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

ARTIFICIAL INTELLIGENCE

PRACTICAL 04

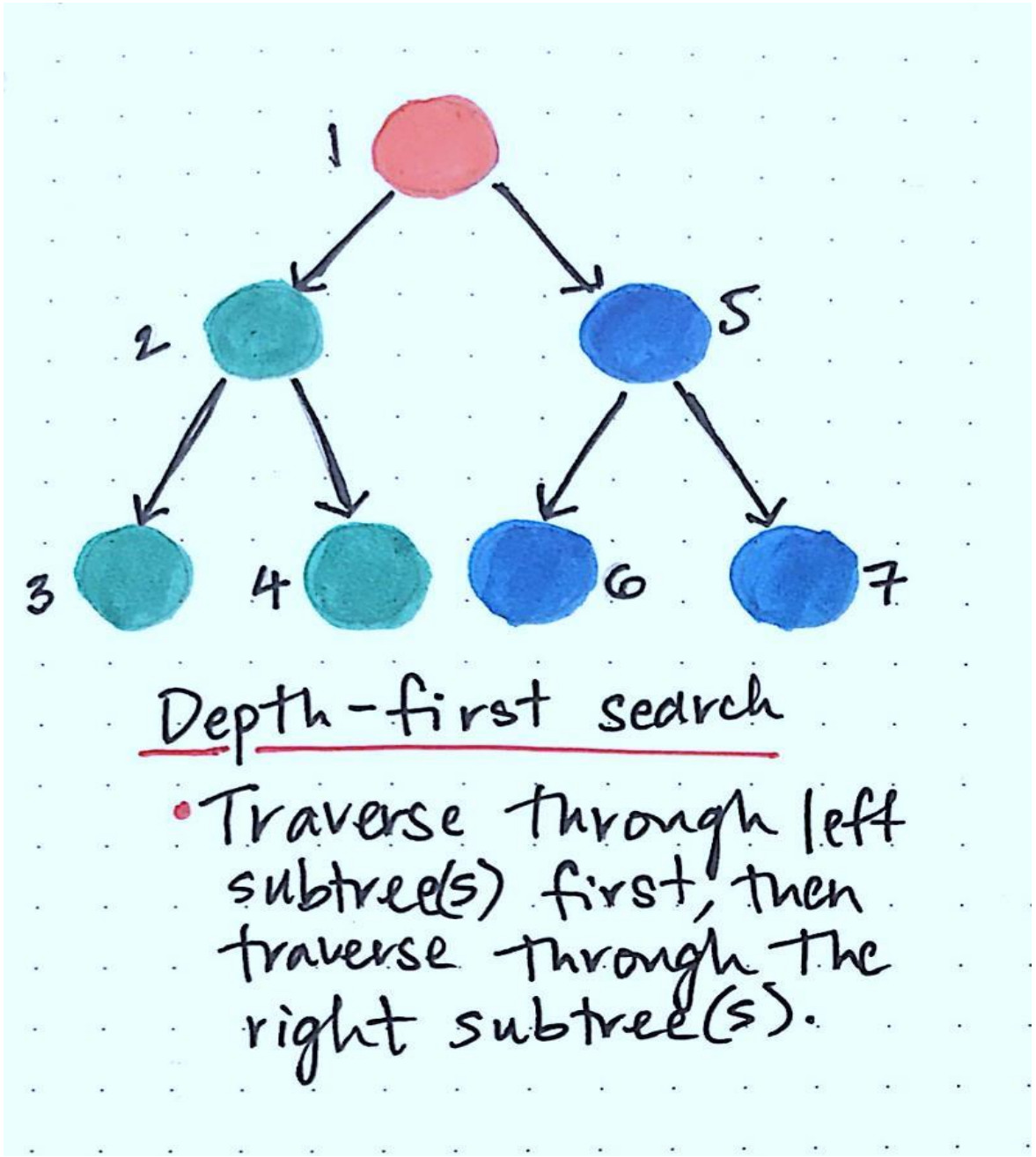
OBJECT: GRAPH TRAVERSAL: DEPTH FIRST SEARCH

Outline

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

Required Tools

- PC with windows
- Anaconda environment



APPLICATIONS OF DEPTH FIRST SEARCH

Following are the problems that use DFS as a building block.

1. For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree. DFS is at the heart of Prims and Kruskals algorithms.

2. Detecting cycle in a graph
   A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

3. Path Finding

   DFS algorithm can be used to find a path between two given vertices u and z.
   i) Call DFS(G, u) with u as the start vertex.
   ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
   iii) As soon as destination vertex z is encountered, return the path as the
   contents of the stack

4. Topological Sorting
   DFS is an intermediate step for topological sorting.

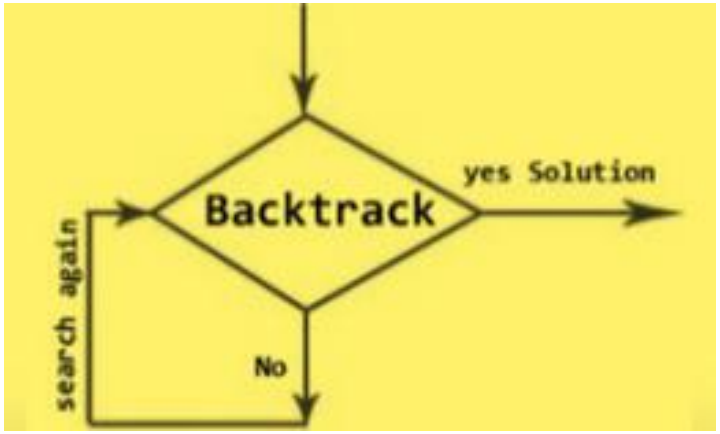
5. Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based algo for finding Strongly Connected Components)

6. DFS is very helpful in solving almost all the maze puzzles. Most of the maze puzzles can be converted into Graph problems and traversals result into the solution. DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.

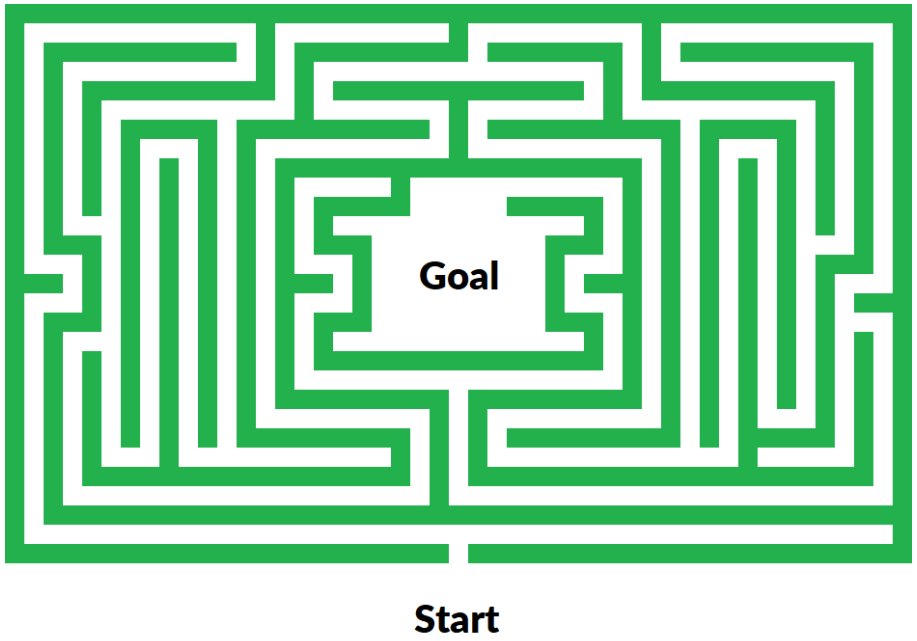
APPLICATIONS OF DEPTH FIRST SEARCH

- Backtracking

  Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.



- Solving puzzles with only one solution such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

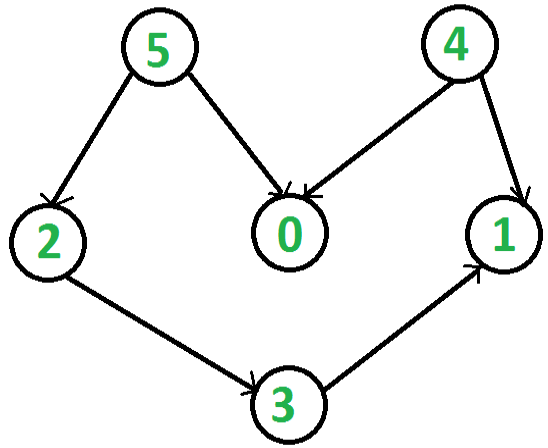


APPLICATIONS OF DEPTH FIRST SEARCH

- Topological Sorting

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs.

For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, and the vertex ‘4’ should also be printed before vertex ‘0’.



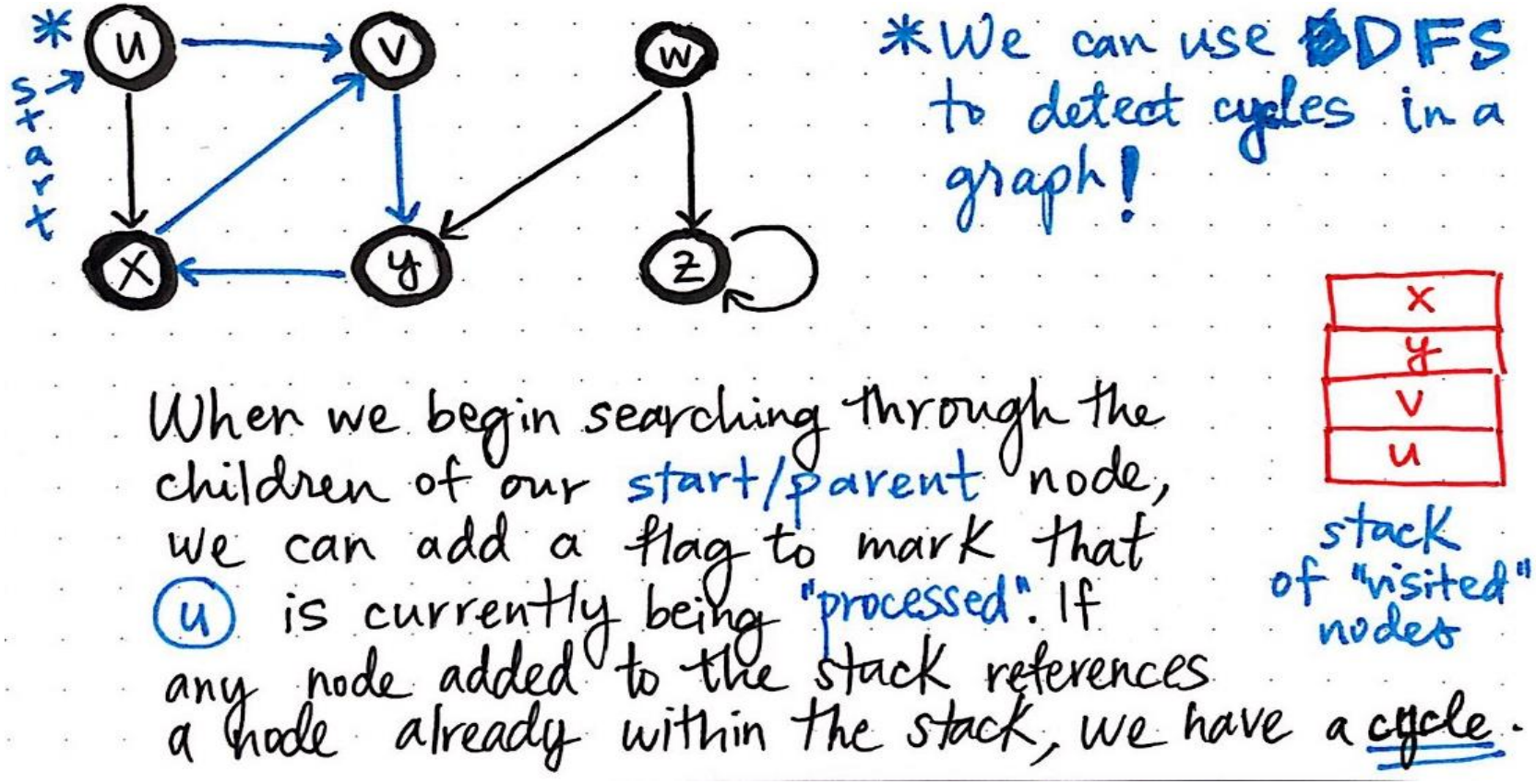
For example, a topological sorting of the following graph is “5 4 2 3 1 0”.

For example, another topological sorting of the following graph is “4 5 2 3 1 0”.

APPLICATIONS OF DEPTH FIRST SEARCH

- Detecting cycle in a graph

graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.



DFS(DEPTH FIRST SEARCH)

Depth First Search

Depth First Search (DFS) searches deeper into the problem space. Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

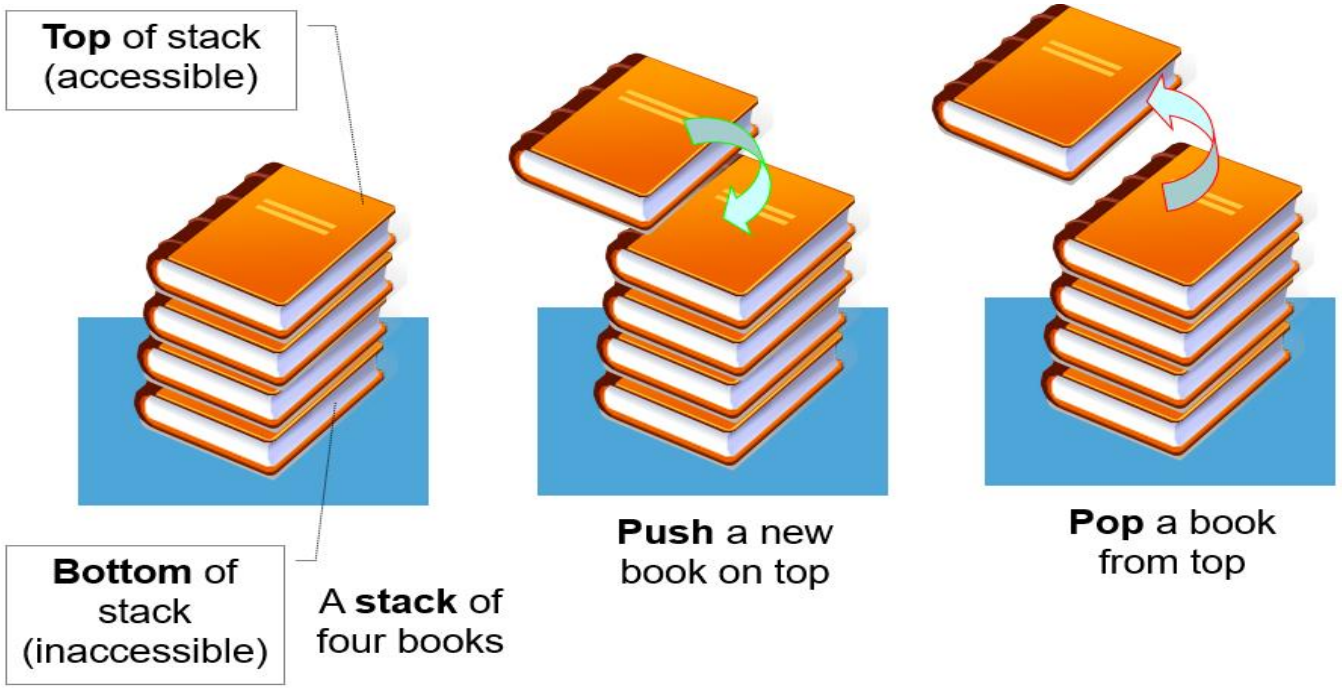
The Intuition behind DFS

- The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
- Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

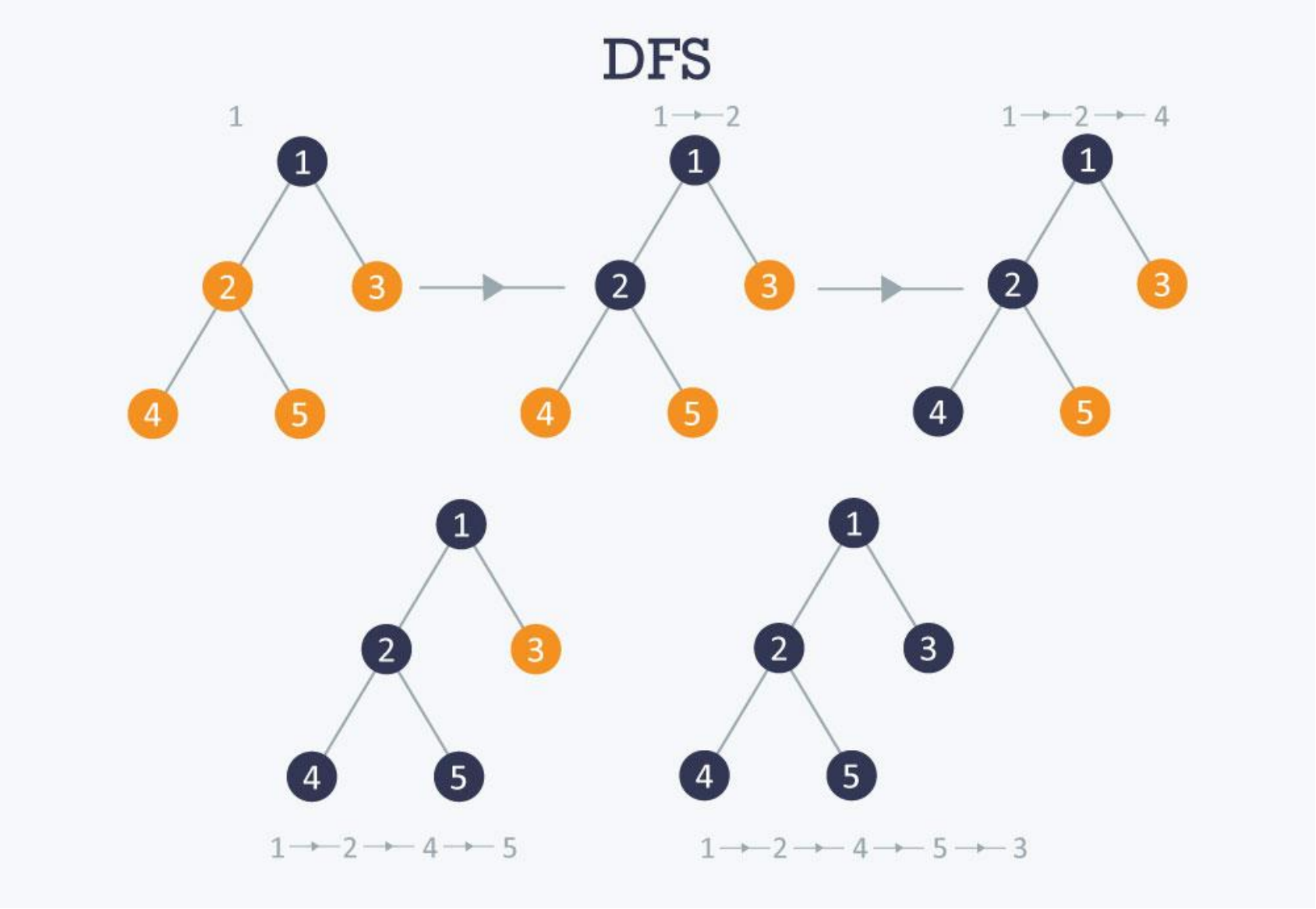
DFS(DEPTH FIRST SEARCH) ALGORITHM

- This recursive nature of *DFS can be implemented using stacks*. The basic idea is as follows:

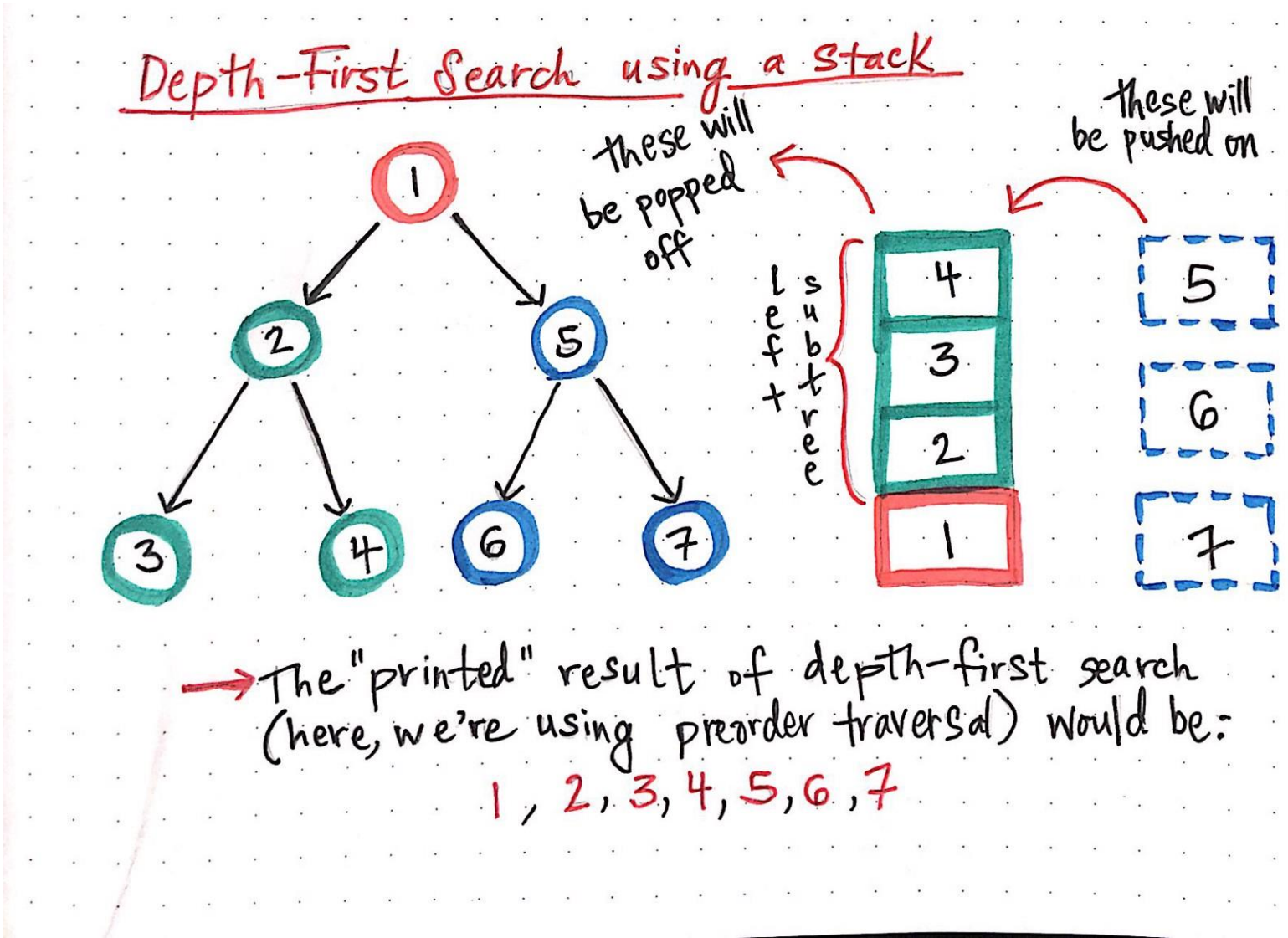
Rule 1: Choose some starting vertex (node), mark it as visited and push it on the stack.
Rule 2: Visit adjacent unvisited vertex. Mark it visited. Display it. Push it on the stack.
Rule 3: If no adjacent vertex found, pop out vertex from stack.
Rule 4: Repeat Rule 2 and Rule 3



DFS(DEPTH FIRST SEARCH)-EXAMPLE 1

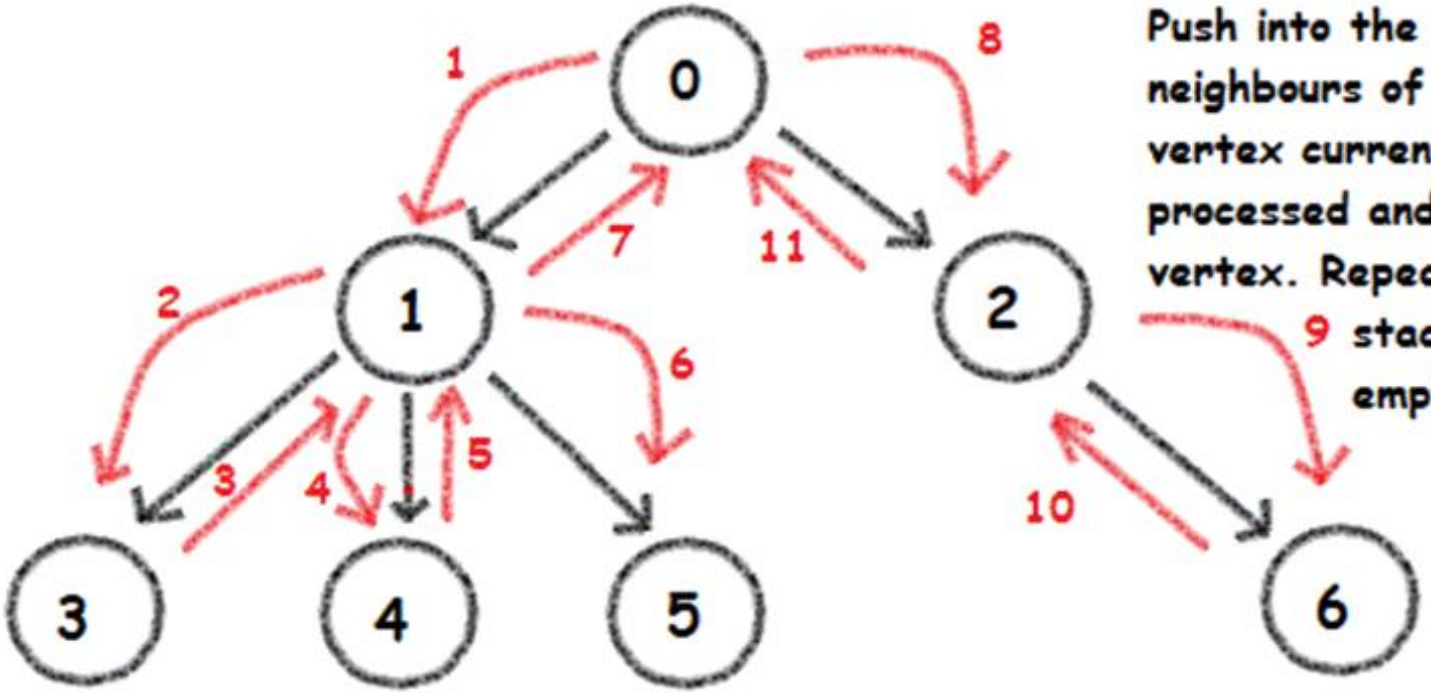


DEPTH FIRST SEARCH ALGORITHM-EXAMPLE 2



DEPTH FIRST SEARCH ALGORITHM-EXAMPLE 3

Red arrows indicate the order of search.

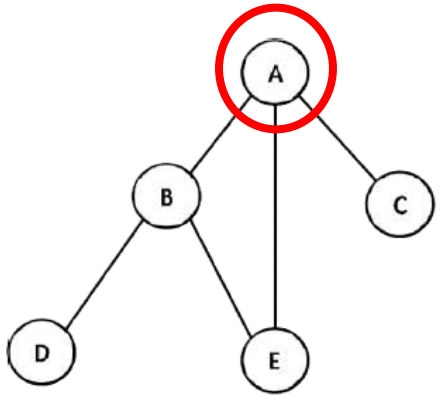


Depth First Search

Push into the stack the neighbours of the vertex currently being processed and Pop the vertex. Repeat until stack is not empty.

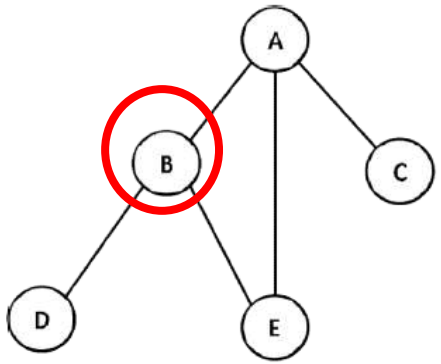
Vertex	Stack
	0
0	1, 2
1	3, 4, 5, 2
3	4, 5, 2
4	5, 2
5	2
2	6
6	

UNDERSTANDING DFS



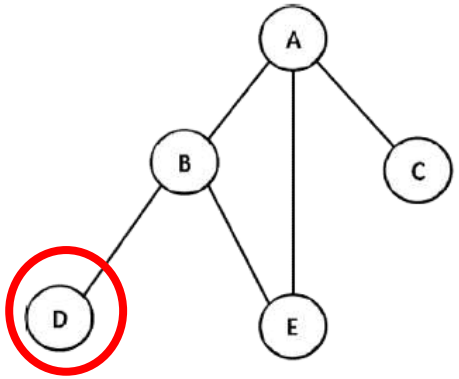
Step 01:
Start with node A. Mark it as visited, display it in path and push it on the stack.

Stack	A						
Visited	A						
Path	A						



Step 02:
Find unvisited adjacent node of A mark it as visited, push it on stack and display it in path.

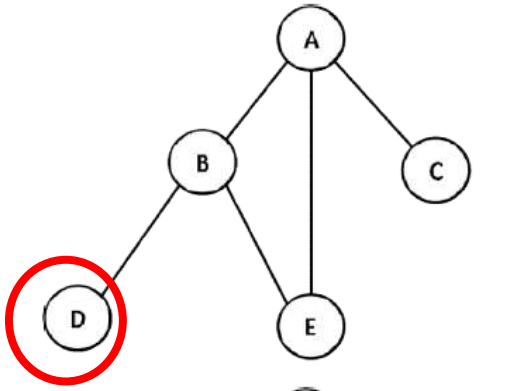
Stack	A	B					
Visited	A	B					
Path	A	B					



Step 03:
Find unvisited adjacent node of B mark it as visited, push it on stack and display it in path.

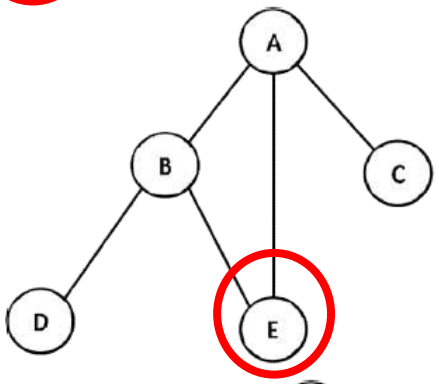
Stack	A	B	D				
Visited	A	B	D				
Path	A	B	D				

UNDERSTANDING DFS



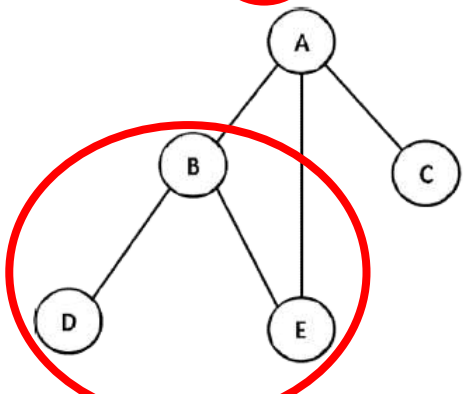
Step 04:
There is no unvisited adjacent node from D;
pop out node from stack (D).

Stack	A	B					
Visited	A	B	D				
Path	A	B	D				



Step 05:
Find unvisited adjacent node of B mark it as visited, push it on stack and display it in path.

Stack	A	B	E				
Visited	A	B	D	E			
Path	A	B	D	E			



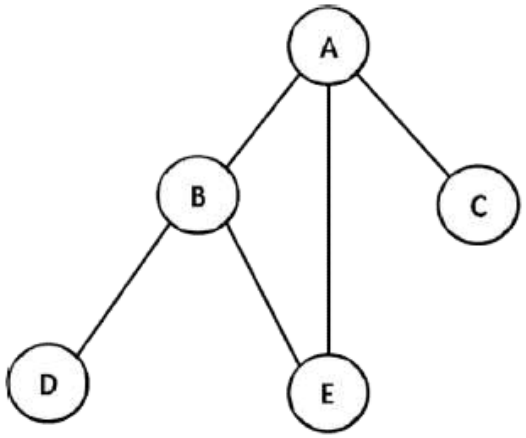
Step 06:
There is no unvisited adjacent node from E;
pop out node from stack (E).

Stack	A	B					
Visited	A	B	D	E			
Path	A	B	D	E			

Step 07:
There is no unvisited adjacent node from B;
pop out node from stack (B).

Stack	A						
Visited	A	B	D	E			
Path	A	B	D	E			

UNDERSTANDING DFS



As the stack is empty all the nodes have been visited, the algorithms stops here.

The traversal path using breadth first search is:
A → B → D → E → C

Step 08:
Find unvisited adjacent node of A mark it as visited, push it on stack and display it in path.

Stack	A	C					
Visited	A	B	D	E	C		
Path	A	B	D	E	C		

Step 09:
There is no unvisited adjacent node from C; pop out node from stack (C).

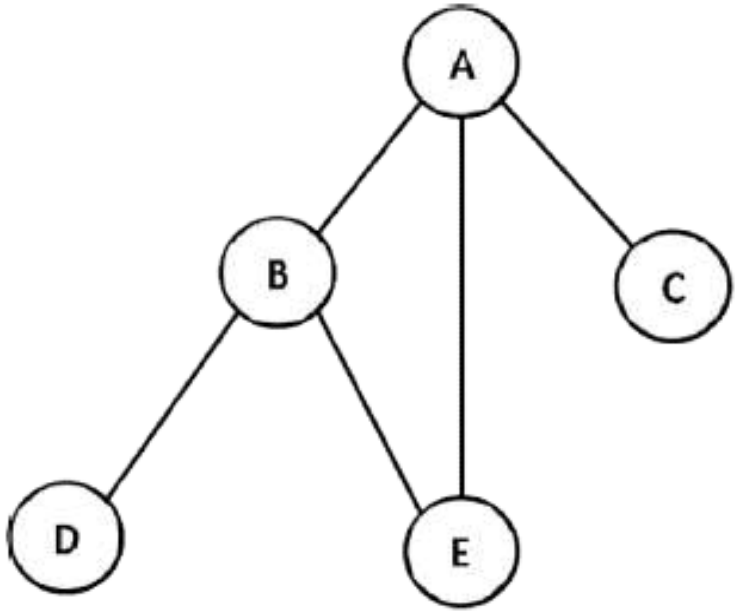
Stack	A						
Visited	A	B	D	E	C		
Path	A	B	D	E	C		

Step 10:
There is no unvisited adjacent node from A; pop out node from stack (A).

Stack							
Visited	A	B	D	E	C		
Path	A	B	D	E	C		

PYTHON CODE FOR DFS IMPLEMENTATION

EXAMPLE 1: Find the traversal path for following graph:

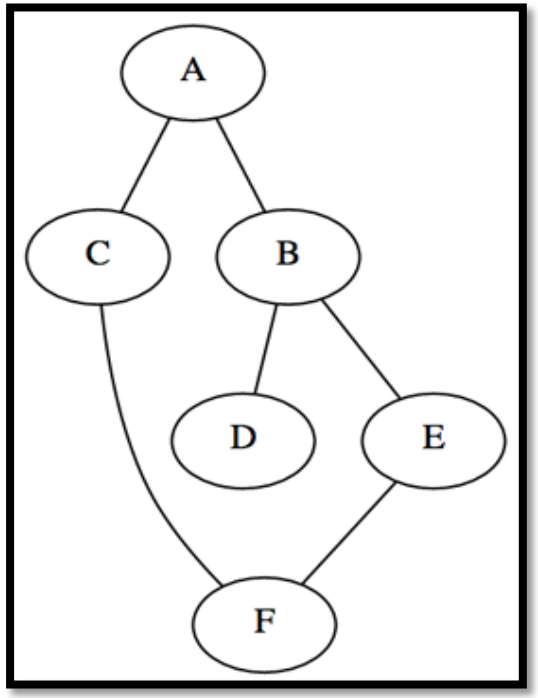


Traversal Path in DFS

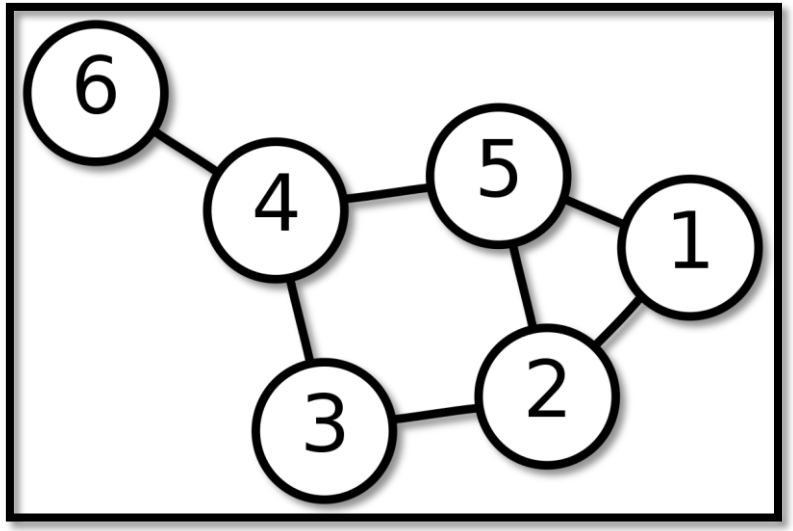
```
1  graph1 = {
2      'A' : ['B','E','C'],
3      'B' : ['D','E'],
4      'C' : [],
5      'D' : [],
6      'E' : []
7      }
8
9  def dfs(graph, node, visited):
10     if node not in visited:
11         visited.append(node)
12         for n in graph[node]:
13             dfs(graph,n, visited)
14     return visited
15
16 visited = dfs(graph1,'A', [])
17 print(visited)
```

PRACTICE

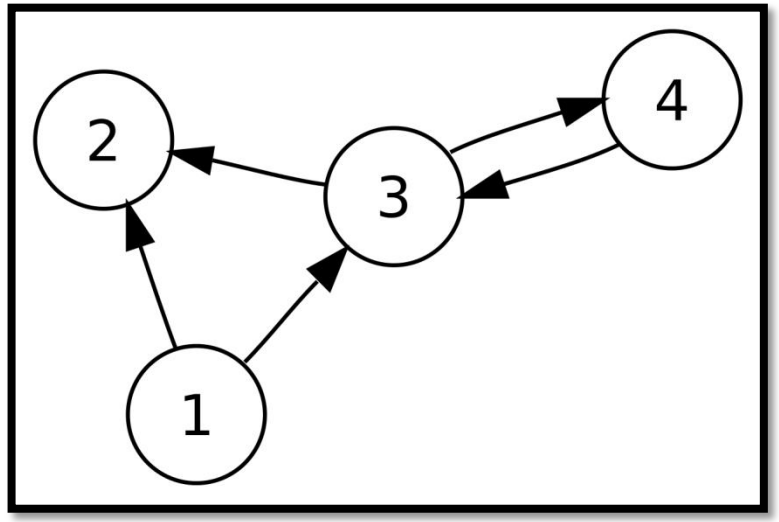
TASK 1 : Find the traversal path for following graphs.



A



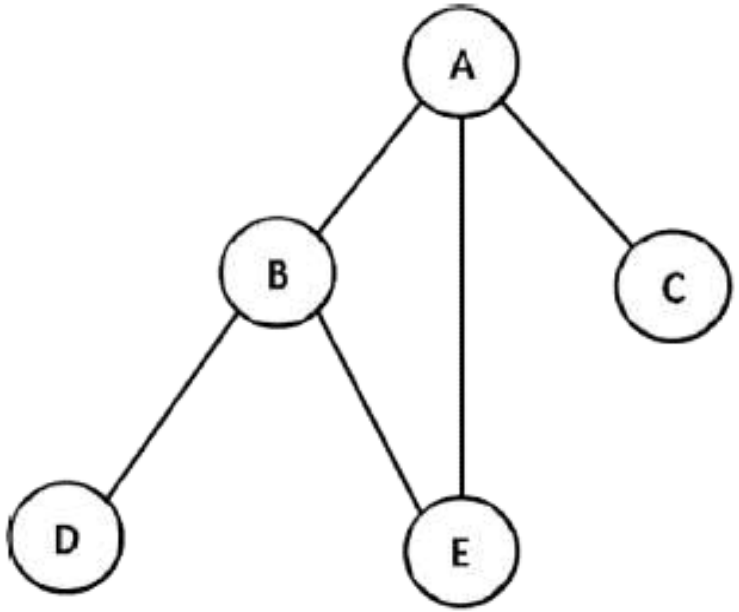
B



C

PYTHON CODE FOR DFS PATH FINDING

EXAMPLE : SHOW A PATH BETWEEN TWO NODES.

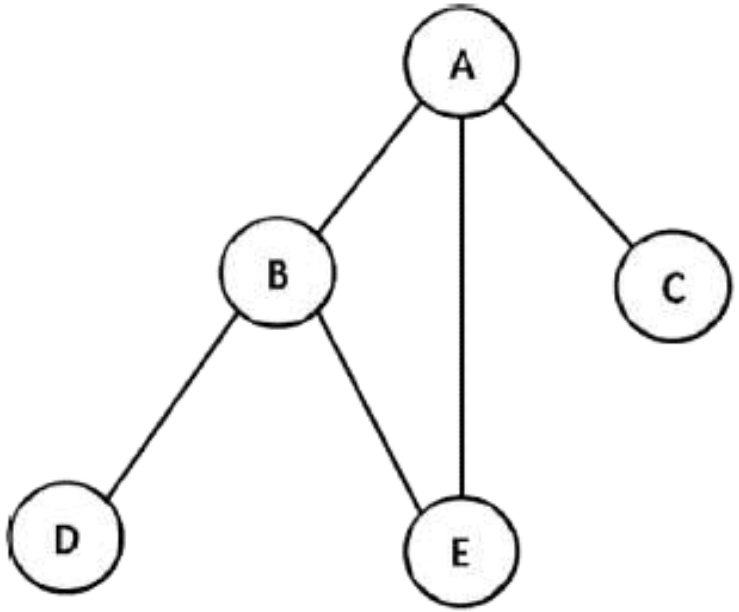


A PATH BETWEEN TWO NODES

```
graph = {
    'A' : set(['B','E','C']),
    'B' : set(['D','E']),
    'C' : set([]),
    'D' : set([]),
    'E' : set([])
    }
def find_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    if not graph.has_key(start):
        return None
    for node in graph[start]:
        if node not in path:
            newpath = find_path(graph, node, end, path)
            if newpath: return newpath
    return None
print(find_path(graph,'A','E'))
```

PYTHON CODE FOR DFS PATH FINDING

EXAMPLE : SHOW ALL THE PATHS

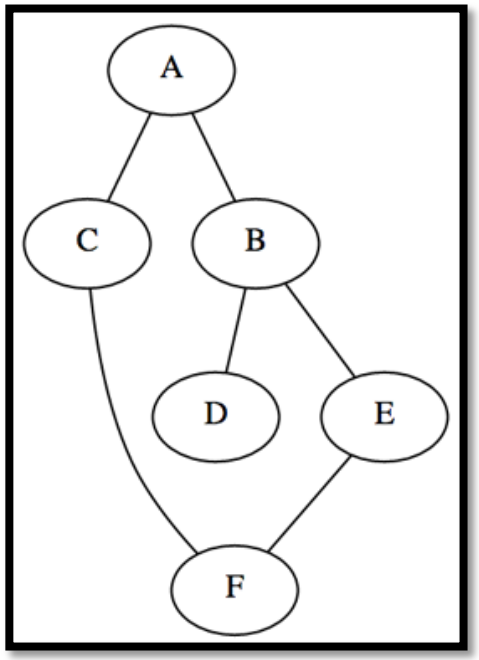


ALL PATHS BETWEEN TWO NODES

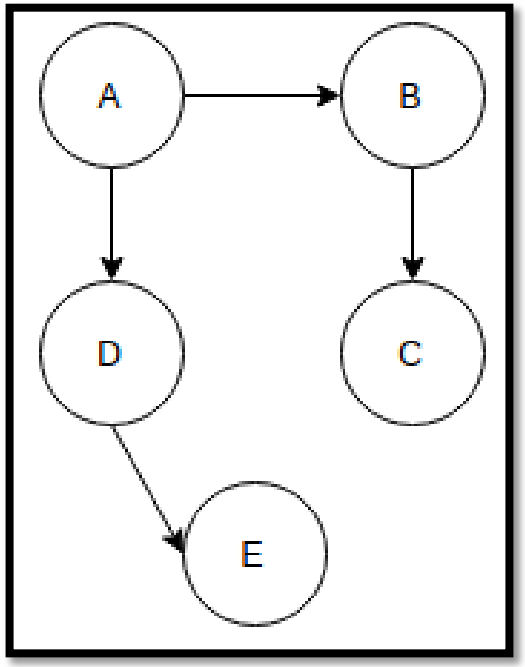
```
graph = {
     'A' : set(['B','E','C']),
     'B' : set(['D','E']),
     'C' : set([]),
     'D' : set([]),
     'E' : set([])
     }
def find_all_paths(graph, start, end, path=[]):
     path = path + [start]
     if start == end:
          return [path]
     if not graph.has_key(start):
          return []
     paths = []
     for node in graph[start]:
          if node not in path:
               newpaths = find_all_paths(graph, node, end, path)
               for newpath in newpaths:
                    paths.append(newpath)
     return paths
print(find_all_paths(graph,'A','E'))
```

PRACTICE

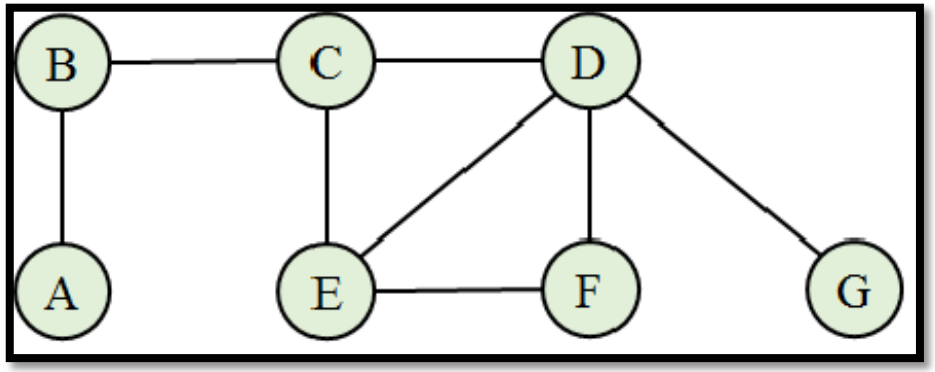
TASK 2 : Find the single path & all paths between any two nodes for the following graphs.



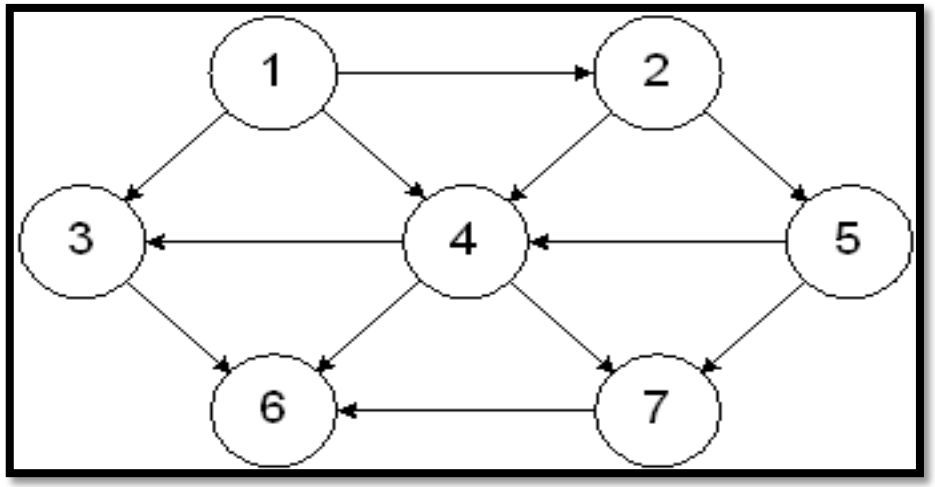
A



B



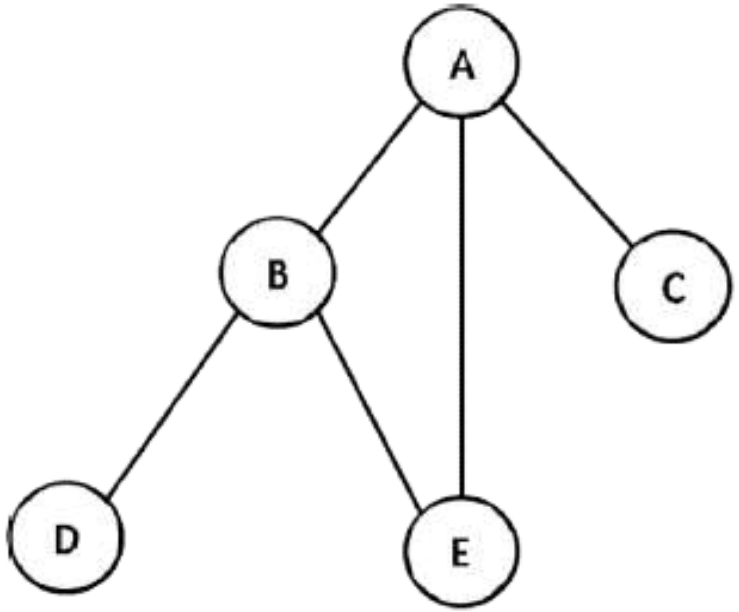
C



C

PYTHON CODE FOR DFS PATH FINDING

EXAMPLE : FIND SHORTEST PATH FOR FOLLOWING.

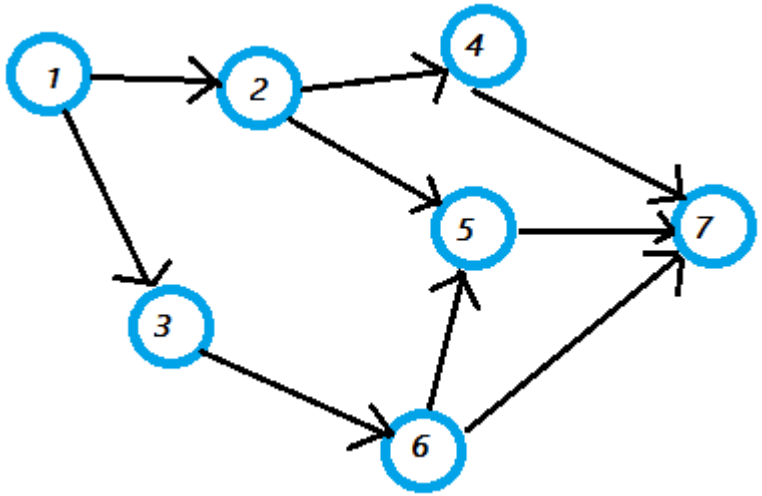


SHORTEST BETWEEN TWO NODES

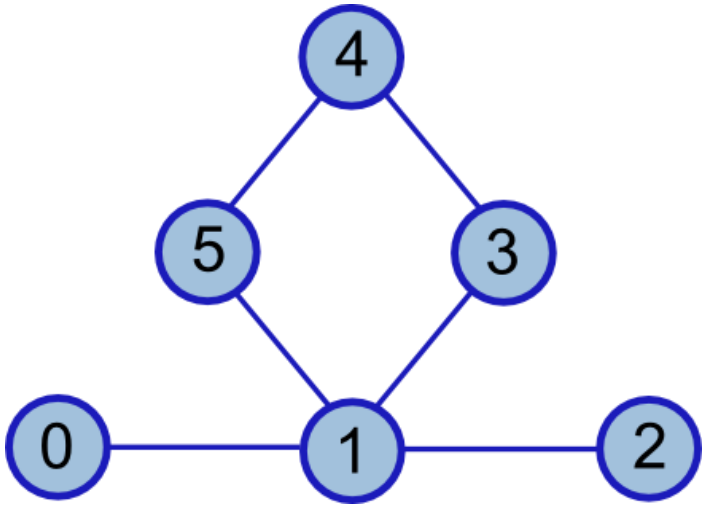
```
graph = {
    'A' : set(['B','E','C']),
    'B' : set(['D','E']),
    'C' : set([]),
    'D' : set([]),
    'E' : set([])
    }
def find_shortest_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    if not graph.has_key(start):
        return None
    shortest = None
    for node in graph[start]:
        if node not in path:
            newpath = find_shortest_path(graph, node, end, path)
            if newpath:
                if not shortest or len(newpath) < len(shortest):
                    shortest = newpath
    return shortest
print(find_shortest_path(graph,'A','E'))
```

PRACTICE

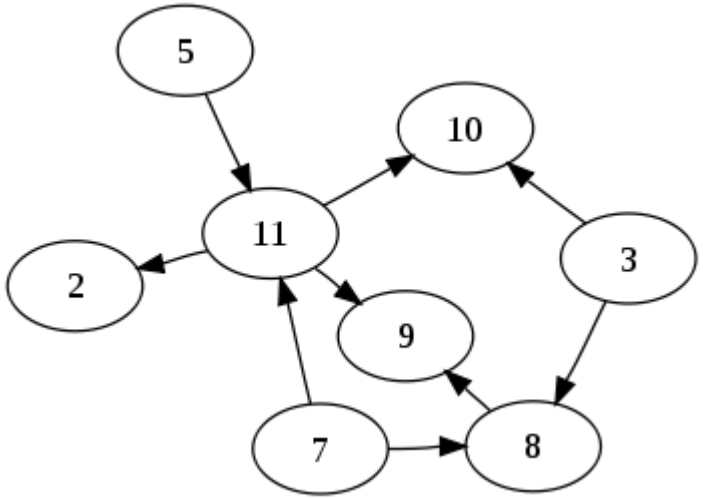
TASK 3 : Find the shortest paths between any two nodes for the following graphs.



A



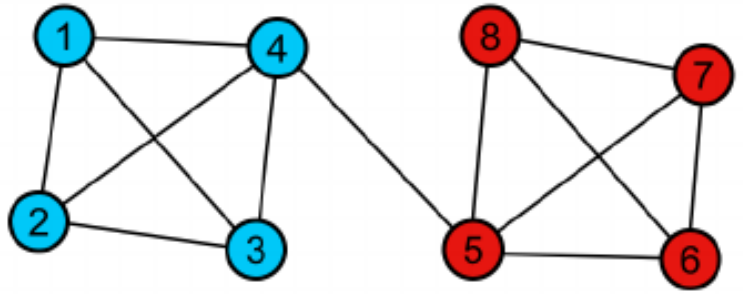
B



C

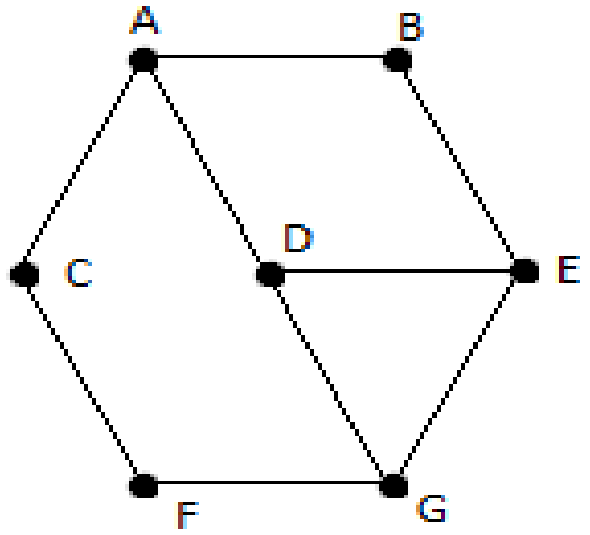
EXERCISE

Question 1:
Consider the following graph:



a. Apply DFS to find traversal path.
b. Find single path between 1 & 6.
c. Find all paths between 1 & 6.
d. Find shortest path between 1 & 6.

Question 2:
Consider the following graph:



a. Apply DFS to find traversal path.
b. Find single path between A & G.
c. Find all paths between A & G.
d. Find shortest path between A & G.