Lecture 14 - More Graphs

Announcements:

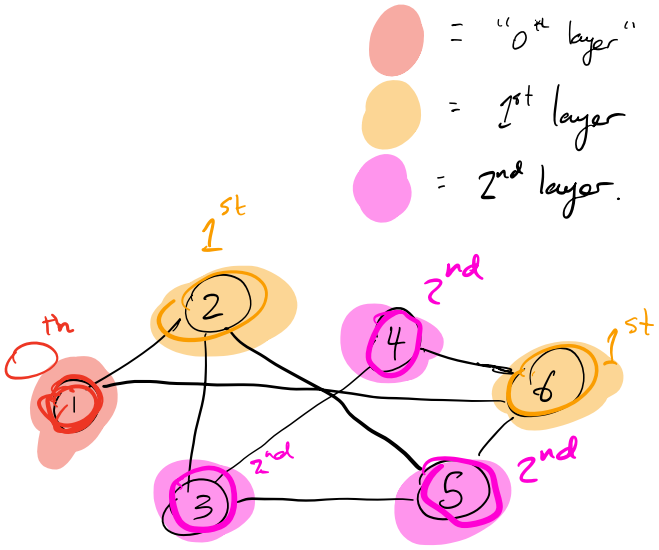
- Sorting Assignment out - start early!
- Midterm grades out probably tomorrow
- Mid-quarter feedback form out tomorrow

  get +2 midterm points for filling out!

Last time: 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 " "
   .....

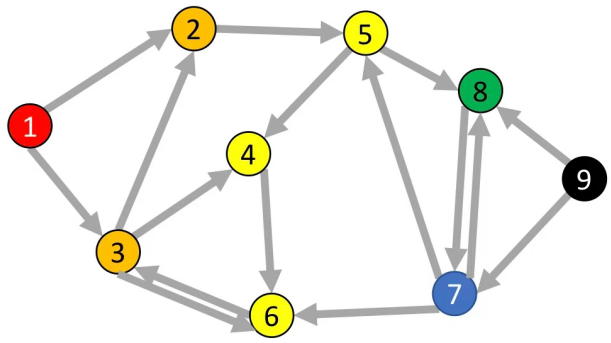


BFS Pseudocode

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

            if u not visited already:
                found.enqueue(u)
                mark u as "visited"
```

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
    found.enqueue(s)
    layer/depth(s) = 0
    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)
```

Idea: when vertex is seen first, record its depth!

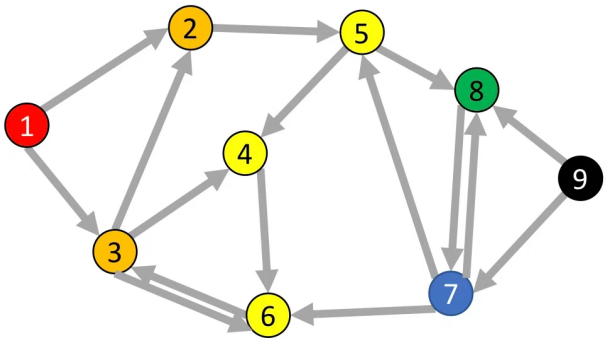
Shortest Path:

We can use a very similar idea to find shortest path from s to t!

idea: instead of (or in addition to) recording depth of v, record v's "previous": the

first u that adds v to the queue.

Ex|



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

queue:

Longest Path?

Traversal #2: 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

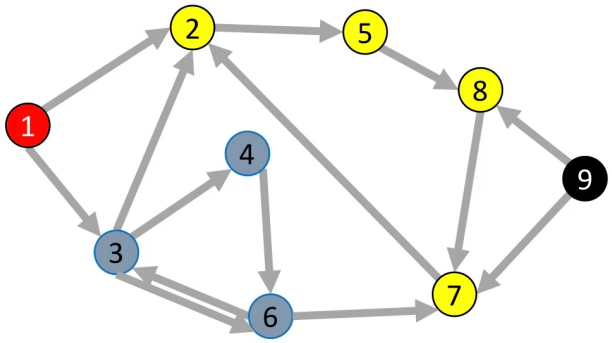
```
dfs(G,s):
    found = new empty stack
    found.push(s)
    mark s as "visited"
    while found is not empty:
        curr = _____
        for u in _____:
            if _____:
                found.p      (u)
                mark u as "visited"
```

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			

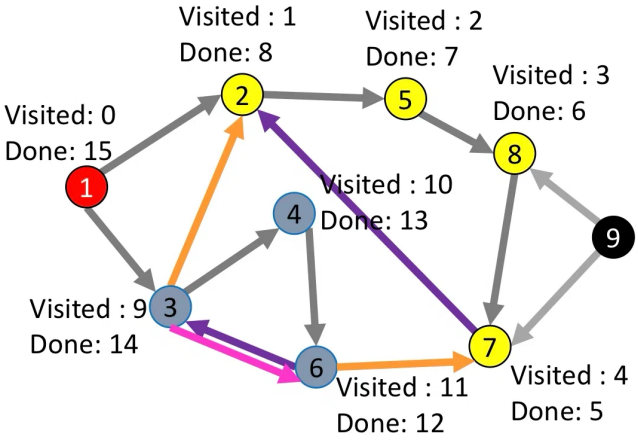
call stack :

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

⟹

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
```

Single Source Shortest Path

Task: given start node s in a weighted graph with non-negative edge weights, find the shortest path from s to all other vertices (or a specific node)

Dijkstra's Algorithm!

idea: when a vertex is closest "not done" node to start, we have found its shortest path!

Dijkstra's Pseudocode

① extract node curr from priority queue
this makes curr "done"

② for each not done neighbor of curr:
- update its distance if we have found a better path.

Seen = added to PQ

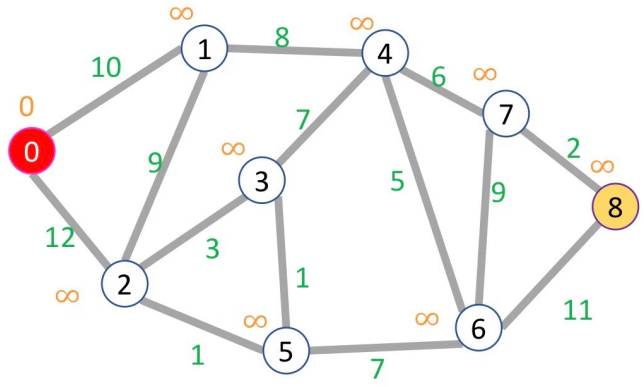
done = removed from PQ

BFS Pseudocode

① extract node curr

② for each not visited neighbor of curr:
add it to queue
depth of neighbor = depth(curr) + 1

Ex!



node	seen?	done?	distance
0	True		0
1			∞
2			∞
3			∞
4			∞
5			∞
6			∞
7			∞
8			∞

PQ: