Lecture 13 (Graphs 2)

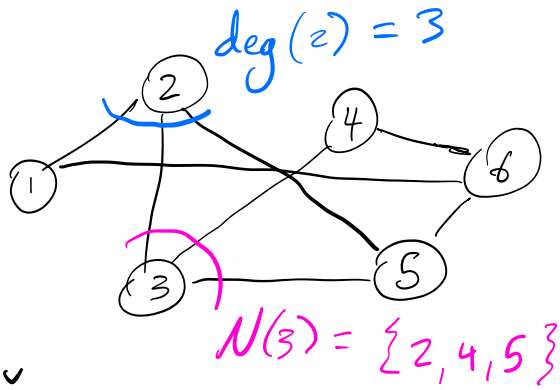
Def (Undirected) $G = (V, E)$

V = set of vertices/nodes

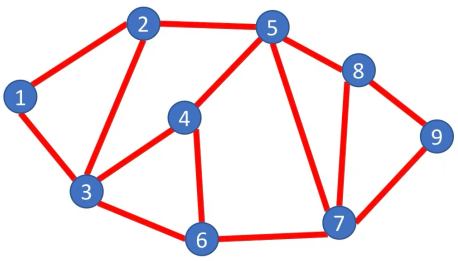
$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{(1,2), (1,6), (4,6), \ldots\}$

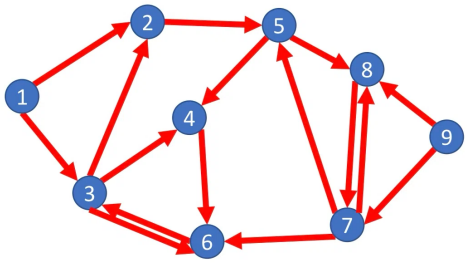
degree: $\deg(v)$ = # of edges touching v.



neighbors/adjacent: $N(v)$ = all nodes connected with single edge to v



Connected



Not (strongly) Connected

But is "weakly" connected.

maximum # of edges?

undirected, simple:

directed, simple:

"dense":

"sparse":

In general, $|E|$ and $|V|$ very different, so runtime should be a function of both $|E|$ and $|V|$

Question: if G connected, min # of edges?

Def | (Tree)

Graph ADT

nodes with edges in between

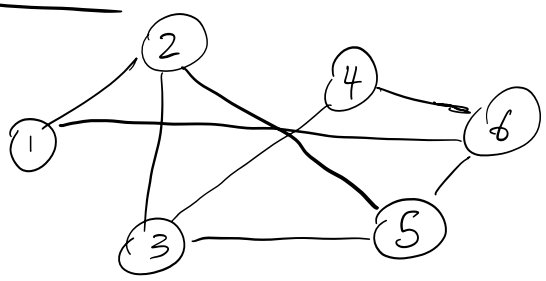
weighted?
directed?
simple?

operations

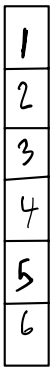
- addEdge
- removeEdge
- exists
- getNeighbors

Data Structures for graphs?

$G = (V, E) =$



Adjacency List



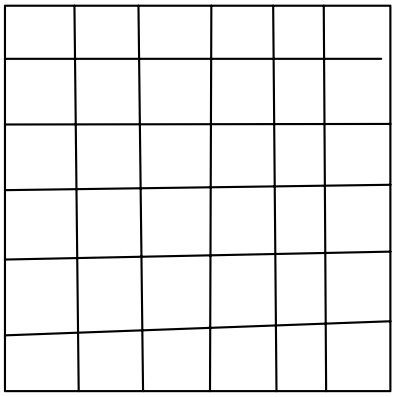
$n := |V| \quad m := |E|$

memory?

Runtimes

- addEdge?
- removeEdge?
- exists?
- getNeighbors?
  - outgoing
  - incoming

Adjacency Matrix



memory?

Runtimes

- addEdge?
- removeEdge?
- exists?
- getNeighbors?
  - outgoing
  - incoming

Adjacency list perhaps most common in practice

many graphs are relatively sparse

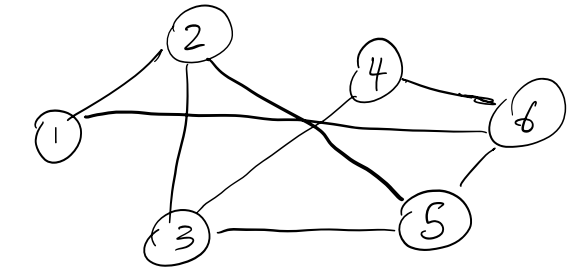
Graph traversals

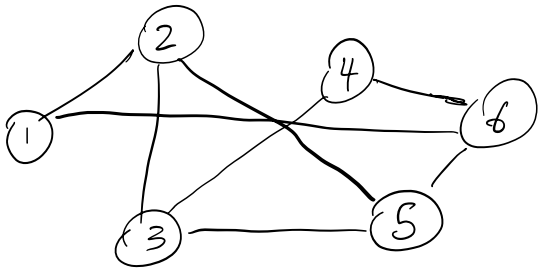
question: given G, how can/should I visit all the vertices?

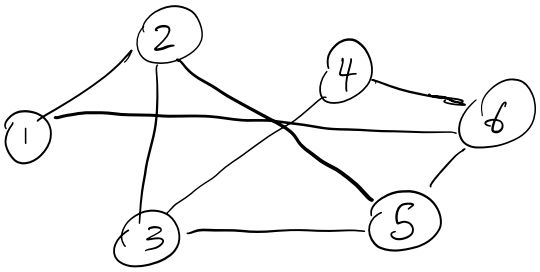
Breadth first Search

Idea: pick source node v

1) process all nodes 1 away from v
   all neighbors
2) process all nodes 2 away from v
   all neighbors' neighbors
3) " " 3 " "
   . . . . .







What can we learn?

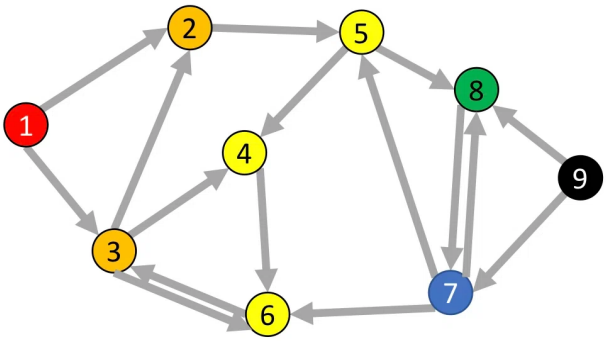
BFS Pseudocode

```
bfs(G, v):
    found = new empty queue
    found.enqueue(v)
    mark v as "visited"
    while found is not empty:
        curr = _____
        for u in _____:
```

```
        if _____:
            found.enqueue(u)
            mark u as "visited"
```

Runtime?

Ex



node	visited?	depth
1	True	0
2		
3		
4		
5		
6		
7		
8		
9		

queue:

Find Distance Pseudocode

```
findDistance(G, s, t)
    found = new empty queue
    layer = 0
    found.enqueue(s)
    mark s as "visited"
    while found is not empty:
        curr = found.dequeue
        layer = _____
        for u in neighbors(curr):
            if u not visited:
                found.enqueue(u)
                mark u as "visited"
                depth(u) = _____
    return depth(t)
```

Shortest (unweighted) path?

Depth First Search

Idea: explore each path as deep as possible:

1) visit start node s
2) visit a neighbor of s, then all nodes reachable from that neighbor, ...
3) repeat with next neighbor

What can we learn from this?

DFS non-recursive Pseudocode

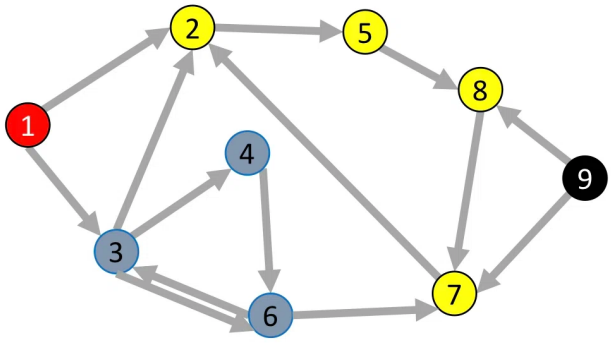
```
dfs(G,s):
    found = new empty stack
    found.push(s)
    mark s as "visited"
    while found is not empty:
        curr = _____
        for u in _____:
            if _____:
                found.p      (u)
                mark u as "visited"
```

runtime?

recursive implementation?

```
dfs(G, curr):
    mark curr as "visited"
    for u in _____:
        if _____:
            dfs(G, u)
    mark curr as "done"
```

Ex]



node	visited?	done?	other...
1	True	0	
2			
3			
4			
5			
6			
7			
8			
9			

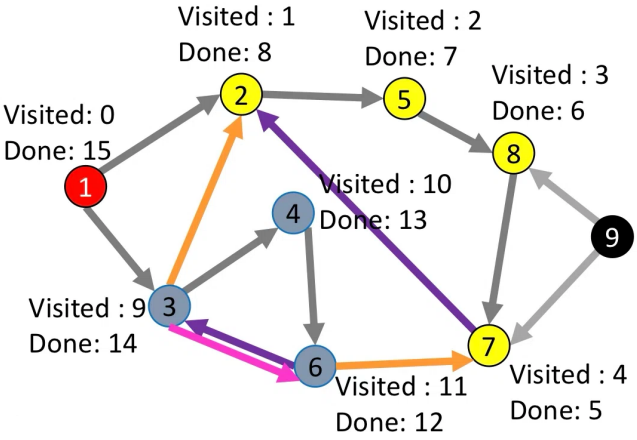
call stack:

Characterizing Edges in DFS

Tree Edges

cross edges $t_{done}(b) < t_{visit}(a)$

forward edges $t_{visit}(a) < t_{visit}(b) < t_{done}(b) < t_{done}(a)$



back edges $t_{visit}(b) < t_{visit}(a) < t_{done}(a) < t_{done}(b)$

current nodes neighbor is "in progress" / reachable

$\Rightarrow$

Cycle Detection

idea: look for a back edge!

```
cycleDetect(G, curr):
    mark curr as "visited"
    cycleFound = False
    for u in neighbors(curr):
        if u visited but not done:
            cycleFound = True
        if u not visited and not cycleFound:
            cycleFound = cycleDetect(G, u)
    mark curr as "done"
    return cycleFound
```

```
hasCycle(G)
    for each node v:
        if v not done:
            if hasCycle(G, v):
                return True
    return False
```