THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

COMP2013
Data Structures and Algorithms

Lecture 12
Graphs

Dr. Jieming Shi

Social Networks

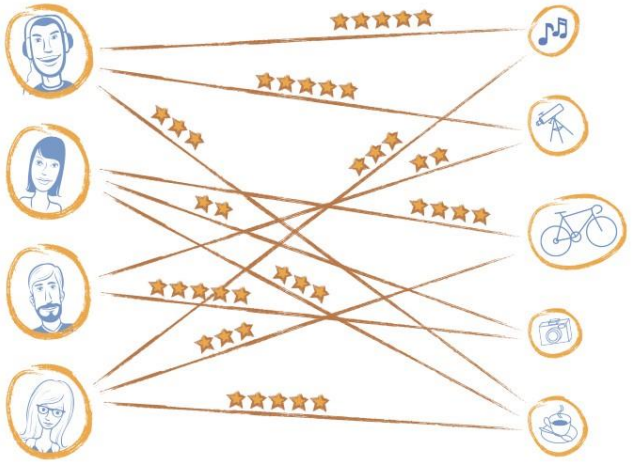
Nodes (Vertices): users, images, etc (all kinds of entities)
Edges: friendship, followship, etc (all kinds of relationships)

Community detection; Link prediction; Product/Ad advertisement; Node classification

User-Item Graphs

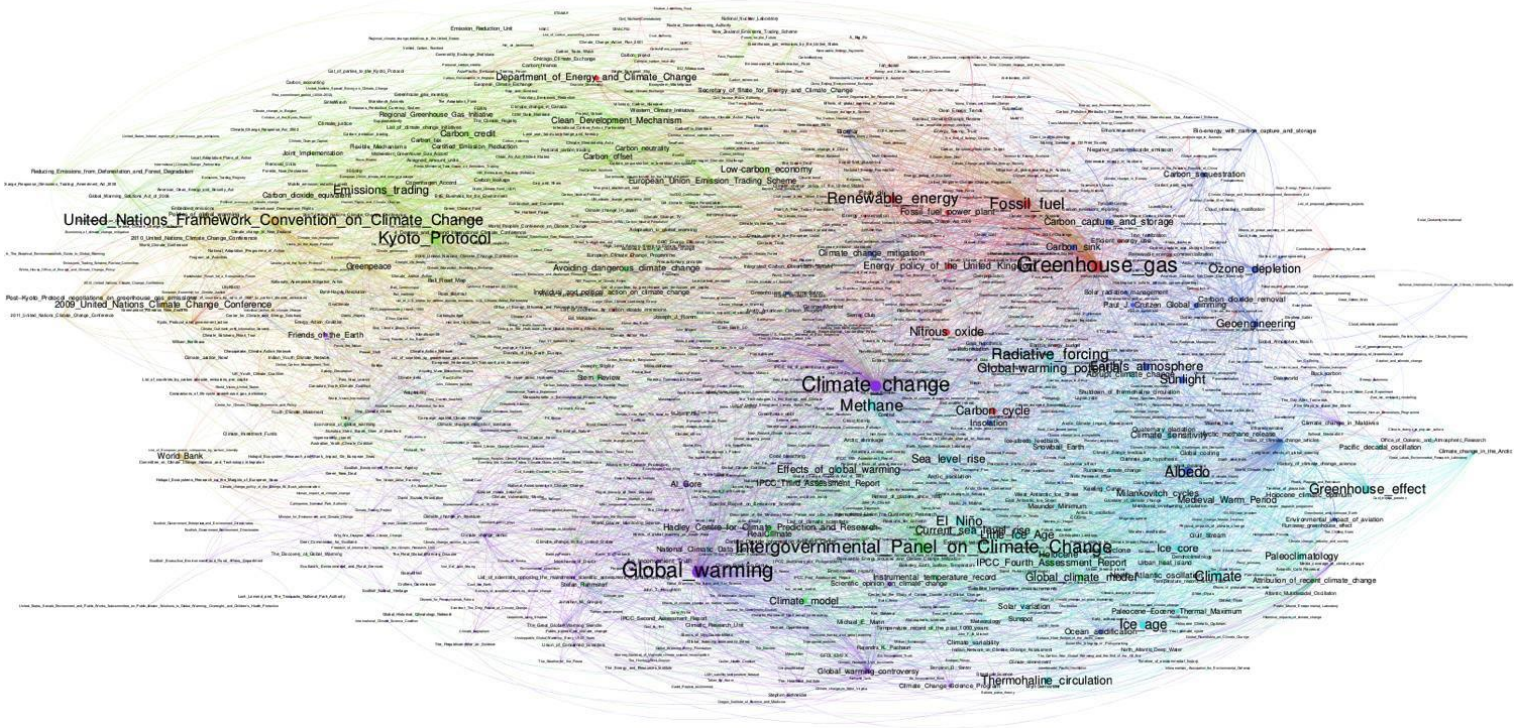
- Bipartite Graphs
- Vertices: users, items
- Edges: ratings, purchasing,
- reviewing …

- Recommendation systems

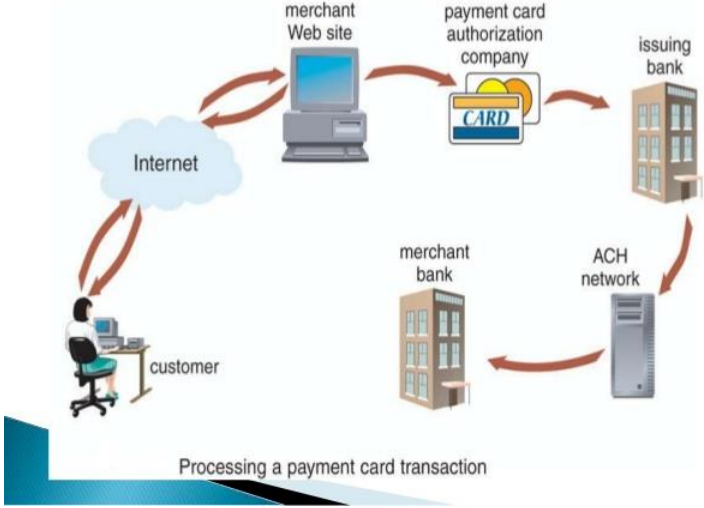


Web Graphs

- Vertices: webpages
- Edges: hyperlinks



Transaction Networks

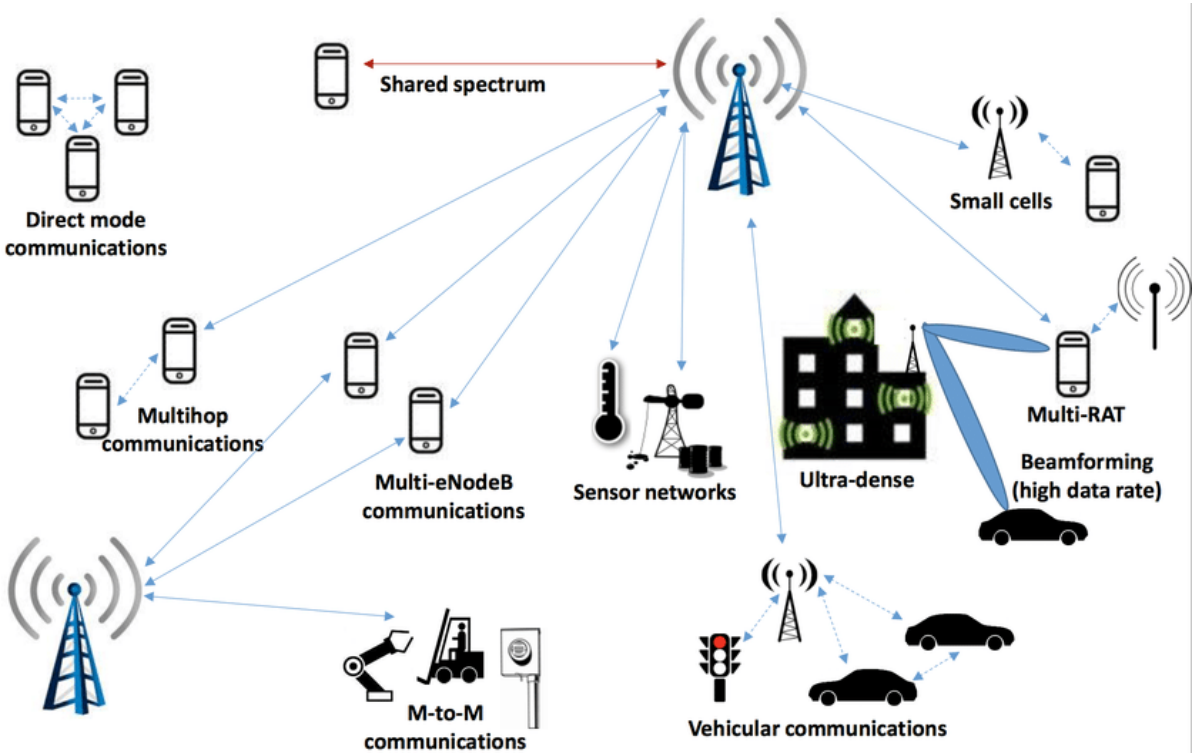


PayPal

WeChat Pay

Alipay™

Telecommunication Networks

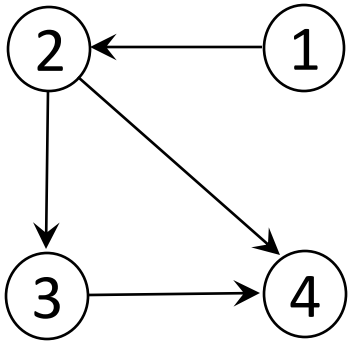


Airline Networks

Graph

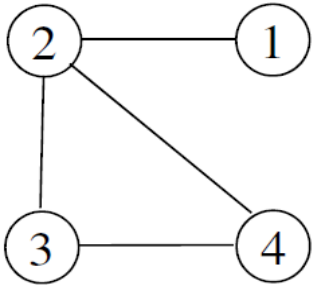
- Graph model $G = (V, E)$
  - V is a set of vertices
  - E is a set of edges; each edge is a pair of vertices
- Example:
  - $V = \{1, 2, 3, 4\}$
  - $E = \{(1,2), (2,3), (3,4), (2,4)\}$

Graph	Vertex	Edge
Road network	Road Junction	Road Segment
Social Network	Person	Friend Relationship
Computer Network	Computer/Router	Network Link

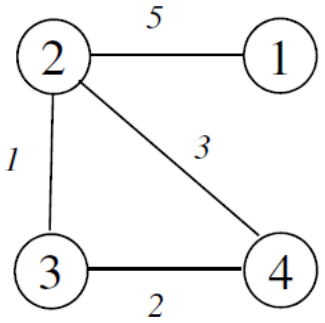


Weighted vs Unweighted

- Unweighted graph
  - An edge does not have any weight
    - i.e., every edge has the same unit weight 1
  - E.g., Facebook friend relationship

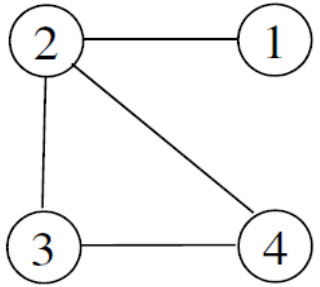


- Weighted graph
  - An edge has a weight
  - E.g., road segment: length
  - E.g., communication link: bandwidth

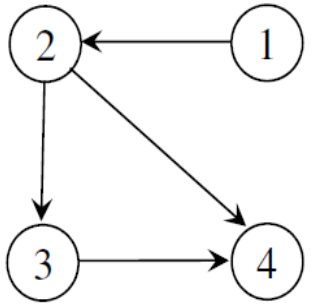


Directed vs Undirected

- Undirected graph
  - An edge can be traveled in both ways
  - E.g., Facebook friend relationship

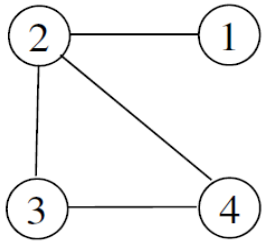


- Directed graph
  - An edge can be traveled in one way
  - E.g., Twitter follower relationship

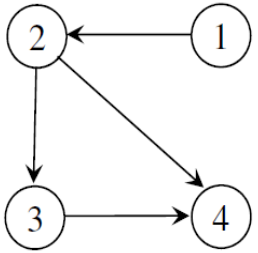


Neighbors and Degree of a Node

- Undirected Graph:
  - Neighbors of node 2: $N(2) = \{1, 3, 4\}$
  - Degree of node 2 is 3

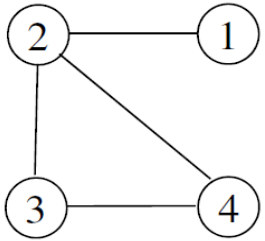


- Directed Graph:
  - In-neighbors of node 2: $N_{in}(2) = \{1\}$
  - In-degree of node 2 is 1
  - Out-neighbors of node 2: $N_{out}(2) = \{3, 4\}$
  - Out-degree of node 2 is 2

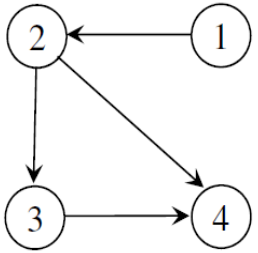


Paths

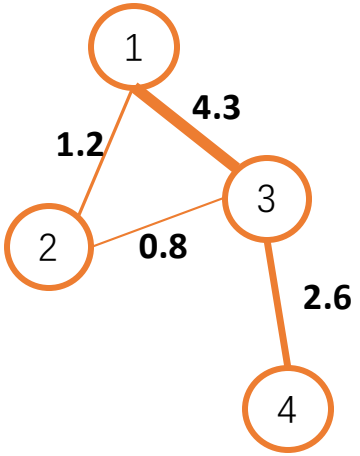
- Undirected Graph: undirected paths
  - 1<->2<->4<->2<->3 is a path with length 4
  - 1<->2<->3 is a path with length 2
    - This is a *shortest* path between 1 and 3
    - Shortest path *distance* between 1 and 3 is 2



- Directed Graph: directed paths
  - 1->2->3->4 is a path with length 3
  - 1->2->4 is a path with length 2
    - This is the *shortest* path from 1 to 4
    - Shortest path *distance* from 1 to 4 is 2
  - There is no path from 4 to 1
    - Shortest path distance from 4 to 1 is infinite.



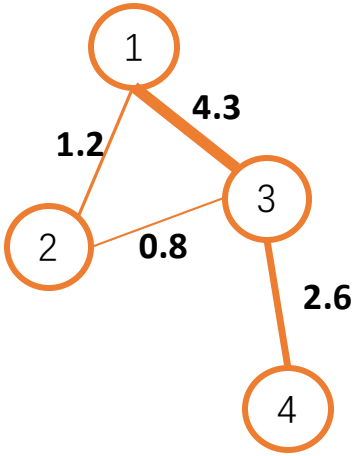
Graph data structures



Edge List

E	Node1	Node2	Weight
1	1	2	1.2
2	1	3	4.3
3	2	3	0.8
4	3	4	2.6

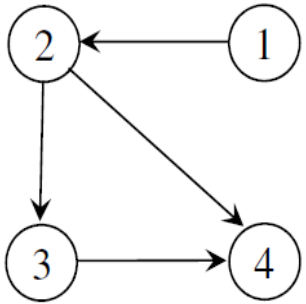
Graph data structures



Adjacency Matrix (Symmetric for undirected graph)

A	1	2	3	4
1	0	1.2	4.3	0
2	1.2	0	0.8	0
3	4.3	0.8	0	2.6
4	0	0	2.6	0

Graph data structures

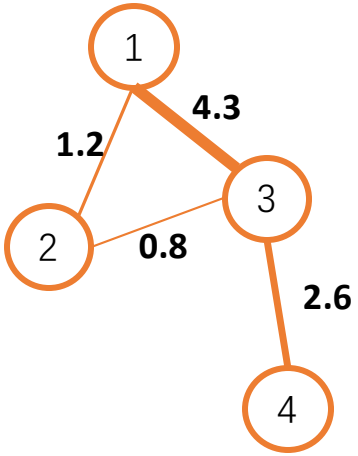


Adjacency Matrix (Asymmetric)

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

Graph data structures

- $G.Adj[v]$: the adjacency list of node v
  - i.e., the neighbors of v



Adjacency Lists

1: 2 (1.2) 3 (4.3)
2: 1 (1.2) 3 (0.8)
3: 1 (4.3) 2 (0.8) 4 (2.6)
4: 3 (2.6)

Graph Traversal

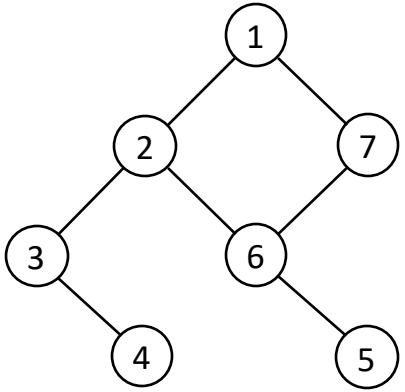
- Graph Traversal
  - Start from a source vertex *s*
  - Visit *all* vertices of the graph in a certain order
  - Each node is marked as visited only once

- Algorithms for graph traversal
  - Breadth-first search (BFS)
  - Depth-first search (DFS)

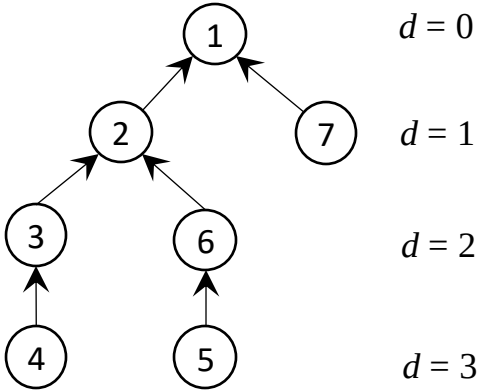
Breadth-first search (BFS)

- Breadth-first search (BFS)
  - Given a source vertex s in a graph $G = (V, E)$
  - Visit all vertices closer to s before any vertex farther from s
- It builds a breadth-first tree implicitly
  - s is the root
  - $u.p$ is the *parent* of u
  - $u.d$ is the "*distance*" from s to u

Distance from s to u: the shortest path distance from s to u



Example: $s = 1$



$d = 0$

$d = 1$

$d = 2$

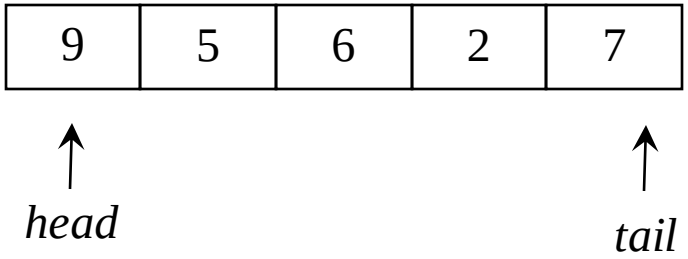
$d = 3$

Applications of BFS

- Finding shortest paths from node *s* to other nodes
- Finding shortest path distances
- Detecting cycles
- Constructing BFS tree/forest

Queue

- First In, First Out (FIFO)
- Operations
  - EnQueue(Q, x): O(1) time
    - Append x to the tail of the Q
  - DeQueue(Q): O(1) time
    - Pop out the head of the Q, and update head to be next element in the Q

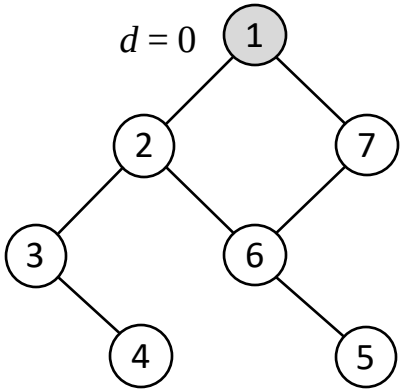


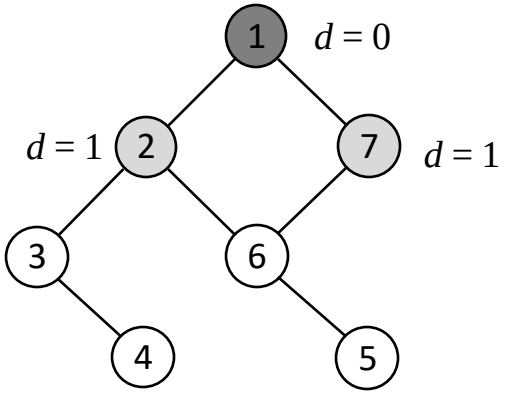
Example: BFS(G, 1)

[Beginning]
Q = (1: 0)

We are given the source vertex s = 1

[Iteration 1]
Q = (2: 1), (7: 1)





Which vertex will be examined in the next iteration?

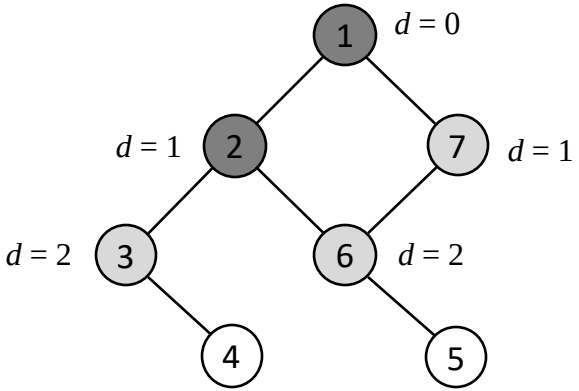
Light gray: vertices in Q

Dark gray: dequeued vertices

Example: BFS(G, 1)

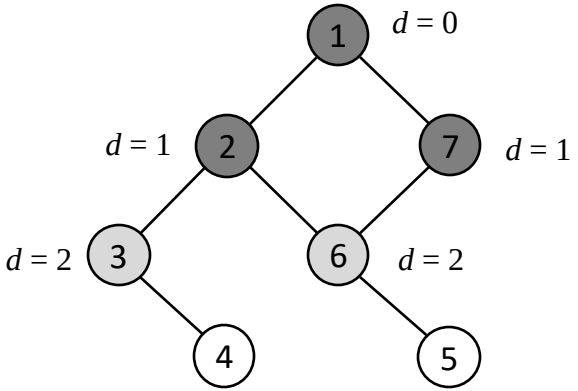
[Iteration 2]

Q = (7: 1), (3: 2), (6: 2)



[Iteration 3]

Q = (3: 2), (6: 2)



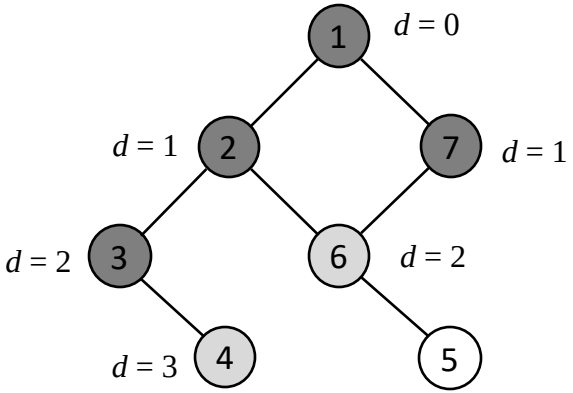
Light gray: vertices in Q

Dark gray: dequeued vertices

Example: BFS(G, 1)

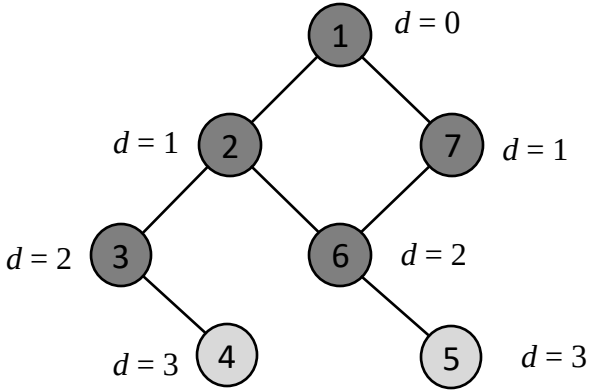
[Iteration 4]

Q = (6: 2), (4: 3)



[Iteration 5]

Q = (4: 3), (5: 3)



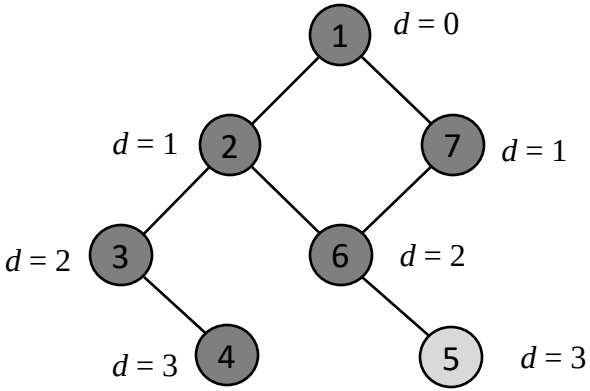
Light gray: vertices in Q

Dark gray: dequeued vertices

Example: BFS(G, 1)

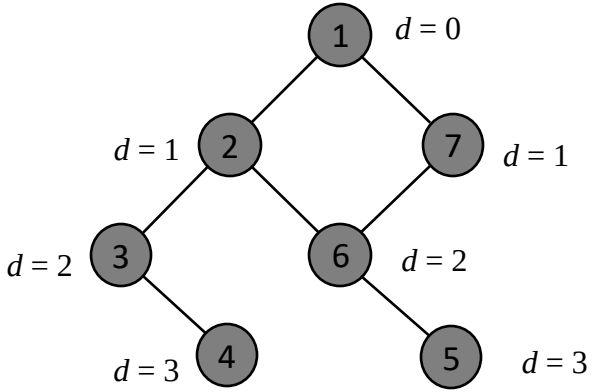
[Iteration 6]

Q = (5: 3)



[Iteration 7]

$Q = \varnothing$



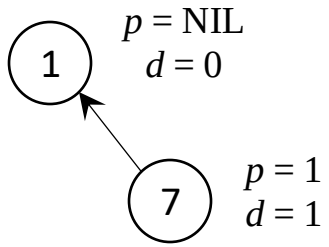
Light gray: vertices in Q

Dark gray: dequeued vertices

BFS algorithm

BFS(G, s)

```
 1  for each vertex u ∈ G.V – {s}
 2      u.d ← ∞ ; u.p ← NIL
 3  s.d ← 0 ; s.p ← NIL
 4  Q ←  create a FIFO queue
 5  ENQUEUE(Q, s)
 6  while Q ≠ ∅
 7      u ← DEQUEUE(Q)
 8      for each v ∈ G.Adj[u]
 9          if v.d = ∞
10              v.d ← u.d + 1
11              v.p ← u
12              ENQUEUE(Q, v)
```



Idea of the algorithm

- Each vertex u has two attributes
  - $u.p$: the parent of u
  - $u.d$: the "distance" from s to u
  - *Unseen* vertex has infinite distance
- Manage vertices in a first-in-first-out (FIFO) **queue** Q
  - Each vertex u (of Q) has been seen and it has finite distance $u.d$
  - Any *unseen* adjacent vertex v has the distance: $u.d + 1$

BFS algorithm

BFS(G, s)

```
 1  for each vertex u ∈ G.V – {s}
 2      u.d ← ∞ ; u.p ← NIL
 3  s.d ← 0 ; s.p ← NIL
 4  Q ←  create a FIFO queue
 5  ENQUEUE(Q, s)
 6  while Q ≠ ∅
 7      u ← DEQUEUE(Q)
 8      for each v ∈ G.Adj[u]
 9          if v.d = ∞
10              v.d ← u.d + 1
11              v.p ← u
12              ENQUEUE(Q, v)
```

Time complexity

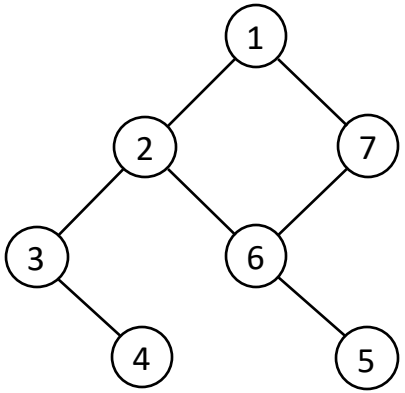
- Lines 1-2: O(|V|) time
- Lines 3-5: O(1) time
- Consider the loop of Lines 6-11
- Total time of Lines 8-9: O(|E|)
  - For each dequeued vertex, we examine its adjacency list once
  - Total size of all adjacency lists: O(|E|)
- Total time of Lines 10-11, 7: O(|V|)
  - Lines 10-12 are executed for v only when $v.d = \infty$
  - Each vertex is enqueued at most once
- Total time: O(|V| + |E|)

Breadth-First Search (BFS)

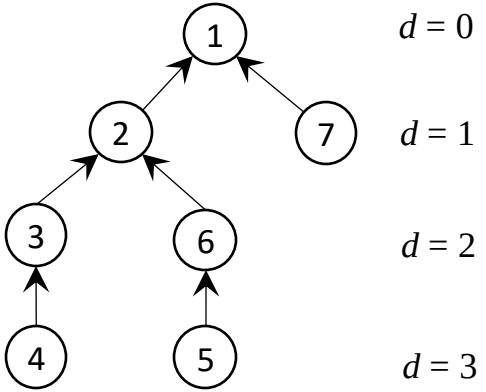
- Breadth-first search (BFS)
  - Given a source vertex s in a graph $G = (V, E)$
  - Visit all vertices closer to s before any vertex farther from s

BFS generates the shortest paths (distance) from s to other nodes in a graph G

It is possible to build different BFS trees of source s in G, but the distance between s and a node u is the same in different BFS trees

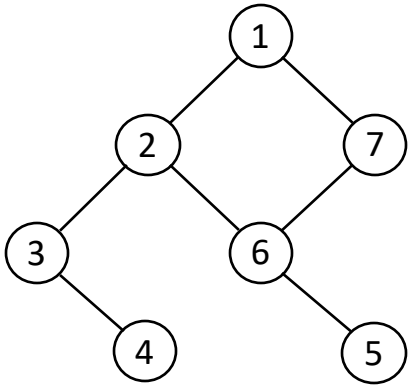


Example: $s = 1$

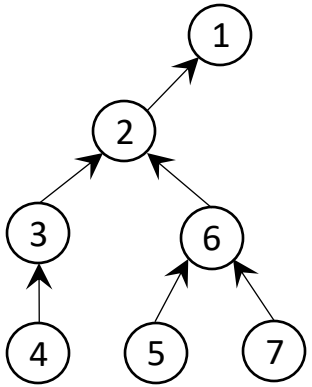


Depth-first search (DFS)

- Depth-first search (DFS)
  - Given a source vertex s in a graph $G = (V, E)$
  - Visit a vertex "deeper" in the graph whenever possible
- It builds a depth-first tree implicitly
  - s is the root
  - u.p is the parent of u
  - u.depth is the tree depth of u



Example: $s = 1$



depth = 0

depth = 1

depth = 2

depth = 3

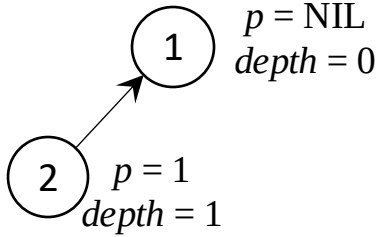
DFS Algorithm

DFS-Main (G, s)

```
1  for each vertex u ∈ G.V
2      u.depth ← ∞
3      u.p ← NIL
4  s.depth ← 0
5  DFS (G,s)
```

DFS (G,u)

```
1  for each v ← G.Adj[u]
2      if  v.depth = ∞
3          v.depth ← u.depth + 1
4          v.p ← u
5          DFS (G,v)
```



Idea of the algorithm

- Each vertex u has two attributes
  - $u.p$: the parent of u
  - $u.depth$: the depth of u
  - *Unseen* vertex has depth as ∞
- Recursion
  - v is an adjacent vertex of u
  - Base case: adjacent vertex v has depth $\neq \infty$, thus v has been seen
  - Recursive case: update the depth of adjacent vertex v to $u.depth + 1$, and then call DFS recursively at v

DFS Algorithm

DFS-Main (G, s)

```
1  for each vertex u ∈ G.V
2      u.depth ← ∞
3      u.p ← NIL
4  s.depth ← 0
5  DFS (G,s)
```

DFS (G,u)

```
1  for each v ← G.Adj[u]
2      if  v.depth = ∞
3          v.depth ← u.depth + 1
4          v.p ← u
5          DFS (G,v)
```

Time complexity

DFS-Main (G, s)

- Lines 1-3: O(|V|) time
- Line 4: O(1) time
- Line 5: ??? time

DFS (G, s)

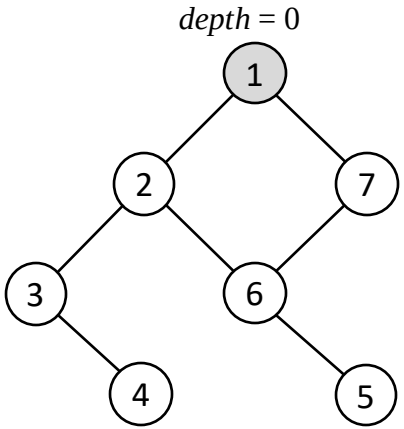
- How many recursive calls of DFS function in G?
  - DFS is called only once for every node when *v.depth* is updated
  - *v.depth* is updated only once for v
  - The number of DFS calls is O(|V|)
- The total cost of Line 3-5 in all recursive calls is O(|V|)
- In the call to DFS(G,u), the adjacency list of u is examined
- For every node u in G, its adjacency list examined once and only once
- For all the adjacency lists of G, how many edges?
  - O(|E|)
- The total cost of Line 1-2 of all recursive calls is O(|E|)
- DFS(G,s) requires O(|V| + |E|)

Example: DFS(G, 1)

We are given the source vertex $s = 1$

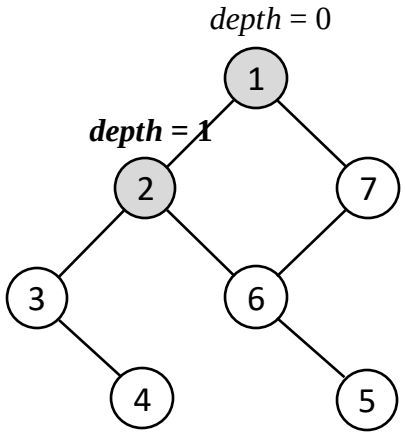
[Step 1]

DFS(*G*, 1)



[Step 2]

DFS(*G*, 1)
→ **DFS(*G*, 2)**

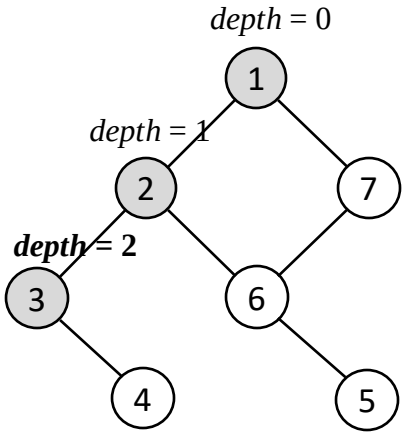


Updated items in bold type

Example: DFS(G, 1)

[Step 3]

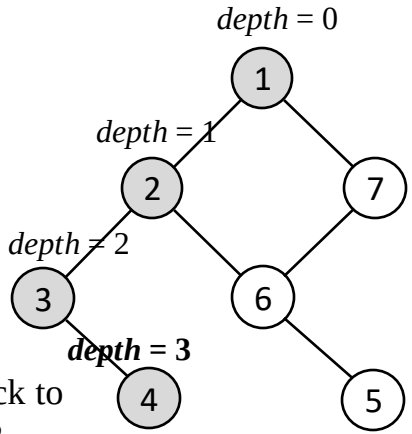
DFS(*G*, 1)
→ DFS(*G*, 2)
→ **DFS(*G*, 3)**



[Step 4]

DFS(*G*, 1)
→ DFS(*G*, 2)
→ DFS(*G*, 3)
→ **DFS(*G*, 4)**

Then we backtrack to vertex 3, vertex 2

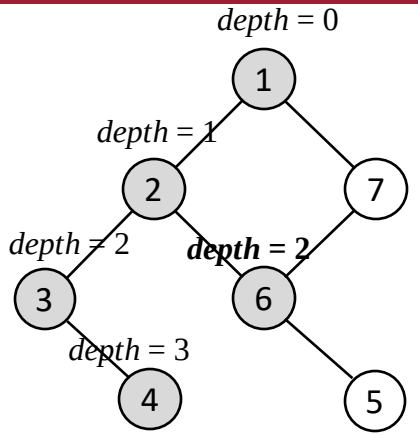


Updated items in bold type

Example: DFS(G, 1)

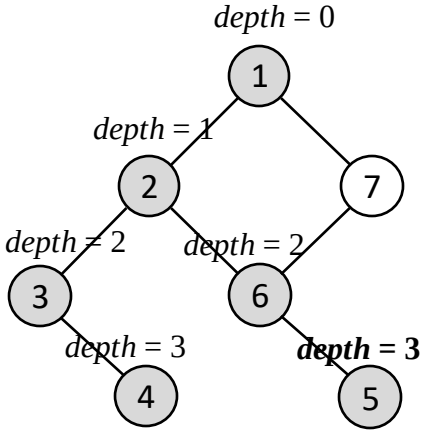
[Step 5]

DFS(*G*, 1)
→ DFS(*G*, 2)
→ **DFS(*G*, 6)**



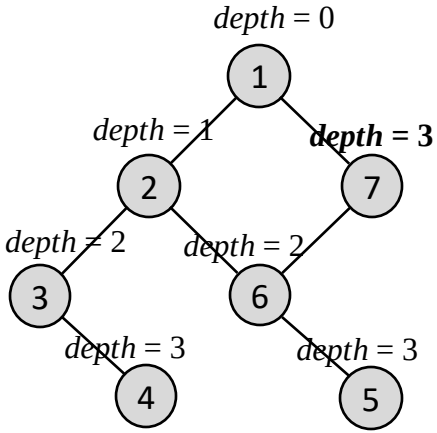
[Step 6]

DFS(*G*, 1)
→ DFS(*G*, 2)
→ DFS(*G*, 6)
→ **DFS(*G*, 5)**



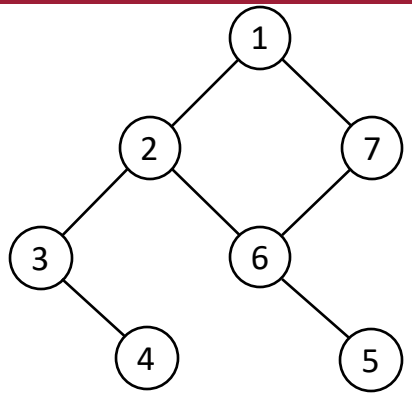
[Step 7]

DFS(*G*, 1)
→ DFS(*G*, 2)
→ DFS(*G*, 6)
→ **DFS(*G*, 7)**

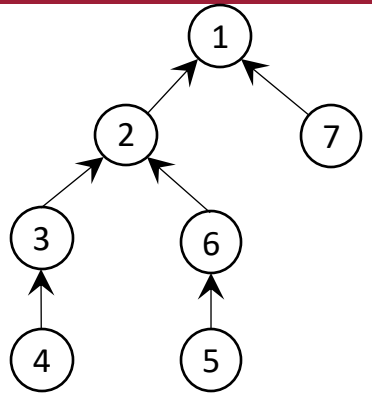


Updated items in bold type

Distance in BFS vs Depth in DFS



Example: $s = 1$



$d = 0$

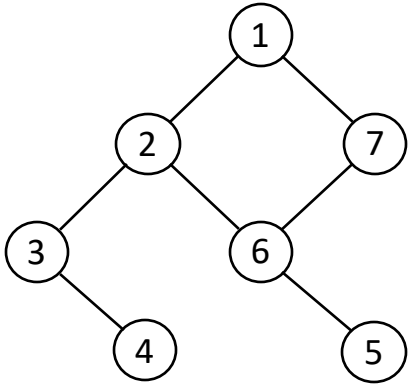
$d = 1$

$d = 2$

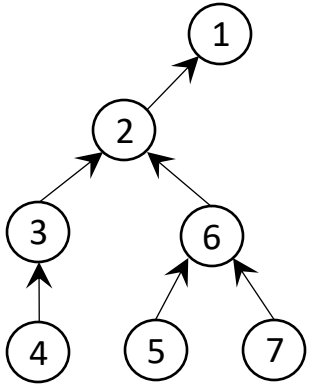
$d = 3$

BFS

Shortest path distance between vertices is independent from BFS.
The depth of a node w.r.t. source vertex s depends on DFS process.



Example: $s = 1$



depth = 0

depth = 1

depth = 2

depth = 3

DFS

Thank you