CS2040

Tutorial 8: Graphs and Traversal

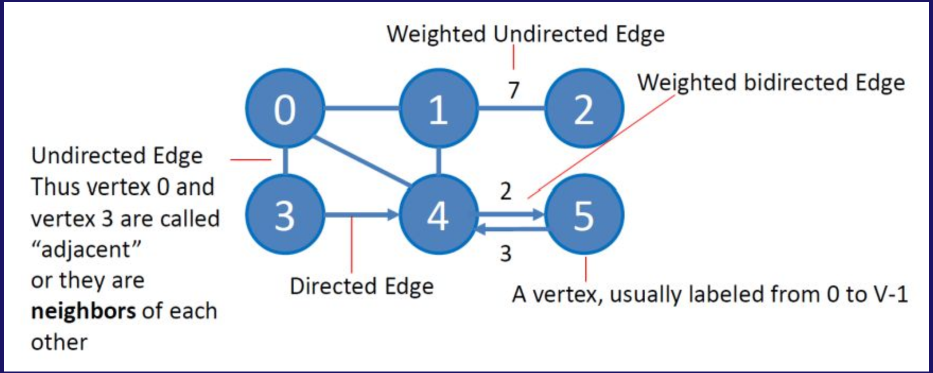
Nicholas **Russell** Saerang (russellsaerang@u.nus.edu)

:D

GRAPH

Graph

A set of vertices and edges that connect some pair of vertices.
For this module, we ignore “multi graph” where there can be more than one edge between a pair of vertices.


Weighted Undirected Edge
Weighted bidirected Edge
Undirected Edge
Thus vertex 0 and vertex 3 are called “adjacent” or they are **neighbors** of each other
Directed Edge
A vertex, usually labeled from 0 to V-1
0
1
2
3
4
5
7
2
3

Please refer to the lecture slides for the graph terminologies!

Graph Data Structures

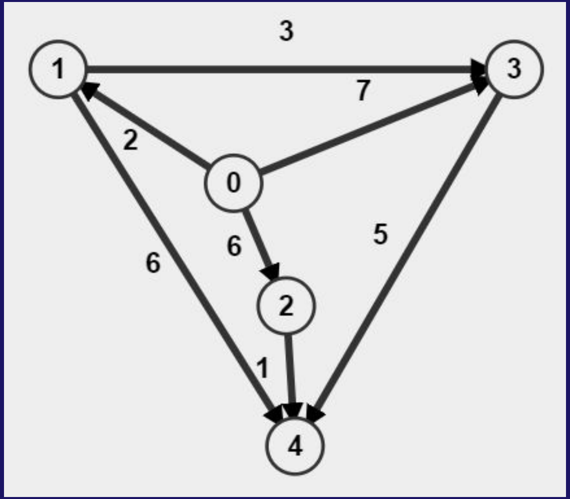
Adjacency Matrix

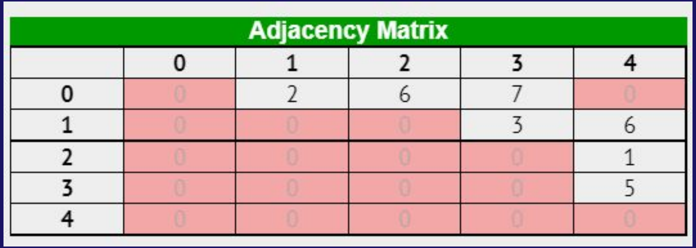
AdjMatrix[i][j] = 1 (or the weight of the edge) if there is an edge between i and j, 0 otherwise.

Space complexity: O(V^2)

Properties:

- Query to know whether a and b has a direct edge is O(1).
- Good for dense graph (Floyd Warshall's APSP, see Week 12)
- O(V) to enumerate neighbors of a vertex





Adjacency Matrix

	0	1	2	3	4
0	0	2	6	7	0
1	0	0	0	3	6
2	0	0	0	0	1
3	0	0	0	0	5
4	0	0	0	0	0

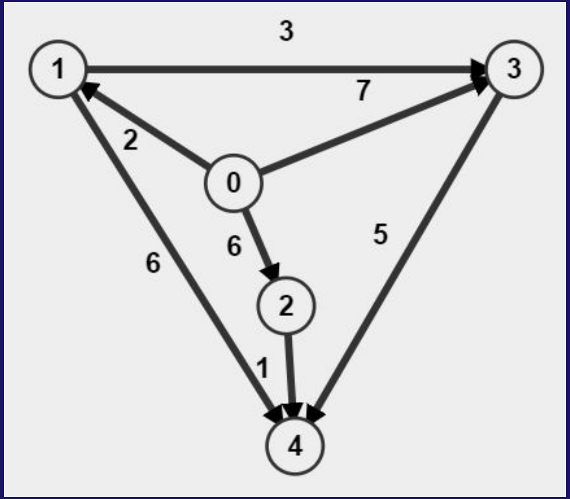
Adjacency List

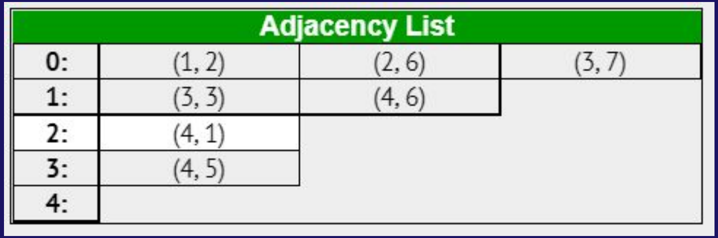
AdjList[i] stores list of i's neighbors. For weighted graph, stores pair(neighbor, weight).

Space complexity: O(V + E)

Properties:

- Query to iterate all k neighbours is O(k).
- Good for sparse graph (Dijkstra's SSSP, DFS/BFS, Prim's MST, see Week 11)
- O(k) to check the existence of edge i-j





Adjacency List			
0:	(1, 2)	(2, 6)	(3, 7)
1:	(3, 3)	(4, 6)	
2:	(4, 1)		
3:	(4, 5)		
4:			

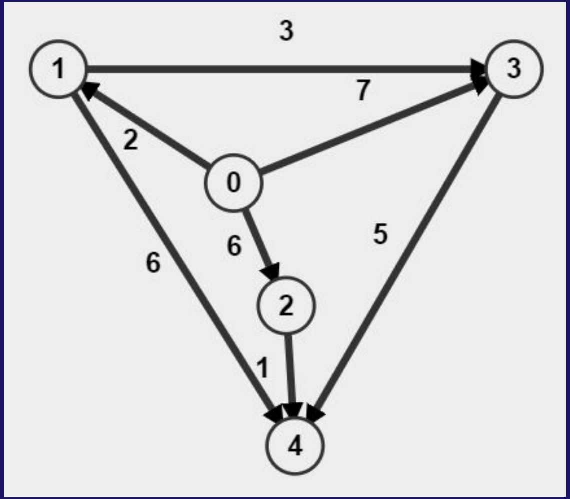
Edge List

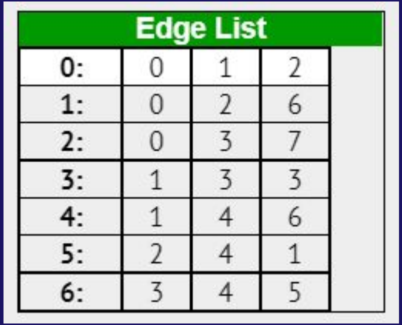
For each edge i, EdgeList[i] stores an (integer) triple{u, v, w(u, v)}.

Space complexity: O(E)

Properties:

- Usually used for Kruskal's (MST) as we need to sort edges by weight





Edge List			
0:	0	1	2
1:	0	2	6
2:	0	3	7
3:	1	3	3
4:	1	4	6
5:	2	4	1
6:	3	4	5

Graph Traversal

BFS (use adjacency list!)

- Use queue to maintain order of traversal
- Array visited of size V to differentiate visited vs unvisited vertices
- Array p of size V to record the parent in order to generate a path
- Time complexity: O(V + E)
  - Every vertex is in the queue once, when queue is empty (all vertices are dequeued), all E edges are examined

DFS (use adjacency list!)

- Use stack / recursion (implicit stack)
- Array visited of size V to differentiate visited vs unvisited vertices
- Array p of size V to record the parent in order to generate a path
- Time complexity: O(V + E) (similar analysis to BFS)

Graph Traversal

Some follow-up questions:

- What if I'm using BFS/DFS on an adjacency matrix?
- What if I'm using BFS/DFS on an edge list?

Key idea: the two algorithms try to enumerate all neighbours for each vertices, so think of the time complexity to enumerate all neighbours for one vertex, then sum for all vertices.

Adjacency matrix:
O(V) to enumerate on one vertex, so $O(V + V^2) = O(V^2)$ overall

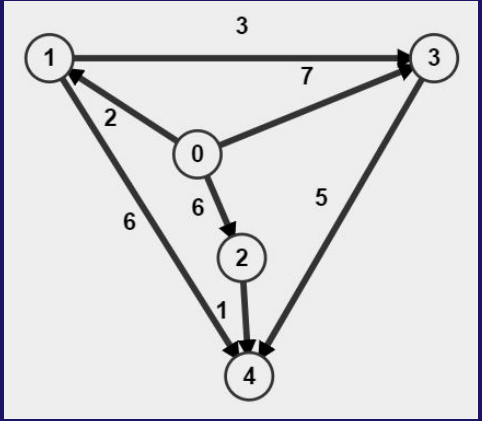
Edge list:
O(E) to enumerate on one vertex, so O(V + VE) = O(VE) overall

Adjacency list:
O(deg(v)) to enumerate on a vertex v, so O(V + sum(deg(v))) = O(V + E) overall

Topological Sorting

TLDR:
Given a Directed Acyclic Graph (DAG), find an ordering of vertices such that for every directed edge from u to v, u comes before v in the ordering.

This ordering is called topological ordering.



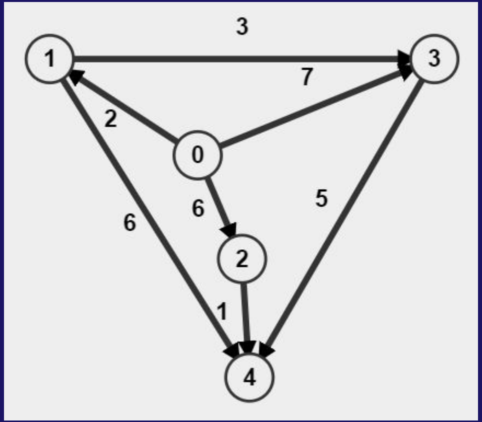
Topological Sorting

How it works:

- Kahn's Algorithm

  Keep track of the indegrees of the unprocessed vertices in an array and propagate information in a queue.

  E.g. the graph below has an indegree array of [0, 1, 1, 2, 3].
  Similar to BFS, we process vertices of indegree 0 and then for every edge decrease the indegree accordingly.



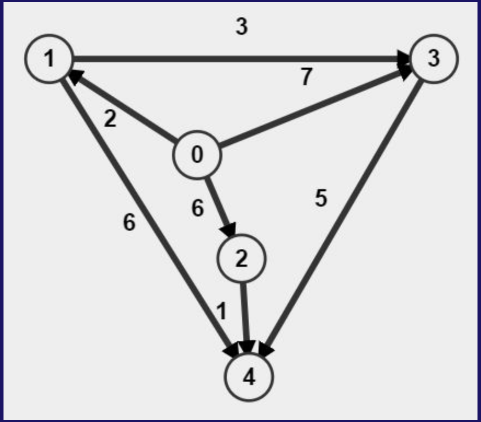
Topological Sorting

How it works:

- Kahn's Algorithm

  E.g. the graph below has an indegree array of [0, 1, 1, 2, 3].
  Since only vertex 0 has indegree 0, our queue is now [0].

```
pop 0 → decrease indegree of 1, 2, 3
  indegree = [0, 0, 0, 1, 3]
    since indegree of 1 and 2 are both 0, enqueue both
    queue is now [1, 2]
pop 1 → decrease indegree of 3 and 4
  indegree = [0, 0, 0, 0, 2]
    since indegree of 3 is 0, enqueue 3
    queue is now [2, 3]
pop 2 → decrease indegree of 4
  indegree = [0, 0, 0, 0, 1]
    queue is now [3]
pop 3 → decrease indegree of 4
  indegree = [0, 0, 0, 0, 0]
    since indegree of 4 is 0, enqueue 4
    queue is now [4]
pop 4 → nothing happens because no more neighbors
```



Red text tells you the topological ordering. If there are less than V vertices outputted, the graph has no valid topological ordering!

Topological Sorting

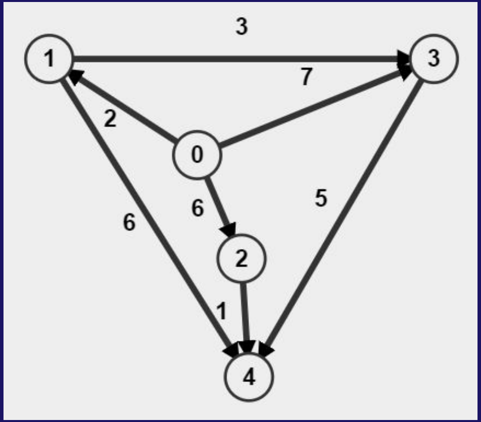
How it works:

- DFS Toposort Algorithm

  Treat as DFS but output the reverse of the post-processing order.
  Keep running DFS from any unvisited vertex until everything is visited.

  E.g. 0 is unvisited so we can start there.

```
visit(0)
  visit(1)
    visit(3)
      visit(4)
      → output 4, toposort = [4]
    visit(4)
    → output 3, toposort = [3, 4]
  → output 1, toposort = [1, 3, 4]
  visit(2)
  → output 2, toposort = [2, 1, 3, 4]
  visit(3)
→ output 0, toposort = [0, 2, 1, 3, 4]
```



Topological Sorting

How it works:

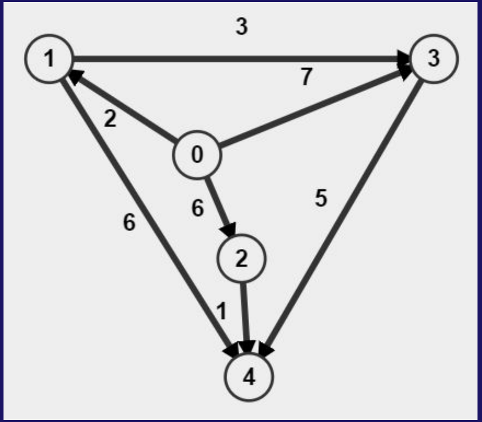
- DFS Toposort Algorithm

  E.g. Let’s try starting from other vertices instead.

  ```
  visit(3)
    visit(4)
    → output 4, toposort = [4]
  → output 3, toposort = [3, 4]
  ```

  We have not visited 0, 1, 2. Let’s visit 1.

  ```
  visit(1)
    visit(3)
    visit(4)
  → output 1, toposort = [1, 3, 4]
  ```



Topological Sorting

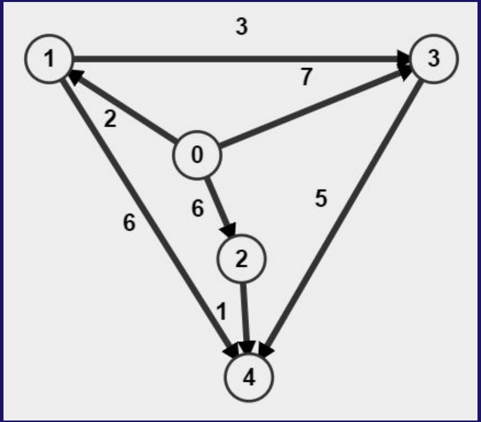
How it works:

- DFS Toposort Algorithm

  E.g. Let’s try starting from other vertices instead.

  We have not visited 0 and 2. Let’s visit 0.

```
visit(0)
  visit(1)
  visit(2)
    visit(4)
  → output 2, toposort = [2, 1, 3, 4]
  visit(3)
→ output 0, toposort = [0, 2, 1, 3, 4]
```



01

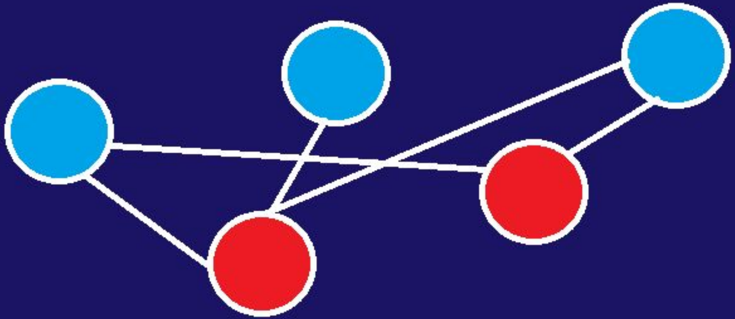
BIPARTITE GRAPH DETECTION

Bipartite Graph Detection

A bipartite graph is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V; but there is no edge between vertices in U and also no edge between vertices in V.

Given an undirected graph with n vertices and m edges, we wish to check if it is bipartite.

Describe the most efficient algorithm you can think of to check whether a graph is bipartite. What is the running time of your algorithm?



Bipartite Graph Detection

Observation(s):

A bipartite graph has the following two properties:

1. 2-colourable: it is possible to assign a colour to every vertex in the graph such that every vertex is coloured one of two colours (say red or blue), such that no two adjacent vertices are coloured with the same colour.
2. No odd-length cycles: every cycle in the graph contains an even number of edges.

A graph possessing any one of the above properties will also possess the other two properties, i.e.

(graph is bipartite) iff (graph is 2-colourable) iff (graph has no odd-length cycles).

Bipartite Graph Detection

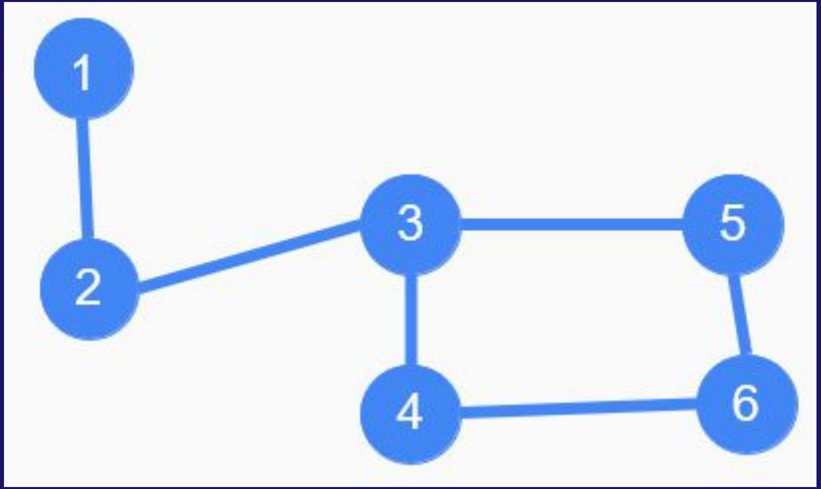
Therefore, to check if a graph is bipartite, we can simply check if it is 2-colourable.

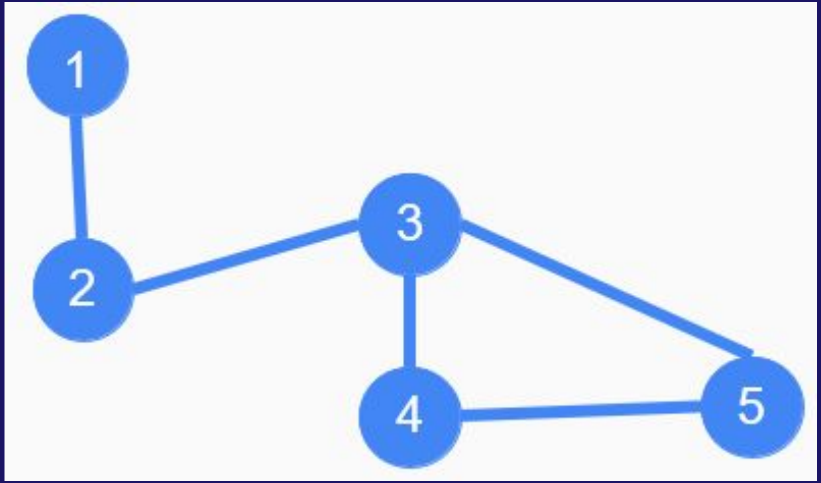
1. We run DFS on the graph, colouring vertices with alternating colours.
2. During the DFS, if we discover that the neighbour of the current vertex is already assigned a colour, and the colour assigned is the same as the colour of the current vertex, we report that the graph is not 2-colourable and hence not bipartite.

   Otherwise, if the DFS manages to assign a colour to every vertex, we report that the graph is bipartite.

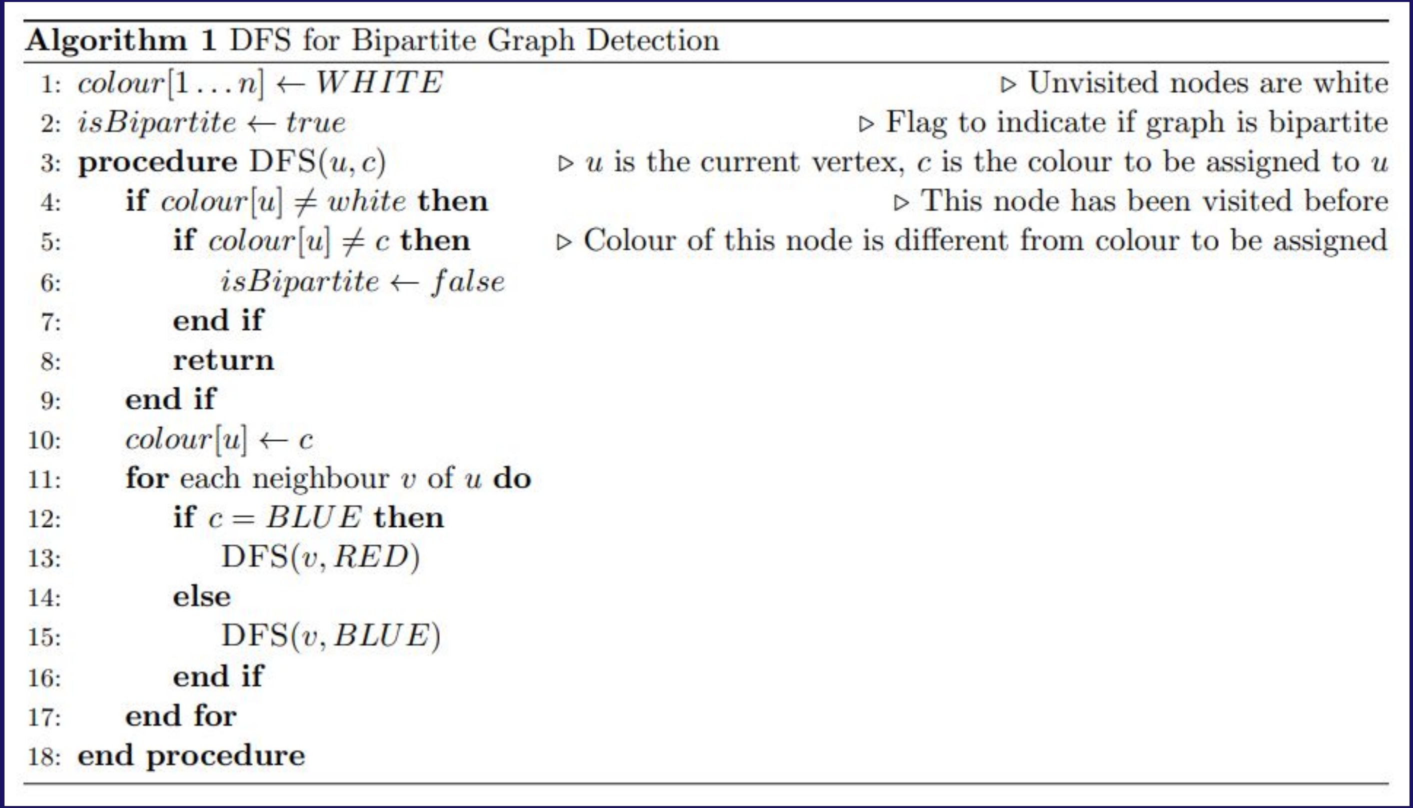
   The running time of this algorithm is $O(n + m)$.

Bipartite Graph Detection





Bipartite Graph Detection



Algorithm 1 DFS for Bipartite Graph Detection

```
$colour[1 \ldots n] \leftarrow WHITE$                    ▷ Unvisited nodes are white
$isBipartite \leftarrow true$                            ▷ Flag to indicate if graph is bipartite
procedure DFS($u, c$)                                    ▷ $u$ is the current vertex, $c$ is the colour to be assigned to $u$
    if $colour[u] \neq white$ then                       ▷ This node has been visited before
        if $colour[u] \neq c$ then                       ▷ Colour of this node is different from colour to be assigned
            $isBipartite \leftarrow false$
        end if
        return
    end if
    $colour[u] \leftarrow c$
    for each neighbour $v$ of $u$ do
        if $c = BLUE$ then
            DFS($v, RED$)
        else
            DFS($v, BLUE$)
        end if
    end for
end procedure
```

02

CYCLE DETECTION

Question 2a

Given a graph with n vertices and m edges, we wish to check if the graph contains a cycle.

First, consider the case of an undirected graph. Describe an algorithm to check if the graph contains a cycle.

Question 2a

First, consider the case of an undirected graph. Describe an algorithm to check if the graph contains a cycle.

Method 1
If the graph does not contain cycles, then it must be a forest: a set of connected components where every connected component is a tree.

One property of trees is: the number of vertices $|V|$ in the tree and the number of edges $|E|$ in the tree is related by $|E| = |V| - 1$.

Therefore, for each component, we can perform a DFS to count the number of vertices $|V|$ and edges $|E|$ in that component, and check if $|E| = |V| - 1$.
Runs in $O(n + m)$ time.

Question 2a

First, consider the case of an undirected graph. Describe an algorithm to check if the graph contains a cycle.

Method 2
The DFS algorithm can be modified to detect cycles in an undirected graph by searching for back-edges, an edge that goes from the current vertex in the DFS to a vertex, other than the parent vertex, that has already been visited before.

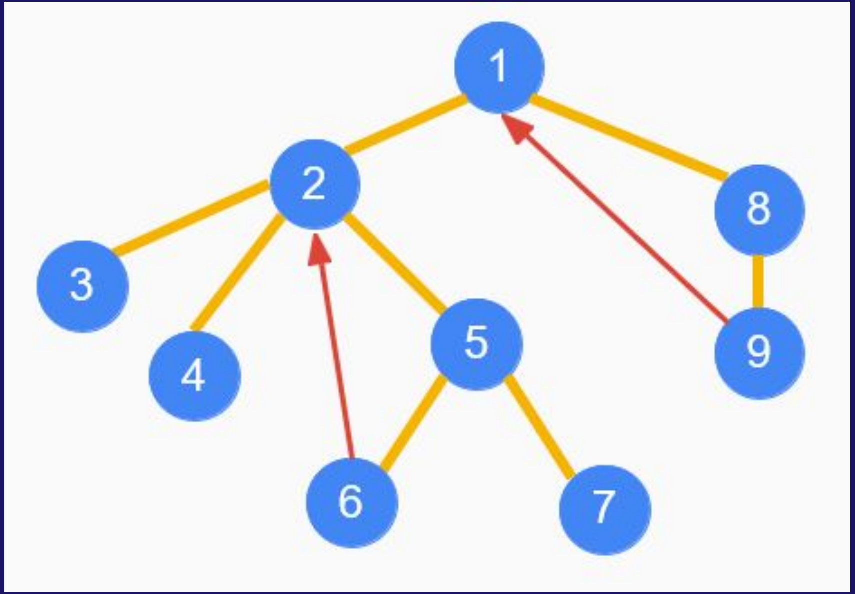
The presence of back-edges indicates the presence of a cycle, since it means we are able to visit the same vertex more than once in the algorithm.

Runs in $O(n + m)$ time.

Question 2a

First, consider the case of an undirected graph. Describe an algorithm to check if the graph contains a cycle.

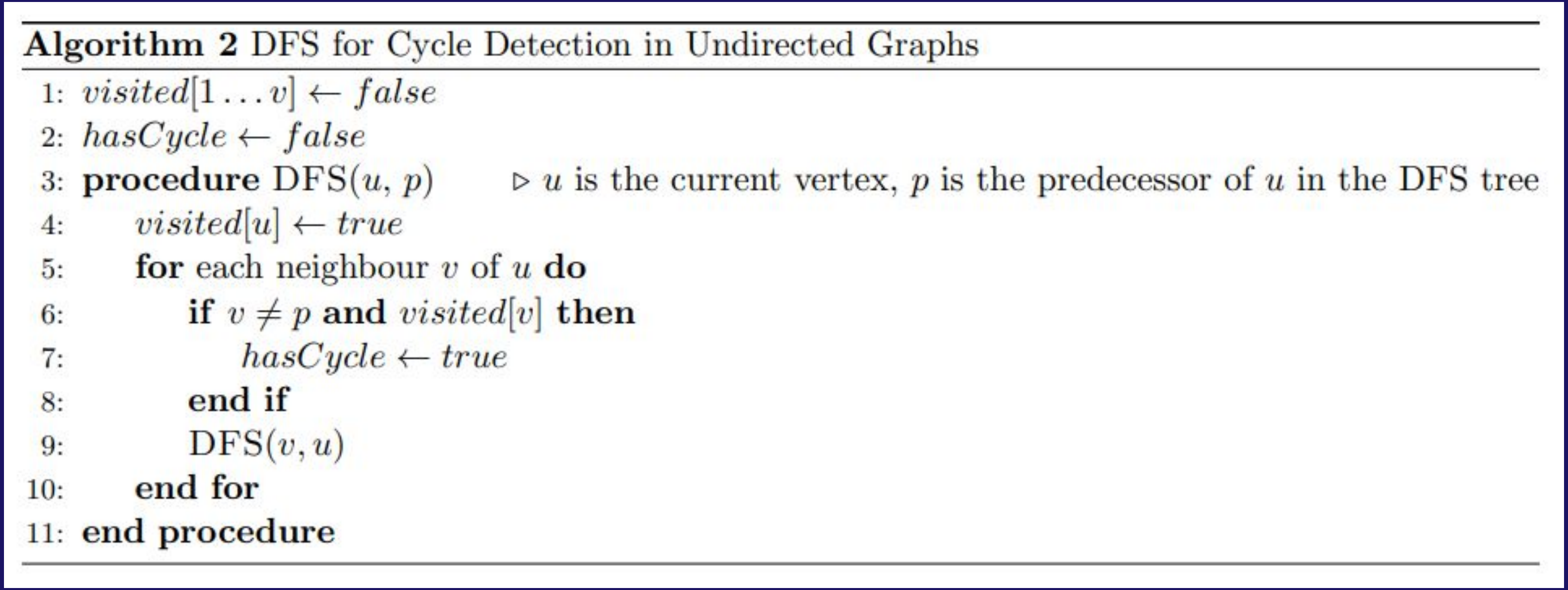
Method 2



Question 2a

First, consider the case of an undirected graph. Describe an algorithm to check if the graph contains a cycle.

Method 2



Algorithm 2 DFS for Cycle Detection in Undirected Graphs

```
1: $visited[1 \ldots v] \leftarrow false$
2: $hasCycle \leftarrow false$
3: procedure DFS($u$, $p$)        ▷ $u$ is the current vertex, $p$ is the predecessor of $u$ in the DFS tree
4:     $visited[u] \leftarrow true$
5:     for each neighbour $v$ of $u$ do
6:         if $v \neq p$ and $visited[v]$ then
7:             $hasCycle \leftarrow true$
8:         end if
9:         DFS($v, u$)
10:    end for
11: end procedure
```

Question 2b

Given a graph with n vertices and m edges, we wish to check if the graph contains a cycle.

Describe an algorithm to check if a directed graph contains a cycle.

Question 2b

Describe an algorithm to check if a directed graph contains a cycle.

Method 1

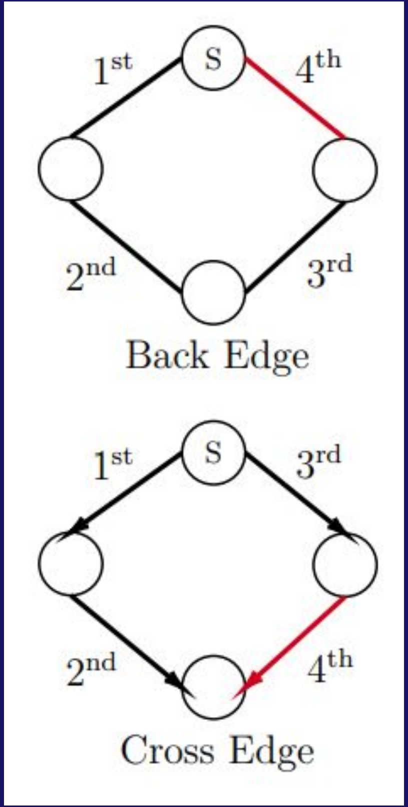
A directed graph that contains a cycle is not a Directed Acyclic Graph (DAG). Therefore, such a graph does not have a topological ordering. Hence, we can run any topological sorting algorithm on the graph, and check if a valid topological ordering can be found.

Question 2b

Describe an algorithm to check if a directed graph contains a cycle.

Method 2
Notice that Algorithm 2 no longer works for the case of a directed graph, as directed graphs contain cross edges.

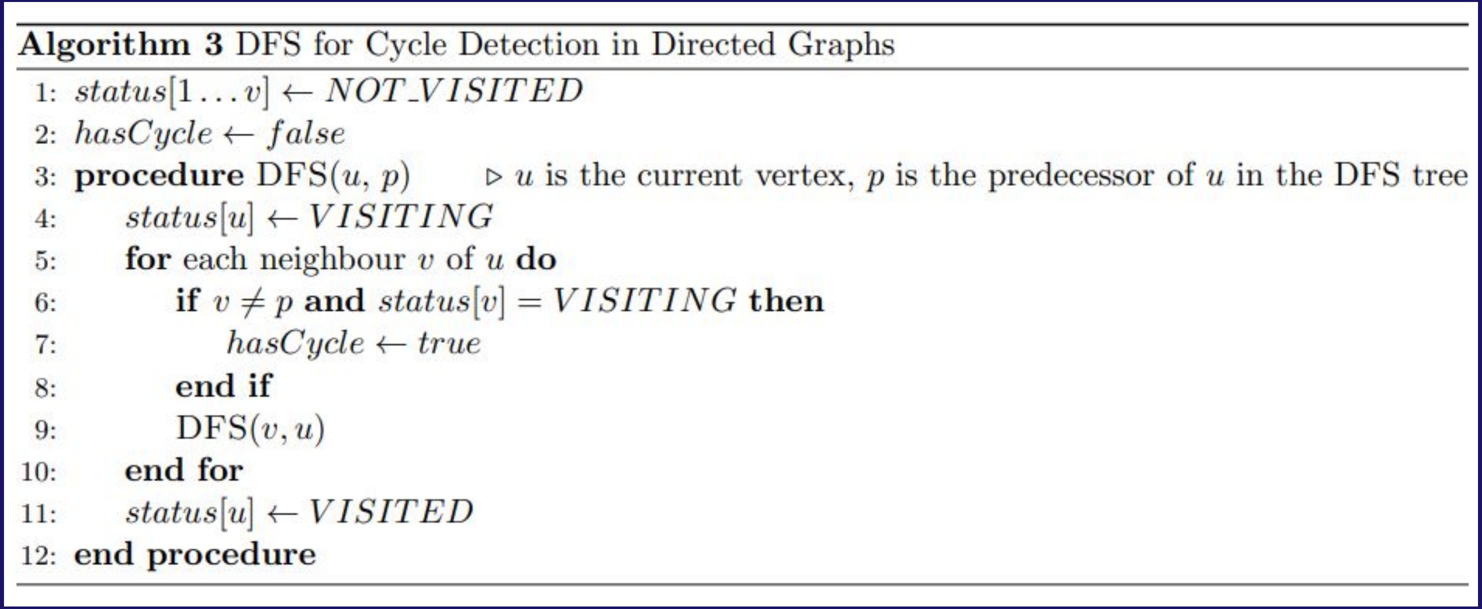
A cross edge is an edge that connects a vertex to a previously visited vertex that is not the parent vertex of the current vertex, and that is not in the same branch of recursion used by DFS to visit the current vertex. Cross edges do not generate cycles, and we can modify Algorithm 2 to avoid reporting a cycle when a cross edge is detected.



Question 2b

Describe an algorithm to check if a directed graph contains a cycle.

Method 2



Algorithm 3 DFS for Cycle Detection in Directed Graphs

```
$status[1 \dots v] \leftarrow NOT\_VISITED$
$hasCycle \leftarrow false$
procedure DFS($u$, $p$)        ▷ $u$ is the current vertex, $p$ is the predecessor of $u$ in the DFS tree
    $status[u] \leftarrow VISITING$
    for each neighbour $v$ of $u$ do
        if $v \neq p$ and $status[v] = VISITING$ then
            $hasCycle \leftarrow true$
        end if
        DFS($v, u$)
    end for
    $status[u] \leftarrow VISITED$
end procedure
```

Question 2b

Describe an algorithm to check if a directed graph contains a cycle.

Method 3

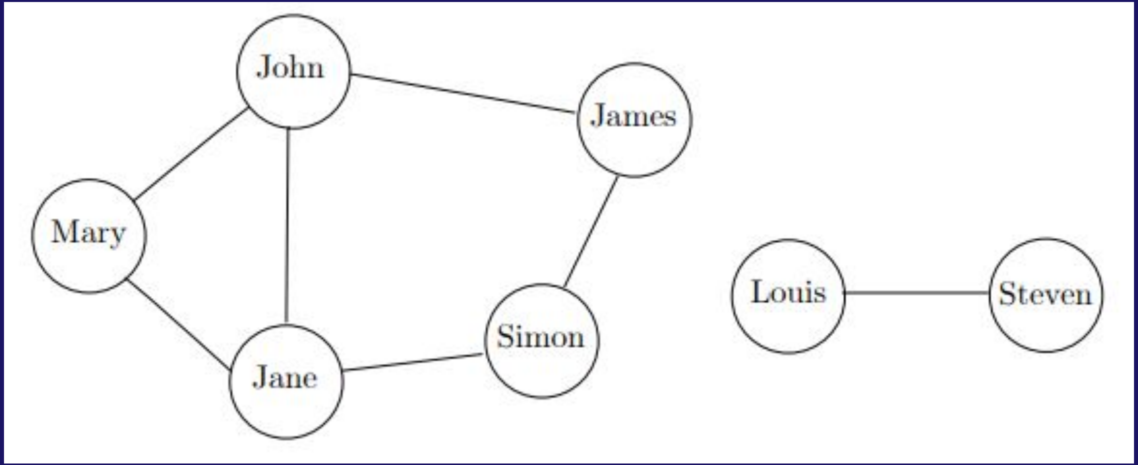
The third way to solve this is to run Kosaraju's algorithm on the directed graph. If the number of SCCs we get is equal to the number of vertices then there is no cycle in the directed graph, otherwise it must mean there is at least one SCC with more than 1 vertex. In that/those SCC(s), there must be a cycle so that each vertex in the SCC can visit every other vertex in the SCC.

03

F·R·I·E·N·D·S NETWORK

Friends Network

A graph with n vertices and m edges, where the vertices represent



Peter's friends and the edges represent friends who know each other directly. Peter wants to find out if a given pair of friends X and Y know each other directly (e.g. John and Jane in the example).

Question 3a

What is the most appropriate graph data structure to store his friends graph in this scenario?

Answer:
An adjacency matrix!

Question 3b

How would he answer his query using the graph data structure you have proposed in Q3a?

Answer:
Check if the the entry in row X, column Y of the adjacency matrix is set to true. This can be done in O(1) time.

Question 3c

Next, Peter wants to know if a given pair of friends X and Y are related to each other indirectly, that is, there is no edge from X to Y but there is at least one path from X to Y with more than 1 edge (e.g. Mary is related indirectly to Simon through Jane among other possibilities in the example).

What is the most appropriate graph data structure to store his friends graph in this scenario?

Answer:
An adjacency list!

Question 3d

Describe an algorithm that answers his query using the graph data structure you have proposed in Q3c. What is the running time of your algorithm?

Answer:

- Scan through the neighbours of X in the adjacency list and check if Y is connected directly to X. If so, we report that X is not indirectly connected to Y. This scan takes $O(n)$ time, since X can be connected to at most $n - 1$ other vertices.
- Otherwise, we perform a DFS starting from X to check if Y is in the same connected component as X. This takes $O(n + m)$ time.

Overall, our algorithm takes $O(n + m)$ time.

Question 3e

Describe the most efficient algorithm you can think of to answer the k queries. What is the running time of each query?

Solution 1
Doing part 3d for exactly k times?

That will take $O(k(n + m))$ time overall.

Can we do better?

Question 3e

Describe the most efficient algorithm you can think of to answer the k queries. What is the running time of each query?

Solution 2

Using UFDS of size n?

Preprocess by merging every two nodes connected by an edge in the graph. We can do this by simply going through all edges (i, j) and merge i and j accordingly. This should take $O(m\ \alpha(n))$ time.

For each query (x, y):

- Check if x and y are directly connected using the adjacency matrix we had from 3b in $O(1)$ time. If they are directly related, then they cannot be indirectly related.
- Otherwise, check if x and y are on the same set using the UFDS in $O(\alpha(n))$ time.

Overall, the running time is $O((m + k)\alpha(n))$.

Can we do better?

Question 3e

Describe the most efficient algorithm you can think of to answer the k queries. What is the running time of each query?

Solution 3
By extending the idea in 3d, we perform a DFS on the adjacency list to label each connected component in the graph. Each vertex is labeled with the component number of the component it belongs to, and we can store the component numbers in an array.

This preprocessing step takes $O(n + m)$ time.

Question 3e

Describe the most efficient algorithm you can think of to answer the k queries. What is the running time of each query?

Solution 3
To answer each query of whether two given friends X and Y are indirectly related, we first check if X and Y are directly related in O(1) time using the adjacency matrix as done in 3b. If they are directly related, then they cannot be indirectly related.

Otherwise, we can check if the component numbers of X and Y are the same in O(1) time. If their component numbers are the same, they are in the same connected component and thus indirectly related, else they are not related at all.

Thus, each query will take O(1) time.

Overall, this solution takes O(n + m + k) time, where k is the number of queries.

04

SKYSCRAPERS

Question 4a

There are n skyscrapers in a city, numbered from 1 to n. You would like to order the skyscrapers by height, from the tallest to the shortest. Unfortunately, you do not know the exact heights of the skyscrapers.

Suppose that you have m pieces of information about the skyscrapers. Each piece of information tells you that skyscraper x is taller than skyscraper y for some pair of skyscrapers x and y.

Describe the most efficient algorithm you can think of to output any one possible ordering of the buildings by height which is consistent with the m pieces of information given. What is the running time of your algorithm?

Why not bubble sort?

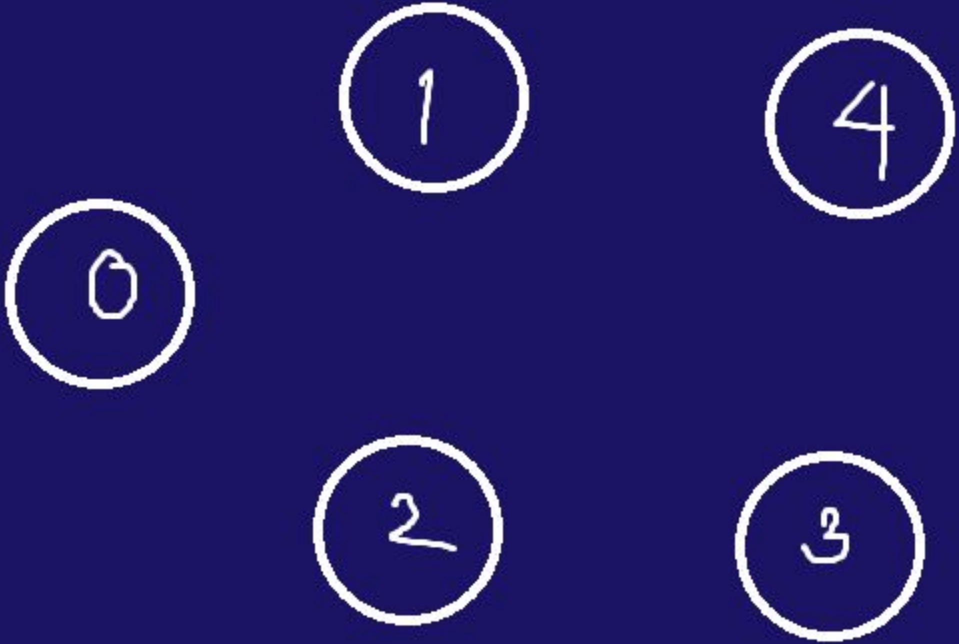
Question 4a

We can model the graph by representing the n skyscrapers as vertices, and the m pieces of information as directed edges.

If the piece of information tells you that skyscraper x is taller than skyscraper y, then we represent that as a directed edge from x to y. Specifically, the graph will be a Directed Acyclic Graph (DAG), as the graph cannot have cycles (the heights cannot have a circular relation).

Getting a possible ordering of heights consistent with the information given thus equivalent to finding a topological ordering of the vertices. We can run the Topological Sort algorithm which has time complexity O(n + m).

Question 4a



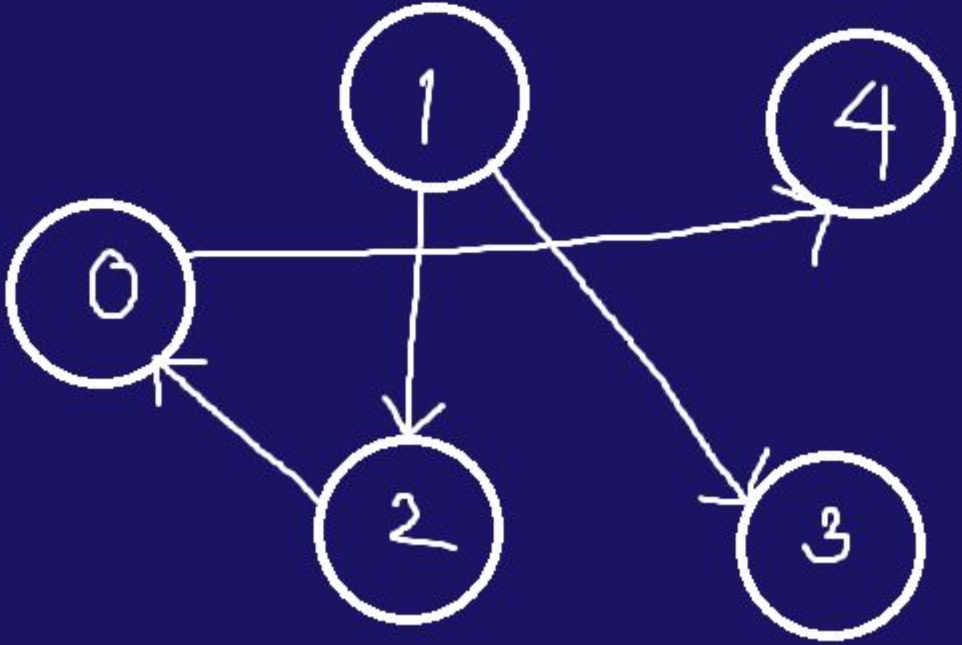
$h_1 > h_2$

$h_2 > h_0$

$h_1 > h_3$

$h_0 > h_4$

Question 4a



$h_1 > h_2$

$h_2 > h_0$

$h_1 > h_3$

$h_0 > h_4$

Question 4b

Suppose that you have m pieces of information about the skyscrapers. Each piece of information tells you one of the following regarding two skyscrapers x and y:

- x is taller than y
- x has the same height as y

Describe the most efficient algorithm you can think of to output any one possible ordering of the buildings by height. What is the running time of your algorithm?

Question 4b

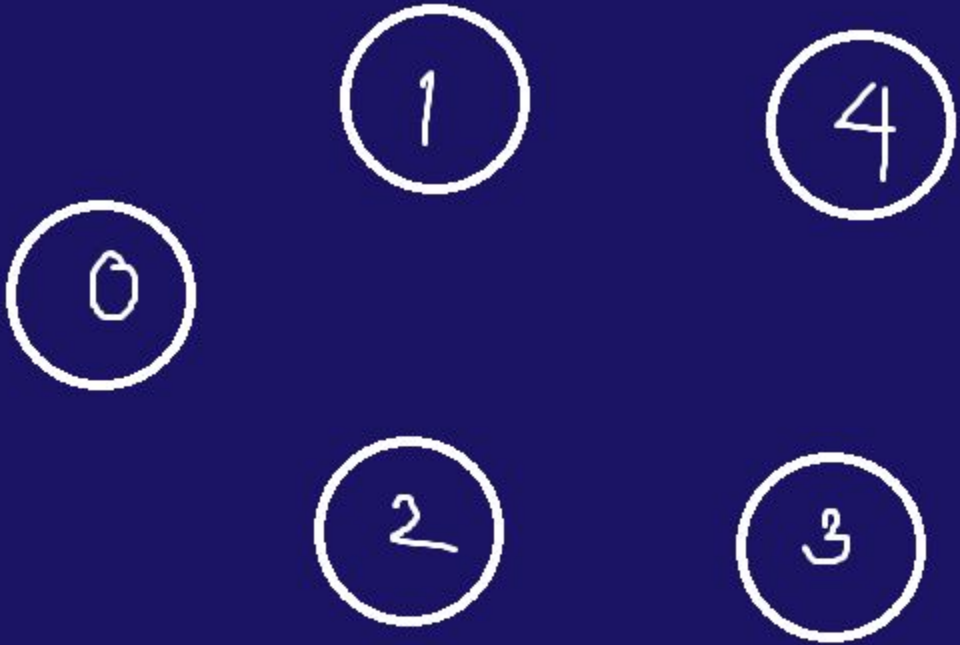
Since there can be skyscrapers with the same height, we cannot directly apply the Topological Sort algorithm in 4a since we have two different types of relations.

However, using a UFDS we can “combine” the skyscrapers with the same height as one single vertex by reading in all the “equality” relations and doing a unionSet operation on each pair of equal skyscrapers to create our combined “vertex”.

Thus, each “vertex” will represent one or more skyscrapers with the same height. This allows us to again obtain a DAG, which we can now apply the Topological Sort algorithm with O(n + m) time complexity.

This method is called edge contraction.

Question 4b



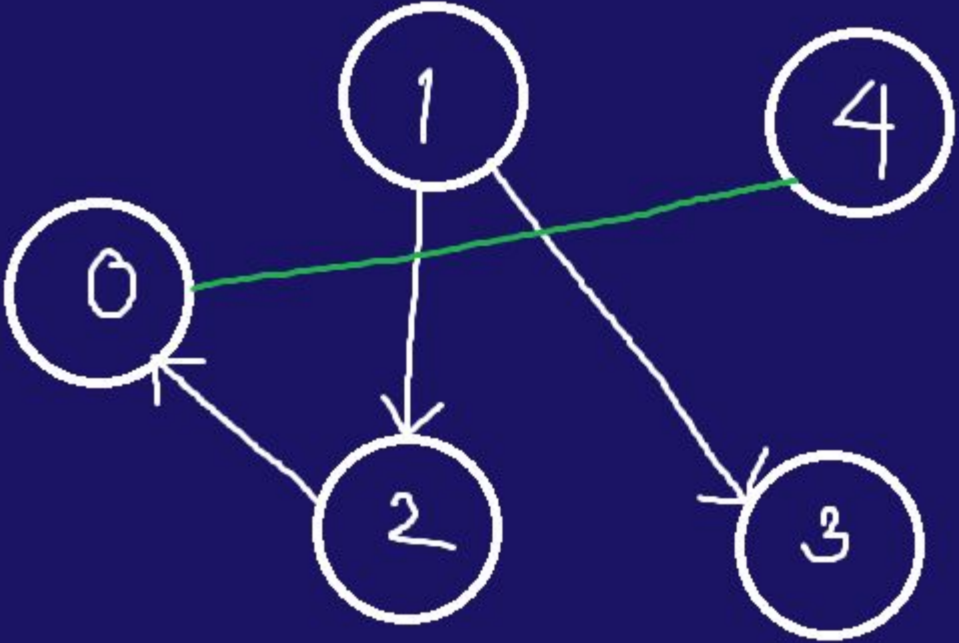
$h_1 > h_2$

$h_2 > h_0$

$h_1 > h_3$

$h_0 = h_4$

Question 4b



$h_1 > h_2$

$h_2 > h_0$

$h_1 > h_3$

$h_0 = h_4$

Question 4b

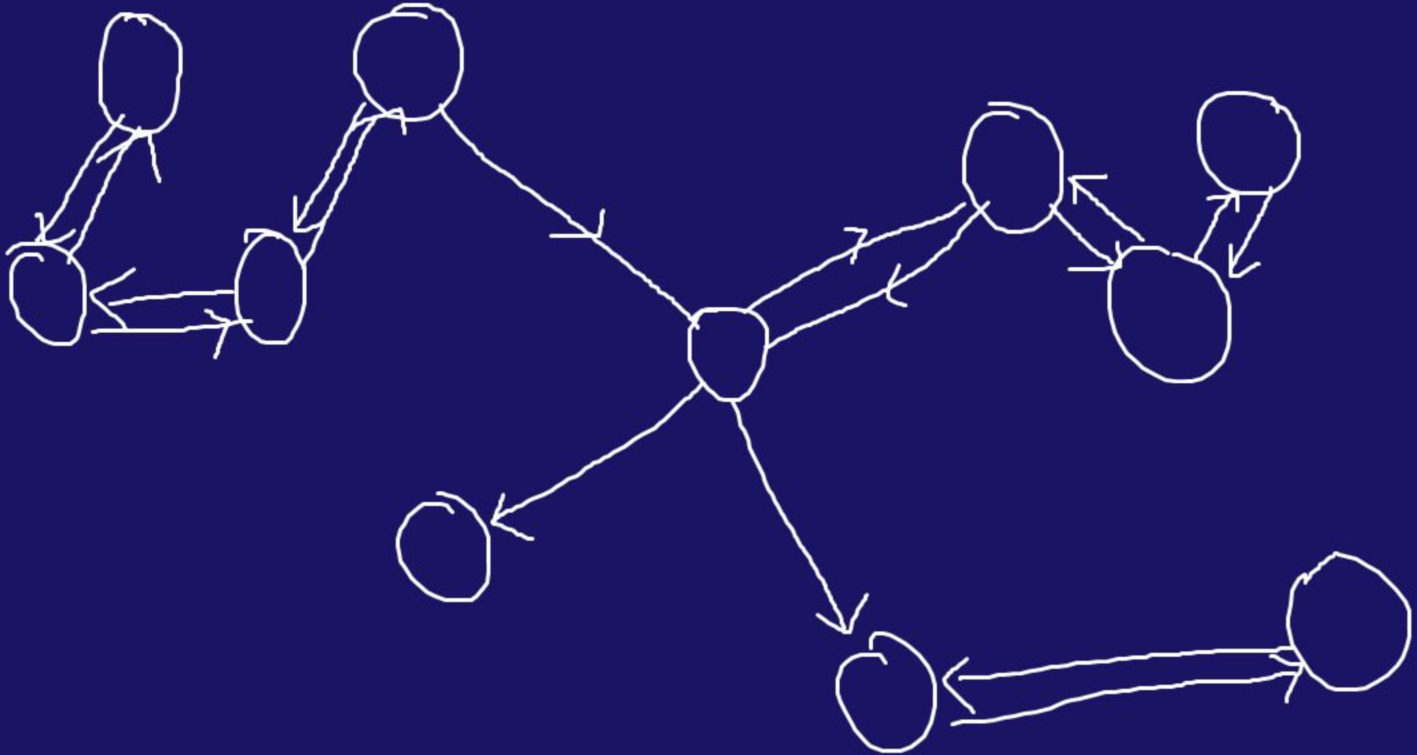
Instead of using UFDS, we can connect two skyscrapers with the same height with a bi-directional edge.

Thus all vertices which are of the same height must form a SCC (here we consider case of 2 vertices linked by a bi-directed edge as forming a cycle).

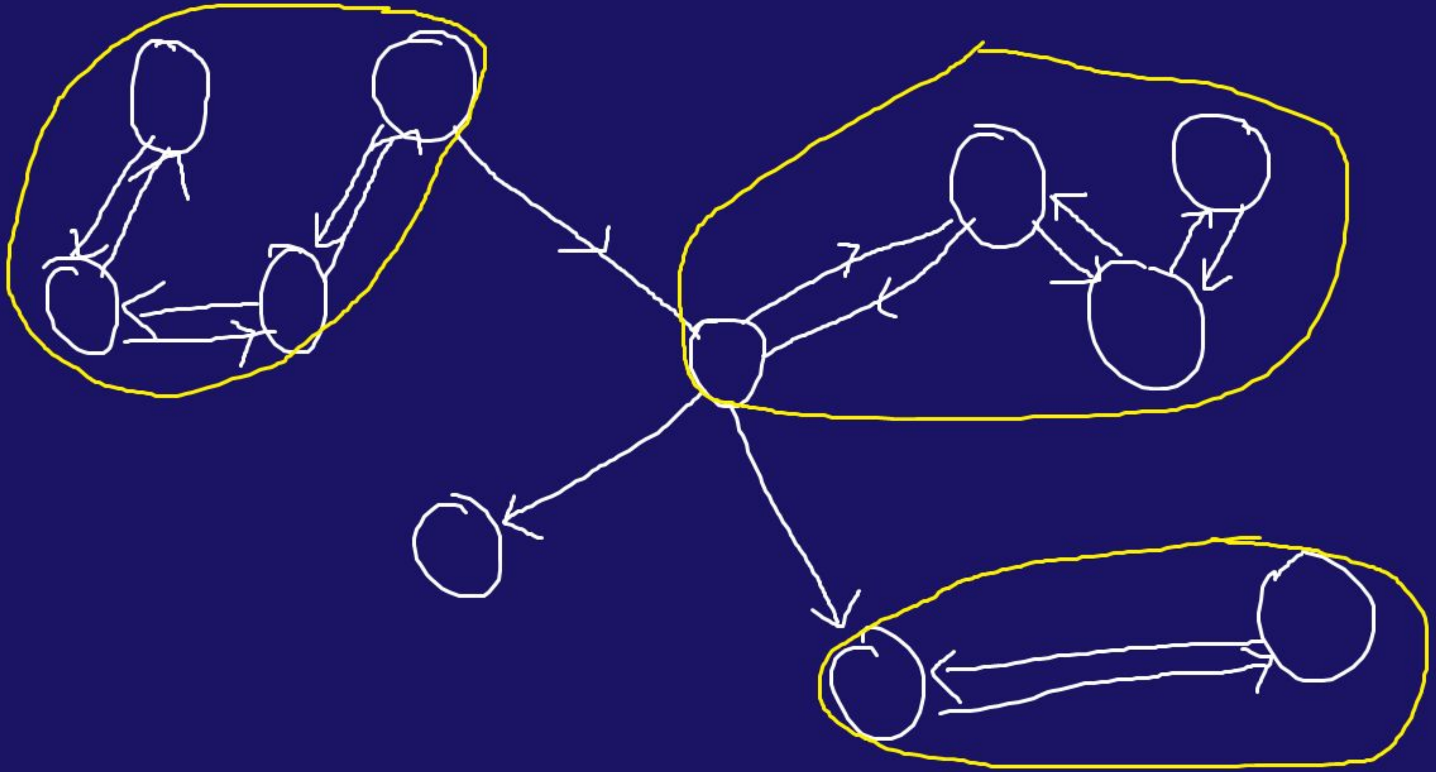
Run Kosaraju's algorithm to count SCCs and label all vertices in each SCC by the current SCC count (vertices in 1st SCC will be labeled 1, vertices in 2nd SCC will be labeled 2, etc).

If each of the SCC is viewed as a vertex, you can see that the graph is now a DAG and thus Kosaraju's algorithm will go through the SCCs in topological ordering. After running Kosaraju, simply sort the vertices based on increasing SCC labeling and you will have a valid ordering.

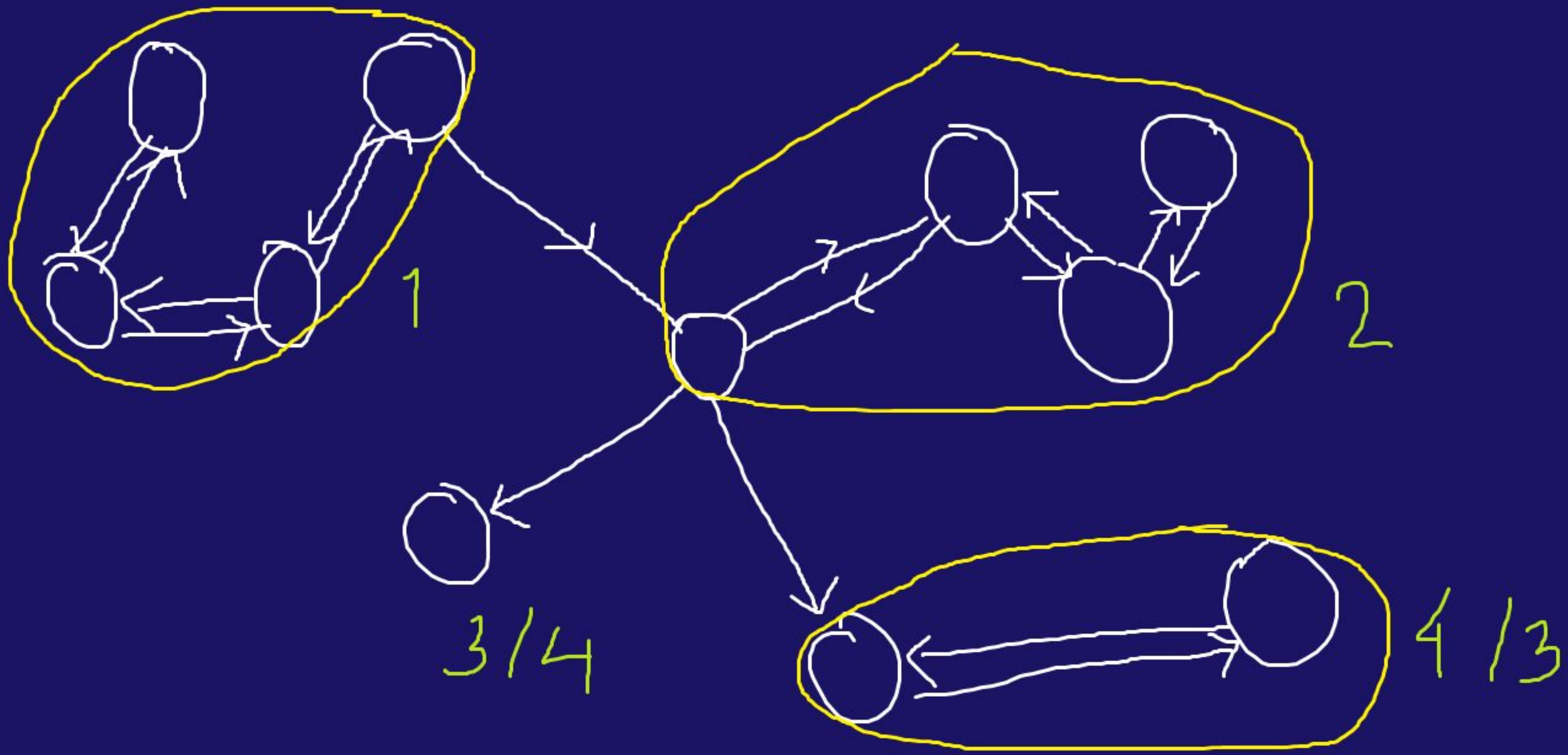
Question 4b



Question 4b



Question 4b



05

EXTRAS

because of Zoom :)

Strongly Connected Graph

A directed graph is called strongly connected if it is possible to reach any vertex from any vertex.

Given a directed graph, can you check if it is strongly connected in O(V + E)?

Strongly Connected Graph

A directed graph is called strongly connected if it is possible to reach any vertex from any vertex.

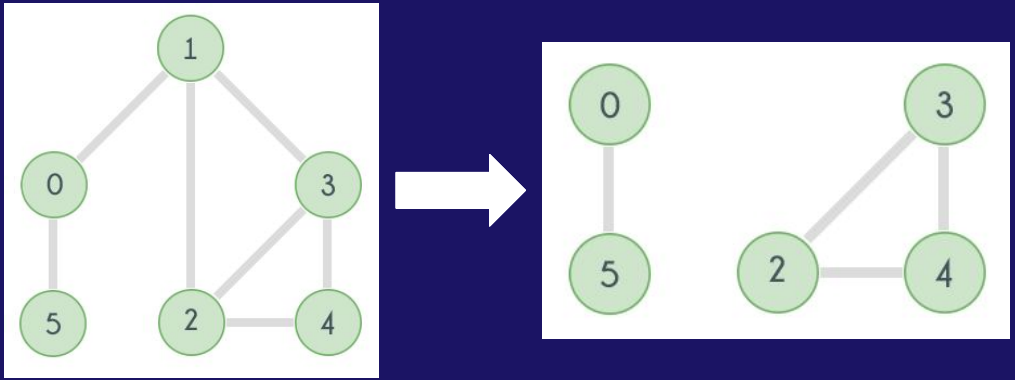
Given a directed graph, can you check if it is strongly connected in O(V + E)?

Yes, simply run Kosaraju and check if the number of SCC is exactly one. Otherwise, there is at least a pair of vertices (x, y) such that you can’t reach y from x.

Bridges and Articulation Points (challenging!)

An edge is called a bridge if, upon removal of the edge, the graph becomes disconnected. (e.g. edge 0-1 below)

A vertex is called an articulation point if, upon removal of the vertex, the graph becomes disconnected. (e.g. vertex 1)



Design an algorithm to find all bridges and articulation points in a graph. Your algorithm should run in O(V + E).

Bridges and Articulation Points (challenging!)

Let's start with finding bridges first.

Naive approach:
For every edge e, remove e from G, check if the number of connected components increases, then re-add e back to G.

The total time complexity is $O(E * (V+E)) = O(E^2 + VE)$ because you're running BFS/DFS to count CCs for E times.

Bridges and Articulation Points (challenging!)

Optimal approach:
While not officially taught in CS2040, we are to run a modified version of DFS called Tarjan’s algorithm!

Maintain two arrays disc and low to store the discovery time of each vertex and the lowest discovery time of its possible predecessors.

Bridges and Articulation Points (challenging!)

Optimal approach:
Let’s observe how DFS algorithm can evolve to Tarjan’s algorithm.
By default we already have the visited array.

We simply repeatedly call DFS on any unvisited vertices. If the vertex u has no parent, set par = -1.

```
DFS(u, par)
    visited[u] = true
    for v in graph[u]:
        if not visited[v]:
            DFS(v, u)
        else:
            do nothing
```

Bridges and Articulation Points (challenging!)

Optimal approach:
Next, we keep track of number of childrens.

```
DFS(u, par)
    visited[u] = true
    children = 0
    for v in graph[u]:
        if not visited[v]:
            children += 1
            DFS(v, u)
        else:
            do nothing
```

Bridges and Articulation Points (challenging!)

Optimal approach:
Next, we keep track of the discovery time in the array disc.

```
set global variable time = 0
DFS(u, par)
    visited[u] = true
    children = 0
    disc[u] = time
    time += 1
    for v in graph[u]:
        if not visited[v]:
            children += 1
            DFS(v, u)
        else:
            do nothing
```

Bridges and Articulation Points (challenging!)

Optimal approach:
Next, we keep track of the lowest discovery time of the preds in low.

```
set global variable time = 0
DFS(u, par)
    visited[u] = true
    children = 0
    disc[u] = time
    low[u] = time
    time += 1
    for v in graph[u]:
        if not visited[v]:
            children += 1
            DFS(v, u)
            low[u] = min(low[u], low[v]) // update low after we run DFS completely on v
        else:
            do nothing
```

Bridges and Articulation Points (challenging!)

Optimal approach:
However, you also need to take into account of the back-edges!

```
set global variable time = 0
DFS(u, par)
    visited[u] = true
    children = 0
    disc[u] = time
    low[u] = time
    time += 1
    for v in graph[u]:
        if not visited[v]:
            children += 1
            DFS(v, u)
            low[u] = min(low[u], low[v])
        else if par != v: // this is a back-edge u→v
            low[u] = min(low[u], disc[v])
```

Bridges and Articulation Points (challenging!)

Finally check if an edge u-v is a bridge! The idea is that there should not be a back-edge to vertex u or its ancestors on the DFS tree.

```
set global variable time = 0
DFS(u, par)
    visited[u] = true
    children = 0
    disc[u] = time
    low[u] = time
    time += 1
    for v in graph[u]:
        if not visited[v]:
            children += 1
            DFS(v, u)
            low[u] = min(low[u], low[v])
            if low[v] > disc[u]:
                u-v is a bridge!
        else if par != v:
            low[u] = min(low[u], disc[v])
```

Bridges and Articulation Points (challenging!)

Now let's find the articulation points (also known as the cut vertices), which also uses Tarjan's algorithm!
Suppose we're back to this state just before the final checking:

```
set global variable time = 0
DFS(u, par)
    visited[u] = true
    children = 0
    disc[u] = time
    low[u] = time
    time += 1
    for v in graph[u]:
        if not visited[v]:
            children += 1
            DFS(v, u)
            low[u] = min(low[u], low[v])
        else if par != v:
            low[u] = min(low[u], disc[v])
```

Bridges and Articulation Points (challenging!)

Now to decide if a vertex u is a cut vertex?

```
set global variable time = 0
DFS(u, par)
    visited[u] = true
    children = 0
    disc[u] = time
    low[u] = time
    time += 1
    for v in graph[u]:
        if not visited[v]:
            children += 1
            DFS(v, u)
            low[u] = min(low[u], low[v])
            // Case 1:
            if u is not the root of DFS tree (e.g. par != -1) and low[v] >= disc[u]:
                u is a cut vertex!
        else if par != v:
            low[u] = min(low[u], disc[v])
    // Case 2:
    if u is the root of DFS tree (e.g. par == -1) and children > 1:
        u is a cut vertex!
```

THE END!