DEPARTMENT OF
COMPUTER SYSTEMS ENGINEERING
MEHRAN UNIVERSITY OF ENGINEERING AND TECHNOLOGY, JAMSHORO

ARTIFICIAL INTELLIGENCE

PRACTICAL 03

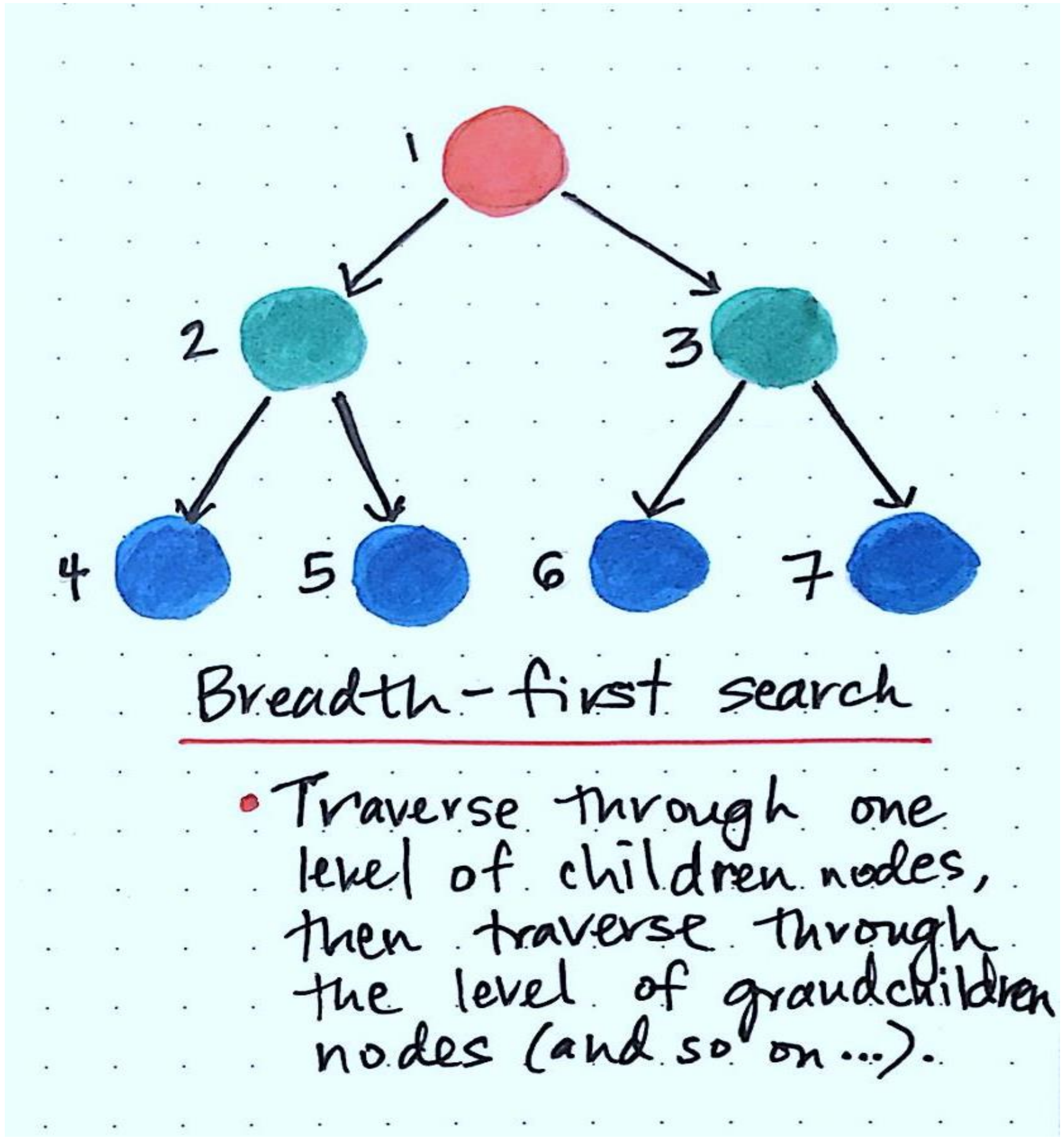
OBJECT: GRAPH TRAVERSAL: BREADTH FIRST SEARCH

Outline

- Graph traversal
- Application of BFS
- Introduction to BFS
- Algorithm of BFS
- Understanding BFS
- Implementing BFS in python
- Shortest path finding
- Find all paths between two nodes

Required Tools

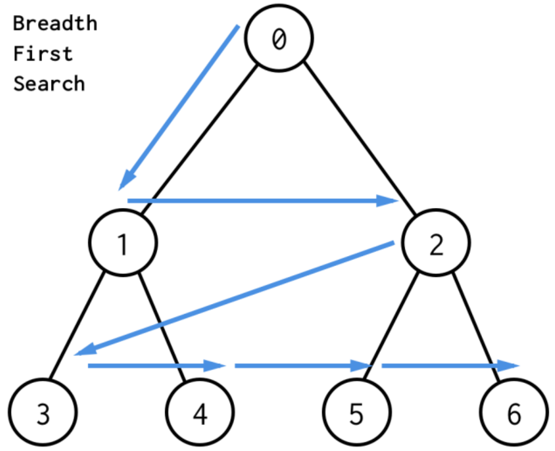
- PC with windows
- Anaconda environment

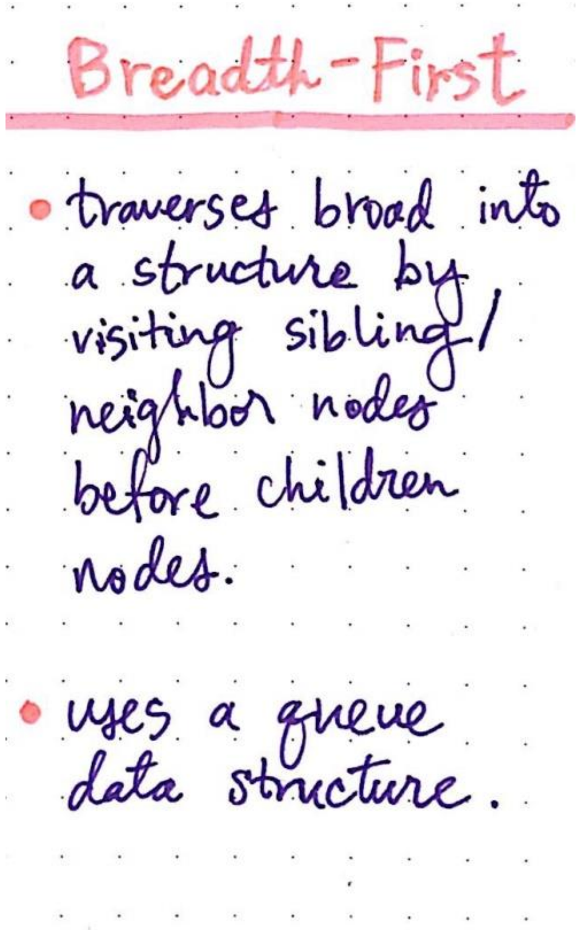

1
2
3
4
5
6
7
Breadth-first search
• Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

GRAPH TRAVERSAL

- It is also known as “graph search”
- It is process of visiting each vertex & edge in a graph.
- While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once.
- The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.
- During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

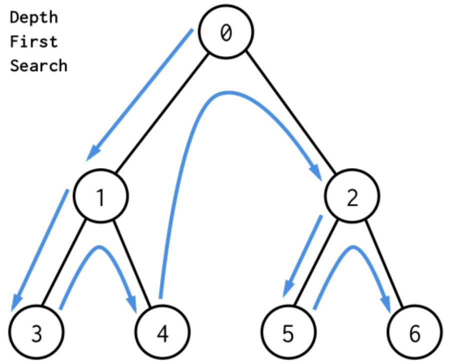
WAYS TO TRAVERSE A GRAPH

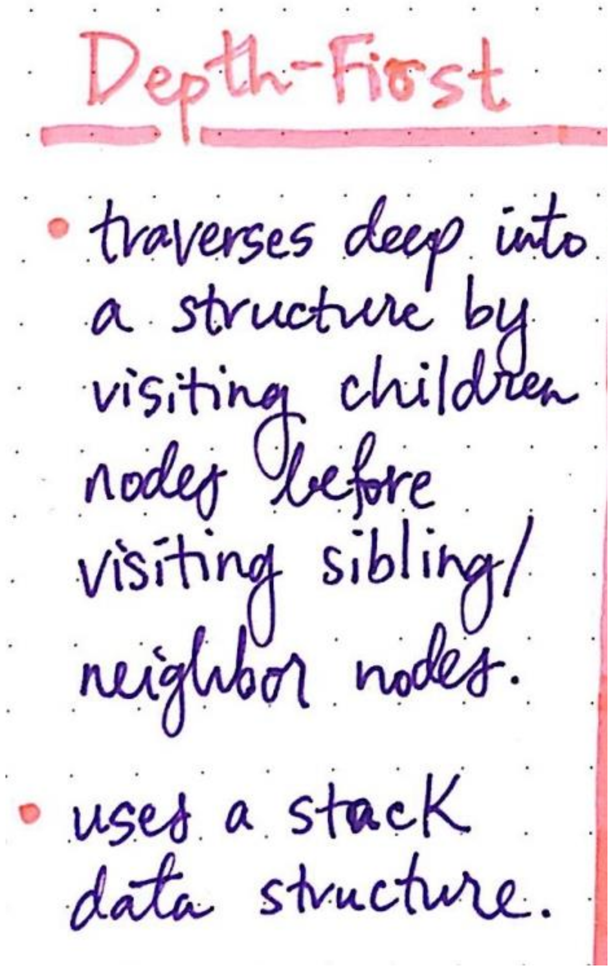




Breadth-First

- traverses broad into a structure by visiting sibling/neighbor nodes before children nodes.
- uses a queue data structure.

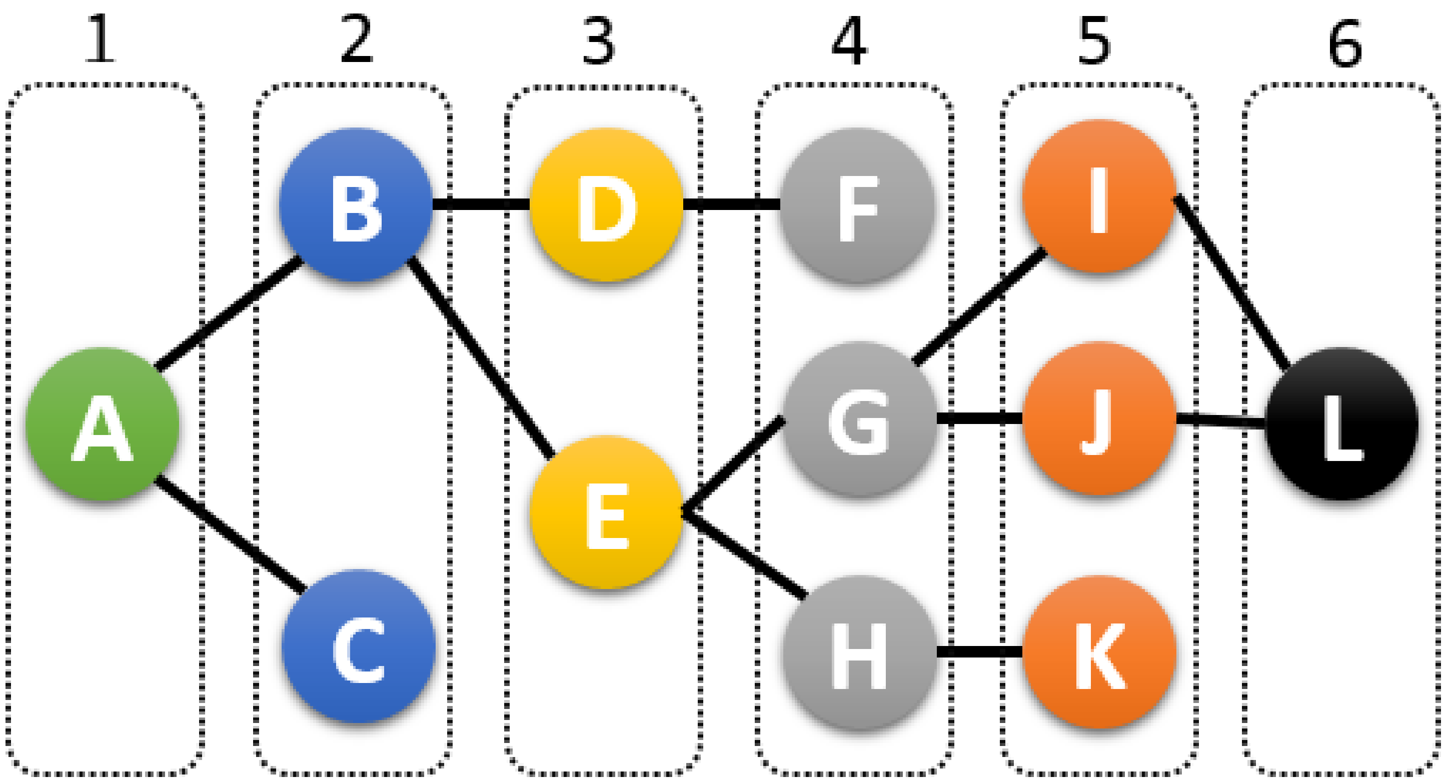




Depth-First

- traverses deep into a structure by visiting children nodes before visiting sibling/neighbor nodes.
- uses a stack data structure.

WE'LL BE LEARNING BFS IN THIS LAB..



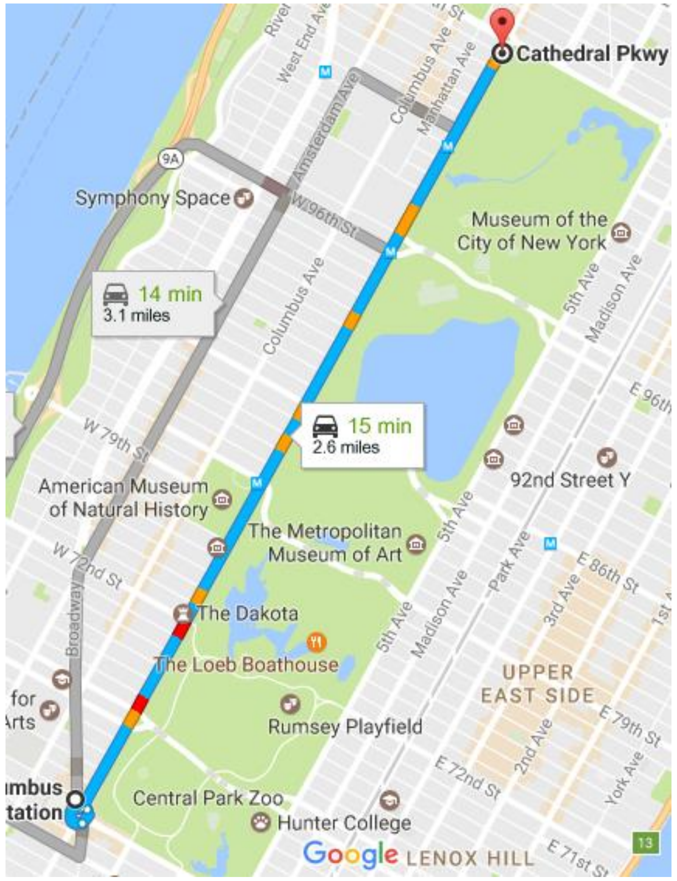
Breadth-First Search Levels

REAL LIFE APPLICATION OF BFS

Breadth First Search is basically used to find a **shortest path** between any two nodes in a graph. Since, a graph can be used to represent a large number of real life problems such as

GPS Navigation systems:

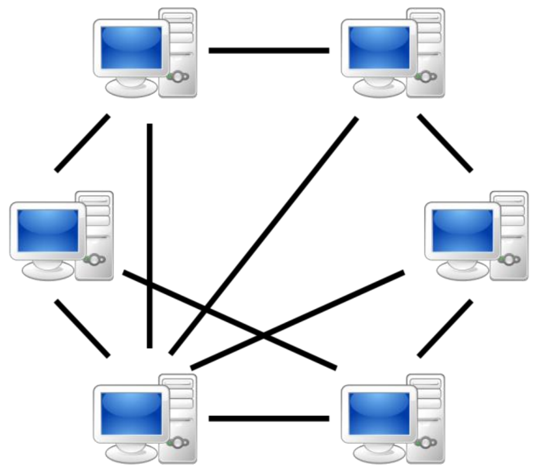
- Navigation systems such as the Google Maps, use BFS.
- They take your location to be the source node and your destination as the destination node on the graph.
- (A *city can be represented as a graph by taking* ***landmarks as nodes*** *and the* ***roads as the edges*** *that connect the nodes in the graph.)*
- BFS is applied and the shortest route is generated which is used to give directions or real time navigation.



REAL LIFE APPLICATION OF BFS

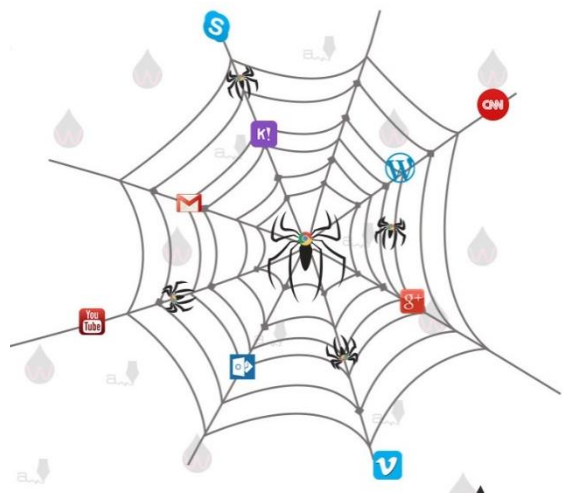
Computer Networks:

- Peer to peer (P2P) applications such as the torrent clients need to locate a file that the client *(one who wants to download the file)* is requesting.
- This is achieved by applying BFS on the hosts *(one who supplies the file) on a network.*
- Your computer is the host and it keeps traversing through the network to find a host for the required file *(maybe your favorite movie)*.



Web Crawlers

- A web crawler (also known as a web spider or web robot) is a program or automated script which browses the World Wide Web in a methodical, automated manner.
- They can be used to analyze what all sites you can reach by following links randomly on a particular website.



REAL LIFE APPLICATION OF BFS

Facebook:

- It treats each user profile as a node on the graph and two nodes are said to be connected if they are each other's friends.

- Infact, apply BFS on the facebook graph and you will find that **any two people** are connected with each other by **atmost** five nodes in between.

- To say, that you can reach any random person in the world by traversing **5 nodes** in between. What do you think is the new facebook **"Graph Search"**? *(It is not directly BFS, but a lot of modifications over classic graph search algorithms.)*

BFS (BREADTH FIRST SEARCH)

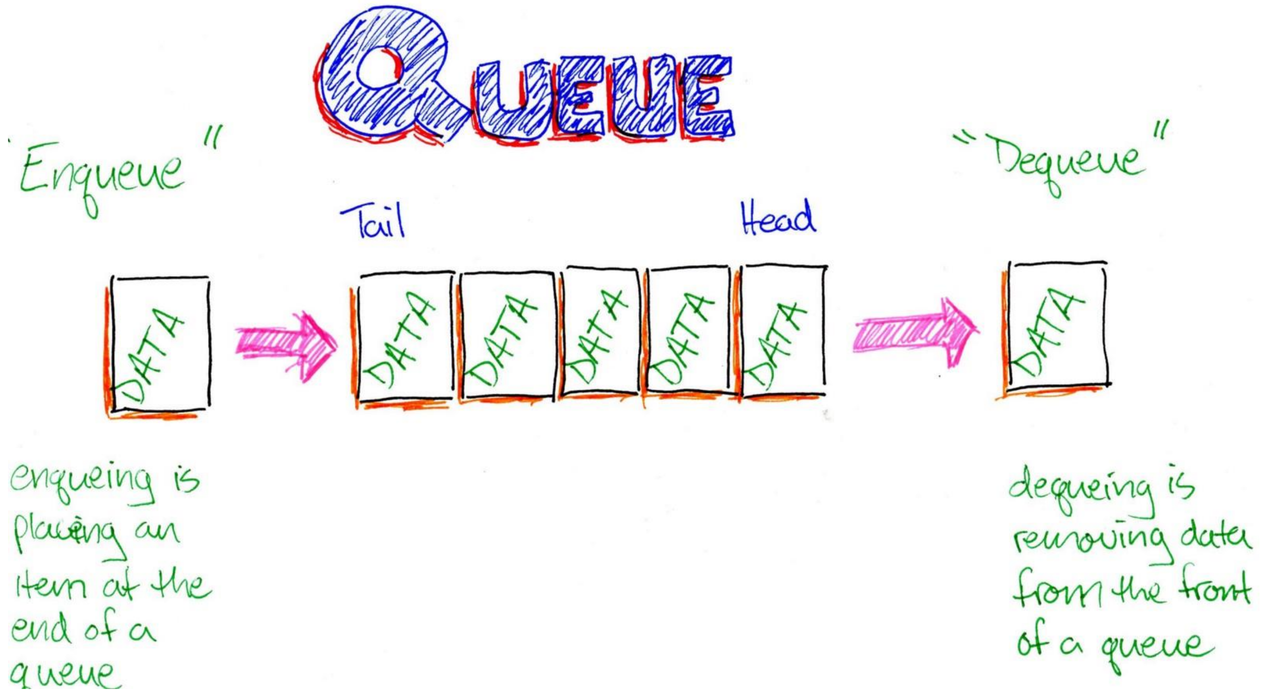
- BFS is an AI search algorithm that can be used for finding solutions to a problem.
- BFS implements a specific strategy for visiting all the nodes (vertices) of a graph.
- BFS starts with a node, then it checks the neighbours of the initial node, then the neighbours of the neighbours, and so on.

- Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a search key) and explores the neighbor nodes first, before moving to the next level neighbors.

- If there is a solution, breadth-first search is guaranteed to find it, and if there are several solutions, breadth-first search will always find the shallowest goal state first.

BFS (BREADTH FIRST SEARCH)

- BFS visits all the nodes of a graph (connected component) following a breadth ward motion. In other words, BFS starts from a node, then it checks all the nodes at distance one from the starting node, then it checks all the nodes at distance two and so on. In order to remember the nodes to be visited, *BFS uses a queue (data structure).*


QUEUE
"Enqueue"
Tail
Head
"Dequeue"
DATA
enqueing is placing an item at the end of a queue
dequeing is removing data from the front of a queue

First In, First Out
FIFO

BFS(BREADTH FIRST SEARCH) ALGORITHM

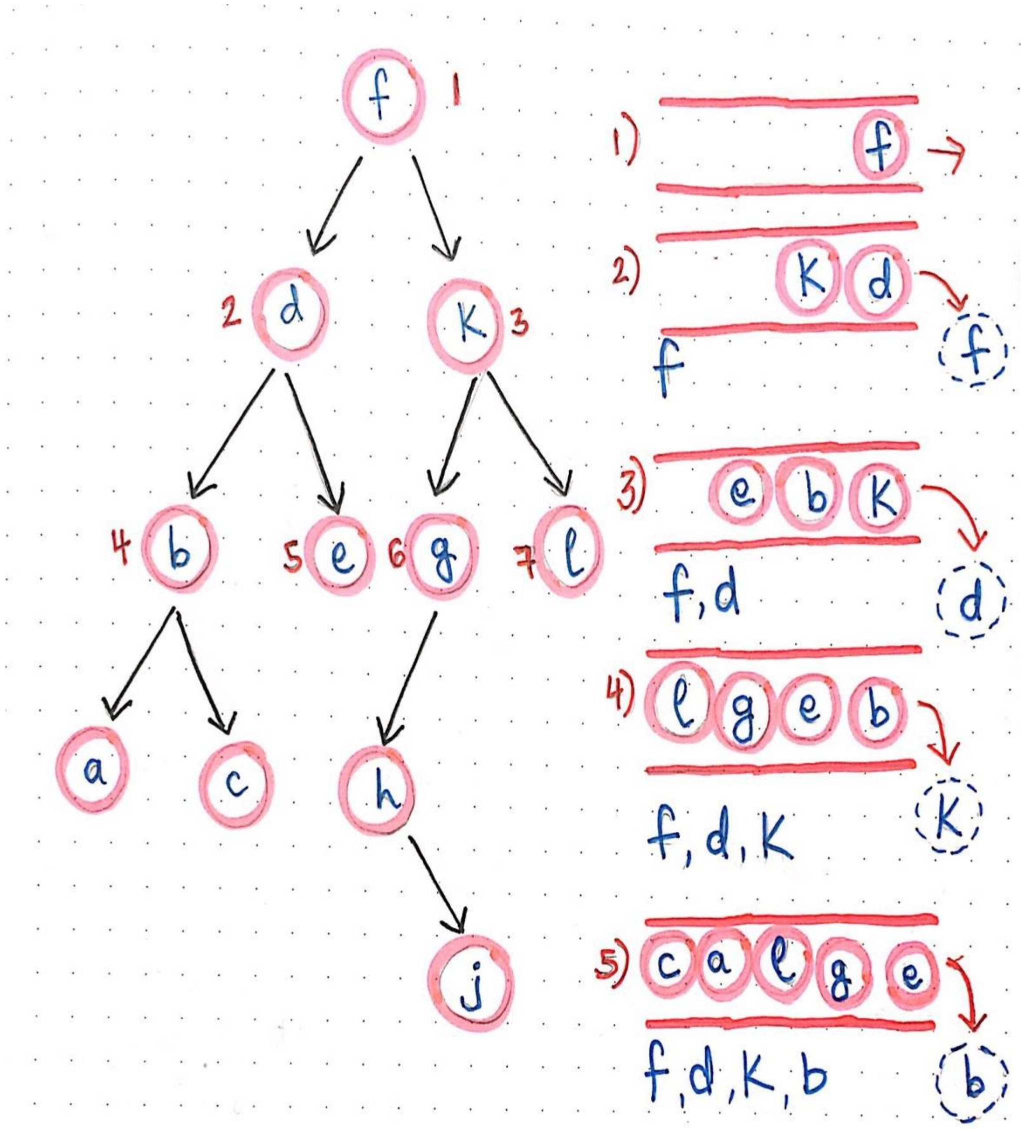
Graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. It uses a queue to mark the visited nodes so that they should not be repeated again.

Rule 1: Choose some starting vertex (node) and insert it in queue.

Rule 2: Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.

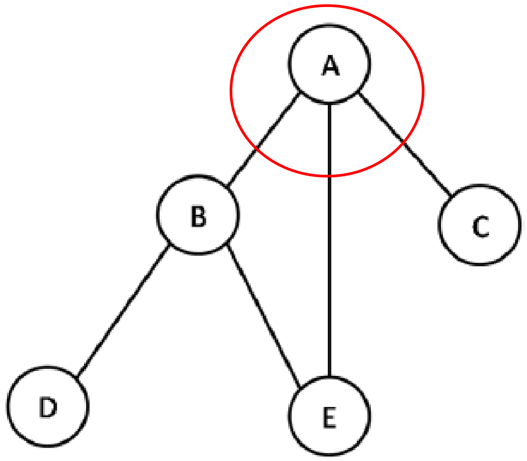
Rule 3: If no adjacent vertex found, remove the first vertex from queue.

Rule 4: Repeat Rule 2 and Rule 3 until queue is empty.



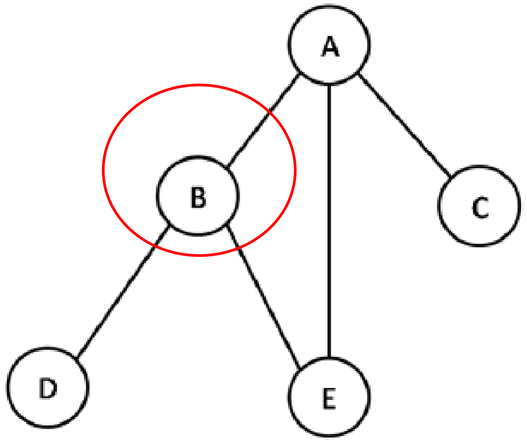
EXAMPLE OF BFS

- Consider the following undirected graph:



Step 01:
Start with node A. Mark it as visited and display it in path.

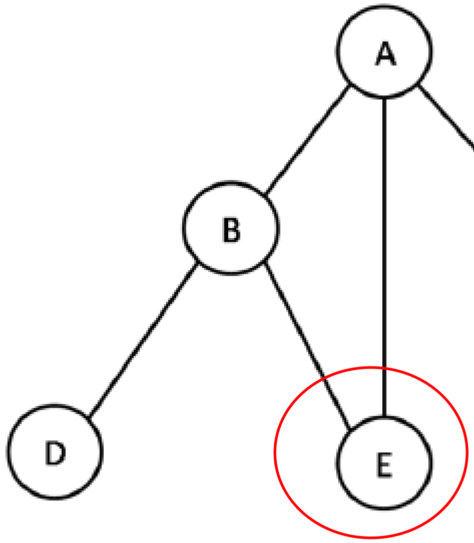
Queue							
Visited	A						
Path	A						



Step 02:
Find unvisited adjacent node of A mark it as isited, place it in queue and display it in path.

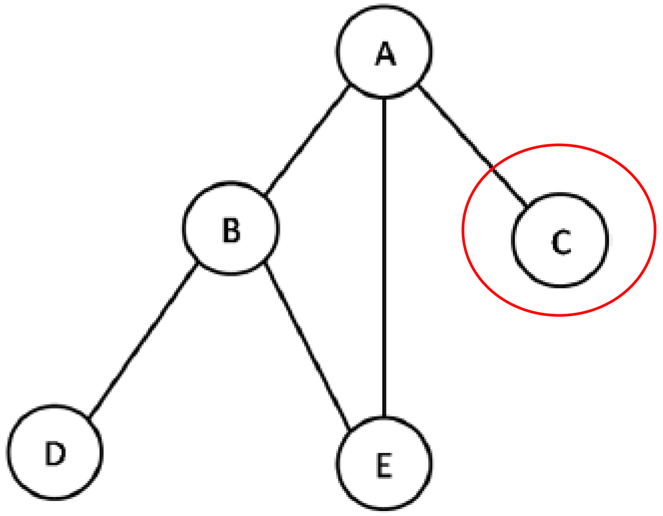
Queue	B						
Visited	A	B					
Path	A	B					

EXAMPLE OF BFS



Step 03:
Next unvisited adjacent nodes from A is E;
mark it as visited, place it in queue and display
it in path.

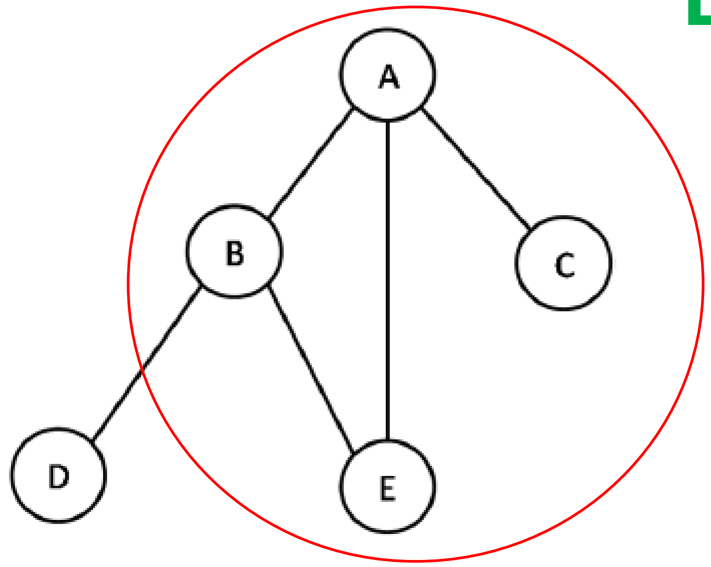
Queue	E	B					
Visited	A	B	E				
Path	A	B	E				



Step 04:
Next unvisited adjacent nodes from A is C;
mark it as visited, place it in queue and display
it in path.

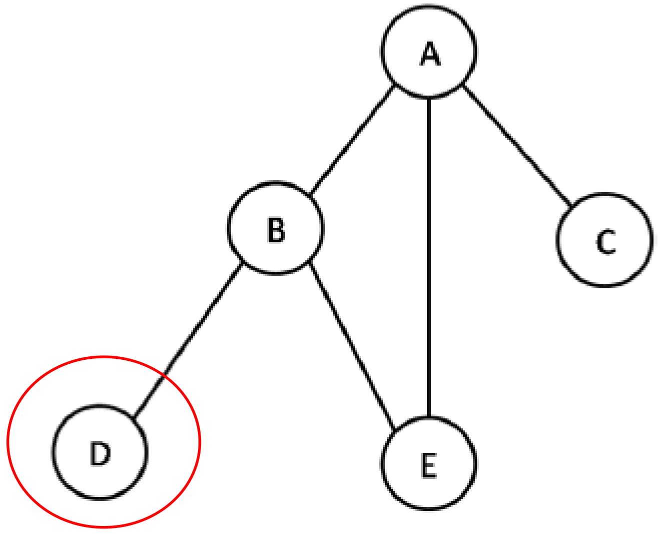
Queue	C	E	B				
Visited	A	B	E	C			
Path	A	B	E	C			

EXAMPLE OF BFS



Step 05:
There are no further unvisited adjacent nodes from A; dequeue node from queue (B).

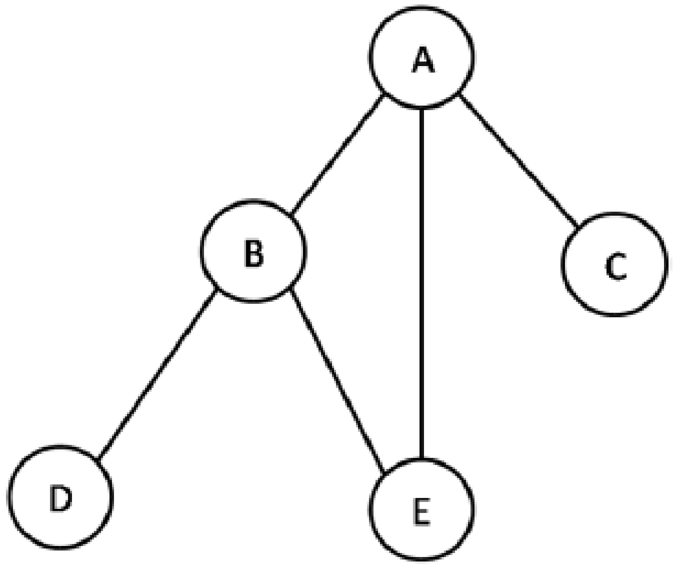
Queue	C	E					
Visited	A	B	E	C			
Path	A	B	E	C			



Step 06:
Find any unvisited adjacent node of B; mark it as visited, place it in queue and display it in path.

Queue	D	C	E				
Visited	A	B	E	C	D		
Path	A	B	E	C	D		

EXAMPLE OF BFS



Step 07:
There are no further unvisited adjacent nodes from B; dequeue node from queue (E); find any unvisited adjacent nodes;

Queue	D	C					
Visited	A	B	E	C	D		
Path	A	B	E	C	D		

Step 08:
There are no further unvisited adjacent nodes from E; dequeue node from queue (C); find any unvisited adjacent nodes;

Queue	D						
Visited	A	B	E	C	D		
Path	A	B	E	C	D		

Step 09:
There are no further unvisited adjacent nodes from C; dequeue node from queue (D); find any unvisited adjacent nodes;

Queue							
Visited	A	B	E	C	D		
Path	A	B	E	C	D		

As the queue is empty all the nodes have been visited, the algorithms stops here. The traversal path suing breadth first search is:

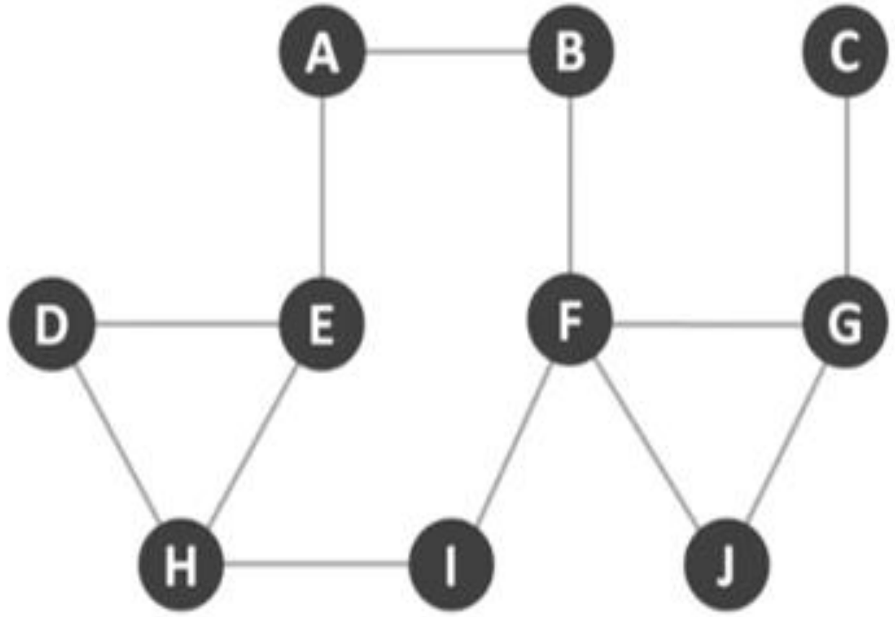
A → B → E → C → D

BFS (BREADTH FIRST SEARCH) TRAVERSE IN PYTHON

STEP 01

An effective/elegant method for implementing adjacency lists in Python is using dictionaries. The keys of the dictionary represent nodes, the values have a list of neighbors.

```
graph =   {
          'A': ['B','E'],
          'B': ['F'],
          'C': ['G'],
          'D': ['E', 'H'],
          'E': ['A','D','H'],
          'F': ['B','G', 'I', 'J'],
          'G': ['C','F','J'],
          'H': ['D','E','I'],
          'I': ['F','H'],
          'J': ['F','G']
          }
```



GRAPH A

BFS (BREADTH FIRST SEARCH) TRAVERSE IN PYTHON

STEP 02

Traversing a graph: Visiting all the nodes of a connected component with BFS, is as simple as implementing the steps of the algorithm.

TASK 1:
Generate output for Graph A using this code

```
# visits all the nodes of a graph (connected component) using BFS
def bfs_connected_component(graph, start):
    # keep track of all visited nodes
    explored = []
    # keep track of nodes to be checked
    queue = [start]

    # keep looping until there are nodes still to be checked
    while queue:
        # pop shallowest node (first node) from queue
        node = queue.pop(0)
        if node not in explored:
            # add node to list of checked nodes
            explored.append(node)
            neighbours = graph[node]

            # add neighbours of node to queue
            for neighbour in neighbours:
                queue.append(neighbour)
    return explored
```

Call the function:

```
print(bfs_connected_component(graph,'A'))
```

BFS SHORTEST PATH IN PYTHON

For this task, the function we implement should be able to accept as argument a graph, a starting node (e.g., ‘G’) and a node goal (e.g., ‘D’). If the algorithm is able to connect the start and the goal nodes, it has to return the path.

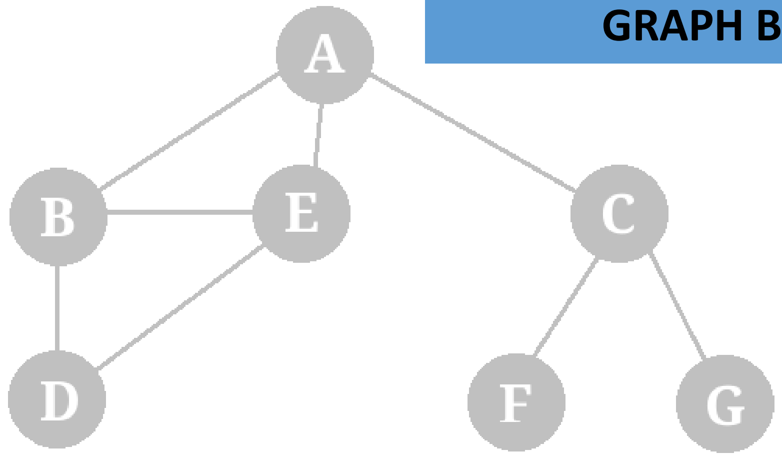
The nice thing about **BFS** is that it a**lways returns the shortest path, even if there is more than one path that links two vertices.**

There are a couple of main differences between the implementations of BFS for traversing a graph and for finding the shortest path. First, in case of the shortest path application, we need for the queue to keep track of possible paths (implemented as list of nodes) instead of nodes. Second, when the algorithm checks for a neighbour node, it needs to check whether the neighbour node corresponds to the goal node. If that’s the case, we have a solution and there’s no need to keep exploring the graph.

BFS SHORTEST PATH IN PYTHON

For this task, the function we implement should be able to accept as argument a graph, a starting node (e.g., ‘G’) and a node goal (e.g., ‘D’). If the algorithm is able to connect the start and the goal nodes, it has to return the path. The nice thing about **BFS** is that it a**lways returns the shortest path, even if there is more than one path that links two vertices.**

GRAPH B



```
# sample graph implemented as a
dictionary
graph = {
            'A': ['B', 'C', 'E'],
            'B': ['A','D', 'E'],
            'C': ['A', 'F', 'G'],
            'D': ['B‘, ‘E’],
            'E': ['A', 'B','D'],
            'F': ['C'],
            'G': ['C']
      }
```

BFS SHORTEST PATH IN PYTHON-METHOD 1

```
# finds shortest path between 2 nodes of a
graph using BFS
def bfs_shortest_path(graph, start, goal):
    # keep track of explored nodes
    explored = []
    # keep track of all the paths to be checked
    queue = [[start]]

    # return path if start is goal
    if start == goal:
        return "That was easy! Start = goal“
```

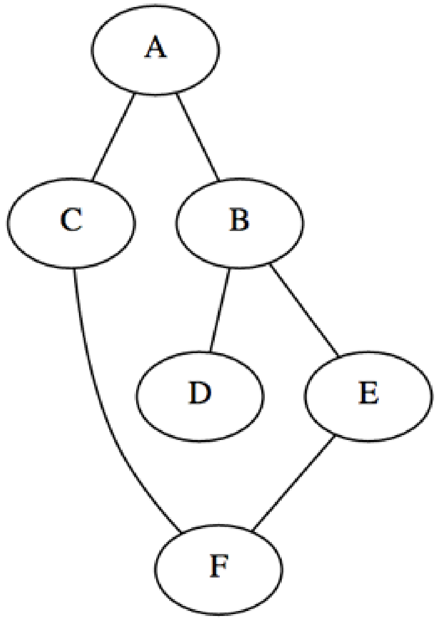
TASK 2:
Find The shortest path FOR GRAPH A & B , between two particular nodes.

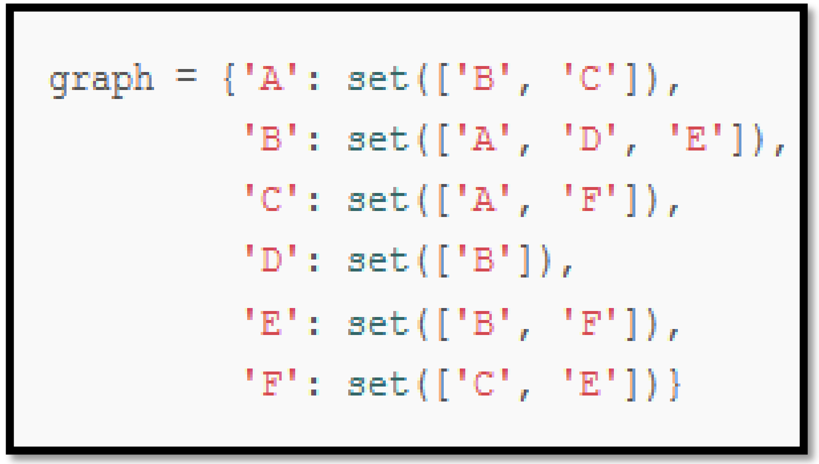
```
# keeps looping until all possible paths have been checked
while queue:
    # pop the first path from the queue
    path = queue.pop(0)
    # get the last node from the path
    node = path[-1]
    if node not in explored:
        neighbours = graph[node]
        # go through all neighbour nodes, construct a new path and
        # push it into the queue
        for neighbour in neighbours:
            new_path = list(path)
            new_path.append(neighbour)
            queue.append(new_path)
            # return path if neighbour is goal
            if neighbour == goal:
                return new_path

        # mark node as explored
        explored.append(node)

# in case there's no path between the 2 nodes
return "So sorry, but a connecting path doesn't exist :("
```

```
bfs_shortest_path(graph, 'G', 'D')
```

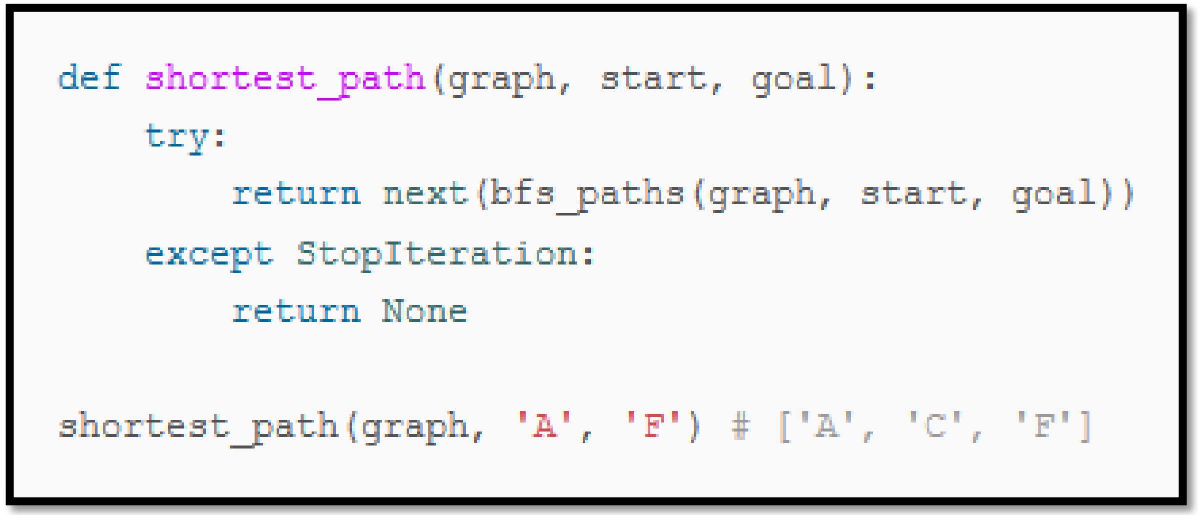
SHORTEST PATH- 2nd method

Knowing that the shortest path will be returned first from the BFS path generator method we can create a useful method which simply returns the shortest path found or ‘None’ if no path exists. As we are using a generator this in theory should provide similar performance results as just breaking out and returning the first matching path in the BFS implementation.



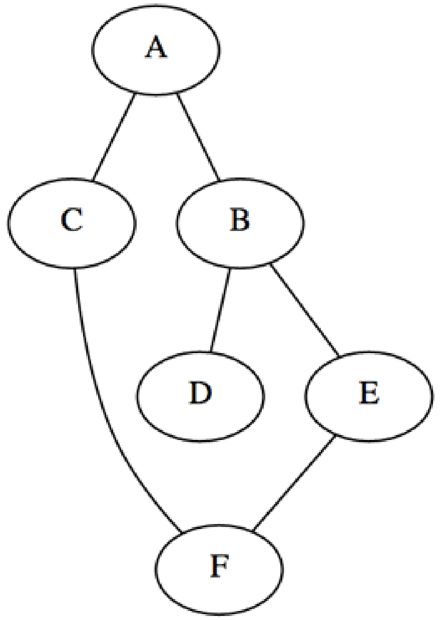


```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```



```
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None

shortest_path(graph, 'A', 'F') # ['A', 'C', 'F']
```

BFS ALL PATH



Paths

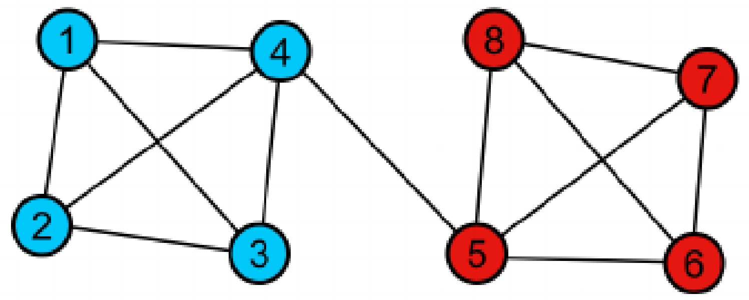
This implementation can again be altered slightly to instead return all possible paths between two vertices, the first of which being one of the shortest such path.

```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

```
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                queue.append((next, path + [next]))

list(bfs_paths(graph, 'A', 'F')) # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

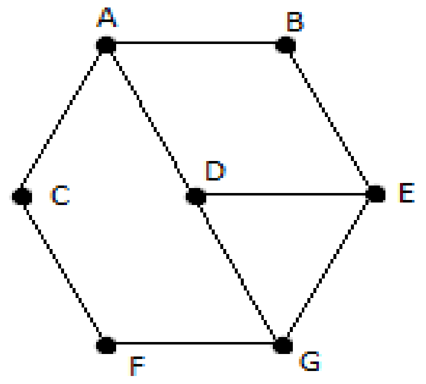
EXERCISE

Question 1:
Consider the following graph:



a. Apply BFS to find to every possible node present in graph.
   Starting from 1.
a. Find all paths between 1 & 6.
b. Find shortest path between 1 & 6.

Question 2:
Consider the following graph:



a. Apply BFS to find to every possible node present in graph. Start from A.
b. Find all paths between A & G.
c. Find shortest path between A & G.