CS 341 – Algorithms

Lecture 7 – Directed Graphs

2 June 2021

Today's Plan

1. Directed Graphs, Reachability, BFS/DFS
2. Strongly Connected Graphs
3. Directed Acyclic Graphs
4. Strongly Connected Components

HW 2 is posted

ready to solve Q1, Q2, Q4

Directed Graphs

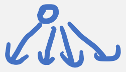
If uv is a directed edge, then u is the **tail** of the edge and v is the **head** of the edge.

The **in-degree** of a vertex v, denoted by $\mathrm{indeg}(v)$, is the number of edges with v being the head.

The **out-degree** of a vertex v, denoted by $\mathrm{outdeg}(v)$, is the number of edges with v being the tail.

indeg = 4
outdeg = 2

sink if outdeg = 0

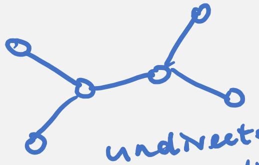


source if indeg = 0

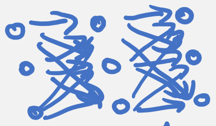
Directed graphs are useful in modeling asymmetric relations (e.g. web links, one-way streets).

Google

A directed graph G is a **directed acyclic graph** if there is no directed cycles in G.

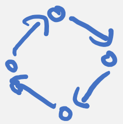


undirected acyclic



could have many edges

directed acyclic graph



X

Connectivity in Directed Graphs

Given two vertices s, t, we say t is **reachable** from s if there is a directed path from s to t.

A directed graph $G = (V, E)$ is **strongly connected** if for every pair of vertices $u, v \in V$,

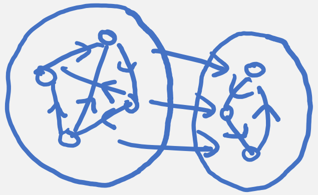
u is reachable from v and v is reachable from u.

A subset $S \subseteq V$ is called strongly connected if for every pair of vertices $u, v \in S$,

u is reachable from v and v is reachable from u.

A subset $S \subseteq V$ is called a **strongly connected component** if S is a maximally strongly connected subset,

i.e. S is strongly connected but $S + v$ is not strongly connected for any $v \in V - S$.



Questions

We are interested in designing efficient algorithms for the following basic questions:

1. Determine if an input graph is a strongly connected graph.

2. Determine if an input graph is a directed acyclic graph.

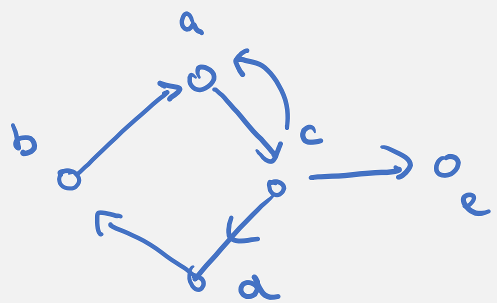
3. Find all strongly connected components of a directed graph.

It will turn out that all these problems can be solved in $O(m + n)$ time,

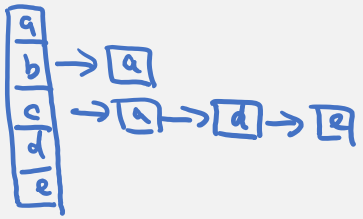
but they are not as easy to solve as in the undirected analogs especially Q3.

Data Structures

Both adjacency matrix and adjacency list can be defined for directed graphs.



$$\begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & & 0 & 1 & & \\ b & 1 & & & & \\ c & 1 & & & & 1 \\ d & & & & & \\ e & & & 0 & & \end{array}$$



We will only use adjacency list in this lecture, as only this allows us to design $O(m+n)$ time algorithms.

DFS

Both DFS and BFS are defined as in undirected graphs, except that we only explore out-neighbors.

Input : A directed graph $G = (V, E)$ and a vertex s.

Output : All vertices reachable from s.

[Main program] visited[v] = false $\forall v \in V$. time = 1. visited[s] = true. explore(s).

explore (u) // recursive function explore.

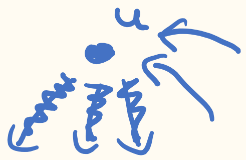
start[u] = time. time $\leftarrow$ time + 1.

for each $\boxed{\text{out}}$-neighbor v of u

if visited[v] = false

visited[v] = true. explore(v).

finish[u] = time. time $\leftarrow$ time + 1. put u in Q



BFS and DFS

We can also define BFS for directed graphs analogously, by only exploring out-neighbors.

When a vertex v is first visited, we remember its parent as the vertex u when v is first visited from.

The edges $(v, parent[v])$ from a tree, and both BFS tree and DFS tree are defined this way.

Exercises

1. Time Complexity is $O(m+n)$.

2. A vertex t is reachable from s if and only if $visited[t] = true$.

3. The set of vertices reachable from s forms a "directed cut".

4. BFS can be used to compute a shortest path from s to all other vertices in $O(m+n)$ time.

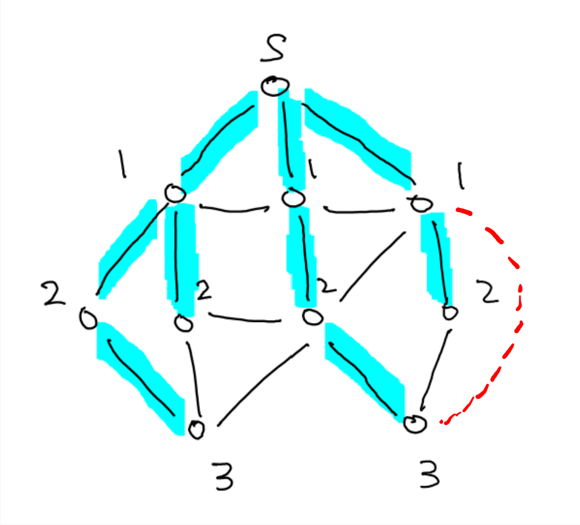
BFS Tree

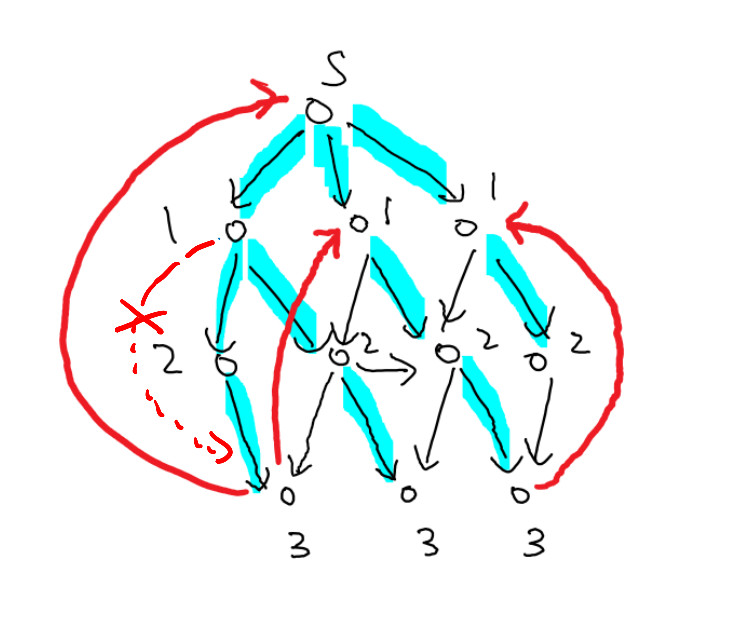
By setting $dist[v] = dist[u] + 1$, we can compute all shortest path distances from s.

In undirected graphs, for all non-tree edges uv, $dist[u] - 1 \leq dist[v] \leq dist[u] + 1$.

In directed graphs, there could be non-tree edges uv with large difference between $dist[u]$ and $dist[v]$,

but they must be "backward edges".

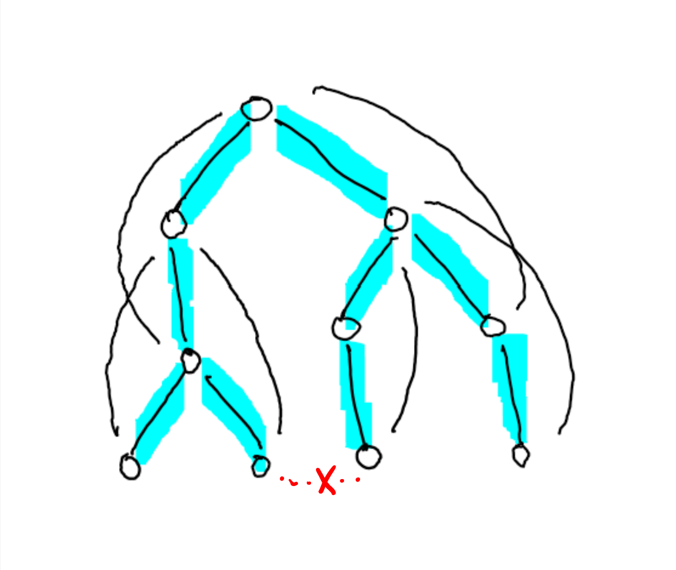


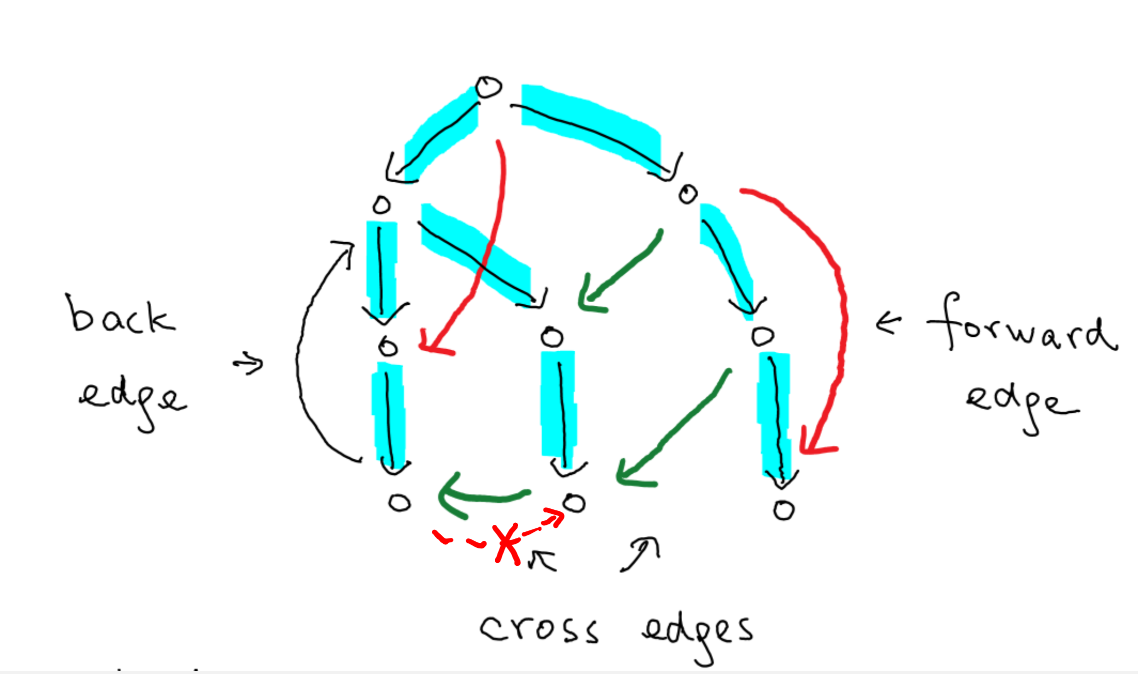


DFS Tree

In undirected graphs, all non-tree edges are back edges as we proved in L06.

In directed graphs, some non-tree edges could be “cross edges” or “forward edges”.





Structures in directed graphs are a bit more complicated.

Today's Plan

1. Directed Graphs, Reachability, BFS/DFS
2. Strongly Connected Graphs
3. Directed Acyclic Graphs
4. Strongly Connected Components

Homework 2 is posted

Q1, Q2, Q4 ready

Strongly Connected Graphs

Input: A directed graph $G = (V, E)$.

Output: Yes if G is strongly connected; no otherwise.

Strongly connected : $\forall\, u, v \in V$, $u \to v$, $u \leftarrow v$

$\Theta(n^2)$ pairs to check.

naive: ① for each pair, check $\exists$ a path u to v , v to u ,

total time : $O(n^2 \cdot (n+m))$

② for each vertex v, whether all vertices can be reached from v

total time = $O(n \cdot (n+m))$

What did we do for undirected graphs? choose <u>one s</u> . check whether it can reach all other vertices

$O(n)$ pairs

What is a "succinct" condition to check for directed graphs?

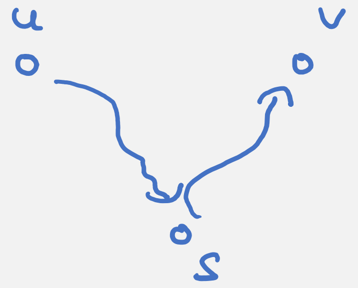
Observation

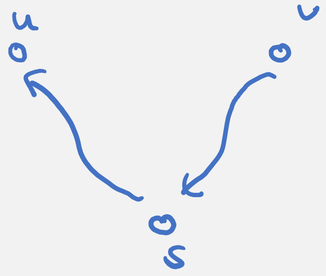
Claim. G is strongly connected if and only if (every vertex $v \in V$ is reachable from s) $\leftarrow$ one BFS/DFS
and (s is reachable from every vertex $v \in V$), where s is an arbitrary vertex.

proof

$\Rightarrow$) trivial. by definition.

$\Leftarrow$)





$\exists$ walk

$\Rightarrow \exists$ path

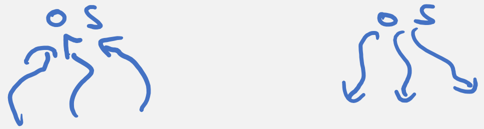
$\square$

only need to check $O(n)$ pairs

How do we check whether s is reachable from every vertex $v \in V$?

Trick

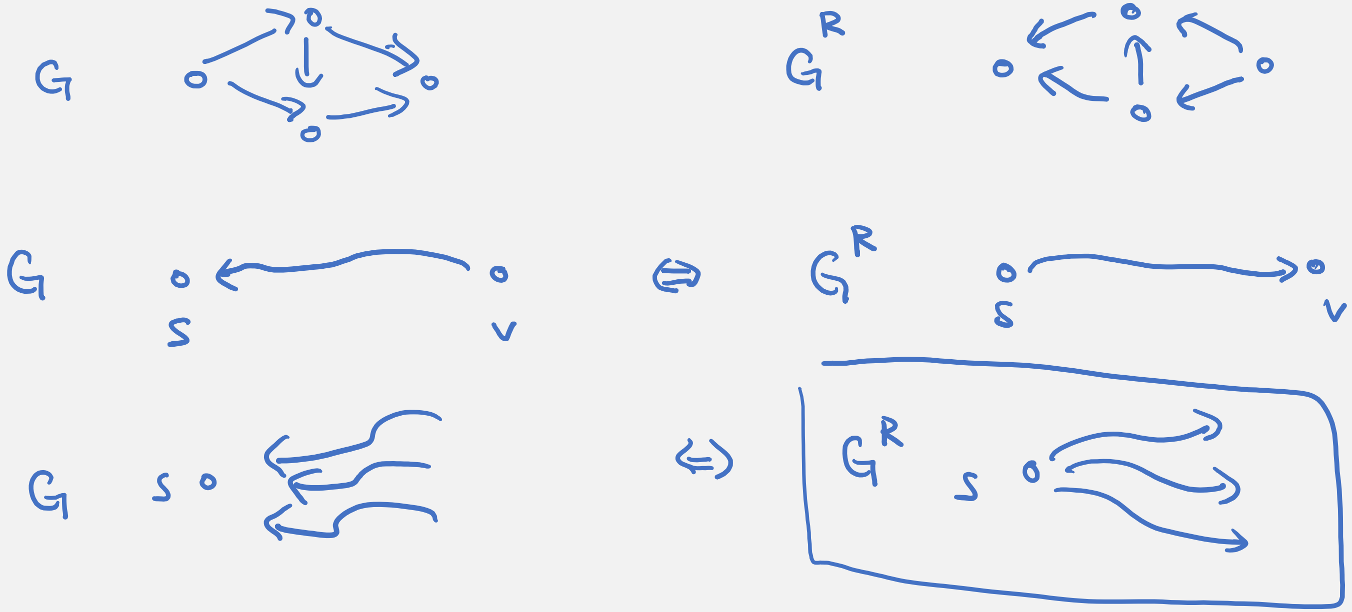
Idea: Reverse the graph!



Claim: Given $G = (V, E)$, we reverse the direction of all the edges to obtain $G^R = (V, E^{\leftarrow})$.

Then, there is a path from v to s in G if and only if there is a path from s to v in G^R.

So, s is reachable from every $v \in V$ in G if and only if every $v \in V$ is reachable from s in G^R.



solve this using one BFS/DFS

Algorithm

1. Check whether all vertices in G are reachable from s by one BFS/DFS. $O(m+n)$

$O(m+n)$

2. Reverse the direction of all the edges in G to obtain G^R.

in G - s is reachable from all $v \in V$

$\Updownarrow$

3. Check whether all vertices in G^R are reachable from s by one BFS/DFS. $O(m+n)$

4. If both yes, return "strongly connected"; otherwise return "not strongly connected".

correctness : based on 4 . based on the claim 2 slides ago.

complexity : $O(m+n)$

Today’s Plan

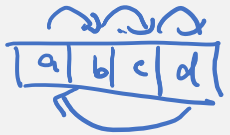
1. Directed Graphs, Reachability, BFS/DFS
2. Strongly Connected Graphs
3. Directed Acyclic Graphs
4. Strongly Connected Components

Directed Acyclic Graphs

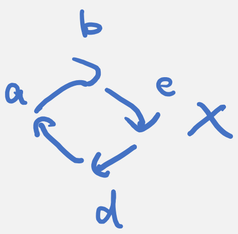
Directed acyclic graphs are useful in modeling dependency relations (e.g. course prerequisites).

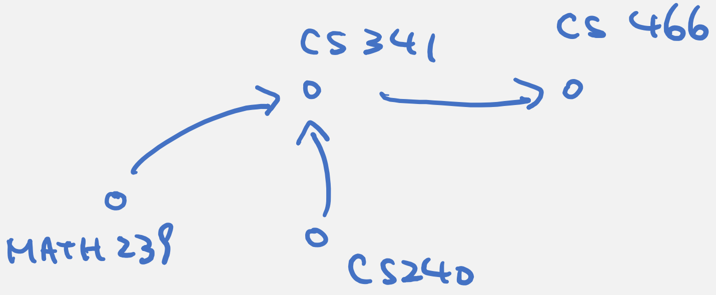
In such situations, it is useful to find an ordering of the vertices so that all the edges go forward.

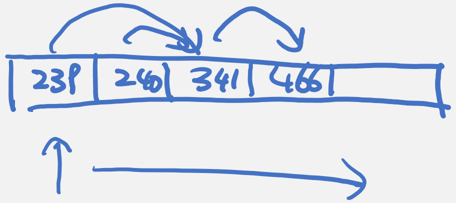
This is called a **topological ordering** of the vertices.



directed acyclic graphs : no directed cycles







Topological Ordering

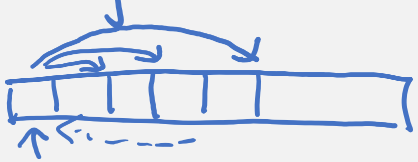
Proposition. A directed graph is acyclic if and only if there is a topological ordering of the vertices.

proof $(\Leftarrow)$ If $\exists$ a topological ordering.

$\Rightarrow$ no directed cycles

$\Leftarrow$ If $\exists$ a directed cycle, then $\nexists$ a topological ordering.

$\Rightarrow)$ goal: DAG $\Rightarrow$ $\exists$ a topological ordering.



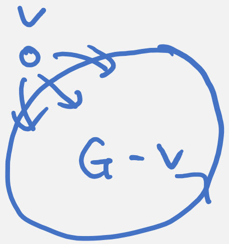
must be a vertex with indeg $= 0$

need to show:

$\boxed{\forall \text{ DAG } \exists \text{ a vertex of indegree } 0}$

this is also sufficient

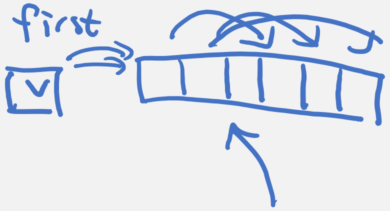
by induction G



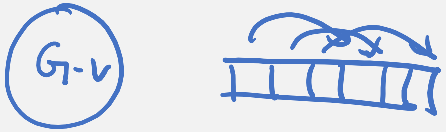
G − v is also acyclic

by I.H. $\exists$ a topological ordering of G − v

first



topological ordering of G



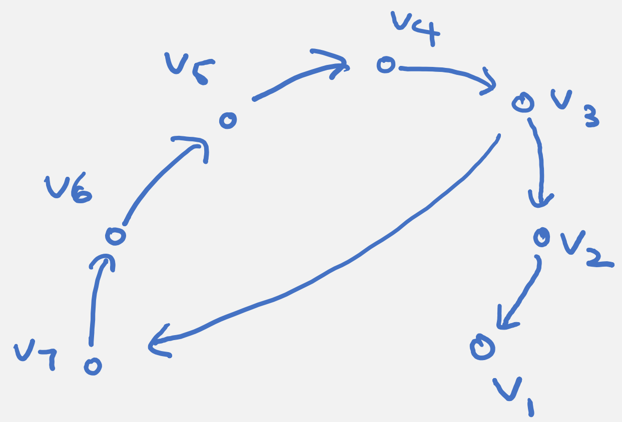
topological ordering of G − v

Topological Ordering

Proposition. A directed graph is acyclic if and only if there is a topological ordering of the vertices.

prove : $\forall$ DAG $\exists$ a vertex of indegree 0.

contrapositive : $\forall$ vertex indegree ≥ 1 $\Rightarrow$ directed cycle



$\exists$ a vertex u
with uv_4

1. u is visited
$\Rightarrow$ $\exists$ a directed cycle

2. u is not visited
but this argument cannot repeat forever, as the graph is finite

$\square$

First Approach

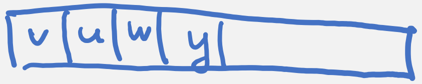
Just follow the above proof.

repeatedly find a vertex of indegree zero

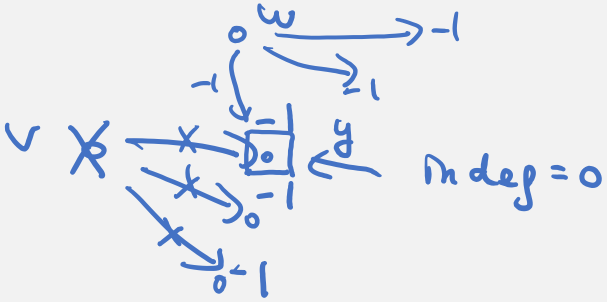
put it in the beginning of the topological ordering

can be implemented in $O(m+n)$ time.

- beginning, read the graph, put every vertex of indegree 0 into a queue



- iteration.



Second Approach

This is probably less intuitive, but the idea will be useful in the next problem as well.

Idea: Do a DFS on the whole graph.

any ordering of the vertices $1 \le 2 \le \dots \le n$

initially. visited[i] = false $\forall i$

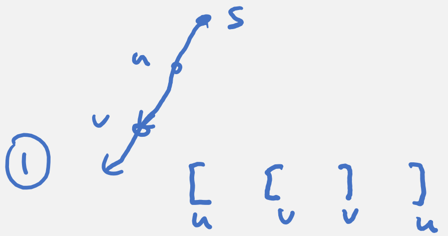
$O(m+n)$ time

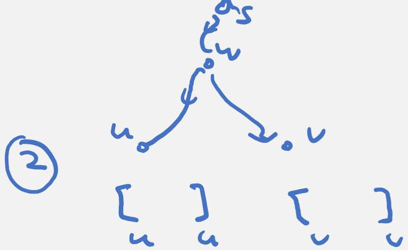
$$\left\{\begin{array}{l} \text{for } 1 \le i \le n \\ \quad \text{if visited}[i] = \text{false} \\ \qquad \text{DFS}(i) \end{array}\right.$$

find all connected components in undirected graphs in $O(m+n)$ time.

Finishing Times

Lob





We will use the parenthesis property of starting and finishing times, which still holds for directed graphs.

finish[u] > finish[v]

Lemma. If G is directed acyclic, then for any edge uv, $finish[v] < finish[u]$ for any DFS.

<u>proof</u> case 1: start[v] < start[u]

since G DAG and uv ∈ E,

⇒ u is not reachable from v

⇒ u cannot be a descendant of v in DFS tree

⇒ [start[u], finish[u]] and [start[v], finish[v]] are disjoint by parenthesis property.

⇒ $\underbrace{start[v] \quad finish[v]} \quad \underbrace{start[u] \quad finish[u]}$

⇒ finish[v] < finish[u].

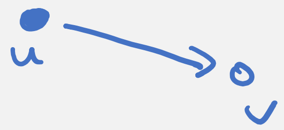
Finishing Times

We will use the parenthesis property of starting and finishing times, which still holds for directed graphs.

Lemma. If G is directed acyclic, then for any edge uv, $finish[v] < finish[u]$ for any DFS.

proof

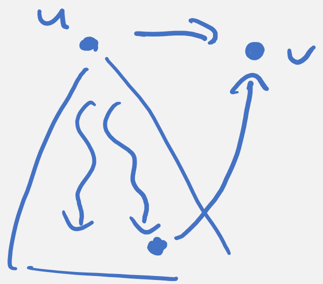
case 2: start[u] < start[v]



this proof is similar to L06

" all non-tree edges are back edges ?"

1



v descendant of u

2

v child of u

either case:

v descendant of u

by parenthesis

[[ ]]

u v v u

$\Rightarrow$ finish[u] > finish[v]. □

Algorithm

1. Run DFS on the whole graph. $O(n+m)$

2. Output the ordering with decreasing finishing time. → $O(n+m)$, no need to sort.

3. Check if it is a topological ordering. → If yes, return "acyclic". If not, return "not acyclic".

Correctness: If not acyclic, then ∄ topological ordering ⇒ algorithm "not acyclic"

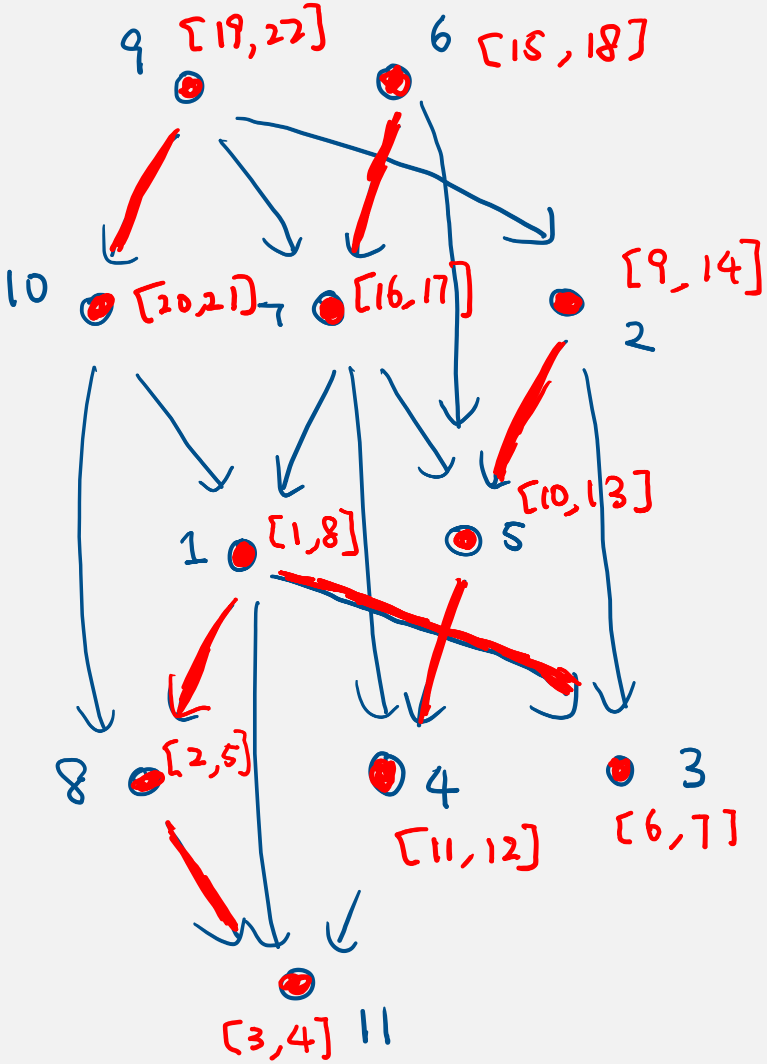
If acyclic, then lemma says $u \to v$: finish[u] > finish[v] for all $uv \in E$

decreasing finishing time ordering ⇒ all the edges go forward □

Time: $O(n+m)$

Exercise: not acyclic, output a directed cycle.

Example



9, 10, 6, 7, 2, 5, 4, 1, 3, 8, 11

finish[u] > finish[v] if G DAG

Today's Plan

1. Directed Graphs, Reachability, BFS/DFS
2. Strongly Connected Graphs
3. Directed Acyclic Graphs
4. Strongly Connected Components

1. Homework 2 is posed, due on June 14
2. Discuss take-home midterm on June 28

Strongly Connected Components

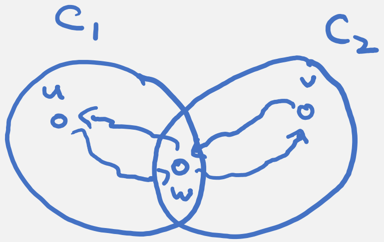
Now we consider the more difficult problem of finding all connected components of a directed graph.

We will combine and extend the previous ideas to obtain an $O(m+n)$-time algorithm.

First, we understand better the structure of a general directed graph.

Claim. Two strongly connected components must be vertex disjoint. C_1 and C_2

proof



$\Rightarrow$ $\forall u, v \in C_1 \cup C_2$

$u \longrightarrow v$ $\quad$ $v \longrightarrow u$

$C_1 \cup C_2$ strongly connected

but it contradicts that

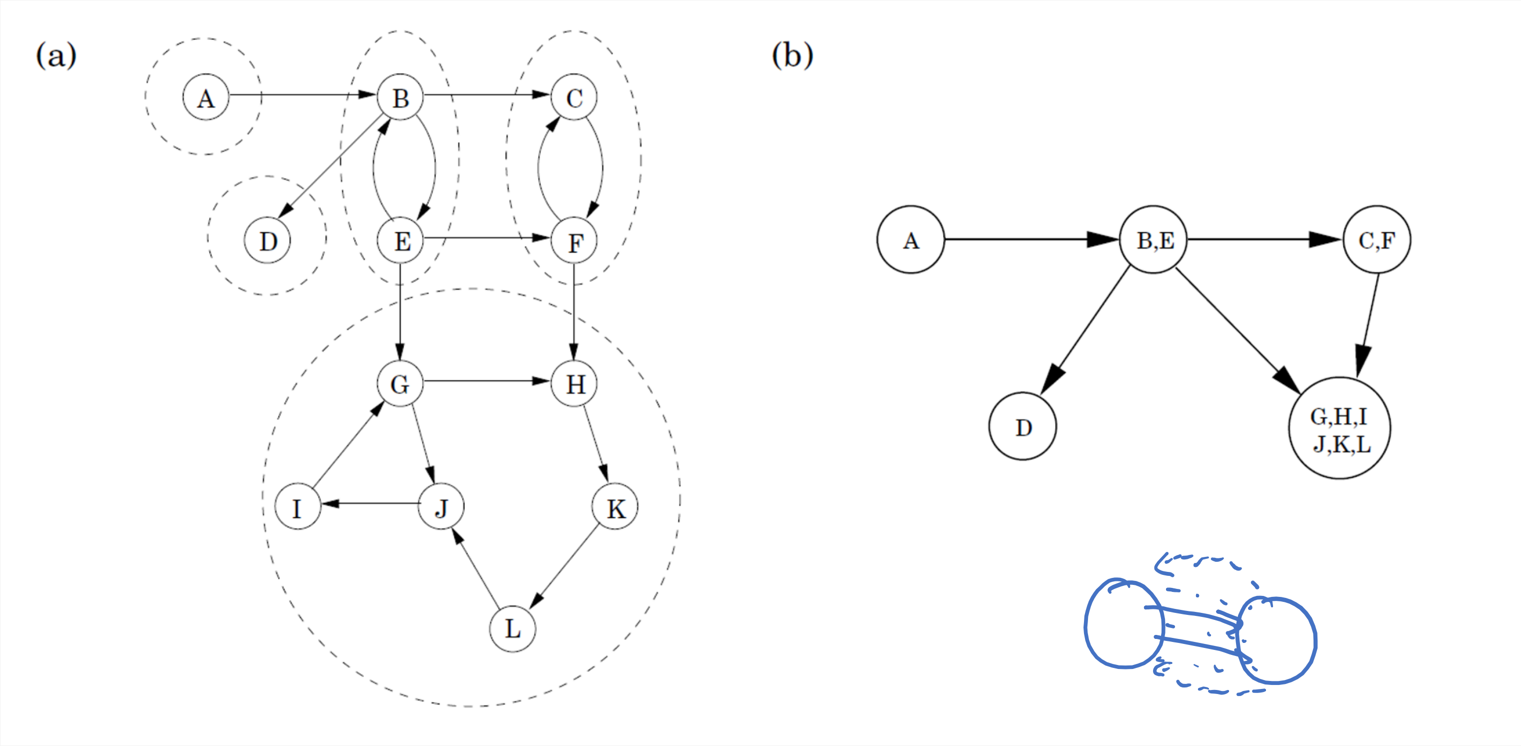
C_1, C_2 are maximal strongly connected subsets.

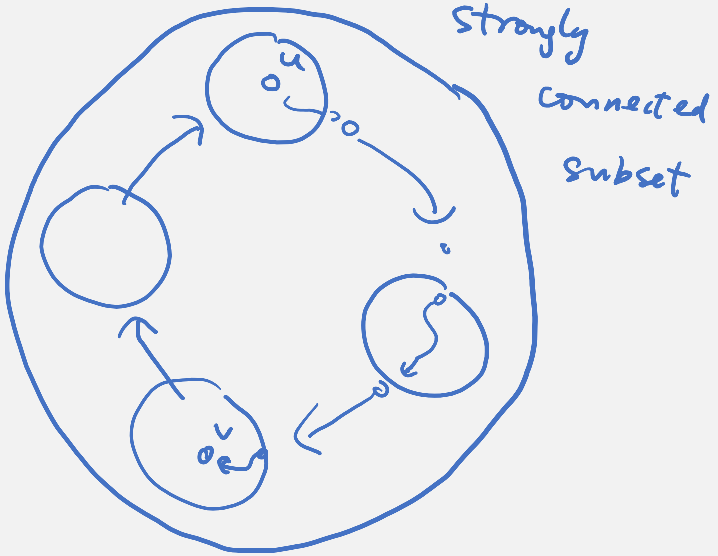
$\square$

Structure of a Directed Graph

Observation. When every strongly connected component is "contracted" into a single vertex,

then the resulting directed graph is acyclic.





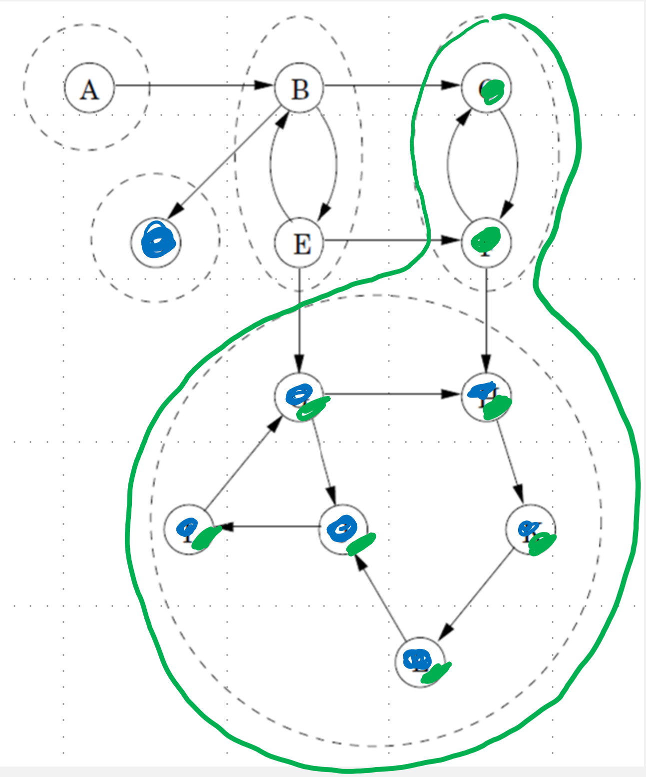
So, a general directed graph is a directed acyclic graph on its strongly connected components.

Idea 1

How do we identify the vertices in a strongly connected component easily?

One natural attempt is do a BFS/DFS on a vertex v, and hope that it identifies the SCC containing v.

But this doesn't always work.

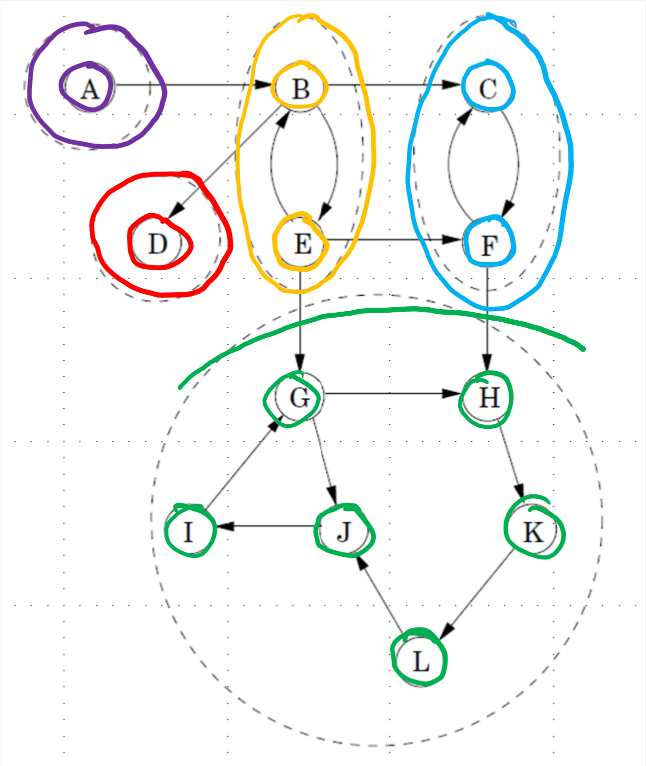


<u>Observation</u>. Suppose we start a BFS/DFS on a "sink component" (a component with no outgoing edges), then we can identify the vertices in that sink component.

Idea 1: Cut Out Sink Components

This suggests the following strategy.

1. Find a vertex v in a sink component C.
2. Do a DFS / BFS to identify C.
3. Remove C from the graph and repeat.

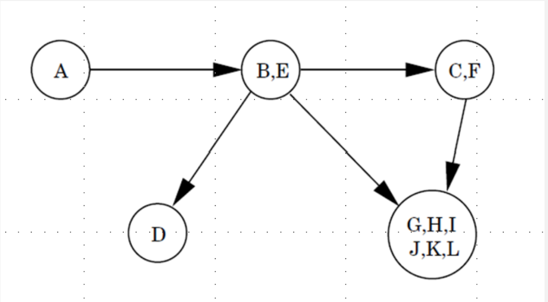


How do we find a vertex in a sink component efficiently?

It doesn't look easy, especially we have to find such a vertex many times (one for each component), and yet the total time complexity should be $O(n+m)$.

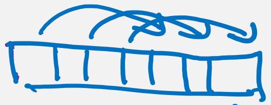
Idea 2: Topological Sort

Recall that a directed graph is a directed acyclic graph on its strongly connected component.



For directed acyclic graphs, if we do a DFS on the whole graph,

then the vertex with the smallest finishing time is a sink.



sink

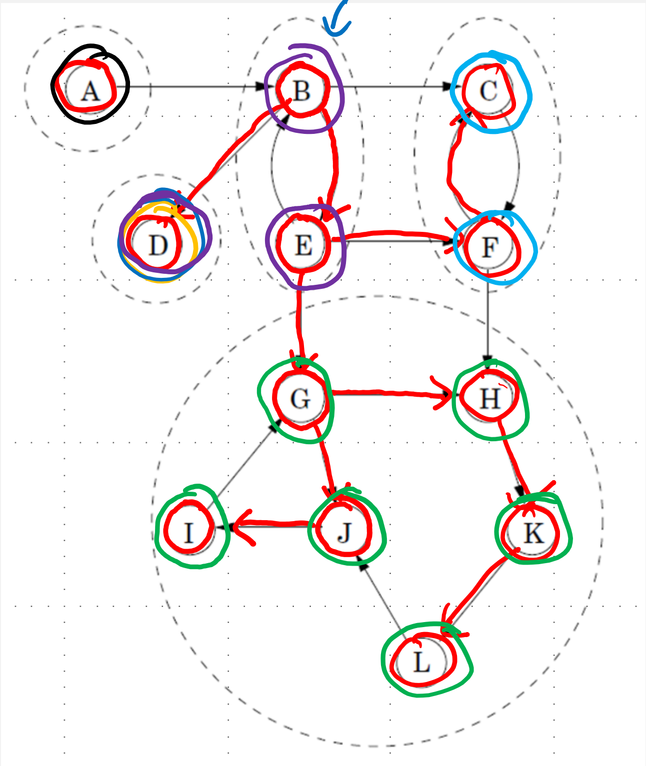
Let's try this in a general directed graph.

→ decreasing finishing time

A, B, D, E, F, C, G, H, K, L, J, I

← DFS on whole graph

start DFS here

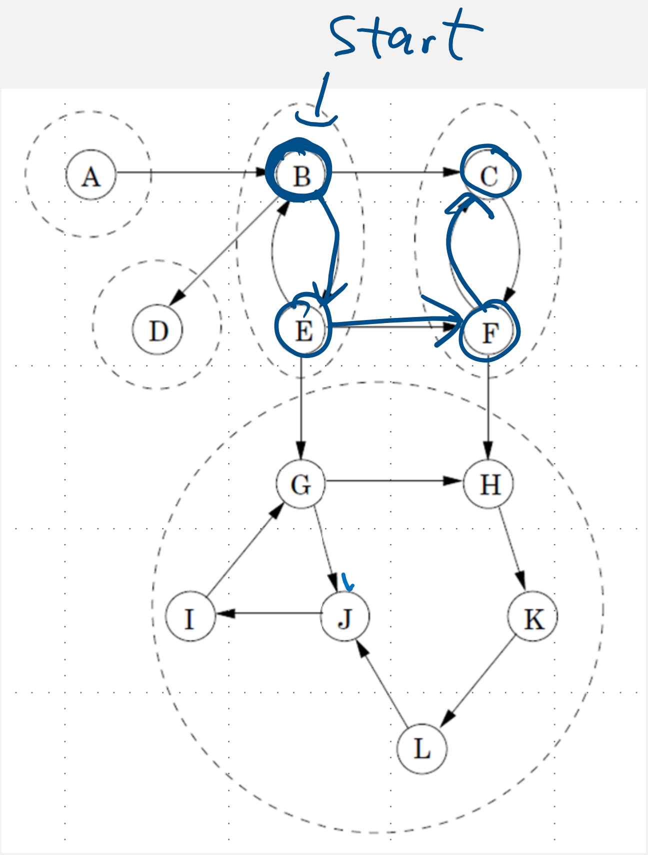


A Counterexample

This is a nice strategy, but unfortunately it doesn't always work.

C has the smallest finishing time.

but it is not in a sink component.



If we look at the proof about finishing times of vertices in a DAG,

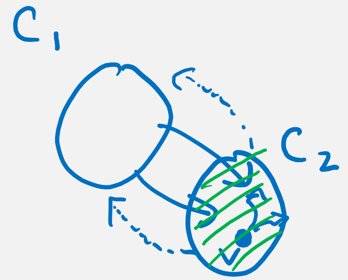
then we realize that the proof still works in one way.

Idea 3

Lemma. If C_1 and C_2 are strongly connected components and there are directed edges from C_1 to C_2, then the largest finishing time in C_1 is larger than the largest finishing time in C_2.

proof

same as the lemma about topological ordering



C_2 finished before C_1 started

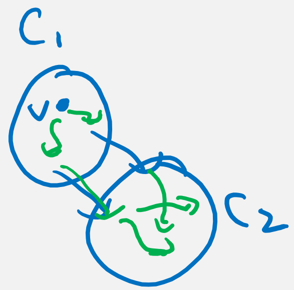
① first vertex to be visited in $C_1 \cup C_2$ is in C_2

then every vertex in C_2 is reachable from a

but no vertex in C_1 is reachable from a.

$\Rightarrow$ every vertex in C_2 will be finished before any vertex in C_1 started

② first vertex to be visited in $C_1 \cup C_2$ is in C_1



$C_1 \cup C_2$ can be reached from v

v will have the largest finishing time.

$\square$

Idea 3: Source Components

So, if we do a DFS on the whole graph, and then sort the vertices in decreasing finishing time.

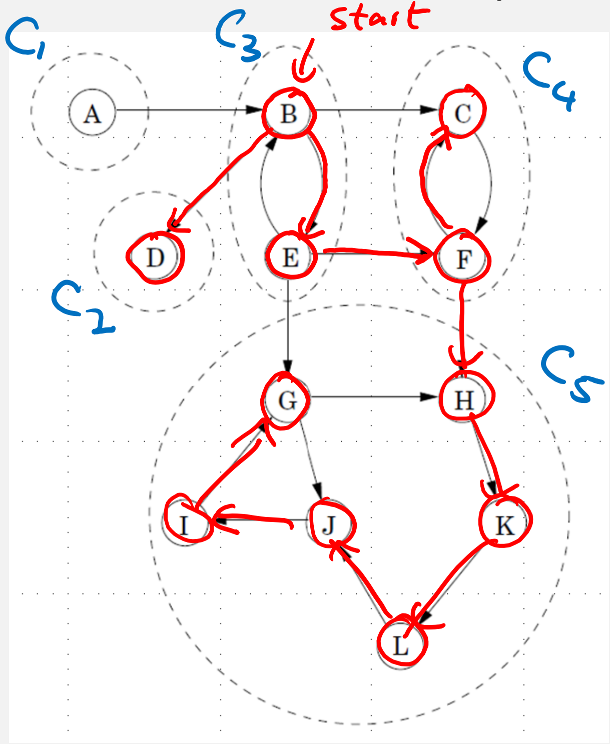
And then do a DFS again using this ordering.

$C_1 \to C_3 \to C_4$, $C_3 \to C_2$, $C_3 \to C_5$, $C_4 \to C_5$

Then we will always first visit an "ancestor component" before we first visit a "descendant component"

DFS in this ordering $\longrightarrow$

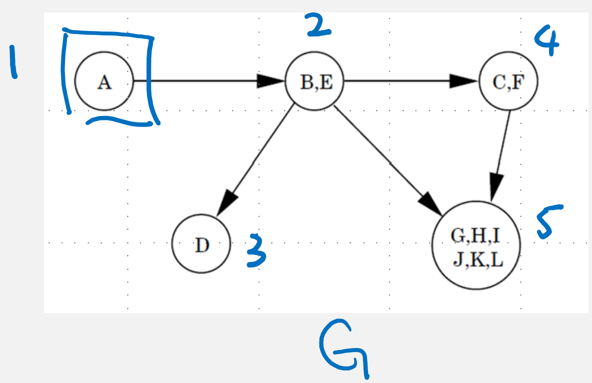
$\underbrace{A}_{C_1}, \underbrace{B}_{C_3}, \underbrace{D}_{C_2}, E, \underbrace{F}_{C_4}, \underbrace{H}_{C_5}, K, L, J, I, G, C$

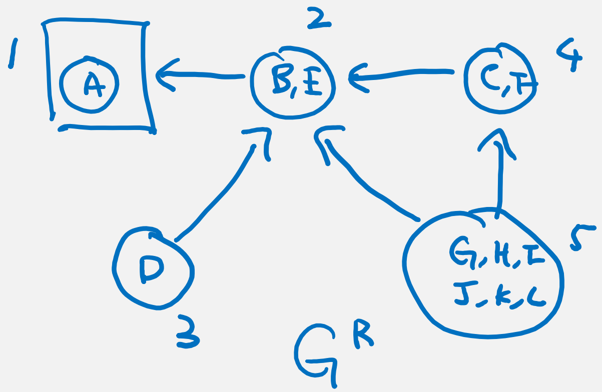


But this is not what we want, we want to first visit a descendant component before we first visit an ancestor component, so that we can cut out the sink components one at a time.

Idea 4: Reverse the Graph

1. The strongly connected components in G and G^R are the same.





2. The source components in G become the sink components in G^R and vice versa.

So, the ordering in G visiting ancestor components before descendant components

is an ordering in G^R visiting descendant components before ancestor components.

This is exactly what we want (although in G^R, it doesn't matter as SCCs in G and G^R are the same).

Algorithm

1. Run DFS on the whole graph G using an arbitrary ordering of vertices.

2. Order the vertices in decreasing order of finishing times obtained in step 1.

$O(m+n)$

3. Reverse the graph G to obtain the graph G^R

$O(m+n)$

4. Follow the ordering in step 2 to explore the graph G^R to cut out the components one at a time.

Algorithm

(i) Let i be the vertex of i-th largest finishing time in step 2.

(ii) Let $c = 1$. // It is a variable counting the number of strong connected components.

(iii) For $1 \le i \le n$ do ← DFS using the ordering of decreasing finishing time

$O(m+n)$

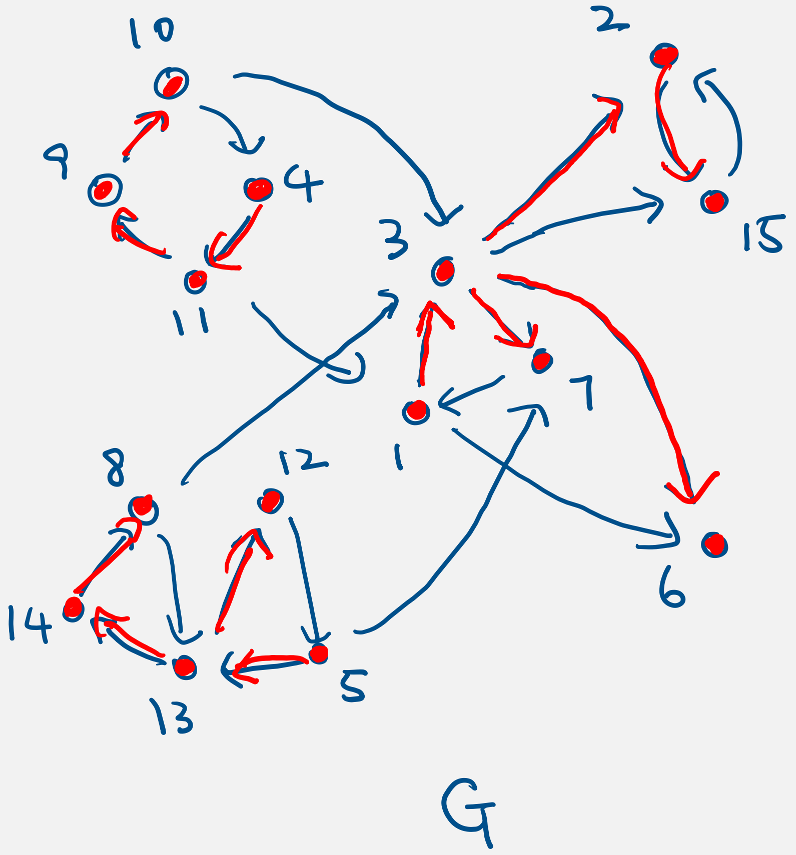
If visited[i] = false

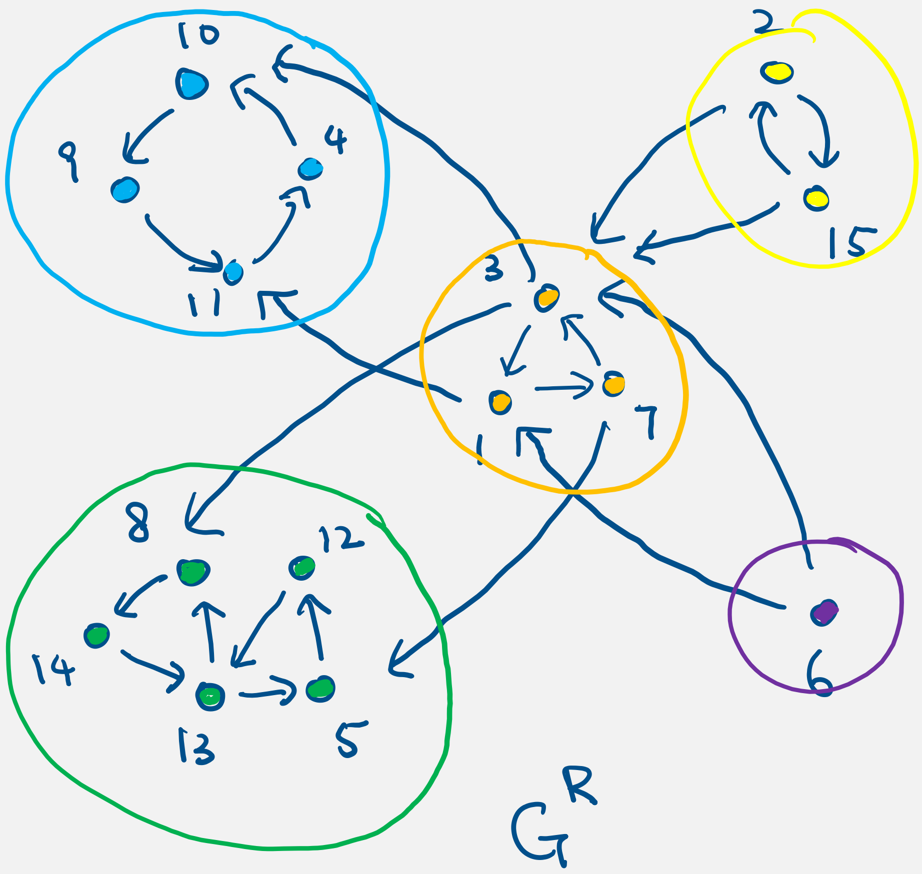
DFS (G^R, i)

Mark all the vertices reachable from i in G^R in this iteration to be in component c.

$c \leftarrow c+1$

Example





5, 13, 14, 8, 12, 4, 11, 9, 10, 1, 3, 6, 2, 15, 7

Take-Home Midterm

Content from L01.pdf to L07.pdf.

Will be posted on piazza on June 28 9am EST, and the deadline is on June 29 9am EST.

No late submission.

Will post more information on piazza.

Will post midterm questions from previous years.

Midterm questions will be easier than those in homework.

Allowed to use lecture notes, tutorials notes, and information on piazza.

Not allowed to use any other references (including the reference books).

Definitely no communications with others and no looking up of other resources (e.g. internet).