07-13

Monday, July 13, 2020 6:59 PM

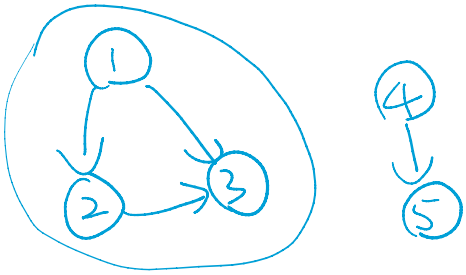
Graph { $G=(V,E)$ directed / undirected

represent graph [adj. list / adj. matrix]

BFS(G, s)

- v1: traversal
- v2: traversal / shortest distance.

O(m)




BFS(G, 1)

what if want traverse all vertices

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

$0 \le m \le n(n-1)$

#1 O(n)

#2 O(n+m) O(V+E)

G=(V,E)

Word Ladder

- dictionary

word ladder:

- dictionary
- start "cat"
  end "dog"
- want a seq changes from start to end
  { can only change single letter
  intermediate words have to be in dictionary

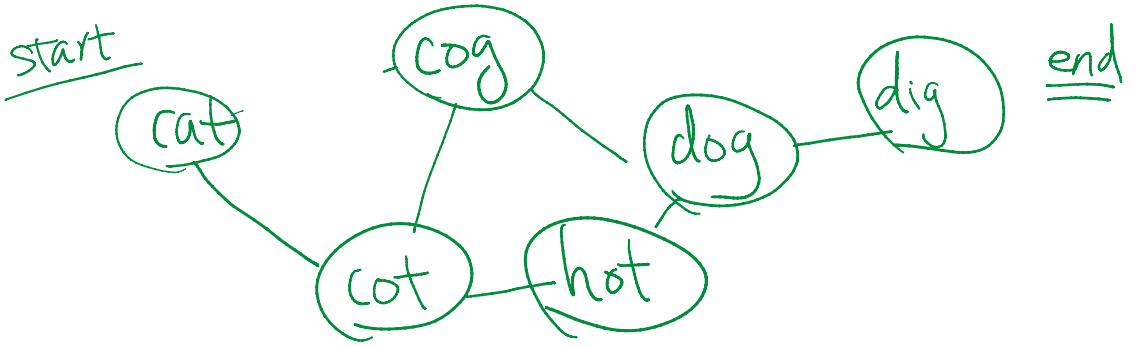
cat ⇒ cot ⇒ cog ⇒ dog

3 changes

Q1: if soln exists??

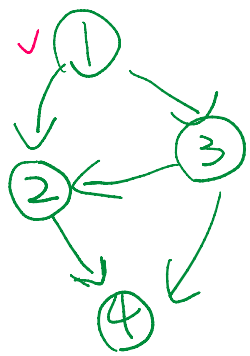
Q2: if exists, can you show me (soln

Q3: what is shortest/best soln




DFS (depth first search)

DFS (deep first ...)

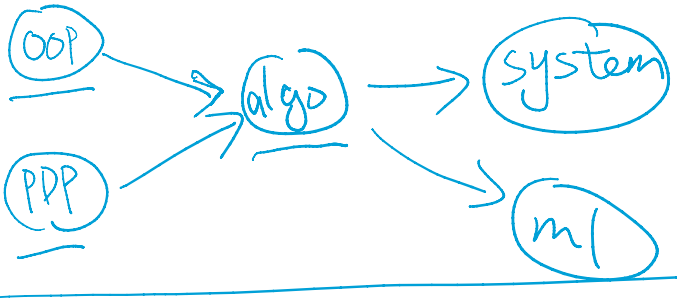



DFS(G, 1)

1, 2, 4, 3

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

DAG: Directed Acyclic Graph



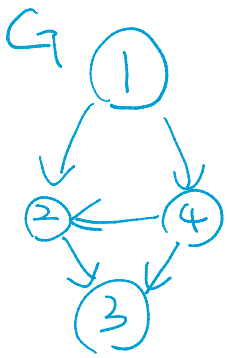

DAG

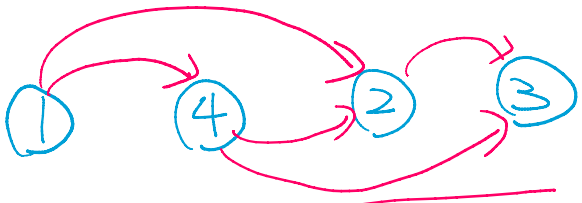

seq build

make
cmake

# seq build

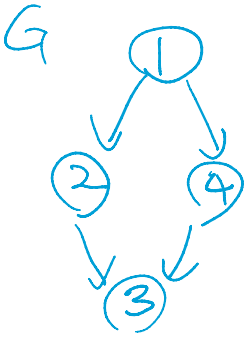
## DAG ⇒ topological sort / order




from left to right


×




1, 2, 4, 3

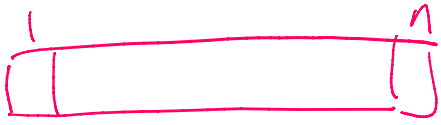
1, 4, 2, 3

algo 1 [dfs]: easy to write / hard to understand

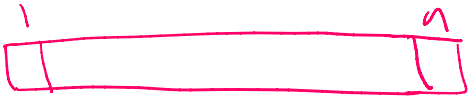
algo 2 [bfs]: hard to write / easy to understand

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

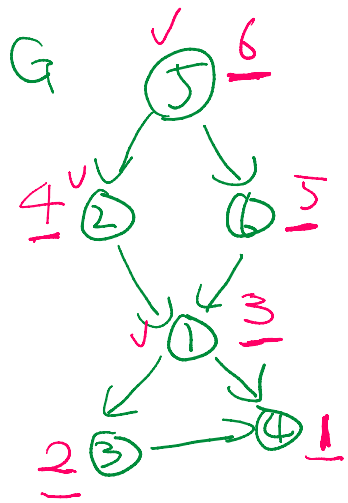
visited




finish_time



```
}
finish_time[u] = time;  time += 1;
}

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




sort by finish_time dec:

5, 6, 2, 1, 3, 4

algo 1: ~~O(n+m)~~ dfs_all to get finish_time

O(nlgn) sort by finish_time.

O(nlgn + m)

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

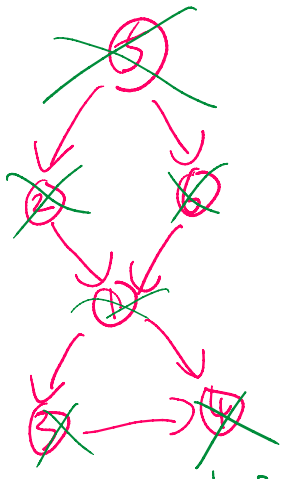
```
        if (visited[v] == false)
            dfs(G, v, visited, l)
    }
    l.append(u);
}

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

time $O(n+m)$

space $O(n)$

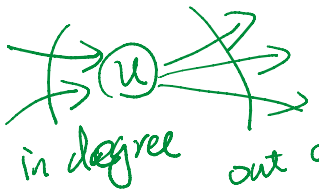
---

G




in degree

out degree




5, 2, 6, 1, 3, 4

$O(n+m)$

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

(1)

```
for (u = 1...n) {
    for (v in G.neighbors(u))
```

indegree 0 1 0 1 0 2 0 1 0 1

queue 5 2 6 1 3 4

$O(n+m)$

```
for (v in G.neighbors(u))
    indegree[v] += 1;
```

G is given in adj. list

(2) $O(n)$

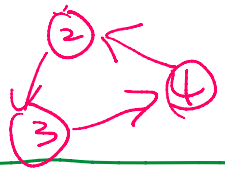
```
for (u = 1, ..., n) {
    if (indegree[u] == 0)
        q.enqueue(u);
}
```

(3) $O(n+m)$

```
while (q.empty() == false) {
    u = q.dequeue();
    for (v in G.neighbor(u)) {
        indegree[v] -= 1;
        if (indegree[v] == 0) {
            q.enqueue(v);
        }
    }
}
```

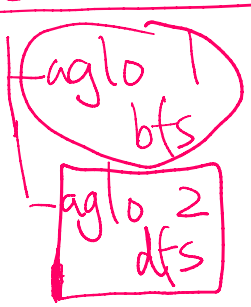
r.t. $O(n+m)$

space $O(n)$




(A) dfs (B) bfs (C) both (D) neither

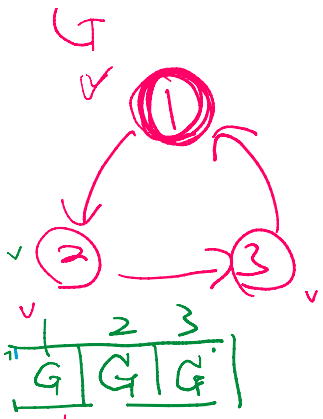
Cycle detection — DAG

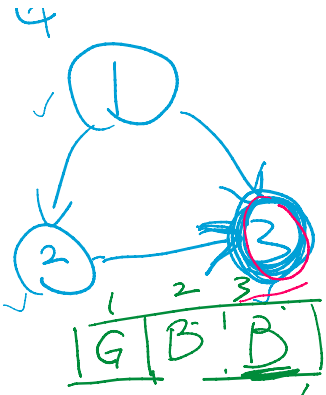



[the same topological sort]

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

G G




```
for (v in G.neighbors(u)) {
    if (visited[v] == false)
        dfs(G, v)
    else
        "there is cycle!!!"
}
```

color  W: vertex not visited

G: vertex visited, but not complete yet

B: vertex visited and completed

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

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

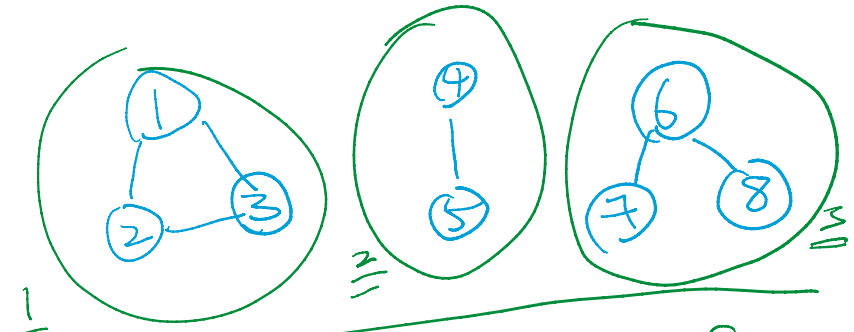
---

(A) bfs  (B) dfs  (C) both  (D) neither

---

connected component in undirected graph

# connected component in undirected graph




$u, v \in C.C.$

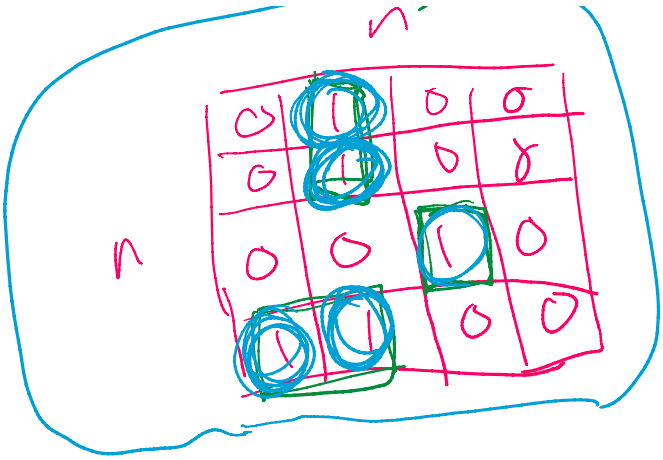
$\Leftrightarrow \exists\ u \rightsquigarrow v$
$v \rightsquigarrow u$

G

How many C.C.?

BFS(G, u)

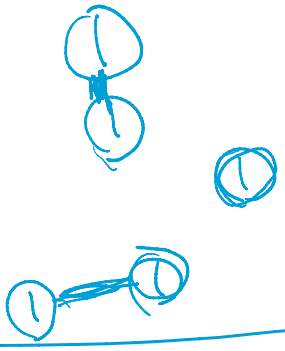
```
BFS_ALL(G) {
    cc = 0;
    visited[1...n];
    for (u = 1....n) {
        if (visited[u] == false) {
            BFS(G, u);
            cc += 1;
        }
    }
    return cc;
}
```



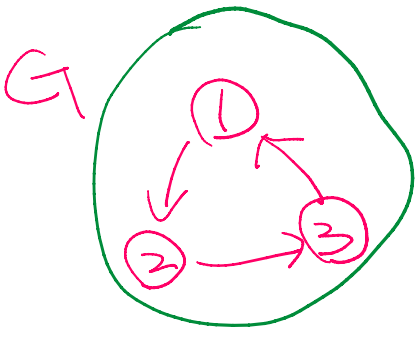

0 / 1

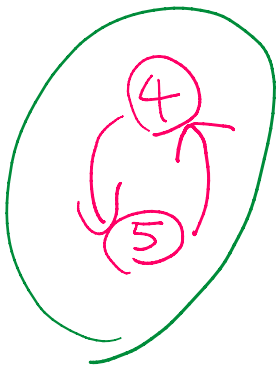
water land

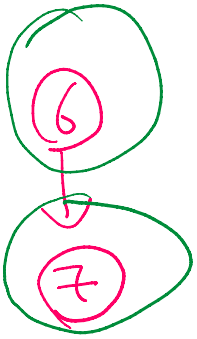
# of islands



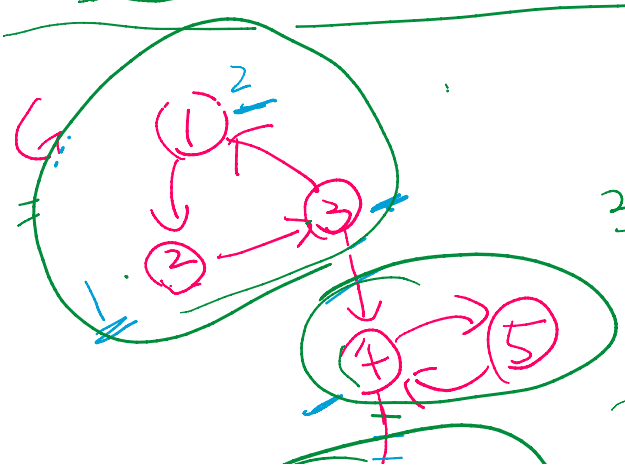
Strongly connected component [directed graph]

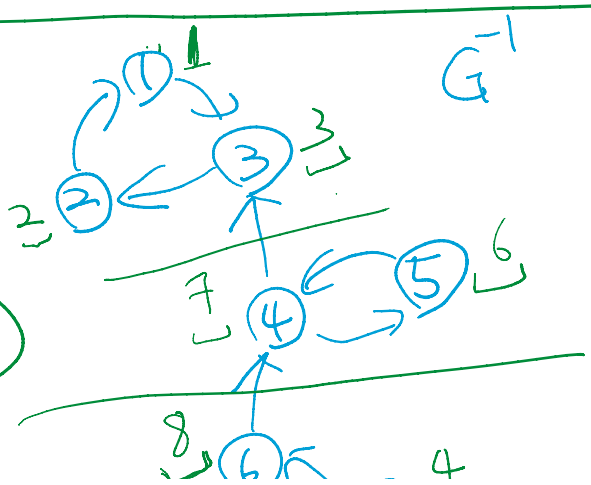


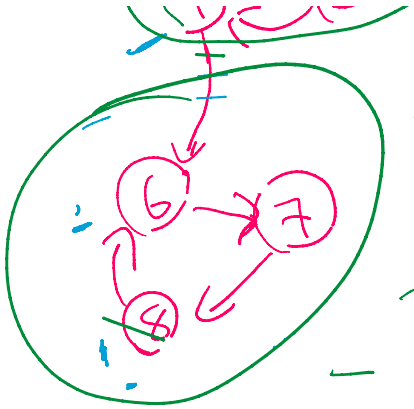



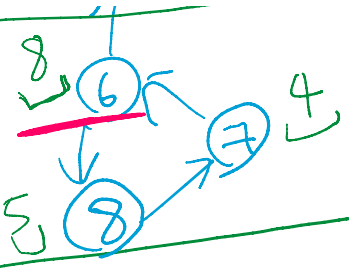




{1, 2, 3}  {4, 5}  {6}  {7}






- dfs w/ finish_time in $G^T$

⇒ go over vertex ~~inc~~ dec of finish_time