Artificial Intelligence
State Space Search Algorithms

Introduction

In this lab, we will concentrate on programming two basic searching strategies: Depth-First Search (DFS) and Breadth-First Search (BFS) in Java programming language.

Representing state space as Graph

In graph structure, we have two basic properties, vertex and edge. In state space, we can represent state as vertex and edge as state-transition, thus we can take advantage of graph algorithms.

Now, you should implement a Graph class as in **Figure 1**.

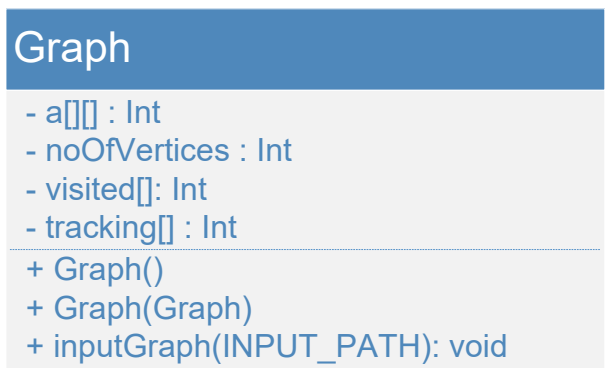
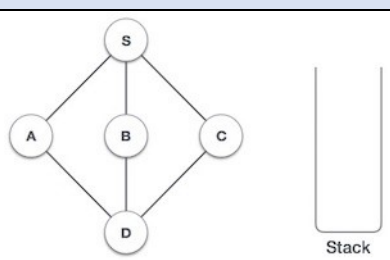
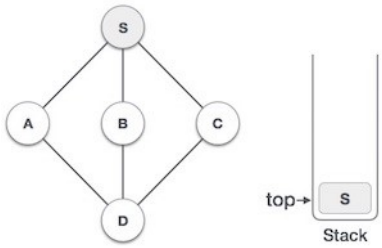
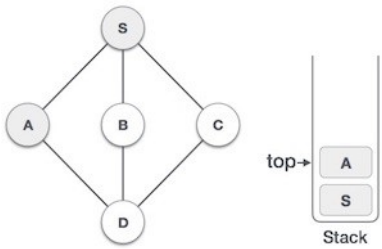
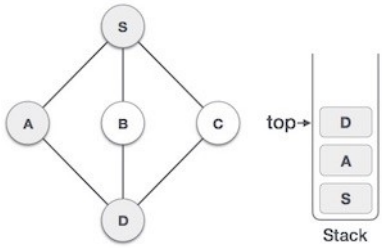
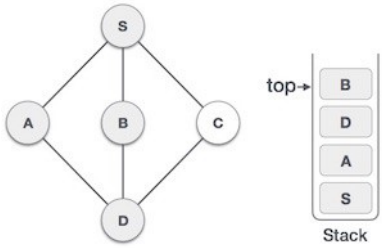
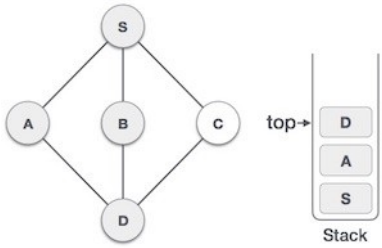
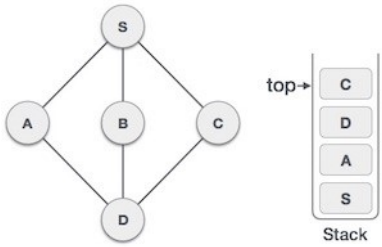


Figure 1 Graph class

Implementing Depth-first search algorithm

DFS is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. To illustrate the steps in running **DFS** to find a path between two nodes, let's see the example below.

Step	*Traversal*	*Description*	*Result*
1		Initialize the stack.	

2		Mark **S** as visited and push onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes, and we pick the node in alphabetical order, which is **A**.	**S**
3		Mark **A** as visited and push 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, that means, we visit **D**.	**S→A**
4		Visit **D** and mark it visited and push onto the stack. Here we have **B** and **C** nodes which are adjacent to **D** and both are unvisited. And we choose **B**, since alphabetical order.	**S→A→D**
5		Choose **B**, mark it visited and push onto the stack. Here **B** does not have any unvisited adjacent node. So we pop **B** from the stack.	**S→A→D→B**
6		We check the top of current stack for return to previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of stack.	**S→A→D→B**
7		Only unvisited adjacent node from **D** is **C** now. So we visit **C**, mark it visited and push onto the stack.	**S→A→D→B→C**

At this point, you knew how the **DFS** traverses the graph from the ***start*** node to the other nodes. We can summarize the **DFS** algorithm in the following rules:

- Visit adjacent unvisited vertex of the current node, mark it visited, tracking its previous node, push it onto the stack.
- If no adjacent vertex found for current node, pop up a vertex from stack.
- Repeat step 1 and 2 until stack is empty.

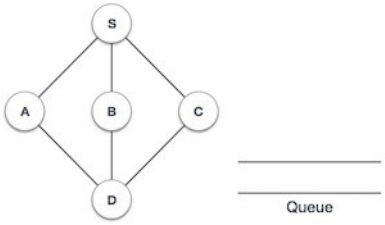
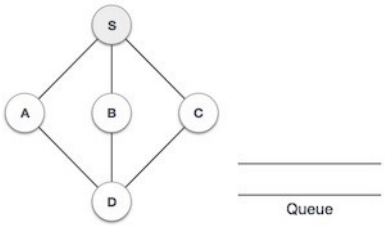
Now, we present a Java implementation of **DFS** algorithm in the following figure.

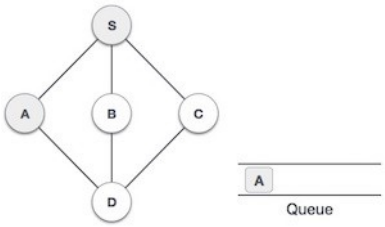
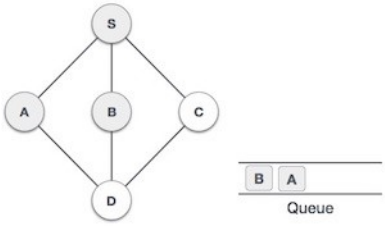
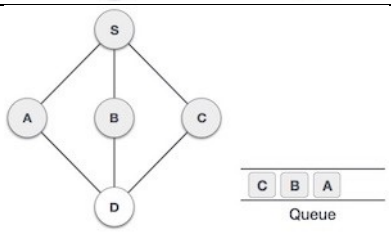
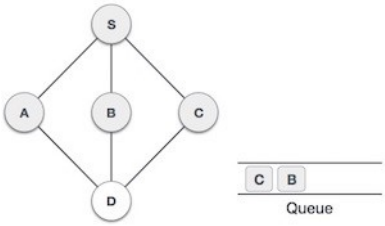
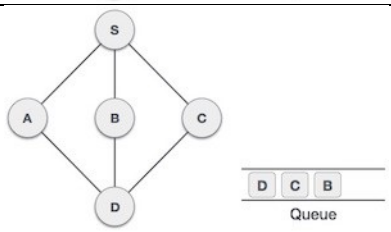
```java
private void _DFS(int start)
{
        Stack<Integer> stack = new Stack<Integer>();
        stack.add(start);

        while(!stack.isEmpty())
        {
                start = stack.pop();
                g.visited[start] = 1;

                for(int i = g.noOfNode - 1; i > 0; i--)
                {
                        if(g.A[start][i] != 0 && g.visited[i] == 0)
                        {
                                stack.push(i);
                                g.tracking[i] = start;
                        }
                }
        }
}
```

Figure 2 DFS algorithm

Implementing Depth-first search algorithm

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. To illustrate the steps in running **BFS** to find a path between two nodes, let's see the example below.

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

3		We then see unvisited adjacent node from **S**. In this case, we have three nodes but alphabetically we choose **A**, mark it visited and enqueue it.	**S→A**
4		Next unvisited adjacent node from **S** is **B**. We mark it visited and enqueue it.	**S→A→B**
5		Next unvisited adjacent node from **S** is **C**. We mark it visited and enqueue it.	**S→A→B→C**
6		Now **S** is left with no unvisited adjacent nodes. So we dequeue and find **A**.	**S→A→B→C**
7		From **A** we have **D** as unvisited adjacent node. We mark it visited and enqueue it.	**S→A→B→C→D**

We can summarize the **BFS** algorithm in the following rules:

- Visit adjacent unvisited vertex of the current node, mark it visited, tracking its previous node, enqueue it.
- If no adjacent vertex found, remove the first vertex from queue, which means, dequeue.
- Repeat step 1 and 2 until queue is empty.

Now, we present a Java implementation of **BFS** algorithm in the following figure.

```java
private void _BFS(int start)
{
	Queue<Integer> queue = new LinkedList<Integer>();
	queue.add(start);

	while(!queue.isEmpty())
	{
		start = queue.remove();
		g.visited[start] = 1;

		for(int i = 0; i < g.noOfNode; i++)
		{
			if(g.A[start][i] != 0 && g.visited[i] == 0)
			{
				queue.add(i);
				g.visited[i] = 1;
				g.tracking[i] = start;
			}
		}
	}
}
```

Figure 3 BFS algorithm

Homework

1. Implement the Graph class and two algorithms, DFS and BFS in Java.