Elementary Graph Algorithms

Textbook, Chapter 23, Sections 23.1, 23.2, 23.3, and 23.4
Please concentrate on the algorithms, their complexity and the main results, you may ignore the proofs of this chapter.

CSCE310: Data Structures and Algorithms

`www.cse.unl.edu/~choueiry/S01-310/`

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104
`choueiry@cse.unl.edu`, Tel: (402)472-5444

Methods for:

- representing graphs
- searching graphs

Searching algorithms:

- follow systematically the edges to visit the vertices
- discover structural information of graph
- are central to Algorithmic Graph Theory

Outline

- Representation: Adjacency lists
- Representation: Adjacency matrices
- Breadth-first search (BFS)
- Depth-first search (DFS)
- Topological search of a DAG: as application of Depth-first search
- (Strongly connected components in a directed graph)

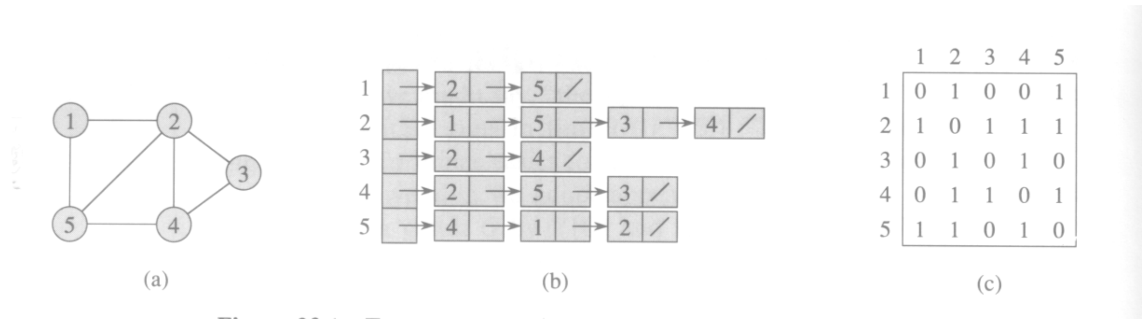
Representations of Graphs: $G = (V, E)$

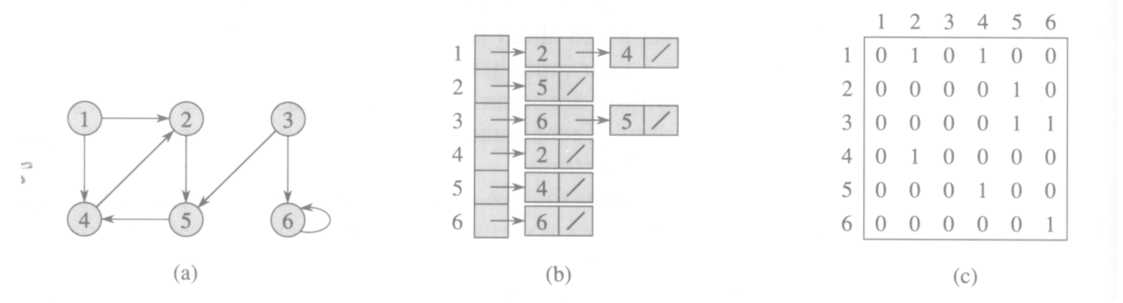
Complete graph: $\|E_{max}\| = \frac{\|V\|(\|V\|-1)}{2}$
Connected graph: $\|E_{min}\| = (\|V\| - 1)$
Sparse graph: $\|E\|$ much smaller than $\|E_{max}\| = O(\|V\|^2)$
Dense graph: $\|E\|$ close to $\|E_{max}\| = O(\|V\|^2)$

Two standard representations:

1. Adjacency list: preferable for sparse graphs
2. Adjacency matrix: preferable for dense graphs
   also for quickly checking whether two vertices are connected

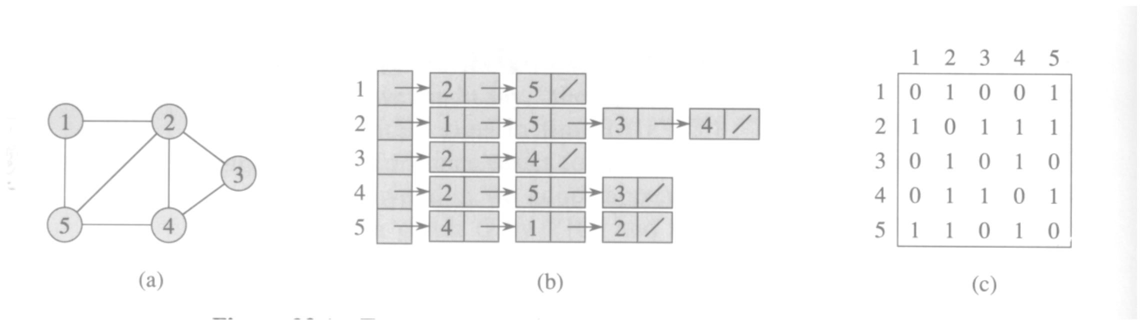
Adjacency-list representation:



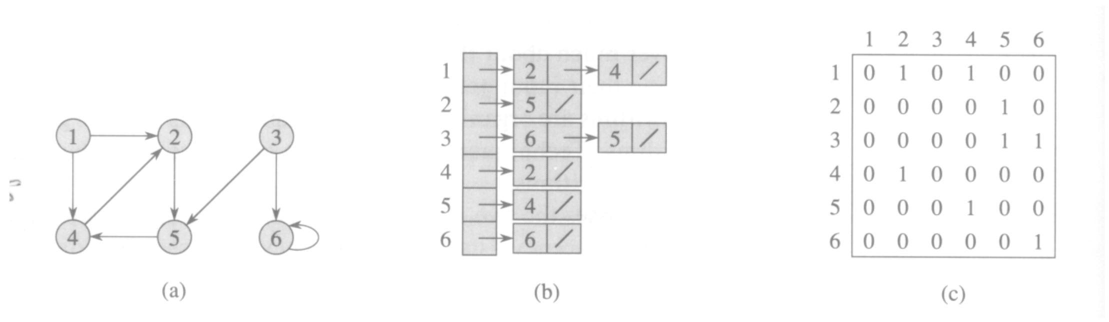


- Array Adj of $\|V\|$ lists
- One list per vertex $u \in V$
- $\forall\, u \in V$, $Adj[u]$ contains pointers to vertices $v \in V$ adjacent to u
- Pointers in adjacency lists stored in arbitrary order

Adjacency-list representation:


(a) (b) (c)

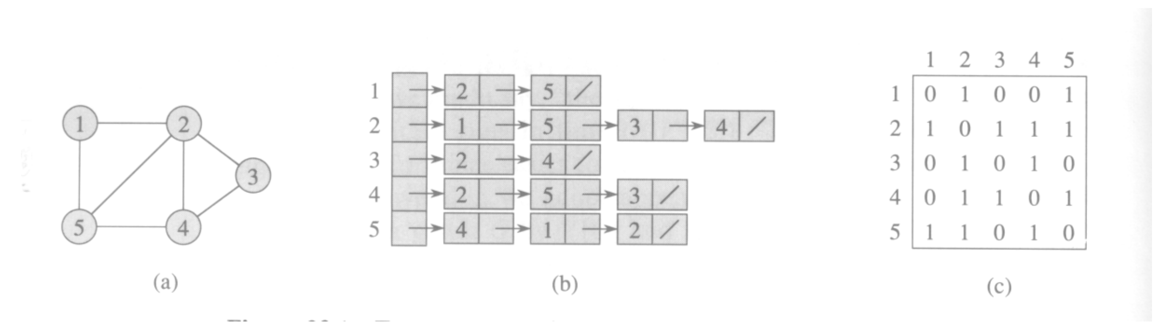
$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$


(a) (b) (c)

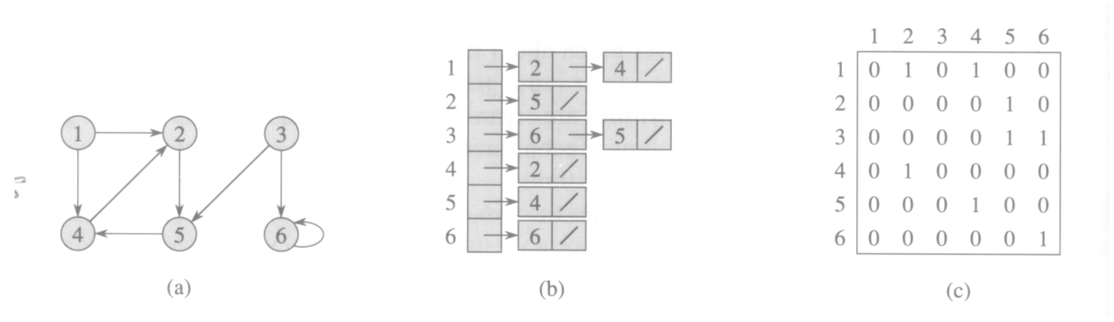
$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

- Directed graph: sum (length (all adjacency lists)) = $\|E\|$
- Undirected graph: sum (length (all adjacency lists)) = $2\|E\|$
  Since every edge appears twice, once per each vertex
- In all cases: space is $O(max(V, E)) = O(V + E)$

Adjacency-list representation:



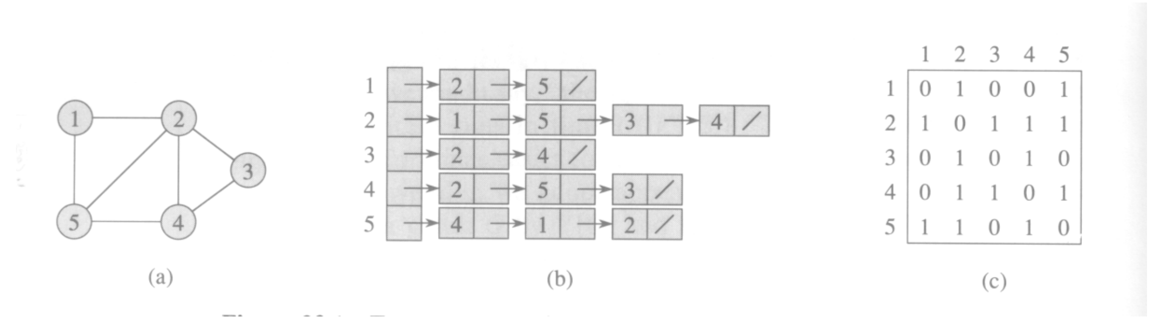
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

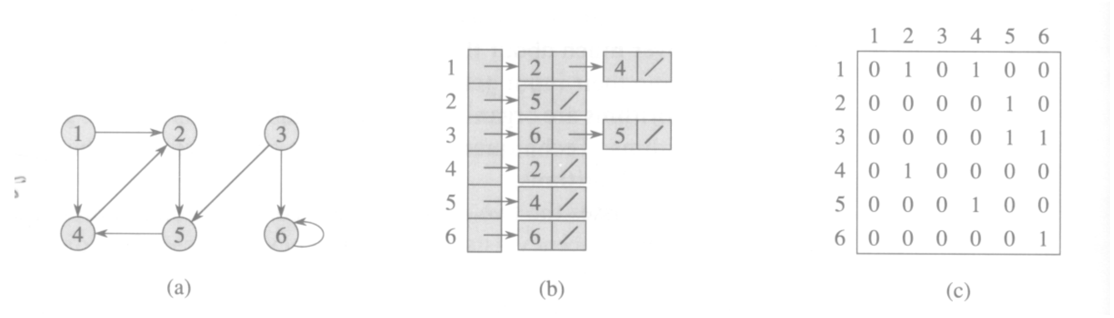
- Advantage: can easily represent weighted graphs: $w(u,v)$ of $(u,v) \in E$ can be stored with (pointer to) v in $Adj[u]$
- Inconvenient: To determine if (u,v) is in graph, we must search v in $Adj[u]$, $O(\|V\|)$

Adjacency-matrix representation:



	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

(a) (b) (c)

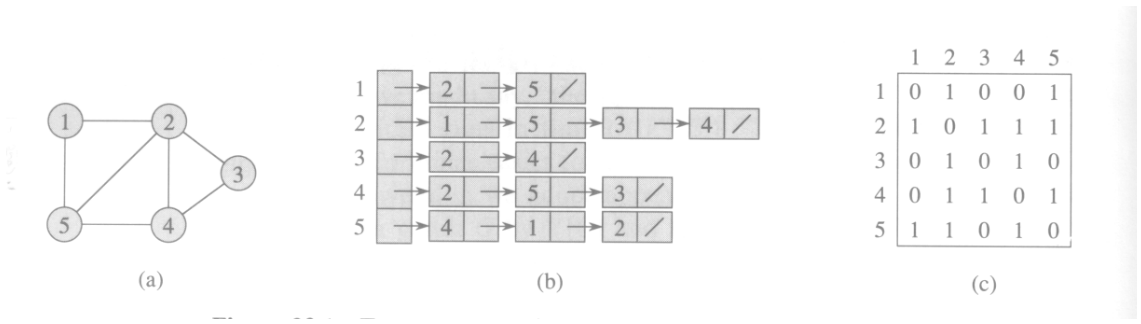


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

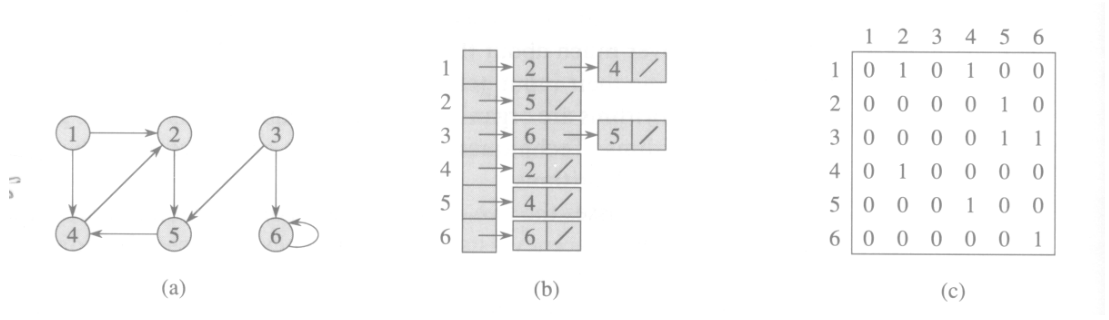
(a) (b) (c)

- Vertices are numbered arbitrarily: 1, 2, $\ldots$, $\|V\|$
- $A = (a_{ij})$ is a $\|V\| \times \|V\|$ matrix with $a_{ij} = 1$ if $i, j \in E$ and $a_{ij} = 1$ otherwise

Adjacency-matrix representation:



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

- Requires $\Theta(V^2)$ space, $\underline{\text{independent of } \|E\|}$
- Undirected graph: $A = A^T$, we can save space below diagonal
- Weighted graphs: $w(u,v)$ stored in $A[u,v]$
- (u,v) does not exist: $A[u,v] = \texttt{Nil}$, or sometimes $A[u,v] = 0, \infty$, depending on application

Adjacency list vs. Adjacency matrix

- Adjacency matrix preferable when graphs are small
- When graphs are weighted, matrix can store weight at no additional space cost

Breadth-first Search (BFS): what is does

Given: $G = (V, E)$ and source (vertex $s \in E$)

- Systematically explores edges of G
  to 'discover' every vertex e reachable from s
- Computes distance (fewest #edges) from s to all reachable vertices v
- Produces breadth-first tree, rooted at s and containing all reachable vertices v
- Path in tree from v to s is the shortest path
- Works both on directed and undirected graphs

BFS: how it does it

- First discovers all vertices at distance 1 of s, then distance 2 of s, etc.
- Discovers all vertices at distance k from s before discovering any vertices at distance $k+1$
- Expands frontier between discovered and undiscovered nodes uniformly, across breadth of frontier, also called the <u>fringe</u>

BFS: ‘coloring’

- Progress monitored by coloring vertices: white, gray, black. White: not visited, Black: fully expanded, Gray: visited but not fully expanded (fringe),
- All nodes start white, and later may become grey then black
- A vertex is <u>discovered</u> first time it is encountered (becomes gray or black)
- A vertex u is black when every v with $(u, v) \in E$ is grey or black (has been discovered)
- A grey vertex may have some white neighbors: gray vertices are the fringe, the frontier between discovered and undiscovered vertices

BFS: mechanism

- Initially, contains only s, root
- When a white vertex v is discovered during the scanning of the adjacency list of an already discovered vertex u, v and (u, v) are added to the tree: u is the predecessor of parent of v in the three: $\pi[v] = u$. At most one parent per vertex.
- If u is on path in the tree from root s to a vertex v, u is ancestor of v and v is descendant of u

BFS: procedure

- Assumes adjacency-list representation
- $color[u]$: color of a vertex
- $\pi[u]$: predecessor, or parent, or u
  $\pi[s] = \texttt{Nil}$ and $\pi[u] = \texttt{Nil}$ if u is white
- $d[u]$: distance from s to u, and is computed by the algorithm
- Q: queue, first-in first-out to manage fringe/gray vertices

BFS: pseudocode

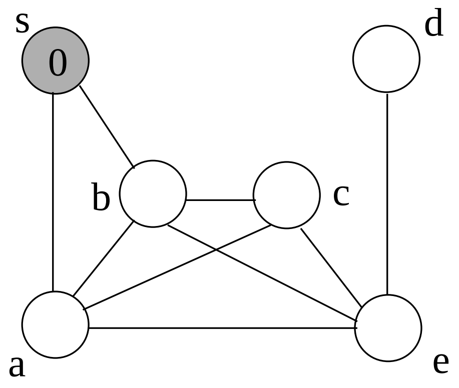
```
BFS($G, s$)
for each vertex $u \in V[G] - \{s\}$
    do $color[u] \leftarrow$ WHITE
       $d[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
$color[s] \leftarrow$ GRAY
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ NIL
$Q \leftarrow \{s\}$
while $Q \neq \emptyset$
    do $u \leftarrow head[Q]$
       for each $v \in Adj[u]$
           do if $color[v] =$ WHITE
                 then $color[v] \leftarrow$ GRAY
                      $d[v] \leftarrow d[u] + 1$
                      $\pi[v] \leftarrow u$
                      ENQUEUE($Q, v$)
       DEQUEUE($Q$)
       $color[u] \leftarrow$ BLACK
```

BFS: explanation of pseudocode

- Lines 1-4: paint every vertex u, white, set $d[u]$ and $\pi[u]$
- Line 5-6-7: initialize s, $color[s]$, $d[s]$, $\pi[s]$
- Line 8: initializes Q, puts s in the fringe
  Q will contain gray vertices
- Lines 9–18: main loop, iterates until Q empty (all nodes black, expanded)
- Line 10: pops vertex from head of queue
- Line 11: considers each vertex v in $Adj[u]$, if v is white (it is not discovered),
- Line 13-16: then BFS discovers it: update color, parent, distance and put in tail of Q
- Lines 17-18: When all neighbors of u have been examined, u is blackened and removed from Q (fully expanded)

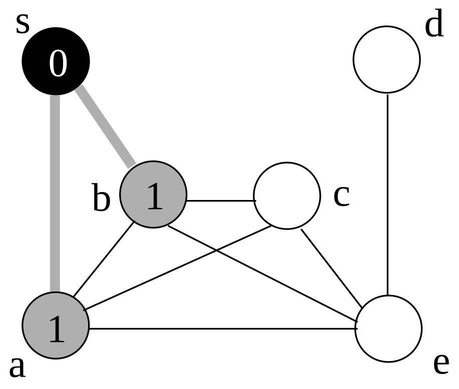
BFS: Example

Courtesy of Dr. Cusack



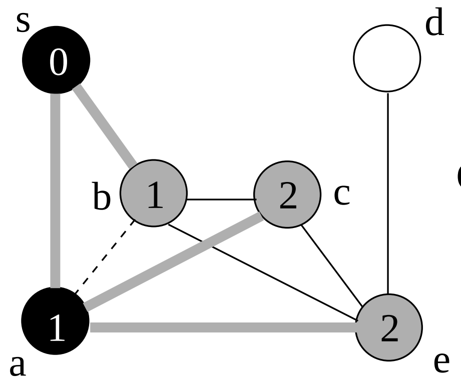
Q=[s]

v	adjacent	d	p	color
s	a,b	0	nil	gray
a	s,b,c,e	inf	nil	white
b	s,a,c,e	inf	nil	white
c	b,a,e	inf	nil	white
d	e	inf	nil	white
e	a,b,c,d	inf	nil	white



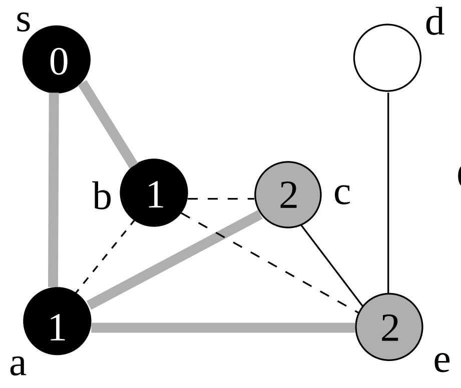
Q=[a,b]

v	adjacent	d	p	color
s	a,b	0	nil	black
a	s,b,c,e	1	s	gray
b	s,a,c,e	1	s	gray
c	b,a,e	inf	nil	white
d	e	inf	nil	white
e	a,b,c,d	inf	nil	white



Q=[b,c,e]

v	adjacent	d	p	color
s	a,b	0	nil	black
a	s,b,c,e	1	s	black
b	s,a,c,e	1	s	gray
c	b,a,e	2	a	gray
d	e	inf	nil	white
e	a,b,c,d	2	a	gray

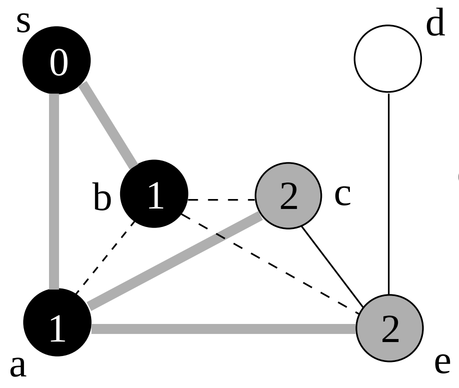


Q=[c,e]

v	adjacent	d	p	color
s	a,b	0	nil	black
a	s,b,c,e	1	s	black
b	s,a,c,e	1	s	black
c	b,a,e	2	a	gray
d	e	inf	nil	white
e	a,b,c,d	2	a	gray

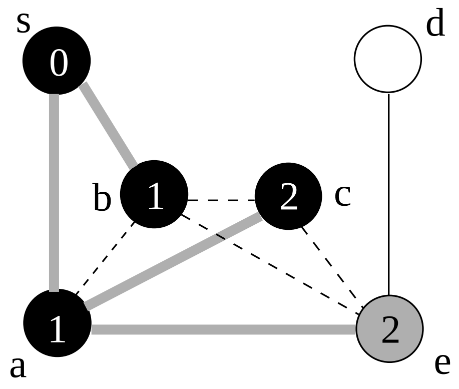
BFS: Example

Courtesy of Dr. Cusack



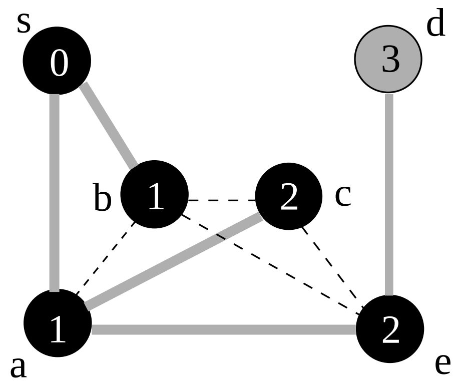
Q=[c,e]

v	adjacent	d	p	color
s	a,b	0	nil	black
a	s,b,c,e	1	s	black
b	s,a,c,e	1	s	black
c	b,a,e	2	a	gray
d	e	inf	nil	white
e	a,b,c,d	2	a	gray



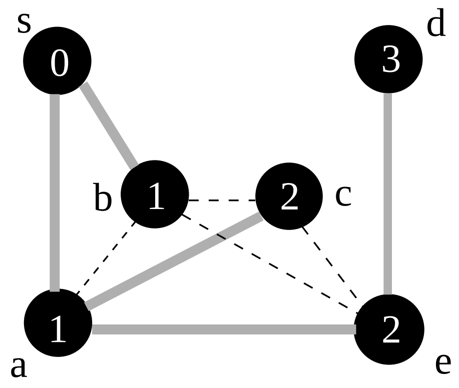
Q=[e]

v	adjacent	d	p	color
s	a,b	0	nil	black
a	s,b,c,e	1	s	black
b	s,a,c,e	1	s	black
c	b,a,e	2	a	black
d	e	inf	nil	white
e	a,b,c,d	2	a	gray



Q=[d]

v	adjacent	d	p	color
s	a,b	0	nil	black
a	s,b,c,e	1	s	black
b	s,a,c,e	1	s	black
c	b,a,e	2	a	black
d	e	3	e	grey
e	a,b,c,d	2	a	black



Q=[]

v	adjacent	d	p	color
s	a,b	0	nil	black
a	s,b,c,e	1	s	black
b	s,a,c,e	1	s	black
c	b,a,e	2	a	black
d	e	3	e	black
e	a,b,c,d	2	a	black

BFS: Complexity

- Line 12: ensures that every node is examined (and examined) at most once, and hence dequeued at most once.
- Enqueuing, Dequeuing: $O(1)$
- Thus, total time for queue operations: $O(V)$
- Since $Adj[u]$ is scanned only before u is dequeued, it is scanned at most once
- Sum of length of adjacency lists is $\Theta(E) \Rightarrow$ time spent on total scanning is $O(E)$
- Overhead for initialization: $O(V)$
- Total running time of BFS is $O(V + E)$, linear in size of the adjacency-list representation of G

Shortest path: Definitions

- The shortest-path distance $\delta(s, v)$ from s to v is defined as the minimum #edges in any path from s to v, or ∞ if there is no path from s to v.
- A path of length of $\delta(s, v)$ is a shortest path from s to v
- Important result: BFS computes shortest-path distances (d) and shortest paths (BF-tree) to all reachable vertices

BFS: Analysis

$G = (V, E)$: (un)directed graph and $s \in V$: an arbitrary vertex

- **Lemma 23.1:** $\forall (u, v) \in E, \delta(s, v) \leq \delta(s, u) + 1$
- **Lemma 23.2:** When BFS in run on G from source s, upon termination, $\forall v \in V, d[v] \geq \delta(s, v)$
- To prove $d[v] \geq \delta(s, v)$, we need to look at how Q operates
- **Lemma 23.2:** During BFS, suppose $Q = \langle v_1, v_2, \ldots, v_r \rangle$ (v_1 is head and v_r tail), then $d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_{i+1}]$ for $i = 1, 2, \ldots, r - 1$
- **Theorem 23.4:** Correctness of BFS.
  BFS discovers every vertex $v \in V$ reachable from the source s
  Upon termination, $d[v] = \delta(s, v)$
  $\forall v \neq s$, reachable from s, one of the shortest path from s to v is the shortest path from s to $\pi[v]$ followed by the edge $(\pi[v], v)$

Breadth-first tree

```
Print-Path($G, s, v$)
If $v = s$
  then print $s$
  else if $\pi[v] =$ nil
        then print "no path from" $s$ "to" $v$ "exists"
        else Print-Path($G, , s, \pi[v]$)
             print $v$
```

- By Lemma 23.5: BFS builds the BF-tree as it searches the graph: $\pi[v]$
- Procedure runs in time linear in the number of vertices in the path: each recursive call is for a path of one vertex shorter

Breadth-first tree

The π field of each vertex defines the <u>predecessor tree</u> of each node

The breadth-first tree of G is the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ where:

- $V_\pi = \{v \in V \ : \pi[v] \neq Nil\} \cup \{s\}$
- $E_\pi = \{(\pi[v], v) \in E \ : v \in V_\pi - \{s\}\}$

Depth-first search: Principle

- Searched <u>deeper</u> in the graph whenever possible
- Explores the most recently discovered node v that still has unexplored neighbors
- When all neighbors of v have been explored, it backtracks to explore the unexplored neighbors of the parent of v, if any
- Repeats until discovering all nodes reachable from an original source vertex
- If unexplored nodes remain, choose one as new source, and repeat procedure
- Repeat until all vertices are discovered

Depth-first search: time-stamp

- Each node has two “time-stamps”, $d[v]$ and $f[v]$
- $d[v]$ records when it is discovered
- $f[v]$ records when we are done considering its adjacency list (and we backtrack) (time-stamps $\in [1 \ldots 2\|V\|]$
- $d[u] < f[u]$, used in other algorithms such as topological sort)

Depth-first search: coloring

Nodes are colored as they are visited, a node u is

- first white, before time $d[u]$
- gray when discovered, between time $d[u]$ and $f[u]$
- and black when finished, (i.e., when adjacency list has been visited completely, when we backtrack over it), after time $f[u]$

Depth-first search: data structures

- timestamps, d, f
- color, c
- predecessor, π

Depth-first forest

- Whenever a node v is discovered while scanning the adjacency list of a node u, $\pi(v) \rightarrow u$ (π = predecessor)
- Predecessor subgraph:
  $E_\pi = \{(\pi[v], v) \in E \ : v \in V \text{ and } \pi[v] \neq Nil\}$
- Predecessor subgraph may have several trees, a forest
- Each node of V appears in exactly one tree: forest is made up of disjoint trees

Depth-first forest: pseudocode

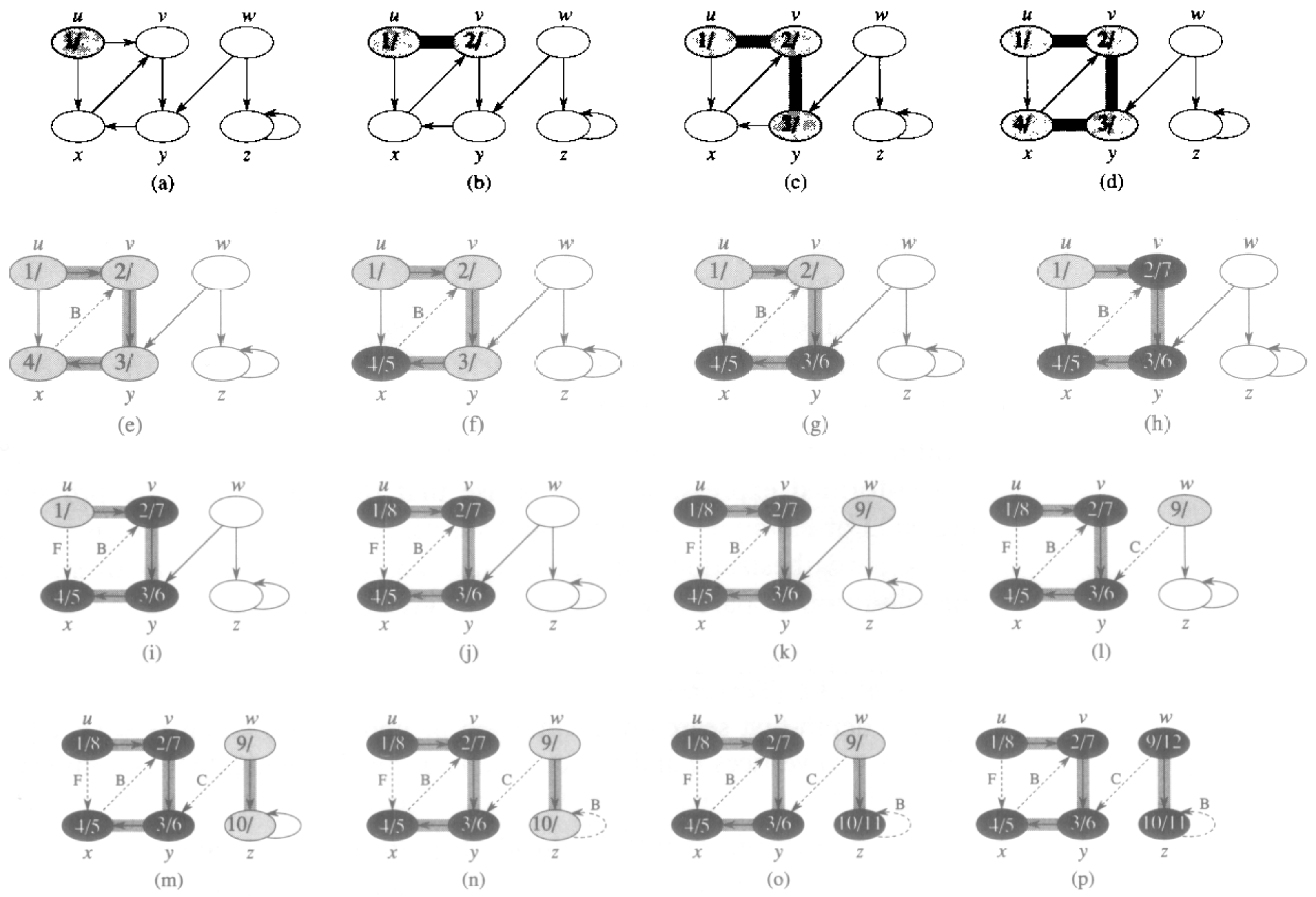
```
DFS($G$)
1  for each vertex $u \in V[G]$
2      do $color[u] \leftarrow$ WHITE
3         $\pi[u] \leftarrow$ NIL
4  $time \leftarrow 0$
5  for each vertex $u \in V[G]$
6      do if $color[u] =$ WHITE
7            then DFS-VISIT($u$)
```

```
DFS-VISIT($u$)
1  $color[u] \leftarrow$ GRAY          $\triangleright$ White vertex $u$ has just been discovered.
2  $d[u] \leftarrow time \leftarrow time + 1$
3  for each $v \in Adj[u]$        $\triangleright$ Explore edge $(u, v)$.
4      do if $color[v] =$ WHITE
5            then $\pi[v] \leftarrow u$
6                 DFS-VISIT($v$)
7  $color[u] \leftarrow$ BLACK         $\triangleright$ Blacken $u$; it is finished.
8  $f[u] \leftarrow time \leftarrow time + 1$
```

Depth-first forest: running time

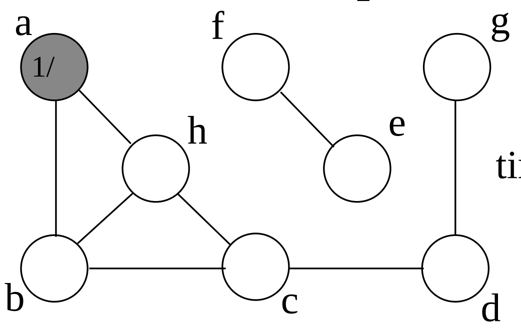
- Lines 1-2, 5-7: $\Theta(V)$ (exclusive call to `DFS-Visit`)
- `DFS-Visit` called only on white nodes, so, once on every node
- In `DFS-Visit`, lines 3-6 is called $\|Adj[v]\|$ times, in total $\Theta(E)$
- Total cost is $\Theta(V + E)$

Depth-first forest: Example



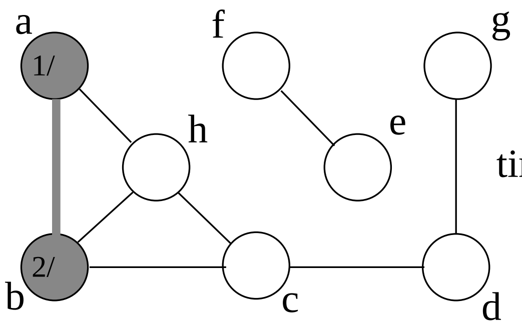
BFS: Example

Courtesy of Dr. Cusack



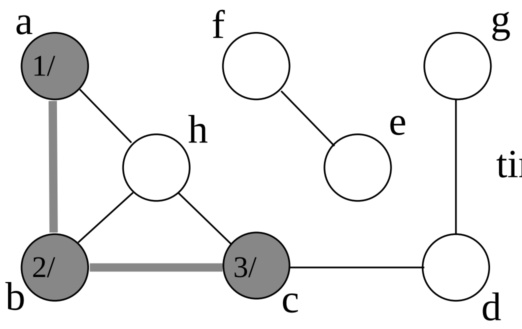
time=1

v	a	b	c	d	e	f	g	h
p	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL
d	1							
f								
c	G	W	W	W	W	W	W	W



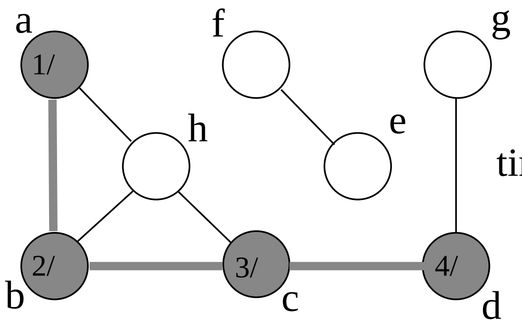
time=2

v	a	b	c	d	e	f	g	h
p	NIL	a	NIL	NIL	NIL	NIL	NIL	NIL
d	1	2						
f								
c	G	G	W	W	W	W	W	W



time=3

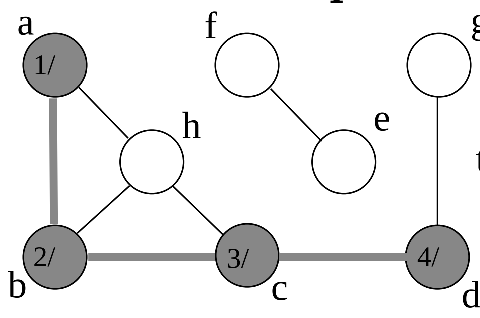
v	a	b	c	d	e	f	g	h
p	NIL	a	b	NIL	NIL	NIL	NIL	NIL
d	1	2	3					
f								
c	G	G	G	W	W	W	W	W



time=4

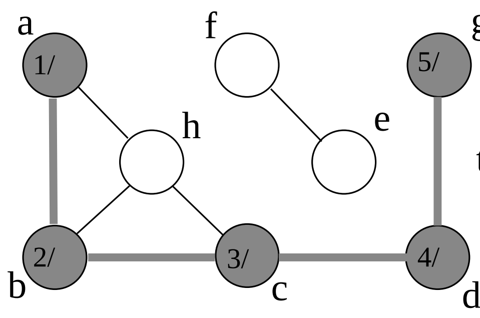
v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	NIL	NIL
d	1	2	3	4				
f								
c	G	G	G	G	W	W	W	W

BFS: Example *Courtesy of Dr. Cusack*



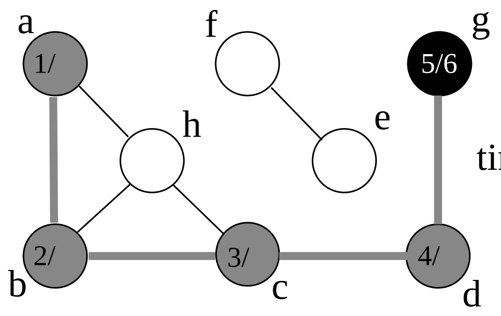
time=4

v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	NIL	NIL
d	1	2	3	4				
f								
c	G	G	G	G	W	W	W	W



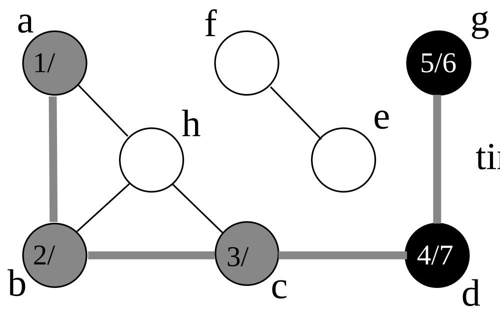
time=5

v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	NIL
d	1	2	3	4			5	
f								
c	G	G	G	G	W	W	G	W



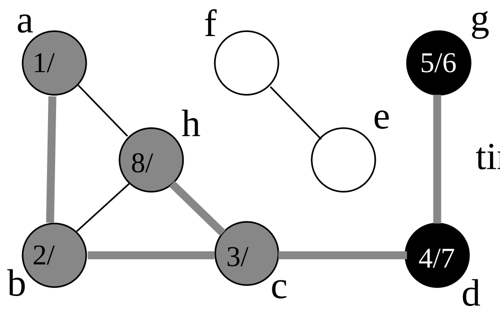
time=6

v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	NIL
d	1	2	3	4			5	
f							6	
c	G	G	G	G	W	W	B	W



time=7

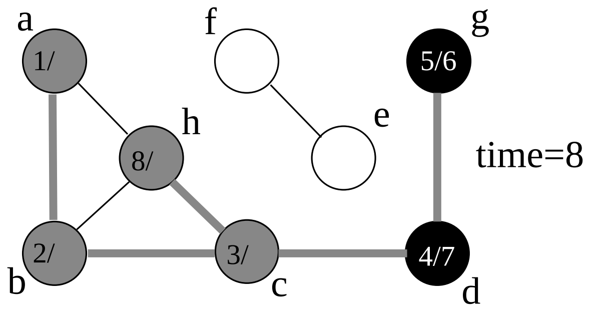
v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	NIL
d	1	2	3	4			5	
f				7			6	
c	G	G	G	B	W	W	B	W



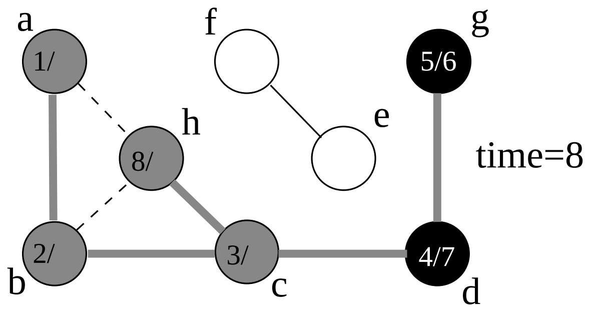
time=8

v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	c
d	1	2	3	4			5	8
f				7			6	
c	G	G	G	B	W	W	B	G

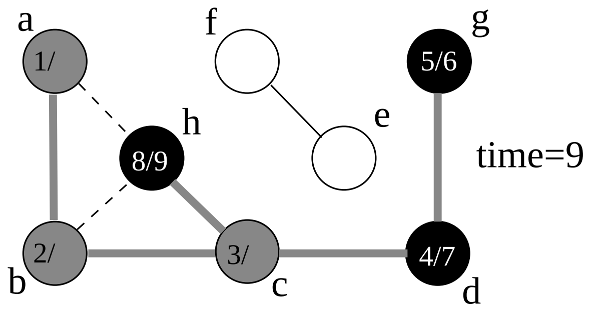
BFS: Example *Courtesy of Dr. Cusack*



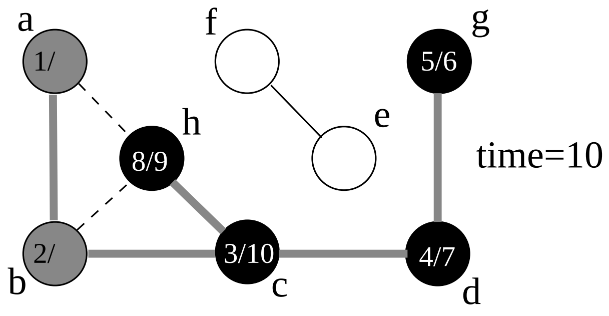
v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	c
d	1	2	3	4			5	8
f				7			6	
c	G	G	G	B	W	W	B	G



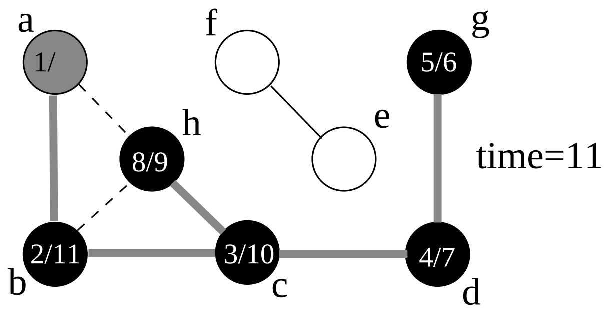
v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	c
d	1	2	3	4			5	8
f				7			6	
c	G	G	G	B	W	W	B	G



v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	c
d	1	2	3	4			5	8
f				7			6	9
c	G	G	G	B	W	W	B	B



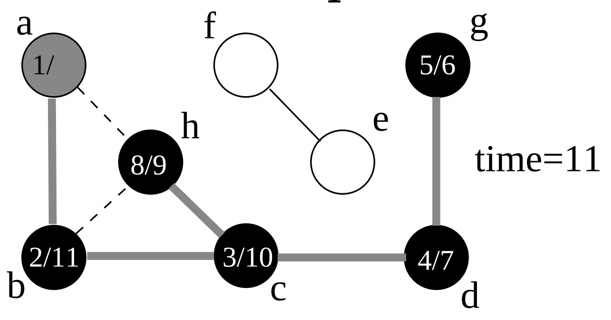
v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	c
d	1	2	3	4			5	8
f			10	7			6	9
c	G	G	B	B	W	W	B	B



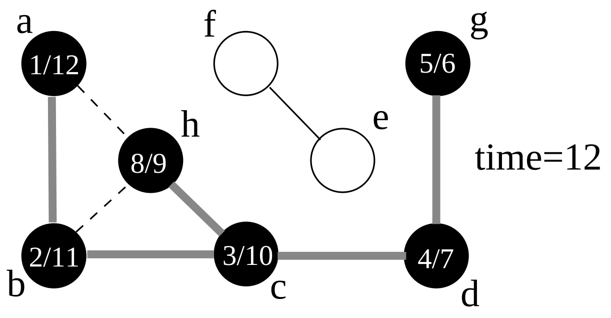
v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	c
d	1	2	3	4			5	8
f		11	10	7			6	9
c	G	B	B	B	W	W	B	B

BFS: Example

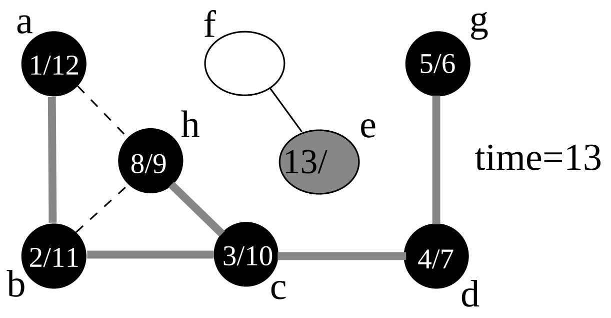
Courtesy of Dr. Cusack



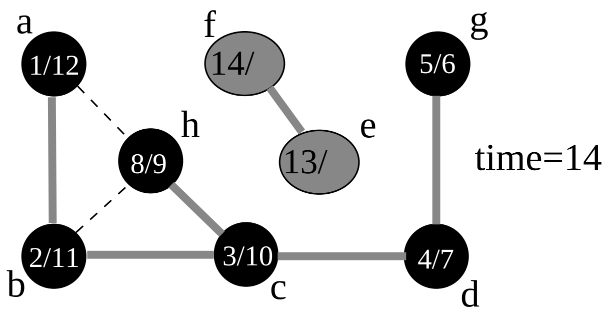
v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	c
d	1	2	3	4			5	8
f		11	10	7			6	9
c	G	B	B	B	W	W	B	B



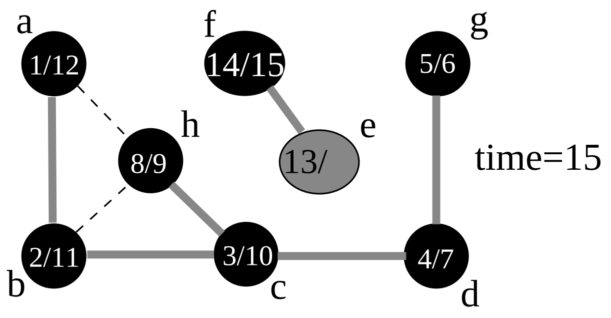
v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	c
d	1	2	3	4			5	8
f	12	11	10	7			6	9
c	B	B	B	B	W	W	B	B



v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	NIL	d	c
d	1	2	3	4	13		5	8
f	12	11	10	7			6	9
c	B	B	B	B	G	W	B	B



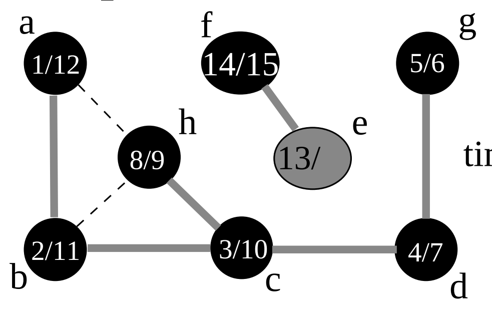
v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	e	d	c
d	1	2	3	4	13	14	5	8
f	12	11	10	7			6	9
c	B	B	B	B	G	G	B	B



v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	e	d	c
d	1	2	3	4	13	14	5	8
f	12	11	10	7		15	6	9
c	B	B	B	B	G	B	B	B

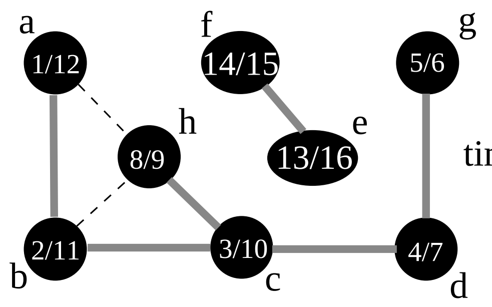
BFS: Example

Courtesy of Dr. Cusack



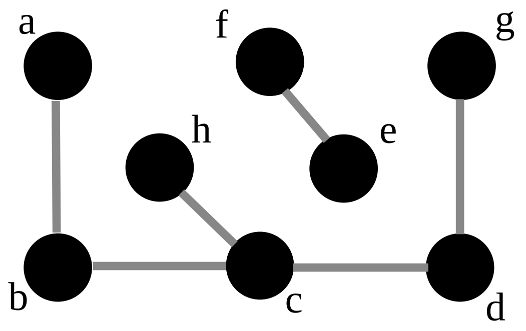
time=15

v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	e	d	c
d	1	2	3	4	13	14	5	8
f	12	11	10	7		15	6	9
c	B	B	B	B	G	B	B	B



time=16

v	a	b	c	d	e	f	g	h
p	NIL	a	b	c	NIL	e	d	c
d	1	2	3	4	13	14	5	8
f	12	11	10	7	16	15	6	9
c	B	B	B	B	B	B	B	B



Topological sort

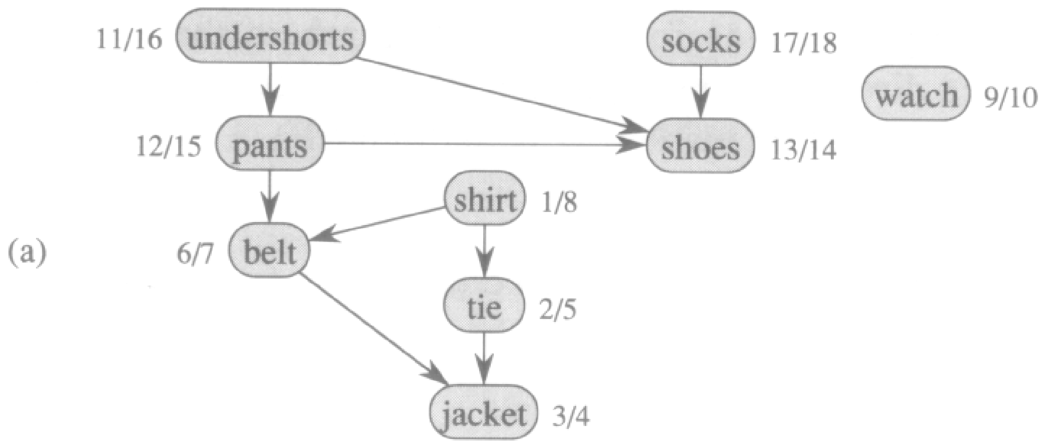
Topological sort of a dag $G = (V, E)$ is a linear ordering of all nodes in G such that for every edge directed/oriented from a node u to a node v, u appears before v in the ordering

Topological sort is an ordering of the nodes along a horizontal line so that all directed edges for from left to right

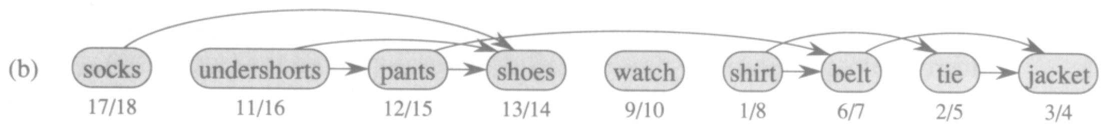
When an directed graph is <u>not</u> acyclic, a topological sort does not exist

DAG: Example

A directed edge indicates that garment u must be donned before garment v



One possible order of getting dressed: topological sort!



Topological sort: pseudocode

```
Topological-Sort($G$)
  • Call DFS($G$) to compute $f(v)$ of each vertex
  • As a vertex is finished (its $f(v)$ computed) insert it onto the
    front of a linked list
  • return the linked list of vertices
```

Complexity of `Topological-Sort(`G`)` is $\Theta(V+E)$:
— DFS(G) is $\Theta(V+E)$
— insertion in front of list $\Theta(1)$