Graph

DFS @ Graph

Find Eventual Safe States (directed) `dfs` `detect cycle`
Is Graph Bipartite? `dfs` `detect odd cycle`
Is Bipartition Possible? `dfs` `detect odd cycle`
Reconstruct Itinerary `eulerian path` `dfs, post-order`

BFS @ Graph

Keys and Rooms `bfs [initial room]`
Minimum Height Trees `bfs [leaf nodes]`
K-Similar Strings (K swaps)
Clone Graph `old2new`

shortest path

Evaluate Division `all-pairs`
Network Delay Time `Dijkstra Algorithm`
Cheapest Flights Within K Stops `Bellman_Ford_Within_(K+1)`
The Maze I (hasPath) `bfs` `rolling ball`
The Maze II (shortest distance)
The Maze III (shortest path to hole)

Topological Sort (dfs: postorder) (bfs: indegree)

Detect Cycle in Directed Graph `any cycle` `dfs: visited=0,1,2`
Course Schedule I `any cycle` `dfs: visited=0,1,2`
Course Schedule II `any topsort` `dfs: visited=0,1,2`
Alien Dictionary `any topsort` `dfs: visited=0,1,2`
Sequence Reconstruction **`unique topsort`** `bfs: [0-indegree]`

以下 Notes 总结于 Graph 系列视频

- **DFS** `O(V+E)`

- find path, detect (odd) cycle `coloring`
- find strongly connected components, bridges, articulation points `low-link`
- topology sort, eulerian path `dfs + post-order`

```
def dfs(at):
    visited[at] = 1  # 1 表示开始访问
    for to in E[at]:
        if visited[to] == 0 : dfs(to)
        elif visited[to] == 1: # do something
        elif visited[to] == 2: # do something
    visited[at] = 2  # 2 表示结束访问
```

- **BFS** `O(V+E)`
  - `pushed` is entry door for bfs queue, it records whatever went in
  - `poped` is exit door for bfs queue, it records whatever went out

```
""" 使用 pushed """
bfs = deque([u])
pushed[u] = True
while bfs:
    at = bfs.popleft()
    for to in E[at]:
        if not pushed[to]:
            bfs.append(to)
            pushed[to] = True

""" 使用 poped """
bfs = collections.deque([u])
while bfs:
    at = bfs.popleft()
    poped[at] = True
    for to in E[at]:
        if not poped[to]:
            bfs.append(to)
```

- **Eulerian Path/Circuit (All Edges Once)**
  - Existence `Connected Component` `Ignore Singletons`
    - Circuit @ Undirected: even degree; @ Directed: indegree == outdegree
    - Path @ Undirected: zero or two odd degree; @ Directed: at most one out-in==1, at most one in-

out==1, rest in==out

  - Path
    - `dfs + post-order`, similar to `TopoSort`
    - `def dfs(at): while G[at]: dfs(G[at].pop(0)); path.append(at)`

- **Topological Sort on DAG**
  - `DFS + post-order`
  - `BFS + queue[0-indegree nodes]`
- **Single Source Shortest/Longest Path on DAG**
  - Single Source Shortest Path on DAG: `Topological Sort + Edge Relaxation`
  - Single Source Longest Path on DAG: `negate edge weights`
- **Single Source Shortest Path on Unweighted Graph**
  - BFS
- **Single Source Shortest Path on Weighted Graph**
  - Dijkstra's `O((E+V)logV)` `non-negative edge weights`
  - Bellman-Ford `O(VE)` `has negative edge weights`

```
dist[u] # shortest distance from s to u
prev[u] # previous node from s to u

def edge_relax(s -> u -> v):
    dist_sv = dist[u] + E[u][v]
    relax = dist_sv < dist[v]
    if relax: dist[v], prev[v] = dist_sv, u
    return relax

# def Dijkstra(V, E, s) # assert nonnegative(E)
todo = [(0, s)]
done = [False ...]
while todo:
    minVal, u = heappop(todo)
    if dist[u] < minVal: continue       # ignore u
    done[u] = True
    for v in E[u]:
        if not done[v]:
            if edge_relax(s -> u -> v):
                heappush(todo, (dist[v], v))  # lazy update, duplicate nodes

# def Bellman_Ford(V, E, s)
for k in range(|V|-1): for (u, v) in E: edge_relax(s -> u -> v)
# detect all the nodes affected by negative cycle
for k in range(|V|-1): for (u, v) in E: if edge_relax(s -> u -> v): dist[v] = -inf

# def Bellman_Ford_Within_K(V, E, s)
for k in range(K):
   old_dist = dist.copy()
   for (u, v) in E: D[v] = min(dist[v], old_dist[u] + E[u][v])
```

- **All Pairs Shortest Path on Weighted Graph**
    - Floyd-Warshall `O(V^3)` `DP: dist(i, j, range(k))`

```
dist[i][j] # shortest distance from i to j
next[i][j] # next node from i to j

def edge_relax(i -> k -> j):
    dist_ij = dist[i][k] + E[k][j]
    relax = dist_ij < dist[i][j]
    if relax: dist[i][j], next[i][j] = dist_ij, next[i][k]
    return relax

# def Floyd_Warshal(dist)
for k in V: for (i, j) in E: edge_relax(i -> k -> j)
# detect all the nodes affected by negative cycle
for k in V: for (i, j) in E: if edge_relax(i -> k -> j): dist[i][j] = -inf
```

- **Minimum Spanning Tree**
  - Kruskal's `sort edges` `union find`
  - Prim's `similar to Dijkstra's`
- **Bridges and Articulation Points (Undirected Graph)**
  - bridge condition: `id[u] < low[v]`
  - articulation point condition:
    - starting node `outEdgeCount >= 2`
    - other nodes: `id[u] < low[v]` (bridge) or `id[u] == low[v]` (cycle)
- **Strongly Connected Components (Directed Graph)**
  - self-contained cycles
  - node started a connected component: `id[u] == low[u]`

```
ID, id, low, visited = [0], [0 ...], [0 ...], [0 ...]
bridges = []
isArt = [False ...]
sccCount = 0

"""Find Bridges"""
for at in V: if not visited[at]: dfs(at, -1)
def dfs(at, parent):
    visited[at] = 1
    id[at] = low[at] = ID[0] = ID[0] + 1
    for to in E[at]:
        if to == parent: continue
```

```
        if not visited[to]:
            dfs(to, at)
            low[at] = min(low[at], low[to])
            if id[at] < low[to]: bridges.append((at, to))
        else:
            low[at] = min(low[at], low[to])

"""Find Articulation Point"""
for at in V: if not visited[at]:
    outEdgeCount = [0]
    dfs(at, at, -1)
    isArt[at] = outEdgeCount[0] >= 2
def dfs(root, at, parent):
    if parent == root: outEdgeCount[0] += 1
    visited[at] = 1
    id[at] = low[at] = ID[0] = ID[0] + 1
    for to in E[at]:
        if to == parent: continue
        if not visited[to]:
            dfs(root, to, at)
            low[at] = min(low[at], low[to])
            if id[at] <= low[to]: isArt[at] = True
        else:
            low[at] = min(low[at], low[to])

"""Find SCC"""
stack, onStack = [], [false ...]
for at in V: if not visited[at]: dfs(at)
def dfs(at):
    visited[at] = 1
    id[at] = low[at] = ID[0] = ID[0] + 1
    stack.append(at); onStack[at] = True
    for to in E[at]:
        if not visited[to]: dfs(to)
        if onStack[to]: low[at] = min(low[at], low[to])
    if id[at] == low[at]:
        sccCount += 1
        while True:
            node = stack.pop(); onStack[node] = False; low[node] = id[at]
            if node == at: break
```