STACK FOR depthFirst(a)



			b	
		f	f	f
	c	c	c	c
a	a	a	a	a

```
depthFirst_Graph (v) {  
    v.visited = true  
    for each w s.t. (v,w) is in E  
        if !(w.visited)  
            depthFirst_Graph(w)  
}
```

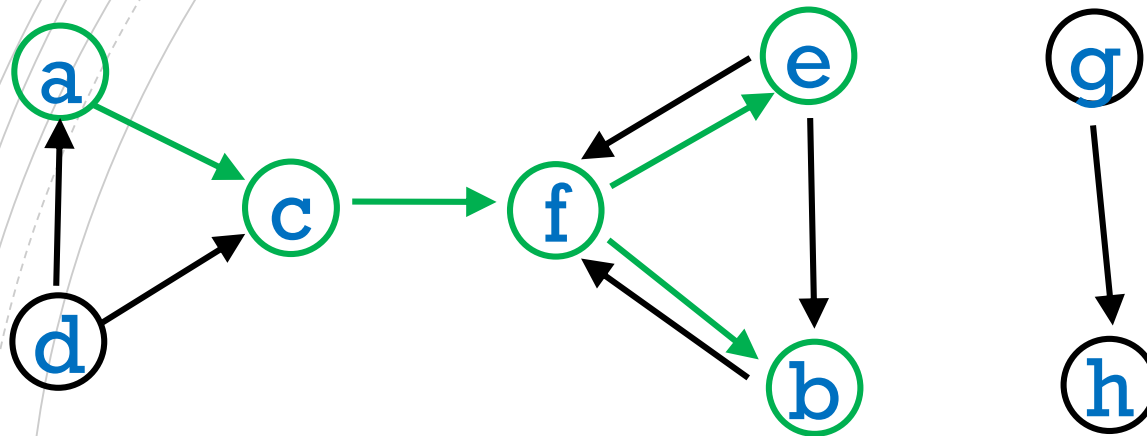

CALL STACK FOR depthFirst(a)



			b		e
		f	f	f	f
	c	c	c	c	c
a	a	a	a	a	a

```
depthFirst_Graph (v) {  
    v.visited = true  
    for each w s.t. (v,w) is in E  
        if !(w.visited)  
            depthFirst_Graph(w)  
}
```

CALL STACK FOR depthFirst(a)

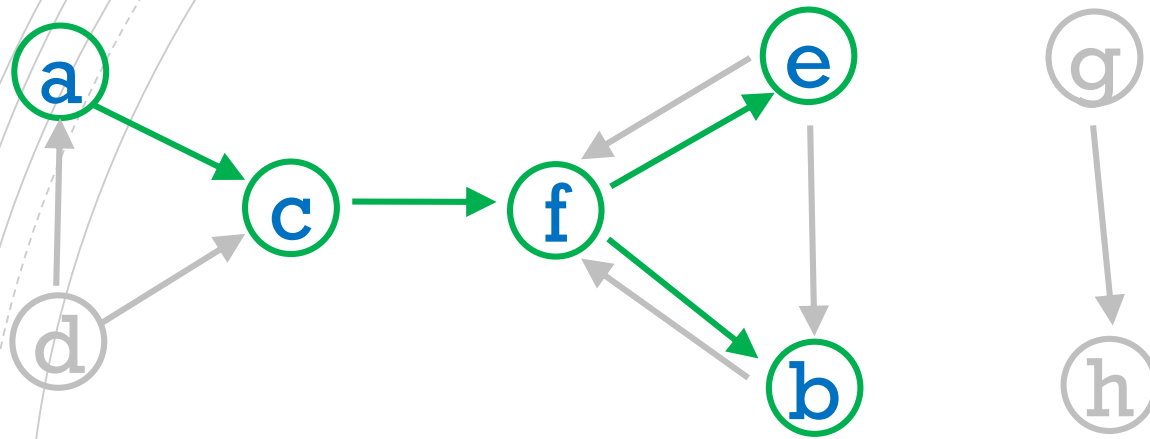


			b		e			
		f	f	f	f	f		
	c	c	c	c	c	c	c	
a	a	a	a	a	a	a	a	a

```
depthFirst_Graph (v) {  
    v.visited = true  
    for each w s.t. (v,w) is in E  
        if !(w.visited)  
            depthFirst_Graph(w)  
}
```

CALL TREE

root



The diagram illustrates the evolution of a word through a series of stages, represented by columns of letters. The letters are arranged in a grid-like structure, with some letters appearing in multiple columns. The letters are: 'a', 'c', 'f', 'b', and 'e'. The letters 'a', 'c', and 'f' are in blue, while 'b' and 'e' are in black. The letters are arranged in a way that suggests a progression from left to right, with 'a' at the bottom left, 'c' above it, 'f' above that, and 'b' and 'e' at the top. The letters 'a', 'c', and 'f' are repeated in each column, while 'b' and 'e' are only present in the final columns. The letters are arranged in a way that suggests a progression from left to right, with 'a' at the bottom left, 'c' above it, 'f' above that, and 'b' and 'e' at the top. The letters 'a', 'c', and 'f' are repeated in each column, while 'b' and 'e' are only present in the final columns.

GRAPH TRAVERSALS

- Unlike tree traversal for rooted tree, a graph traversal started from some arbitrary vertex does not necessarily reach all other vertices.
- *Knowing which vertices can be reached by a path from some starting vertex is itself an important problem. You will learn about such graph 'connectivity' problems in COMP 251.*
- The order of nodes visited depends on the order of nodes in the adjacency lists.

GRAPH TRAVERSALS

- **Q: Can we do non-recursive graph traversals?**

GRAPH TRAVERSALS

- **Q: Can we do non-recursive graph traversals?**
- **A: Yes, similar to tree traversal: use a stack or a queue.**

RECALL: DEPTH FIRST TREE TRAVERSAL (WITH A SLIGHT VARIATION)

```
treeTraversalUsingStack(root) {  
  initialize empty stack s  
  s.push(root)  
  while s is not empty {  
    cur = s.pop()  
    visit cur  
    for each child of cur {  
      s.push(child)  
    }  
  }  
}
```

Visit a node *after*
popping it from the
stack.

Every node in the tree
gets pushed, and
popped, and visited.

GENERALIZE TO GRAPH

```
graphTraversalUsingStack(v) {  
    initialize empty stack s  
    v.visited = true  
    s.push(v)  
    while s is not empty {  
        cur = s.pop()  
        visit cur // do something  
        for each w in cur.adjList  
            if(!w.visited) {  
                w.visited = true  
                s.push(w)  
            }  
        }  
    }  
}
```

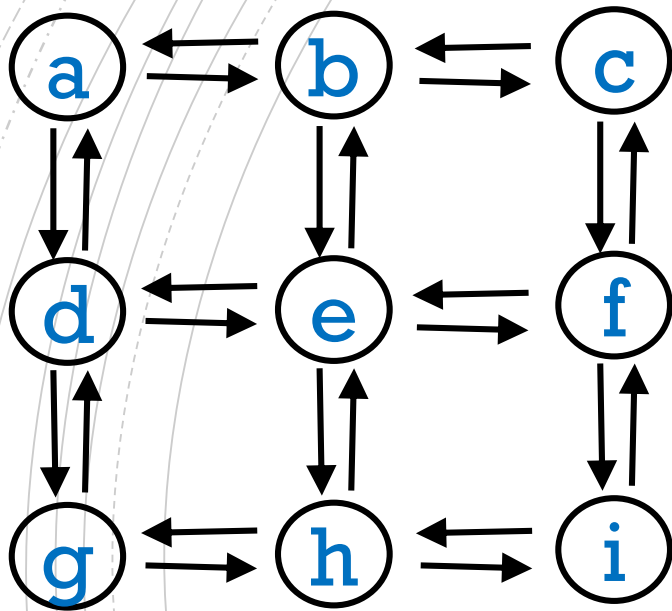
Indicate as “reached” a node *before pushing* it onto the stack. We do that by updating the field `visited`.

Visit the node (perform some operations) after it gets popped from the stack.

Every node in the graph gets *reached*, pushed, popped, and visited.

EXAMPLE: graphTraversalUsingStack(a)

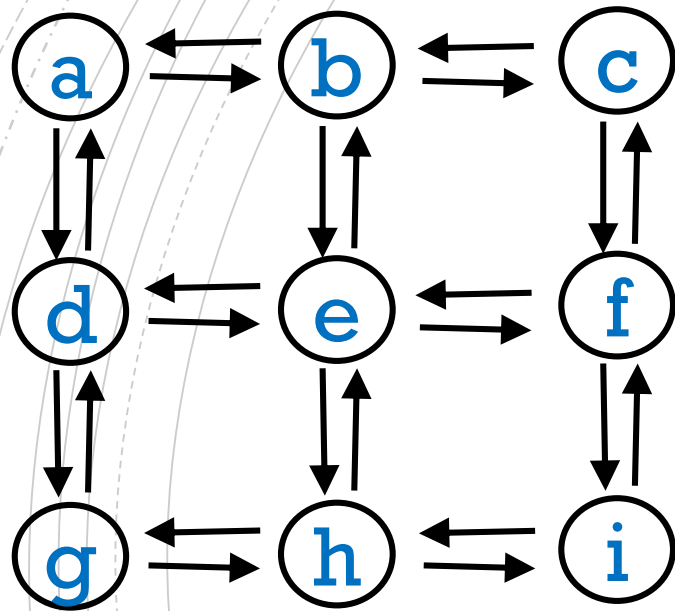
Order of visit: a



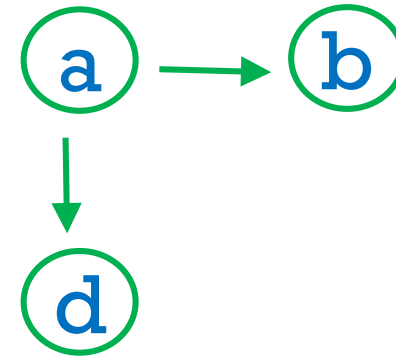
a

a

EXAMPLE: graphTraversalUsingStack(a)

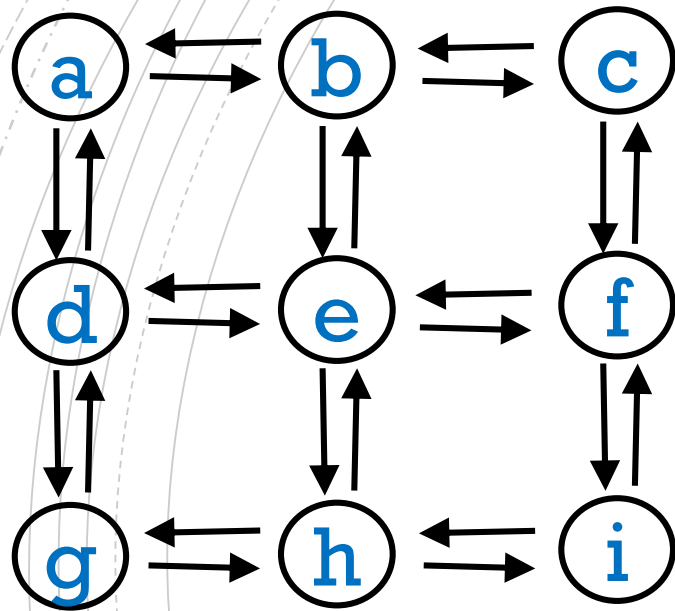


Order of visit: **a**

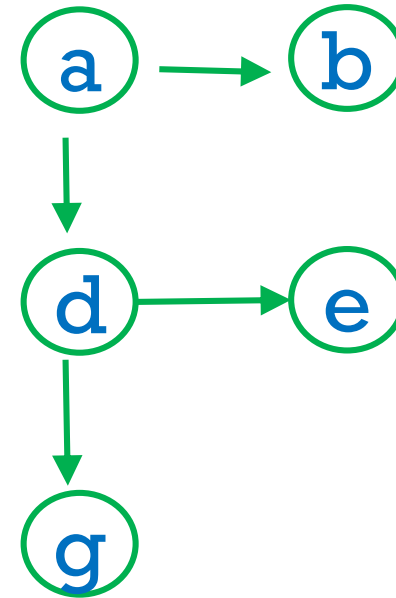


a _ b d

EXAMPLE: graphTraversalUsingStack(a)

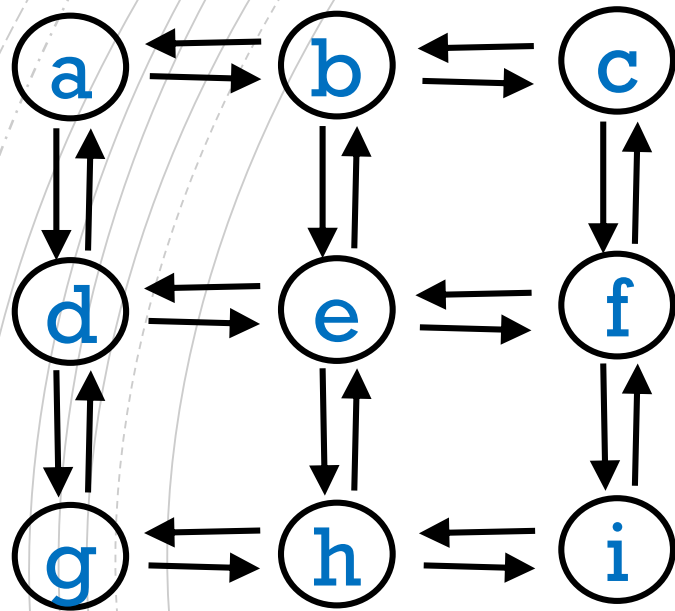


Order of visit: **ad**

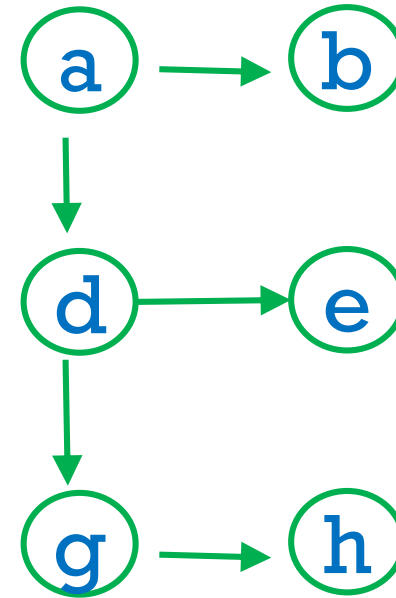


a _ b d g
b b e
b b

EXAMPLE: graphTraversalUsingStack(a)

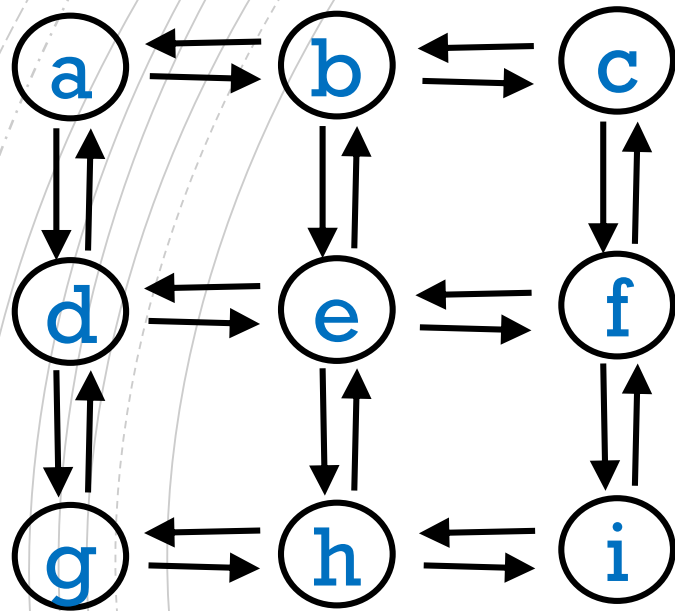


Order of visit: **adg**

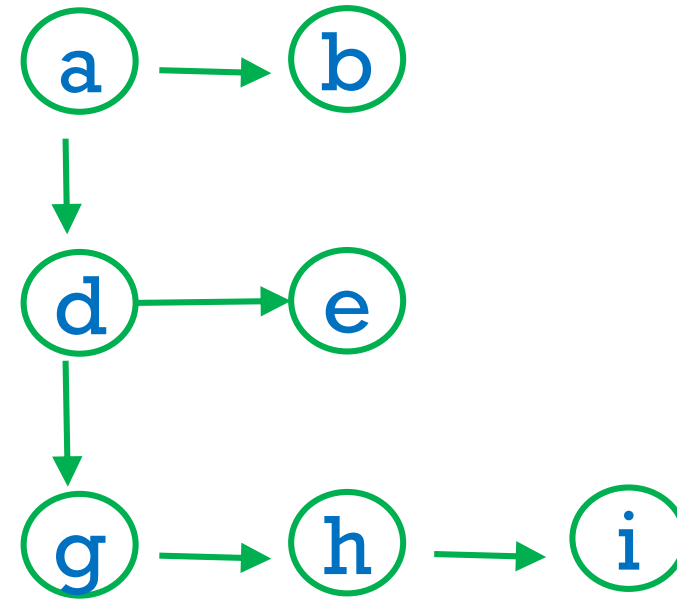


				g		h
		d		e	e	e
a	_	b	b	b	b	b

EXAMPLE: graphTraversalUsingStack(a)

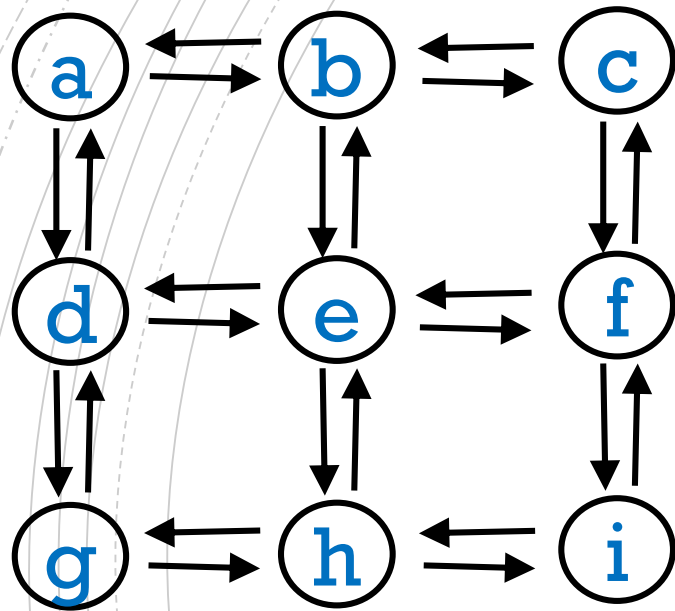


Order of visit: **adgh**

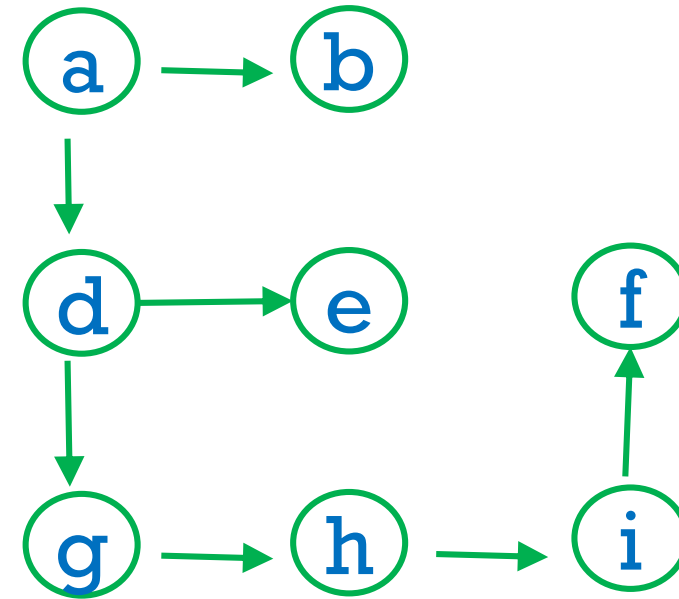


				g		h		i
		d		e	e	e	e	e
a	_	b	b	b	b	b	b	b

EXAMPLE: graphTraversalUsingStack(a)



Order of visit: **adghi**

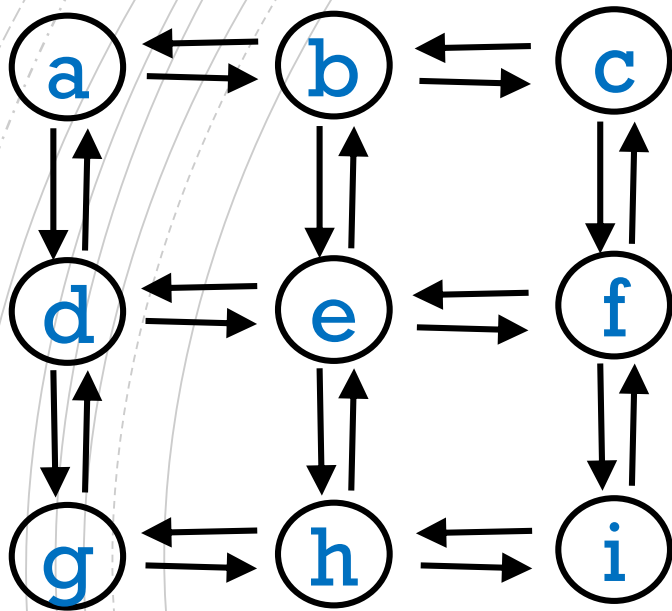


g h i f

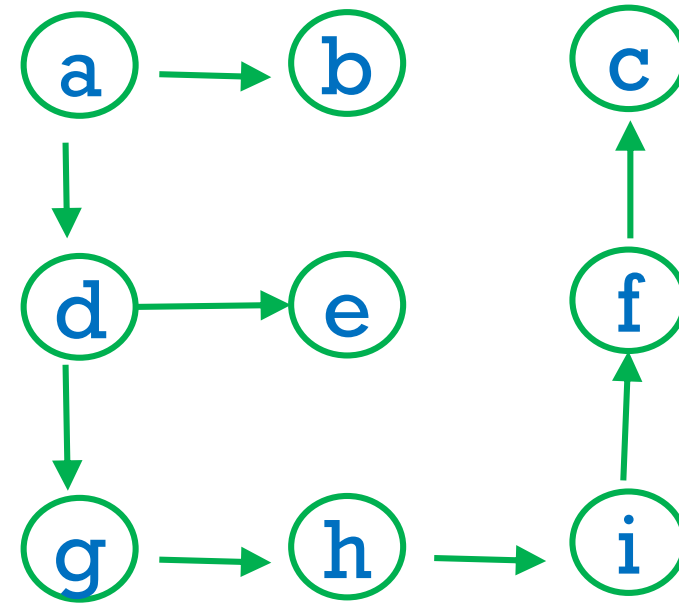
d e e e e e e e

a b b b b b b b b b

EXAMPLE: graphTraversalUsingStack(a)



Order of visit: **adghif**

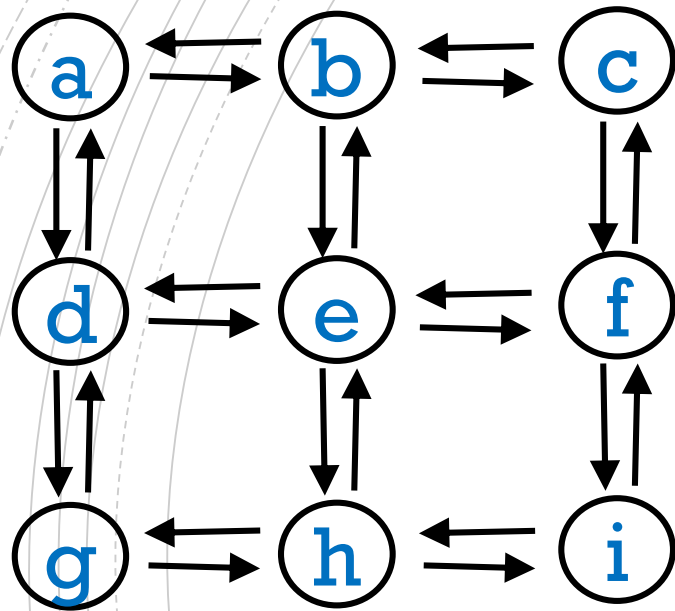


g h i f c

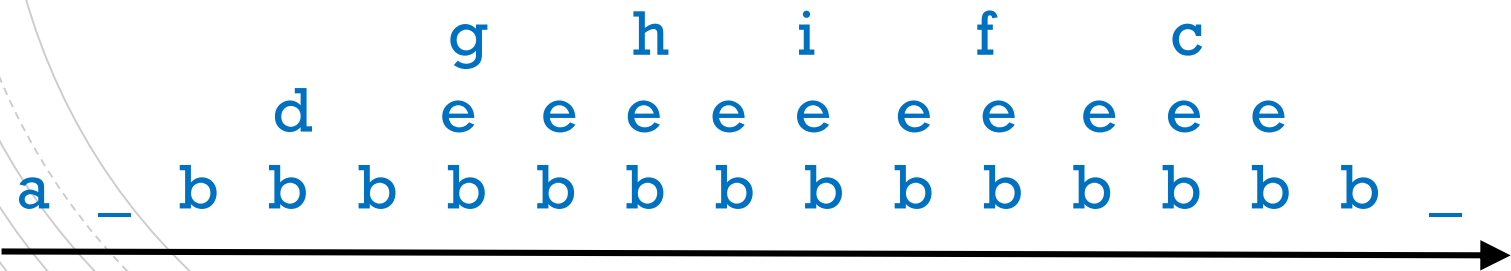
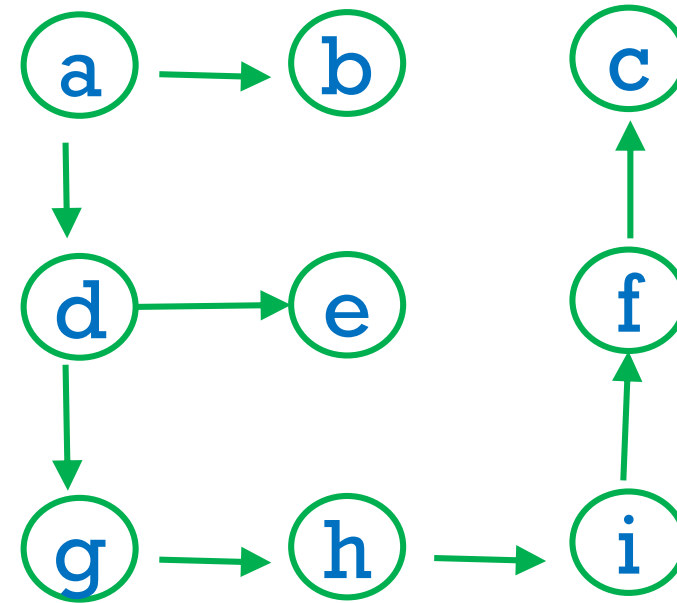
d e e e e e e e e e

a b b b b b b b b b b b

EXAMPLE: graphTraversalUsingStack(a)

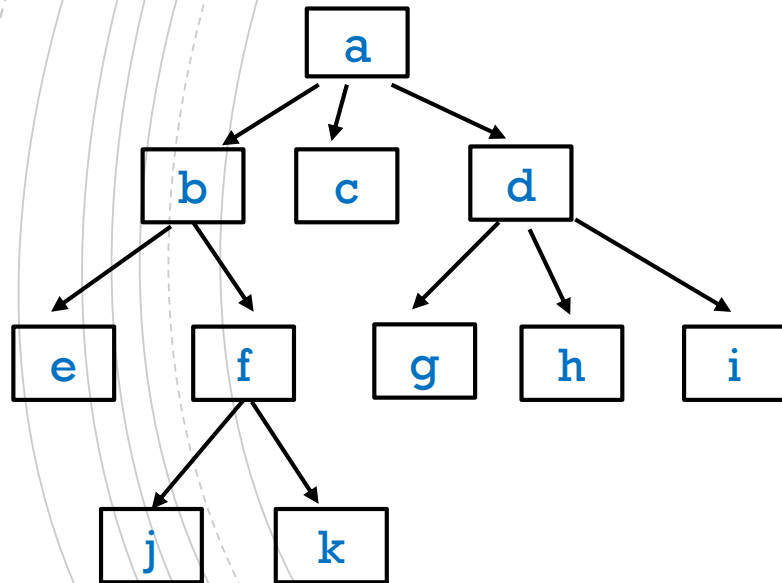


Order of visit: **adghifceb**



RECALL: BREADTH FIRST TREE TRAVERSAL

for each level i
visit all nodes at level i



```
treeTraversalUsingQueue(root) {  
    initialize empty queue q  
    q.enqueue(root)  
    while q is not empty {  
        cur = q.dequeue()  
        visit cur  
        for each child of cur  
            q.enqueue(child)  
    }  
}
```

BREADTH FIRST GRAPH TRAVERSAL

Given an input vertex, visit all vertices that can be reached by paths of length 1, 2, 3, 4,

BREADTH FIRST GRAPH TRAVERSAL

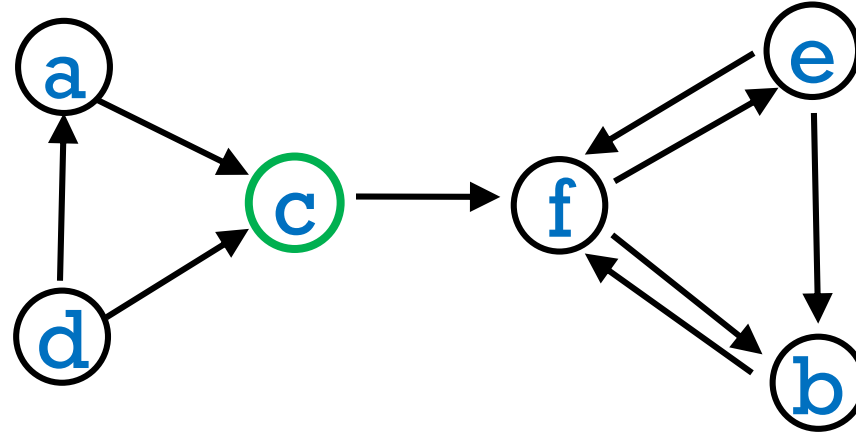
```
graphTraversalUsingQueue(v) {  
    initialize empty queue q  
    v.visited = true  
    q.enqueue(v)  
    while q is not empty {  
        cur = q.dequeue()  
        for each w in cur.adjList {  
            if(!w.visited) {  
                w.visited = true  
                q.enqueue(w)  
            }  
        }  
    }  
}
```

EXAMPLE

graphTraversalUsingQueue(c)

queue

c



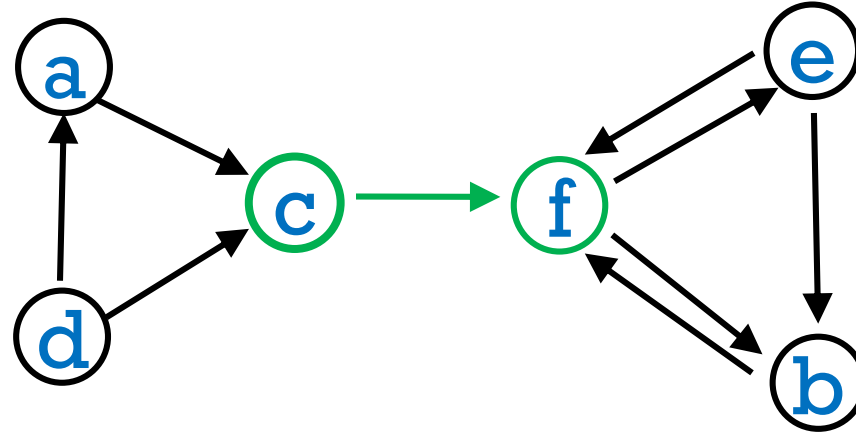
EXAMPLE

graphTraversalUsingQueue(c)

queue

c

f



EXAMPLE

graphTraversalUsingQueue(c)

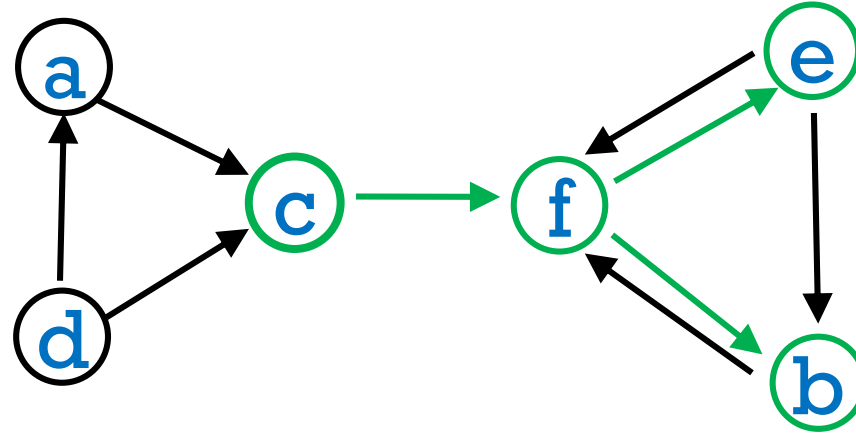
queue

c

f

b

Both 'b', 'e' are visited
and enqueued before
'b' is dequeued.



EXAMPLE

graphTraversalUsingQueue(c)

queue

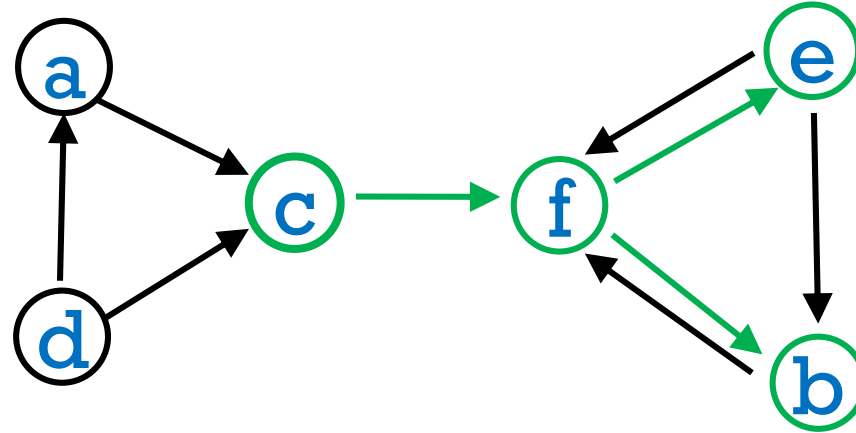
c

f

b e

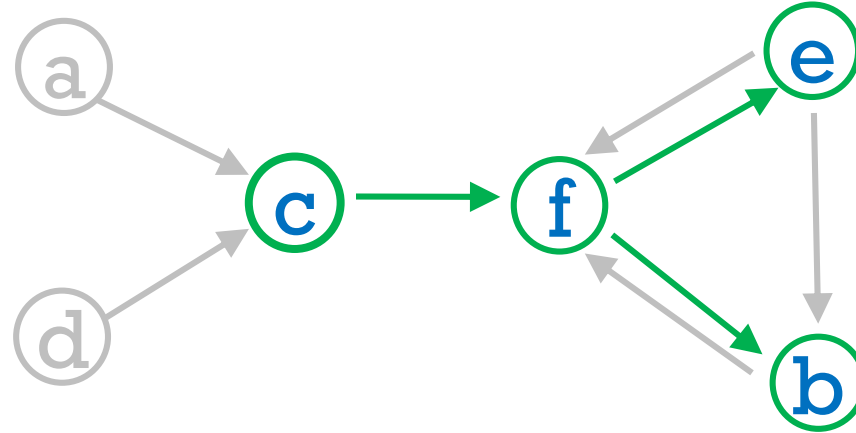
e

—



EXAMPLE

`graphTraversalUsingQueue(c)`



It defines a tree whose root is the starting vertex.
It finds the shortest path (number of edges) to all vertices reachable from the starting vertex.

RECALL: HOW TO IMPLEMENT A GRAPH CLASS IN JAVA?

```
class Graph<T> {  
    ArrayList<Vertex<T>> vetexList;  
  
    class Vertex<T> {  
        ArrayList<Edge> adjList;  
        T element;  
        boolean visited;  
    }  
  
    class Edge {  
        Vertex endVertex;  
        double weight;  
        :  
    }  
}
```

PRIOR TO TRAVERSAL!

```
for each  $w$  in  $V$   
   $w.visited = false$ 
```

How should we implement this?

PRIOR TO TRAVERSAL!

```
for each w in V  
    w.visited = false
```

```
class Graph<T> {  
    ArrayList<Vertex<T>> vertexList;  
    :  
    public void resetVisited() {  
  
    }  
}
```

PRIOR TO TRAVERSAL!

```
for each w in V  
    w.visited = false
```

```
class Graph<T> {  
    ArrayList<Vertex<T>> vertexList;  
    :  
    public void resetVisited() {  
        for(Vertex<T> v : vertexList)  
            v.visited = false;  
    }  
}
```

The background features a series of concentric circles in a light gray color, centered around the middle of the frame. A solid dark red rectangle is positioned in the center, containing the text 'SHORTEST PATH'. Below this rectangle is a horizontal white line, followed by another solid dark red rectangle of the same width.

SHORTEST PATH

MODELING AS GRAPHS

Input:

- Directed graph $G = (V, E)$
- Weight function $w: E \rightarrow \mathbb{R}$

Weight of path $p = \langle v_0, v_1, \dots, v_n \rangle$

$$w(p) = \bigcirc_{k=1}^n w(v_{k-1}, v_k) = \text{sum of edges weights on path } p$$

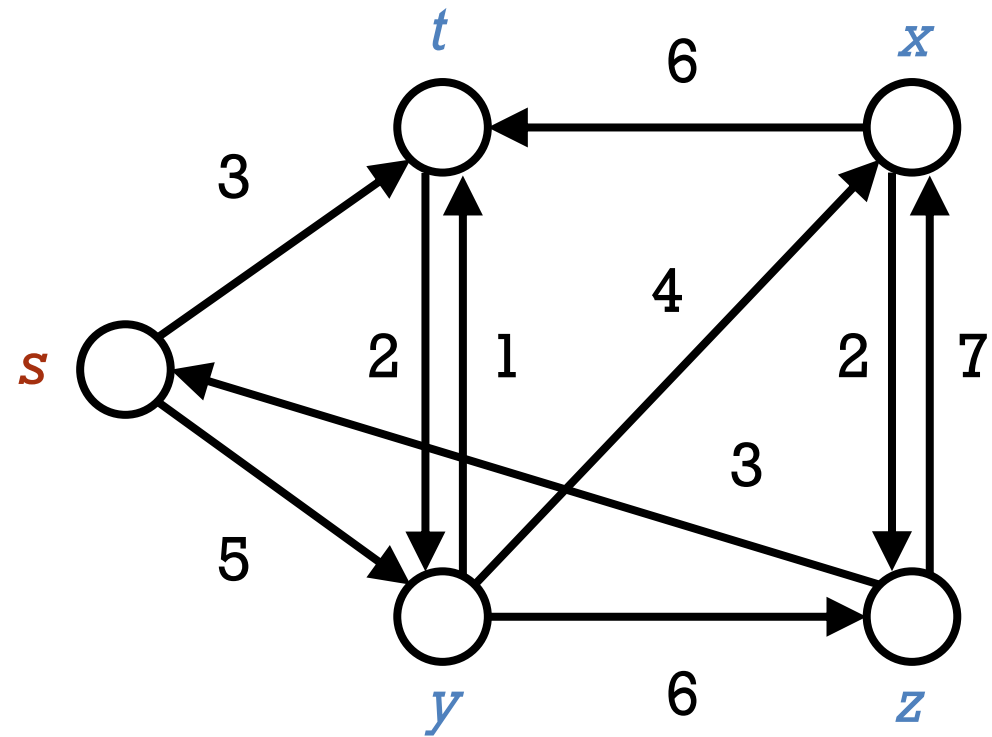
Shortest-path weight u to v :

$$\delta(u, v) = \begin{cases} \min \left\{ w(p) : u \xrightarrow{p} v \right\} & \text{If there exists a path } u \rightsquigarrow v. \\ \infty & \text{Otherwise.} \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$

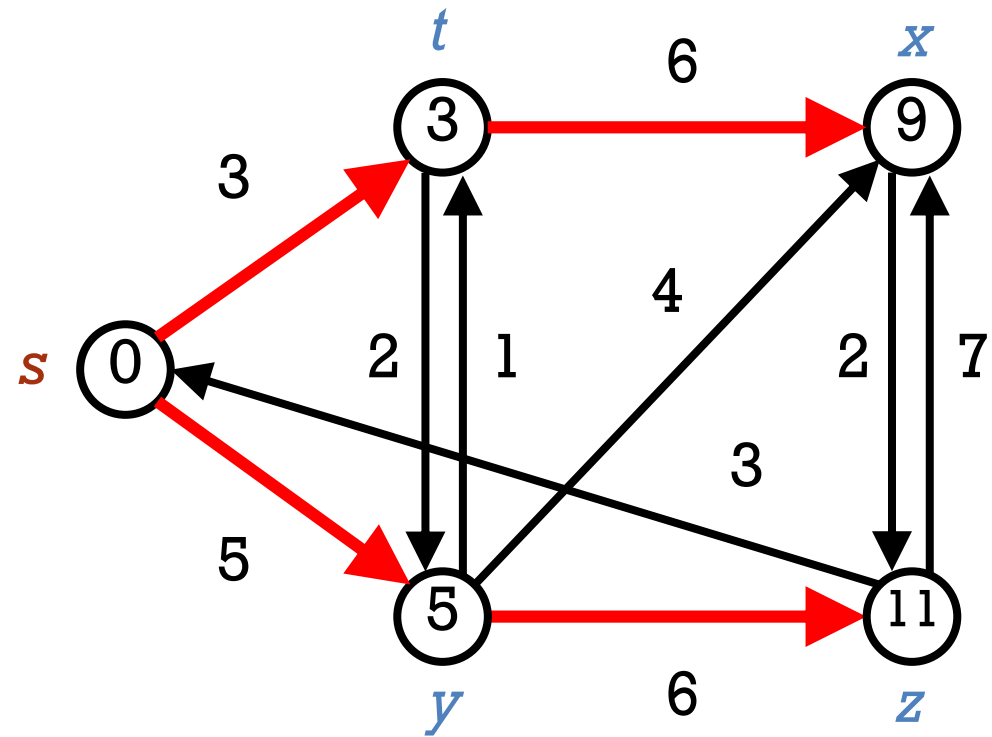
Generalization of breadth-first search to weighted graphs

EXAMPLE



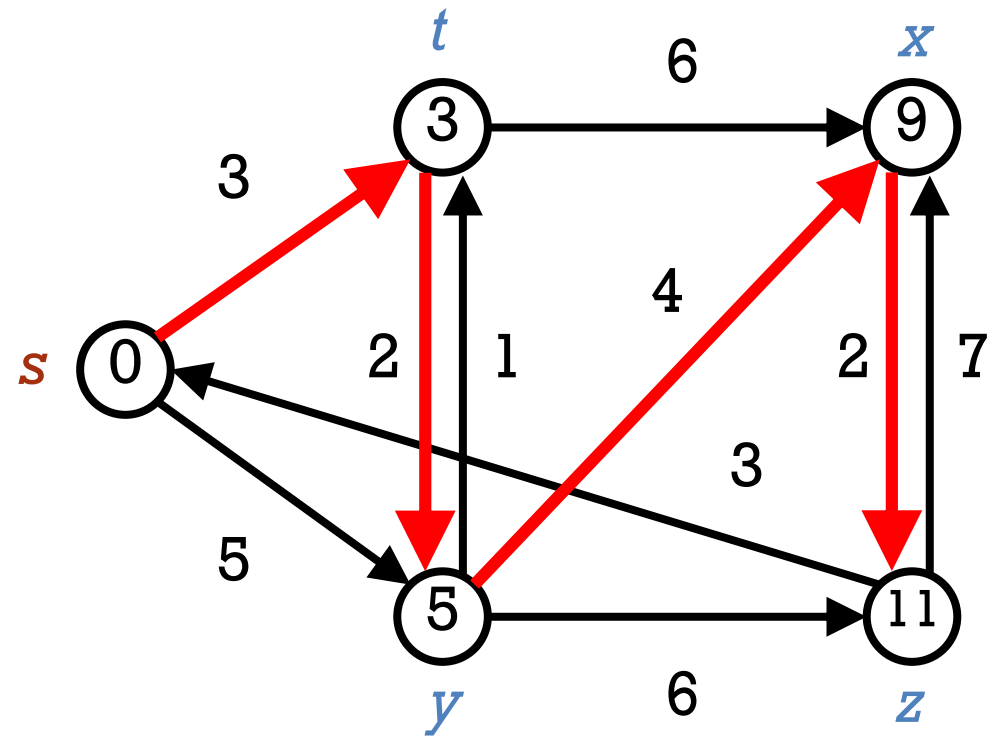
Shortest path from s ?

EXAMPLE



Shortest paths are organized as a tree.
Vertices store the length of the shortest path from *s*.

EXAMPLE



Shortest paths are not necessarily unique!

VARIANTS

- **Single-source:** Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$.
- **Single-destination:** Find shortest paths to a given destination vertex.
- **Single-pair:** Find shortest path from u to v .
Note: No way to know that is better in worst case than solving the single-source problem!
- **All-pairs:** Find shortest path from u to v for all $u, v \in V$.

PRINCIPLE OF A SINGLE-SOURCE SHORTEST-PATH ALGORITHM

For each vertex $v \in V$:

- $d[v] = \delta(s, v)$.
 - Initially, $d[v] = \infty$.
 - Reduces as algorithms progress, but always maintain $d[v] \geq \delta(s, v)$.
 - Call $d[v]$ a **shortest-path estimate**.
- $\pi[v]$ = predecessor of v on a shortest path from s .
 - If no predecessor, $\pi[v] = \text{null}$.
 - π induces a tree - **shortest-path tree**

$\delta(s, v)$ is the absolute shortest path

$d[v]$ is our current estimate of the shortest path

GENERIC ALGORITHM STRUCTURE

1. Initialization
2. Scan vertices and relax edges

The algorithms differ in the order and how many times they relax each edge.

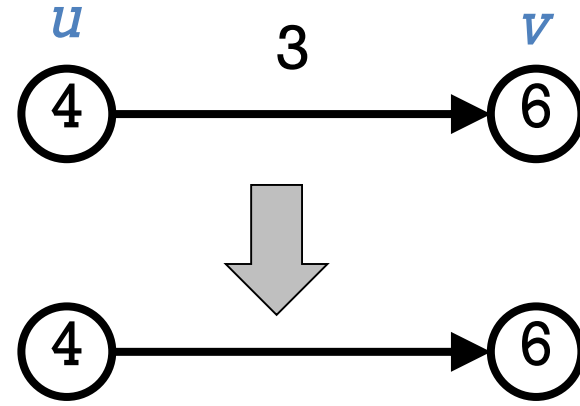
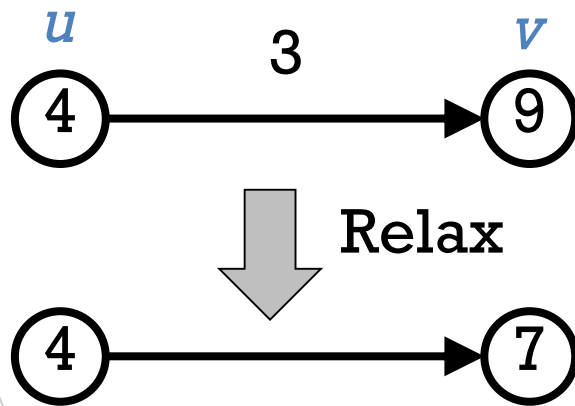
INITIALIZATION

```
INIT-SINGLE-SOURCE( $V, s$ )  
  for each  $v \in V$  do  
     $d[v] \leftarrow \infty$   
     $\pi[v] \leftarrow \text{null}$   
   $d[s] \leftarrow 0$ 
```

This is used to reduce $d[v]$ during the execution of the algorithm.

RELAXING AN EDGE

```
RELAX ( $u, v, w$ )  
  if  $d[v] > d[u] + w(u, v)$  then  
     $d[v] \leftarrow d[u] + w(u, v)$   
     $\pi[v] \leftarrow u$ 
```



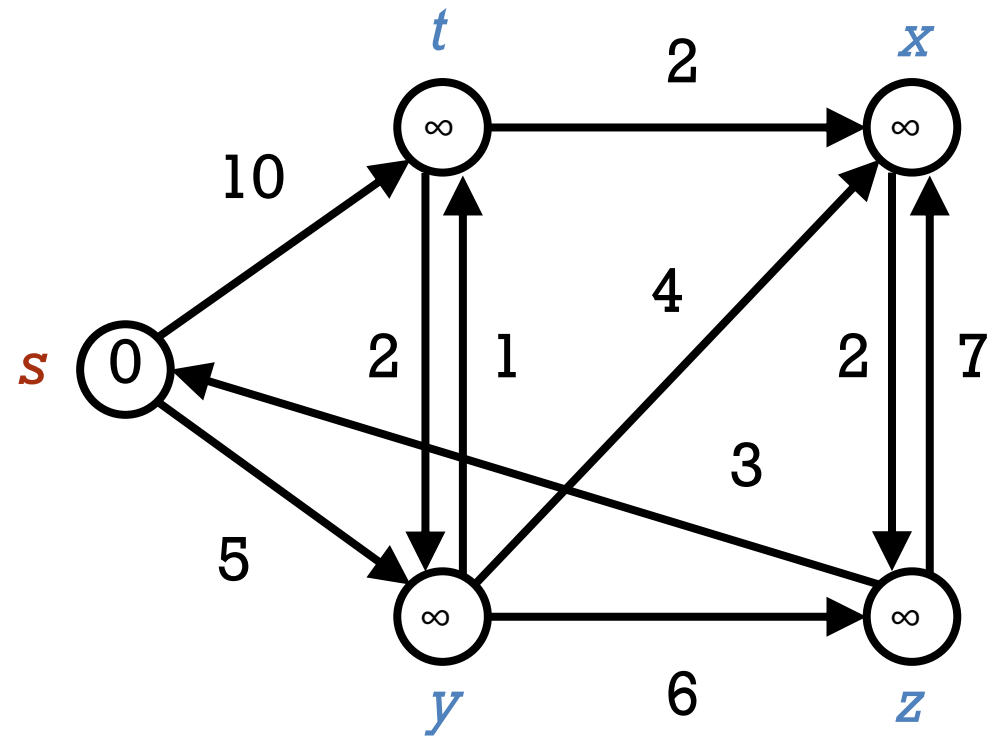
DIJKSTRA'S ALGORITHM

- No negative-weight edges.
- Weighted version of BFS:
 - Instead of a FIFO queue, uses a priority queue.
 - Keys are shortest-path weights ($d[v]$).
- Have two sets of vertices:
 - S = vertices whose final shortest-path weights are determined,
 - Q = priority queue = $V \setminus S$.

DIJKSTRA'S ALGORITHM

```
DIJKSTRA( $V, E, w, s$ )  
INIT-SINGLE-SOURCE( $V, s$ )  
 $S \leftarrow \emptyset$   
 $Q \leftarrow V$   
while  $Q \neq \emptyset$  do  
     $u \leftarrow \text{REMOVE-MIN}(Q)$   
     $S \leftarrow S \cup \{u\}$   
    for each vertex  $v \in \text{Adj}[u]$  do  
        RELAX( $u, v, w$ )
```

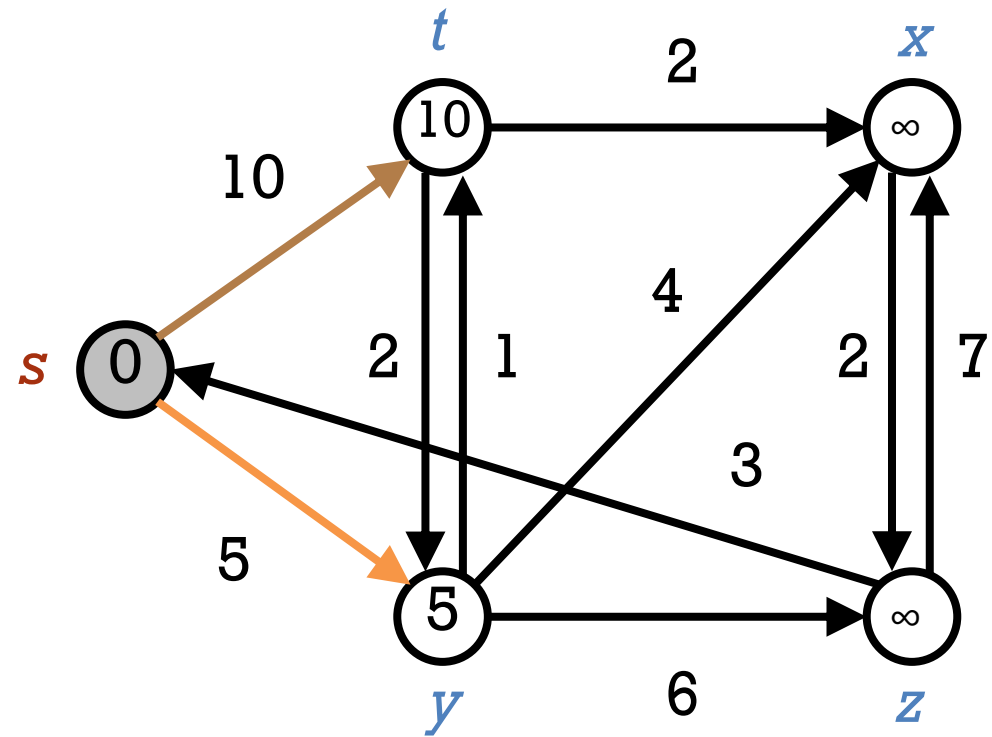

EXAMPLE



Q

s	t	y	x	z
---	---	---	---	---

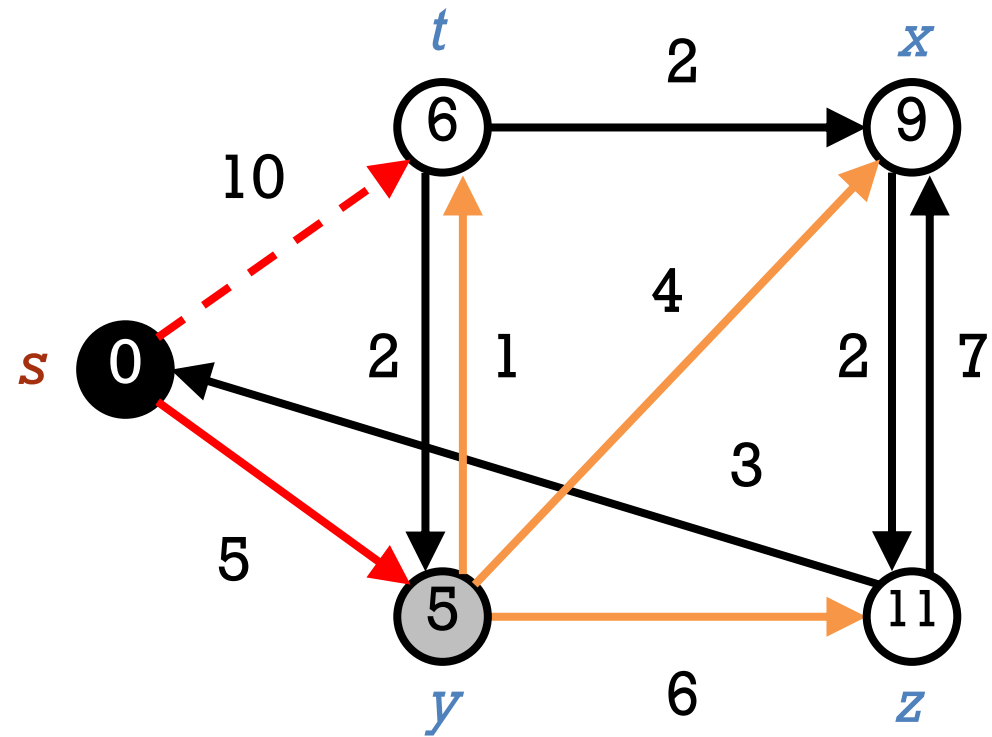
EXAMPLE



Q

y	t	x	z	
---	---	---	---	--

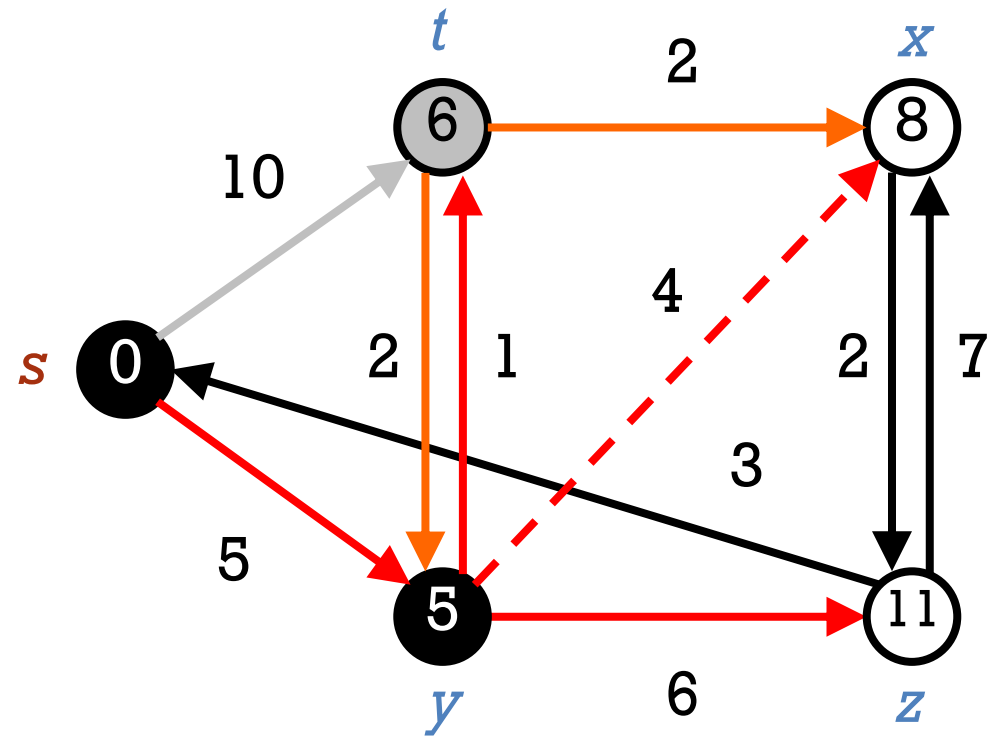
EXAMPLE



Q

t	x	z		
-----	-----	-----	--	--

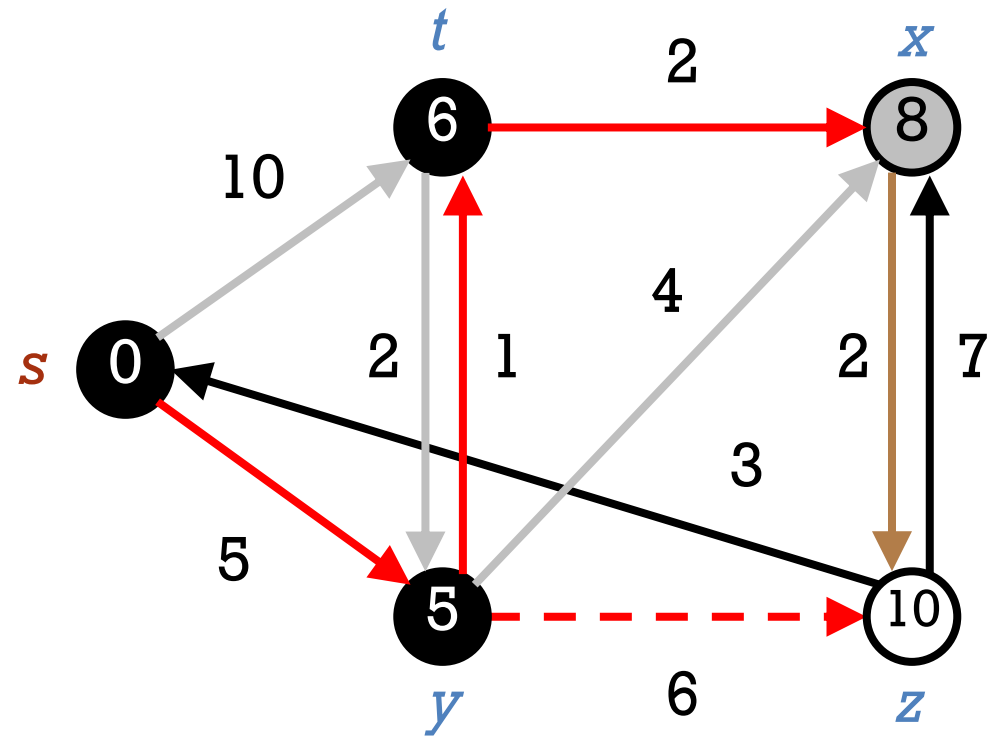
EXAMPLE



Q

<i>x</i>	<i>z</i>			
----------	----------	--	--	--

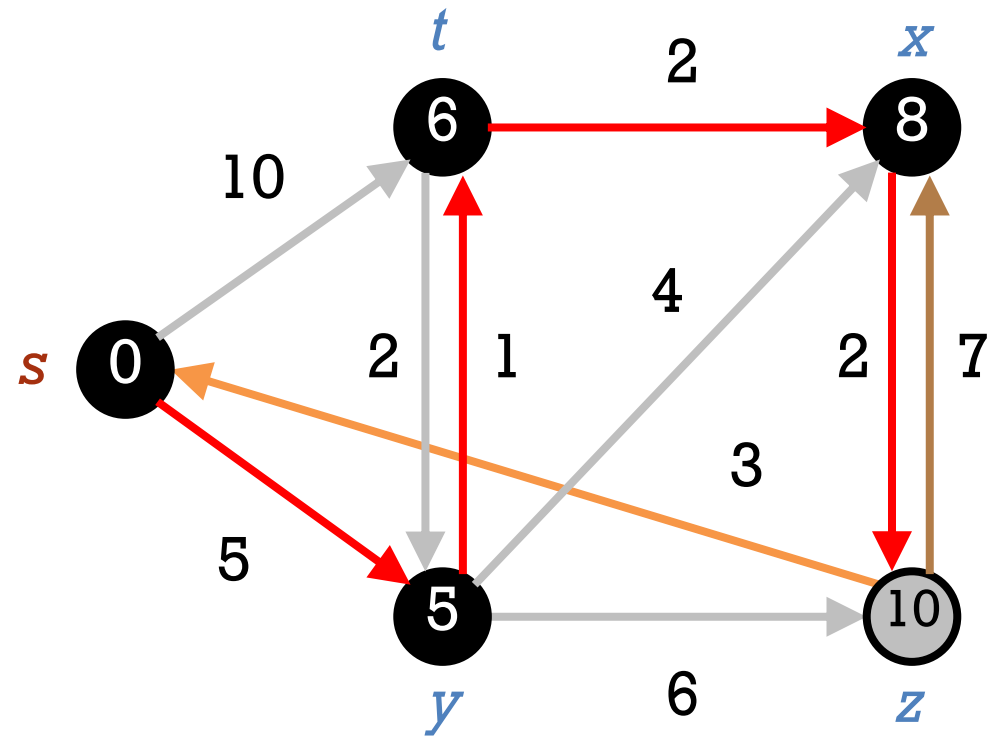
EXAMPLE



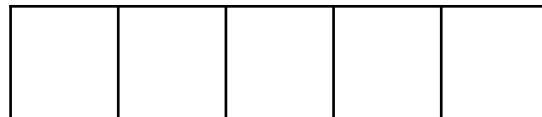
Q

z				
-----	--	--	--	--

EXAMPLE



Q

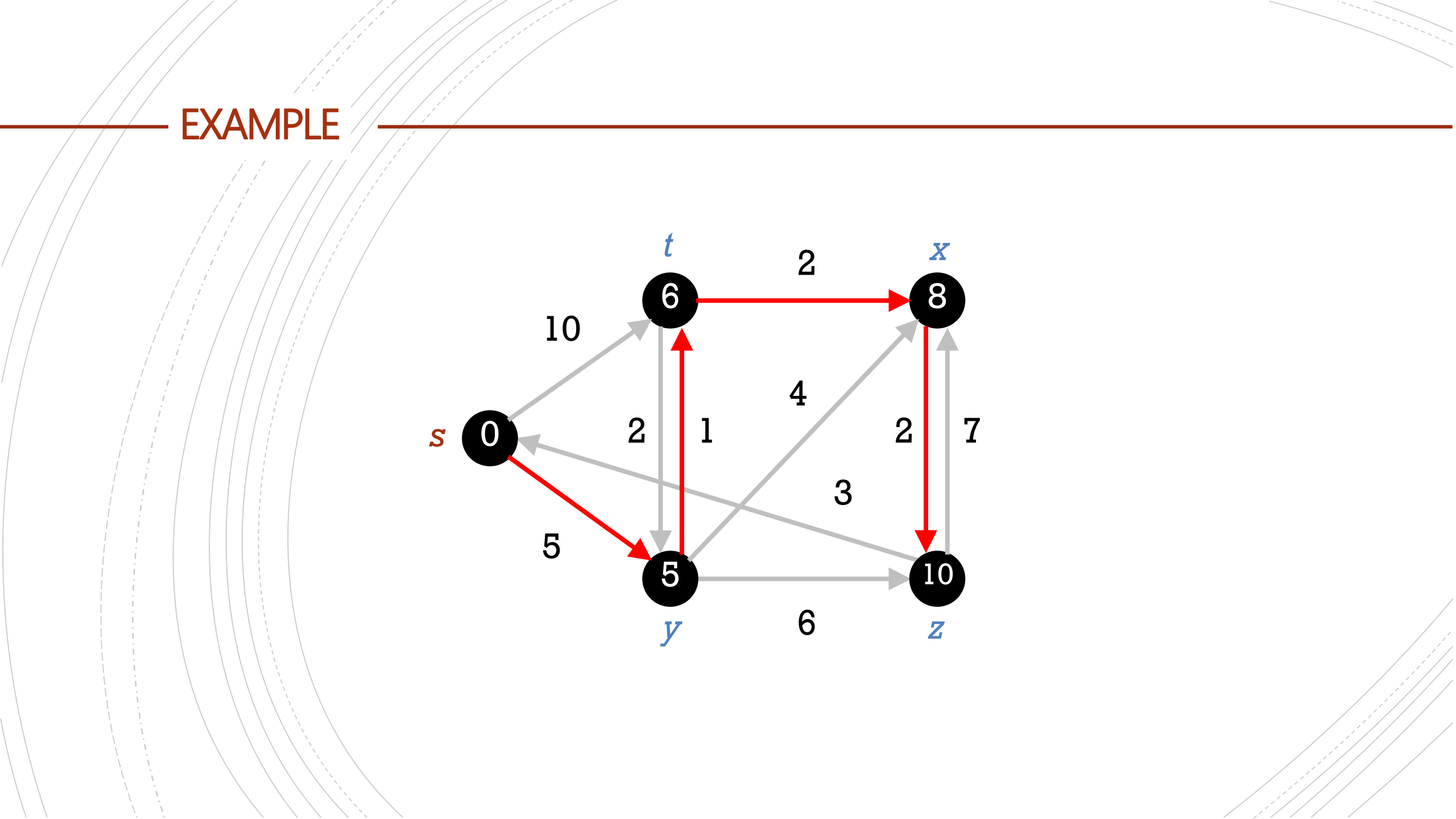


EXAMPLE

The diagram shows a directed graph with 6 nodes (0, 5, 6, 8, 10) and weighted edges. The nodes are labeled with blue letters: s for node 0, t for node 6, x for node 8, y for node 5, and z for node 10. The edges and their weights are:

- Node 0 to Node 6: weight 10
- Node 0 to Node 5: weight 5
- Node 0 to Node 10: weight 3
- Node 5 to Node 6: weight 2
- Node 5 to Node 10: weight 6
- Node 6 to Node 8: weight 2
- Node 6 to Node 10: weight 4
- Node 8 to Node 10: weight 2
- Node 10 to Node 8: weight 7

Red arrows highlight a specific path from node 0 to node 8: 0 → 5 → 6 → 8.



Thank You

