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.

locate → find a path in G

or we want to visit all vertices,
in a "local" organized manner.

BREADTH FIRST SEARCH (BFS)

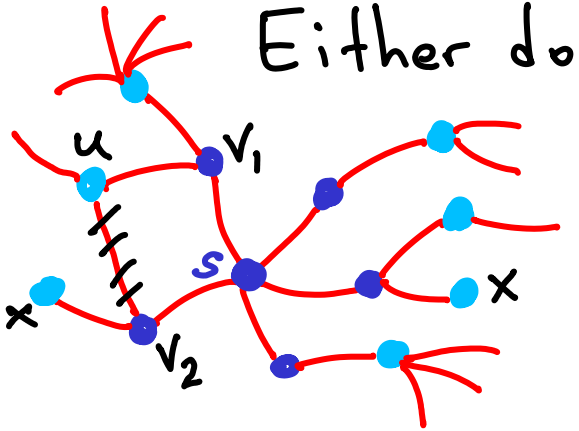
Start by checking if t is a neighbor of s.

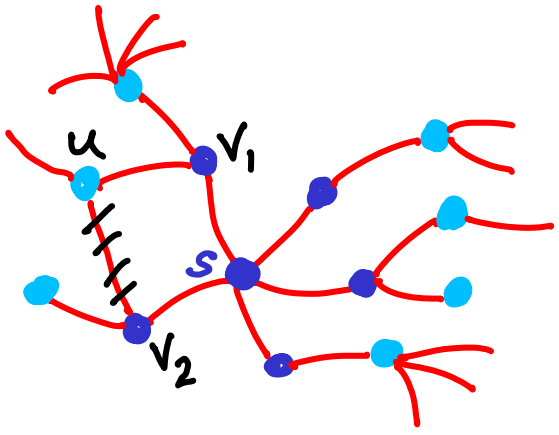
↳ look one step away from s.

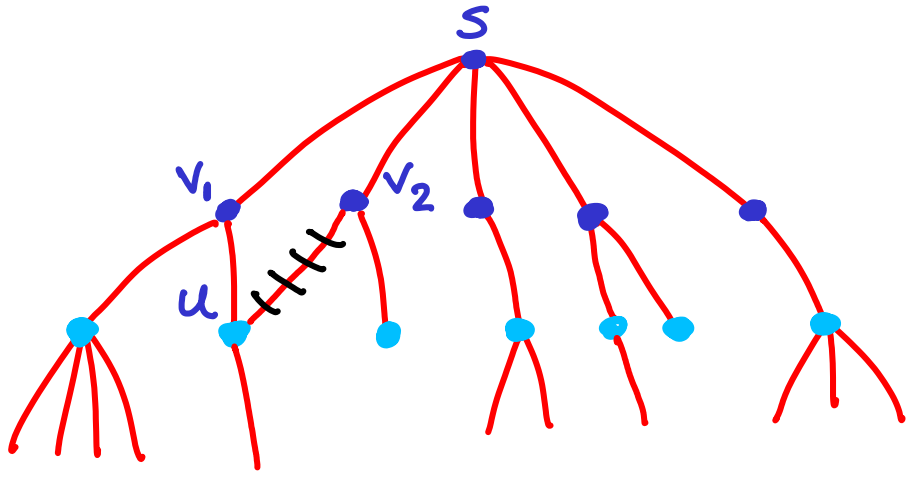
If yes, done. If not, then check all unexplored! neighbors-of-neighbors

↳ one step from each.

Either done, or repeat (dig deeper) ... only on unexplored neighbors!





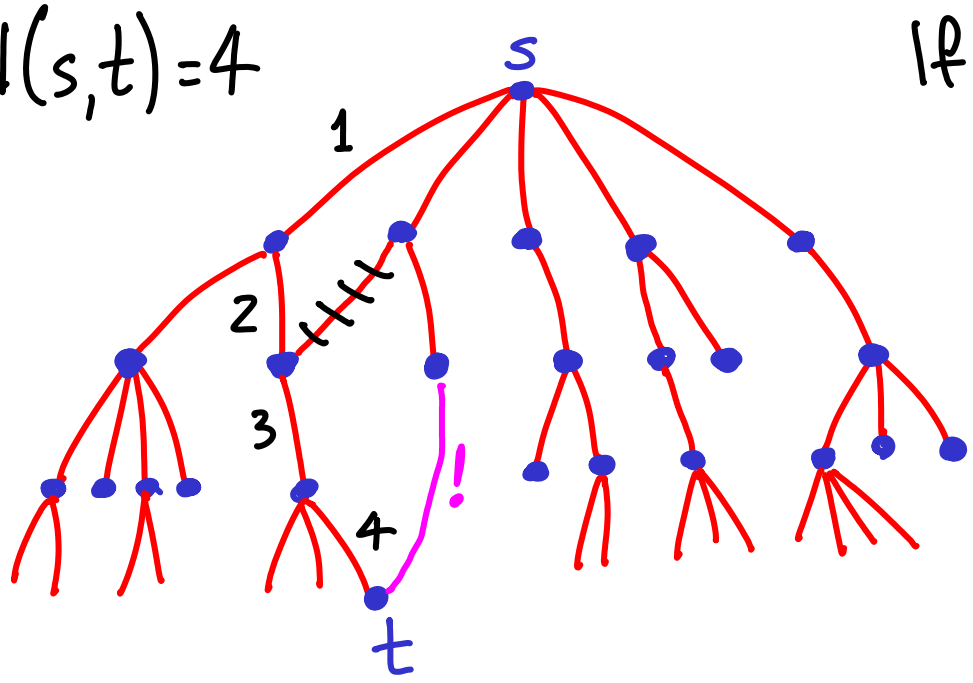


Search follows a tree pattern.

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 (eg. u)



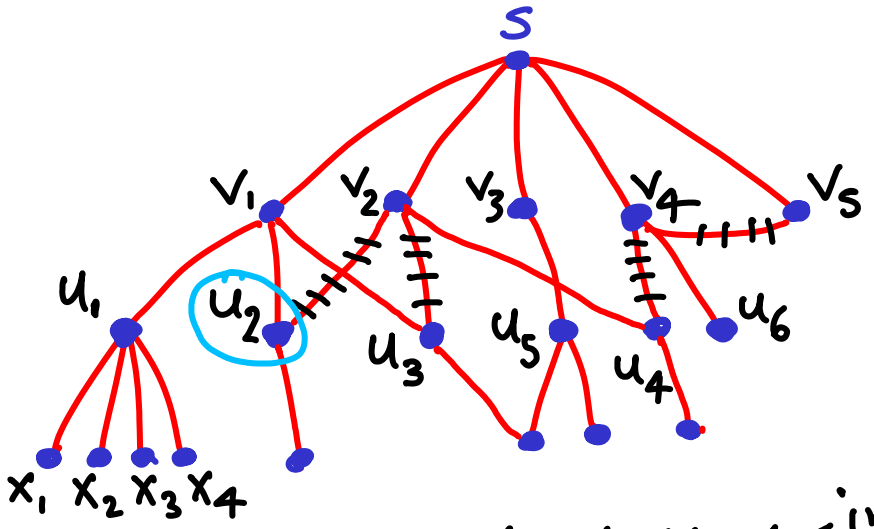
If s & t are in the same connected component then the search will find t.

Even better, BFS will find a shortest path s→t.

(prove by contradiction)

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

$O(|E|)$ (in component of s)



ALGO:

(0) mark s

(1) check Adj[s] : $v_1 \dots v_k$

- if $v_i = t$ DONE
- if $v_i \neq t$
  - mark as visited
  - put in queue : Q

(2) While Q not empty,

- remove first vertex v_f in Q
- check Adj[v_f]: $u_1 \dots u_p$
  - if $u_i = t$ DONE
  - if $u_i \neq t$ & unmarked
    - put u_i in Q.
    - mark u_i

Q: ← in / out

v_1 v_2 v_3 v_4 v_5

v_2 v_3 v_4 v_5 u_1 u_2 u_3

v_3 v_4 v_5 u_1 u_2 u_3 u_4

v_4 v_5 u_1 u_2 u_3 u_4 u_5

v_5 u_1 u_2 u_3 u_4 u_5 u_6

u_1 u_2 u_3 u_4 u_5 u_6

u_2 u_3 u_4 u_5 u_6 x_1 x_2 x_3 x_4

etc

- mark S & put in Q.
- depth(S) = 0

- While Q not empty,
  - x = dequeue(Q)
  - check Adj[x]: $u_1 \dots u_p$:
    - if u_i is unmarked
      - mark u_i & put in Q.
      - depth(u_i) = 1 + depth(x)
      - parent(u_i) = x ; $u_i \rightarrow$ child(x)

just the search process

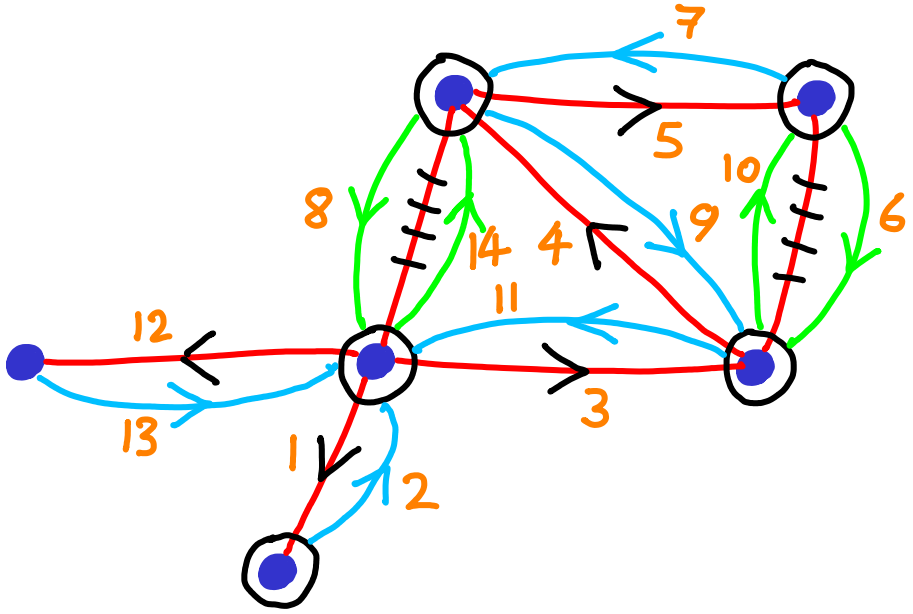
(0) mark S

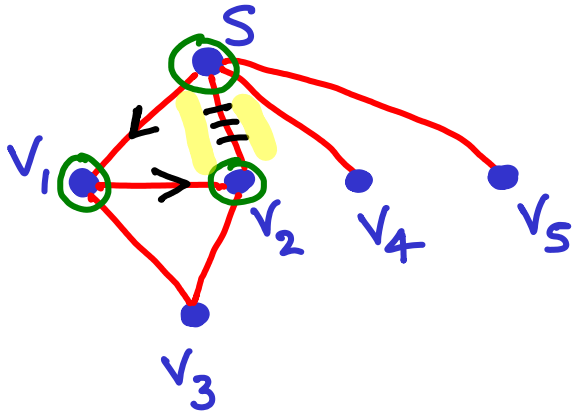
(1) check Adj[S]: $v_1 \dots v_k$
- if $v_i = t$ DONE
- if $v_i \neq t$
  - mark as visited
  - put in queue : Q

(2) While Q not empty,
- remove first vertex v_f in Q
- check Adj[v_f]: $u_1 \dots u_p$
  - if $u_i = t$ DONE
  - if $u_i \neq t$ & unmarked
    - put u_i in Q.
    - mark u_i

DEPTH FIRST SEARCH (DFS)

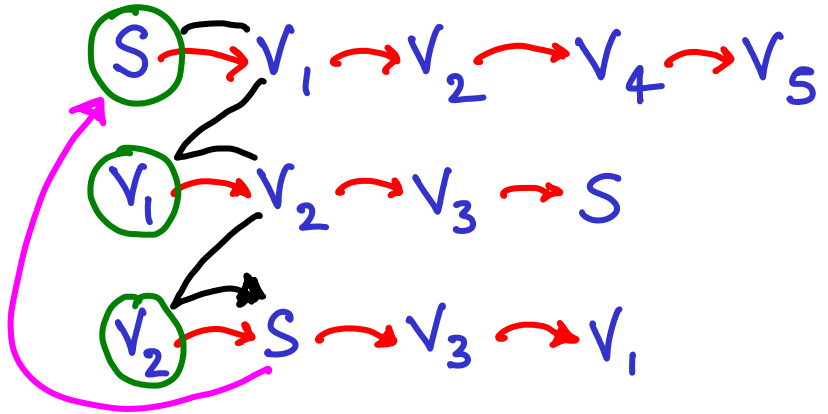
- Follow an unvisited path for as long as possible.
- When you reach a vertex w/ only previously-visited neighbors, back up (from where you came from) & try again.





As with BFS, mark visited nodes.

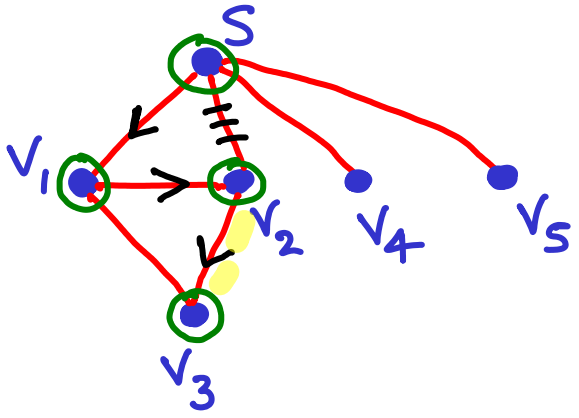
Adjacency list



$V_3 \rightarrow V_2 \rightarrow V_1$

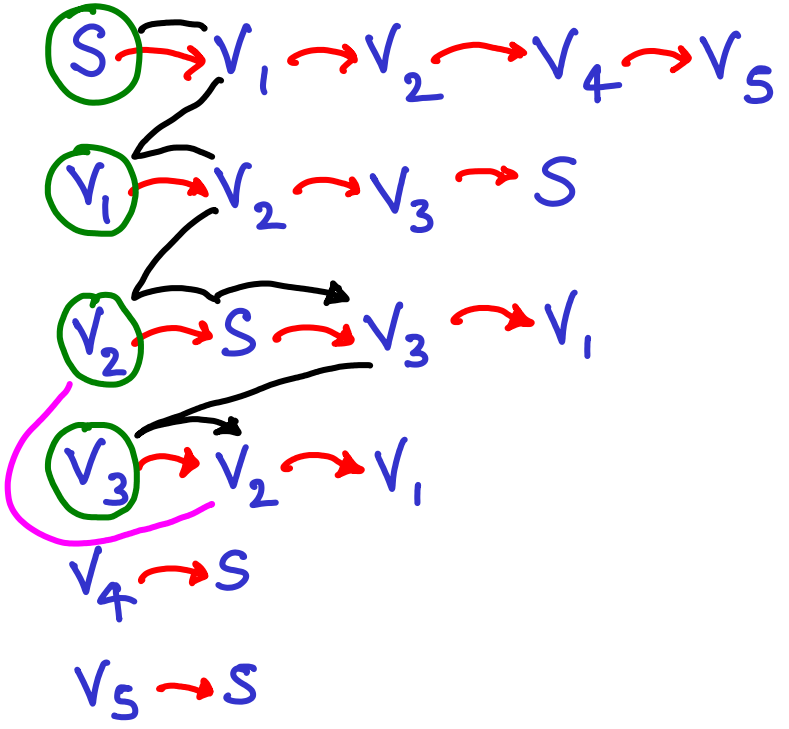
$V_4 \rightarrow S$

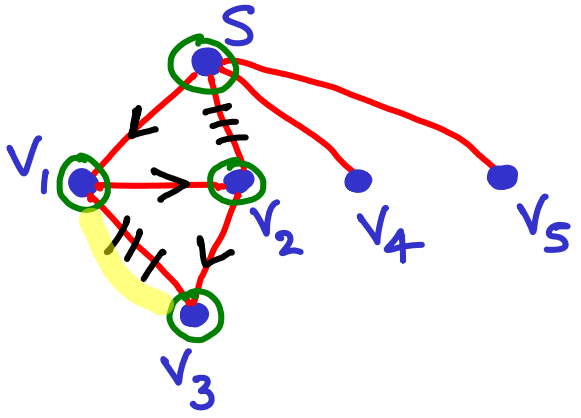
$V_5 \rightarrow S$



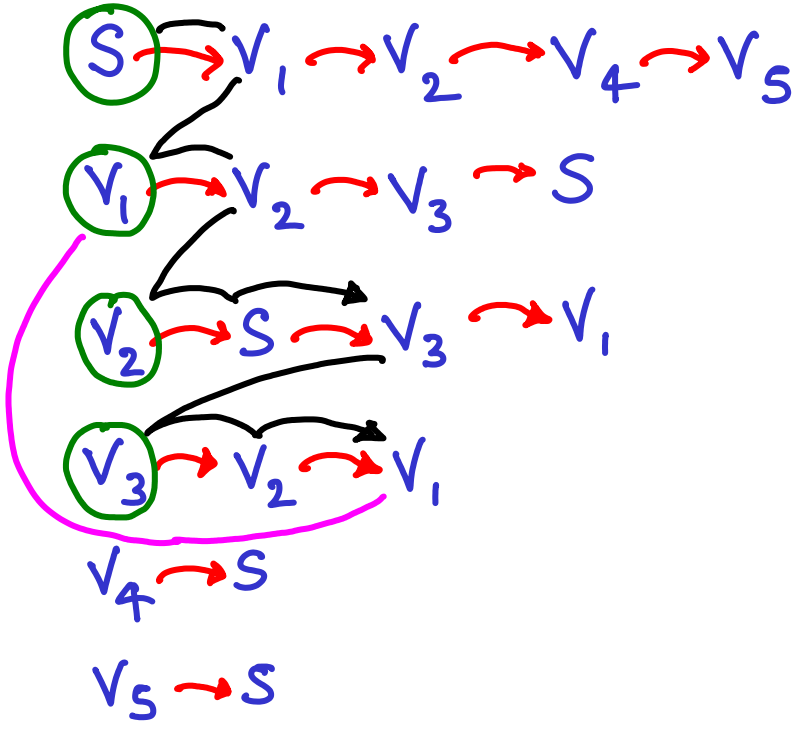
As with BFS, mark visited nodes.

Adjacency list



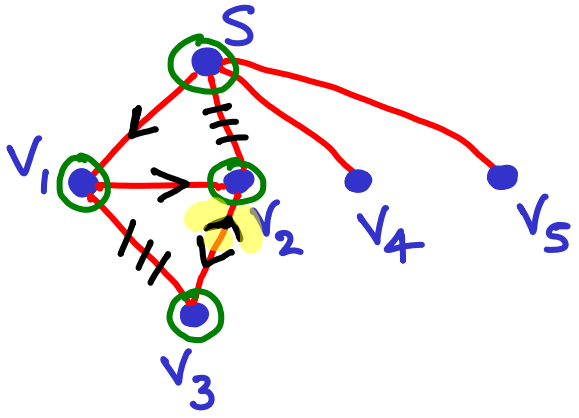


Adjacency list

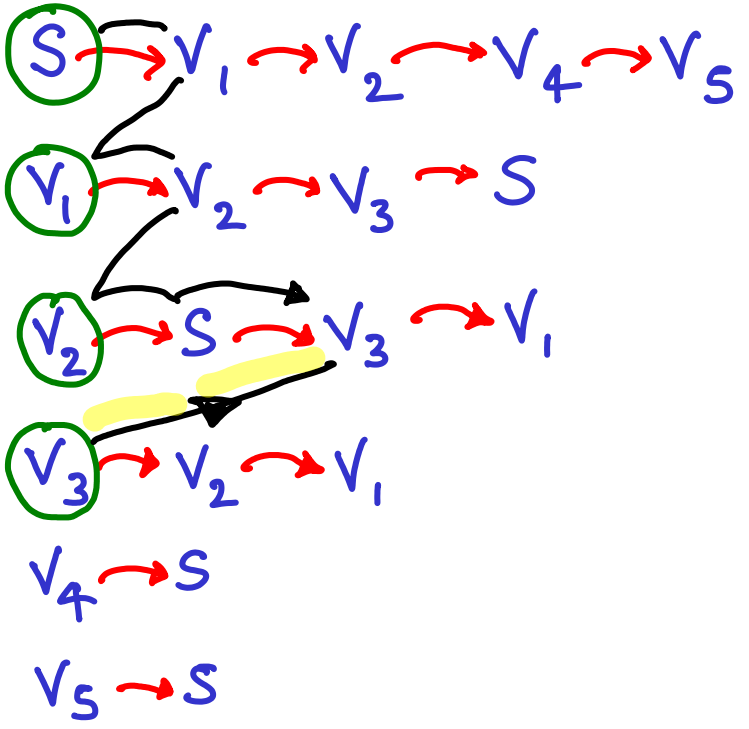


As with BFS, mark visited nodes.

v_3 has nowhere to go

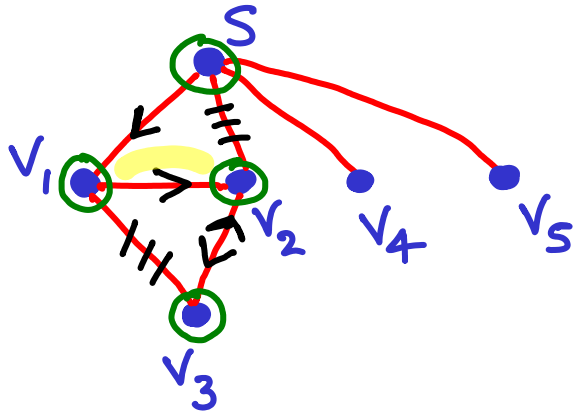


Adjacency list

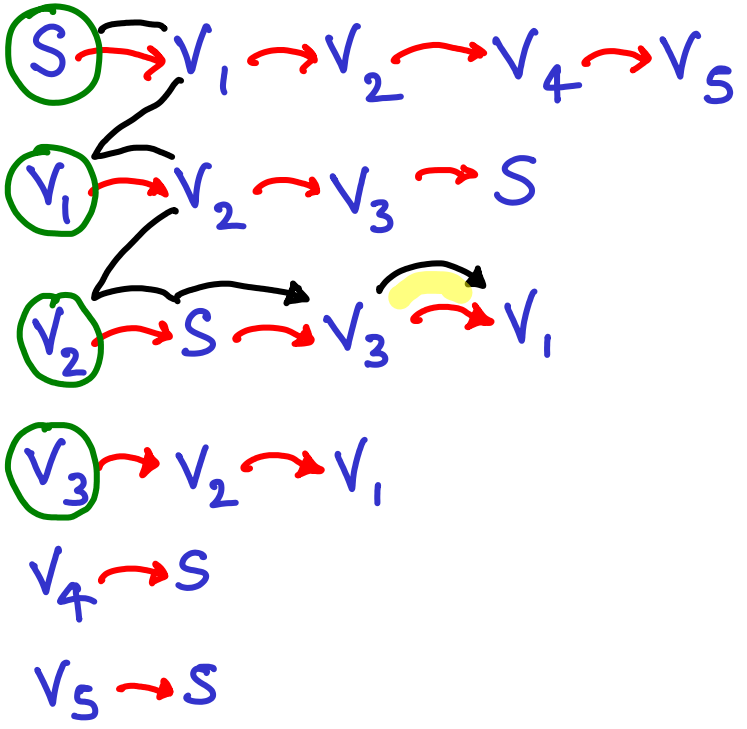


As with BFS, mark visited nodes.

v_3 came from $Adj[v_2]$

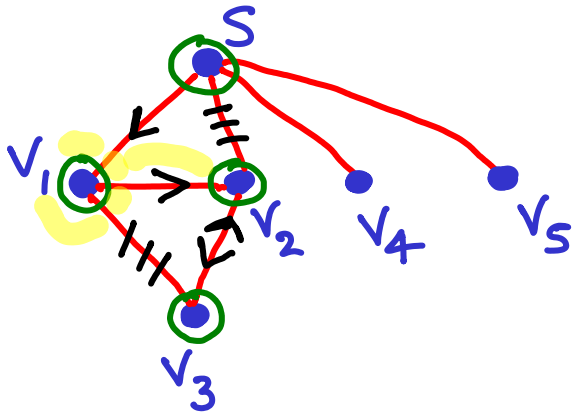


Adjacency list

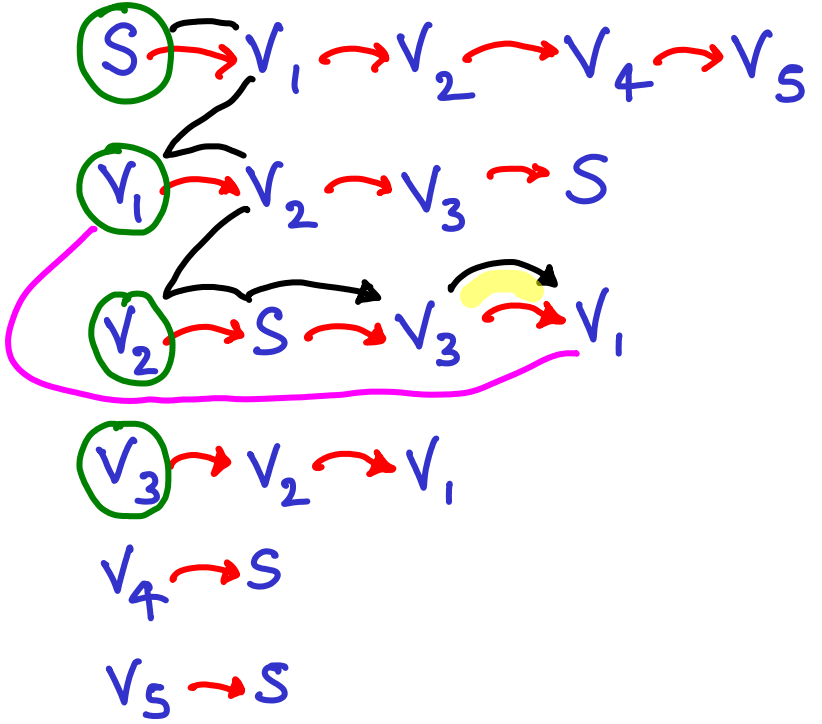


As with BFS, mark visited nodes.

v_2 continues its search



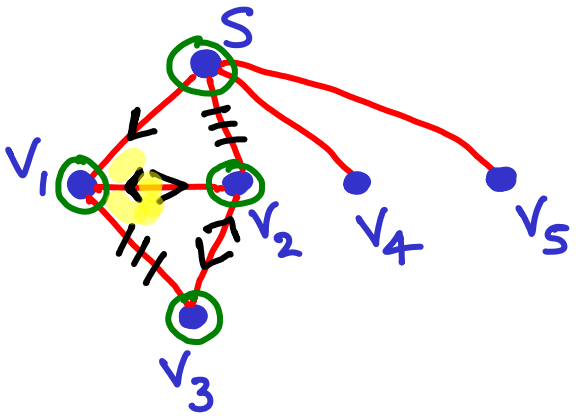
Adjacency list



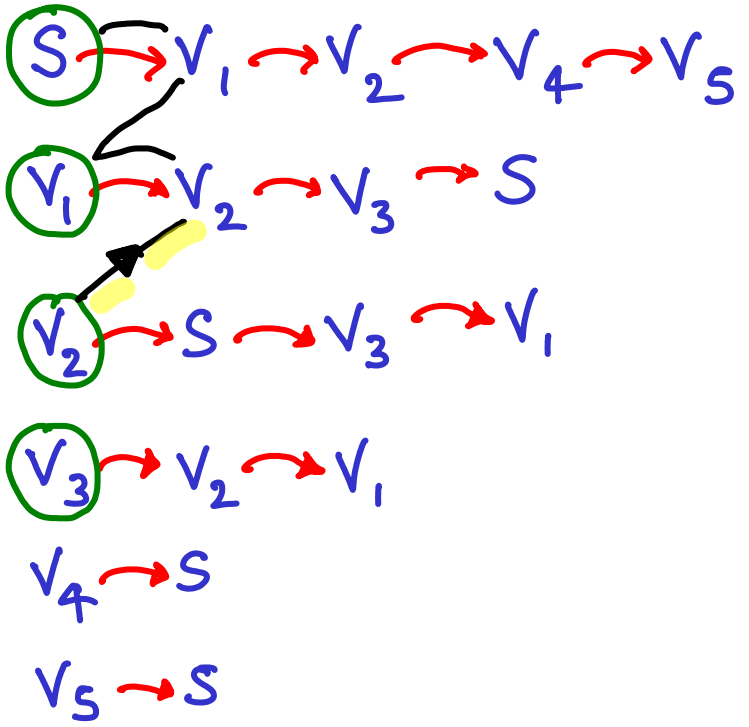
As with BFS, mark visited nodes.

V_2 continues its search

...but V_1 has been visited



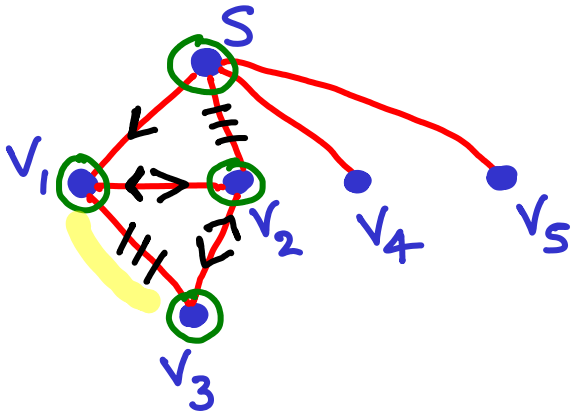
Adjacency list



As with BFS, mark visited nodes.

Now v_2 has nowhere to go.

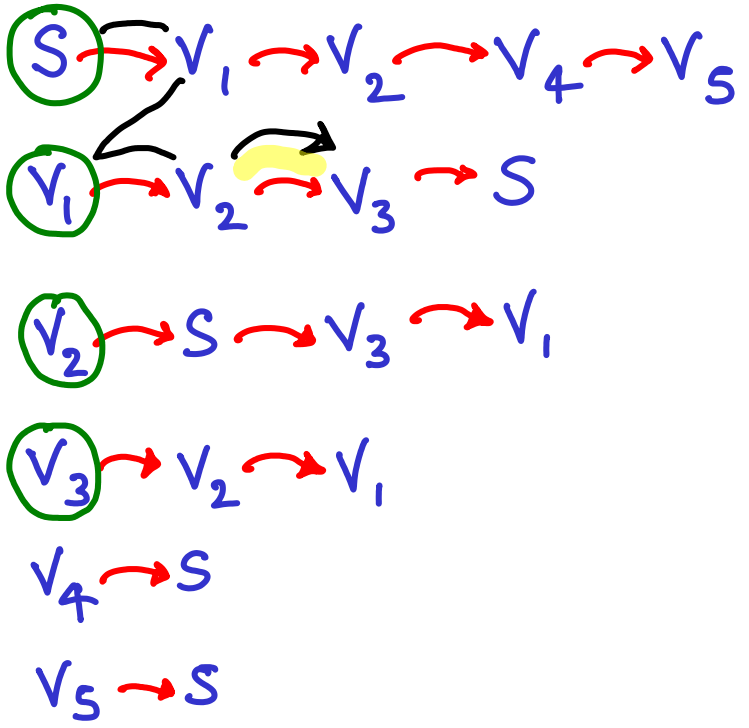
v_2 came from $Adj[v_1]$

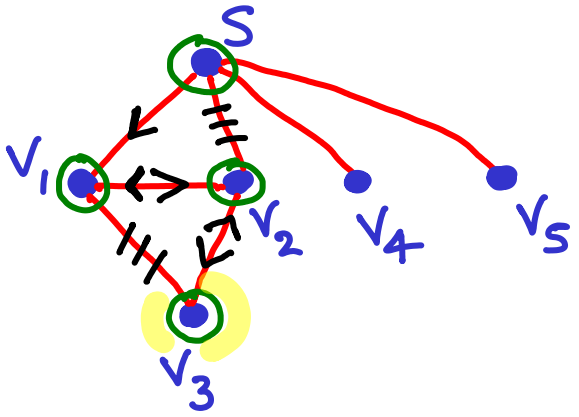


As with BFS, mark visited nodes.

v_1 doesn't know v_3 is marked

Adjacency list

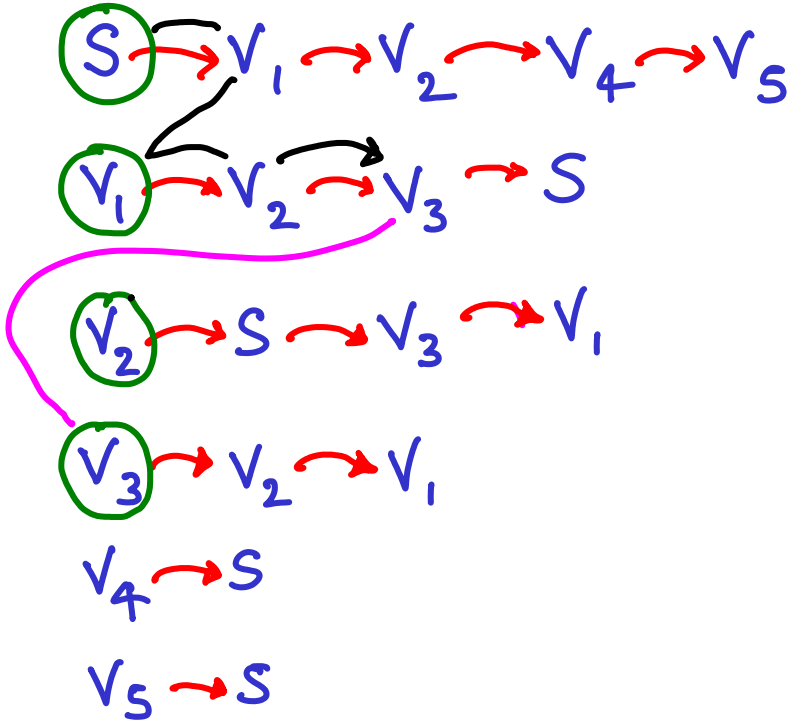


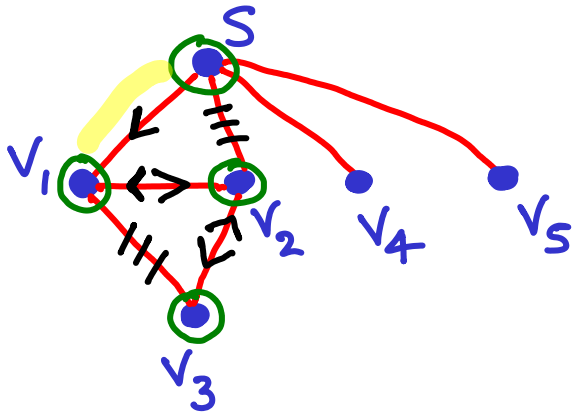


As with BFS, mark visited nodes.

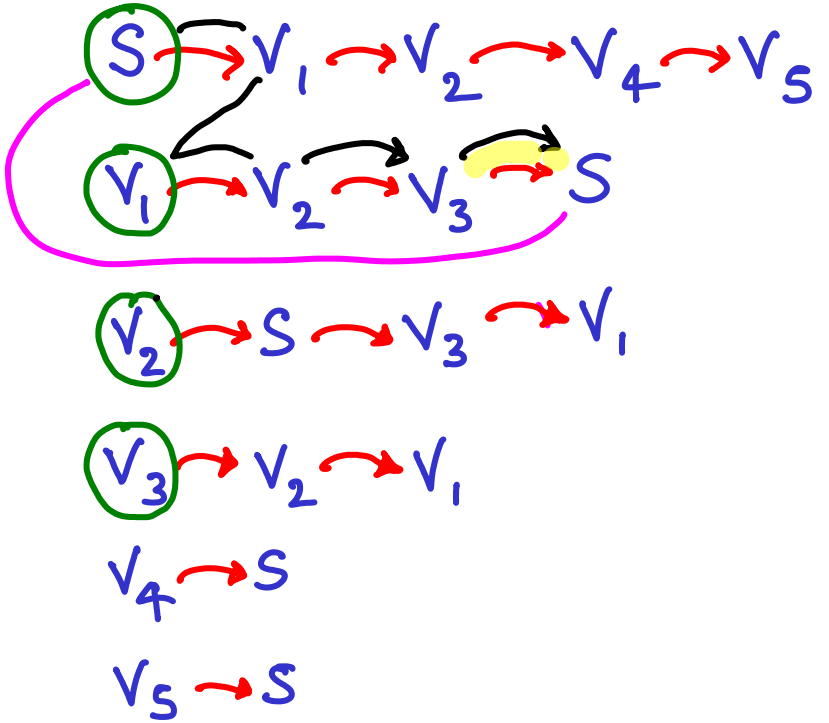
v_1 discovers v_3 is marked

Adjacency list



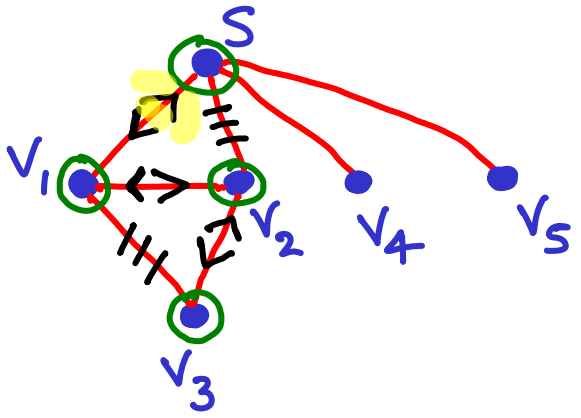


Adjacency list



As with BFS, mark visited nodes.

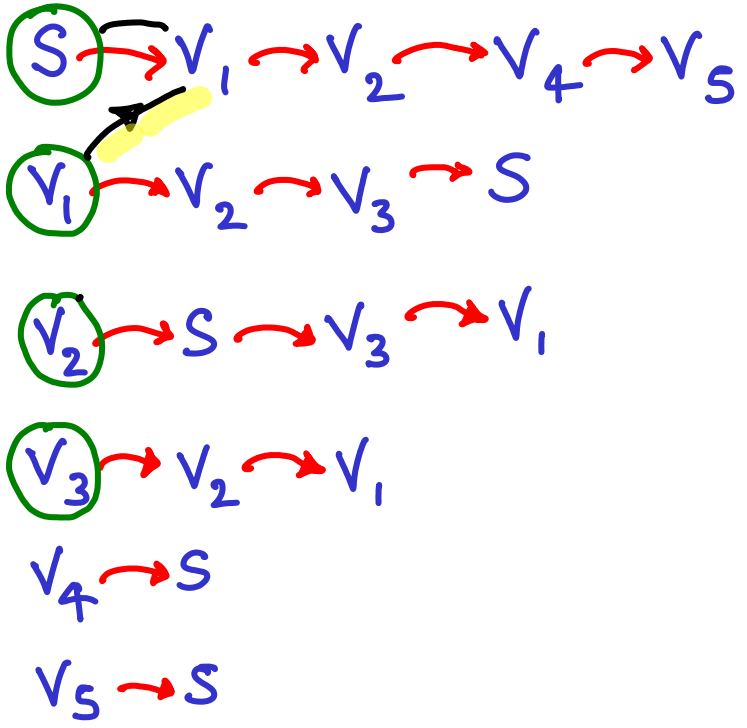
v_1 discovers S is marked and has nowhere else to go

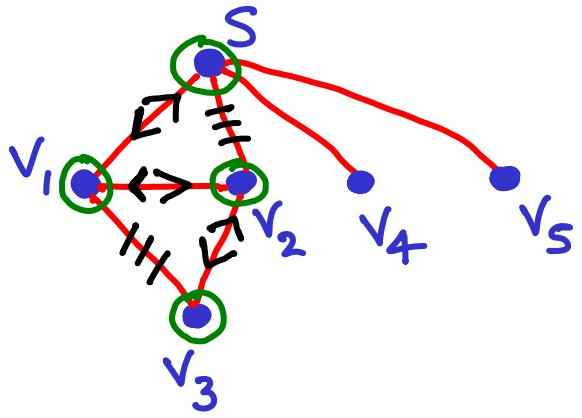


As with BFS, mark visited nodes.

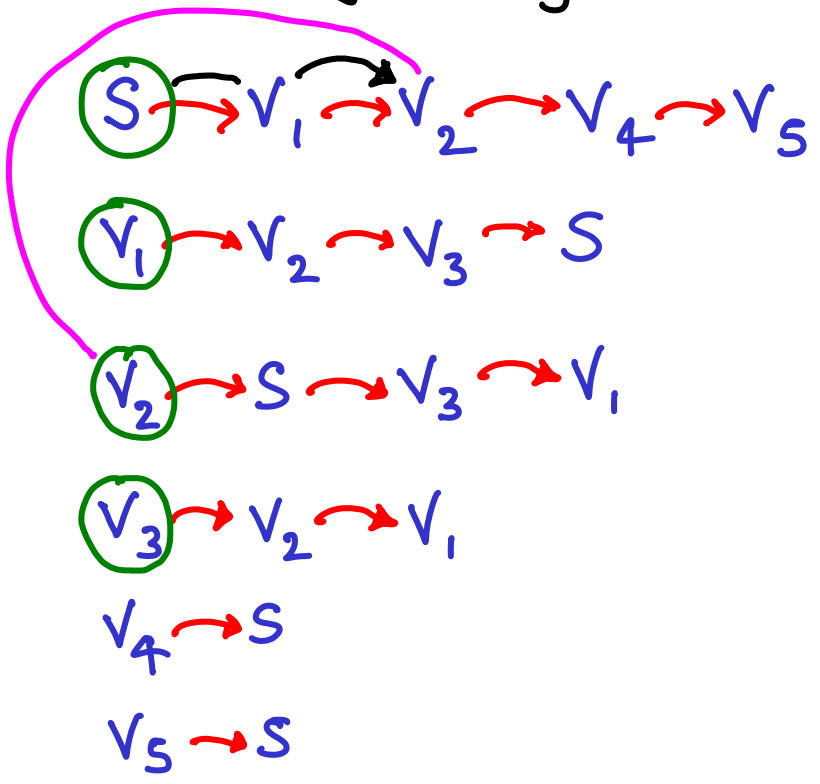
v_1 came from Adj[s]

Adjacency list





Adjacency list

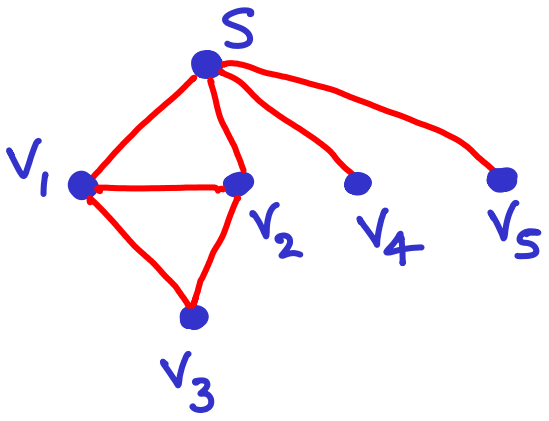


As with BFS, mark visited nodes.

s continues on Adj[s]...

...discovers v_2 is marked

etc



DFS(s)

- mark s
- for every neighbor v_i of s // i.e. scan Adj[s]
  - if $v_i = t$, DONE // only if looking for t
  - if v_i is unmarked
    - set parent(v_i) → s
    - set depth(v_i) → 1 + depth(s)
    - DFS(v_i)

} only if you want to keep the structure (init depth = 0)

Data structure?

Stack

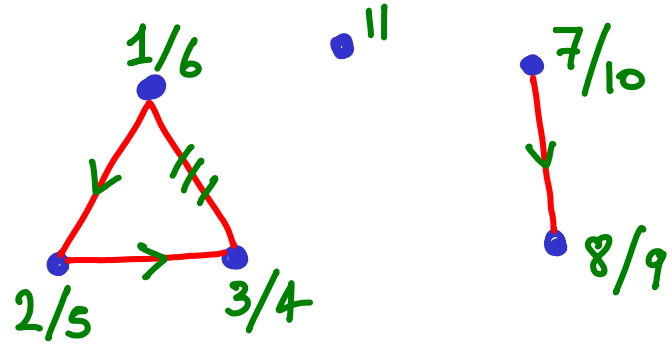
time: O(|E|) // |E| : size of component

(we use every edge twice & E > V-1)

DFS on a non-connected graph G

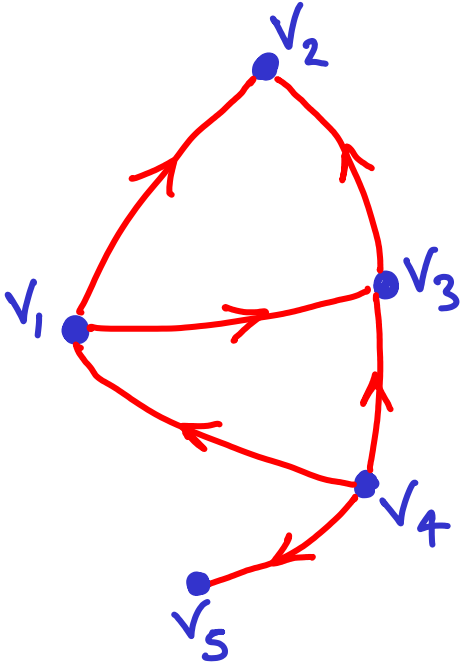
time: $O(V+E)$

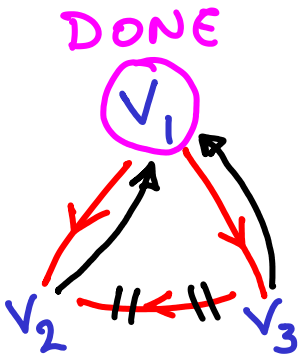
For every vertex v_i in G
    if v_i is unmarked
        DFS(v_i)



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

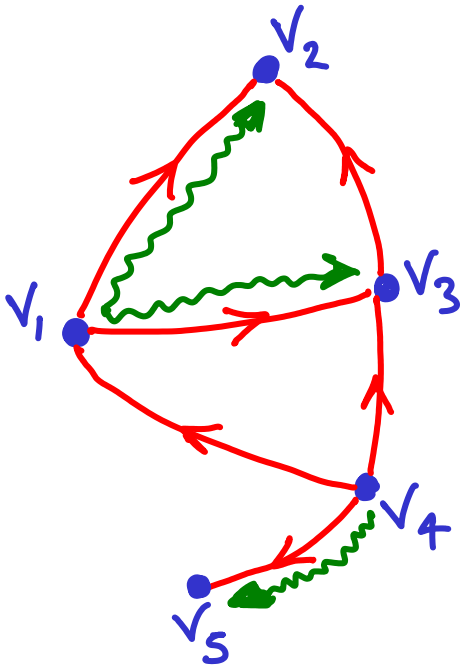
DFS on a directed graph : similar to non-connected (process all vertices)

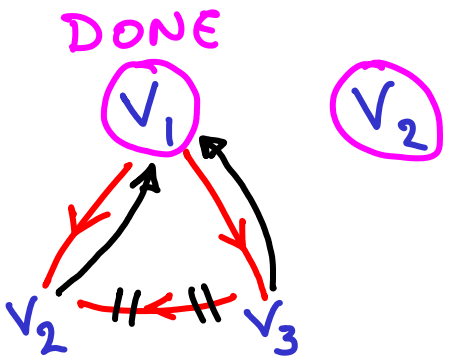




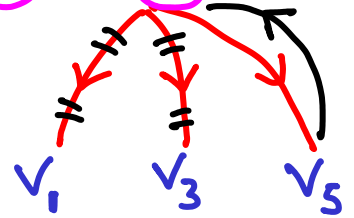
v_2 v_3 v_4 v_5

DFS on a directed graph : similar to non-connected (process all vertices)



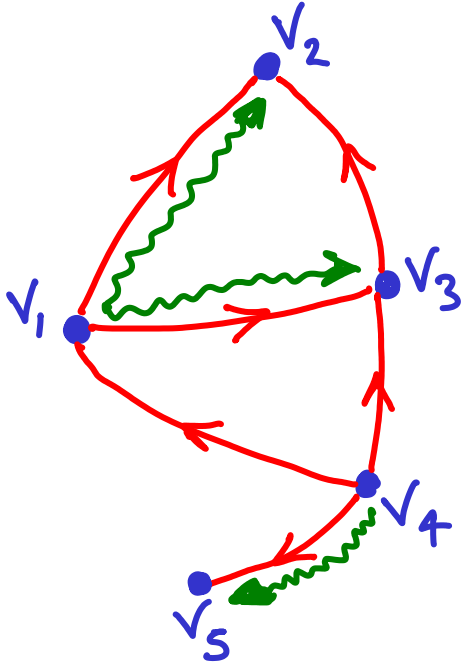


v_2 v_3 v_4 v_5



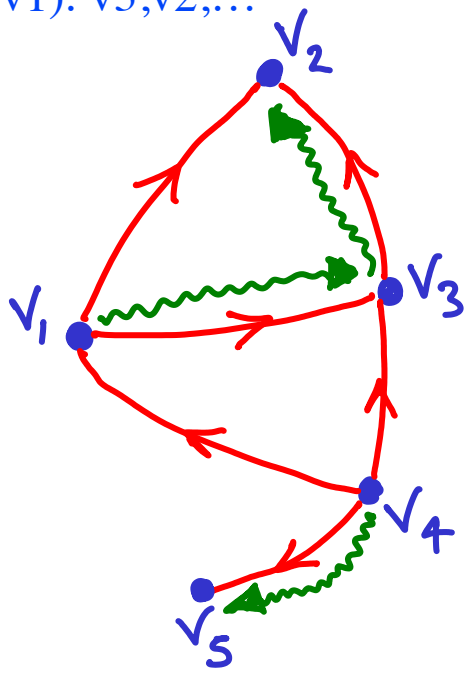
DFS on a directed graph : similar to non-connected (process all vertices)

1,2,3,4,5

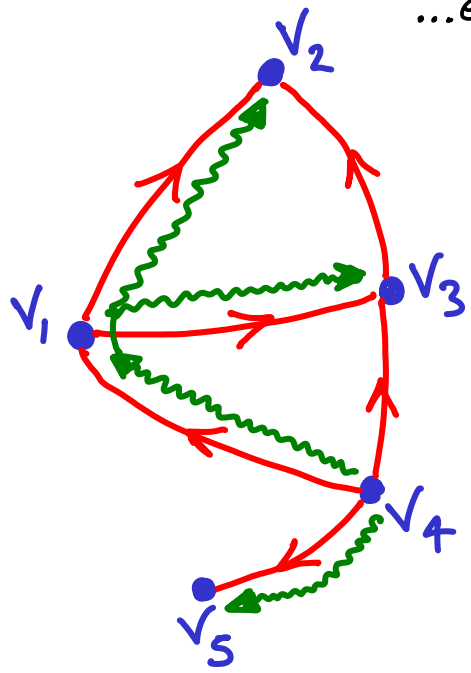


again
1,2,3,4,5

Adj(v1): v3,v2,…



4 before 1,5
… before 3
…etc



3 before 1 in Adj[4]