COMP 1201 Algorithmics

Graphs

Hikmat Farhat

Introduction

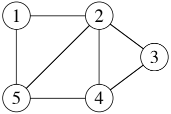
- **Note Most Figures are from Cormen et. al.**
- A **graph** $G = (V, E)$ is a set of vertices V and a set of edges E.
- Each element in E is a pair (v, w) with $v, w \in V$.
- If the pairs are **ordered** then the graph is **directed** (sometimes called **digraph**).
- if $(v, w) \in E$ then we say w is **adjacent** to v
- Usually we associate a **weight** (or **cost**) with each edge.
- A **path** is a sequence of vertices $w_1, \ldots, w_n$ such that $(w_i, w_{i+1}) \in E$.
- the **length** of a path is the number of edges in it

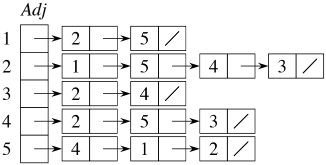
- A path is said to be **simple** if all vertices, except possibly the first and last, are **distinct**.
- A **cycle** is a path such that $w_1 = w_n$.
- in an undirected graph we require that the edges be distinct to have a cycle.
- for example v, w, v should not be considered a cycle since (v, w) and (w, v) are the same edge.
- A graph is said to be **acyclic** if it contains no cycles.
- A graph in which from every vertex there is path to every other vertex is called **connected**.

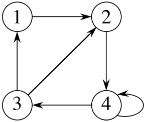
Graph representation

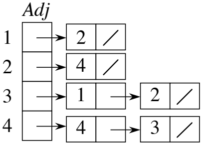
- There are essentially two ways to represent a graph
  - Adjacency matrix.
  - Adjacency list.
- Most of the time adjacency list is better since it is $O(|E| + |V|)$ in memory requirement.
- This is the preferred representation when the graph is sparse, $|E| \ll |V^2|$.
- The adjacency matrix is $O(|V^2|)$ in memory requirement and it is preferred when the graph is **dense**, $|E| \approx |V^2|$.
- In the adjacency matrix representation it is much faster to check whether two vertices are adjacent.

Examples









Matrix

	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

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

Breadth First Search

- As we will see later many algorithms depend on **breadth first search** (BFS).
- Given a graph $G = (V, E)$ and a **source** node s, BFS systematically "discovers" all vertices that can be reached from s.
- It is breadth first because all vertices at distance k from s are discovered **before** any vertex at distance $k + 1$ is discovered.
- BFS works by coloring nodes with two different colors: **white** and **black**.
- A **white** node means it has not been discovered. **Black** means it has been discovered.

Breadth First Search

- The algorithm starts by coloring all nodes white except the source s is colored black.
- It then proceed with the discovery of all of s neighbors.
- Given a node v
  - $v.d$ is the distance (number of links) from s to v.
  - $adj[v]$ is the list of v's neighbors.
  - $v.p$ is the predecessor of v in the path from s to v.

BFS Initialization

Given a graph G and source node s, BFS is initilised as follows:

Algorithm 1: BFS Initialization

function `BFSinit(G,s)`
  foreach $v \in V - \{s\}$ **do**
    $v.color \leftarrow WHITE$
    $v.d \leftarrow 0$
    $v.p \leftarrow NULL$
  $s.color \leftarrow BLACK$
  $s.d \leftarrow 0$
  $s.p \leftarrow NULL$
  $Q \leftarrow \emptyset$
  `ENQUEUE(Q,s)`

BFS Pseudo Code

Algorithm 2: BFS main algorithm

input: A graph $G = (V, E)$ and a source node s

Result: All nodes reachable from s are visited in BFS order.
$v.d$ is the number of hops from s to v

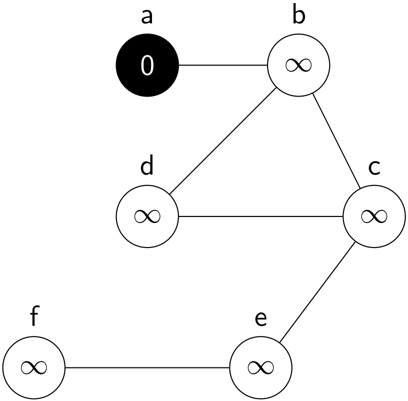
```
function BFS(G,s)
    BFSinit(G,s)
    while Q ≠ ∅ do
        u ← DEQUEUE(Q)
        foreach v ∈ Adj[u] do
            if v.color = WHITE then
                v.color ← BLACK
                v.d ← u.d + 1
                v.p ← u
                ENQUEUE(Q,v)
```

BFS Example

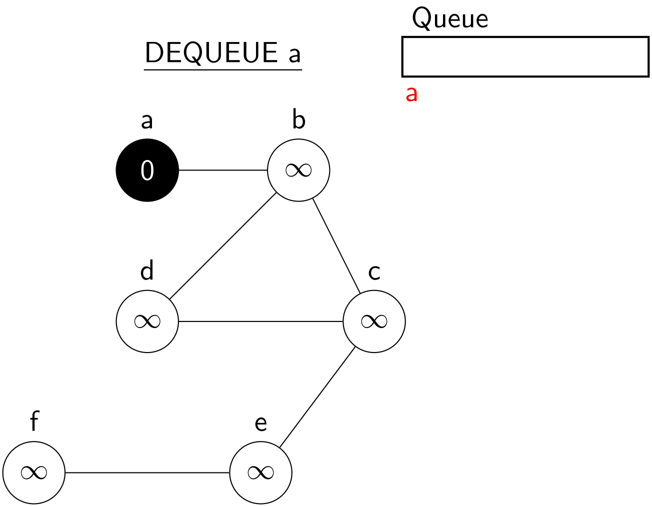
Queue

a

Initialise



BFS Example



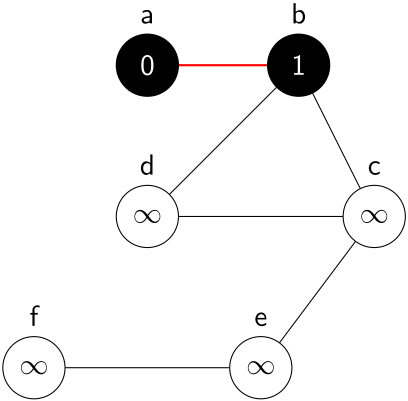
BFS Example

Visit neighbors of a

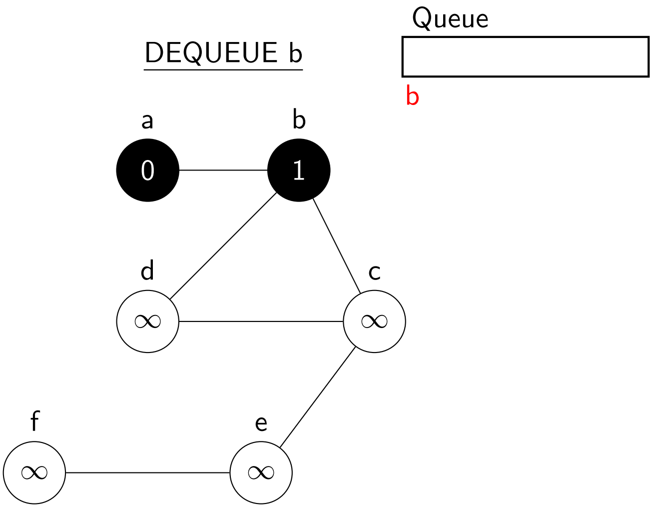
Queue

b

a



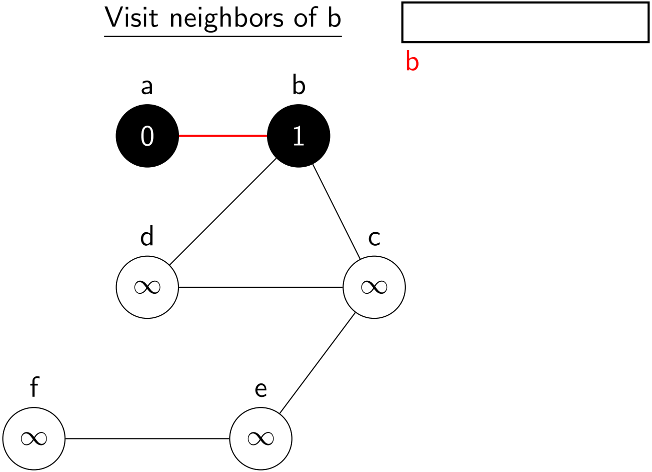
BFS Example



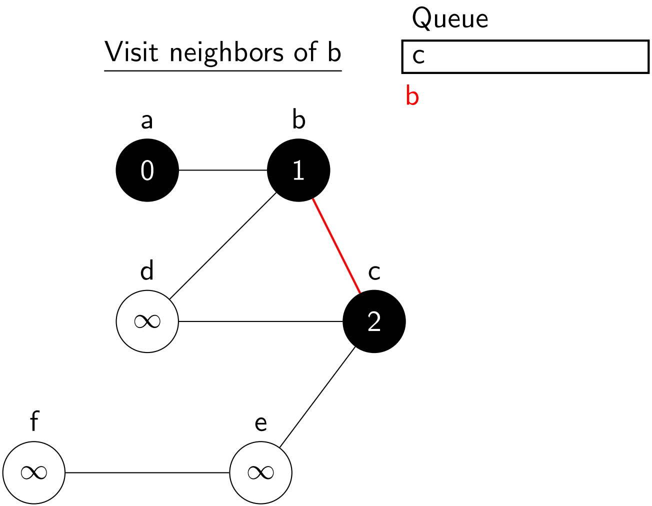
BFS Example

Visit neighbors of b

Queue



BFS Example



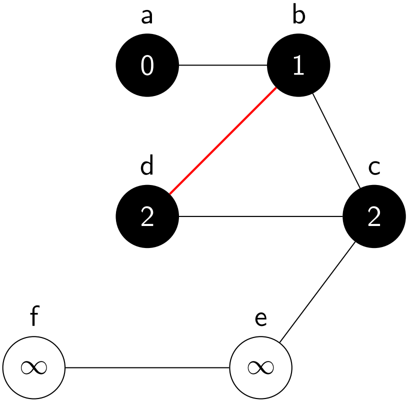
BFS Example

Visit neighbors of b

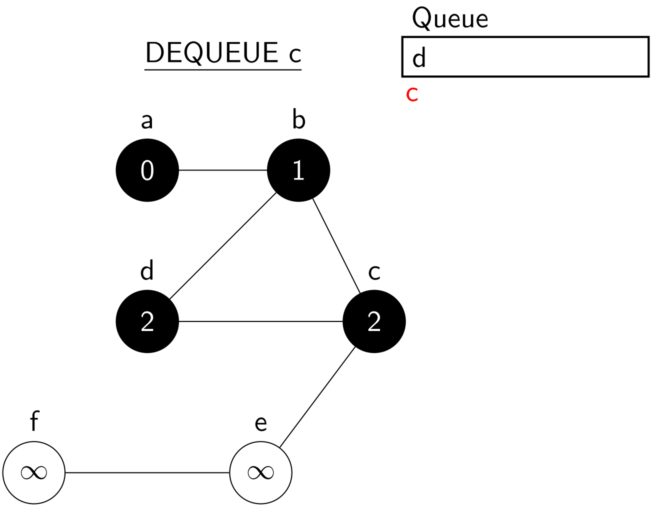
Queue

c d

b



BFS Example



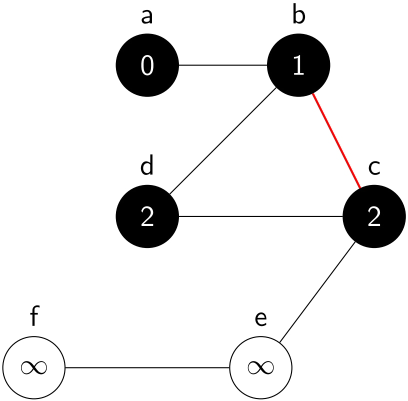
BFS Example

Visit neighbors of c

Queue

d

c



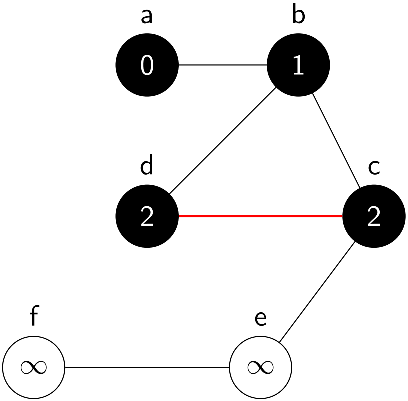
BFS Example

Visit neighbors of c

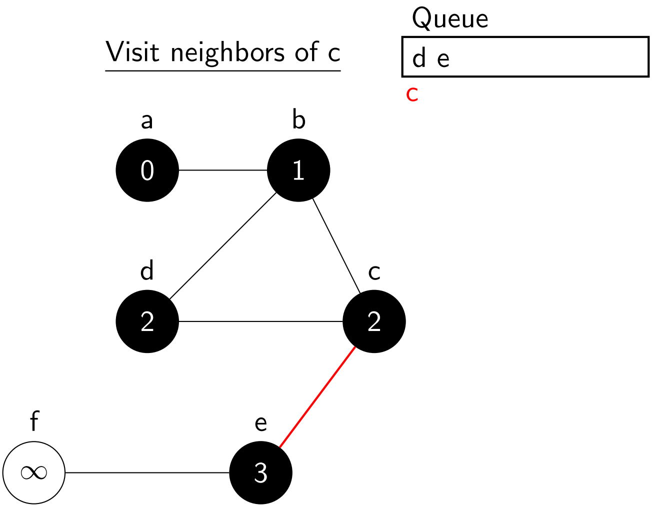
Queue

d

c



BFS Example



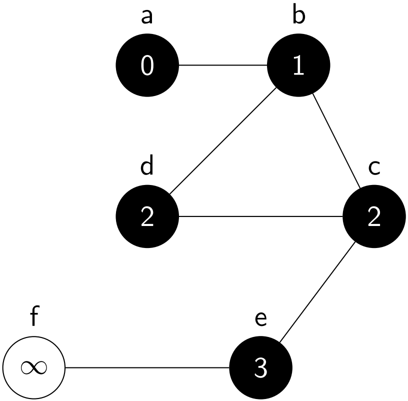
BFS Example

DEQUEUE d

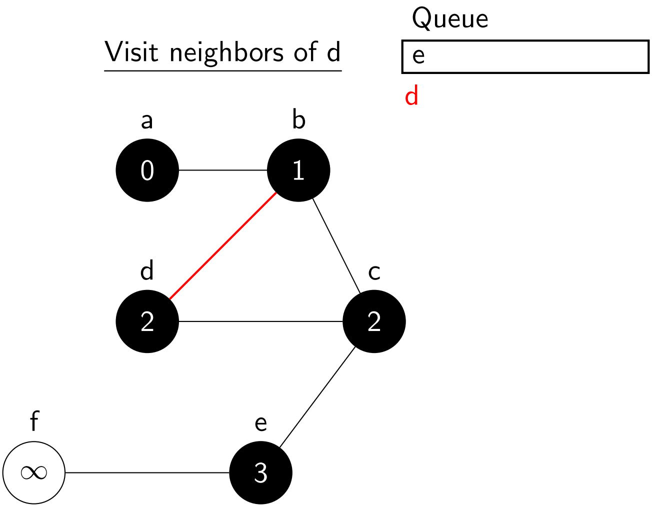
Queue

e

d



BFS Example



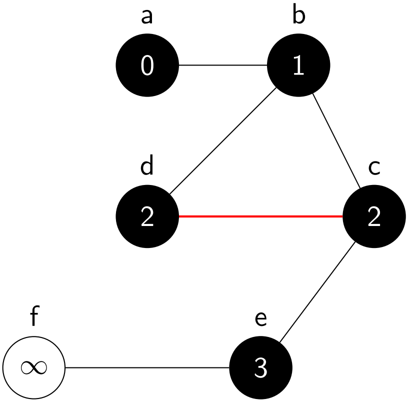
BFS Example

Visit neighbors of d

Queue

e

d

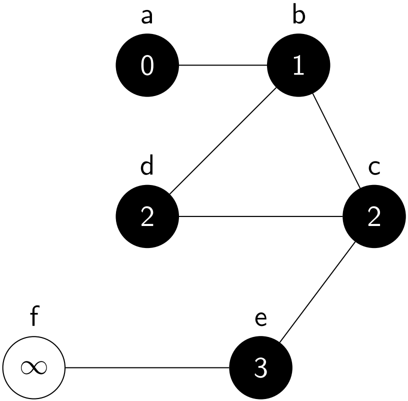


BFS Example

DEQUEUE e

Queue

e

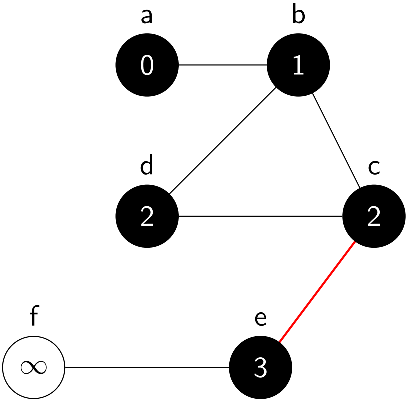


BFS Example

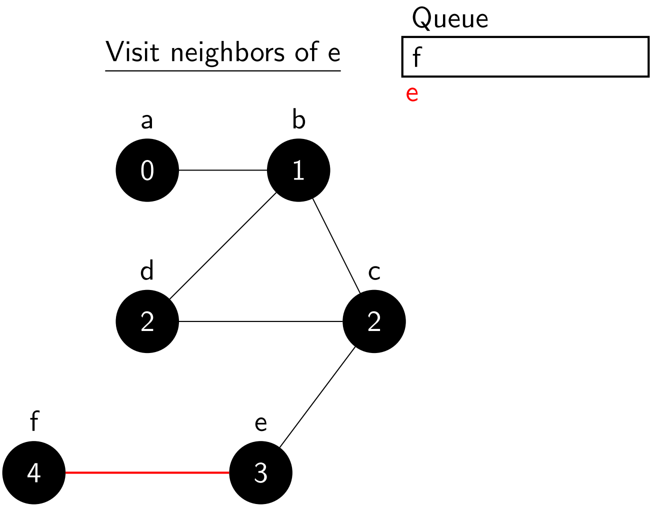
Visit neighbors of e

Queue

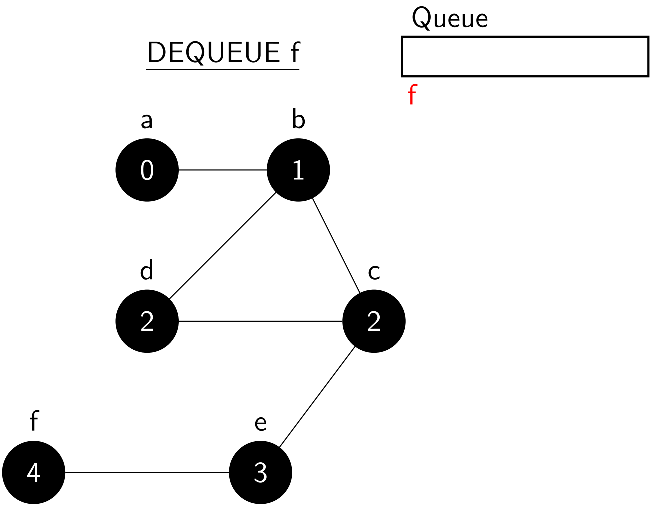
e



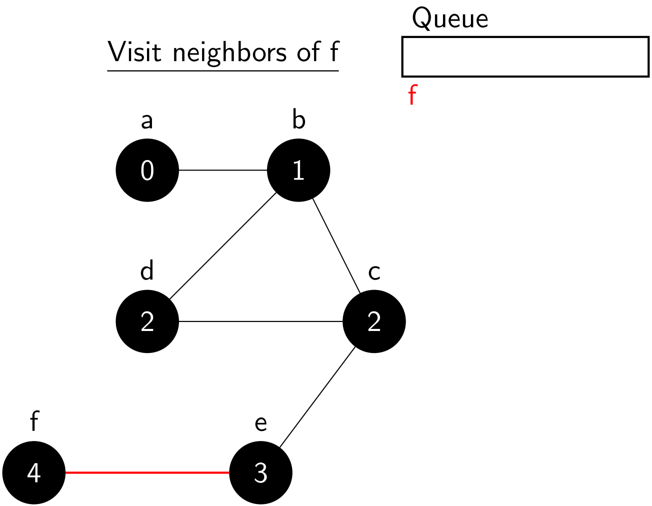
BFS Example



BFS Example



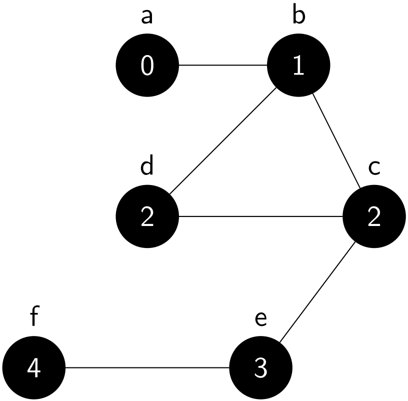
BFS Example



BFS Example

Queue is empty

Queue



Complexity of BFS

- To analyze the complexity of BFS first we note that after initialization no vertex color is set to white.
- The above implies that each vertex is enqueued (and dequeued) only once.
- Since the enqueue/dequeue operations are $O(1)$ then for all nodes it is $O(|V|)$.
- When a vertex is dequeued we scan the adjacency list and the sum of all adjacency list is just $|E|$
- Therefore the total cost of BFS is $O(|V| + |E|)$.

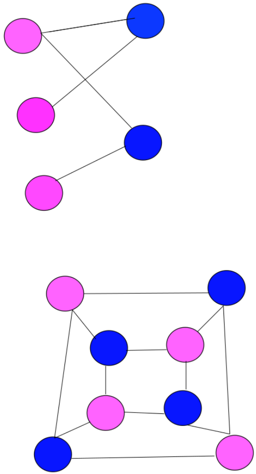
Shortest path length (number of hops)

- Given a graph $G = (V, E)$ and a source node $s \in V$. We define the **shortest-length** distance $\delta(s, v)$ from s to $v \in V$ to be the minimum **number of edges** in any path from s to v.
- When BFS terminates, it discovers every vertex $v \in V$ reachable from a source s and $v.d = \delta(s, v)$,
- Note that the shortest length path from s to v is **composed** of the shortest length path from s to $v.p$ **followed** by the edge $(v.p, v)$.
- The above observation allows us to determine not only the length $\delta(s, v)$ but also the exact path by iterating backwards over $v.p$.

Bipartite Graphs

- A graph $G = \langle V, E \rangle$ is a bipartite graph if V can be partitioned into two sets V_1 and V_2 such that
- $(u, v) \in E \Rightarrow u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$.
- A bipartite graph can be 2-colored. In other words,
- Using only two colors, one can assign a color to each vertex
- such that no two adjacent vertices have the same color.

Example bipartite graphs



Detecting bipartite graphs using BFS

- One can detect if a graph is bipartite using a small modification of BFS
- Instead of coloring a node black when it is "discovered" we either color it red or blue.
- The algorithm starts by coloring the source node red and all other nodes white exactly as it was done in BFS.
- While discovering the neighbors of a node u we color them with the opposite color of u.
- Let $\neg Blue = Red$ and $\neg Red = Blue$.

Detecting bipartite graphs using BFS

function `IsBipartite(G,s)`
  while $Q \neq \emptyset$ **do**
    $u \leftarrow$ `DEQUEUE(Q)`
    foreach $v \in Adj[u]$ **do**
      if $v.color = WHITE$ **then**
        $v.color \leftarrow \neg u.color$
        $v.d \leftarrow u.d + 1$
        $v.p \leftarrow u$
        `ENQUEUE(Q,v)`
      else if $u.color = v.color$ **then**
        return *false*
  return *true*

Depth First Search

- In a **depth first search** DFS edges are explored out of the most recently discovered node.
- As the name implies we go "deeper" whenever it is possible.
- When all the neighbors of a node v are discovered we "backtrack" to the predecessor of v and explore other nodes.
- When we are done discovering the descendants of some source s and some nodes remain undiscovered then one of them is selected as source and the process is repeated.
- When the algorithm is done with a certain node, it records the **discovery time** and **finishing time**

DFS Pseudo Code

Algorithm 3: DFS Algorithm

input: $G = (V, E)$ with adjacency list *adj*
Result: All nodes are visited in DFS order

foreach $v \in V$ **do**
  $v.color \leftarrow WHITE$
  $v.p \leftarrow NULL$
$time \leftarrow 0$
foreach $v \in V$ **do**
  if $v.color = WHITE$ **then**
    DFS-VISIT(adj, v)

DFS-VISIT Pseudo Code

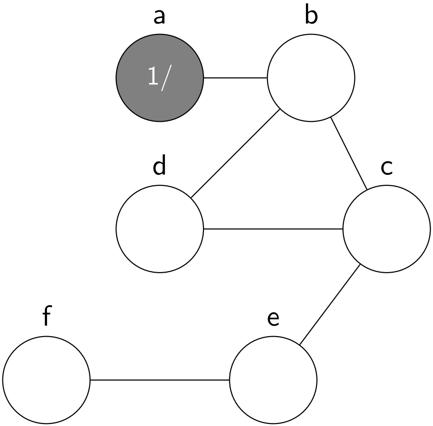
Algorithm 4: DFS-VISIT Algorithm

input: Adjacency list *adj*, node *u*

Result: All nodes reachable from *u* are visited in DFS order

```
function DFS-VISIT(adj, u)
    u.color ← GRAY
    time ← time + 1
    u.d ← time
    foreach v ∈ adj[u] do
        if v.color = WHITE then
            v.p ← u
            DFS-VISIT(v)
    u.color ← BLACK
    times ← time + 1
    u.f ← time
```

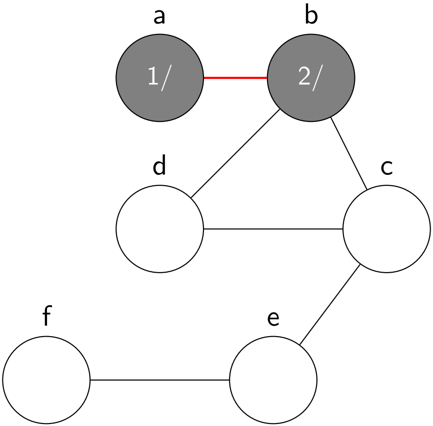
DFS Example



"Pending"

a

DFS Example

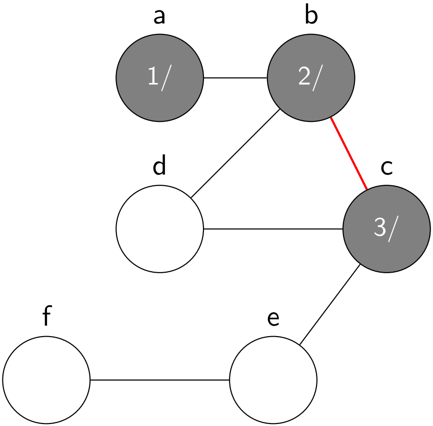


"Pending"

b

a

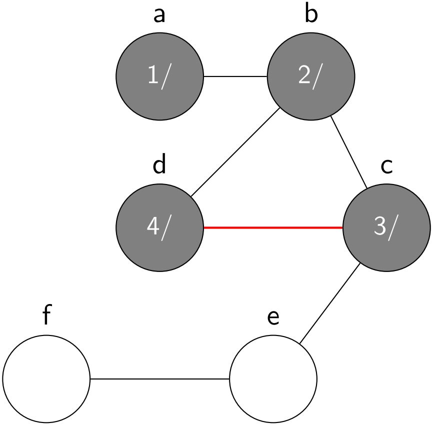
DFS Example



"Pending"

c
b
a

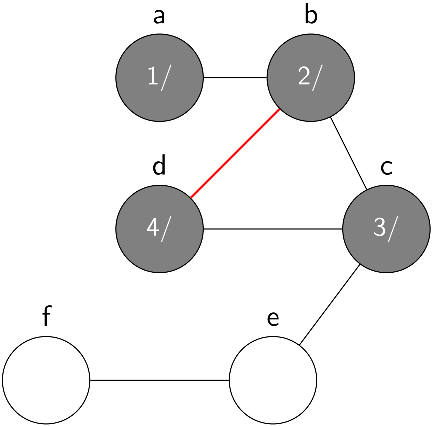
DFS Example



"Pending"

d
c
b
a

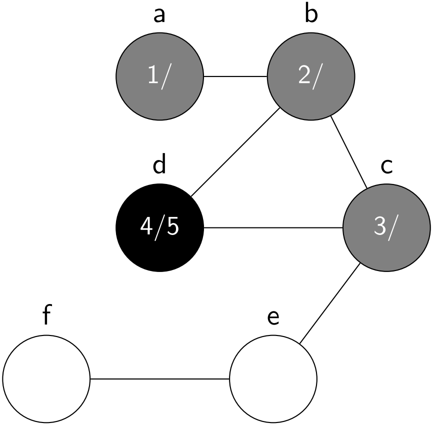
DFS Example



"Pending"

d
c
b
a

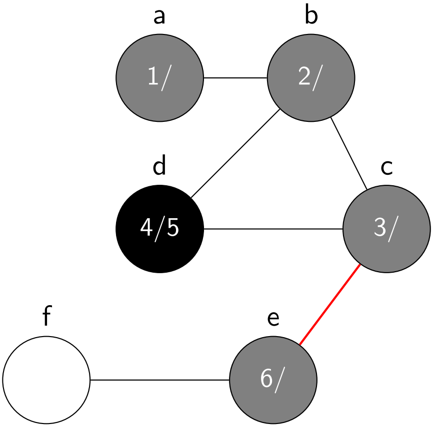
DFS Example



"Pending"

c
b
a

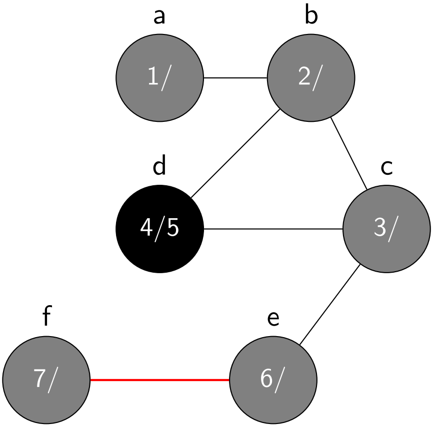
DFS Example



"Pending"

e
c
b
a

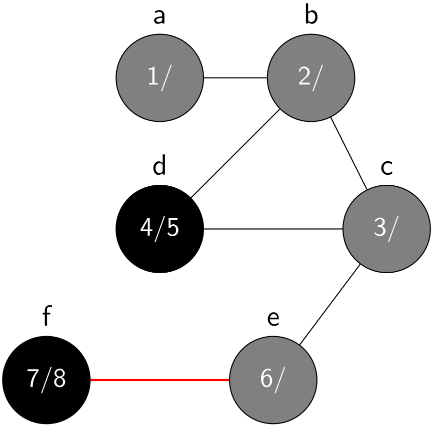
DFS Example



"Pending"

f
e
c
b
a

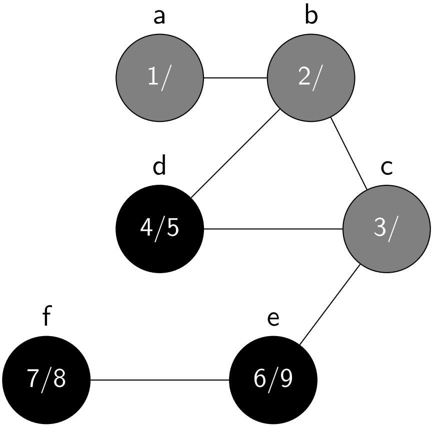
DFS Example



"Pending"

e
c
b
a

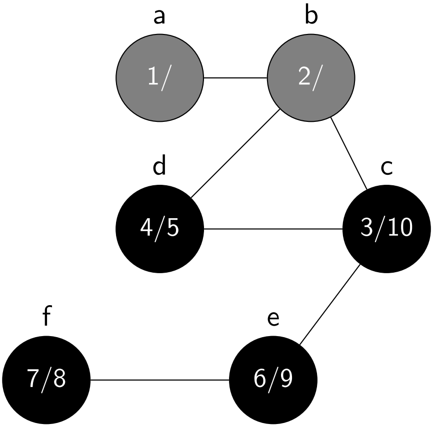
DFS Example



"Pending"

c
b
a

DFS Example

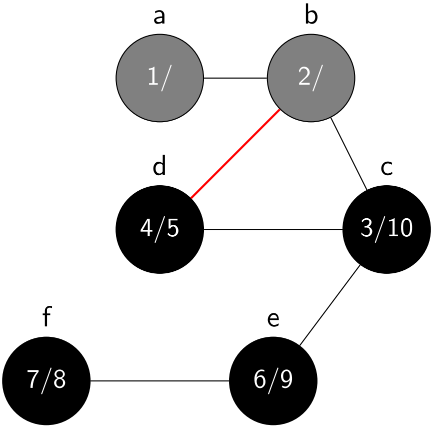


"Pending"

b

a

DFS Example

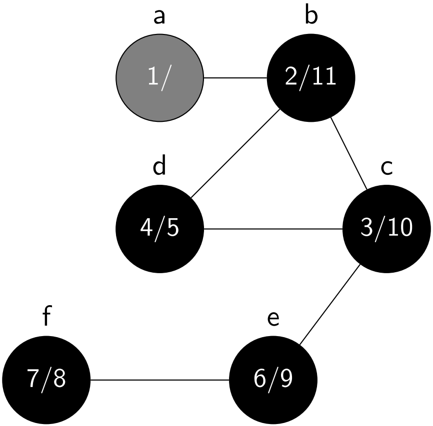


"Pending"

b

a

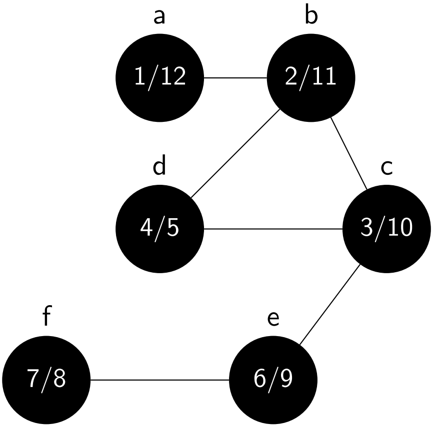
DFS Example



"Pending"

a

DFS Example



Complexity

- The initialization to WHITE is $O(|V|)$
- Then DFS is called $O(|V|)$ times.
- Each time DFS-VISIT is called **only once** for each node because it is called on WHITE nodes only.
- The cost of DFS-VISIT(v) is $O(|adj[v]|)$.
- Thus the cost of all calls to DFS-VISIT is

$$\sum_{v \in V} |adj[v]| = O(|E|)$$

- Therefore the total cost is

$$O(|E| + |V|)$$

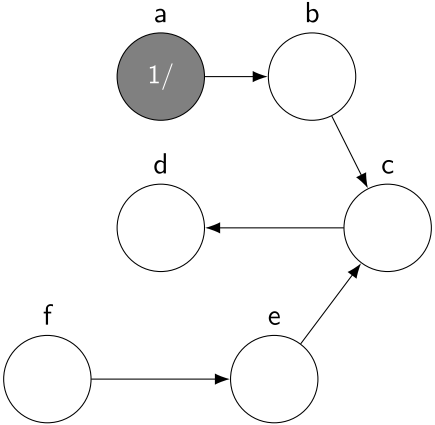
Topological Sorting (ordering)

- Topological sorting is a linear ordering of the nodes of a directed graph $G = (V, E)$ such that for every edge (u, v) from node u to node v, u comes before v in the ordering.
- An example is the scheduling of tasks based on their dependencies.
- Say a task p must be scheduled after a task q if p "depends on" q. This would be represented by an edge (q, p).
- We can implement an efficient topological sort using DFS as follows
  1. Call DFS on the graph.
  2. Every time a node is finished add it to the front of a linked list
  3. When done the resulting list is the topological sort.

DFS Topological Sort Example

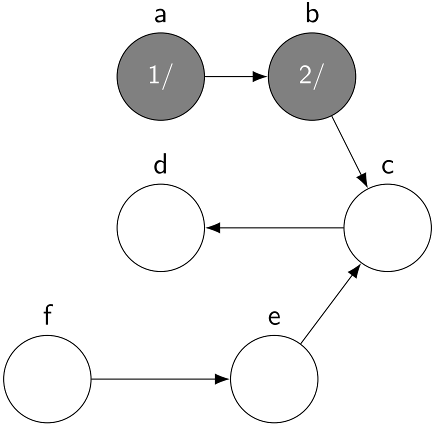
Topological Sort Example

"Linked List"



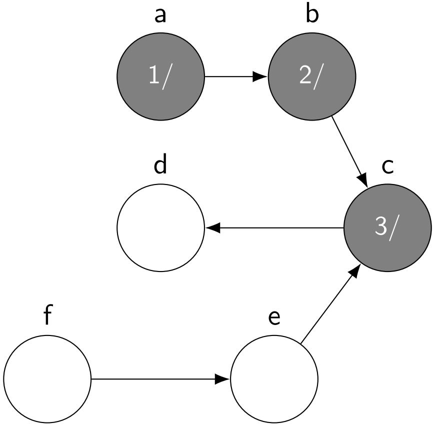
Topological Sort Example

"Linked List"



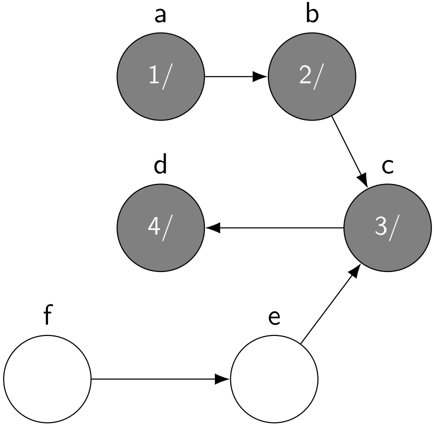
Topological Sort Example

"Linked List"

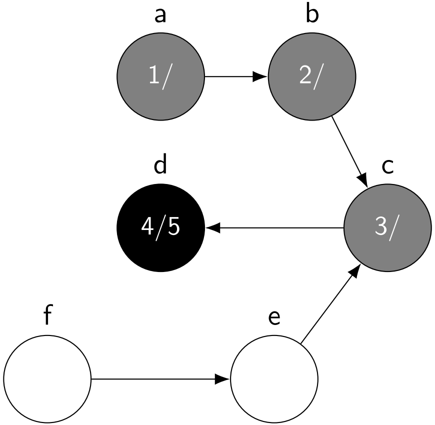


Topological Sort Example

"Linked List"



Topological Sort Example

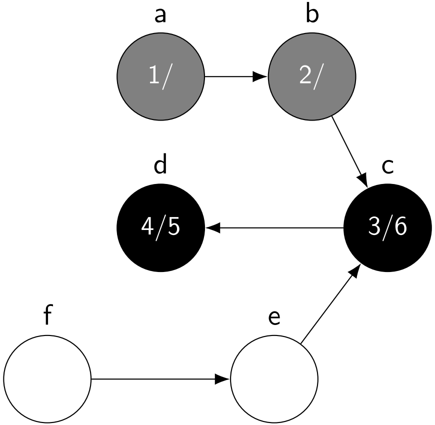


"Linked List"

d

Topological Sort Example

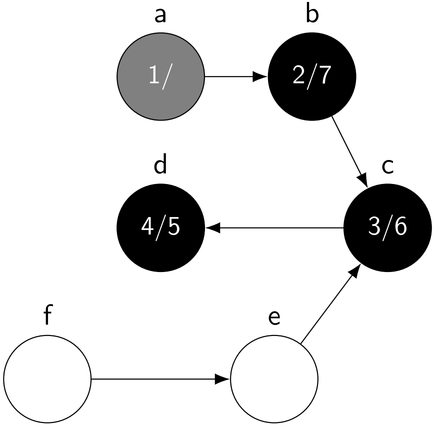
"Linked List"



c-d

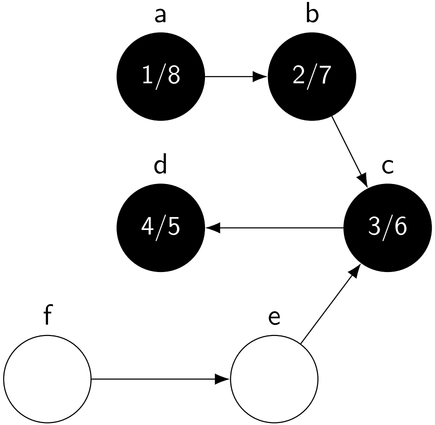
Topological Sort Example

"Linked List"



b-c-d

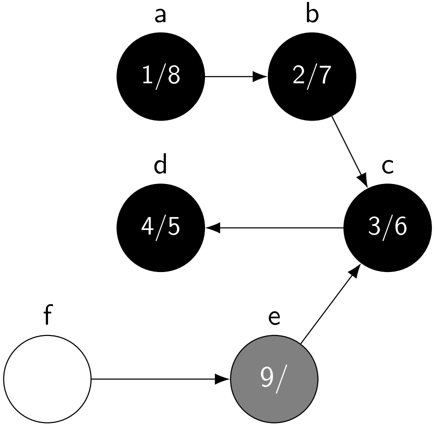
Topological Sort Example



"Linked List"

a-b-c-d

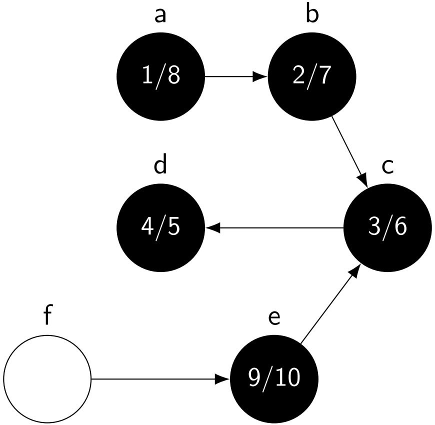
Topological Sort Example



"Linked List"

a-b-c-d

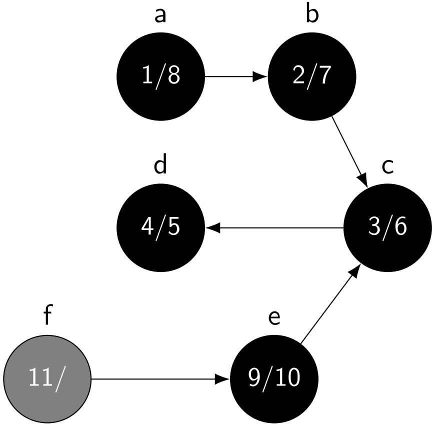
Topological Sort Example



"Linked List"

e-a-b-c-d

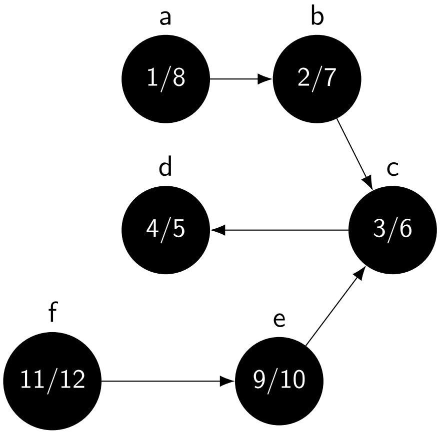
Topological Sort Example



"Linked List"

e-a-b-c-d

Topological Sort Example



"Linked List"

f-e-a-b-c-d

Detecting Cycles in graphs

- One can detect a cycle in a graph using a small modification of DFS
- A graph contains a cycle iff it contains a back edge
- An edge (u, v) is said to be a back edge if v is an ancestor of u.
- v is an ancestor of u iff v has a gray color.

Detecting cycles in graphs

```
function DFS-VISIT(adj, u)
    u.color ← GRAY
    time ← time + 1
    u.d ← time
    foreach v ∈ adj[u] do
        if v.color = WHITE then
            v.p ← u
            DFS-VISIT(adj, v)
        else if v.color = GRAY ∧ v ≠ u.p then
            cycle ← true
    u.color ← BLACK
    times ← time + 1
    u.f ← time
```

Strongly Connected Components

The following algorithm computes the strongly connected components of a graph in $O(n+m)$ time

1. call DFS to compute finishing time for all nodes.
2. create G^T.
3. call DFS in order of decreasing finishing time.

Kosaraju Algorithm

foreach $v \in V$ **do**
    if $v.color = WHITE$ **then**
        DFS-VISIT(v)
Reverse all the edges of G and reset all colors
foreach $v \in V$ *in decreasing finish time* **do**
    if $v.color = WHITE$ **then**
        DFS-VISIT(v)