IQRA University

CSC 471 Artificial Intelligence

Lab# 03

Search Algorithms in Artificial Intelligence

Objective:

This experiment introduces the students to the modeling of different problems as a search problem and their solution exploration using different uniformed search algorithms

Name of Student: **FARAZ ALAM**

Roll No: **13948** **Sec.** **BS-CS**

Date of Experiment: **04/11/2023**

Lab 03: Uninformed and Informed Search in Artificial Intelligence

Search

Search is looking for a sequence of actions that reaches the goal states. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. After formulating a goal and a problem to solve, a search procedure is called to solve it. The solution is then used to guide the actions, whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the goal is reached.

Formulating Problems

There are six components which formulate a problem as a search problem.

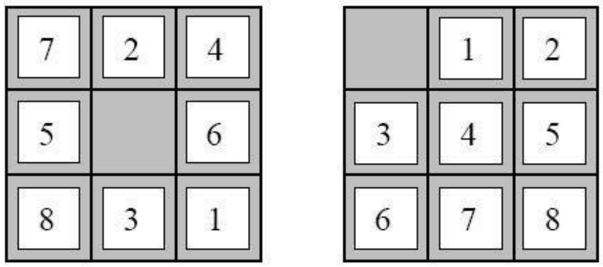
- Initial State
- Actions
- Transition model
- Goal test
- State Space
- Path cost

form a problem. This formulation is abstract, i.e., details are hidden. Abstraction is useful since they simplify the problem by hiding many details but still covering the most important information about states and actions (retaining the state space in simple form), therefore abstraction needs to be valid. Abstraction is called valid when the abstract solution can be expanded to more detailed world. Abstraction is useful if the actions in the solution are easier than the original problem, i.e, no further planning and searching.
Construction of useful and valid abstraction is challenging.

Example Problem: 8-Puzzle Problem

The 8-puzzle is often used as test problem for new search algorithms in AI. The 8-puzzle has 9!/2 = 181,440 reachable states and is easily solved. The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24puzzle (on a 5 × 5 board) has around 1025 states, and random instances take several hours to solve optimally. The 8-puzzle, an instance of which is shown in the figure, consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

- **States:** Description of the location of each of the eight tiles and the blank square.
- **Initial State:** Initial configuration of the puzzle.
- **Actions & transition model:** Moving the blank; left, right, up, or down.
- **Goal Test:** Does the state match the goal state?
- **Path Cost:** Each step costs 1 unit



Start State Goal State

Search Schemes

Searching is the universal technique of problem solving in AI.

Infrastructure for a Search Algorithm

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
     STATE = problem.RESULT(parent.STATE, action),
     PARENT = parent, ACTION = action,
     PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Next, the frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand QUEUE according to its preferred strategy. The appropriate data structure for this is a queue. The operations on a queue are as follows:

- EMPTY? (queue) returns true only if there are no more elements in the queue.
- POP (queue) removes the first element of the queue and returns it.
- INSERT (element, queue) inserts an element and returns the resulting queue.

Types of Search Algorithm

There are some single-player games such as tile games, Sudoku, crossword, etc. The search algorithms help you to search for a particular position in such games. There are two kinds of AI search techniques:

- Uninformed search
- Informed search

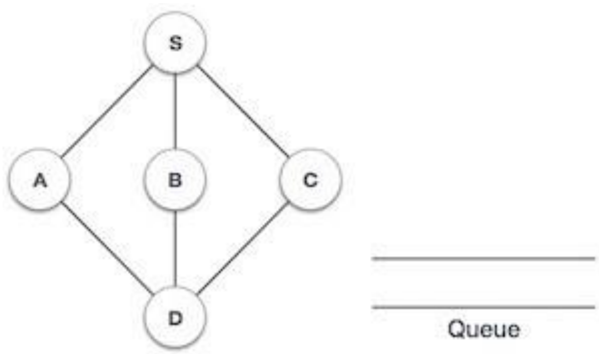
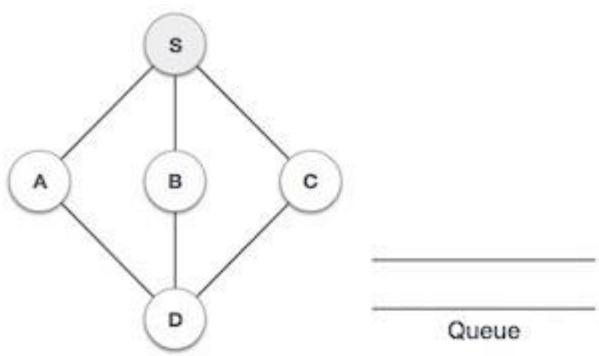
Uninformed Search

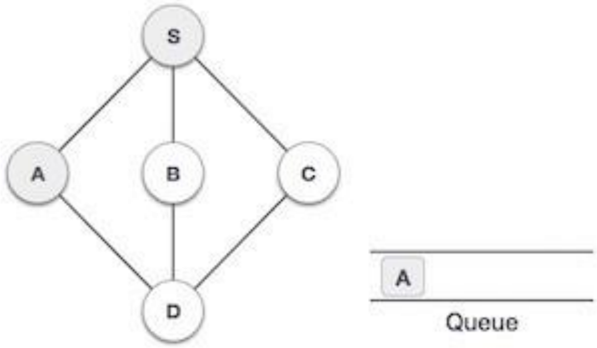
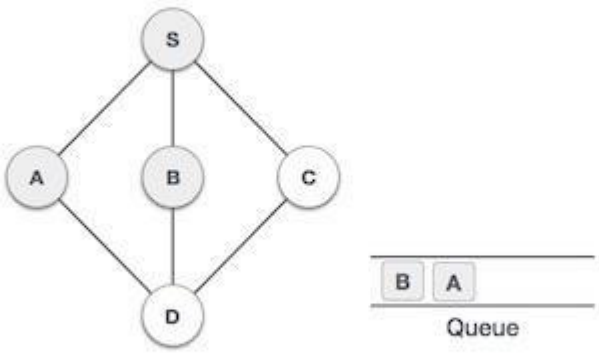
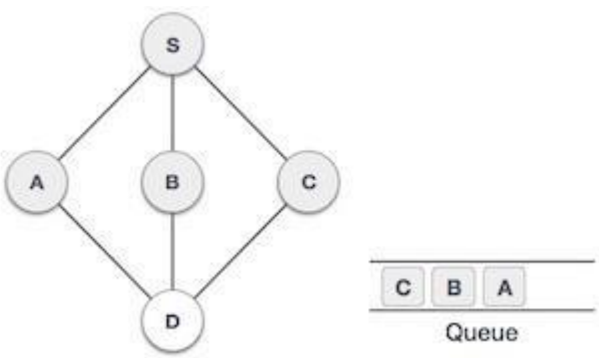
Sometimes we may not get much relevant information to solve a problem. Suppose we lost our car key and we are not able to recall where we left, we have to search for the key with some information such as in which places we used to place it. It may be our pant pocket or may be the table drawer. If it is not there then we have to search the whole house to get it. The best solution would be to search in the places from the table to the wardrobe. Here we need to search blindly with fewer clues. This type of search is called uninformed search or blind search. There are two popular AI search techniques in this category:

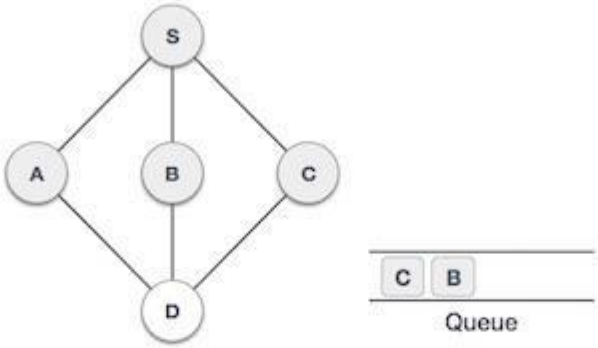
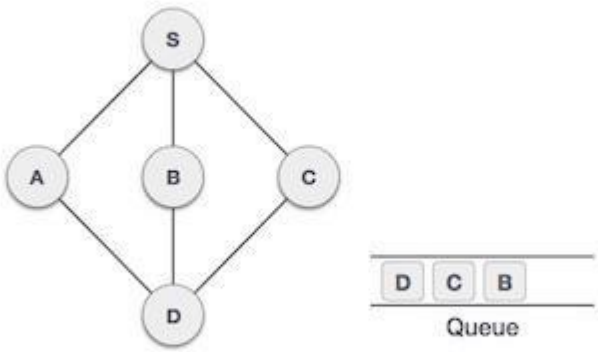
- Breadth first search
- Depth first search

Breadth-First Search

It starts from the root node, explores the neighboring nodes first and moves towards the next level neighbors. It generates one tree at a time until the solution is found. It can be implemented using FIFO queue data structure. This method provides shortest path to the solution.

Step	Traversal	Description
1.		Initialize the queue.
2.		We start from visiting **S** (starting node), and mark it as visited.

3.		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4.		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5.		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6.		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7.		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

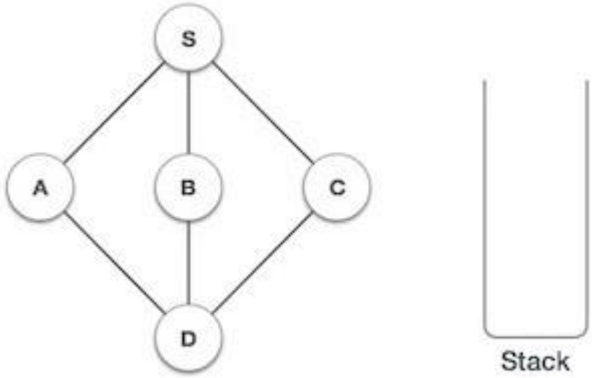
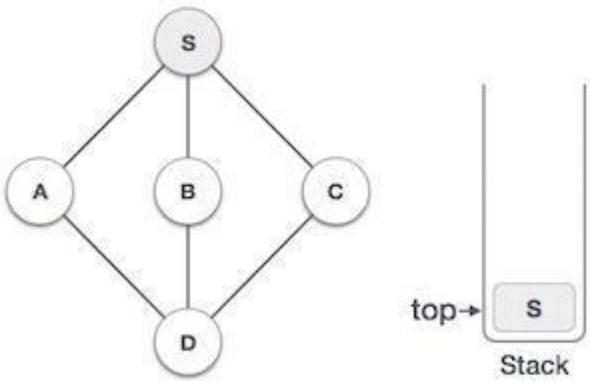
Pseudocode

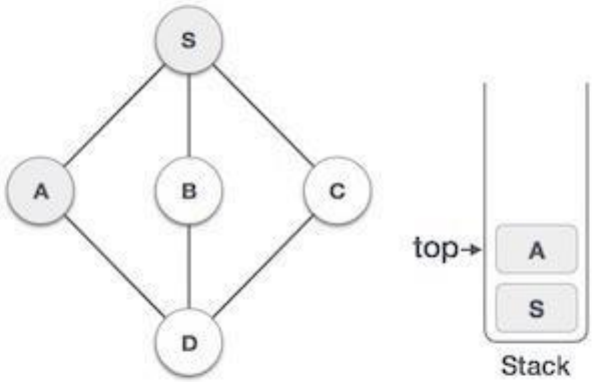
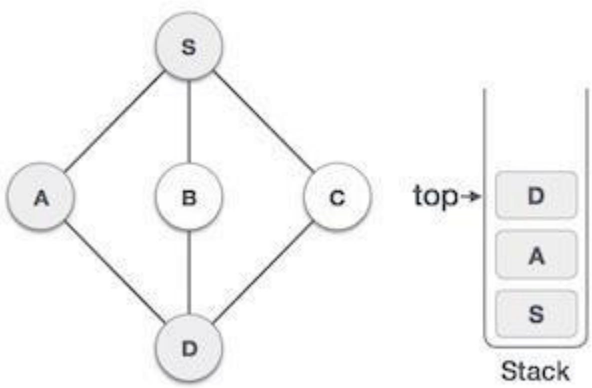
```
BFS (G, s)                   //Where G is the graph and s is the source node
let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices
are marked.
       mark s as
visited.
      while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be visited now
v  =  Q.dequeue( )

          //processing all the neighbours of v
for all neighbours w of v in Graph G
if w is not visited
                        Q.enqueue( w )  //Stores w in Q to further visit its
                                          neighbour
                        mark w as visited.
```

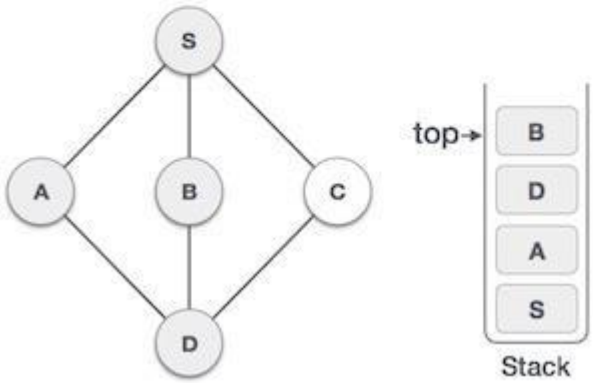
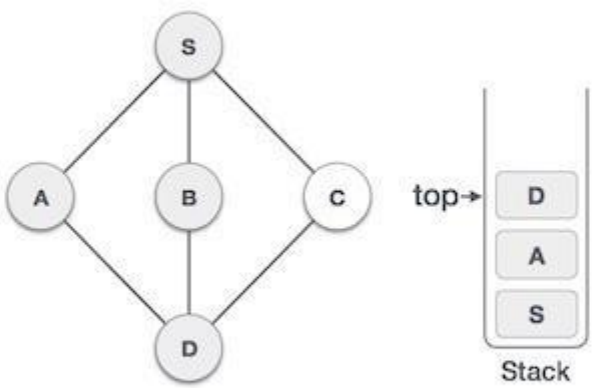
Depth-First Search

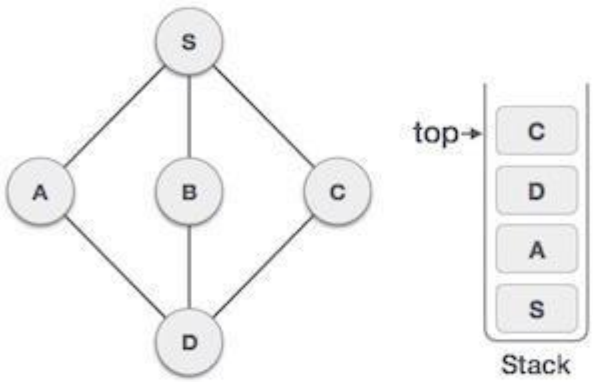
It is implemented in recursion with LIFO stack data structure. It creates the same set of nodes as BreadthFirst method, only in the different order.

Step	Traversal	Description

1.		Initialize the stack.
2.		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3.		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4.		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

5.		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7.	 	Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

Pseudocode

```
  DFS-iterative (G, s):                                          //Where G is graph
and s is source vertex       let S be stack
      S.push( s )            //Inserting s in
stack        mark s as visited.       while ( S is
not empty):
          //Pop a vertex from stack to visit next
v  =  S.top( )
         S.pop( )
         //Push all the neighbours of v in stack that are not
visited            for all neighbours w of v in Graph G:
if w is not visited :
                     S.push( w )
                    mark w as visited


   DFS-recursive(G, s):         mark s as
visited         for all neighbours w of s in
Graph G:             if w is not visited:
                DFS-recursive(G, w)
```

Student Exercise

Task 1

For the algorithm provided to you in the Notebook, can you identify the algorithm that is already implemented? BFS or DFS?

Whichever algorithm you guessed, implement the other variant (do not forget to make a copy of the notebook before diving in).

<u>Answer:</u>

The provided code is for Breadth-First Search (BFS), In BFS, we explore all neighbors of the current node before moving to the next level of nodes.

Now Implementing Depth-First Search (DFS):

jupyter Lab#3Task01 Last Checkpoint: a few seconds ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run Code

In [10]:

```python
graph = { '1': ['2', '3', '4'],
    '2': ['5', '6'],
    '3':[],
    '4': ['7', '8'],
    '5': ['9', '10'],
    '6':[],
    '7': ['11', '12'],
    '8':[],
    '9':[],
    '10':[],
    '11':[],
    '12':[] }
class Node:
    def __init__(self, state, parent, move, cost):
        self.state = state
        self.parent = parent
        self.move = move
        self.path_cost = cost
stack = []
start_state = Node('1', None, None, None)
goal_state = '11'
stack.append(start_state)
path = []
def print_path(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path
while stack:
    current_node = stack.pop()
    if current_node.state == goal_state:
        print('Solution found')
        route = print_path(current_node)
        print(route[::-1])
        break
    for neighbor in graph[current_node.state]:
        state = Node(neighbor, current_node, None, None)
        stack.append(state)
else:
    print('Failure')
```

```
Solution found
['1', '4', '7', '11']
```

Task2

Solve the following problem with the existing code (provided to you in the notebook).

- **Any leaf node (nodes not having any child nodes) are to be represented within the graph (the dictionary named graph). Can you identify the issue that might arise if this is not done. Also provide a solution for the identified issue so that the leaf nodes aren't to be represented within the graph.**

Answer:

If the graph does not include representation of leaf nodes, there may be a problem with the Breadth-First Search (BFS) algorithm. In the given graph, nodes '3', '8', '6', '9', '10', '11', and '12' do not have any children. If these nodes are not included in the graph, the BFS algorithm may face encounter this issue when attempting to explore them.

To address this issue and eliminate the need to represent leaf nodes in the graph, while ensuring the algorithm functions correctly, we can implement a check to verify the existence of a neighbor in the graph dictionary before attempting to explore it.

jupyter Lab#03Task02 Last Checkpoint: a few seconds ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

In [5]:

```python
graph = {
    '1': ['2', '3', '4'],
    '2': ['5', '6'],
    '4': ['7', '8'],
    '5': ['9', '10'],
    '7': ['11', '12'],
}
class Node:
    def __init__(self, state, parent, move, cost):
        self.state = state
        self.parent = parent
        self.move = move
        self.path_cost = cost

def print_path(node):
    path = []
    while node.parent:
        path.append(node.parent.state)
        node = node.parent
    path.append(node.state)
    return path
queue = []
start_state = Node('1', None, None, None)
goal_node = '11'
queue.append(start_state)
path = []

while queue:
    current_node = queue.pop(0)
    if current_node.state == goal_node:
        print('solution found')
        route = print_path(current_node)
        print(route[::-1])
        break
    for neighbor in graph.get(current_node.state, []):
        state = Node(neighbor, current_node, None, None)
        queue.append(state)
else:
    print('failure')
```

```
solution found
['1', '1', '4', '7']
```

- **The current implementation cannot identify any cycles within the graph. This can be achieved by simply adding a visited-node list and exploring that list before expanding any new node. Alter your code (copy of the existing notebook) to identify any cycles within the graph.**

jupyter Lab#03Task02 Last Checkpoint: a minute ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run Code

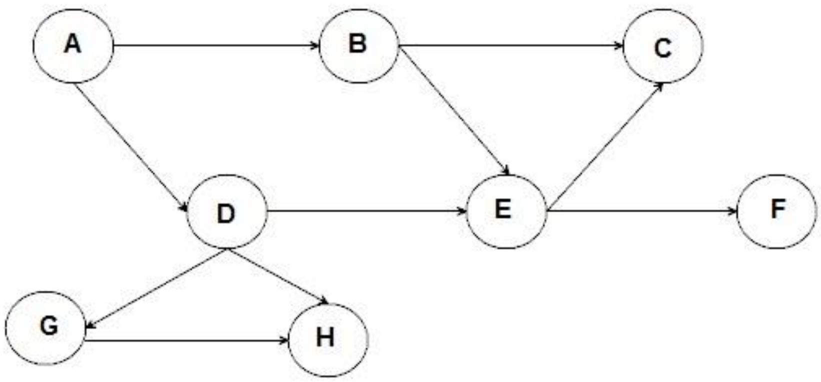
In [13]:

```python
graph = {
    '1': ['2', '3', '4'],
    '2': ['5', '6'],
    '4': ['7', '8'],
    '5': ['9', '10'],
    '7': ['11', '12'],
}
class Node:
    def __init__(self, state, parent, move, cost):
        self.state = state
        self.parent = parent
        self.move = move
        self.path_cost = cost
        self.visited = False
def print_path(node):
    path = []
    while node.parent:
        path.append(node.parent.state)
        node = node.parent
    path.append(node.state)
    return path
visited = set()
queue = []
start_state = Node('1', None, None, None)
goal_node = '11'
queue.append(start_state)
path = []
while queue:
    current_node = queue.pop(0)
    if current_node.state == goal_node:
        print('solution found')
        route = print_path(current_node)
        print(route[::-1])
        break
    if current_node.visited:
        print('Cycle detected!')
        break
    visited.add(current_node.state)
    current_node.visited = True
    for neighbor in graph.get(current_node.state, []):
        if neighbor not in visited:
            state = Node(neighbor, current_node, None, None)
            queue.append(state)
else:
    print('failure')
```

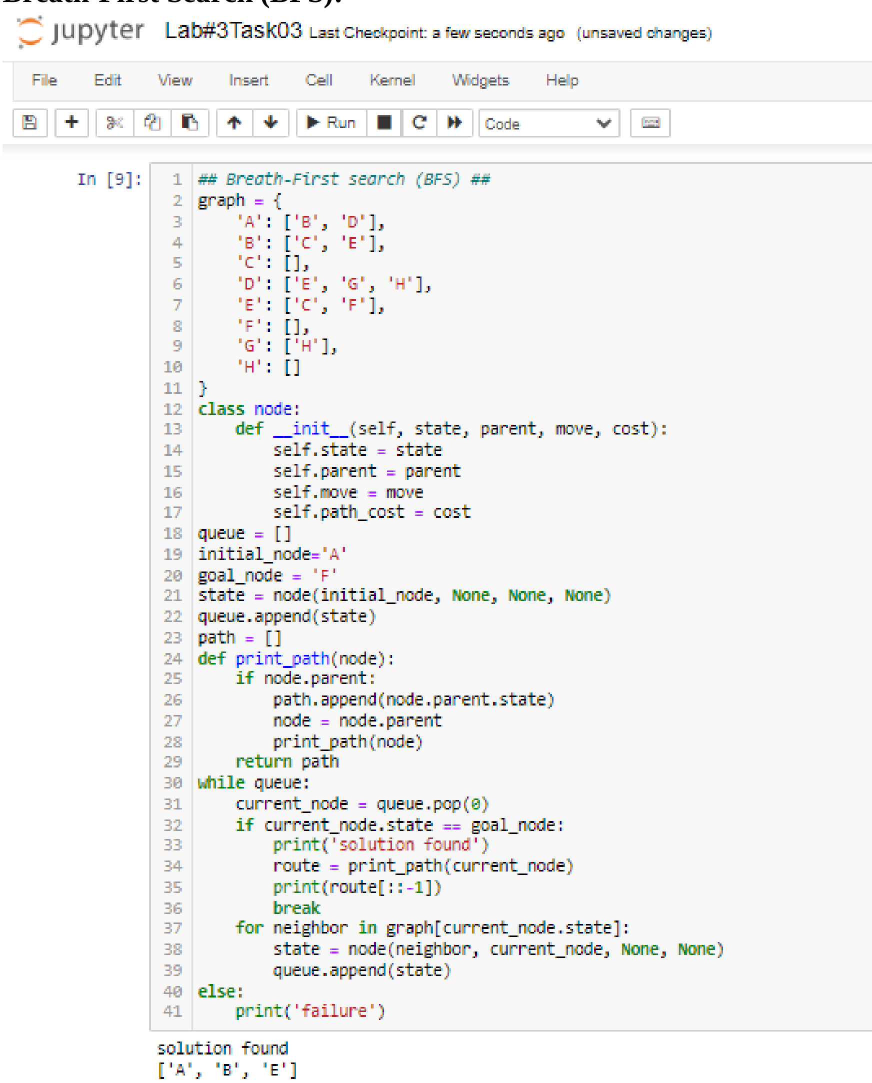
```
solution found
['1', '1', '4', '7']
```

Task 3

Implement BFS and DFS from the following graph: (Initial node A, Goal node F)



Breath-First Search (BFS):




```python
## Breath-First search (BFS) ##
graph = {
    'A': ['B', 'D'],
    'B': ['C', 'E'],
    'C': [],
    'D': ['E', 'G', 'H'],
    'E': ['C', 'F'],
    'F': [],
    'G': ['H'],
    'H': []
}
class node:
    def __init__(self, state, parent, move, cost):
        self.state = state
        self.parent = parent
        self.move = move
        self.path_cost = cost
queue = []
initial_node='A'
goal_node = 'F'
state = node(initial_node, None, None, None)
queue.append(state)
path = []
def print_path(node):
    if node.parent:
        path.append(node.parent.state)
        node = node.parent
        print_path(node)
    return path
while queue:
    current_node = queue.pop(0)
    if current_node.state == goal_node:
        print('solution found')
        route = print_path(current_node)
        print(route[::-1])
        break
    for neighbor in graph[current_node.state]:
        state = node(neighbor, current_node, None, None)
        queue.append(state)
else:
    print('failure')
```

```
solution found
['A', 'B', 'E']
```

Depth-First Search (DFS):

jupyter Lab#3Task03 Last Checkpoint: a few seconds ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

In [10]:

```python
## Depth-First Search (DFS) ##
graph = {
    'A': ['B', 'D'],
    'B': ['C', 'E'],
    'C': [],
    'D': ['E', 'G', 'H'],
    'E': ['C', 'F'],
    'F': [],
    'G': ['H'],
    'H': []
}
class Node:
    def __init__(self, state, parent, move, cost):
        self.state = state
        self.parent = parent
        self.move = move
        self.path_cost = cost
def print_path(node):
    path = []
    while node.parent:
        path.append(node.state)
        node = node.parent
    path.append(node.state)
    return path
stack = []
initial_node='A'
goal_node = 'F'
start_state = Node(initial_node, None, None, None)
stack.append(start_state)
while stack:
    current_node = stack.pop()
    if current_node.state == goal_node:
        print('Solution found')
        route = print_path(current_node)
        print(route)
        break

    for neighbor in graph.get(current_node.state, []):
        state = Node(neighbor, current_node, None, None)
        stack.append(state)
else:
    print('Failure')
```

```
Solution found
['F', 'E', 'D', 'A']
```