SEARCHING IN GRAPHS

We already did the most basic form of search: (neighbor query)

In general, given a vertex \( s \), we want to locate some vertex \( t \), find a path in \( G \)

or we want to visit all vertices, in a "local" organized manner.
BREADTH FIRST SEARCH (BFS)

Start by checking if \( t \) is a neighbor of \( s \).

\( \rightarrow \) Look one step away from \( s \).

If yes, done. If not, then check all neighbors of neighbors.

\( \rightarrow \) One step from each.

Either done, or repeat (dig deeper) ... only on unexplored neighbors!
Search follows a tree pattern.

BFS extends depth by 1 at all possible nodes

- always processing nodes closer to $S$ first
- if any node is rediscovered, pretend it didn't happen (e.g. $u$)
If $s$ and $t$ are in the same connected component then the search will find $t$.

Even better, BFS will find a shortest path $s \rightarrow t$.
(prove by contradiction)

(time? (supposing we can tell instantly whether a vertex is "new")

$O(IEI)$ (in component of $s$)
Algorithm:

1. Mark $s$.

2. While $Q$ is not empty,
   - Remove first vertex $v_f$ in $Q$.
   - Check $\text{Adj}[v_f]$: $u_1, \ldots, u_p$.
     - If $v_f = t$, done.
     - If $v_f \neq t$ and unmarked, put $u_i$ in $Q$.
     - Mark as visited.

Q: $v_1, v_2, v_3, v_4, v_5 \leftarrow \text{in}$

Out: $v_2, v_3, v_4, v_5, u_1, u_2, u_3, u_4, u_5, u_6, x_1, x_2, x_3, x_4$
- Mark S & put in Q.
- depth(S) = 0

While Q not empty,
  x = dequeue(Q)
  check Adj[x]: u, ... up:
    if u is unmarked
      mark u; & put in Q.
      depth(u) = 1 + depth(x)
      parent(u) = x; u; → child(x)
  if v_i = t done
  if v_i ≠ t, mark as visited
  put in queue: Q

(0) Mark S
(1) Check Adj[S]: v_i, ... v_k
    if v_i = t done
    if v_i ≠ t mark as visited
      put in queue: Q

(2) While Q not empty,
    - remove first vertex v_f in Q
    - check Adj[v_f]: u_i, ... u_p
      if u_i = t done
      if u_i ≠ t & unmarked
        put u_i in Q.
        mark u_i

just the search process
DEPTH FIRST SEARCH (DFS)

- Follow an unvisited path for as long as possible.
- When you reach a vertex with only previously-visited neighbors, back up (from where you came from) & try again.
As with BFS, mark visited nodes.

Adjacency list:

- S → V₁ → V₂ → V₄ → V₅
- V₁ → V₂ → V₃ → S
- V₂ → S → V₃ → V₁
- V₁ → V₂ → V₃
- V₄ → S
- V₅ → S
As with BFS, mark visited nodes.
As with BFS, mark visited nodes.

\( v_3 \) has nowhere to go.
As with BFS, mark visited nodes.

$V_3$ came from $\text{Adj}[V_2]$
As with BFS, mark visited nodes.

$V_2$ continues its search.
As with BFS, mark visited nodes.

V₂ continues its search
...but v₁ has been visited
As with BFS, mark visited nodes.

Now \( v_2 \) has nowhere to go. \( v_2 \) came from \( \text{Adj}[v_1] \)
As with BFS, mark visited nodes.

$v_1$ doesn't know $v_3$ is marked.
As with BFS, mark visited nodes.

$v_1$ discovers $v_3$ is marked

Adjacency list:

- $S \rightarrow V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_5$
- $V_2 \rightarrow S \rightarrow V_3 \rightarrow V_1$
- $V_3 \rightarrow V_2 \rightarrow V_1$
- $V_4 \rightarrow S$
- $V_5 \rightarrow S$
As with BFS, mark visited nodes.

$V_1$ discovers $S$ is marked and has nowhere else to go.
As with BFS, mark visited nodes.

\[ V_1 \] came from \( \text{Adj}[S] \)
As with BFS, mark visited nodes.

s continues on Adj[s]...

...discovers v_2 is marked

etc
DFS(s)
- mark s
- for every neighbor \(v_i\) of s  
  if \(v_i = t\), DONE  
  if \(v_i\) is unmarked
    - set parent\((v_i)\) \(\rightarrow\) s
    - set depth\((v_i)\) \(\rightarrow\) 1 + depth\((s)\)
    - DFS\((v_i)\)

Data structure?

Stack

Time: \(O(1|E|)\)  
\(|E|\) : size of component

(we use every edge twice & \(E > V - 1\))
DFS on a non-connected graph $G$

For every vertex $v_i$ in $G$
if $v_i$ is unmarked
DFS($v_i$)

It is also easy to keep a counter to keep track of the "time" at which each vertex is first encountered & fully processed

time: $O(V+E)$
DFS on a directed graph: similar to non-connected (process all vertices)
DFS on a directed graph: similar to non-connected (process all vertices)
DFS on a directed graph: similar to non-connected (process all vertices)

1, 2, 3, 4, 5
Adj(v1): v3, v2, ...

again
1, 2, 3, 4, 5

4 before 1, 5
... before 3
... etc

3 before 1 in
Adj[4]