CSCI 2270: Data Structures

Recitation #11 (Section 101)

Office Hours

- Name: Himanshu Gupta
  Email: himanshu.gupta@colorado.edu

- **Office Hours**
  - **12pm to 2pm on Mondays**
  - **12:30pm to 2:30pm on Fridays**
  - **Same Zoom ID as that of recitation - https://cuboulder.zoom.us/j/3112555724**

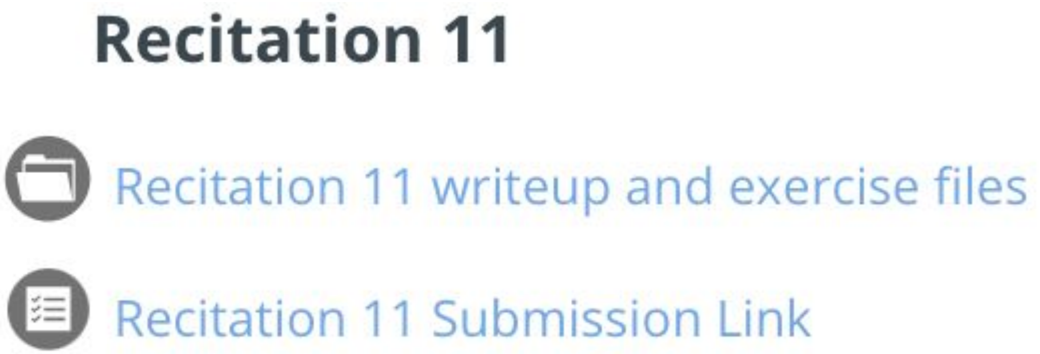
- In case that doesn't work for you, shoot me an email. We will figure something out that works for both of us.

Logistics

- In case you have a question during the recitation, just unmute yourself and speak up. Let's try to make it as interactive as possible.
- You can ask questions via chat as well, but I would prefer if you guys ask your questions verbally.
- I would highly encourage you guys to switch on your cameras as well (if possible). It helps in making the session more lively and interactive.

Logistics: Attendance for Recitation 11

- Upload a single zip file with solved code to Moodle.
- Your points for Recitation 11 depend on this.



- **Due Date - Sunday, April 5 2020, 11:59 PM**

Logistics

- **Midterm 2**
    - Scheduled on April 10 2020, 5 PM (Friday)

    - Tentative format:
        - Conceptual, short answer, multiple choice questions, 1 hour on Moodle
        - Coding questions: due by midnight

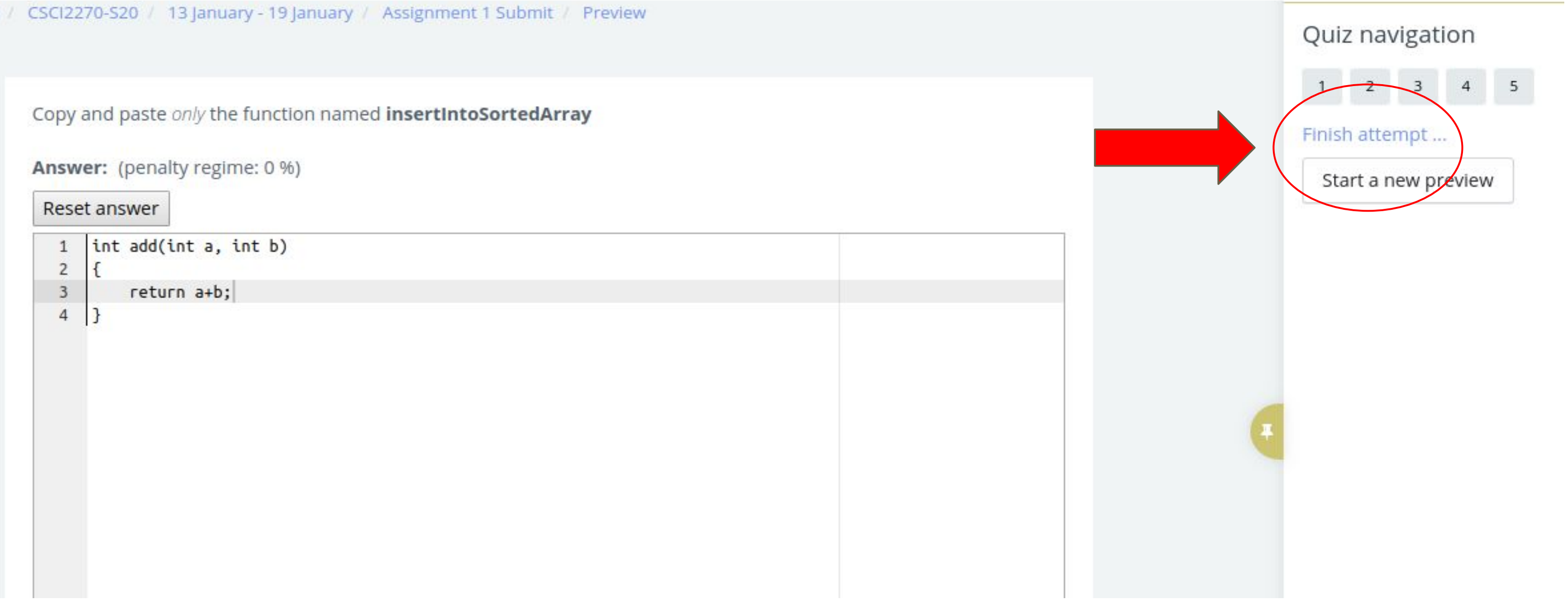
- **Final project**
    - It will be assigned right after Midterm 2, and will be due by on April 26 2020.

Logistics

- **Assignment 8 is due on Sunday, April 5 2020, 11:59 PM**

GOOD LUCK!

Please click on "Finish Attempt" after you are done!



Any questions?

Agenda

- Briefly reviewing Graph representations and BFS Traversal
  - Instructors have requested to do this.
  - It might be a bit boring for those who have understood it completely, but please bear with me.

- Motivation behind Dijkstra's algorithm: Shortest Path in a Weighted Graph
  - It should be covered in detail during the lecture. We won't be discussing this in detail, at least not today.

- **Exercise**
  - **Find whether a given edge is a "bridge" of the graph.**
  - **Not related to Dijkstra's algorithm in any way possible.**

Let's look at how to represent graphs in code.

- Vertex

```
struct vertex{
    int key;
    bool visited = false;
    std::vector<adjVertex> adj;
};
```

```
struct adjVertex{
    vertex *v;
};
```

- **key** stores the value of that vertex
- **visited** allows us to infer if a node has been visited or not while traversing the graph.
- **adj** is a vector of type adjVertex that is our adjacency list. It stores all the vertices that are directly connected to the current vertex/node via an edge/link.
- **adjVertex** is just a struct that has a pointer of type vertex in it.

Let's look at how to represent graphs in code.

- **addVertex** function adds a new vertex with value v
- **addEdge** function adds an edge between vertex v1 and v2
- **vertices** is a vector that stores pointers to all the vertices in the graph

```
class Graph
{
    public:
        void addEdge(int v1, int v2);
        void addVertex(int v);
        void printGraph();

    private:
        std::vector<vertex*> vertices;

};
```

Few basic vector commands

- Access element at index **i** in a vector graph_vertices
  - graph_vertices[i] or graph_vertices.at(i)

- Size of a vector graph_vertices
  - graph_vertices.size()

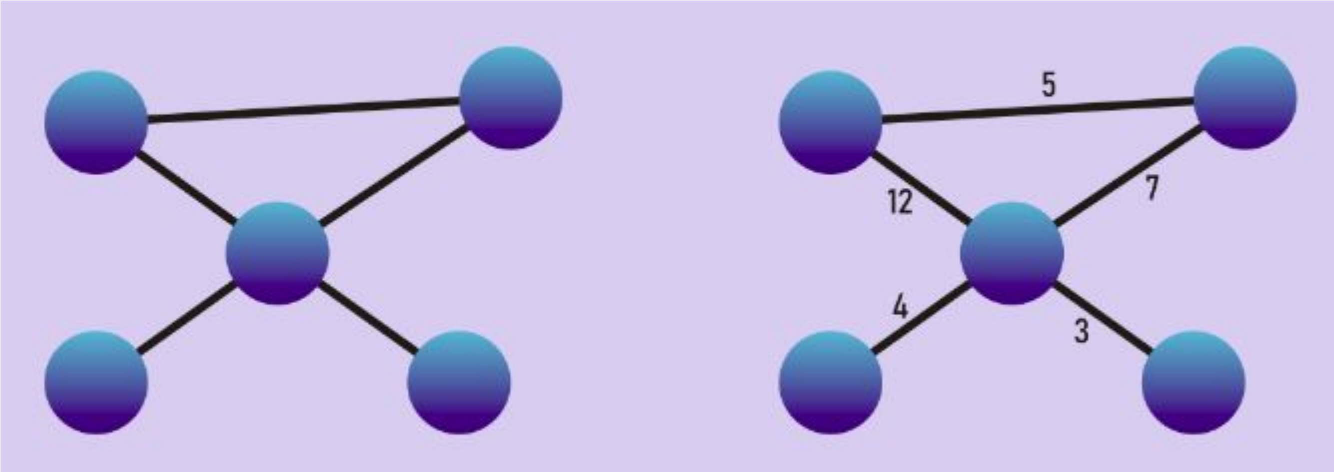
- **Access the value of j^th vertex in the adjacency list of a vertex at index i in vector graph_vertices**
  - **graph_vertices[i]->adj[j].v->key**

Weighted and Unweighted Graphs

- **Unweighted Graphs**
  - None of the edges in the graph has any weight associated to it.
  - This is equivalent to saying that all the edges in the graph have a same weight of value 1.
  - For example: For depicting how many people know each other in a room, where every vertex is a person and an edge between them denotes that they know each other, you don't need those edges to have any weight.

- **Weighted Graphs**
  - All edges in the graph have some weight associated to them. Different edges can have different or same weights.
  - Based on their weights, some edges are preferred more over others.
  - For example: For depicting map of a state, where each vertex is a city, each edge connecting two cities/vertices should have a weight which is equal to the displacement between the cities.

Weighted and Unweighted Graphs



Unweighted Graph

Weighted Graph

Graph Traversal

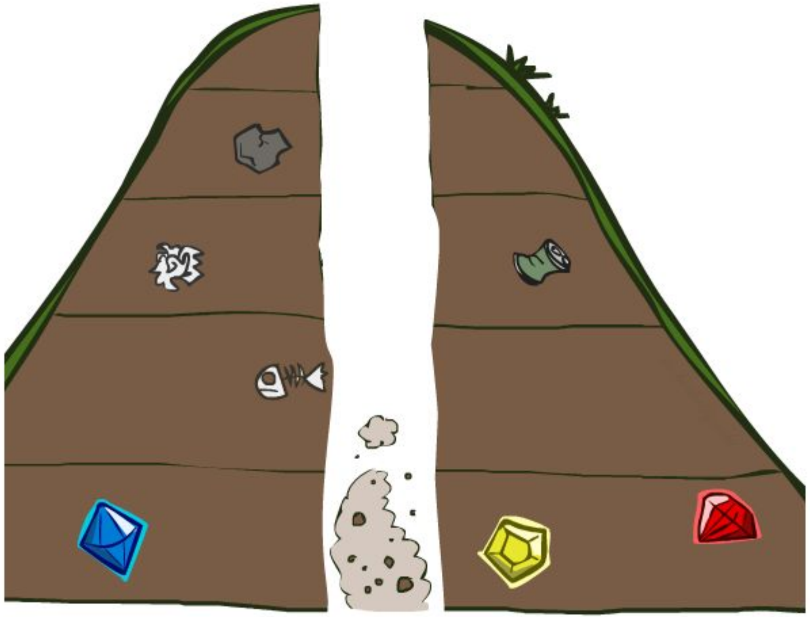
- Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph.

- Such traversals are classified by the order in which the vertices are visited.

- Traversal techniques we will be discussing today -
  - **Depth First Search (DFS)**
  - **Breadth First Search (BFS)**

Depth First Search (DFS)

- Strategy: expand the deepest node first. We use **stack** data structure for doing DFS



Depth First Search (DFS)

- A depth-first search (DFS) is an algorithm for traversing a finite graph.
- DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth.
- Iterative implementation of DFS - creating and using a stack for it
- Recursive implementation of DFS - using recursion to traverse the graph, but those recursive calls are implicitly stored on the system stack anyways.

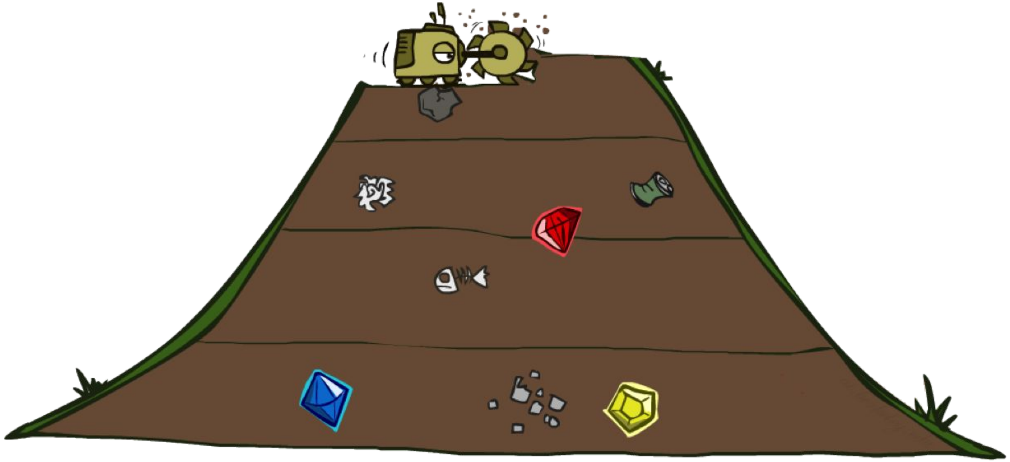
DFS Psuedocode to search for a node in the graph

- Recursive implementation

```
function DFS(G, v)
        v.visited = true
        if v is what we are looking for then
                return v
        for all vertices X in G.adjacencyList(v) do
                if X.visited == false, then
                        To_be_returned = DFS(G,X)
                        if(To_be_returned != NULL)
                                return To_be_returned
        return null
```

Breadth First Search (BFS)

- Strategy: expand the shallowest node first. We use **queue** data structure for doing BFS



Breadth First Search (BFS)

- A breadth-first search (DFS) is an algorithm for traversing a finite graph.
- BFS visits the sibling vertices before visiting the child vertices, and a queue is used in the search process.

BFS Psuedocode to search for a node in the graph

```
function BFS(G, v)
	create a queue Q
	enqueue v onto Q
	v.visited = true
	while Q is not empty do
		w ← Q.dequeue()
		if w is what we are looking for then
			return w
		for all vertices X in G.adjacencyList(w) do
			if X.visited == false, then
				X.visited = true
				enqueue X onto Q
	return null
```

Find shortest path between two nodes in an unweighted graph

- Recitation 10 exercise
- We used BFS traversal to do this.

Find shortest path between two nodes in an unweighted graph

- Psuedocode

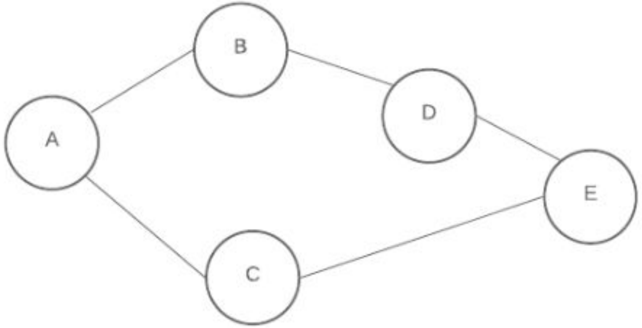
```
find_shortest_path(vertex src, vertex des)
{
    src.visited = True //Mark src vertex as visited
    Create an empty queue Q and enqueue src vertex onto it.
    while Q is not empty
        curr_vertex = Q.dequeue()
        for nebhr in adjacency list of curr_vertex
            if nebhr has not been visited
                nebhr.visited = True //Mark adjacent vertex as visited
                nebhr.distance = curr_vertex.distance + 1 //Increment the distance
                of adjacent vertex
                Q.enqueue(nebhr) //Enqueue the adjacent vertex to the queue
            if( nebhr == des)  //Check if the adjacent vertex is the destination vertex
                return nebhr.distance
}
```

BFS to find length of shortest path on an unweighted graph

src_vertex = A

des_vertex = E

Q = $\boxed{}$



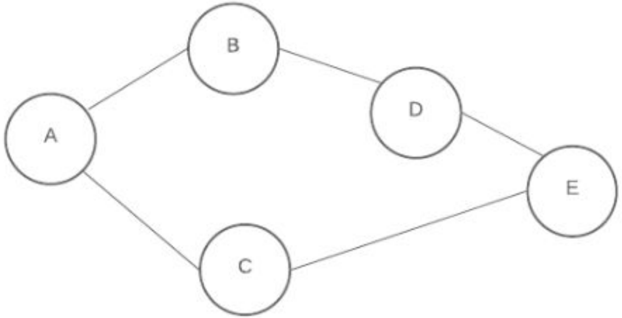
Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	False	False	False	False	False

BFS to find length of shortest path on an unweighted graph

src_vertex = A

des_vertex = E

Q = A

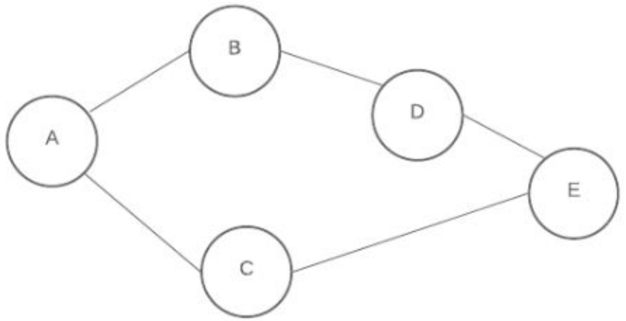


Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	True	False	False	False	False

BFS to find length of shortest path on an unweighted graph

src_vertex = A
des_vertex = E

Q = A



Dequeue vertex from Q and explore its adjacent vertıces

Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	True	False	False	False	False

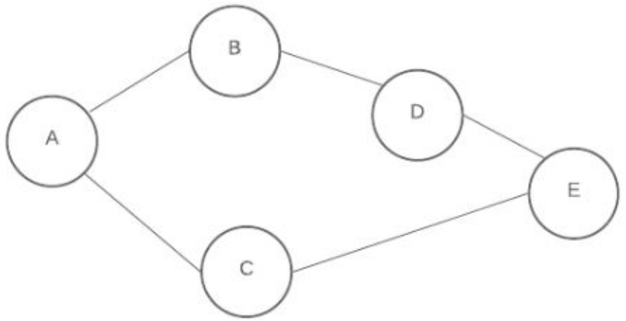
BFS to find length of shortest path on an unweighted graph

src_vertex = A

des_vertex = E

Q =

C	B

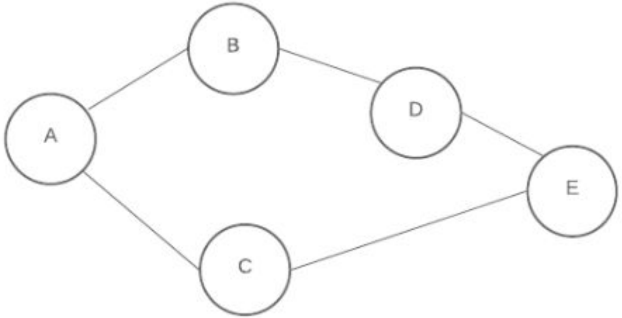


Vertex	A	B	C	D	E
Distance	0	1	1	0	0
Visited	True	True	True	False	False

BFS to find length of shortest path on an unweighted graph

src_vertex = A
des_vertex = E

Q = | C | B |



Dequeue vertex from Q and explore its adjacent vertices

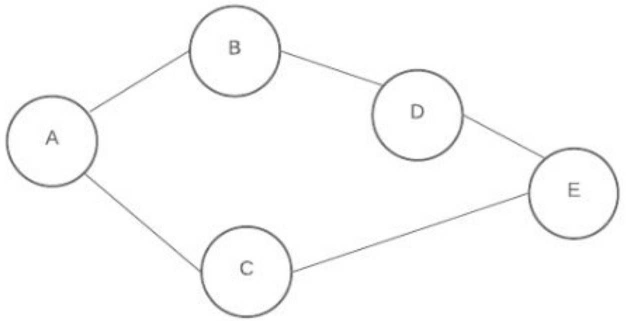
Vertex	A	B	C	D	E
Distance	0	1	1	0	0
Visited	True	True	True	False	False

BFS to find length of shortest path on an unweighted graph

src_vertex = A
des_vertex = E

Q =

D	C



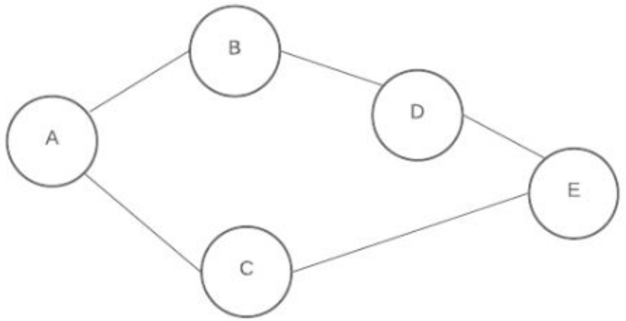
Vertex	A	B	C	D	E
Distance	0	1	1	2	0
Visited	True	True	True	True	False

BFS to find length of shortest path on an unweighted graph

src_vertex = A
des_vertex = E

Q =

D	C



Dequeue vertex from Q and explore its adjacent vertıces

Vertex	A	B	C	D	E
Distance	0	1	1	2	0
Visited	True	True	True	True	False

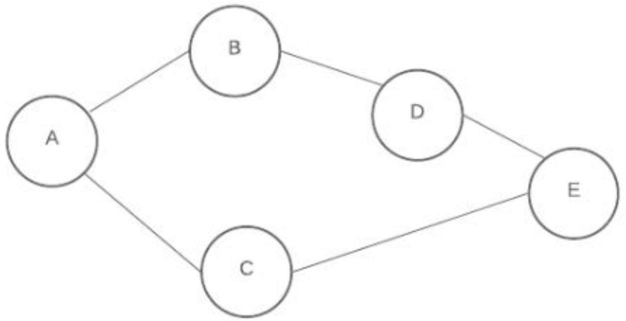
BFS to find length of shortest path on an unweighted graph

src_vertex = A

des_vertex = E

Q =

E	D



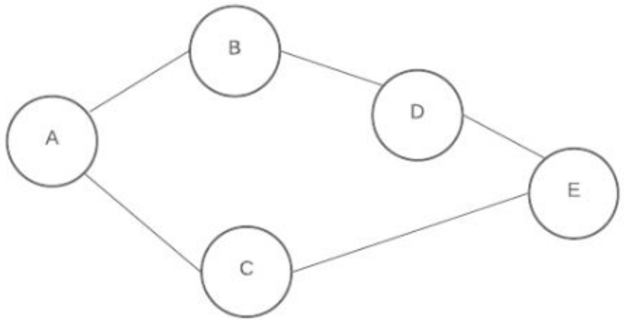
Vertex	A	B	C	D	E
Distance	0	1	1	2	2
Visited	True	True	True	True	True

BFS to find length of shortest path on an unweighted graph

src_vertex = A

des_vertex = E

Q = | E | D |



Therefore, path from A to E has distance 2

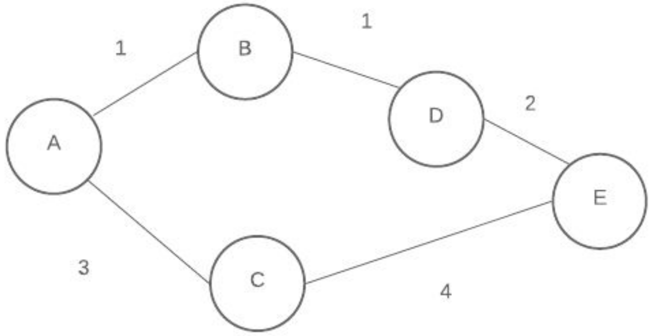
Vertex	A	B	C	D	E
Distance	0	1	1	2	2
Visited	True	True	True	True	True

Let's apply the exact same algorithm to a weighted graph and see the result

BFS to find length of shortest path on a weighted graph

src_vertex = A
des_vertex = E

Q = $\boxed{}$

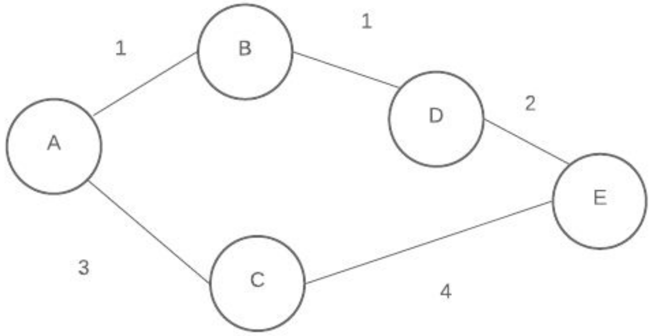


Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	False	False	False	False	False

BFS to find length of shortest path on a weighted graph

src_vertex = A
des_vertex = E

Q = A

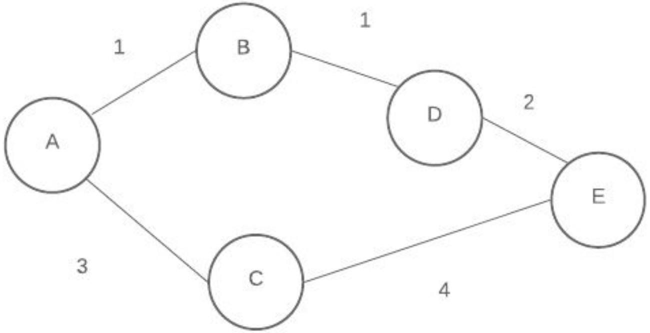


Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	True	False	False	False	False

BFS to find length of shortest path on a weighted graph

src_vertex = A
des_vertex = E

Q = A



Dequeue vertex from Q and explore its adjacent vertices

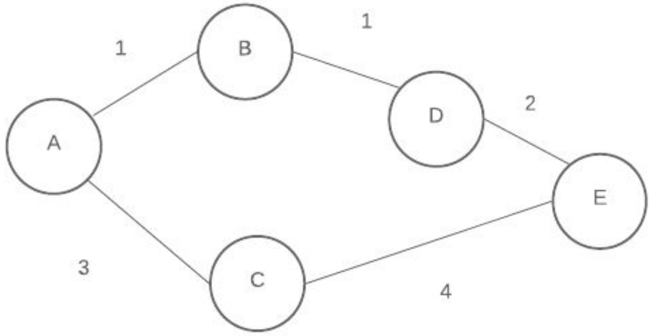
Vertex	A	B	C	D	E
Distance	0	0	0	0	0
Visited	True	False	False	False	False

BFS to find length of shortest path on a weighted graph

src_vertex = A
des_vertex = E

Q =

C	B

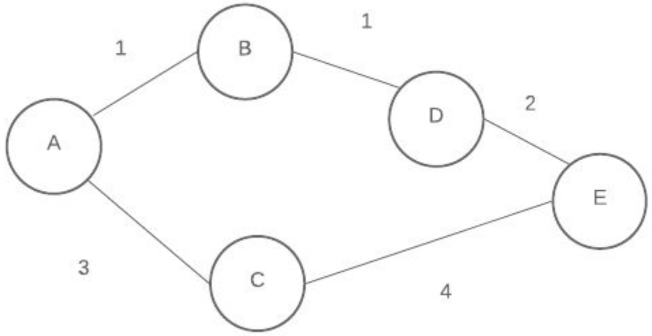


Vertex	A	B	C	D	E
Distance	0	1	3	0	0
Visited	True	True	True	False	False

BFS to find length of shortest path on a weighted graph

src_vertex = A
des_vertex = E

Q = | C | B |



Dequeue vertex from Q and explore its adjacent vertices

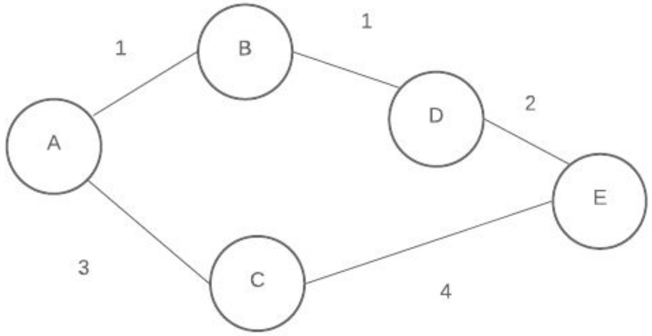
Vertex	A	B	C	D	E
Distance	0	1	3	0	0
Visited	True	True	True	False	False

BFS to find length of shortest path on a weighted graph

src_vertex = A
des_vertex = E

Q =

D	C

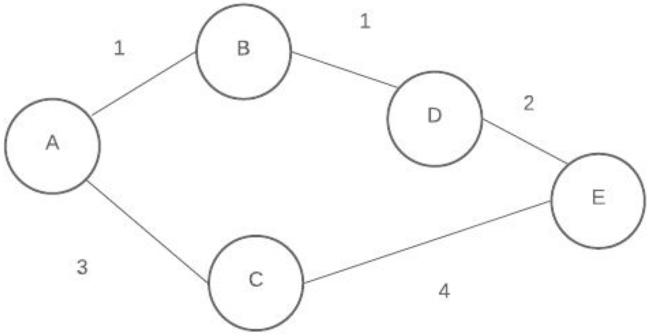


Vertex	A	B	C	D	E
Distance	0	1	3	2	0
Visited	True	True	True	True	False

BFS to find length of shortest path on a weighted graph

src_vertex = A
des_vertex = E

Q = | D | C |



Dequeue vertex from Q and explore its adjacent vertices

Vertex	A	B	C	D	E
Distance	0	1	3	2	0
Visited	True	True	True	True	False

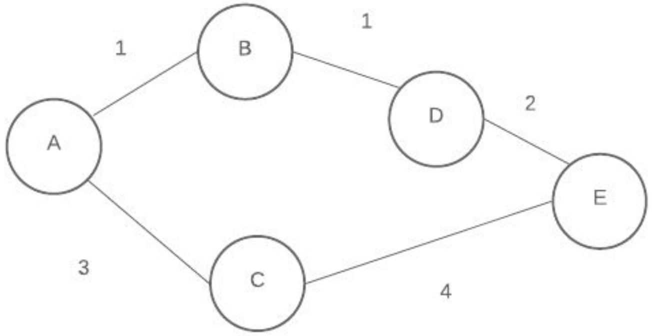
BFS to find length of shortest path on a weighted graph

src_vertex = A

des_vertex = E

Q =

E	D

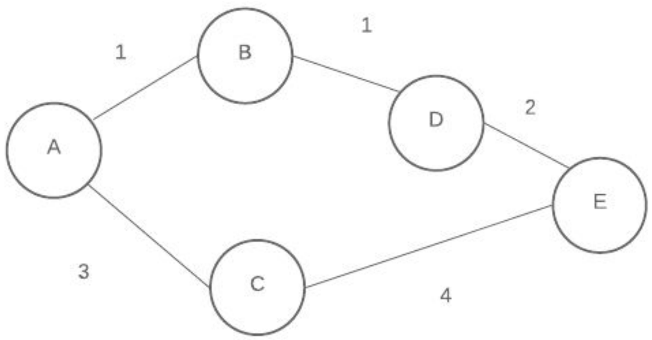


Vertex	A	B	C	D	E
Distance	0	1	3	2	7
Visited	True	True	True	True	True

BFS to find length of shortest path on a weighted graph

src_vertex = A
des_vertex = E

Q = | E | D |



According to this algorithm, path from A to E has distance 7
However, the shortest path from A to E has distance 4 (not 7)

Vertex	A	B	C	D	E
Distance	0	1	3	2	7
Visited	True	True	True	True	True

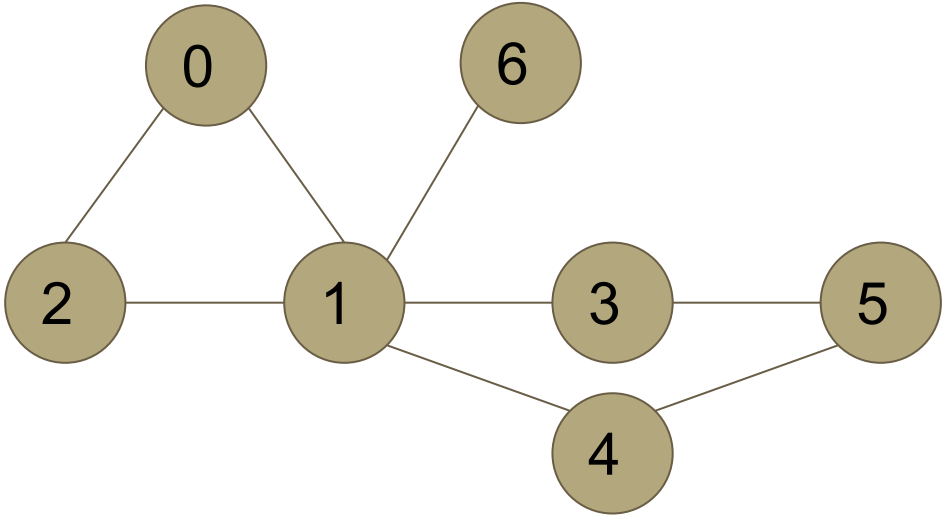
So, BFS fails when finding the shortest path in weighted graphs.

- What can we do to now?

- Dijkstra’s algorithm helps us find shortest path in weighted graphs.
  - Has it been covered in the class yet?
  - Do you guys want me to discuss this in depth or is it clear already?

Exercise:

- **Find if a given edge in a graph is a bridge or not.**



Exercise

- **What is a connected component of a graph?**

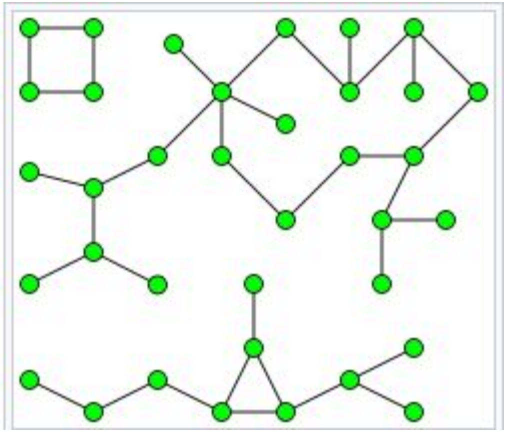
  A connected component of a graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Exercise

- **What is a connected component of a graph?**

  A connected component, of a graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

- **Question - How many connected components are there in this graph?**

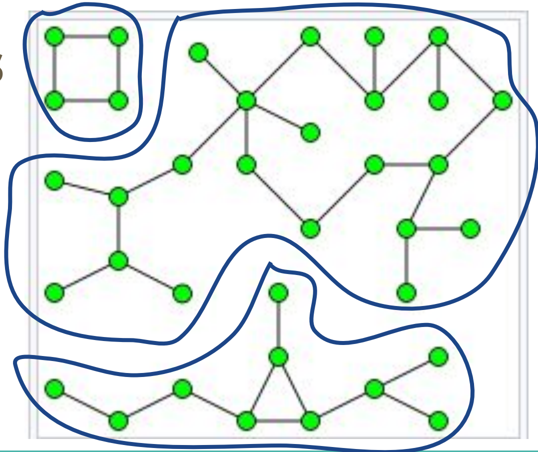


Exercise

- **What is a connected component of a graph?**

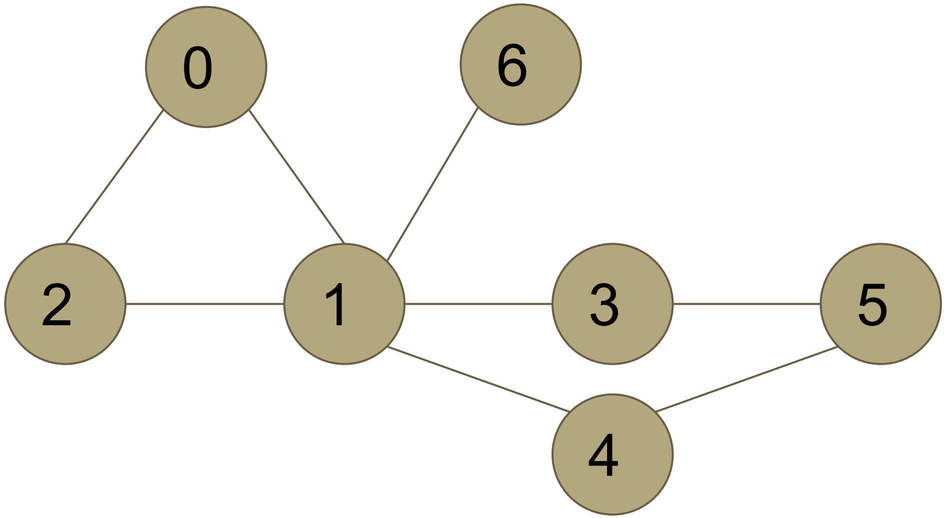
  A connected component, of a graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

- **Question - How many connected components are there in this graph?**
  - **Answer - 3**
- **It can also be visualized as different islands in an ocean.**



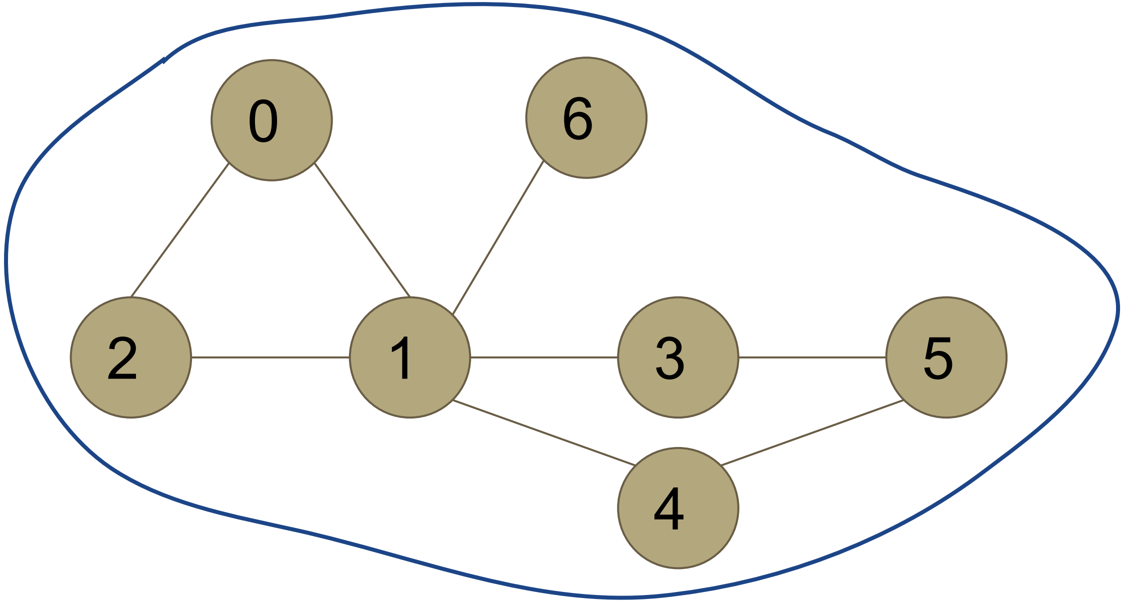
Exercise:

- **How many connected components are there in this graph?**



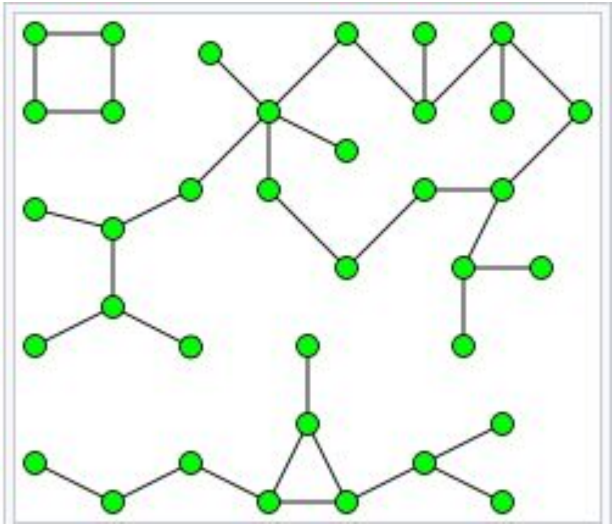
Exercise:

- **How many connected components are there in this graph?**
  - **Answer - 1**



Exercise:

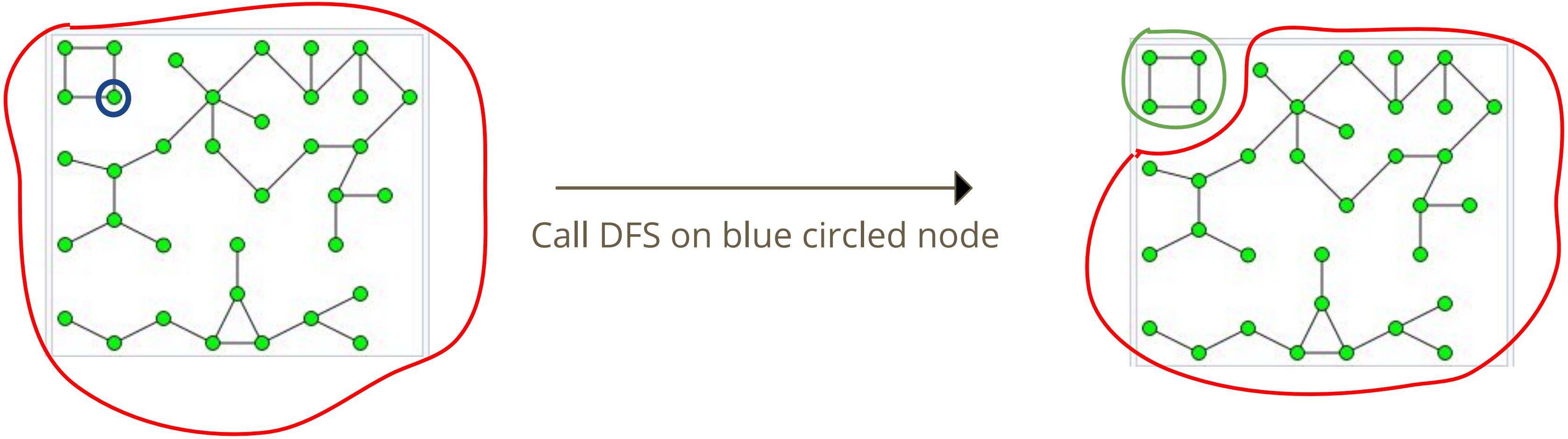
- **How to code to count the number of connected components in a graph?**
  - **Hint - Think about using DFS**



Psuedocode for counting number of connected components

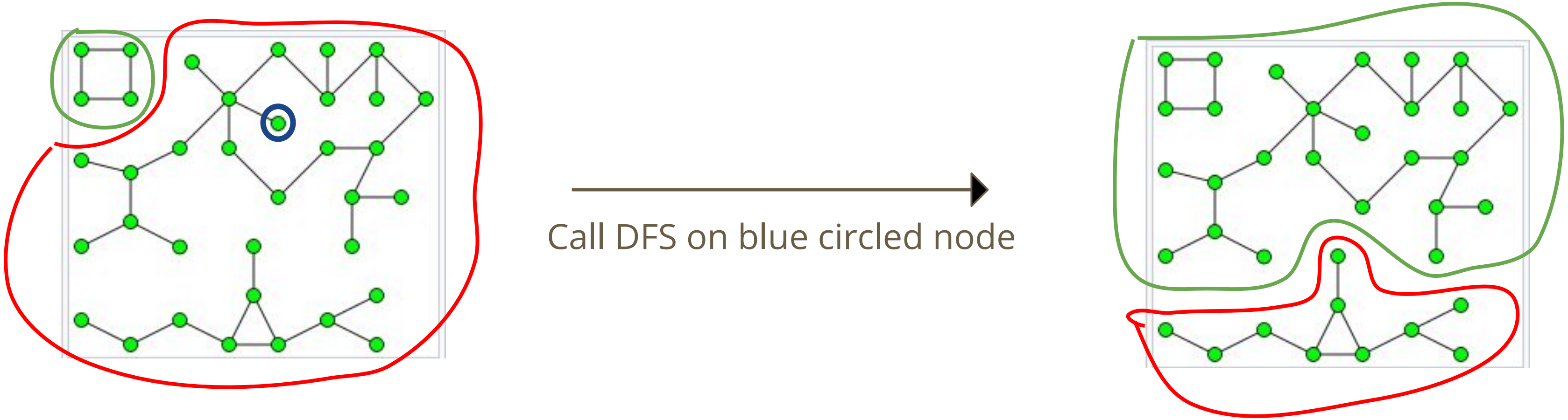
```
num_components = 0;
for vertex in list_of_vertices
{
    if (vertex has not been visited yet)
    {
        Call DFS_Traversal on that vertex
        num_components += 1
        //This is equivalent to counting how many times the
         DFS_Traversal function was called in this loop.
    }
}
```

Psuedocode for counting connected components in action



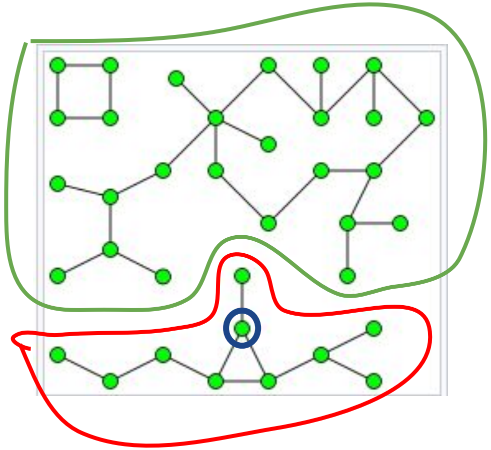
- Red circle engulfs all the vertices that haven't been visited yet.
- Green circle engulfs all the vertices that have been visited.

Psuedocode for counting connected components in action

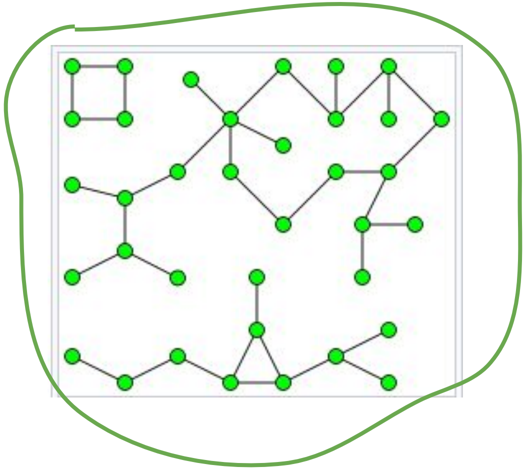


- Red circle engulfs all the vertices that haven’t been visited yet.
- Green circle engulfs all the vertices that have been visited.

Psuedocode for counting connected components in action

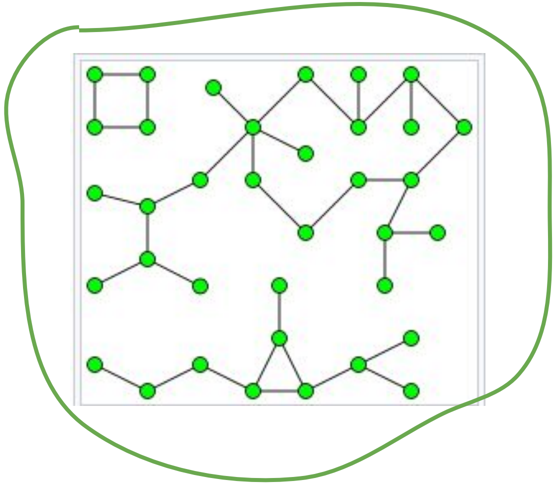


Call DFS on blue circled node



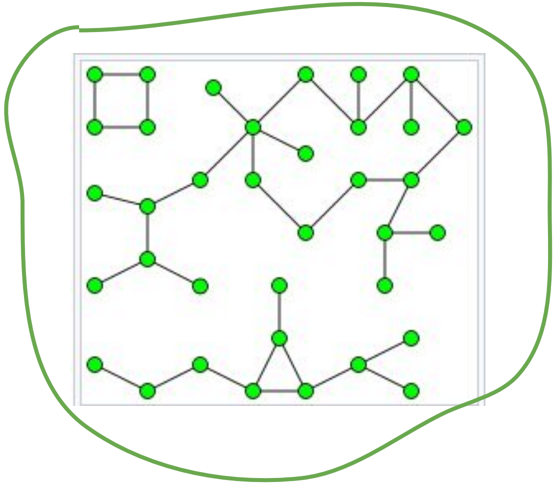
- Red circle engulfs all the vertices that haven’t been visited yet.
- Green circle engulfs all the vertices that have been visited.

Psuedocode for counting connected components in actions



- All the vertices have now been visited and I had to call my DFS traversal function three times. Therefore, the number of connected components in my graph is 3

Psuedocode for counting connected components in actions

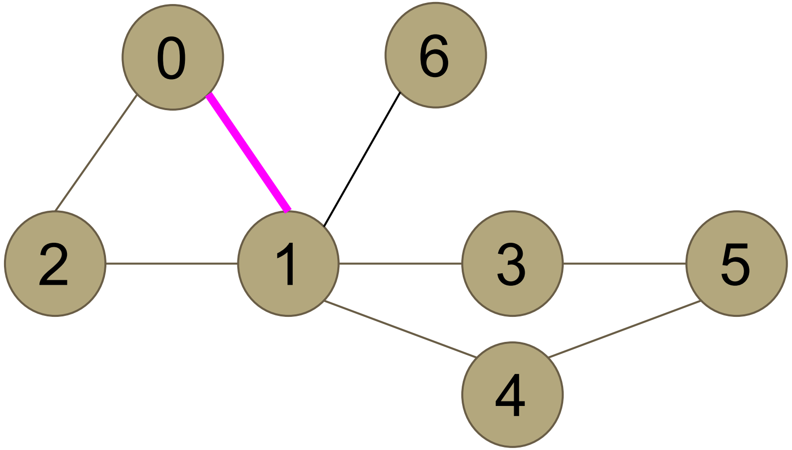


- All the vertices have now been visited and I had to call my DFS traversal function three times. Therefore, the number of connected components in my graph is 3.
- **You have to do exacty this in ASSIGNMENT 8 as well!**

Exercise

- **What does it mean to say that an edge is a bridge?**
  - The recitation write up defines that an edge of the graph is a bridge **if its deletion increases the number of connected components.**

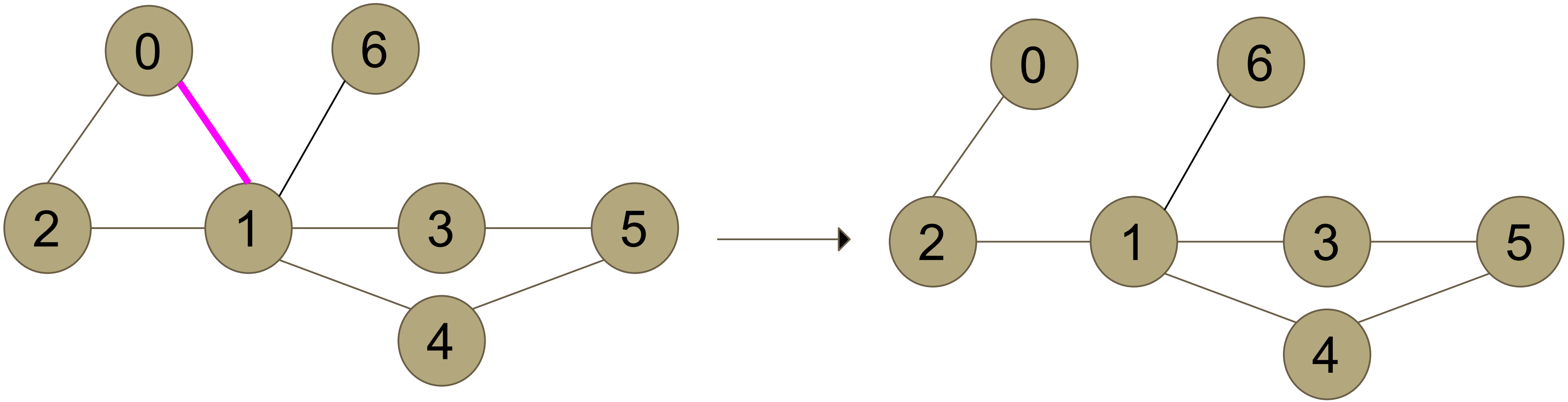
Is this edge a bridge?



- How to check if the pink colored edge is a bridge or not?
    - **Remove that edge and see if the number of connected components increased**

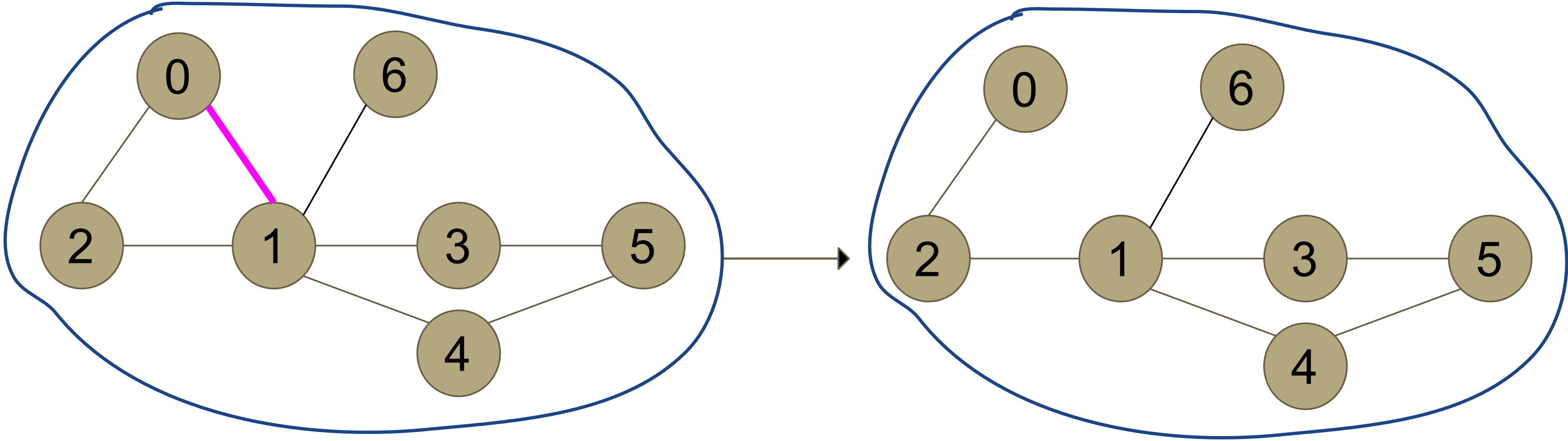
Is this edge a bridge?

Remove the edge connecting 0 and 1



Is this edge a bridge?

Count the number of connected components in the old and the new graph.

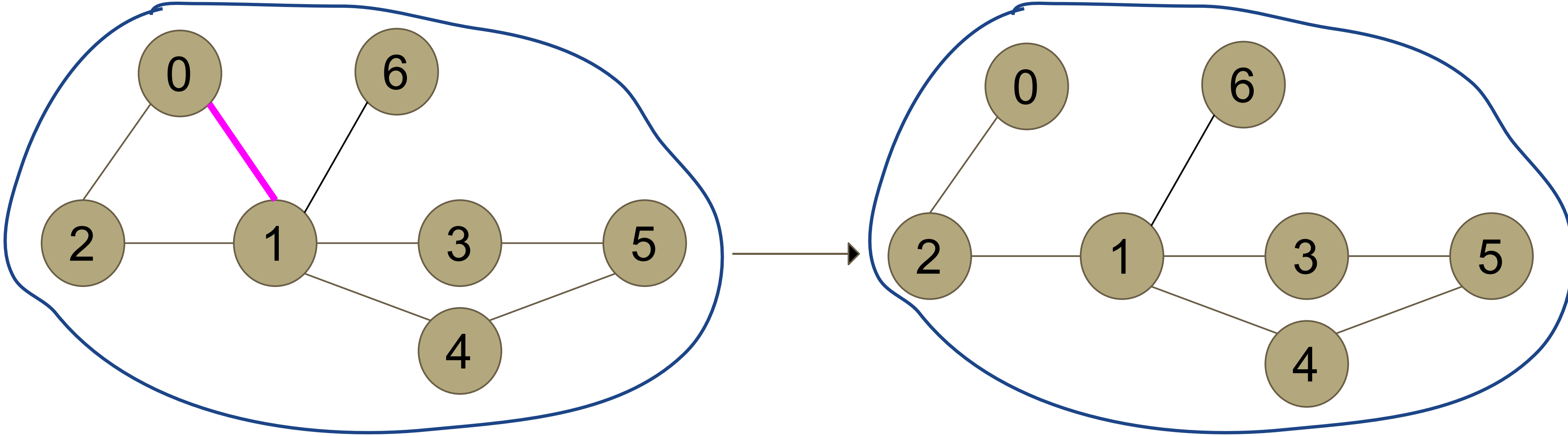


#Connected Components = 1

#Connected Components = 1

Is this edge a bridge?

Count the number of connected components in the old and the new graph.

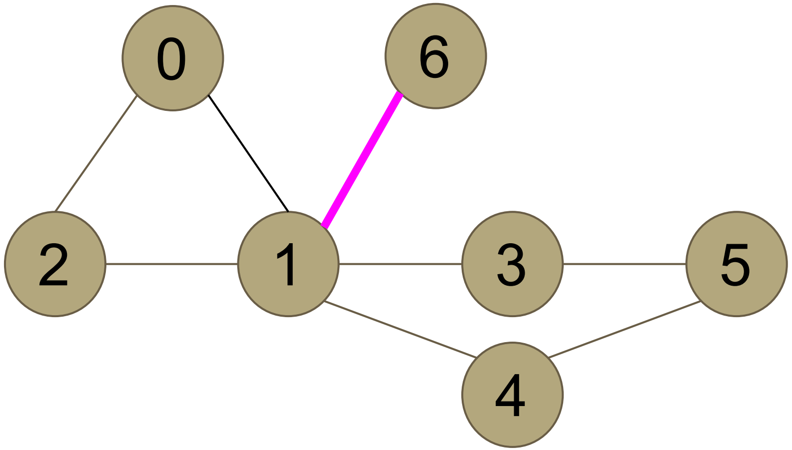


#Connected Components = 1

#Connected Components = 1

Since, the number of connected components didn't change, edge (0,1) is not a bridge.

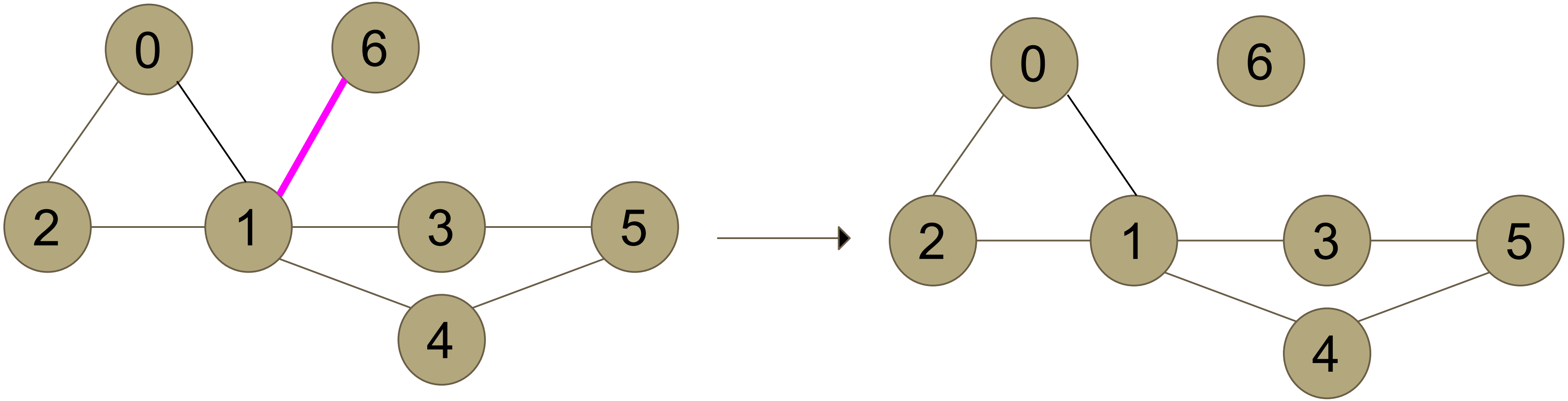
Is this edge a bridge?



- How to check if the pink colored edge is a bridge or not?
  - **Remove that edge and see if the number of connected components increased**

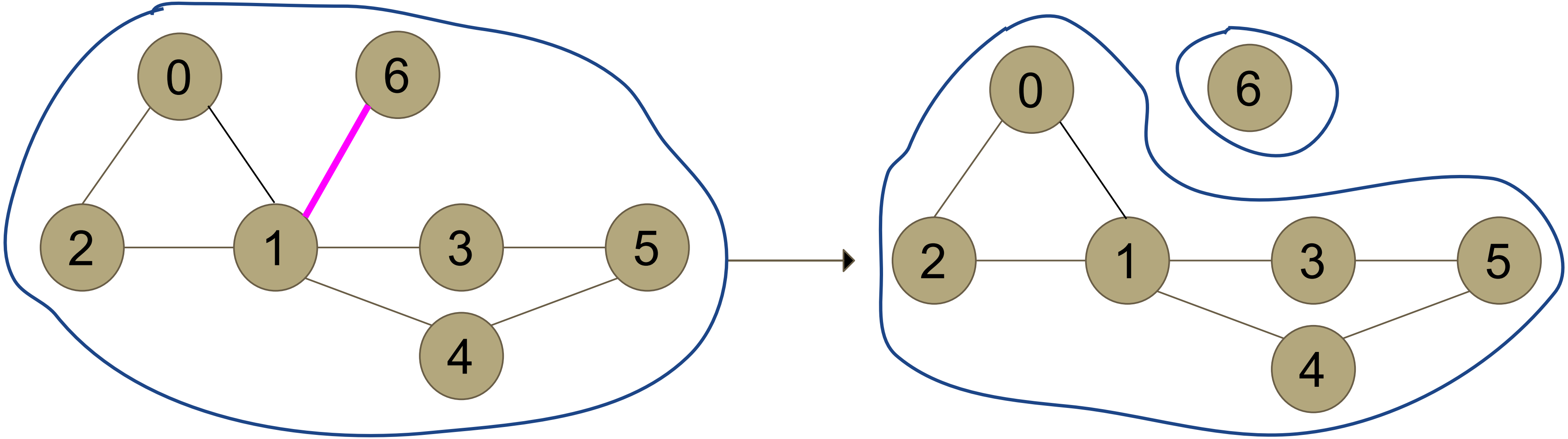
Is this edge a bridge?

Remove the edge connecting 1 and 6



Is this edge a bridge?

Count the number of connected components in the old and the new graph.

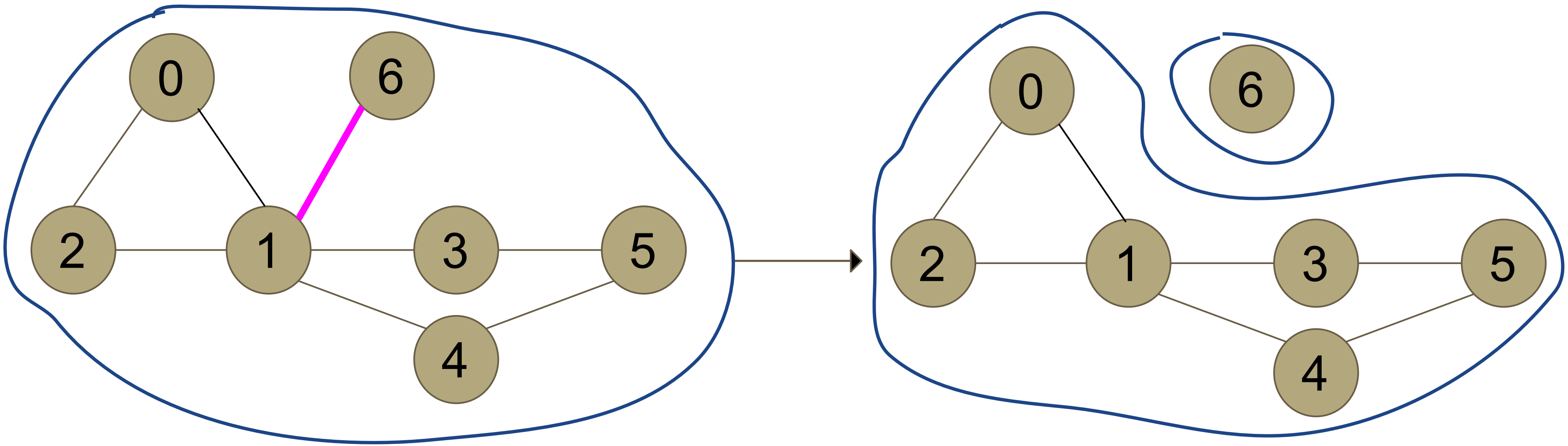


#Connected Components = 1

#Connected Components = 2

Is this edge a bridge?

Count the number of connected components in the old and the new graph.



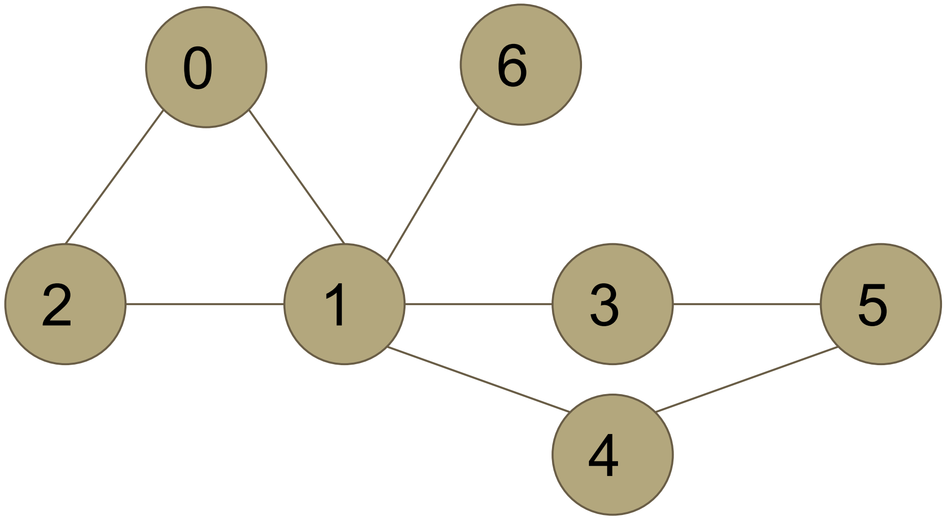
#Connected Components = 1

#Connected Components = 2

Since, the number of connected components increased, edge (1,6) is a bridge.

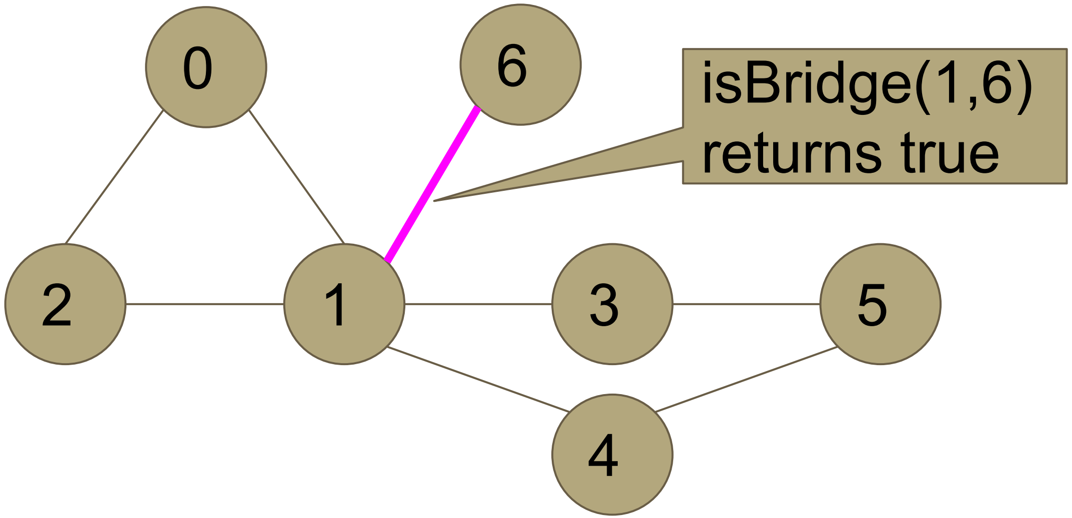
Exercise:

- The boolean function **isBridge(int key1, int key2)** returns true if the edge between vertices with key1 and key2 is a "bridge".



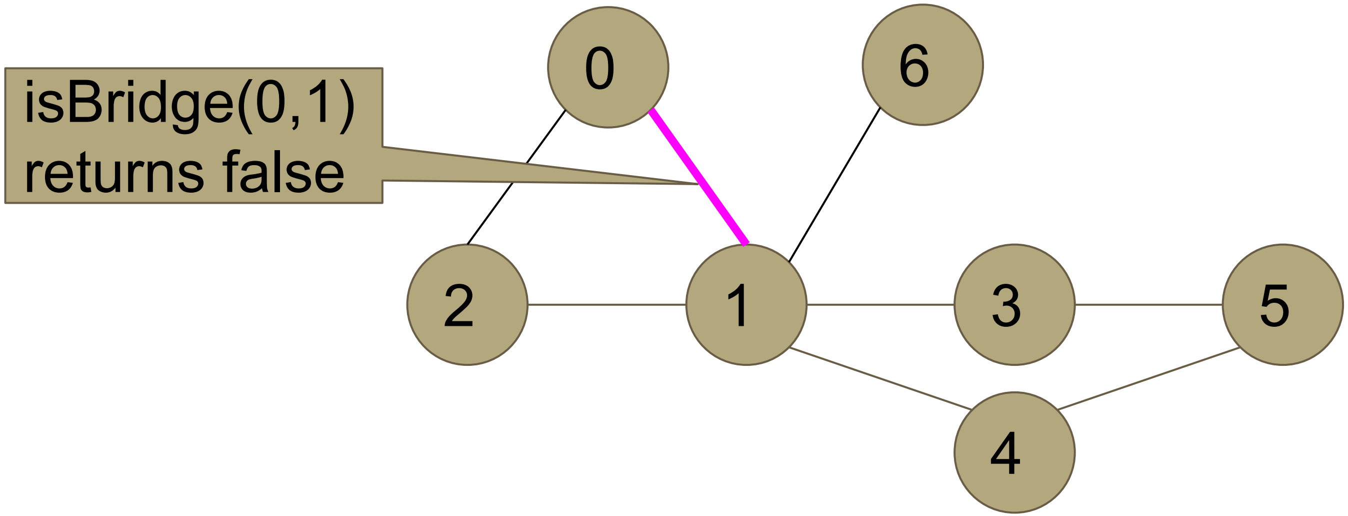
Exercise:

- The boolean function isBridge(int key1, int key2) returns true if the edge between vertices with key1 and key2 is a “bridge”.



Exercise

- The boolean function isBridge(int key1, int key2) returns true if the edge between vertices with key1 and key2 is a “bridge”.



Exercise

- **TODO:**

  Complete the following functions in Graph.cpp

  - void Graph::DFTraversal(vertex *n)
  - void Graph::removeEdge(int key1, int key2)
  - bool Graph::isBridge(int key1, int key2)

DFTraversal(vertex *n) function - Overview

- Traverse the graph and mark all the vertices in the graph “visited” that are reachable from n via some path.

- Psuedocode (recursive function)

```
DFS( vertex n)
{
        Mark vertex n as visited
        for vertex i in adjacency list of n
        {
                if i has not been visited
                {
                        DFS( vertex i)
                }
        }
}
```

removeEdge(int key1, int key2) function - Overview

- Modify the graph to remove the edge between vertex with value key1 and vertex with value key2.

- Psuedocode

```
removeEdge( int key1, int key2)
{
        Identify Vertex V1 with value key1
        Identify Vertex V2 with value key2
        //Removing that edge from the graph
        Remove V2 from adjacency list of V1
        Remove V1 from adjacency list of V2
}
```

isBridge(int key1, int key2) function: Overview

- Step 1: Find number of connected components in the given graph (using the approach we discussed).
- Step 2: Remove the given edge from the graph.
- Step 3: Find number of connected components in the modified graph (using the approach we discussed).
- Step 4: Check if the number of connected components in the modified graph is more than the O.G graph. If it did, then return TRUE, else FALSE.

Expected Output

Removing edge (1, 6)

```
Graph before removing the edge:
0 --> 1 2
1 --> 0 2 3 4 6
2 --> 1 0
3 --> 1 5
4 --> 1 5
5 --> 3 4
6 --> 1
no. of connected components before removal: 1
Graph after removing the edge:
0 --> 1 2
1 --> 0 2 3 4
2 --> 1 0
3 --> 1 5
4 --> 1 5
5 --> 3 4
6 -->
no. of connected components after removal: 2
The edge connecting vertices with keys 1 and 6 is a bridge!
```

Removing edge (0, 2)

```
Graph before removing the edge:
0 --> 1 2
1 --> 0 2 3 4 6
2 --> 1 0
3 --> 1 5
4 --> 1 5
5 --> 3 4
6 --> 1
no. of connected components before removal: 1
Graph after removing the edge:
0 --> 1
1 --> 0 2 3 4 6
2 --> 1
3 --> 1 5
4 --> 1 5
5 --> 3 4
6 --> 1
no. of connected components after removal: 1
The edge connecting vertices with keys 0 and 2 is not a bridge!
```

START CODING

Laughing at corona memes like