Date 2025 04. 30

* AS09 - EC Assignment

ELEMENTARY GRAPH ALGORITHMS

Search → Explore the graph by traveling over the edges to discover the nodes

Why? - To discover something about the graph struture

Tonight: BFS & DFS (two different things)

V - the set of vertices (nodes) and/or $|V|$

E - the set of edges and/or $|E|$

Examples:

$O(VE)$ $O(V+E)$

BREADTH-FIRST SEARCH

Discovers all nodes at distance k before discovering any nodes at distance k+1

- Named this way because expands frontier of discovered nodes uniformily
- Given a graph G, and a source vertex $s \in V$ explore the edges of G to discover every node reachable from S.
    - (if the graph is connected, then this will discover every other node)

→ Computes distance (# of edges)

→ Generates a BFS-Tree that stores shortest path of any node from S.

(work on directed or undirected graphs, no weights)

DATA INSIDE NODES

- Each node will be colored to record its status
- White: Undiscovered
- Grey: In progress
- Black: Complete

* Initally all white

Date

- A vertex is discovered the first time becomes encountered which turns it non-white
  (black = itself and all adjacent nodes have been discovered)
- BFS constructs a BFS-Tree, initally only contains the source nodes.

BFS-T
- Whenever a white vertex v is discovered, the vertex v and edge (u,v) are added to the tree. u is the parent of v.

To Implement BFS-T
↳ Need to store:
- For each vertex
  - Color
  - Parent (aka predecessor node)
  - distance (from s)
- Queue

Graph
- Could be:
  - Custom class?
  - <u>Adjacency List (or matrix)</u>
  - Maze content nxn block of values 0-15

"start"

For each vertex u except for the starting node

```
BFS (graph G, node s) {        // O(V + E) ← Running Time
    // initialization O(V)
    for each vertex u ∈ V - {s}        (or all nodes)
init.   color [u] = white
        p [u] = ∅                      // parent (u) - null   - flag value
        d [u] = ∞                      // distance (u) - flag value
    // initialization of the node we will search from (root)
init.   color [s] = gray
source  d [s] = 0
        p [s] = ∅                      // or null or -1  some flag that there is
        Q.insert (s)                                     no parent
    // main section O(E)
    while (! Q.isEmpty ()) {
        u = Q.deque ()
        for each node v ∈ Adjacent [u] {
            if (color [v] == white) {          // non-discovered node
                color [v] = gray
                p[v] = u
                d[v] = d[u] + 1
                Q.insert (v)
            }
        }
        color [u] = black              // done with it
    } //while
} //BFS
```

Date

Remember for the final [★No duplicate times, each time is unique! Discovery time is <u>always</u> less than finish time!

Depth First Search

- Edges are explored out of the most recently discovered node that still has unexplored edges.
- When all of a node's edges have been explored, backtrack to a predecessor

<u>White, Gray, Black</u>

- Each node is timestamped
  - Each time is unique and is between 1 and 2|V| (1 and 2 nodes)
- d - discovery time
- f - finish time

★ $d[v] < f[v]$ ★

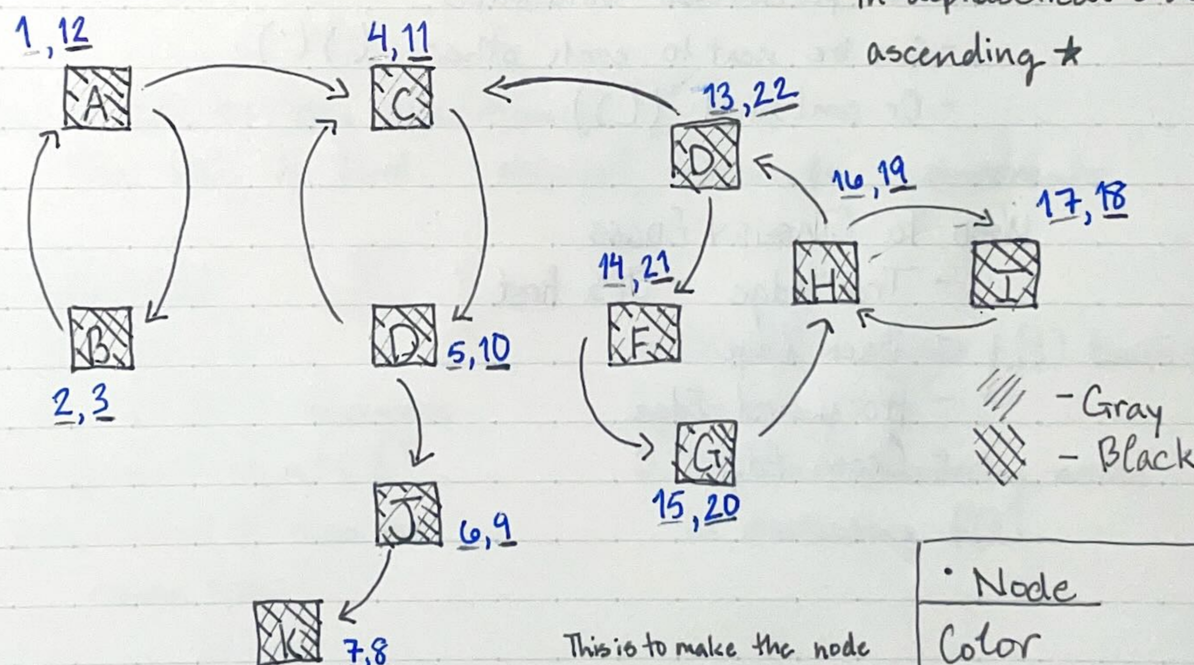
- p - parent

```
DFS(graph G) {
   //color each vertex to white      //u = v∈V
   for each vertex u {
      color[u] = white
      p[u] = Ø              // null or some flag
   } //Note: d[u] and f[u] are uninitialized
   time = 0;
   for each vertex u {
      if(color[u]== white) {
         DFS_visit(u);
      }
   }
}
```

Running Time
$O(V+E)$

```
DFS-visit(node u) {
    color[u] = gray;
    d[u] = ++time;          //increment before setting value
    for each vertex v ∈ Adjacent[u] {
        if (color[v] == white) {
            p[v] = u;
            DFS_visit(v);
        }
    }
    color[u] = black;       //done with it
    f[u] = ++time;
}
```

★ Consider all nodes in alphabetical order ascending ★



This is to make the node be complete

• Node
Color
p - parent
d - discovery time
f - finish time

- Started at A then follow directed paths
- Get to k, check, add _+1 to time then go back up to complete discovering the nodes. Don't have to follow path of directed graph for this.
- Go to closest undiscovered node and repeat until all nodes are discovered
- End should have all nodes be colored back & has 2 number values

// - Gray
X - Black

Date

What did we find out?

*Time <u>never</u> overlaps

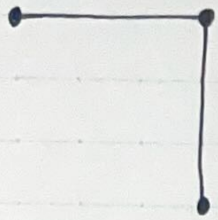
1 —— A —— 12
2 — B — 3
4 —— C —— 11
5 —— D —— 10
6 —— J —— 9
7 — K — 8
13 —— E —— 22
14 —— F —— 21
15 —— G —— 20

16 —— H —— 19
17 — I — 18

- Reveals parentheses structures
  - Can be next to each other ( ) ( )
  - Or contained (( ))

USED TO CLASSIFY EDGES
- Tree Edge - DFS first
- Back Edge -
- Forward Edge
- Cross Edge

Connected



Not Connected

Directed Graph
- Strongly Connected Components
  - Set of nodes that are all mutually reachable
  - Every node is in a set
  - Minimum set size is one.

SCC of Graph G

{A,B} {K}

{C,D} {E,F,G,H,I}

{J}

DFS

How Can We Do This Algorithmically?
- Use DFS to find strongly connected components

To Find SCC

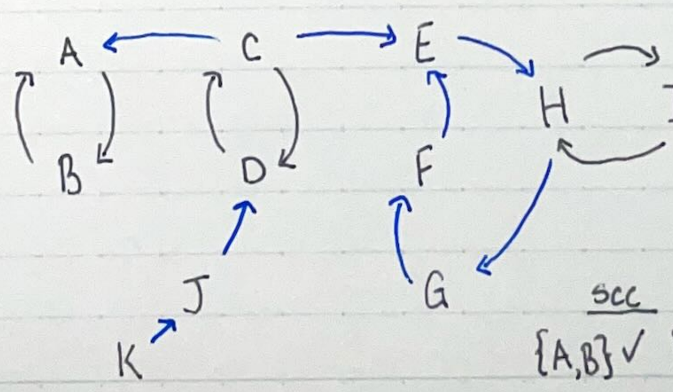
$G = (V, E)$

$G^T = (V, E^T)$ transpose

$(a,b) \in E \Rightarrow (b,a) \in E^T$

Note: G and G^T have the same SCC

1. DFS(G) remember f[t] finish time
2. Generate G^T
3. DFS(G^T) considering nodes in decreasing f[t]



m - Strongly connected paths
- Check after to see if scc is still there

SCC

{A,B}✓ {C,D}✓ {J}✓ {K}✓ {E,F,G,H,I}✓