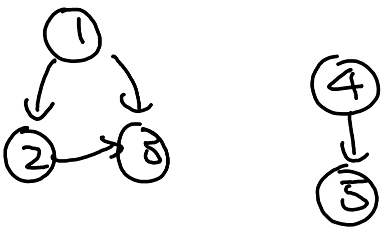
Lecture 10: Graph II

July 13, 2020 9:59 PM

Graph
- $G = (V, E)$ directed / undirected
- represent graph (adj. list / adj. matrix)
- BFS(G, S)
  - V_1 : traversal
  - V_2 - traversal / shortest distance



BFS (G, 1)

- what if want to traverse all vertices?

```
BFS_ALL (G){
  visited [1...n]  // initialized to false
  for (S = 1 ... n){
     if (visited[S] == false){
         BFS (G, S, visited);
     }
  }
}
```

```
BFS(G, s, visited) {
    q.enqueue(s);
    visited[s] = true;
    while (! q.isEmpty()){
        u = q.dequeue();
        for( v in G.neighbors(u)){
            if(! visited[u]){
                q.enqueue(v);
                visited[v] = true;
            }
        }
```

}
}
}

Running time: $O(n+m)$ $O(V+E)$

$O(V)$: marking each vertex as visited ($O(1)$ each)
$O(E)$: for each vertex popped out the queue, need to verify for the neighbors

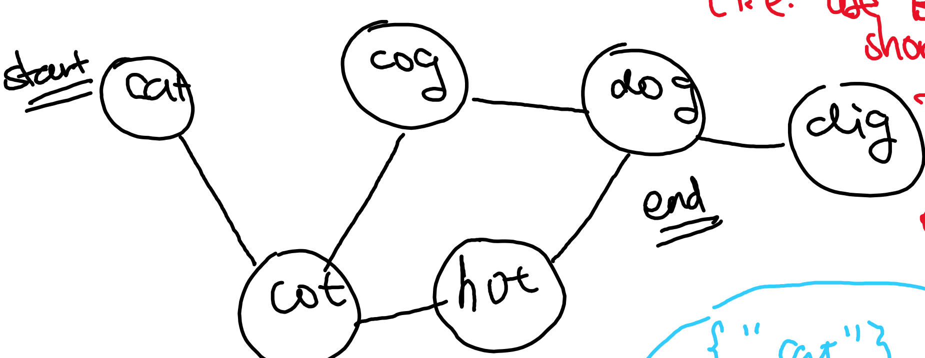
- Word Ladder
  - dictionary
  - <u>start</u> "cat"
    <u>end</u> "dog"

- want a sequence of changes from start to end

{ can only change a single letter
{ intermediate words have to be in dictionary

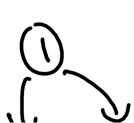
cat ⇒ cot ⇒ cog ⇒ dog
3 changes

Q1: if solution exists?
Q2: if exists, can you show me one solution?
Q3: what is the shortest / best solution?
(i.e. use BFS to find shortest path from start word to end word)

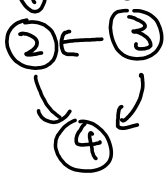


{"cat"}
{"cot"} dist = 1
{"cog", "hot"} d = 2
{"dog"} d = 3

- DFS (depth first search)



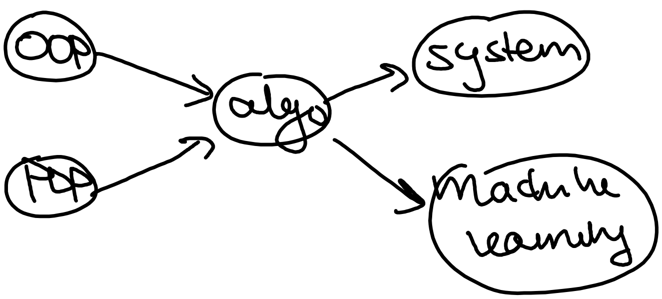
DFS (G, 1)



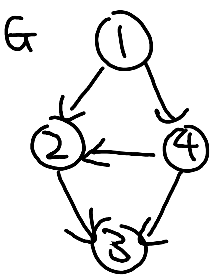
1, 2, 4, 3

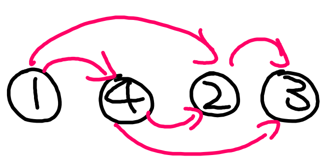
```
DFS(G, u, visited[]){
    visited[u] = true;
    print u; // processing vertex u
    for(v in G.neighbors(u)){
        if(visited[v] == false){
            DFS(G, v, visited);
        }
    }
}
```

<u>DAG</u> : Directed Acyclic Graph



- DAG ⇒ topological sort / order



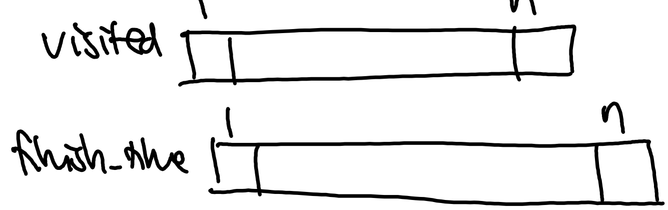


all from left to right

algo 1 (dfs) — easy to write, hard to understand

algo 2 (bfs) — hard to write, easy to understand

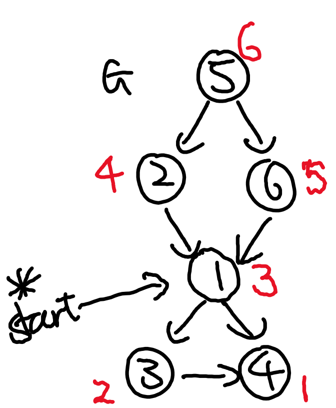
- Algo 1




```
dfs(G,u){
    visited[u] = true;
    for( v in G.neighbors(u)){
        if(visited[v] == false){
            dfs(G, v);
        }
    }
    finish_time[u] = time;
    time += 1;
}
```

postvisit

```
dfs_all(G){
    visited[1...n] // initialized to false
    time = 1;
    finish[1...n];
    for(u = 1...n){
        if(visited[u] == false){
            dfs(G,u);
        }
    }
}
```



Sort by decreasing order of finishing time:

5 → 6 → 2 → 1 → 3 → 4

Summary

- dfs_all to get finish time $O(V+E)$
- sort by finish time decreasing order $O(V \log V)$

- Sort by finish time in decreasing order $O(V\lg V)$

$O(V\lg V + E)$

$\downarrow$

which one dominates depends on the number of edges;

- if graph is really dense, $O(E)$ dominates.
- if graph is sparse, $O(V\lg V)$ dominates.

- Without sorting at the end:

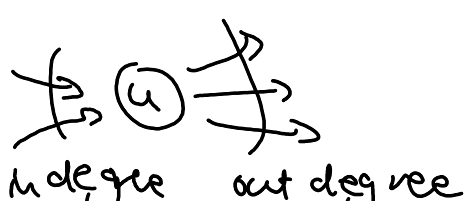
```
dfs(G, u, visited, l){
  visited[u]=true;
  for(v in G.neighbo(u)){
     if(visited[v] == false){
        dfs(G, v, visited, l)
     }
  }
  l.append(u);
}


top_sort(G){
  l=[]
  visited[1...n];
  for(u=1...n){
     if(visited[u] == false){
        dfs(G, u, visited, l);
     }
  }
  return l.reverse();
}
```

Running time:

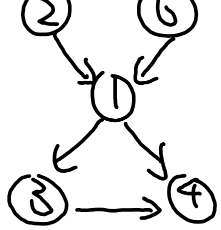
$O(V+E)$

space: $O(V)$

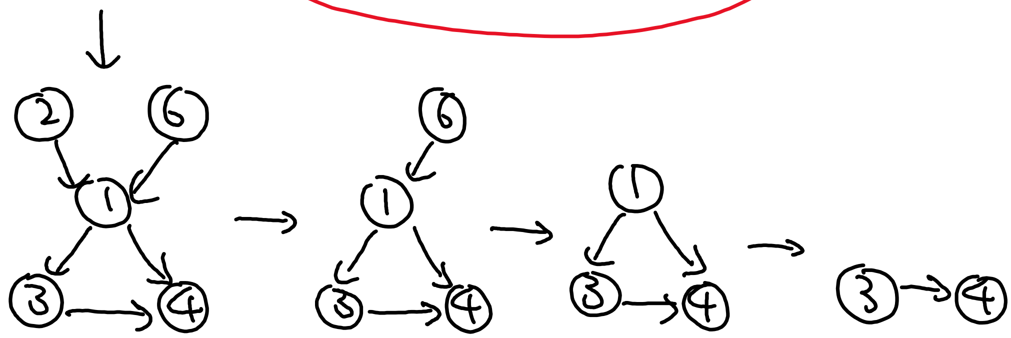
- Algo2

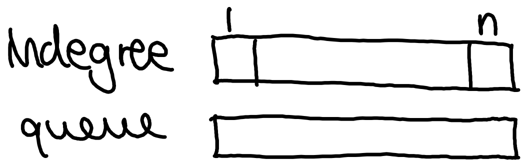


G



5, 2, 6, 1, 3, 4





G is given in adj. list

(1)
```
for (u=1 ... n){
    for (v in G.neighbors(u)) {
        indegree[v]++;
    }
}
```
O(V+E)

(2)
```
for (u=1 ... n){
    if (indegree[u]==0){
        q.enqueue(u);
    }
}
```
O(V)

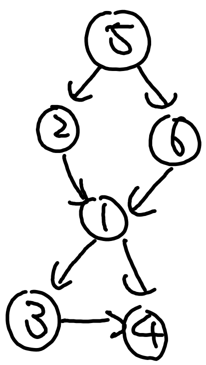
(3)
```
while ( q.empty() == false){
    u=q.dequeue();
    for(v in G.neighbors(u)){
        indegree[v]--;
        if (indegree[v] = =0 ){
            q.enqueue(v);
        }
    }
}
```
O(V+E)

}

Overall: $O(V+E)$

Space: $O(V)$

- Walkthrough



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

queue: | ~~5~~ |

	1	2	3	4	5	6
indegree	2	0	1	2	~~0~~	0

queue: | ~~2~~ | 6 |

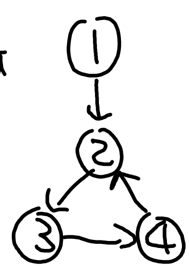
	1	2	3	4	5	6
indegree	1	~~0~~	1	2	~~0~~	0

queue: | ~~6~~ |

• • •

- Cycle detection
  - algo1 (BFS) the "same" as topological sort
  - algo2 (DFS)

① G



	1	2	3	4
indegree	0	2	1	1

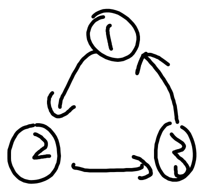
queue{~~1~~}

	1	2	3	4
indegree	~~0~~	1	1	1

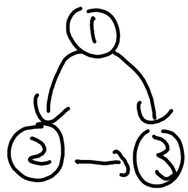
terminates early;
the remaining part form a cycle.

②
```
dfs(G, u){
    visited[u] = true;
    for(v in G.neighbors(u)){
        if(visited[v] == false){
```

```
        dfs(G, v);
      }else{
        // there is a cycle?
      }
    }
}
```



works



doesn't work

<u>color</u> white : vertex not visited
gray : vertex visited, but not complete yet
black : vertex visited and completed.

```
dfs(G, u){  cycle = false;
  color[u] = gray;
  for(v in G.neighbors(u)){
    if (color[v] == white){
      cycle = cycle || dfs(G, v);
    }else if (color[v] == gray){
      return true;
    }
  }
  color[u] = black;
  return cycle;
}
```

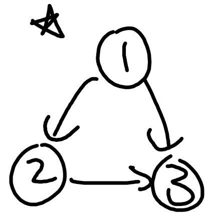
1	2	3
W	W	W

1	2	3
G	W	W

1	2	3
G	G	W

1	2	3
G	G	G

return true;



1	2	3
W	W	W

1	2	3
G	W	W

1	2	3
G	G	W

1	2	3
G	G	G

1	2	3
G	G	B

1	2	3
G	B	B

1	2	3
G	B	B

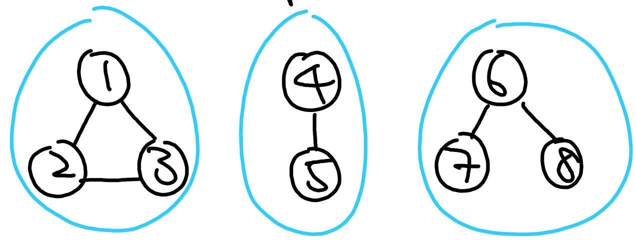
neither white nor gray;

1	2	3
B	B	B

return false;

```
dfs_all(G){
  color[1...n] to white
  for(u=1...n){
    if(color[u]==white){
      dfs(G,u);
    }
  }
}
```

• Connected component in undirected graph connected component in undirected graph



$u, v \in C.C \iff \exists\ u \rightsquigarrow v$
$v \rightsquigarrow u$

if vertices u and v are within the same connected component, then there exists a path from u to v, and from v to u (because of symmetry in undirected graphs)

Given a graph: How many C.C.?
For each vertex, which C.C. does it belong to?

*Same for DFS

```
bfs_all (G){
    cc = 0;
    visited [1...n];
    for (u = 1...n){
        if (visited[u] == false){
            BFS (G, u);
            cc += 1;
        }
    }
    return cc;
}
```

e.g. Leetcode 200

Number of Islands

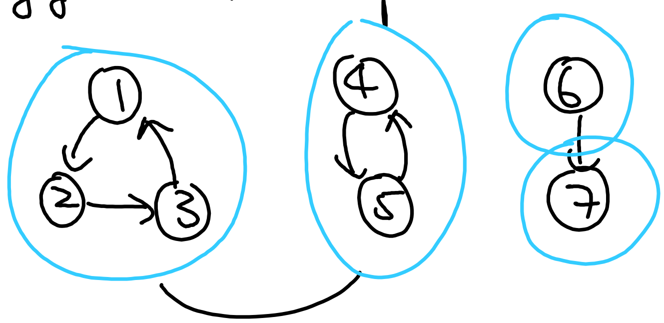
0	1	0	0
0	1	0	0
0	0	1	0
1	1	0	0

```
procedure dfs (G)
for all v ∈ V:
    visited (v) = false;
cc = 0;
for all v ∈ V:
```

```
procedure explore (G, v)

visited (v) = true;
ccnum[v] = cc;
for each edge (v, u) ∈ E:
```

```
if(not visited(v)) {          if not visited(u): explore(u);
    cc++;                     postvisit(v);
    explore(v);
  }
}
```

- Strongly Connected Components (directed graph)

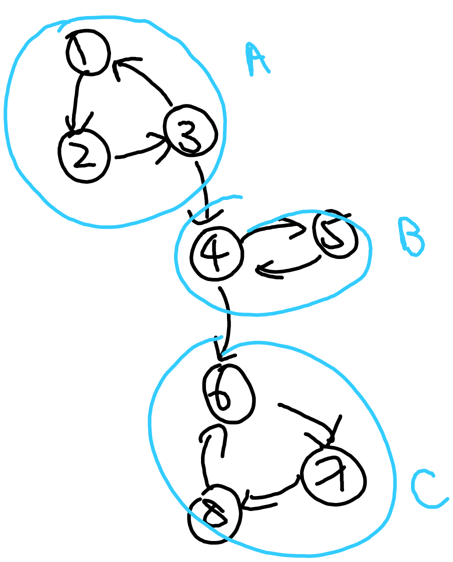


For any vertices u and v, there is a path between u to v and v to u.

A single vertex is also a strongly connected components.

<u>weakly connected components</u>

- u ⇝ v or v ⇝ u
- Three



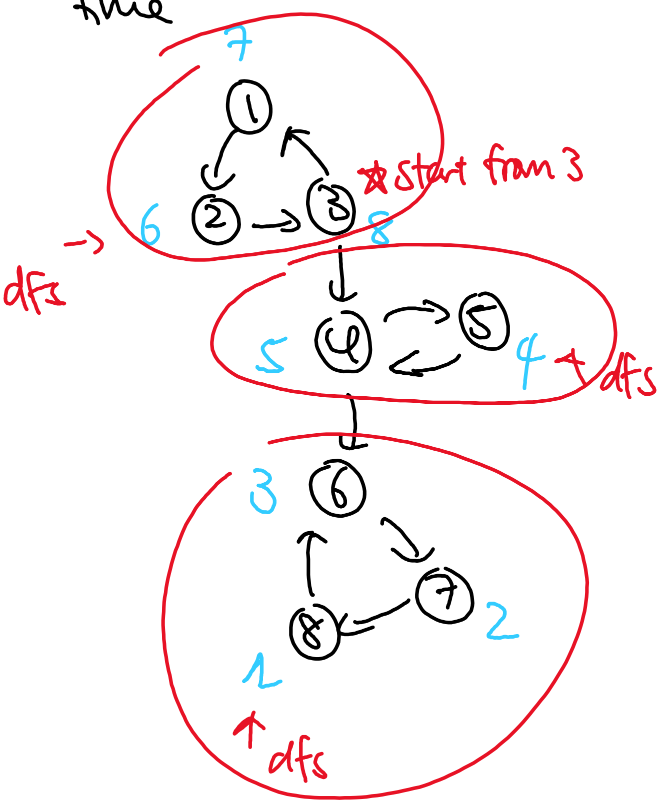
A, B, C form a "DAG"

dfs (7) → component C
dfs (5) → Component B
dfs (2) → Component A

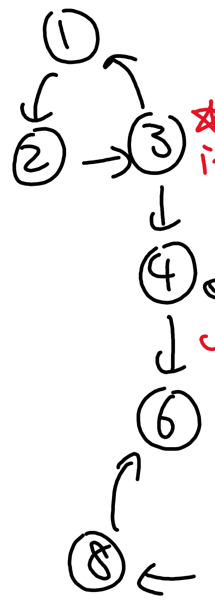
in terms of finishing time:

$C < B < A$

- run dfs with finishing time on G
- Go over vertices using increasing order of finishing time



But:

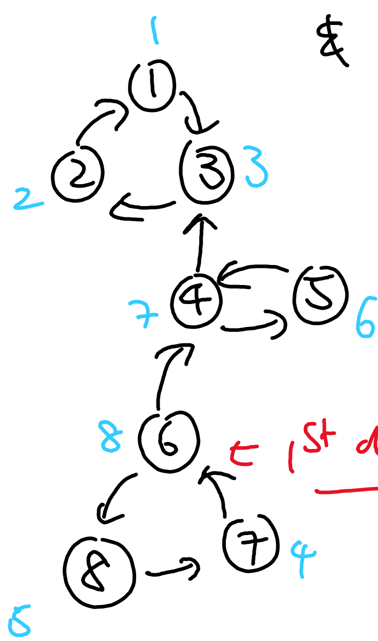


start

if we go to (1) instead of (4) we would conclude wrongly that there is only one SCC.

- Solution: dfs with finishing time in G^{-1} & go over vertices in decreasing order of finishing time

G^{-1}



← 1st dfs on The ORIGINAL GRAPh

Sink component on G becomes source component on G^{-1}

link component of a becomes sub a component of a