— × —

Graphs: Study "pairwise" relations.

"vertices" V

Examples:

- Facebook : vertices = people, edges = "friends"
- WWW : vertices = web-pages, edges = hyperlink
- Road Networks: vertices = cities, edges = highways / roads.
- Flight Network: vertices = airports, edges = flight

$G = (V, E)$ V is a finite set $\{1, 2, \ldots, 6\}$

E : "collection of pairs of V"

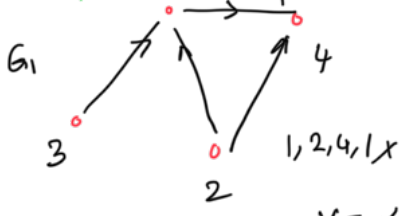
$V \times V = \{(a,b) : a \in V, b \in V\}$ ✓

$E \subseteq V \times V$ ← directed (a,b), (b,a) are not same.

$E \subseteq V \times V$ undirected

s.t. $(a,b) \in E$ then $(b,a) \in E$

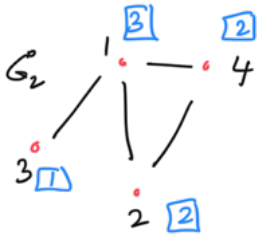
	1	2	3	4
1	0	0	0	1
2	0	1	0	1
3	1	0	0	1,4,1
4	1	0	1	



3,1,4

$V = \{1, 2, 3, 4\}$

$E = \{(1,4), (4,2), (2,1), (3,1)\}$ ← unordered pairs.



$V = \checkmark$

$E = \{(3,1), (2,1), (2,4), (4,1), (1,4)\}$

w 2,1,4,2,1,3,1,4

3,2,4 ✗

3,1,2,4

Definitions:

(1) Neighbour : $\Gamma(u)$ = set of vertices joined to u by an edge

$= \{v : (u,v) \in E\}$ ← undirected. ✓

G_2 : $\Gamma(2) = \{1,4\}$

degree of a vertex u = $|\Gamma(u)|$ ← Euler's thm.

In any undirected graph $\sum_{u \in V} \deg(u) = 2|E|$

Directed: out- neighbours $\Gamma^+(u) = \{v : (u,v) \in E\}$

in- neighbours $\Gamma^-(u) = \{v : (v,u) \in E\}$

out-degree, in-degree

$\sum_{u \in V}$ out-deg(u) $= |E|$

in-deg(u).

(2) Walks / paths / cycles :

A walk W is a sequence of vertices

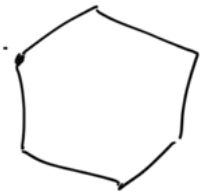
there could be repetitions — $v_1, v_2, \ldots, v_r$ such that $\forall\ i = 1, \ldots, r-1$ $(v_i, v_{i+1}) \in E$.

$v_1 \to v_2 \to v_3 \to v_4 \ldots \to v_r$

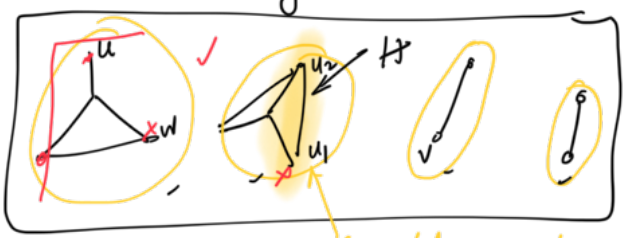
Path : a walk where no vertex is repeated. $v_1, v_2, \ldots, v_r$

Cycle: a walk where only the first and the last vertices are same and other vertices are distinct. } must contain ≥ 2 vertices.

- If there is a walk from v_1 to v_r, then there is a path from v_1 to v_r.



(3) Connectivity: Undirected: G is connected if there is a path from every vertex u to every other vertex v.



G. "maximal connected subgraph"

If $G=(V,E)$ is a graph, then $G'=(V',E')$ is a subgraph of G if $V' \subseteq V$, $E' \subseteq E$.

Formal definition of connected component:

We define relation on V: $u \sim v$ (is related to) if there is a path from u to v in G.

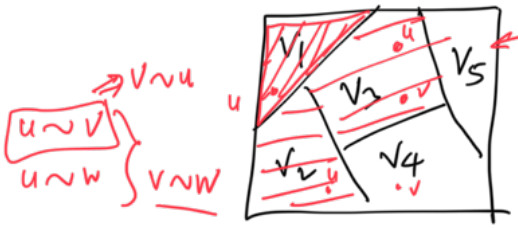
✓ Equivalence Relation:

(i) Reflexive: $u \sim u$

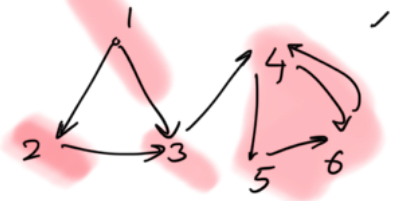
(ii) Symmetric: $u \sim v \Rightarrow v \sim u$

(iii) Transitive: $u \sim v$, $v \sim w \Rightarrow u \sim w$? ✓ we have a walk from u to w.

$u \to v \to w$



Directed Graph: $G=(V,E)$



1 ~ 2 ? No.
4 ~ 5 YES

G is strongly connected if there is a path from every vertex u to every vertex v.

"underlying undirected graph"

"strongly connected" components.

$u \sim v$ if there is a path from u to v AND there is a path from v to u.

(i) reflexive u

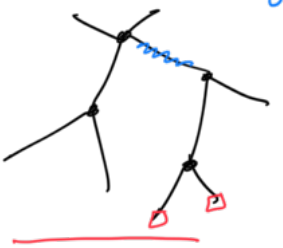
(ii) symmetric ✓

(iii) transitive: $u \sim v$, $v \sim w$: ✓

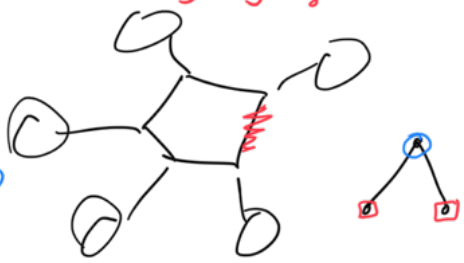
the equivalence classes define strongly connected components.

(4) Trees, forests ... :

Undirected : connected graph without any cycles.



"minimally" connected graphs.
removing any edge disconnects the graph.



Fact : If G is a tree, n vertices, then it has n-1 edges. ⊛

Pf : By induction on n.

n = 1 : • n=1, 0 edges. ✓

Induction Step: Suppose this statement ⊛ is true for any tree on n-1 vertices.

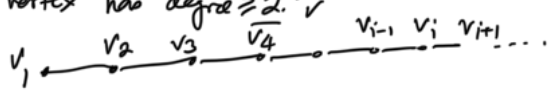
→ Let G be a tree on n vertices. ✓

Leaf : is a vertex of degree 1.

Let us remove a leaf. ✓

suppose every vertex has degree ≥ 2. ✗ ✓

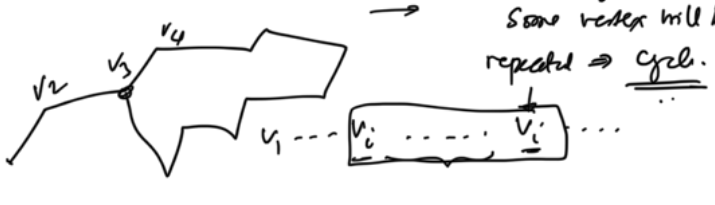
there must be a vertex of degree 1. We form a chain $v_1, v_2, \dots, v_{i-1}, v_i, \dots$ →



∵ the graph is finite
Some vertex will be repeated ⇒ cycle.

"u"

remove u :



G' : n-1 vertices, connected, no cycles

↓

a tree → IH has n-2 edges.

⇒ G has n-1 edges. ✓

— x —

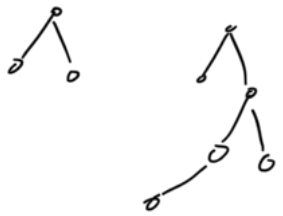
Tree : (i) Connected
(ii) No cycles
(iii) n-1 edges.

(i), (ii) ⇒ (iii) ✓
(i), (iii) ⇒ (ii)
(ii), (iii) ⇒ (i)

Directed : Out-tree, in-tree

"arborescence"

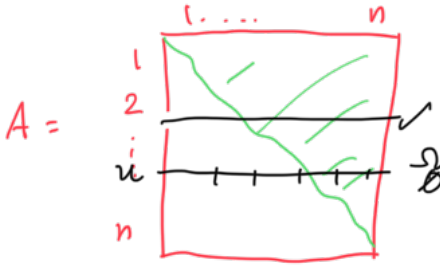
Forest : undirected graph with no cycles. ← each connected component is a tree.



How to store a graph?

What kind of questions do we answer for a graph?

Adjacency Matrix



$A[i,j] = 1$ if $(i,j) \in E$

symmetric.

$A[u,v]$.

- (i) u, v : is there an edge (u,v)? — $O(1)$ — $\geq \deg(u)$
- (ii) What are the neighbors of u?
- (A) Is G connected? strongly connected? — if not, connected components?
- (B) u, v : is there a path from u to v? shortest path?
- (C) Robust?

$A[u,v] = 1 \Rightarrow (u,v) \in E$ } $O(1)$ time

$= 0 \Rightarrow (u,v) \notin E$ }

$\sum_{v}^{n} \deg(v) = 2m$.

$O(n^2)$ storage.

"space complexity"

"network usage complexity" ...

Resource Complexity

$n, \leq \binom{n}{2} \sim \frac{n^2}{2}$.

$\frac{\sum_{v} \deg(v)}{2} = |E|$

In practice, most large graphs are "sparse"

eg, $n = 10^9$, $n^2 = 10^{18}$

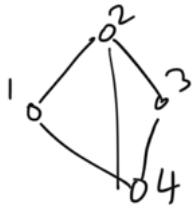
$m = 200 \times 10^9 \sim 10^{11}$ edges

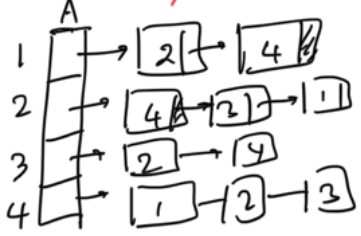
Internet, www

10^9 time. ~ 100

$n = |V|$, $m = |E|$.

Adjacency List: for each vertex v, we store the list of neighbours of v





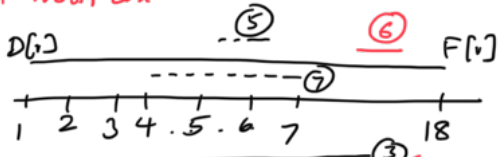
Directed:
- out-neighbours
- in- nbrs.

Space Requirement: $n + \sum_{v \in V} \deg(v) = n + 2m = O(n+m)$.

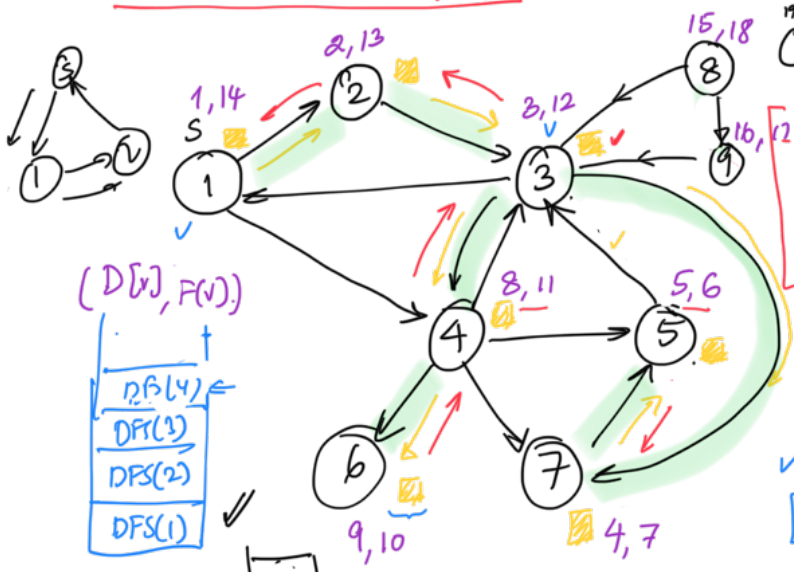
$\Omega(\deg(u))$.

(i) $(u,v) \in E$: degree (u) — $\Omega(n)$ in worst case.

(ii) $M(u)$: degree (u) ← optimal



Traversal on a graph: starting from s, "discover" the graph.



visit vertices

start from s, keep walking.

what are the pitfalls? "dead end"

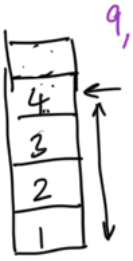
eg, No outgoing edge

- we might be stuck in a loop

"memory" → "backtrack"

(D[v], F(v))

recursion →



Visited [ ] ← array

Parent [u] ← the preceding vertex before u

Stack:

Going to a new vertex: pushing on the stack. ← DFS call start

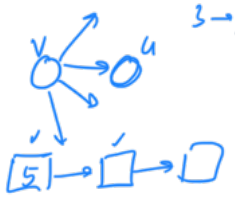
Backtrack: pop from the stack ← DFS fn. ends

DEPTH FIRST SEARCH

clock: 20

```
visited[v] = false ;   DFS(s)

DFS(v) {
    clock++;   D[v] = clock;   ← discovery time
    visited[v] = true ;
    for all neighbours u to v ← (out)   ← at this time we can classify (v,u) as B, F, C.
        if (visited[u] == false) {
            parent[u] = v
            DFS(u) ;
        }
    clock++;
    finishing time → F[v] = clock;
}
```



3 → 1 → 2 → 4

for i = 1 ... n
    if (visited[i] is false)
        DFS(i)

D[u] F[u] < D[v]

Running Time: For each vertex v, we call DFS(v) at most once. ✓

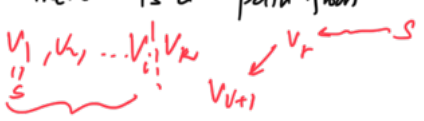
$$\sum_{v} (1 + \text{degree}(v)) = n + 2m = O(n+m).$$

Correctness: $S = \{v :$ there is a path from s to v$\}$. ✓

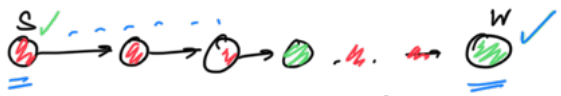
DFS(s): this will exactly visit all the vertices in S

the set of vertices visited by DFS(s) $\subseteq S$

(i) If DFS(s) visits a vertex v, then there is a path from s to v. → use induction



(ii) Let $w \in S$. w gets visited? proof by contradiction.



Suppose w is not visited by DFS(s).

DFS(x) was called. ✓

considered y. → DFS(y) gets called / DFS(y) is not called. visited[y] was true.

visited(y) is true.

How do we explore the entire graph?

```
for s = 1 ... n
    if (visited[s] is false)
        DFS(s)
```

DFS(1)

O(n+m)

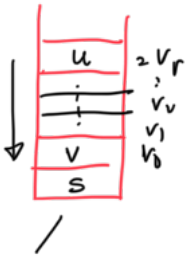
1 2 ... n

DFS(1)

DFS(1) ... DFS(8)

DFS(1) is called exactly once

Observations:



there must be a path from v to u.

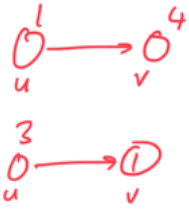
(v_i, v_{i+1}) is an edge.

$\forall \Rightarrow v_0 - v_r$ is a path.

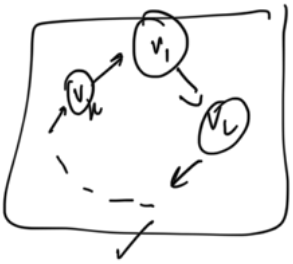
When we call DFS(u), : the stack gives us a path from s to u.

$s \rightarrow u$

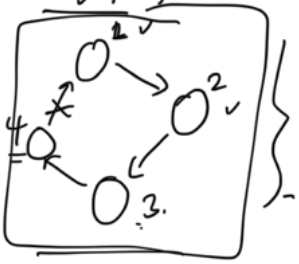
```
x = u;
while (x ≠ s)
    x = parent[x];
```



The edges (parent[u], u) for every u



subgraph of G

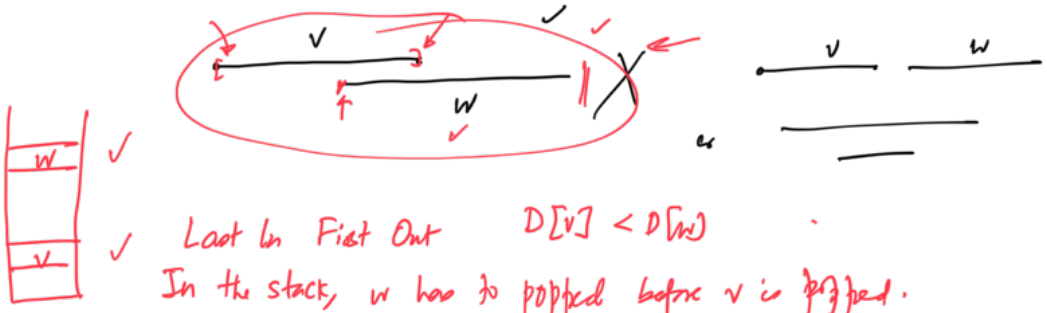


DFS tree is a subgraph of G.

- Kleinberg, Tardos : Algorithm Design.
- Whenever you use induction, what is the induction on?

Observations:

(1) If v and w are two vertices then the intervals $[D[v], F[v]]$, $[D[w], F[w]]$ don't cross.

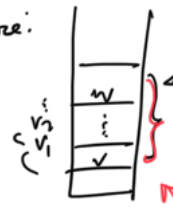


Last In First Out $D[v] < D[w]$

In the stack, w has to popped before v is popped.

$F[w] < F[v]$

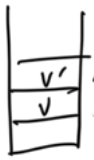
(2) DFS tree:



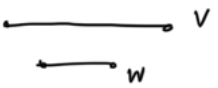
← at some point stack looks like this

Claim: v will be an ancestor of w in the DFS tree.

Is the converse true? Suppose v is an ancestor of w in the DFS Tree, does it mean that stack would have looked like this?



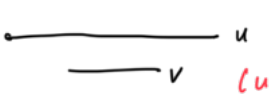
$\Rightarrow$ parent $[v'] = v$. $\Rightarrow$ in the DFS tree, v' is a child of v



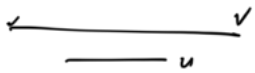
$\Leftrightarrow$ v is an ancestor of w in DFS Tree

(3) 

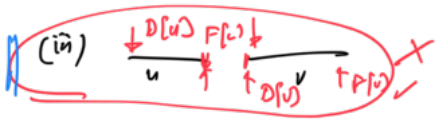
is an edge in G.

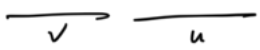
(i) 

FORWARD EDGE (u anc. of v)

(ii) 

BACK EDGE (v is anc. of u)

(iii) 

(iv) 

CROSS EDGE

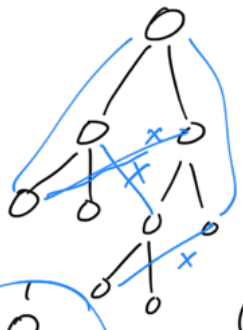
when we end DFS(u), v has not been visited

As a special case, if G is undirected, (i) ≡ (ii) (iii), (iv) ✗

all edges must be back edges.

Application:

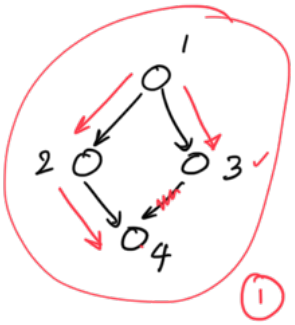
① Given a graph G (directed / undirected), how do we find a cycle?



DFS

Run DFS. if G has any edge other T

than T, it has a cycle.

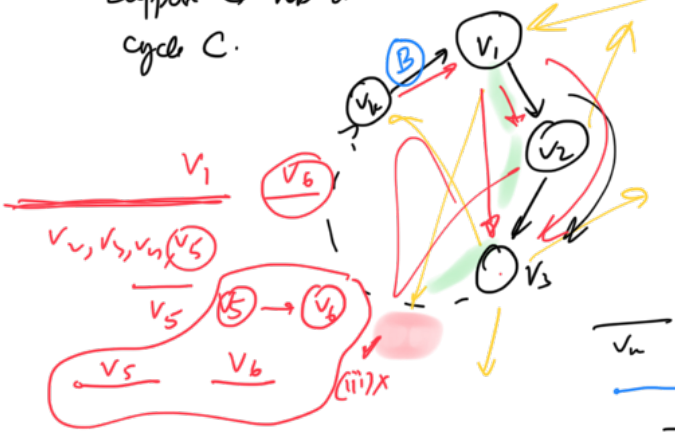


When do we conclude that G has a cycle?

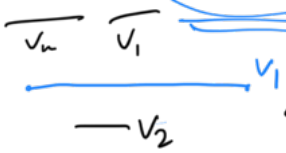
① If there is a back edge, then there is a cycle.

② Is the converse true? If there is a cycle, does there have to be a back edge??

Suppose G has a cycle C.



Suppose v_1 is the vertex with the smallest $D[v]$ value among all vertices in C.



← the intervals $v_2, \ldots, v_k$ ⊛ are all inside the interval for v_1.

(ii) $v_1 \longrightarrow v_2$ X

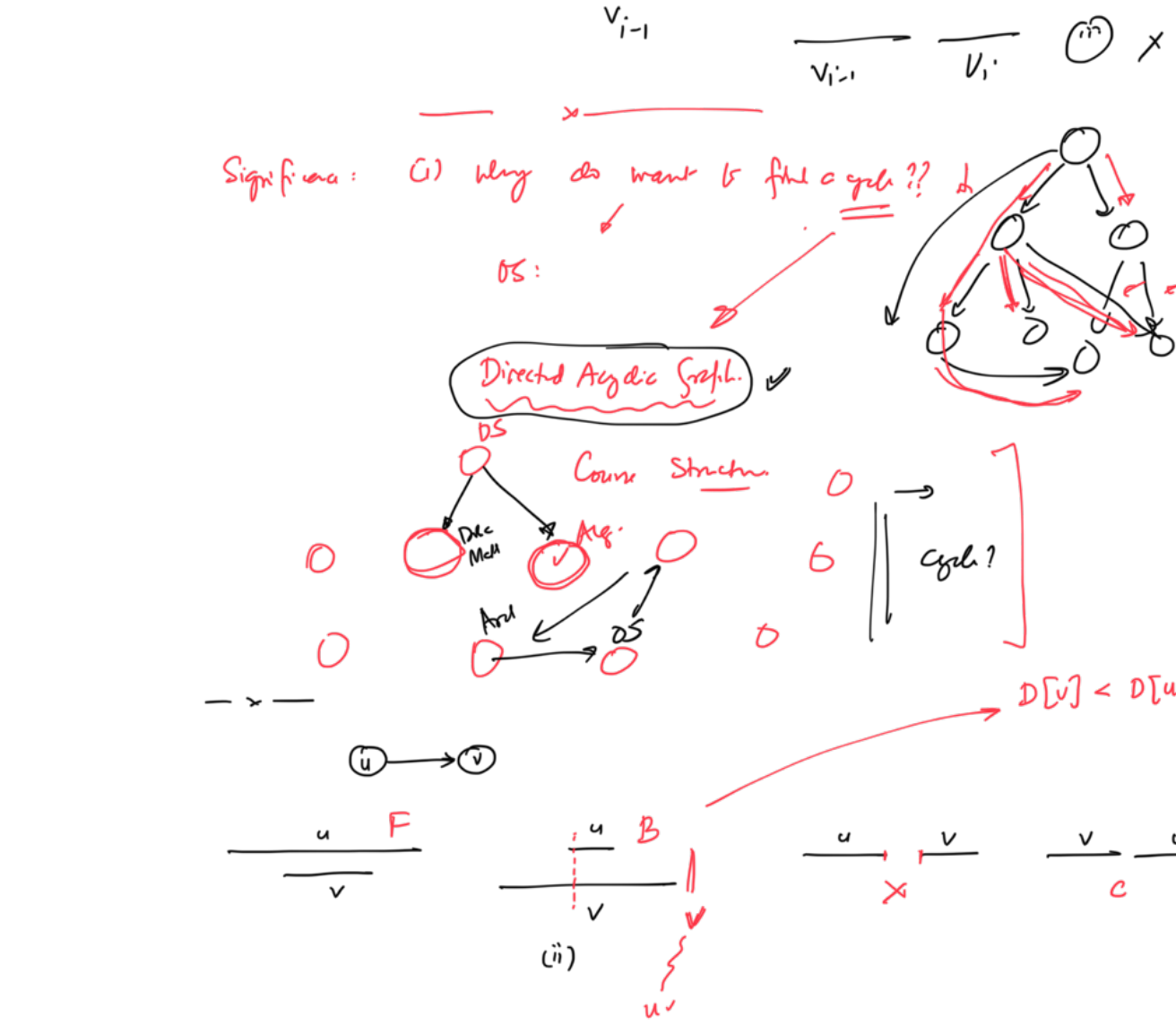
v_3 v_1 v_2 v_2

v_2 v_3 (iii)

Suppose ⊛ is not true ⇒

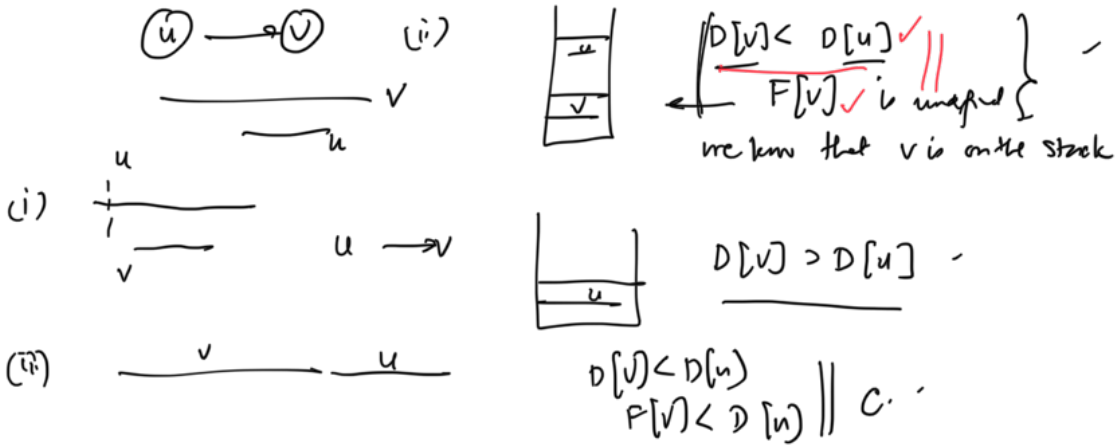
Let i be the smallest index such that

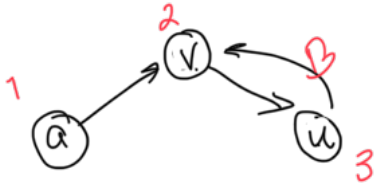
v_i v_1 $v_{i-1} \longrightarrow v_i$



G directed graph: cycle iff there is a back edge.

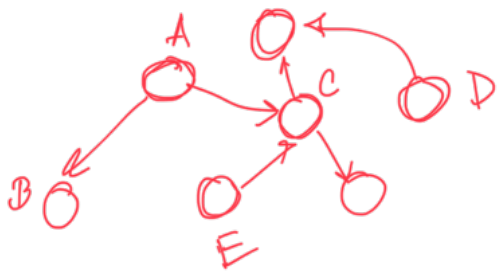
How to tell if an edge is C, B, F while performing DFS:



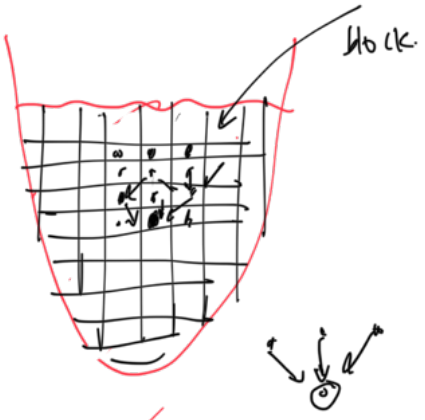


DFS(u) {
;
u → v
3 a

DAG:

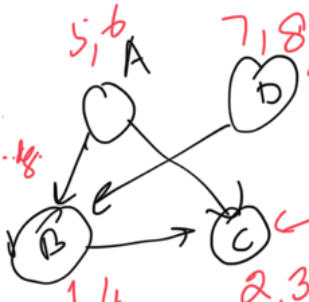


Compilers.



open pit mining.

Topological Sort:

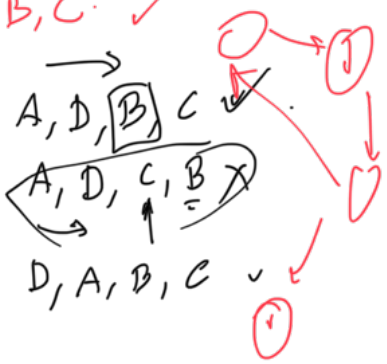


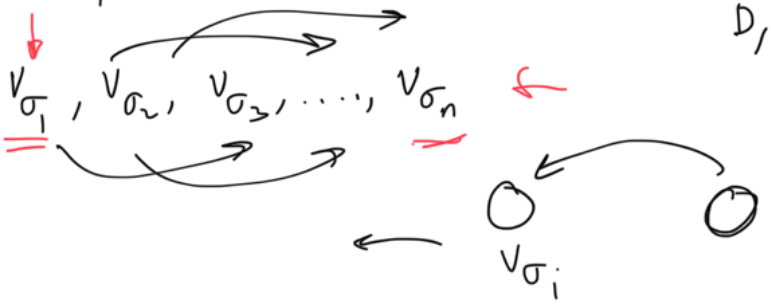
D, A, B, C. ✓

Sink: if its out-deg is 0.

Source: if its in-deg 0

$G = (V, E)$





Arrange the vertices s.t. that all edges go from left to right.

Perform DFS to visit all the vertices. ← D[v], F[v] value. time.

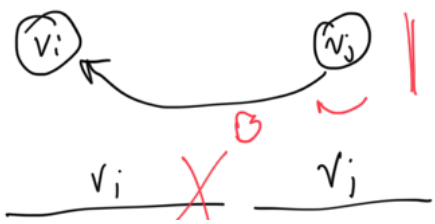
Arrange the vertices in dec. order of F[v] values.

sort? nlogn. O(n) time?

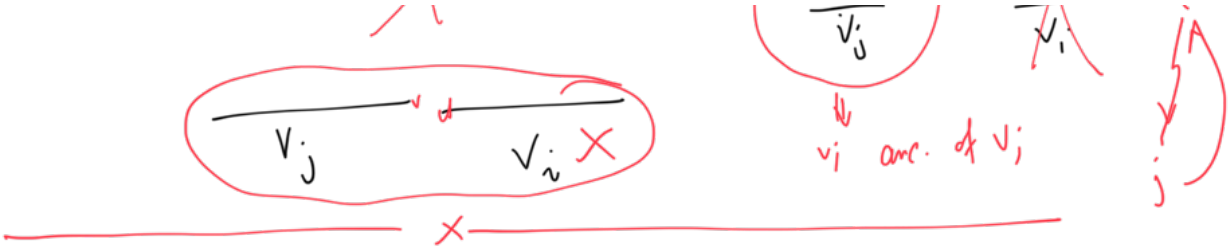
- Whenever a vertex is popped, add it to the front of a list ✓
- 1, 2n. n O(n). BUCKET

Why does it work?

$v_1, v_2, v_3, \ldots, v_n$: dec. order of F[v].

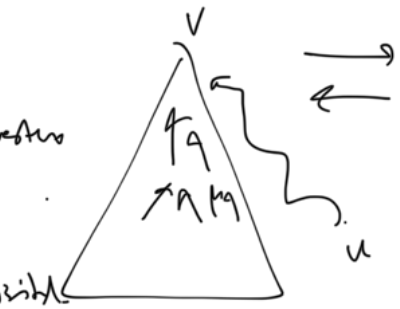


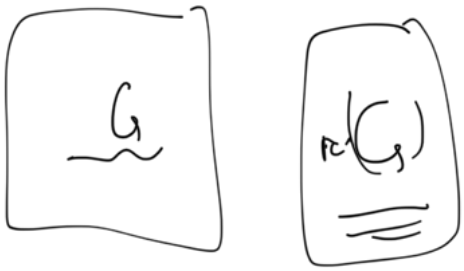
$j > i$ $F[v_i] > F[v_j]$



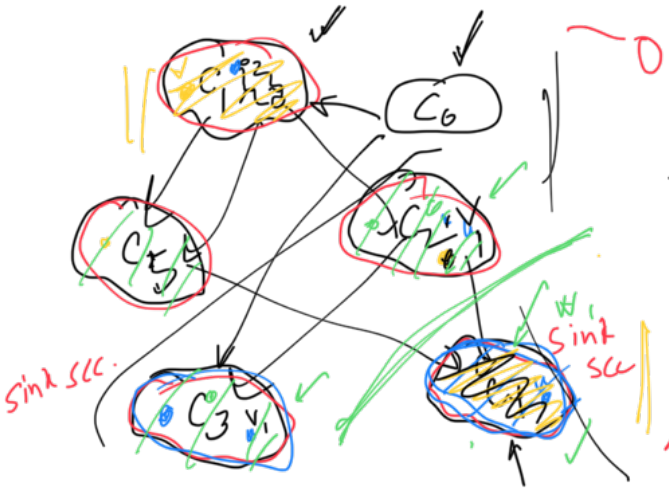
Strongly Connected ?

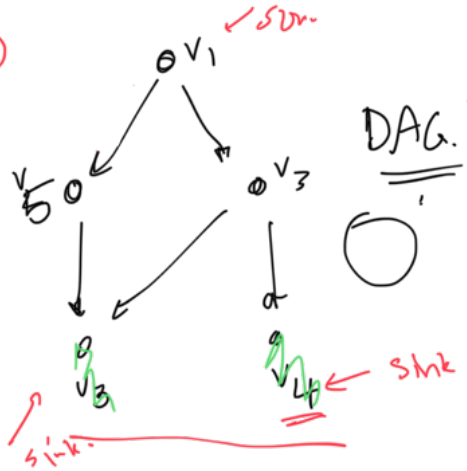
- Run DFS(v) and check that every vertex is visited.
- Reverse (G), DFS(v) : check that every vertex is visited.





SCC : strongly connected comp.





Kosaraju's alg. Suppose we perform DFS from u which is in a sink SCC.

How do we identify a vertex like this?

Run DFS. Look at the vertex with the highest F[v] value.

Always be in C_1 or C_6 : a source SCC.

v_1 v_2 . . . v_n.

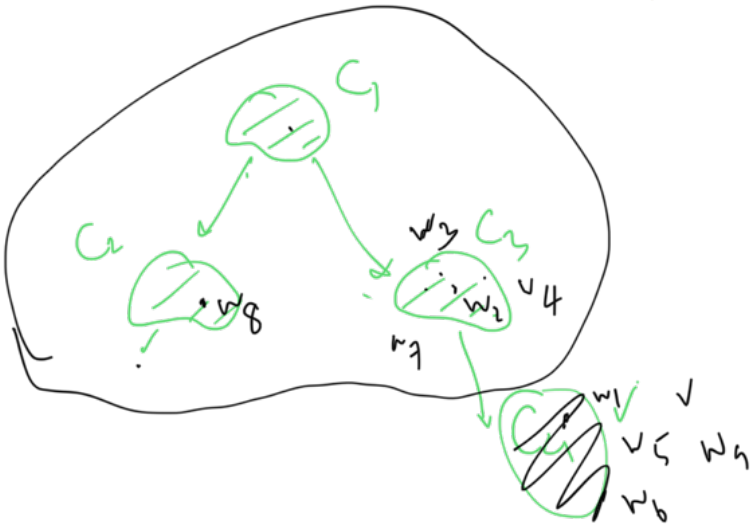
(1) G^{rev} : run DFS on this

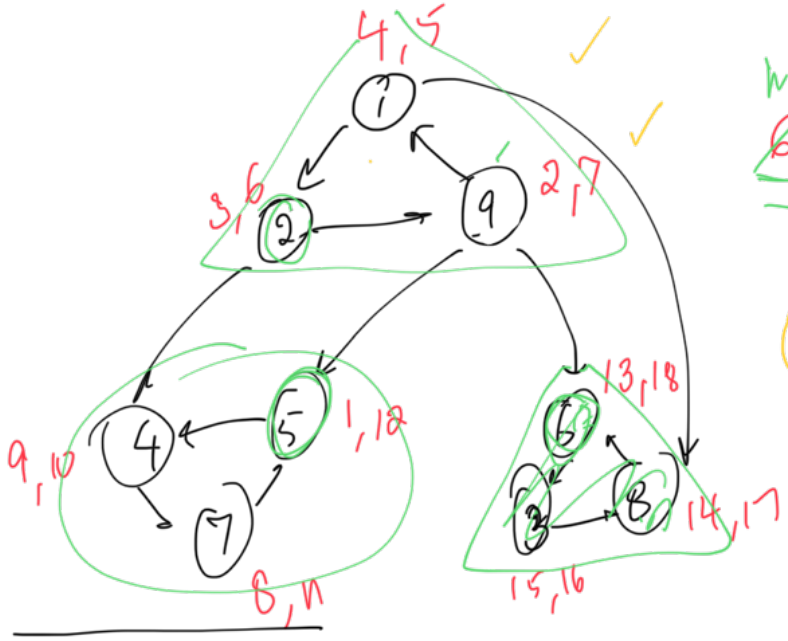
$w_1, w_2, \ldots, w_n$: dec. order of $F[v]$ values.

```
for i = 1 ... n
    if w_i is not visited
        DFS(w_i)
```

on G.

see which are the new vertices which are visited





w_1 w_2 w_3

6, 8, 3, 5, 7, 4, 9, 2, 1

6, 7, 8 ✓

$O(|E|)$: linear time

2/9/21

Size of the input = $O(|V|+|E|)$.

Q5.

G

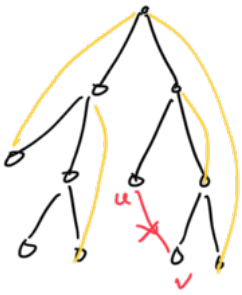
O(1) time.

- One more application of DFS for undirected graphs.
- BFS, connect with Dijkstra, ... Bellman Ford —

DFS for undirected.:

G: graph
T: DFS tree

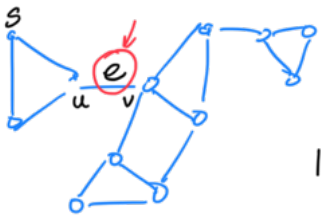
All edges in G − T : back edges.



„(V, E)
G: undirected
=

We say that an edge $e \in E$ is a "cut-edge" if removing e disconnects the graph.

An edge e is a cut-edge iff there is no cycle containg it.

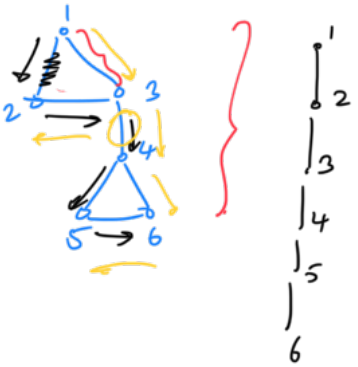


$|E| \geq |V| - 1$

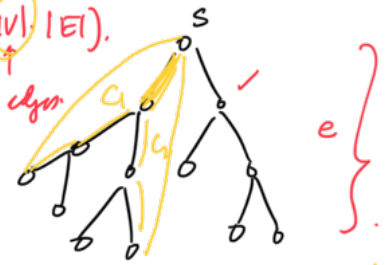
How to check if an edge e is a cut edge? $O(|V|+|E|)$ time using DFS.

How do we find all cut-edges?

$O(|E|^2) \rightarrow O(|V| \cdot |E|)$. ($|E|$ times, $|V|-1$ edges)

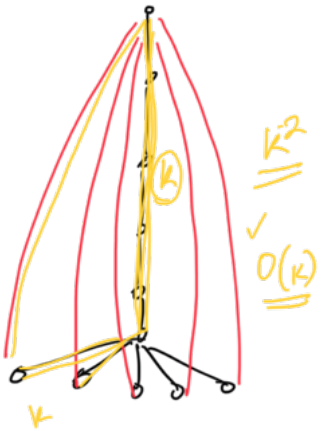


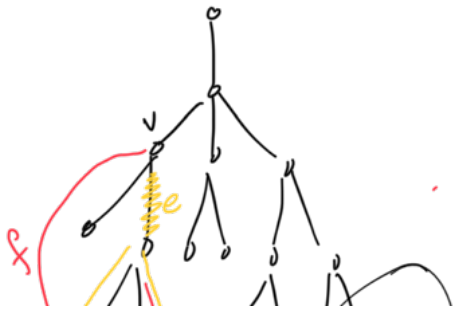
Run DFS:



Claim: Any edge $e \in G - T$ cannot be a cut-edge.

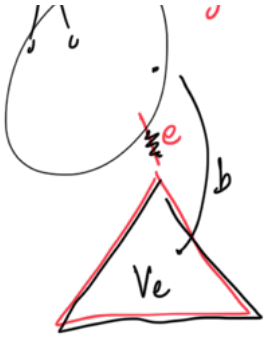
Q: Can we find all cut edges in linear time? $O(|V|+|E|)$





$D[v]$, $F[v]$.

- e is a cut-edge iff there is no non-tree edge which goes from V_e to outside V_e.

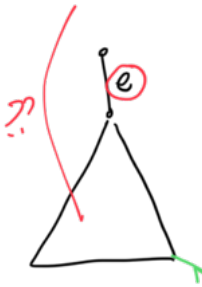


≡

e is a cut-edge iff there is a vertex $u \in V_e$ and $v \notin V_e$ such that $(u,v) \in E$.

back-edge

$D[v] < D[u]$.



linear time!

Do a traversal of the DFS tree and check this for every edge e.

post order / pre order

Suppose we know $f_1, \ldots, f_k$: which of them are cut-edge.

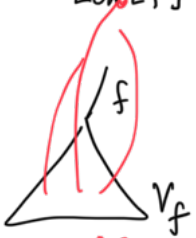
Can we figure out whether e is a cut-edge or not?

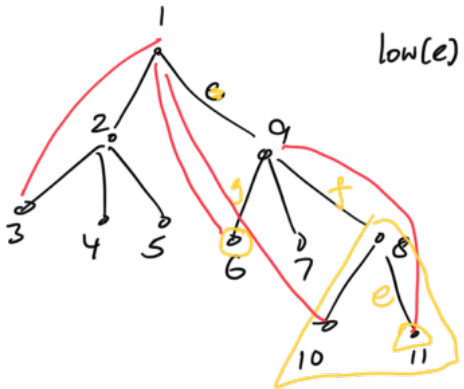
Is e cut-edge?

$f \in$ DFS-tree.

Low[f]: it is the highest vertex u such that there is a back edge (u,v), $v \in V_f$.

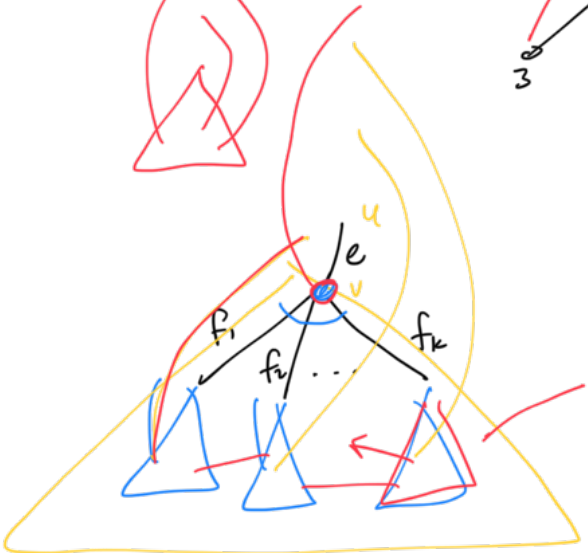
-1 if we cannot escape at all. (ie f is a cut-edge).





low(e)? = 1

Low(e)?
Low(f)



Suppose we know

w_1 w_2 w_k

Low[f_1], Low[f_2], ..., Low[f_k]

Can we figure out Low[e]?

Highest among D[ ] and the back edge leaf v

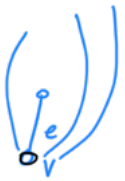
Let w_j be the one with min D[w] value

⊛ Low(e) = w_j if $w_j \neq v$

$w_j \equiv v$?

$= -1$ if $w_i = v$ $\Big\{$ Low[e] $= -1$.

w_i : look at all the edge f_j where Low$[f_j] = -1$ among them pick the vertex with min. D[] value $\Big\}$.



Compute Low(e) $\{$

$\{$ if e is a leaf edge (u,v)

Low[e]: the highest back edge goes out of v.

otherwise

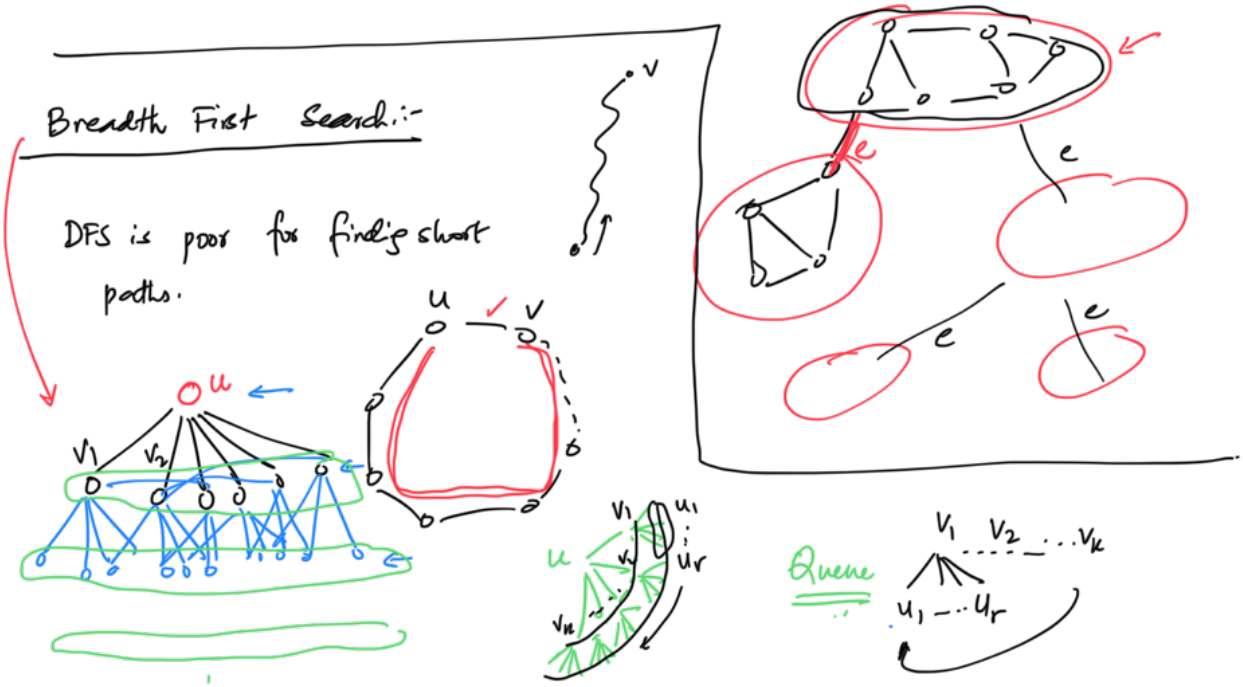
Compt Low(f_1), ... Compt. Low(f_k), Lo

Low(i) = ⊛, highest back. edge goes out of v

$\}$

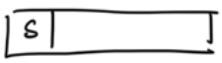
Output all edges with low(i) $= -1$. $O(|V|+|E|)$ time.

Bi-connected: a graph is said to be biconnected if there is no cut. edge.



Queue. ← Enqueue, Dequeue.

Q



Enqueue(s); visited[s] = true. Level(s) = 0.

At each step:

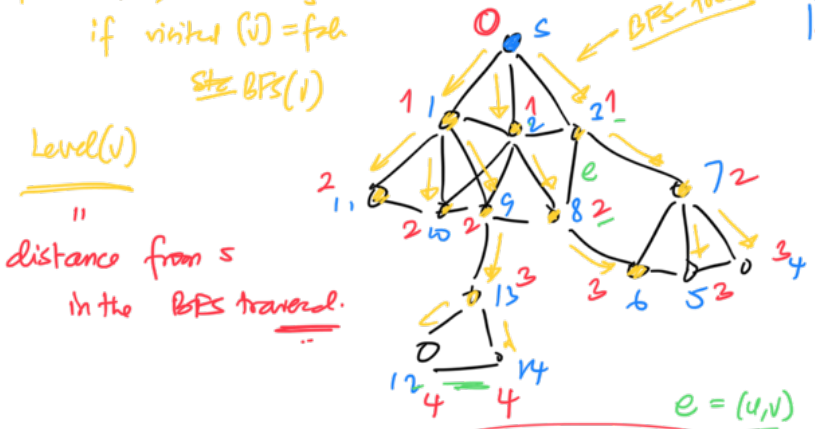
BFS (⊛) repeat $\{$ v = Dequeue(Q) (visited[w] = false) ← $|V|+|E|$

$\{$ ... $\leq$ (...)

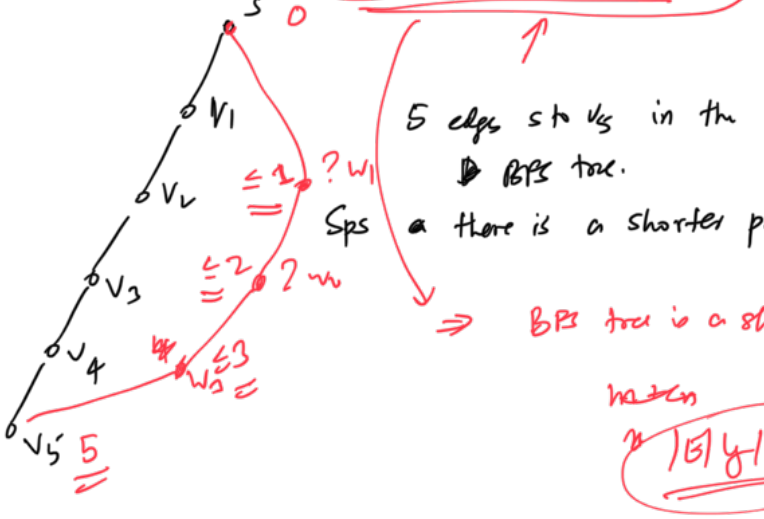
till Q is empty { for all neighbours w of v { Enqueue(w), visited[w] = true. Level(w) = Level(v)+1; p[w] = v; } }

parent of.

for v = 1, ..., n { if visited(v) = false BFS(v) }



Level(v) = distance from s in the BFS traversal.



matches $\leq |E| + |V|$

Maximal: Longest