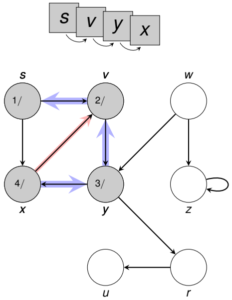


6.1 & 6.2: Graph Searching

Frank Stajano Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

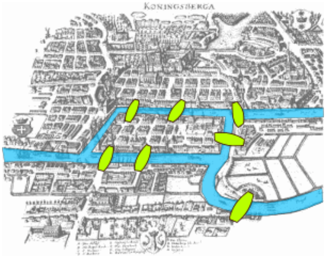
Introduction to Graphs and Graph Searching

Breadth-First Search

Depth-First Search

Topological Sort

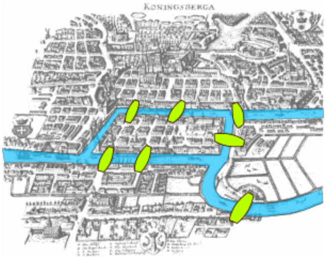
Origin of Graph Theory



Source: Wikipedia

Seven Bridges at Königsberg 1737

Origin of Graph Theory



Source: Wikipedia

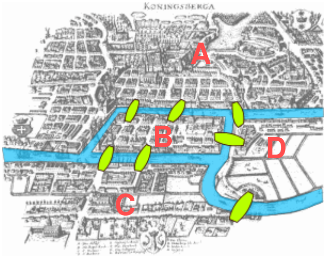
Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?

Origin of Graph Theory



Source: Wikipedia

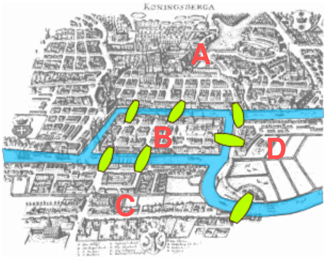
Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?

Origin of Graph Theory



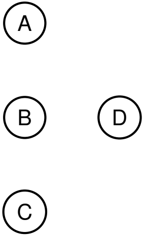
Source: Wikipedia

Seven Bridges at Königsberg 1737

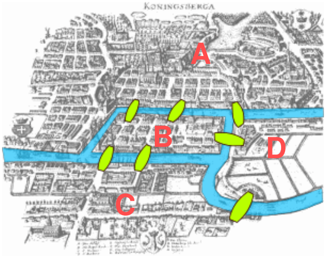
Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?



Origin of Graph Theory

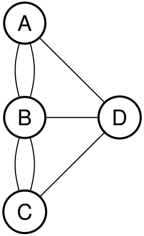


Source: Wikipedia

Source: Wikipedia

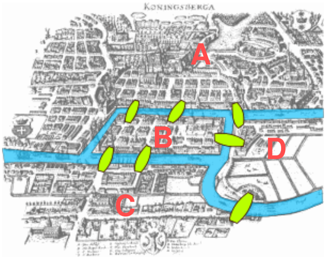
Seven Bridges at Königsberg 1737

Leonhard Euler (1707-1783)



Is there a tour which crosses each bridge **exactly once**?

Origin of Graph Theory

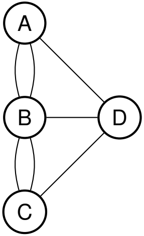


Source: Wikipedia

Seven Bridges at Königsberg 1737

Source: Wikipedia

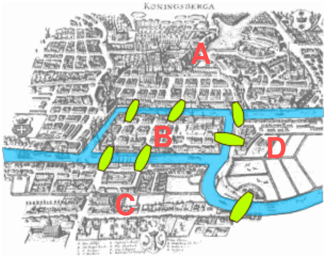
Leonhard Euler (1707-1783)



Is there a tour which crosses each bridge **exactly once**?

Is there a tour which visits every island **exactly once**?

Origin of Graph Theory

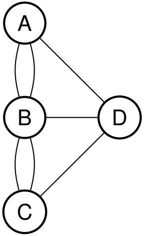


Source: Wikipedia

Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)



Is there a tour which crosses each bridge **exactly once**?

Is there a tour which visits every island **exactly once**?
$\rightsquigarrow$ 1B course: Complexity Theory

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

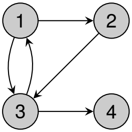
- V: the set of vertices
- E: the set of edges (arcs)

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

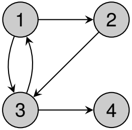


What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)



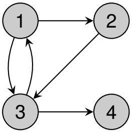
$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

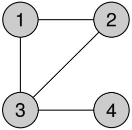


Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$



What is a Graph?

Directed Graph

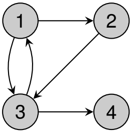
A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

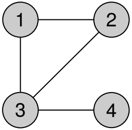
Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$



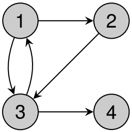
$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

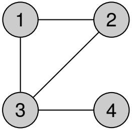


$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

Paths and Connectivity

- A sequence of edges between two vertices forms a path

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

Undirected Graph

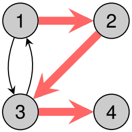
A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

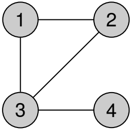
Paths and Connectivity

- A sequence of edges between two vertices forms a path

Path $p = (1, 2, 3, 4)$



$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$



$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

Undirected Graph

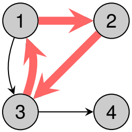
A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

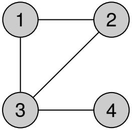
Paths and Connectivity

- A sequence of edges between two vertices forms a path

Path $p = (1, 2, 3, 1)$, which is a cycle



$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$



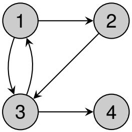
$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

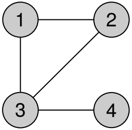


$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

Paths and Connectivity

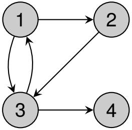
- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

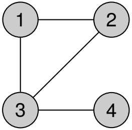


$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

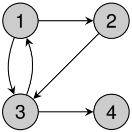
What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

G is not connected



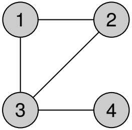
$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

G is connected



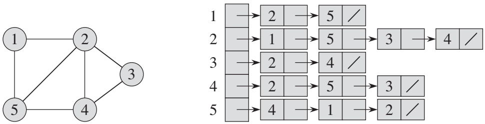
$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

Later: edge-weighted graphs $G = (V, E, w)$

Representations of Directed and Undirected Graphs



(a) (b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Representations of Directed and Undirected Graphs

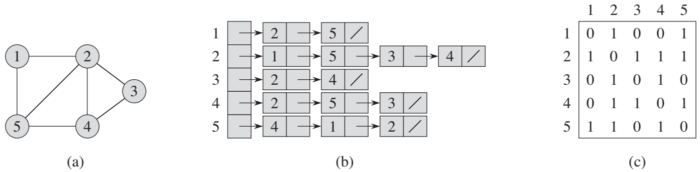


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Most times we will use the adjacency-list representation!

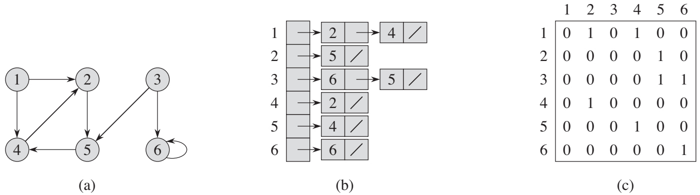


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Overview

Amortized Analysis

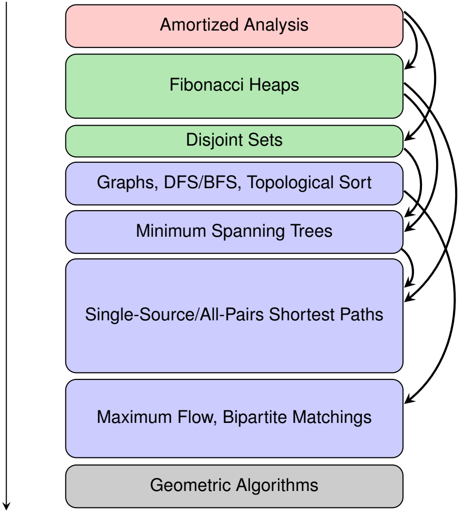
Fibonacci Heaps

Disjoint Sets

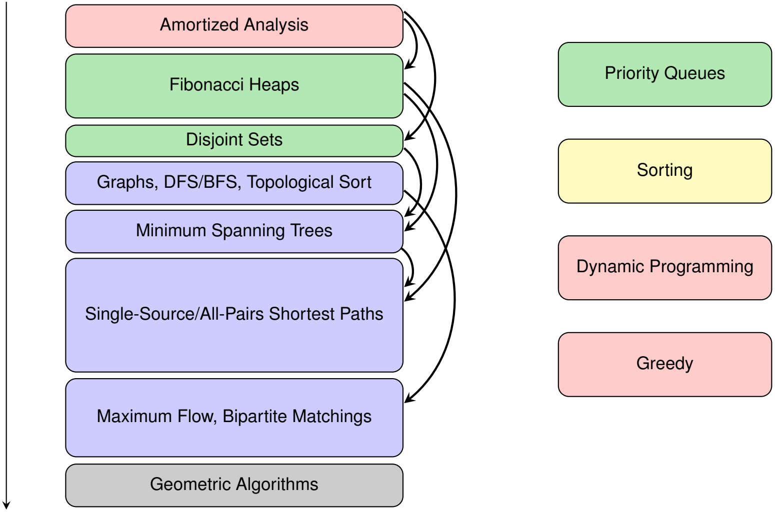
Overview

- Amortized Analysis
- Fibonacci Heaps
- Disjoint Sets
- Graphs, DFS/BFS, Topological Sort
- Minimum Spanning Trees
- Single-Source/All-Pairs Shortest Paths
- Maximum Flow, Bipartite Matchings
- Geometric Algorithms

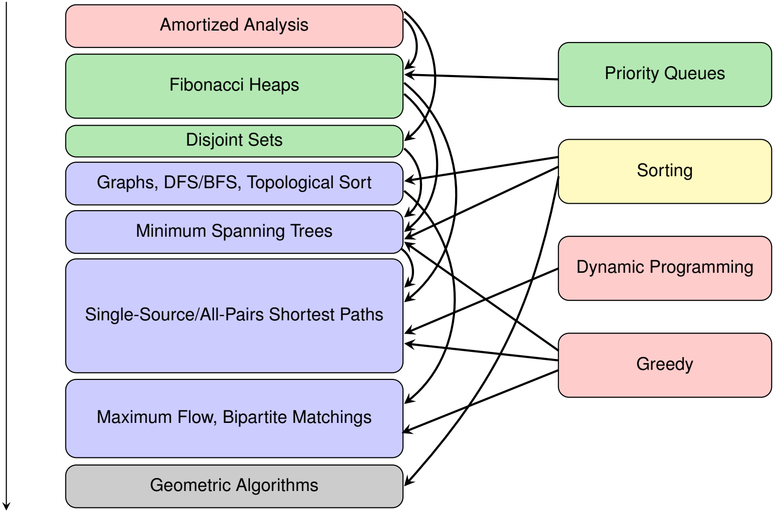
Overview


Amortized Analysis
Fibonacci Heaps
Disjoint Sets
Graphs, DFS/BFS, Topological Sort
Minimum Spanning Trees
Single-Source/All-Pairs Shortest Paths
Maximum Flow, Bipartite Matchings
Geometric Algorithms

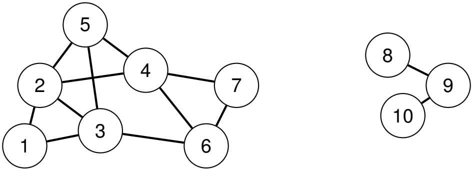
Overview


Amortized Analysis
Fibonacci Heaps
Disjoint Sets
Graphs, DFS/BFS, Topological Sort
Minimum Spanning Trees
Single-Source/All-Pairs Shortest Paths
Maximum Flow, Bipartite Matchings
Geometric Algorithms
Priority Queues
Sorting
Dynamic Programming
Greedy

Overview


Amortized Analysis
Fibonacci Heaps
Disjoint Sets
Graphs, DFS/BFS, Topological Sort
Minimum Spanning Trees
Single-Source/All-Pairs Shortest Paths
Maximum Flow, Bipartite Matchings
Geometric Algorithms
Priority Queues
Sorting
Dynamic Programming
Greedy

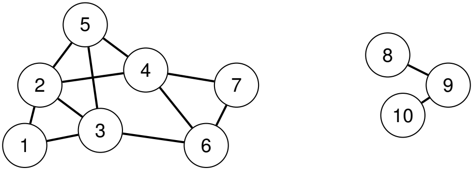
Graph Searching



Overview

- Graph searching means traversing a graph via the edges in order to visit all vertices
- useful for identifying connected components, computing the diameter etc.

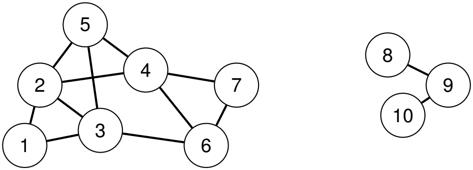
Graph Searching



Overview

- Graph searching means traversing a graph via the edges in order to visit all vertices
- useful for identifying connected components, computing the diameter etc.
- Two strategies: Breadth-First-Search and Depth-First-Search

Graph Searching



Overview

- Graph searching means traversing a graph via the edges in order to visit all vertices
- useful for identifying connected components, computing the diameter etc.
- Two strategies: Breadth-First-Search and Depth-First-Search

Measure time complexity in terms of the size of V and E (often write just V instead of $|V|$, and E instead of $|E|$)

Outline

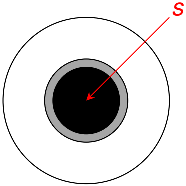
Introduction to Graphs and Graph Searching

Breadth-First Search

Depth-First Search

Topological Sort

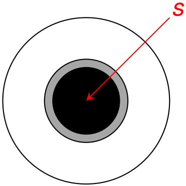
Breadth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s

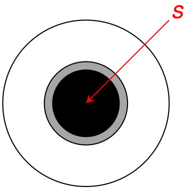
Breadth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s
- BFS sends out a wave from $s \rightsquigarrow$ compute distances/shortest paths

Breadth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s
- BFS sends out a wave from $s \rightsquigarrow$ compute distances/shortest paths
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors (=adjacent vertices)

  Black = Visited and all neighbors

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

   assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
      v.predecessor = None
      v.d = Infinity # .d = distance from s
      v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

  assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
    v.predecessor = None
    v.d = Infinity # .d = distance from s
    v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
   u = Q.extract()
   assert (u.colour == "grey")
   for v in u.adjacent()
      if v.colour = "white"
         v.colour = "grey"
         v.d = u.d+1
         v.predecessor = u
         Q.insert(v)
   u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

  assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
    v.predecessor = None
    v.d = Infinity # .d = distance from s
    v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

    assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
    v.predecessor = None
    v.d = Infinity # .d = distance from s
    v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors
- Runtime ???

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

  assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
    v.predecessor = None
    v.d = Infinity # .d = distance from s
    v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors
- Runtime ???

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

    assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
      v.predecessor = None
      v.d = Infinity # .d = distance from s
      v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors
- Runtime ???

Assuming that all executions of the FOR-loop for u takes $O(|u.adj|)$ (**adjacency list model!**)

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

  assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
      v.predecessor = None
      v.d = Infinity # .d = distance from s
      v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:
  - White = Unvisited
  - Grey = Visited, but not all neighbors
  - Black = Visited and all neighbors
- Runtime ???

Assuming that all executions of the FOR-loop for u takes $O(|u.adj|)$ (**adjacency list model!**)

$$\sum_{u \in V} |u.adj| = 2|E|$$

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
   Run BFS on the given graph G
   starting from source s

  assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
     v.predecessor = None
     v.d = Infinity # .d = distance from s
     v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
   u = Q.extract()
   assert (u.colour == "grey")
   for v in u.adjacent()
      if v.colour = "white"
         v.colour = "grey"
         v.d = u.d+1
         v.predecessor = u
         Q.insert(v)
   u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

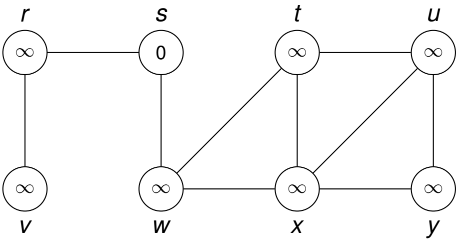
  Black = Visited and all neighbors
- Runtime $O(V + E)$

Assuming that all executions of the FOR-loop for u takes $O(|u.adj|)$ (**adjacency list model!**)

$$\sum_{u \in V} |u.adj| = 2|E|$$

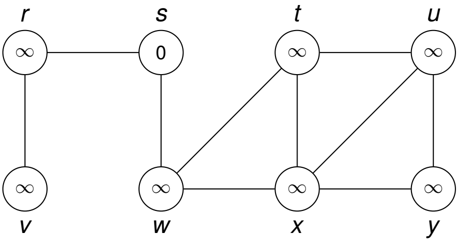
Execution of BFS (Figure 22.3)

Queue:



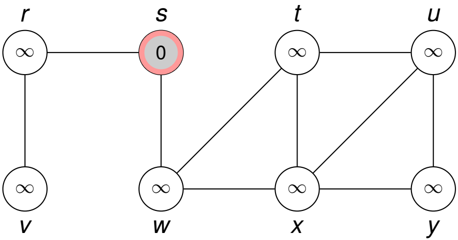
Execution of BFS (Figure 22.3)

Queue: *s*



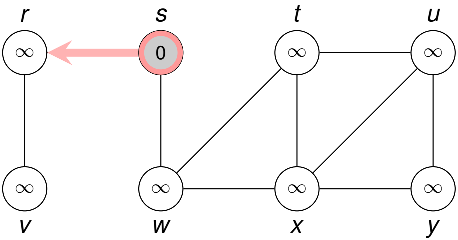
Execution of BFS (Figure 22.3)

Queue: ~~s~~



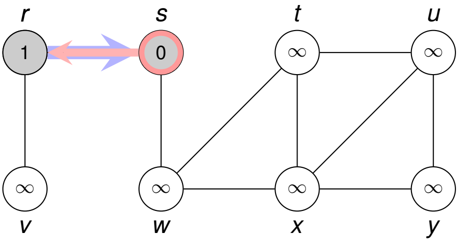
Execution of BFS (Figure 22.3)

Queue: ~~s~~



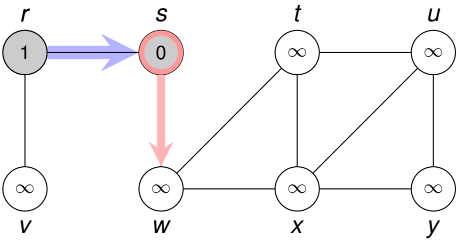
Execution of BFS (Figure 22.3)

Queue: ~~s~~ r



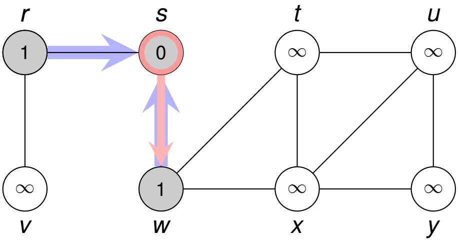
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ *r*



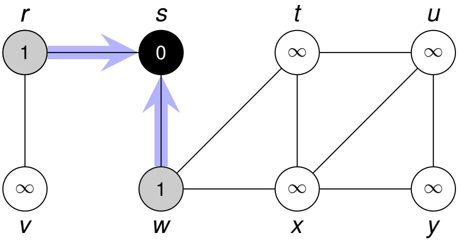
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ *r* *w*



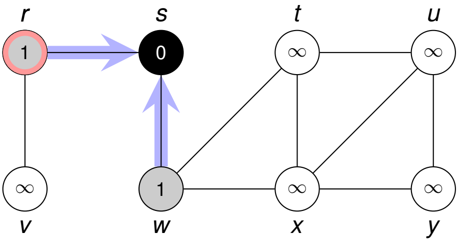
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ *r* *w*



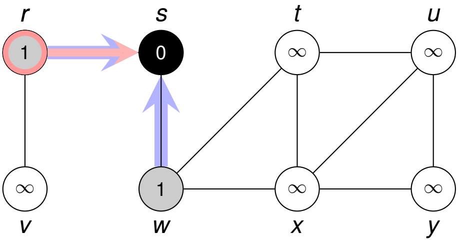
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ *w*



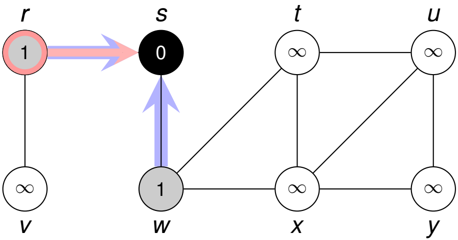
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ *w*



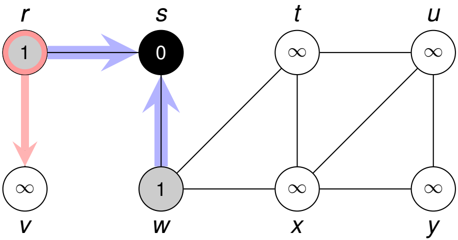
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ *w*



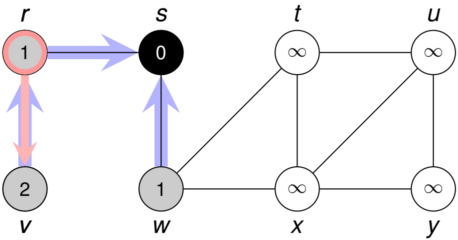
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ *w*



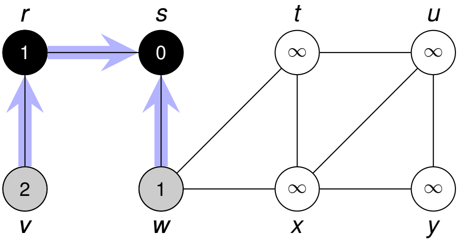
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ w v



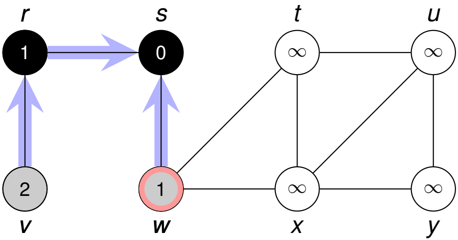
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ *w* *v*



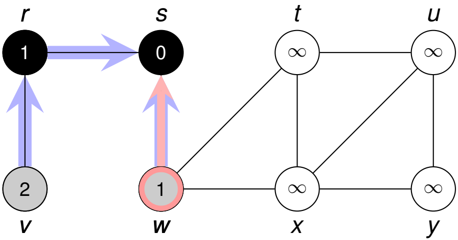
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ *v*



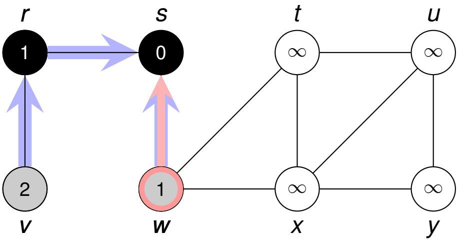
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ v



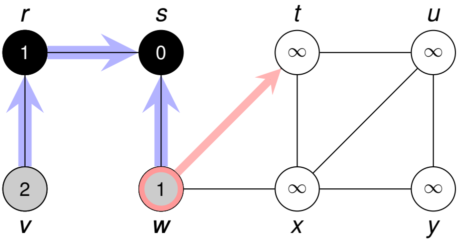
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ v



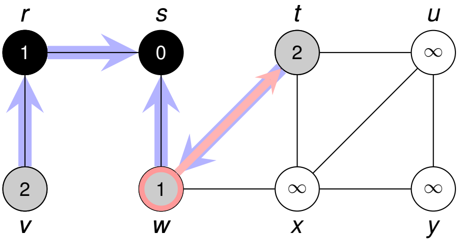
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ v



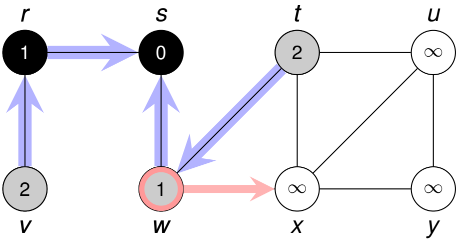
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ v t



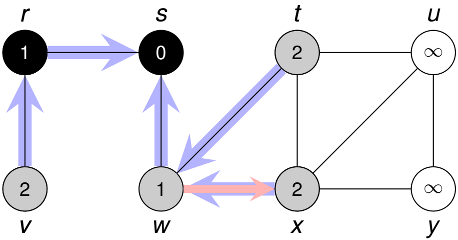
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ v t



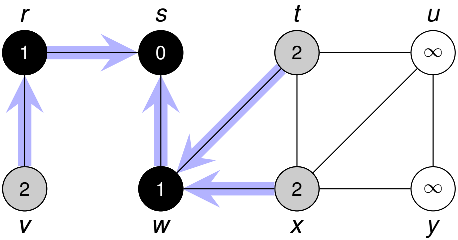
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ *v* *t* *x*



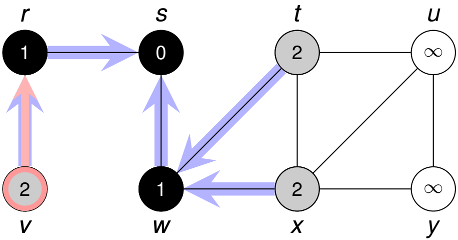
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ *v* *t* *x*



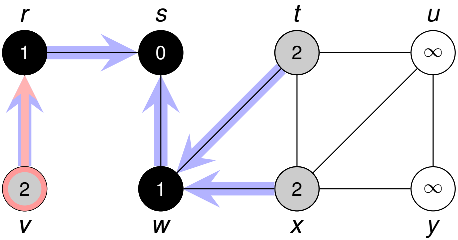
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ *t* *x*



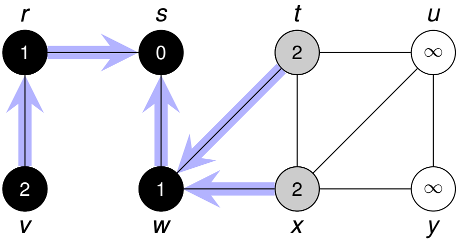
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ *t* *x*



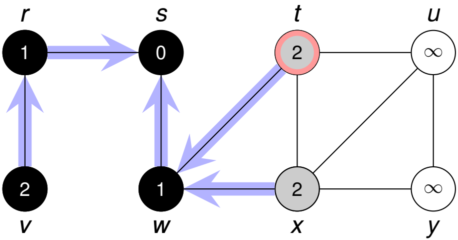
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ *t* *x*



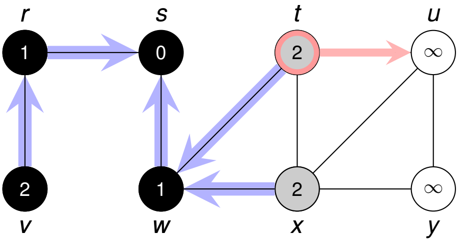
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ x



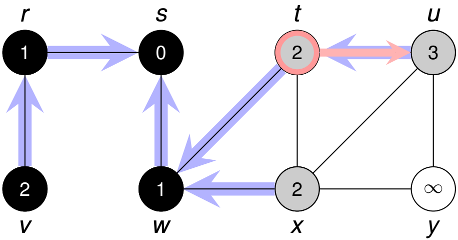
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ x



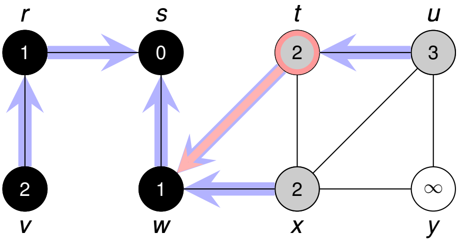
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ *x* *u*



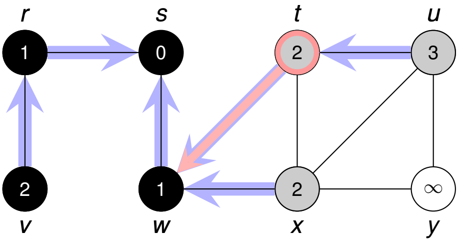
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ *x* *u*



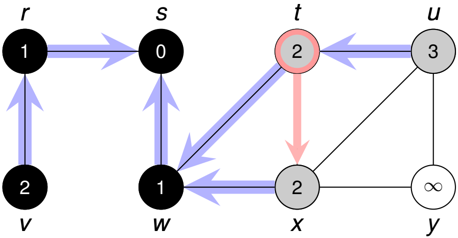
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ *x* *u*



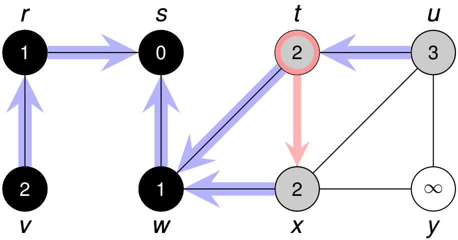
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ *x* *u*



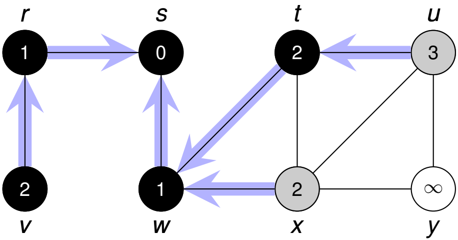
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ *x* *u*



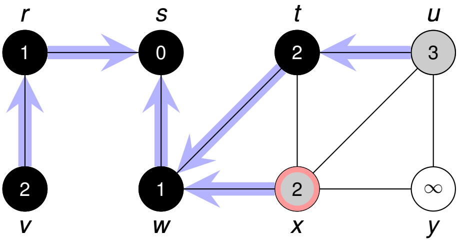
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ x u



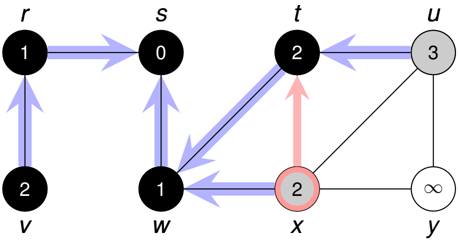
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ *u*



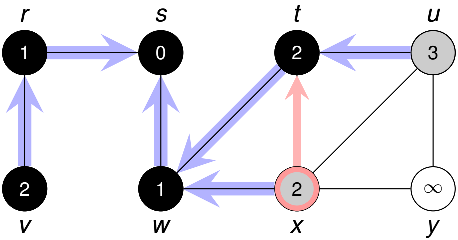
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ u



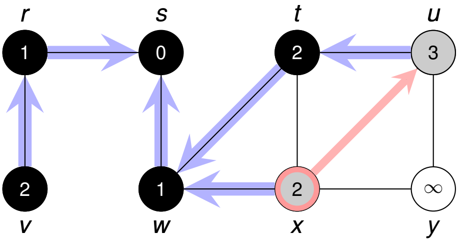
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ u



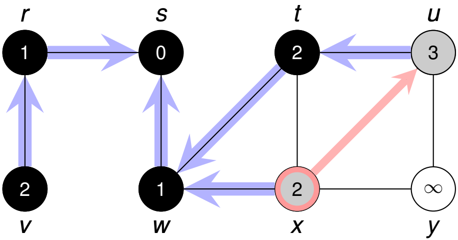
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ u



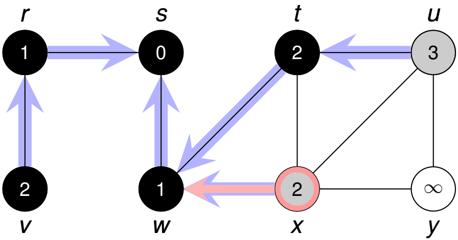
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ *u*



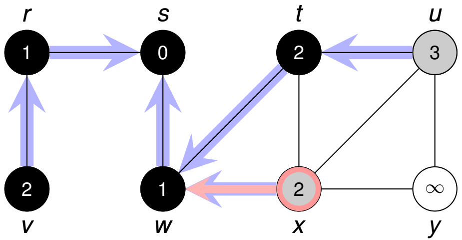
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ u



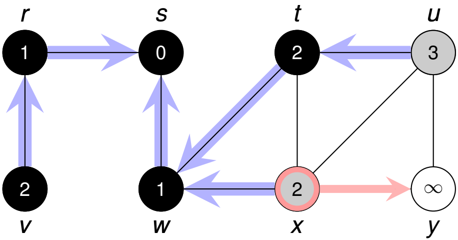
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ u



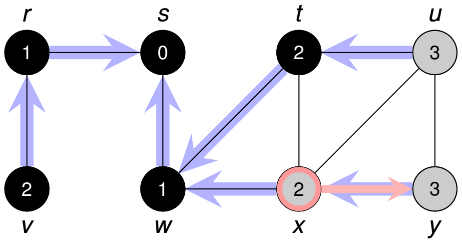
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ *u*



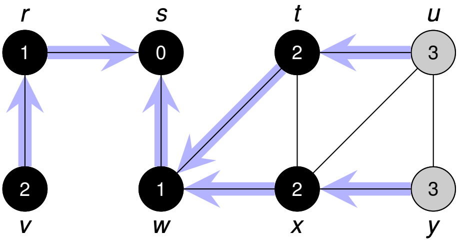
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ *u* *y*

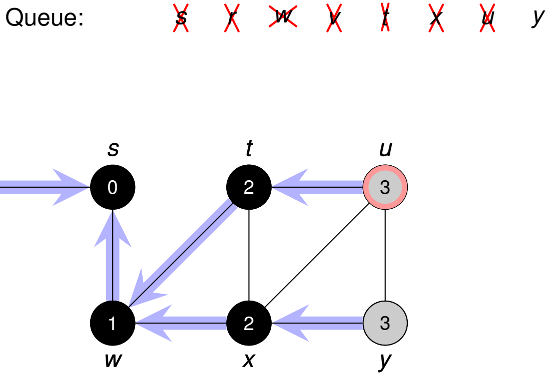


Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ *u* *y*

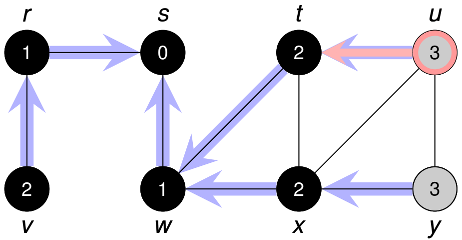


Execution of BFS (Figure 22.3)



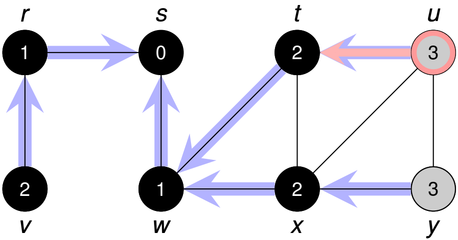
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



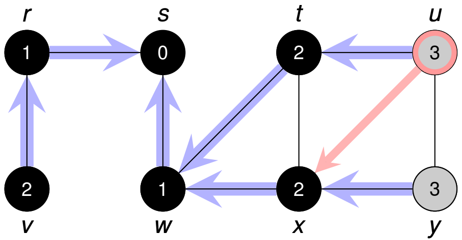
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



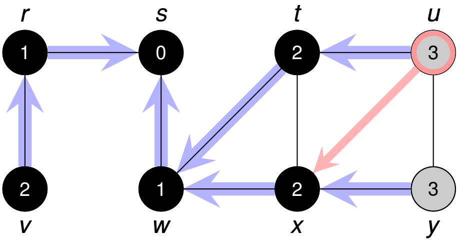
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



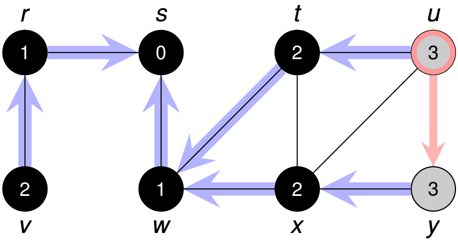
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ ~~*u*~~ *y*



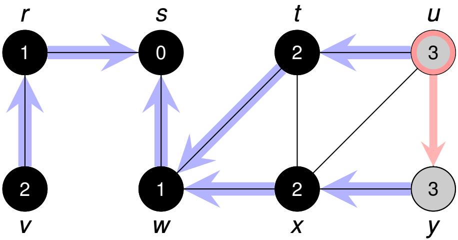
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



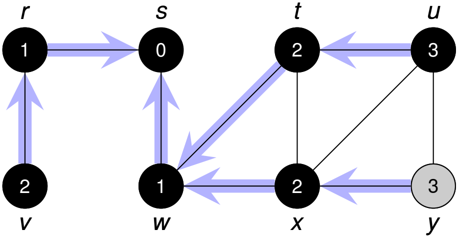
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



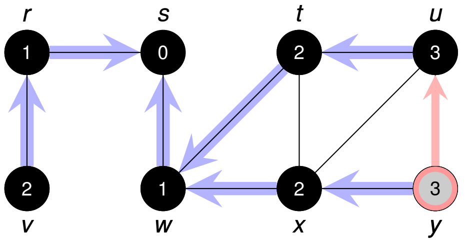
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



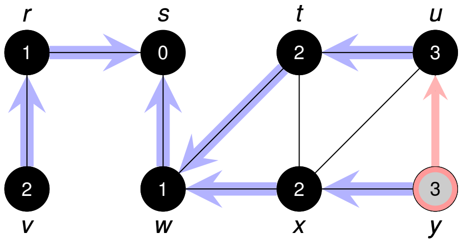
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ ~~*u*~~ *y*



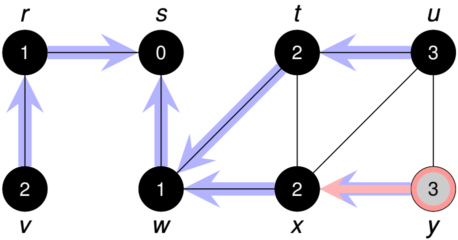
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ ~~y~~



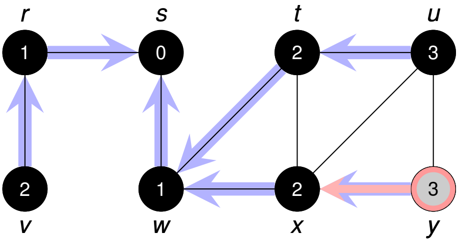
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ ~~y~~



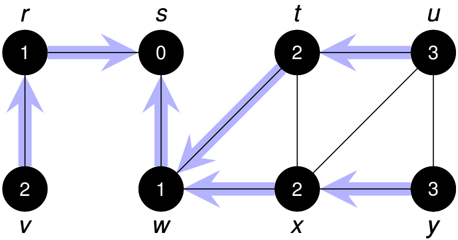
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ ~~y~~



Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ ~~y~~



Outline

Introduction to Graphs and Graph Searching

Breadth-First Search

Depth-First Search

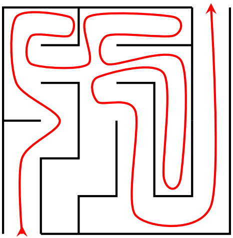
Topological Sort

Depth-First Search: Basic Ideas

Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s

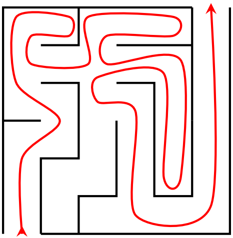
Depth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s
- As soon as we discover a vertex, explore from it $\rightsquigarrow$ Solving Mazes

Depth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s
- As soon as we discover a vertex, explore from it $\rightsquigarrow$ Solving Mazes
- Two time stamps for every vertex: Discovery Time, Finishing Time

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

  assert(s in G.vertices())

  # Initialize graph
  for v in G.vertices():
      v.predecessor = None
      v.colour = "white"
  dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
  s.colour = "grey"
  s.d = time() # .d = discovery time
  for v in s.adjacent()
      if v.colour = "white"
          v.predecessor = s
          dfsRecurse(G,v)
  s.colour = "black"
  s.f = time() # .f = finish time
```

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

  assert(s in G.vertices())

  # Initialize graph
  for v in G.vertices():
      v.predecessor = None
      v.colour = "white"
  dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
  s.colour = "grey"
  s.d = time() # .d = discovery time
  for v in s.adjacent()
      if v.colour = "white"
          v.predecessor = s
          dfsRecurse(G,v)
  s.colour = "black"
  s.f = time() # .f = finish time
```

- We always go deeper before visiting other neighbors

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

   assert(s in G.vertices())

   # Initialize graph
   for v in G.vertices():
      v.predecessor = None
      v.colour = "white"
   dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
   s.colour = "grey"
   s.d = time() # .d = discovery time
   for v in s.adjacent()
       if v.colour = "white"
           v.predecessor = s
           dfsRecurse(G,v)
   s.colour = "black"
   s.f = time() # .f = finish time
```

- We always go deeper before visiting other neighbors
- Discovery and Finish times, *.d* and *.f*

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

   assert(s in G.vertices())

   # Initialize graph
   for v in G.vertices():
       v.predecessor = None
       v.colour = "white"
   dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
   s.colour = "grey"
   s.d = time() # .d = discovery time
   for v in s.adjacent()
       if v.colour = "white"
           v.predecessor = s
           dfsRecurse(G,v)
   s.colour = "black"
   s.f = time() # .f = finish time
```

- We always go deeper before visiting other neighbors
- Discovery and Finish times, *.d* and *.f*
- Vertex Colours:

White = Unvisited

Grey = Visited, but not all neighbors

Black = Visited and all neighbors

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

    assert(s in G.vertices())

    # Initialize graph
    for v in G.vertices():
        v.predecessor = None
        v.colour = "white"
    dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
    s.colour = "grey"
    s.d = time() # .d = discovery time
    for v in s.adjacent()
        if v.colour = "white"
            v.predecessor = s
            dfsRecurse(G,v)
    s.colour = "black"
    s.f = time() # .f = finish time
```

- We always go deeper before visiting other neighbors
- Discovery and Finish times, *.d* and *.f*
- Vertex Colours:

White = Unvisited

Grey = Visited, but not all neighbors

Black = Visited and all neighbors

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

    assert(s in G.vertices())

    # Initialize graph
    for v in G.vertices():
        v.predecessor = None
        v.colour = "white"
    dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
    s.colour = "grey"
    s.d = time() # .d = discovery time
    for v in s.adjacent()
        if v.colour = "white"
            v.predecessor = s
            dfsRecurse(G,v)
    s.colour = "black"
    s.f = time() # .f = finish time
```

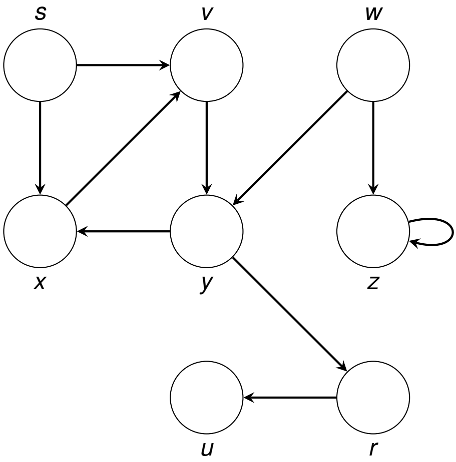
- We always go deeper before visiting other neighbors
- Discovery and Finish times, *.d* and *.f*
- Vertex Colours:

  White = Unvisited

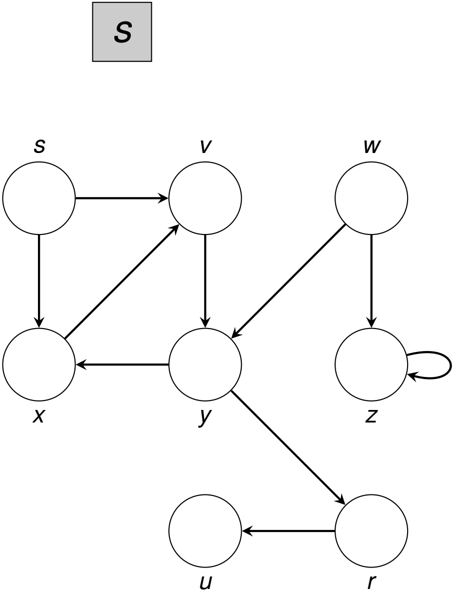
  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors
- Runtime $O(V + E)$

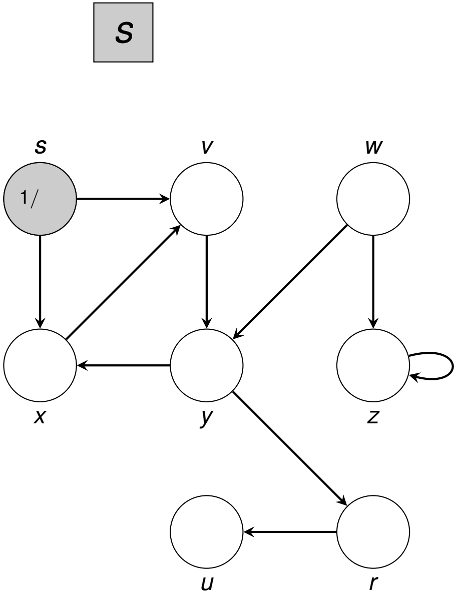
Execution of DFS



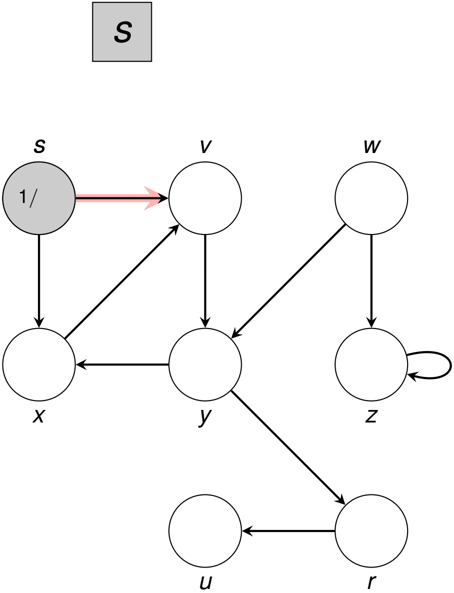
Execution of DFS



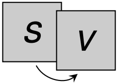
Execution of DFS

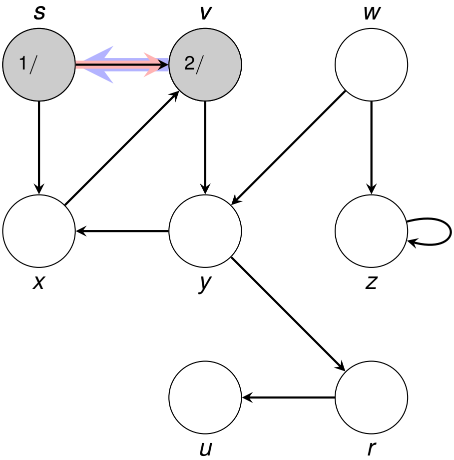


Execution of DFS

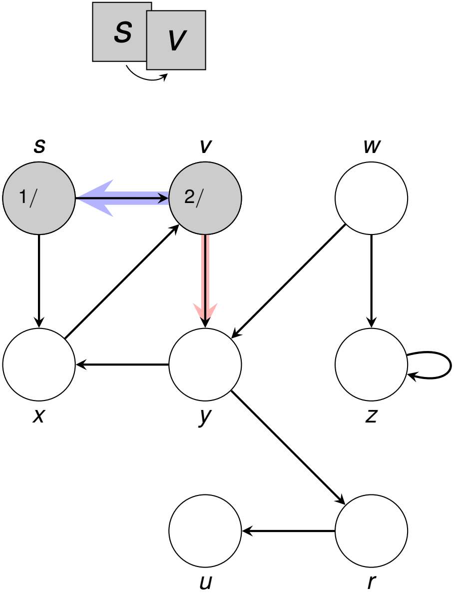


Execution of DFS

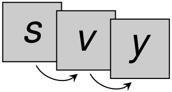


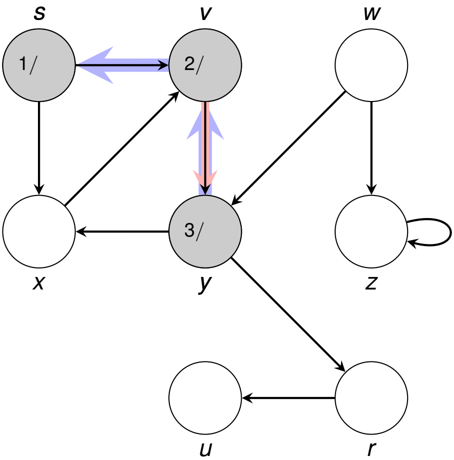


Execution of DFS

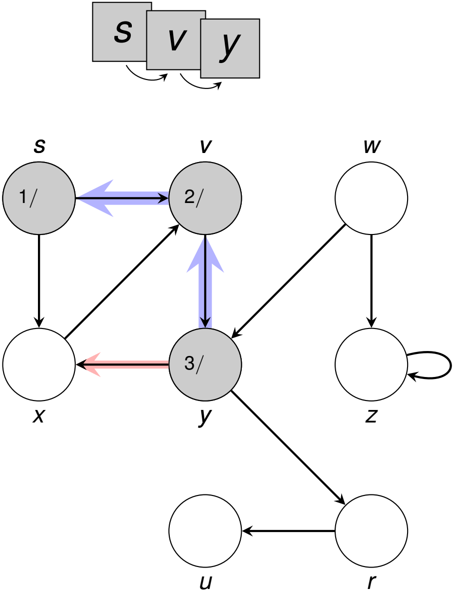


Execution of DFS

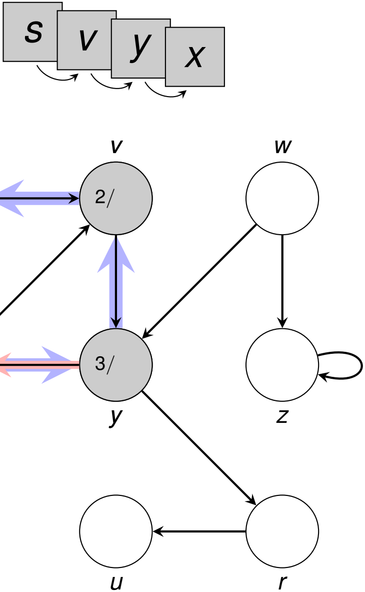




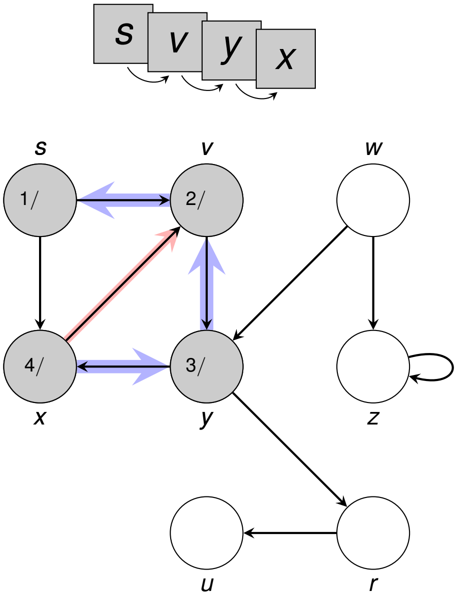
Execution of DFS



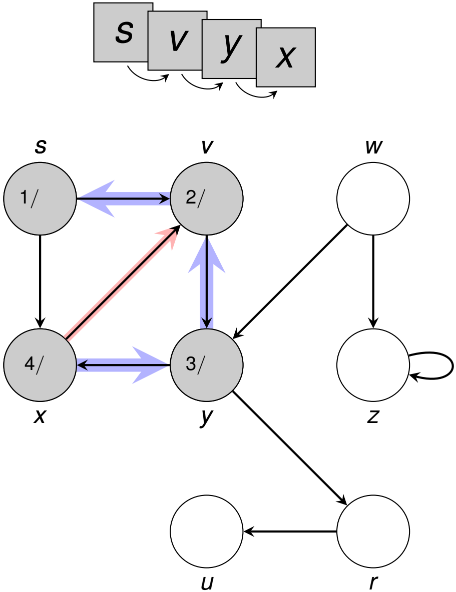
Execution of DFS



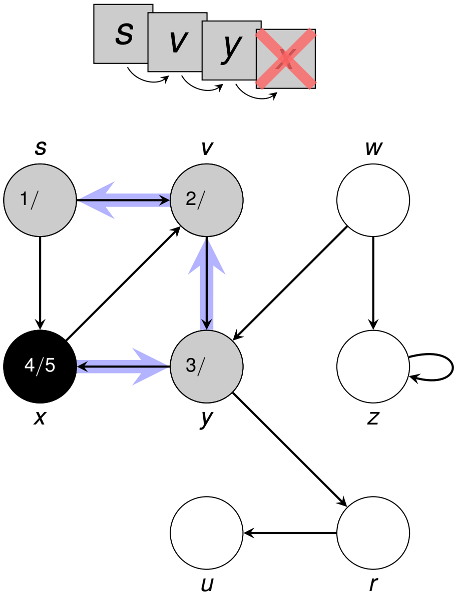
Execution of DFS



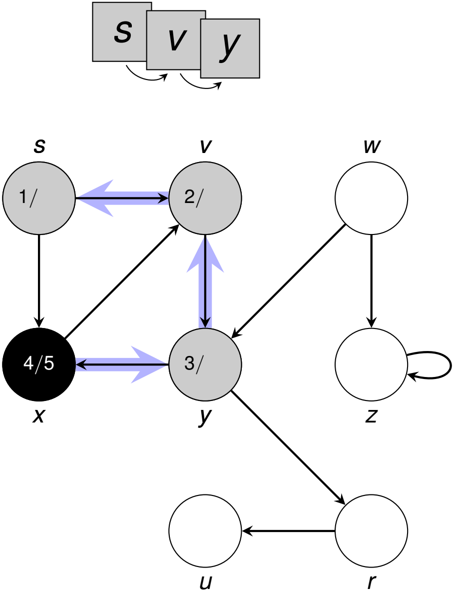
Execution of DFS



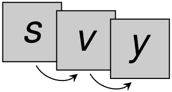
Execution of DFS

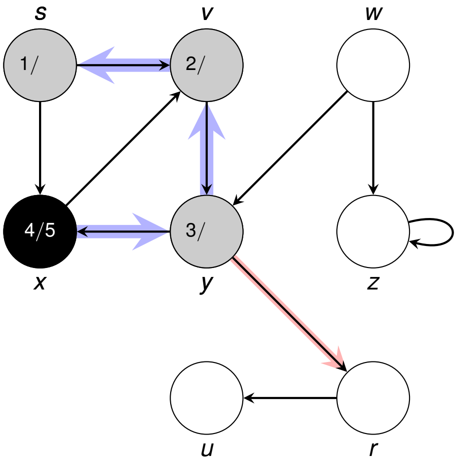


Execution of DFS

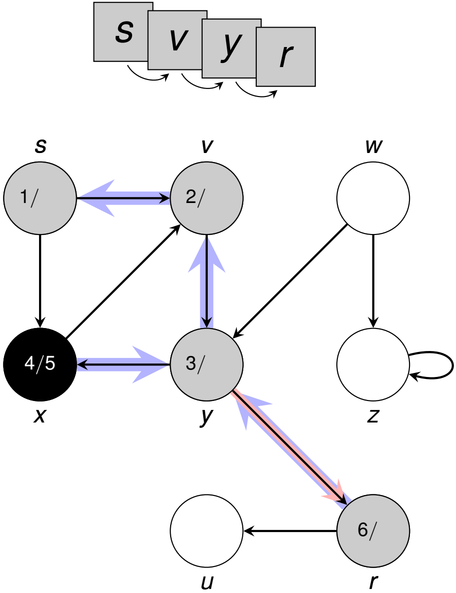


Execution of DFS

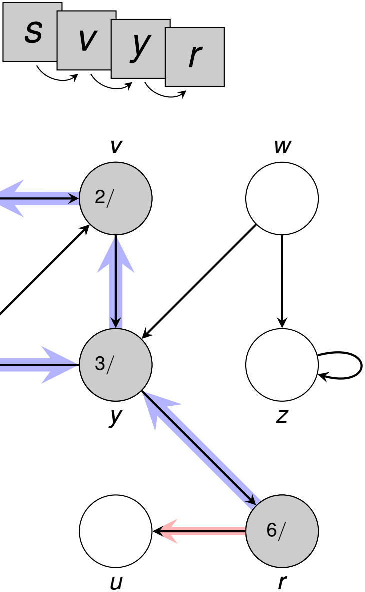




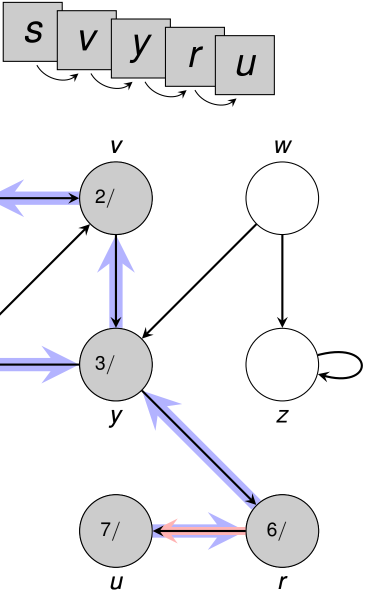
Execution of DFS



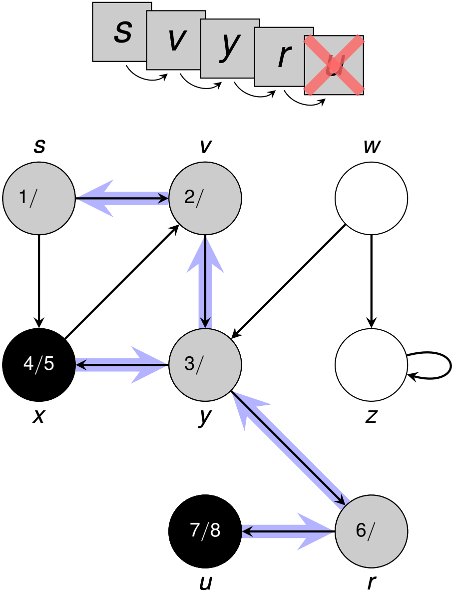
Execution of DFS



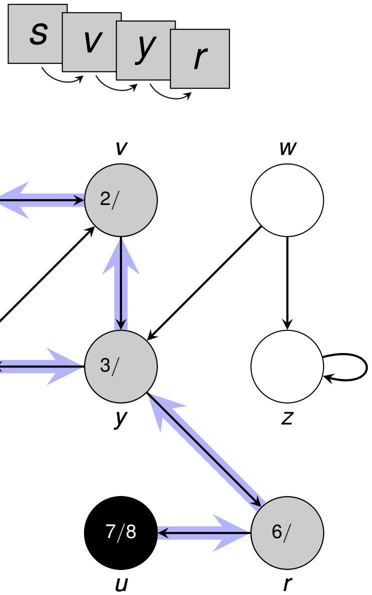
Execution of DFS



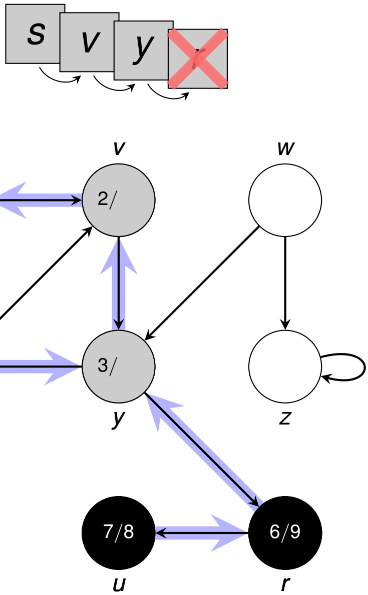
Execution of DFS



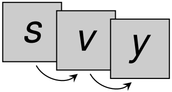
Execution of DFS

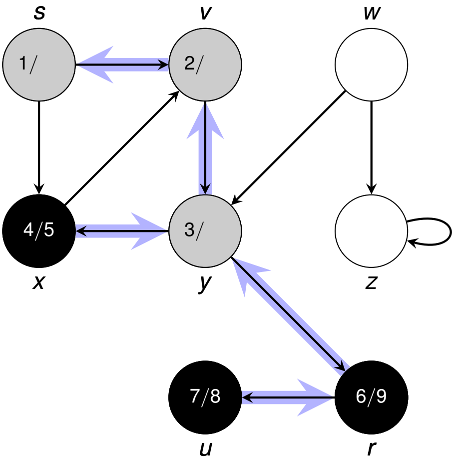


Execution of DFS

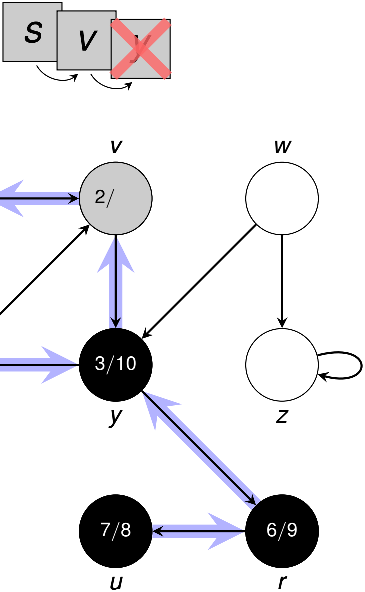


Execution of DFS

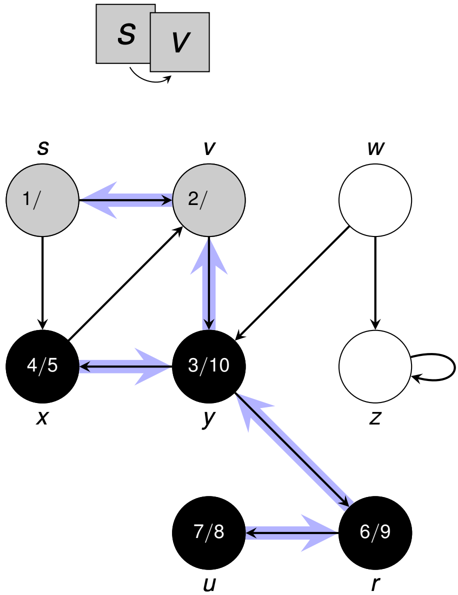




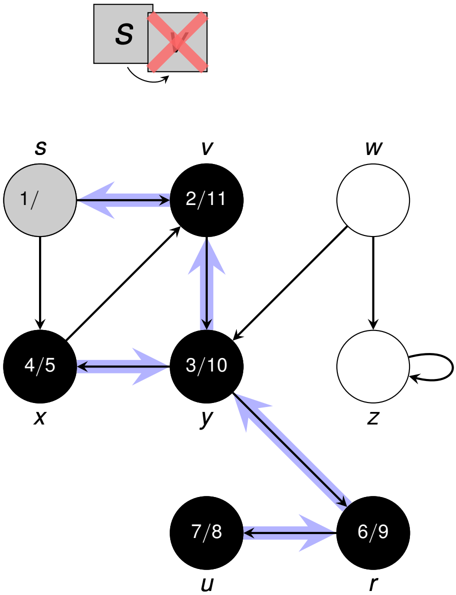
Execution of DFS



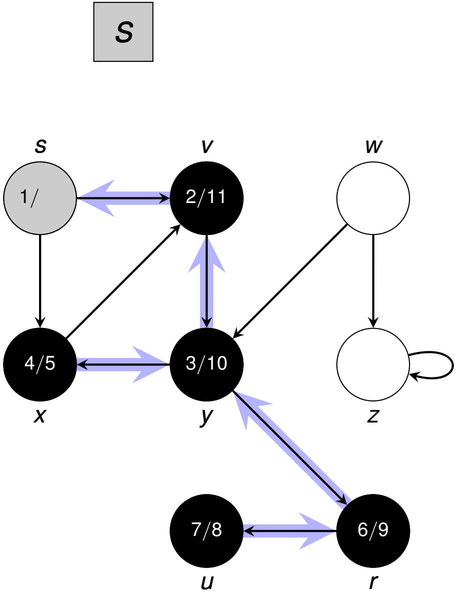
Execution of DFS



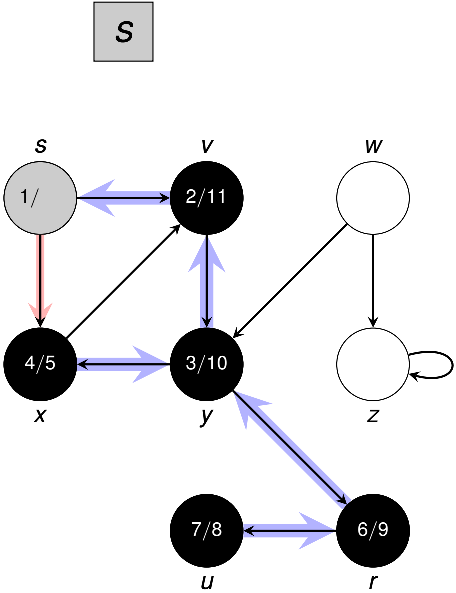
Execution of DFS



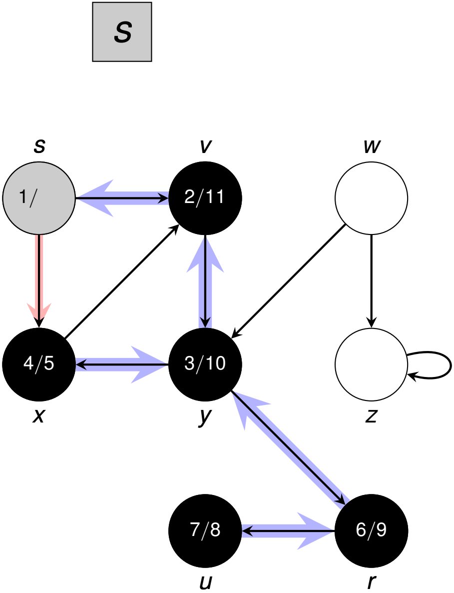
Execution of DFS



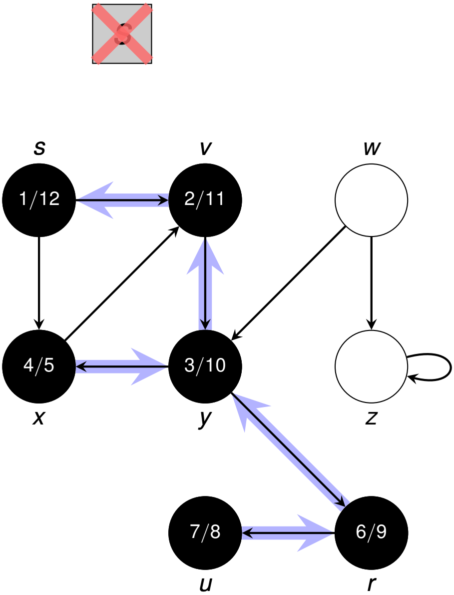
Execution of DFS



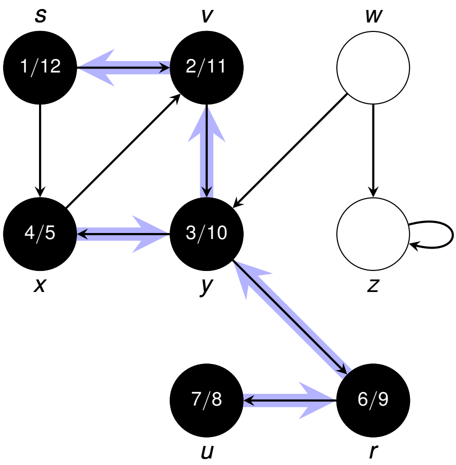
Execution of DFS



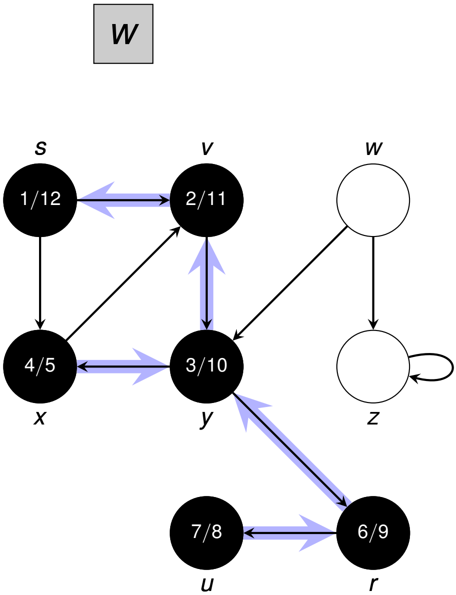
Execution of DFS



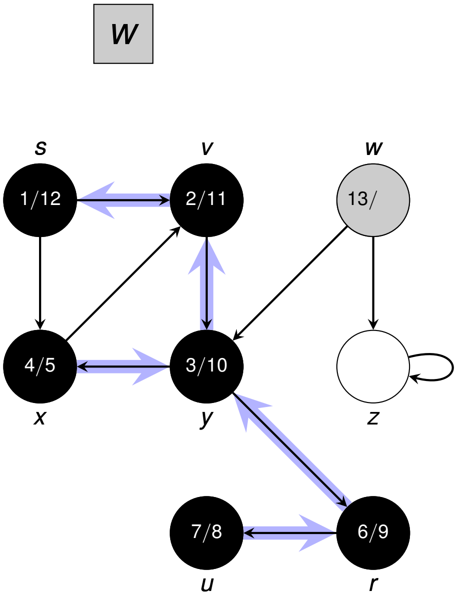
Execution of DFS



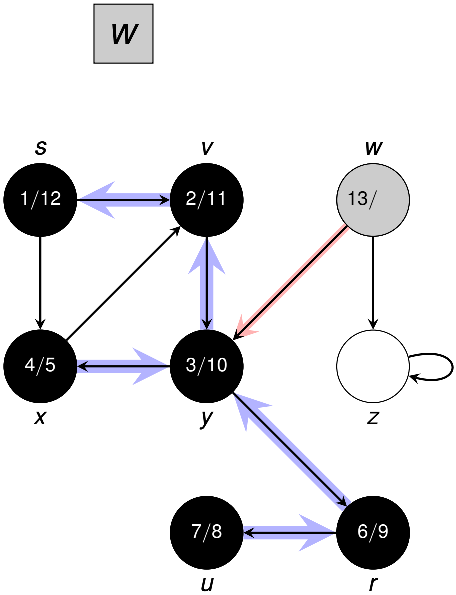
Execution of DFS



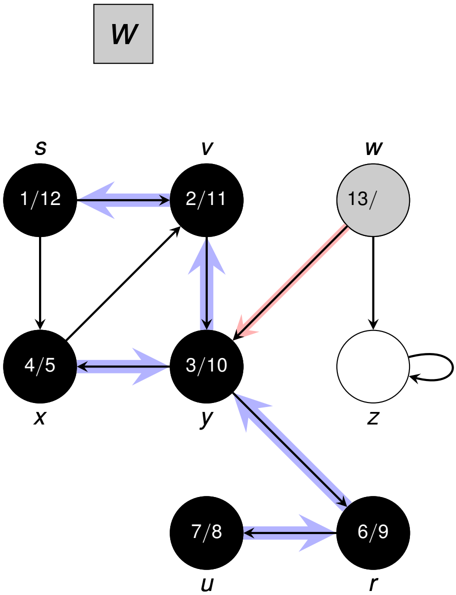
Execution of DFS



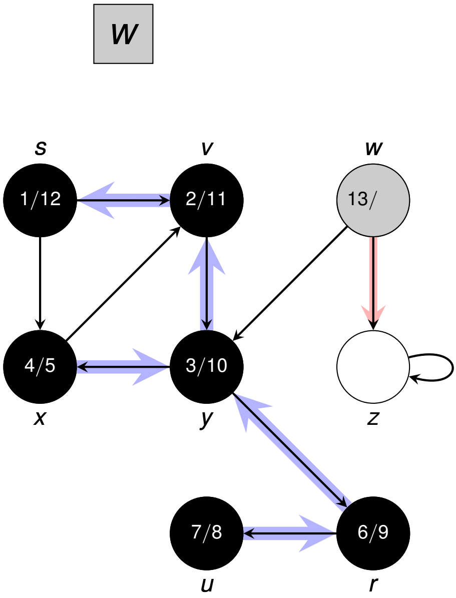
Execution of DFS



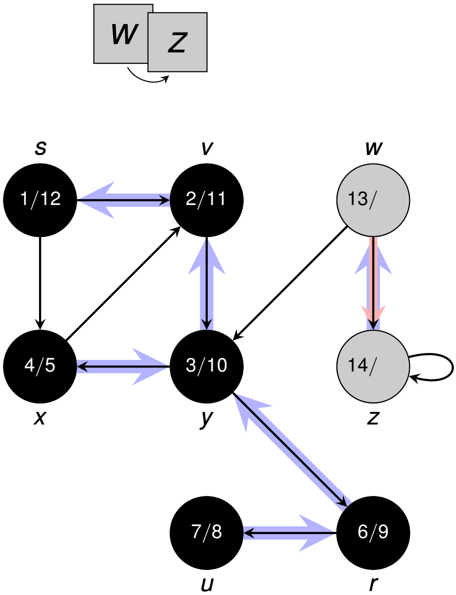
Execution of DFS



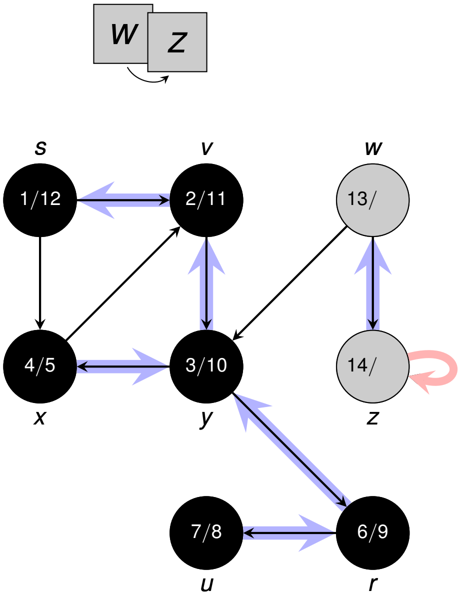
Execution of DFS



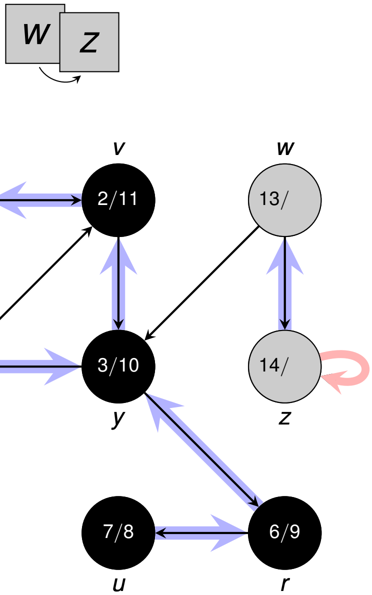
Execution of DFS



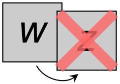
Execution of DFS

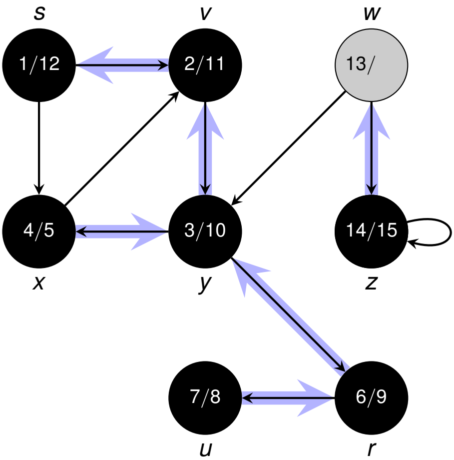


Execution of DFS

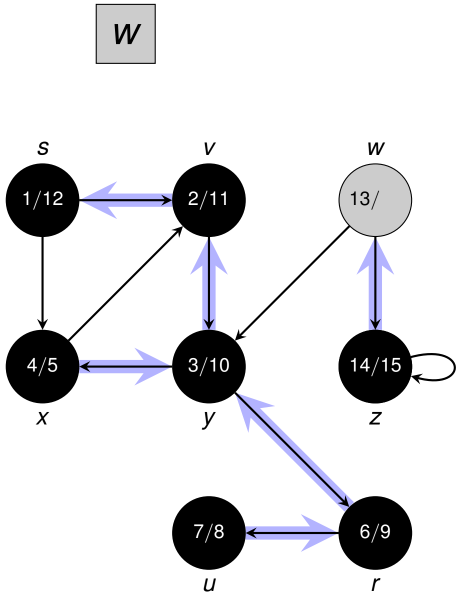


Execution of DFS

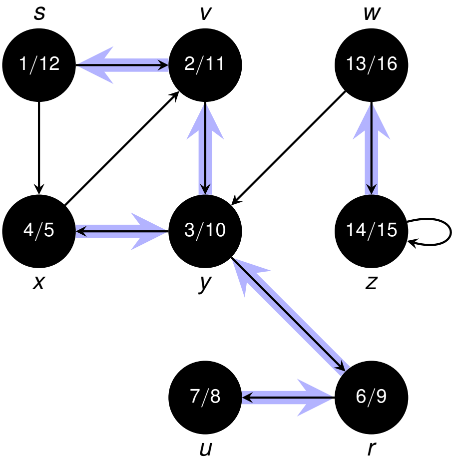




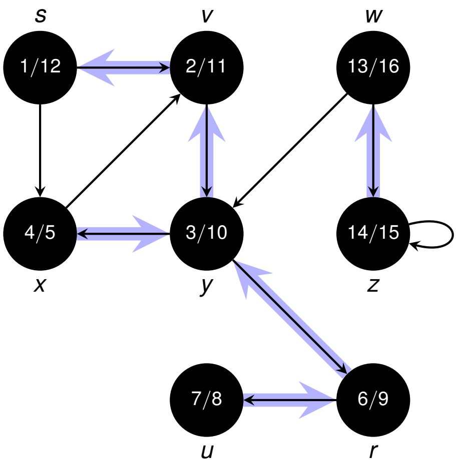
Execution of DFS



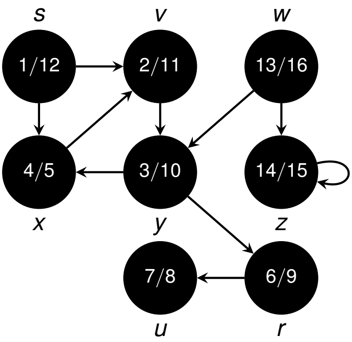
Execution of DFS

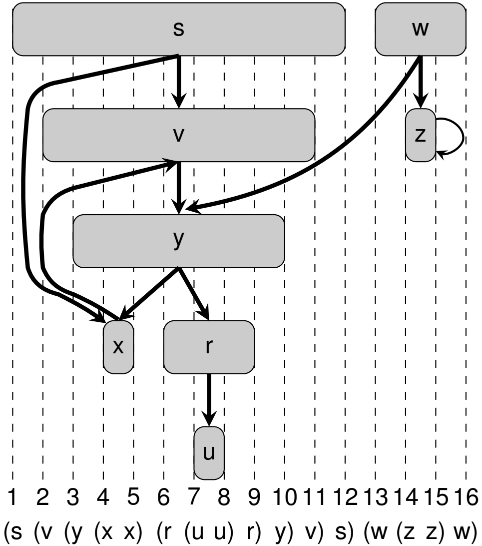


Execution of DFS



Paranthesis Theorem (Theorem 22.7)





Outline

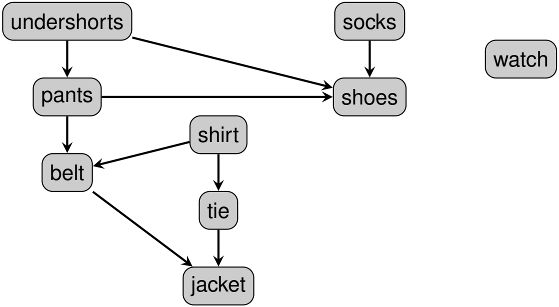
Introduction to Graphs and Graph Searching

Breadth-First Search

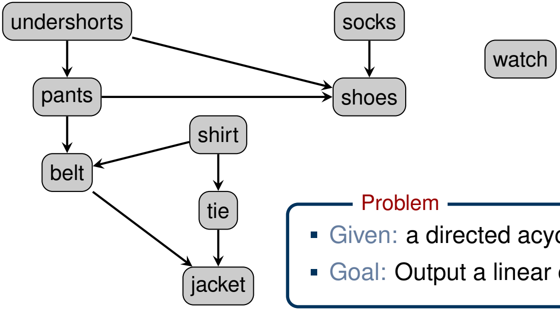
Depth-First Search

Topological Sort

Topological Sort



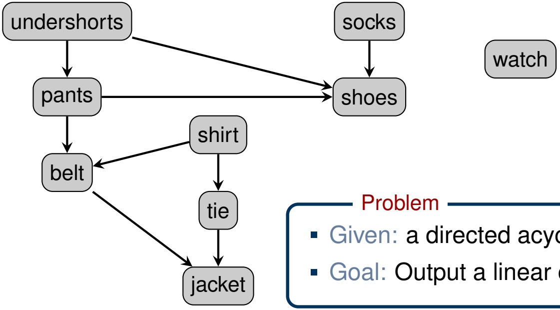
Topological Sort



Problem

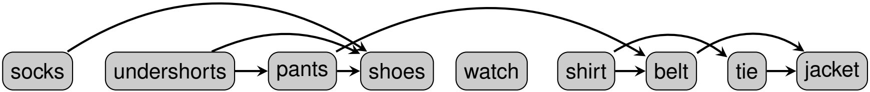
- Given: a directed acyclic graph (DAG)
- Goal: Output a linear ordering of all vertices

Topological Sort

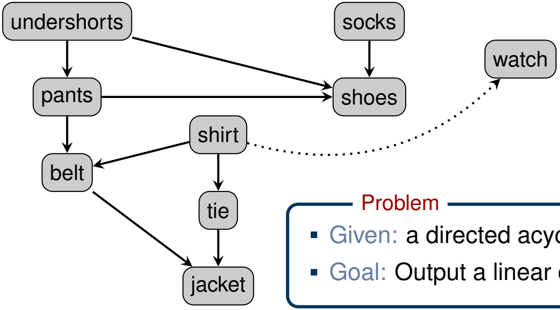


Problem

- Given: a directed acyclic graph (DAG)
- Goal: Output a linear ordering of all vertices

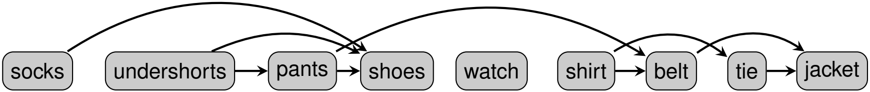


Topological Sort

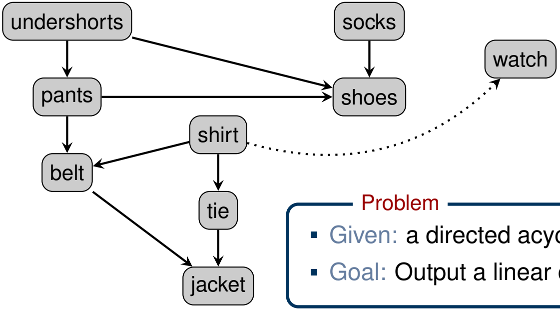


Problem

- Given: a directed acyclic graph (DAG)
- Goal: Output a linear ordering of all vertices

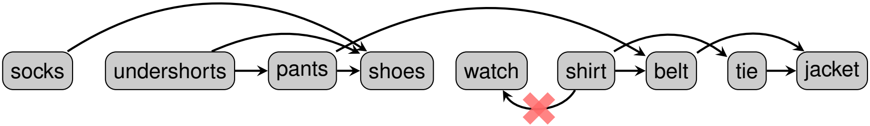


Topological Sort

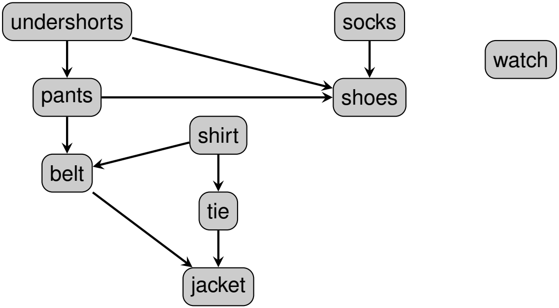


Problem

- Given: a directed acyclic graph (DAG)
- Goal: Output a linear ordering of all vertices



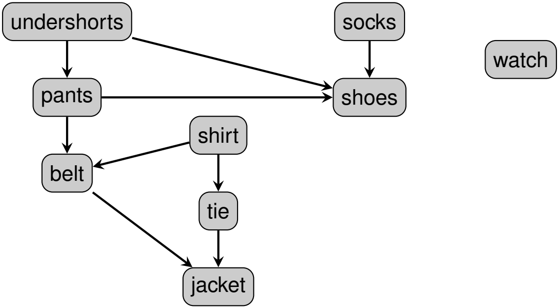
Solving Topological Sort



Knuth's Algorithm (1968)

- Perform DFS's so that all vertices are visited
- Output vertices in decreasing order of their finishing time

Solving Topological Sort

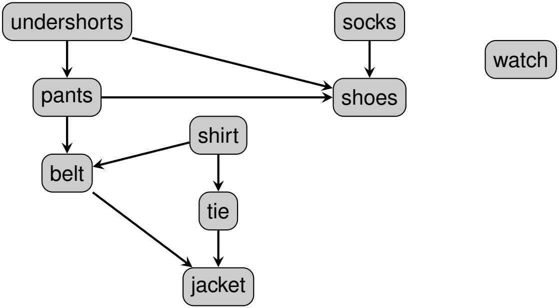


Knuth's Algorithm (1968)

- Perform DFS's so that all vertices are visited
- Output vertices in decreasing order of their finishing time

Runtime $O(V + E)$

Solving Topological Sort



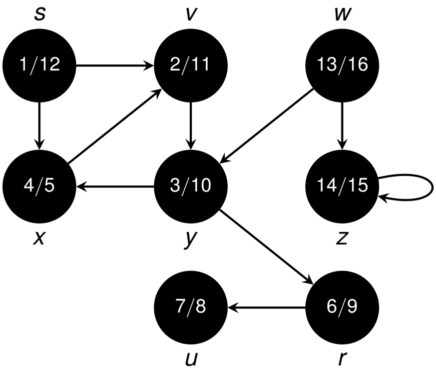
Knuth's Algorithm (1968)

- Perform DFS's so that all vertices are visited
- Output vertices in decreasing order of their finishing time

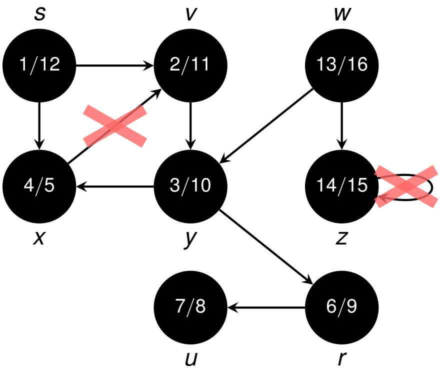
Runtime $O(V + E)$

Don't need to sort the vertices – use DFS directly!

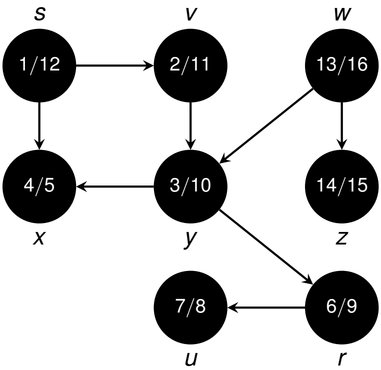
Execution of Knuth's Algorithm



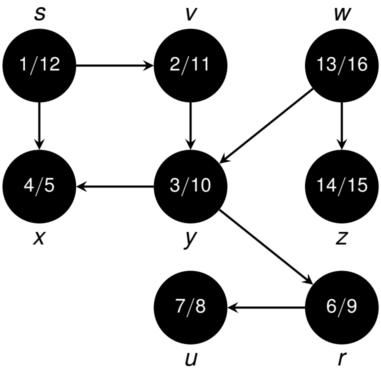
Execution of Knuth's Algorithm



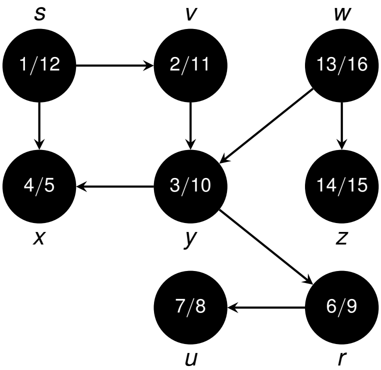
Execution of Knuth's Algorithm



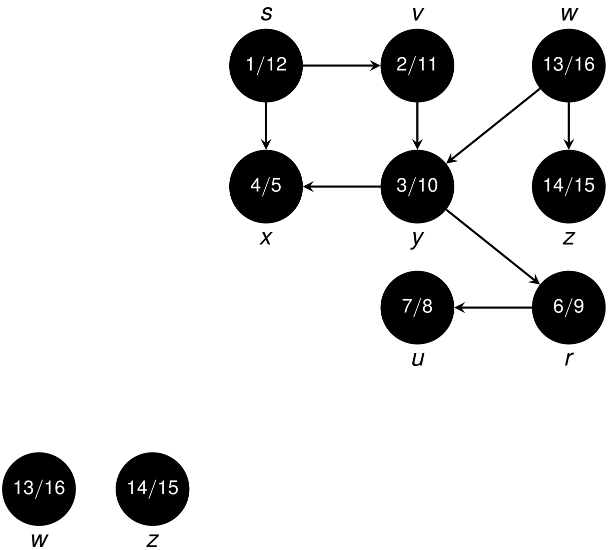
Execution of Knuth's Algorithm



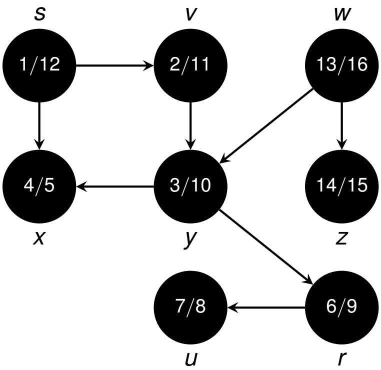
Execution of Knuth's Algorithm

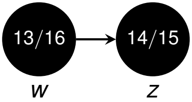


Execution of Knuth's Algorithm

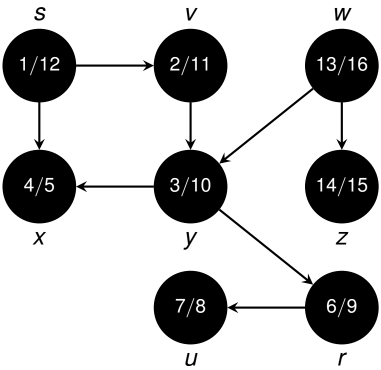


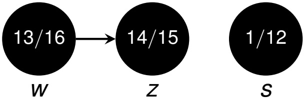
Execution of Knuth's Algorithm



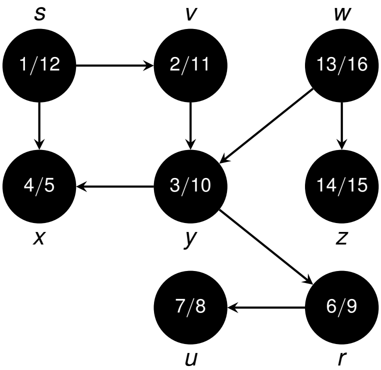


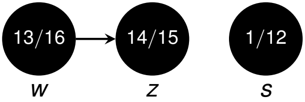
Execution of Knuth's Algorithm



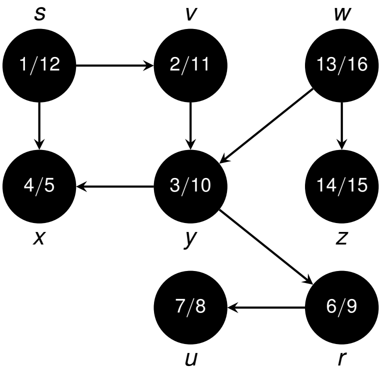


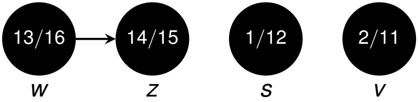
Execution of Knuth's Algorithm



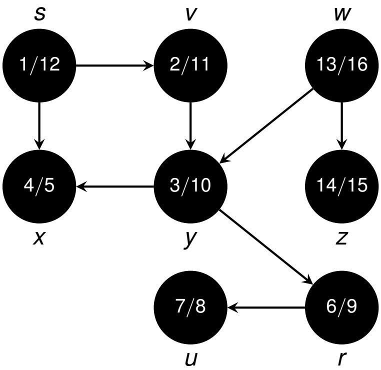


Execution of Knuth's Algorithm

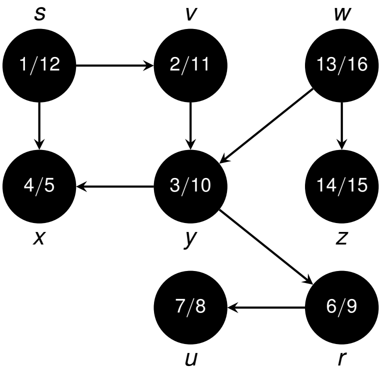


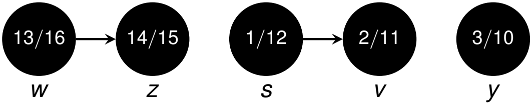


Execution of Knuth's Algorithm

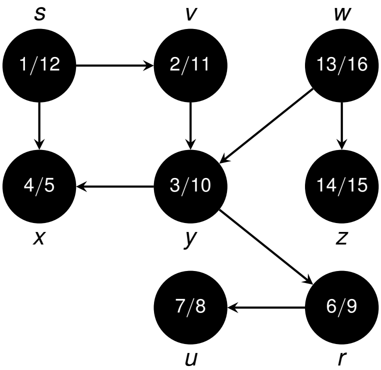


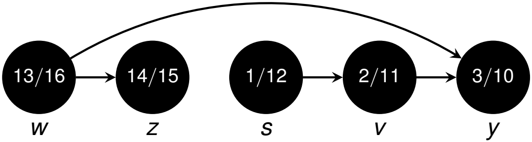
Execution of Knuth's Algorithm



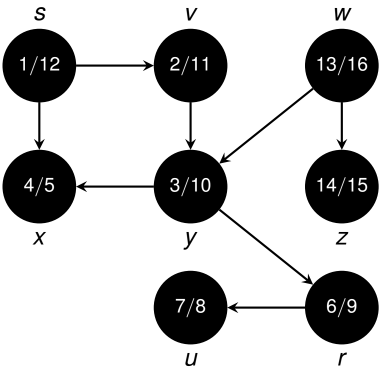


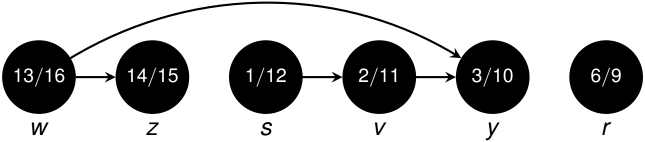
Execution of Knuth's Algorithm



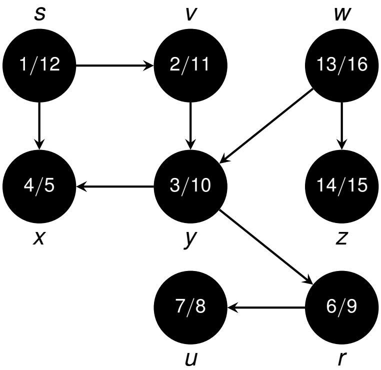


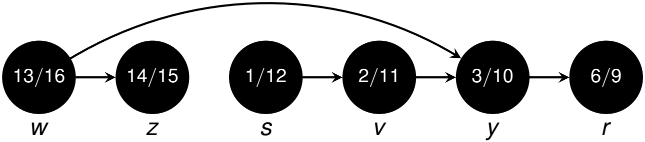
Execution of Knuth's Algorithm



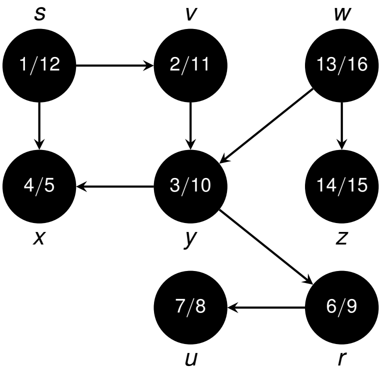


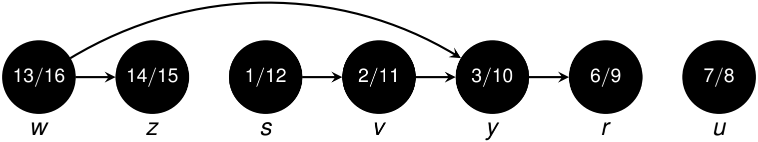
Execution of Knuth's Algorithm



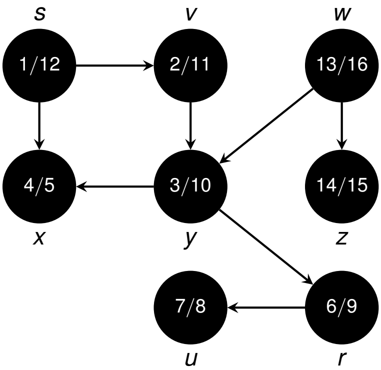


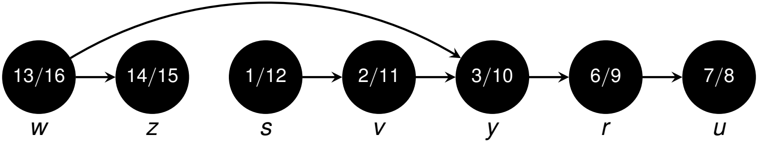
Execution of Knuth's Algorithm



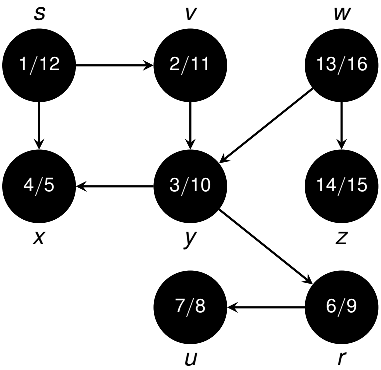


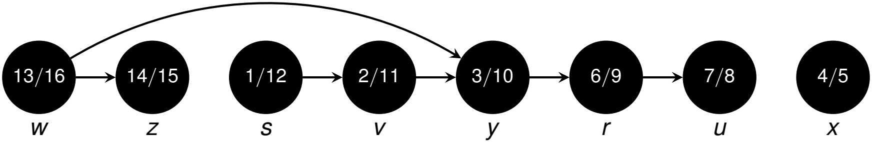
Execution of Knuth's Algorithm



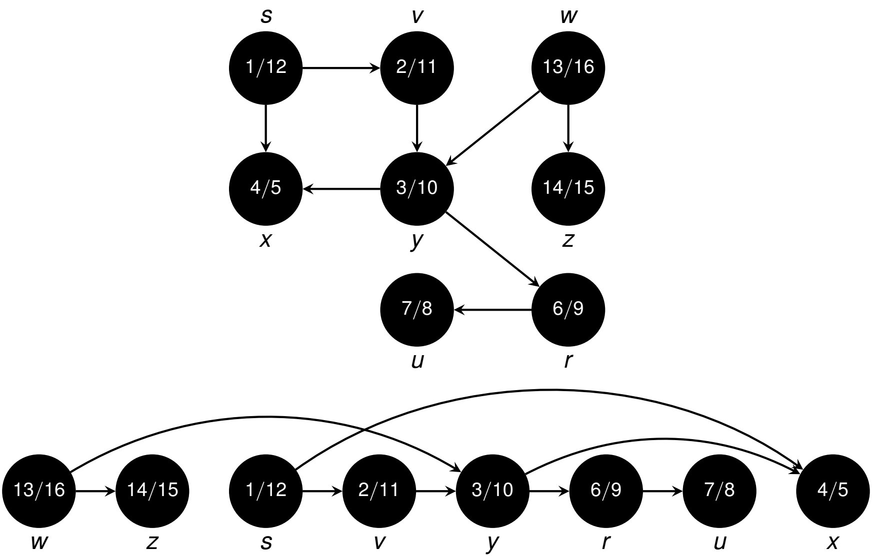


Execution of Knuth's Algorithm

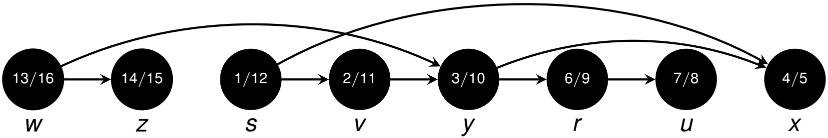




Execution of Knuth's Algorithm



Correctness of Topological Sort using DFS



Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

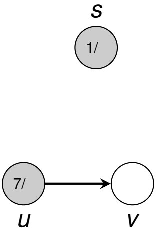
Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,



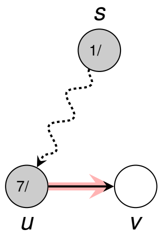
Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$



Correctness of Topological Sort using DFS

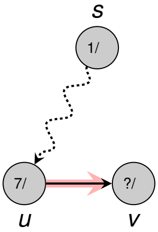
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey,*



Correctness of Topological Sort using DFS

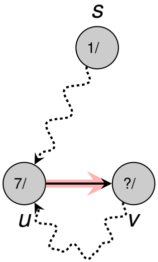
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey,*



Correctness of Topological Sort using DFS

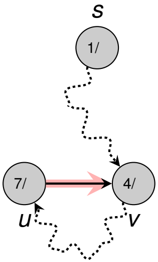
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).



Correctness of Topological Sort using DFS

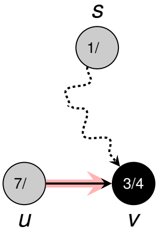
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow u$ is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black,*



Correctness of Topological Sort using DFS

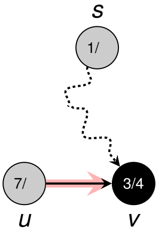
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow u$ is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*



Correctness of Topological Sort using DFS

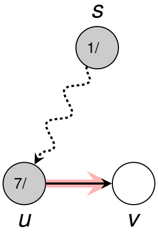
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white,*



Correctness of Topological Sort using DFS

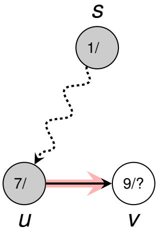
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white, we call DFS(v) and $v.f < u.f$.*



Correctness of Topological Sort using DFS

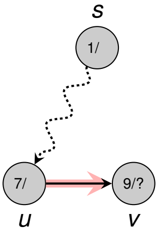
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white, we call DFS(v) and $v.f < u.f$.*



Correctness of Topological Sort using DFS

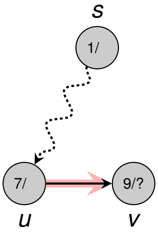
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white, we call DFS(v) and $v.f < u.f$.*



$\Rightarrow$ In all cases $v.f < u.f$, so v appears after u.

Correctness of Topological Sort using DFS

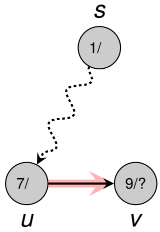
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white, we call DFS(v) and $v.f < u.f$.*

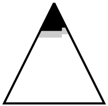


$\Rightarrow$ In all cases $v.f < u.f$, so v appears after u. $\square$

Summary of Graph Searching

Breadth-First-Search

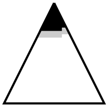
- vertices are processed by a queue
- computes distances and shortest paths
  $\rightsquigarrow$ similar idea used later in Prim's and Dijkstra's algorithm
- Runtime $\mathcal{O}(V + E)$



Summary of Graph Searching

Breadth-First-Search

- vertices are processed by a queue
- computes distances and shortest paths
  $\rightsquigarrow$ similar idea used later in Prim's and Dijkstra's algorithm
- Runtime $\mathcal{O}(V + E)$



Depth-First-Search

- vertices are processed by recursive calls ($\approx$ stack)
- discovery and finishing times
- application: Topogical Sorting of DAGs
- Runtime $\mathcal{O}(V + E)$