Graphs, Matrices and Algorithms 29/Sep/25

I'm SUBRAHMANYAM KALYANASUNDARAM
Dept of CSE, IIT HYDERABAD.
EMAIL: SUBRUK@CSE.IITH.AC.IN

I'll be discussing some basics of graph algorithms over the course of 4 lectures.

Algorithm: A well defined computational procedure that takes some values as input and produces some value or values as output.

Could be decision problem (does this graph have a triangle?) or function computation (given a graph, color graph using the smallest no. of colors).

Lots of applications : Graphs can be used to model different kinds of relations, networks and it could be useful to compute the parameters / properties of such graphs. (Is a graph connected ? Does a graph have a specific subgraph ? Find a shortest path from X to Y.)

How do you measure the performance of algorithms ?

→ An algorithm is a description of a process.

→ The most common resources of computation are time and space.

→ Time will vary with hardware (processor speed, for instance) and with input size.

Say, we want to sort n numbers.

Machine A: Can execute 10^{10} inst/sec

Running algorithm that takes $2n^2$ inst.

Machine B: 10^7 instructions/second.

Running a more efficient algorithm that takes: $50 n \log_2 n$ instructions.

How long does each machine take to sort $n = 10^7$ numbers?

Machine A: $2 * (10^7)^2 / 10^{10} = 20000$ sec

$= 5\frac{1}{2}$ hours (roughly)

Machine B: $\frac{50 * 10^7 * \log_2 10^7}{10^7} \simeq 1163$ sec

(less than 20 min!)

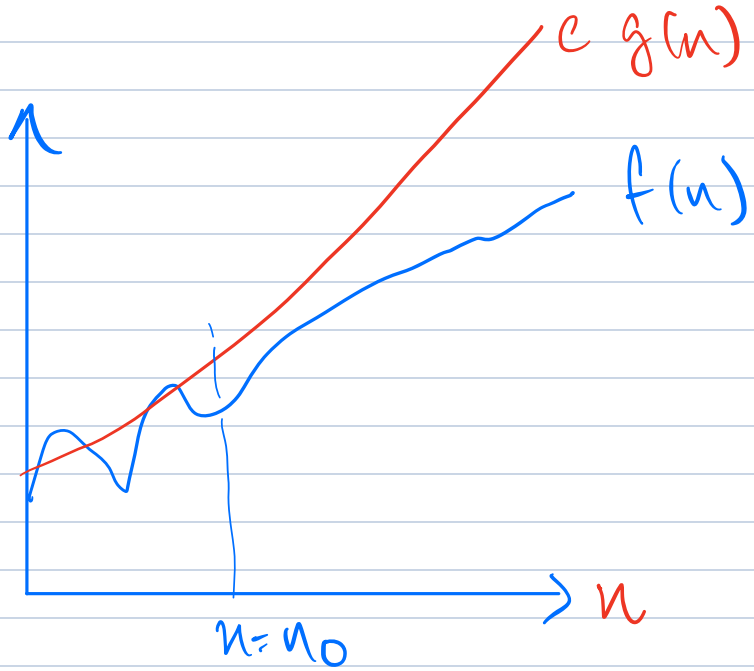
Asymptotic notation (O notation)

Used to understand the rate of growth of a function

→ Ignores constants, and lower order terms.

We say $f(n)$ is $O(g(n))$ if there exist constants $c, n_0 > 0$ such that

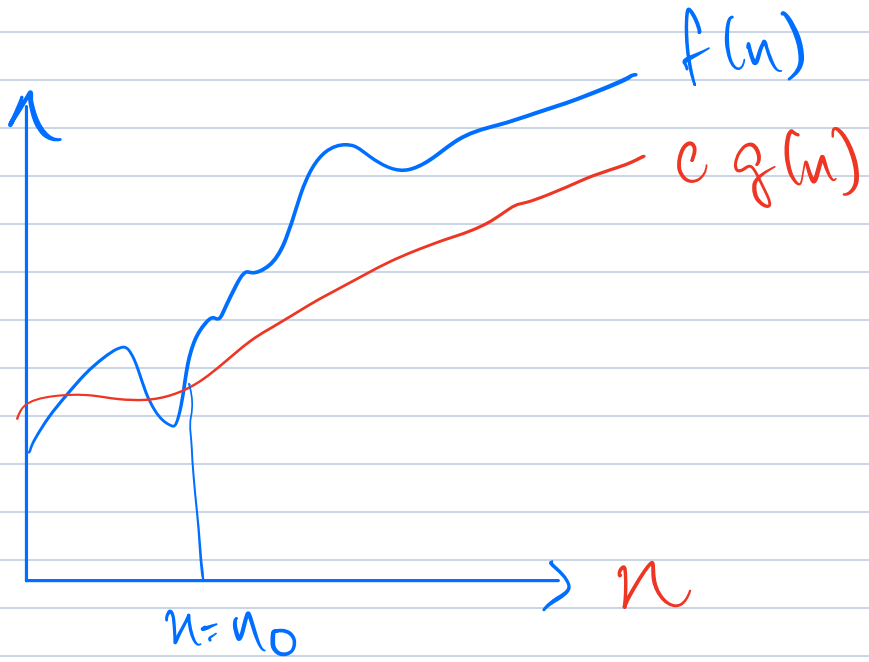
$$0 \leq f(n) \leq c\, g(n) \quad \forall\, n \geq n_0.$$



$f(n)$ is $O(g(n))$

We say $f(n)$ is $\Omega(g(n))$ if there exist $c, n_0 > 0$ such that

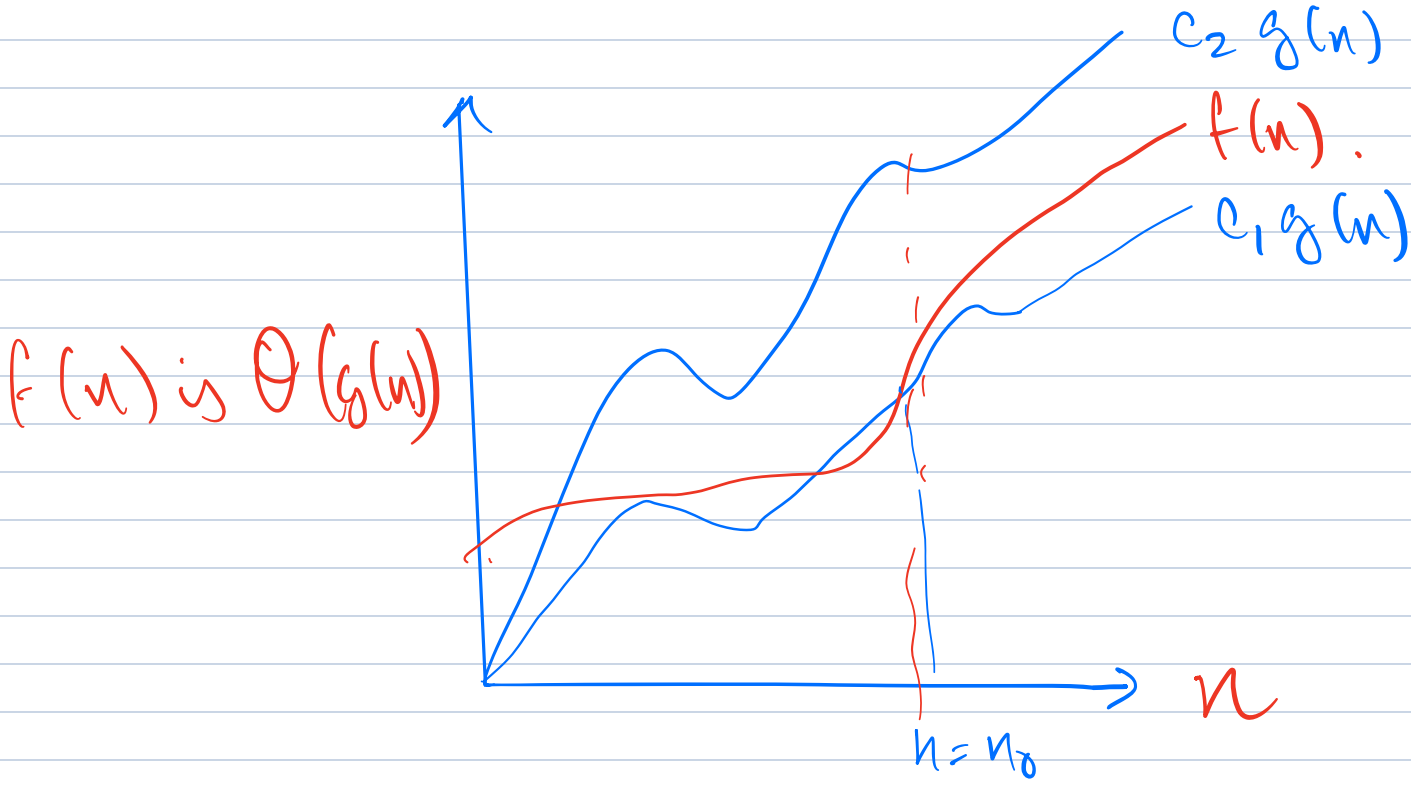
$$0 \leq c\, g(n) \leq f(n) \quad \forall\, n \geq n_0$$



$f(n)$ is $\Omega(g(n))$

We say $f(n) = \Theta(g(n))$ if there exist $c_1, c_2, n_0 > 0$ such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall\, n \geq n_0.$$



Θ : Asymptotic tight bound.

O : Asymptotic upper bound

Ω : Asymptotic lower bound.

→ Little o

We say $f(n) = o(g(n))$ if for any $c > 0$, there is $n_0 > 0$ such that

$$0 \leq f(n) \leq c\, g(n) \quad \forall\ n \geq n_0$$

Another way to think is in terms of limits: $f(n)$ becomes insignificant in

terms of $g(n)$ as $n \to \infty$.

$$\lim_{n \to \infty} f(n)/g(n) = 0.$$

Examples

① $5n^2 + 100n + 10$ is $\Theta(n^2)$

→ $o(n^3)$

② $n\sqrt{n} + 20n$ is $\Theta(n\sqrt{n})$

→ $o(n^2), o(n^3), o(n^{100})$

③ $n^3 + 3^n$ is $\Theta(3^n)$

④ $n^2 \log n + n^2 + n(\log n)^2$
is $\Theta(n^2 \log n)$

⑤ $100 n^2 \log n + 1000 n^2 + 10000 n(\log n)^2$
is $\Theta(n^2 \log n)$

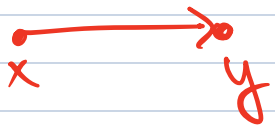
(6) $n^2/\log n + n^{1.99}$ is $\Theta(n^2/\log n)$

$\frac{n^2}{n^{0.01}}$

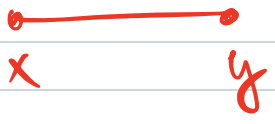
Graphs

We have $G = (V, E)$ a graph, where V is the set of vertices and E is the set of edges.

→ Directed graph: The edges are directed and are ordered pairs (x, y) where $x, y \in V$

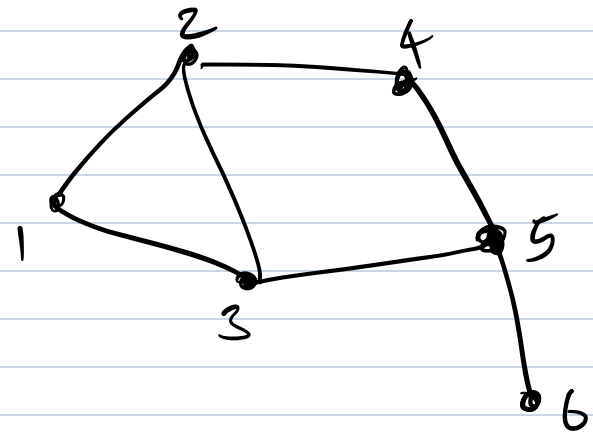


→ Undirected graph: The edges are unordered pairs $\{x, y\}$ where $x, y \in V$



The application will dictate whether the graph is directed or undirected.

How do we represent graphs?



Adjacency matrix

We have an $n \times n$ matrix where $n = |V|$.

The $(i,j)^{th}$ entry is $1 \iff (i,j) \in E$ there is an edge from i to j

Else the $(i,j)^{th}$ entry is 0

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

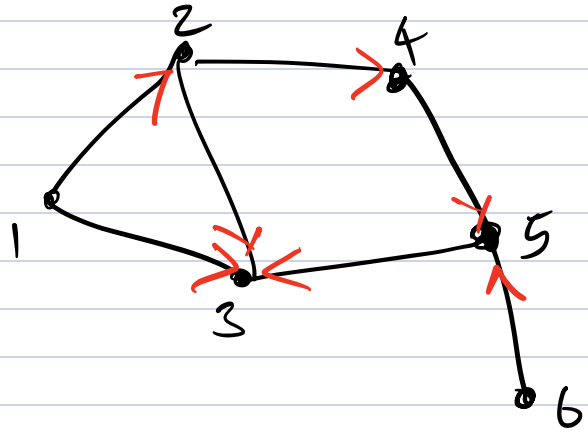
Adj matrix needs $O(n^2)$ space.

But we can immediately say if (i,j) is an edge. $\rightarrow$ Done in $O(1)$ time.

constant

The matrix A is symmetric if and only if the graph is undirected.

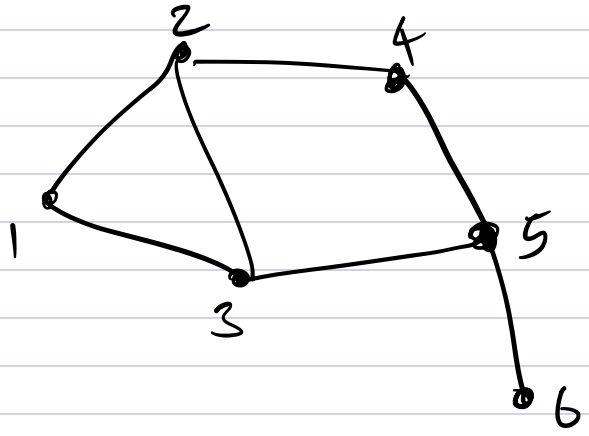
What is the adjacency matrix of this directed graph?



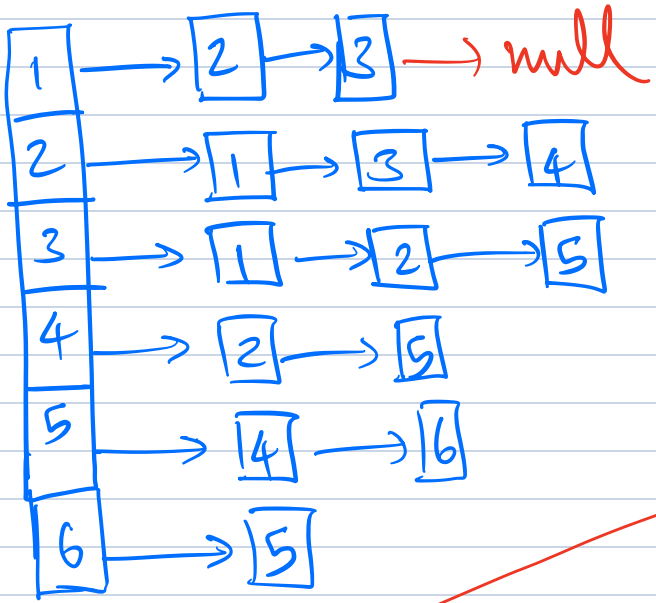
$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$(i,j)^{th}$ entry is 1 if there is an edge from vertex i to vertex j

Adjacency list



This is a different way to represent a graph. Here we have $n = |V|$ linked lists, one for each vertex. Each one contains the list of (out)-neighbors of a vertex.

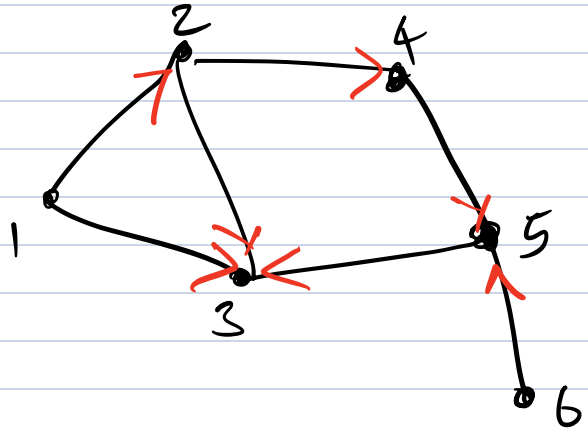


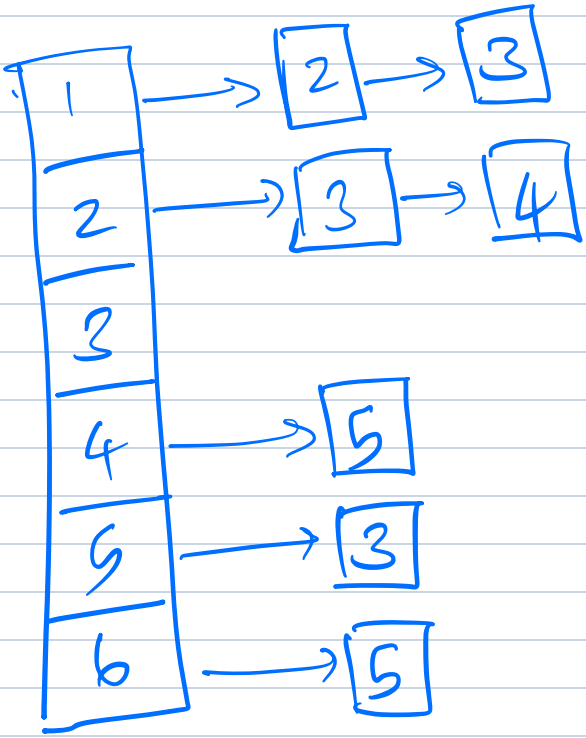
Max no. of edges in an n vertex graph = $\binom{n}{2}$ = $\frac{n(n-1)}{2}$

This takes $O(|E|)$ space. (In an undirected graph, each edge appears two times).

But checking if (i, j) is an edge may not be immediate. It can take $O(|V|)$ time.

What is the adjacency list of this directed graph?





Which representation is better?

Adjacency matrix takes $O(|V|^2) = O(n^2)$ space. But can immediately check adjacencies.

$O(1)$ time

Adjacency list takes only $O(|E|)$ space but can take up to $O(|V|)$ time to check adjacencies.

If $|E|$ is $O(|V|^2)$ then adjacency matrix may be better as adj. list also takes $O(|E|) = O(|V|^2)$ space.

If we have a sparse graph, i.e., where $O(|E|) = O(|V|)$ or $O(|V| \log |V|)$

(something smaller than $O(|V|^2)$)

then it may help to use adj. list.

Graph Exploration

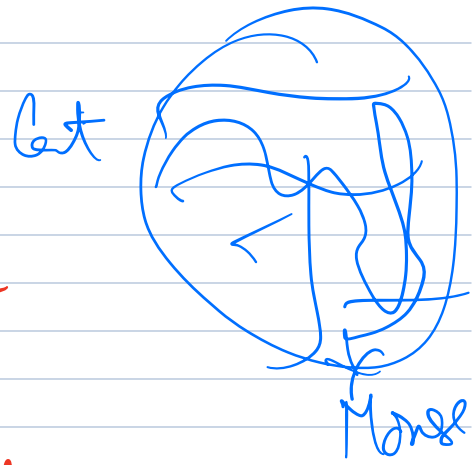
We first consider one of the simplest questions one can ask about a graph.

Q: If we start from a vertex $s \in V$, which other vertices are reachable?

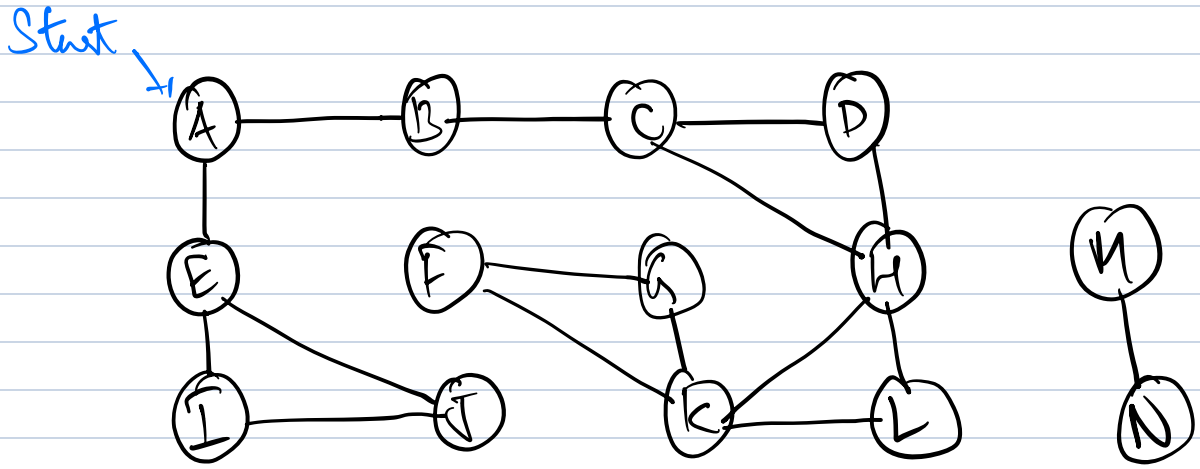
Suppose you are in a maze, how can you find which are the reachable vertices?

→ Rope and Chalk

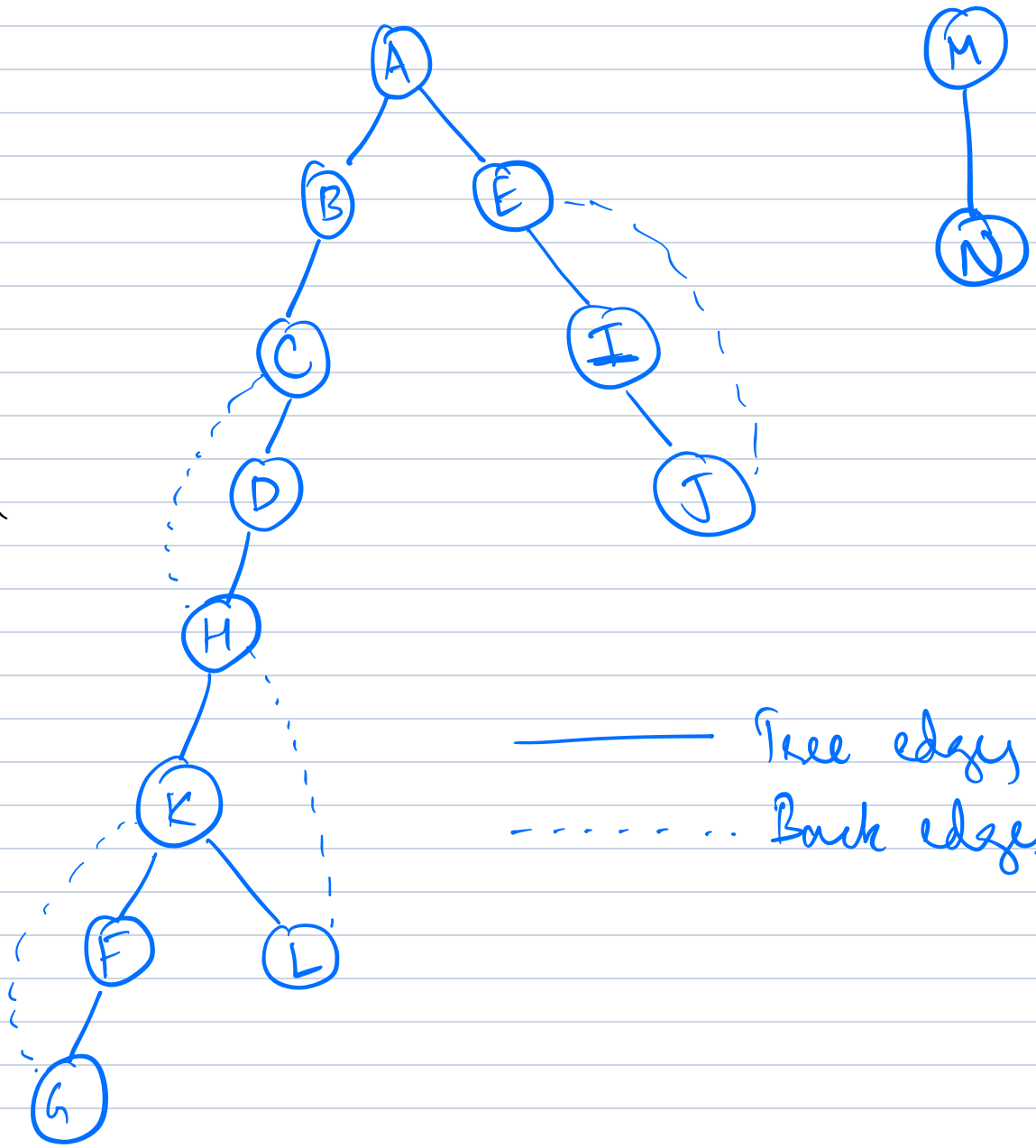
- to return to starting point
- to mark visited nodes.



Can we do the same with a computer?



Depth First Search (DFS)



Depth First Search (DFS)

(We'll later see Breadth First Search)

```
DFS(G)
For all v ∈ V                          } O(|V|)
    visited(v) = false

For all v ∈ V
    If visited(v) = false, Explore(v)
```

```
Explore(w)
visited(w) = True
Pre visit(w)
for each edge (w,z) ∈ E
    If visited(z) = false, Explore(z)
Post visit(w)
```

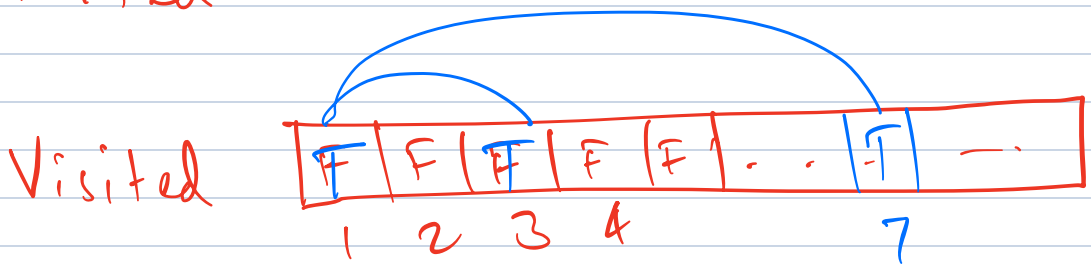
Explore(w) → Discovers all the vertices that are reachable from w.

(w,z) ∈ E → Outgoing edge from w

Pre visit(w) ← This will be defined based on need.

Post visit(w) ← This will be defined based on need.

To execute this, we need an array visited



and a representation of the graph.

Explore (w) determines all the vertices reachable from w that have not already been explored.

Correctness: We can prove the following using induction on k.

> In Explore(w) all vertices that are reachable in $\leq k$ steps from w get visited.

Running time analysis:

Initializing all vertices as not visited : $O(|V|)$

Inside explore(w), (assume previsit, to be $O(1)$ post visit

setting visited to True : $O(1)$.

Each edge (x,y) in G is checked exactly twice,

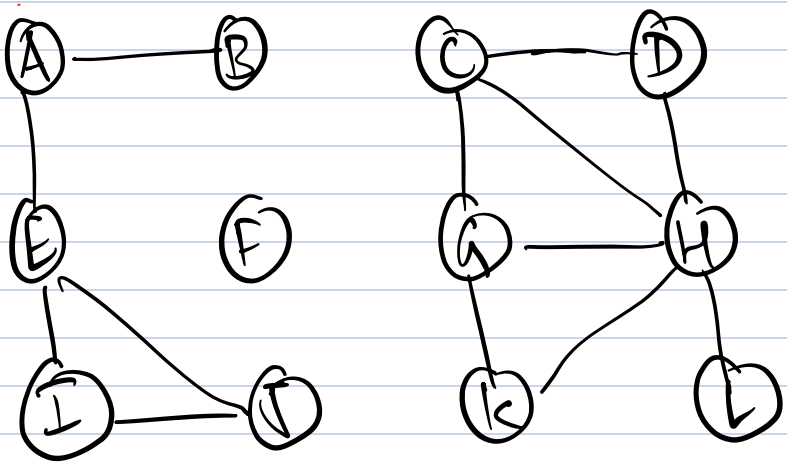
nee during explore(x) and nee during explore(y). This takes $O(|E|)$ time, nee the course of the DFS.

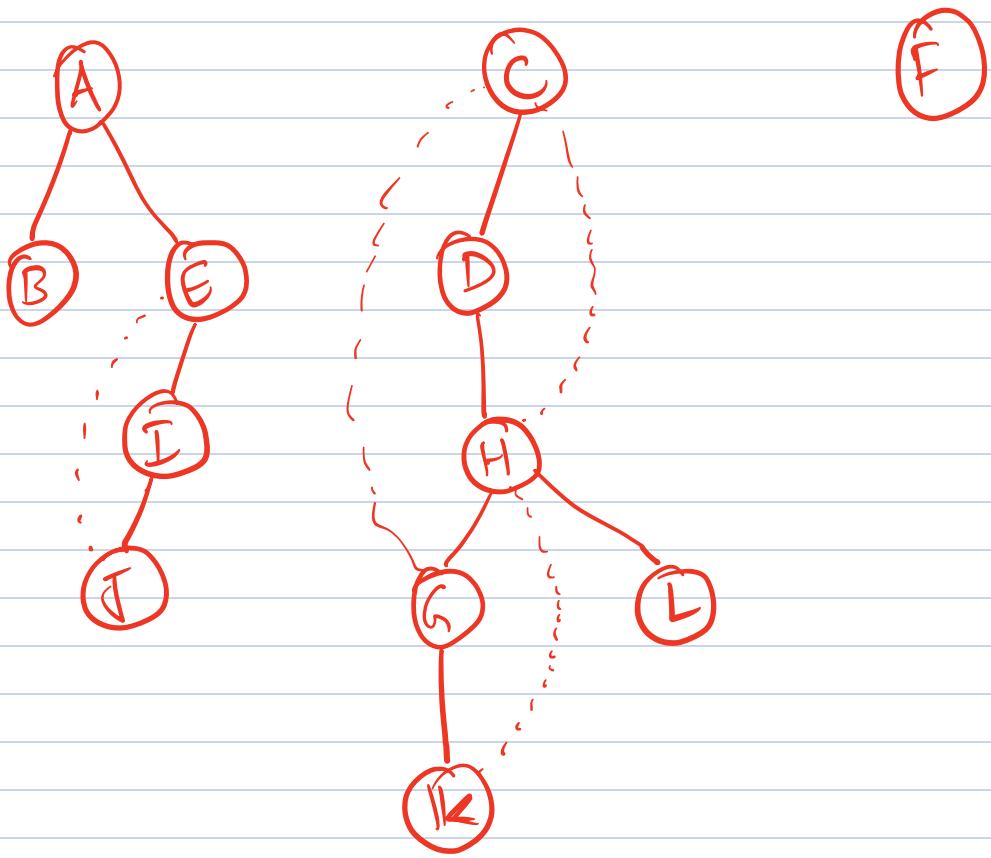
Assuming graph G is given in adj. list format, this takes $O(|V|+|E|)$ time.

Why is this needed?

If G is given as adj matrix, we will need $O(|V|^2)$ time

One application: Want to identify which vertex belongs to which connected component.





CCnum

1	1	2	2	1	3	2	2	1	1	2	2
A	B	C	D	E	F	G	H	I	J	K	L

We modify DFS as follows.

DFS(G)

For all $v \in V$

    visited(v) = false
    ccnum(v) = Empty

cc = 0

For all $v \in V$

    If visited(v) = false

        cc = cc+1
        Explore(v)

Explore(w)

Visited(w) = True

ccnum(w) = cc

For each edge $(w, z) \in E$

    If visited(z) = false, Explore(z)

The running time remains $O(|V|+|E|)$

Previsit and Postvisit orderings

Initially, we set a clock to one. Then we increment it when we begin exploring a vertex and completing the exploration.

Explore(w)

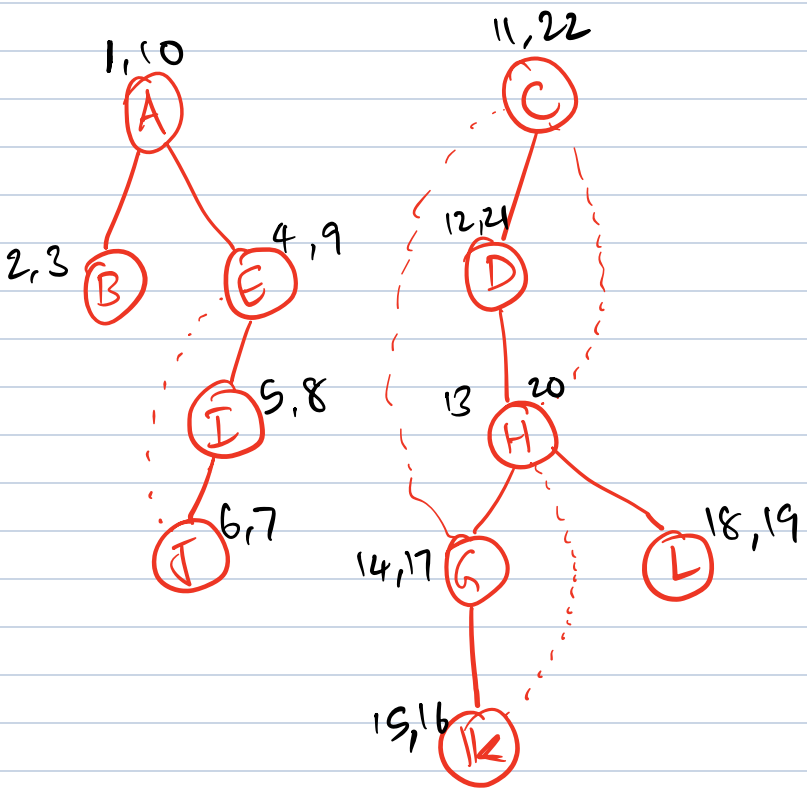
Visited(w) = True

Previsit(w): Pre(w) = clock; clock = clock + 1

for each edge (w, z) ∈ E

If visited(z) = false, Explore(z)

Postvisit(w): Post(w) = clock, clock = clock + 1



[Pre(v), Post(v)]

Observation: For any two $u, v \in V$, $[Pre(u), Post(u)]$ and $[Pre(v), Post(v)]$ are either disjoint or one is contained in the other.

Suppose WLOG, explore(v) is called after explore(u)

Either explore(v) is called as a descendant of explore(u), or after explore(u) is completed and exited.

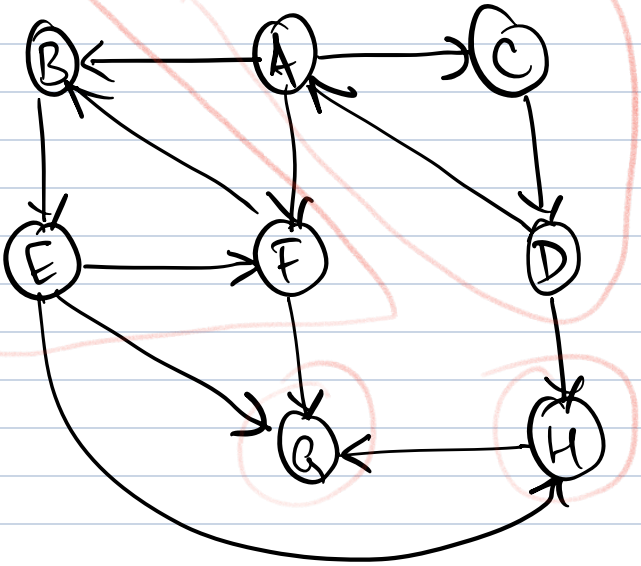
(In other words, is u still in the recursion stack while explore(v) is called?)

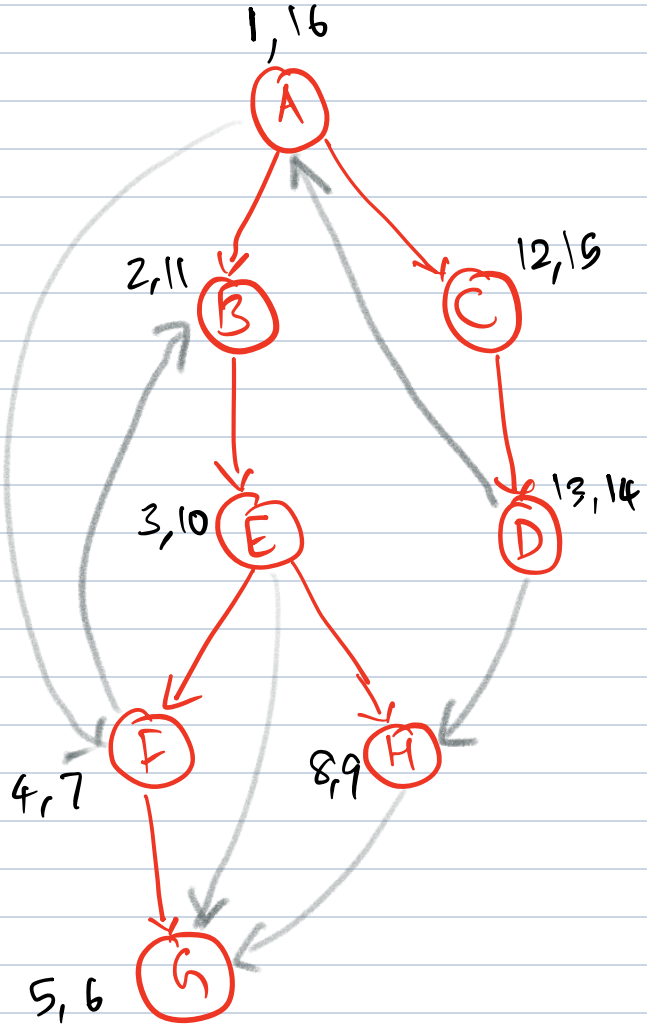
In the former case, $[Pre(v), Post(v)]$ will be contained in $[Pre(u), Post(u)]$.

In the latter case, they will be disjoint.

DFS in Directed Graphs.

In undirected graphs, we just had DFS edges and back edges. Here we will have other type of edges.



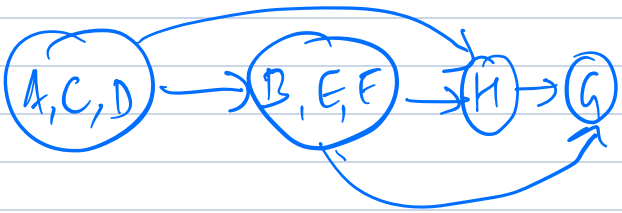


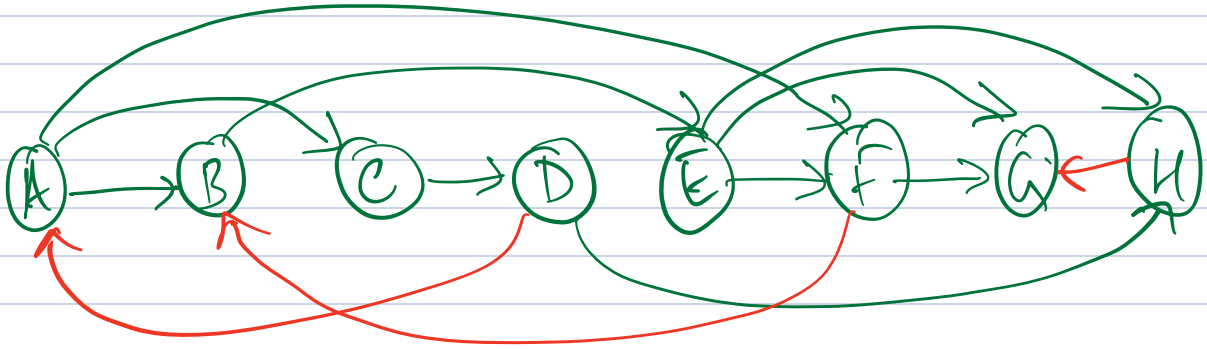
Types of other edges

1. Forward edges: Ancestor to non child descendant
2. Back edges: Descendant to ancestor
3. Cross edges: To neither descendant nor ancestor.

→ Tree edges

→ Other edges.





Question: Is there an ordering of the vertices of this graph so that all the edges go from left to right? → Such an ordering is called topological sort.

No. It's not possible for this graph, since it contains a cycle. So we will need some edge that goes from right to left.

What if a given graph G, does not have a cycle?

↳ Does G necessarily have a topological sort?

How can we identify the edge types?

① For edge (u, v), it is a forward edge or tree edge if ..

$pre(u) < pre(v) < post(v) < post(u)$

v is in u's subtree

② Edge (u, v) is a back edge if

$pre(v) < pre(u) < post(u) < post(v)$

u is in v's subtree.

③ Edge (u, v) is a cross edge if

$pre(v) < post(v) < pre(u) < post(u)$.

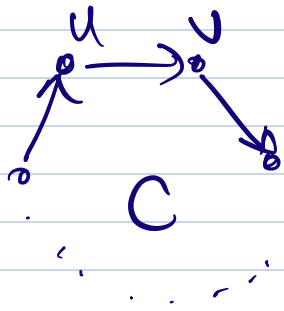
DFS exits from v before it explores u.

Property: A directed graph G has a cycle if and only if DFS tree has a back edge.

Proof: ($\Leftarrow$) Suppose DFS tree has a back edge (u,v). Then u is a descendant of v in the tree. The path from v to u along with edge (u,v) forms a cycle.

($\Rightarrow$) Suppose G has a cycle C. Suppose v is the first vertex in C to be explored by the DFS. So all the other vertices in C are descendants of v in the BFS tree.

Suppose u is the vertex in C that precedes v in the cycle. Then the edge (u,v) is a back edge in the DFS tree.



Want to determine if a given directed graph has a cycle. What is a property that we can use?

Clue: look at the post numbers.

Property: If $\overrightarrow{u,v}$ is a tree edge, forward edge, or cross edge, then post(v) < post(u)

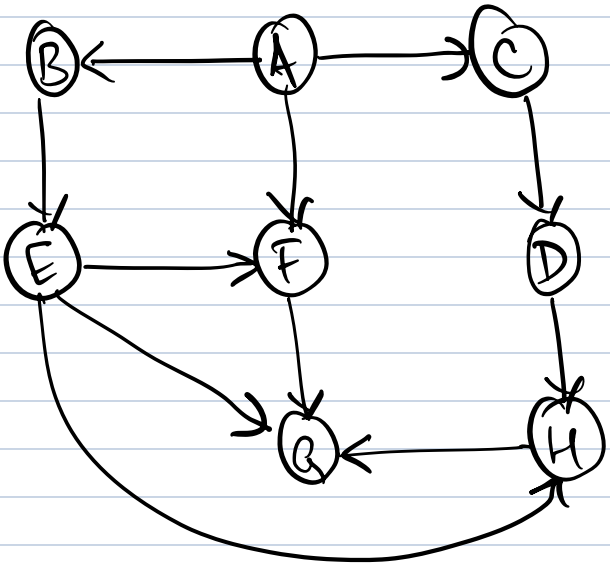
If $\overrightarrow{u,v}$ is a back edge, then post(u) < post(v)

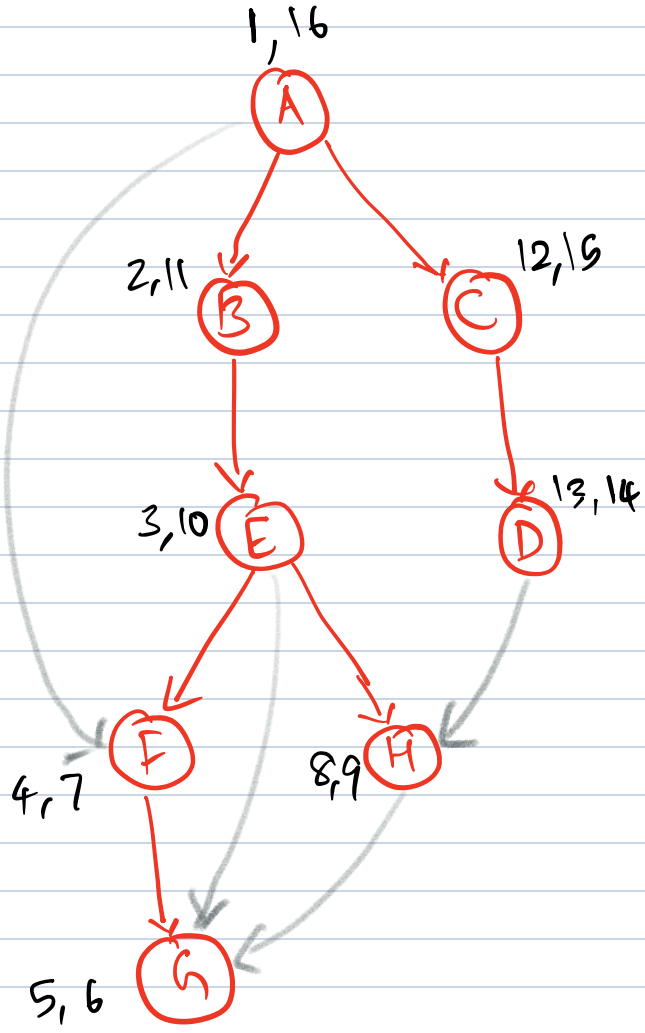
To check if the directed graph has a cycle, we can just check if there is an edge (u, v) with $post(u) < post(v)$.

Topological sort.

Given a DAG (Directed Acyclic Graph), we want to order the vertices such that all the edges are from a vertex earlier in the ordering to a vertex later in the ordering.

HOW CAN WE DO THIS?

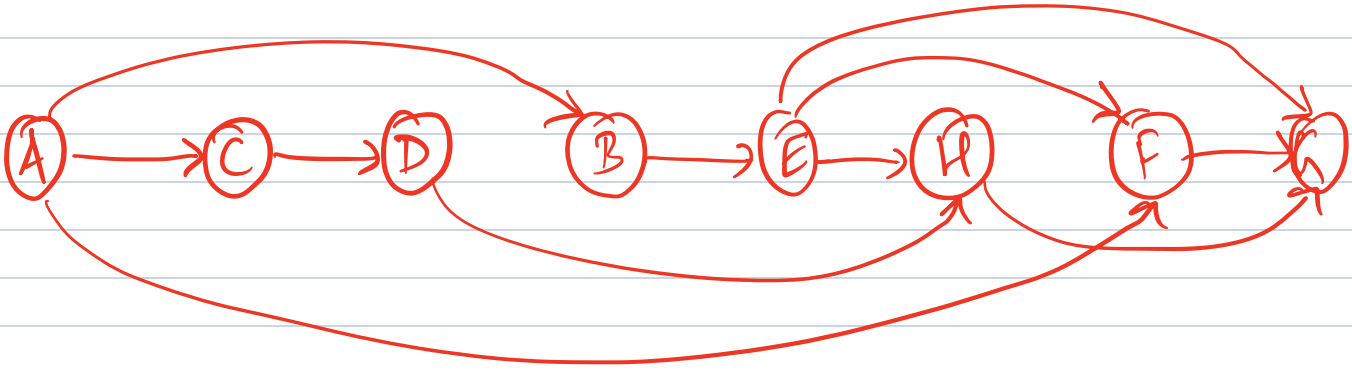




no back edges in DAG!

We already saw that for all edges (u, v) in a DAG, post $(v) <$ post (u).

So we can sort in the descending order of post(·) numbers!



Note: This need not be unique. There may be other such orderings.

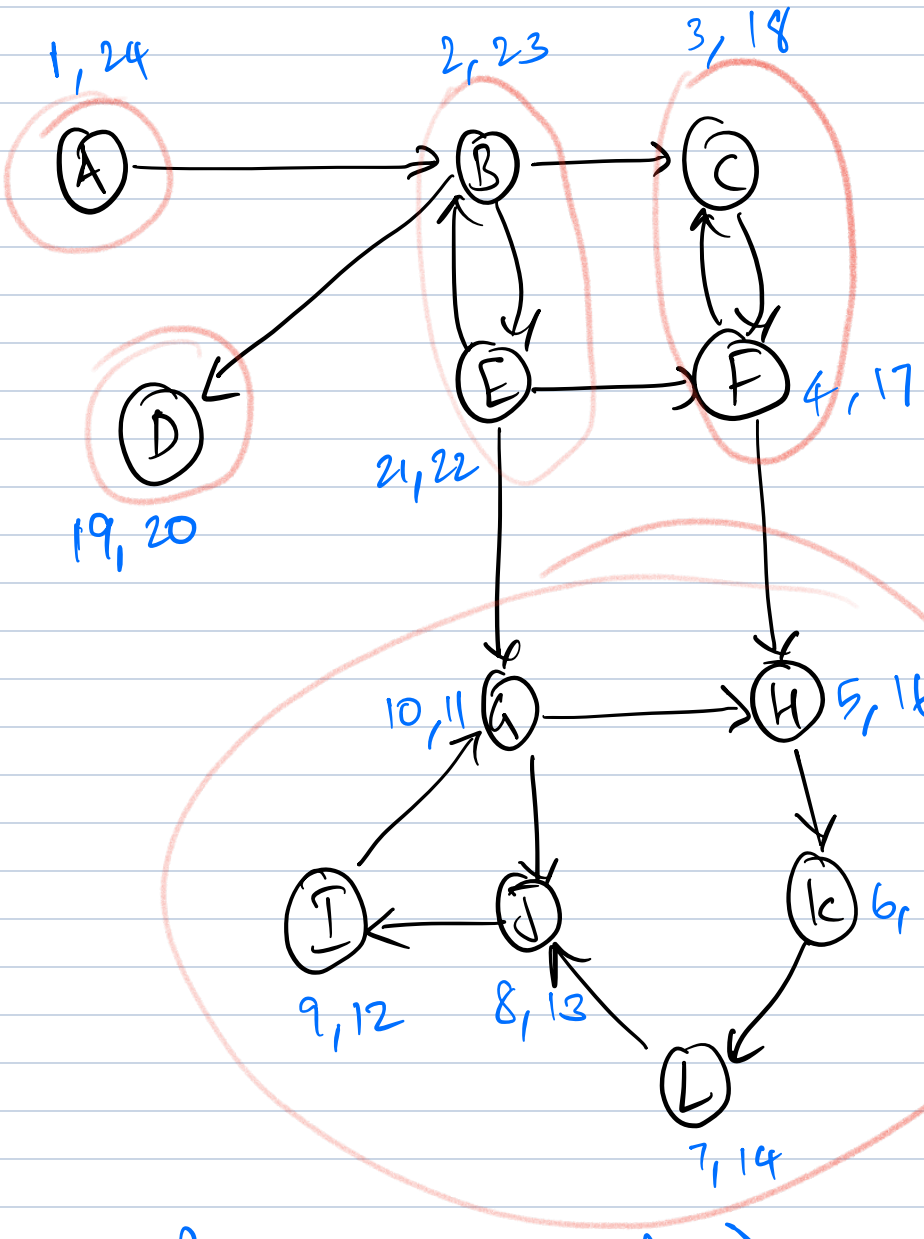
Strongly Connected Components

How can we generalize connected components for directed graphs? It is not as easy or straightforward as undirected graphs.

We say a set $S \subseteq V$ is strongly connected if for any $u, v \in S$, there is a directed path from u to v AND there is a directed path from v to u.

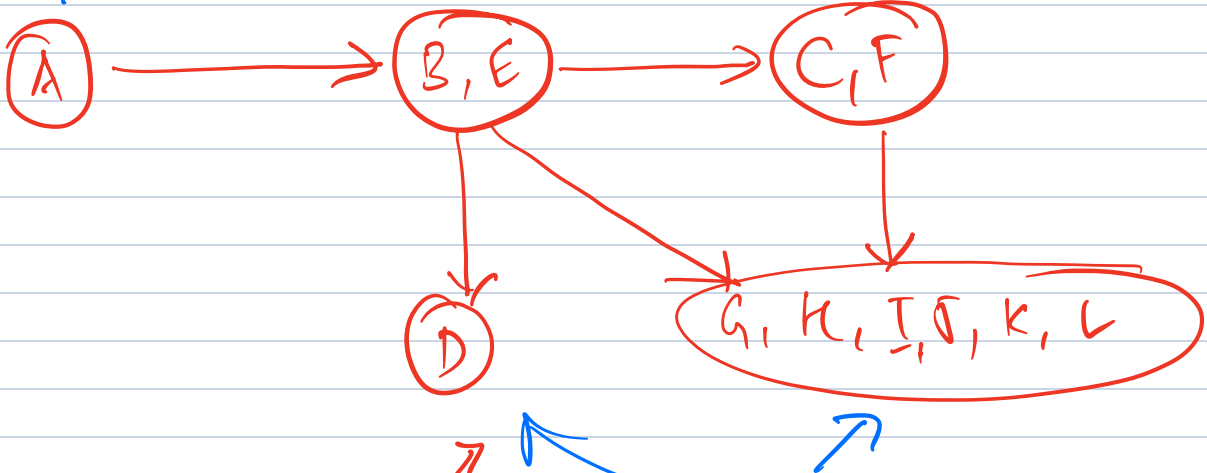
Using this, we can partition V into SCC's (strongly conn. components).

Property: Every directed graph is a DAG of its strongly connected components.



WHY?

Source (no incoming edges)



Meta Graph.

Sinks (no outgoing edges)

maximal

This meta graph consisting of SCC's as vertices, and with an edge from SCC : C_i to SCC : C_j if and only if there exists $u \in C_i$ and $v \in C_j$ such that $u, v \in E(G)$. This meta graph is necessarily acyclic.

Question: Can we recover the above structure efficiently?

This is possible using DFS.

in $O(|V|+|E|)$ time.

Q: What is a way to get one complete SCC?

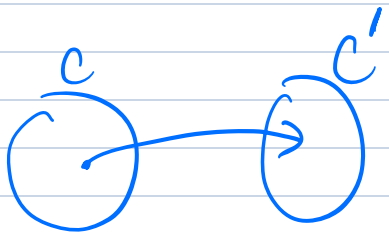
Answer: If we start explore from a vertex u in a sink SCC, then we will terminate once we explore the entire SCC which contains u.

But: How do we get such a vertex u in a sink SCC?

Property 1: In an entire DFS of a directed graph G, the vertex that receives the highest post(·) number must be in a source SCC.

The above is a consequence of the following:

Property 2: If C and C' are SCC's of a directed graph G, and if there is an edge from C to C', then the highest post number of C is greater than the highest post number of C'.



Proof: There are two cases:

① DFS visits C before C': In this case DFS will explore C' and complete C' before coming back and exiting from C. So the first node visited in C will have a higher post number than all of C'.

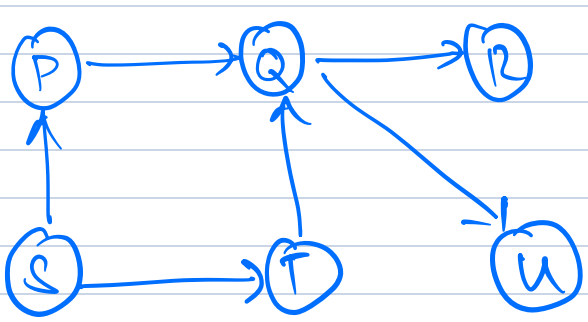
(2) DFS visits C' before C. In this case, DFS explores C' fully and exits before discovering C. So all vertices in C will have higher pre and post numbers as compared to C'.

How do we use this?

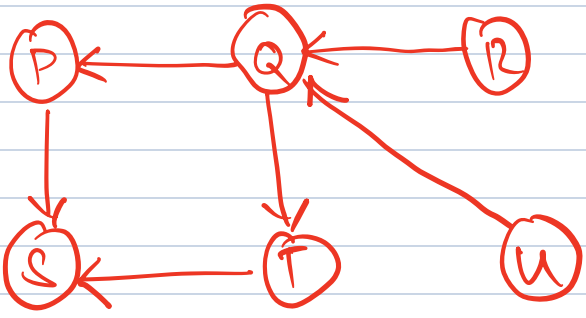
Property 1 helps us recover a vertex in a source SCC. But we need a vertex in a sink!

reverse graph

The solution is to run DFS on the reverse graph G^R, which is the same as G but with all the edges reversed.



G



G^R

S : Source

R, U are sinks

S : Sink

R, U : Source

Question: How can we construct G^R in $O(|V|+|E|)$ time?

Exercise!

Algorithm for SCC

1. Run DFS on G^R.

Can be done in parallel to step 1.

2. Order the vertices in G in the decreasing order of the post numbers in step 1.

3. Run DFS on G where we use the order of the vertices obtained in step 2.

The connected components algorithm applied on this ordering will yield the SCC's.

$2 \times DFS \Rightarrow O(|V| + |E|)$ time.

References: ① Introduction to Algorithms by Cormen, Leiserson, Rivest, Stein

② Algorithms by Dasgupta, Papadimitriou and Vazirani.