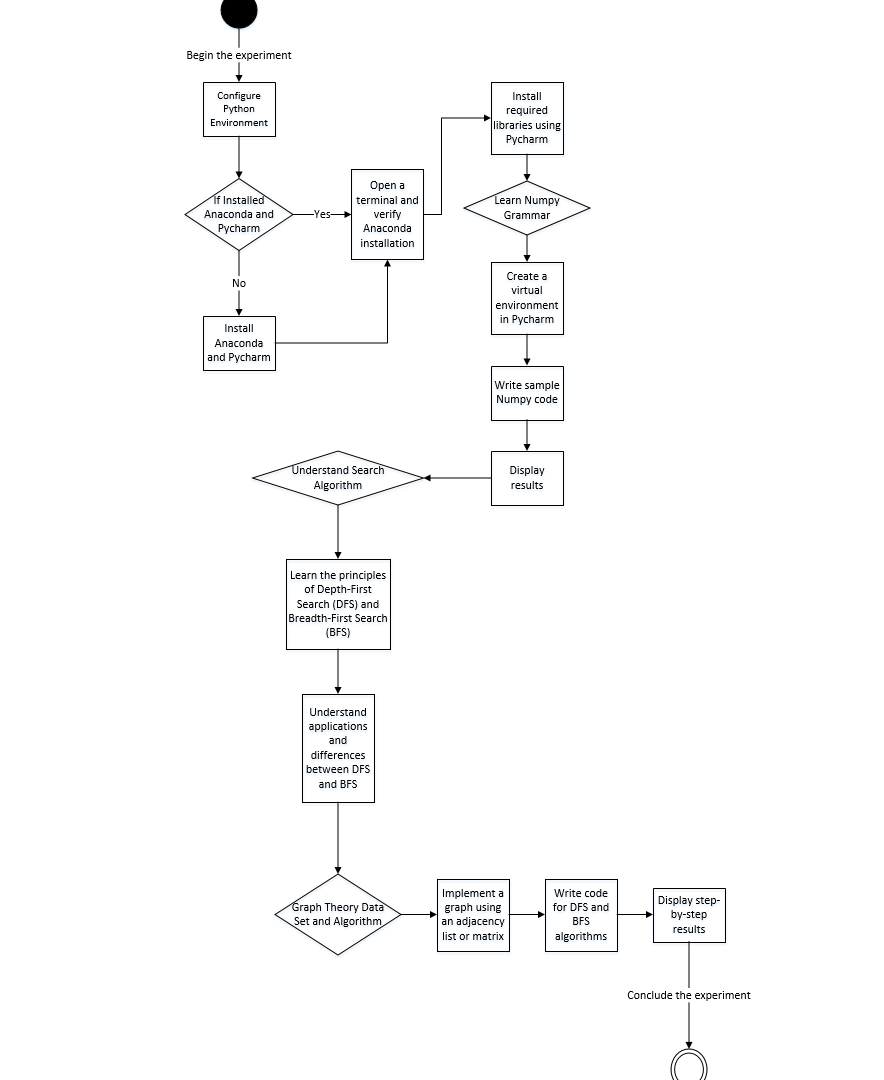
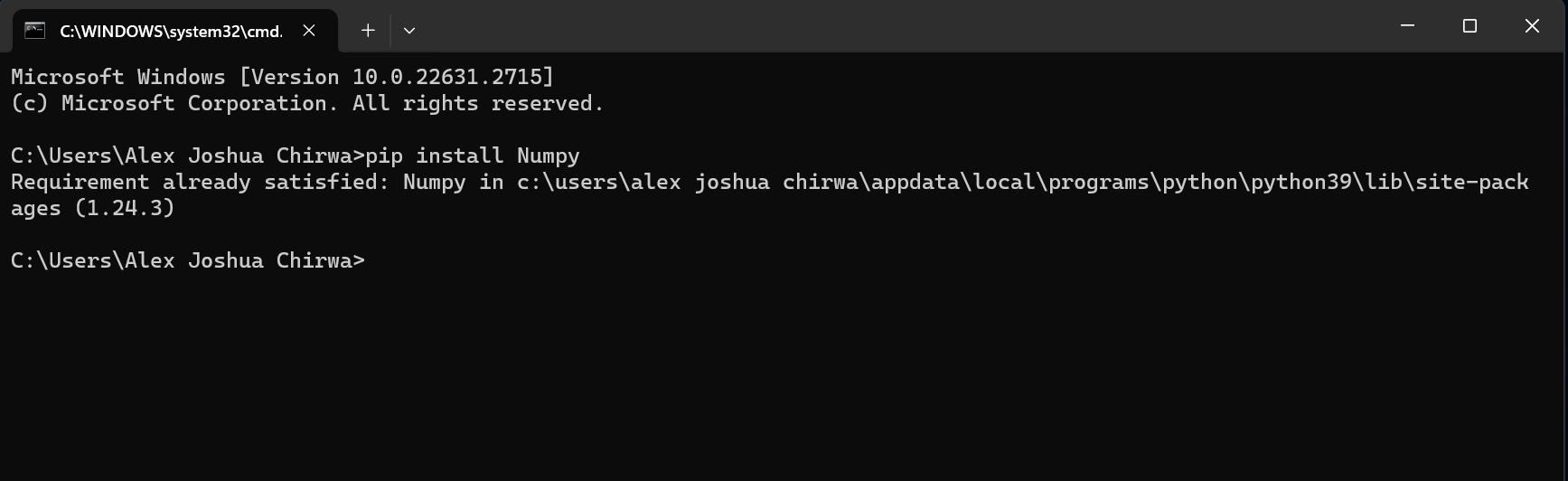
The experimental report gives the content elements in the following order:

1. Procedure flow chart
2. Experimental results and analysis diagram
   1. The configuration process of Python programming environment. Including Anaconda, Pycharm and corresponding algorithm library installation.
   2. Python Numpy syntax learning process.
   3. Explain the principles of breadth-first and depth-first search algorithms.
   4. The code of graph theory data set construction, breadth-first and depth-first search algorithm. Write the code steps and show the results.
   5. Compare the results of different search algorithms.
3. Source code and necessary comments
4. Experimental summary and experience

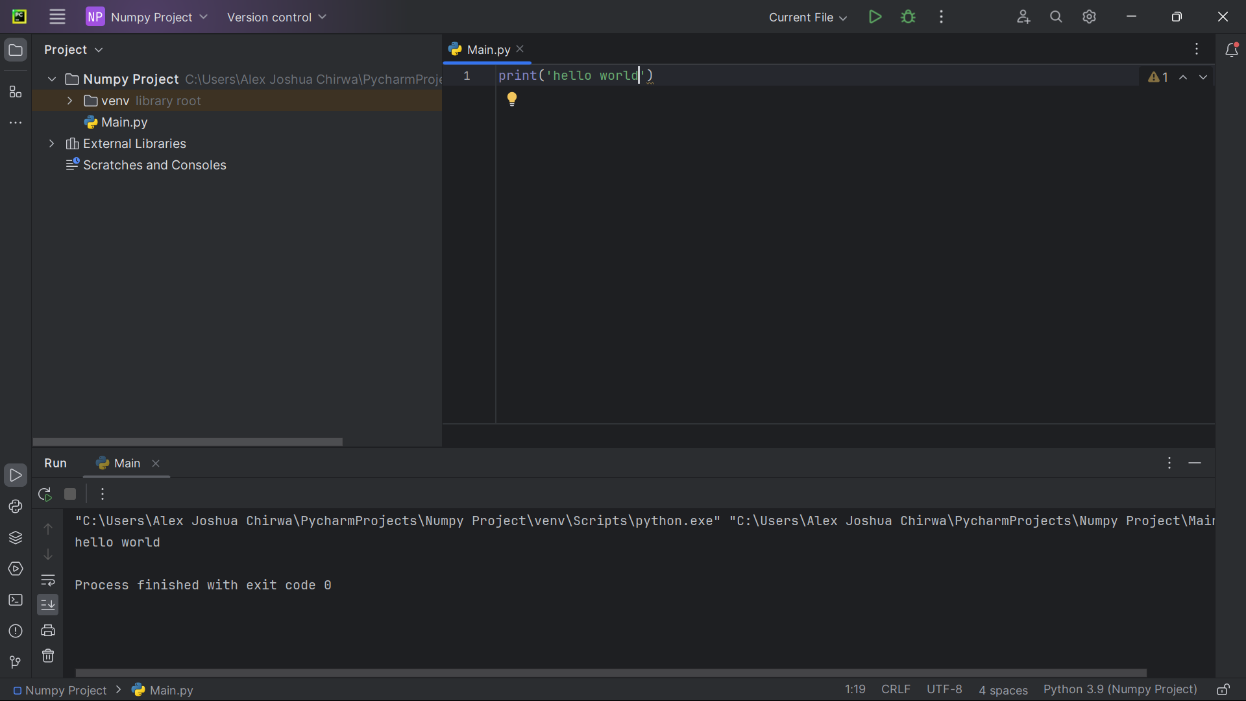


**Installation/Configuration of Anaconda and Pycharm:**

* I already have Anaconda and Pycharm Installed and set up from my previous courses that I had last semester but I can show you, a few screenshots of the available configurations and the already installed Packages for Numpy in terminal.
* To check whether in the above mentioned Pycharm environment has the correctly installed package of Numpy I will open terminal and type in “**pip install Numpy”** to confirm the installation which I will demonstrate below.

****

* Next to see and test if the Pycharm environment is running correctly we will have to test it by typing in a simple code “**print(‘Hello World’)** and see the displayed result in the result terminal



**Principle of Breadth-First and Depth-First Search Algorithms**

**Breadth-First Search Algorithm:**

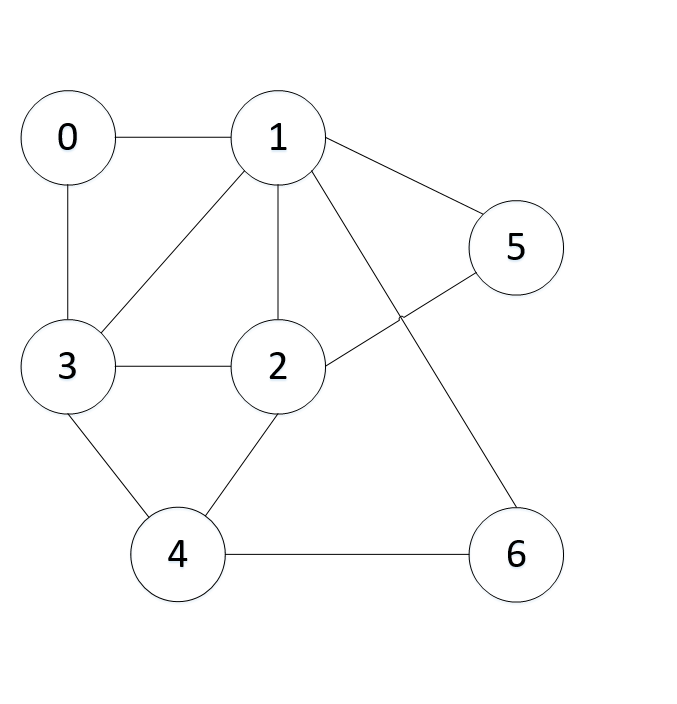
* The BFS algorithm is used to search a tree or a graph data structure for a node that meets a set of criteria.
* It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.
* To avoid processing a node more than once, we divide the vertices into two categories:
* Visited and
* Not visited

**A demonstration on how BFS Algorithms works:**

* Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.
* To do this a queue is used.
* All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

**Illustration:**

**Step1:** initially queue and visited arrays are empty.



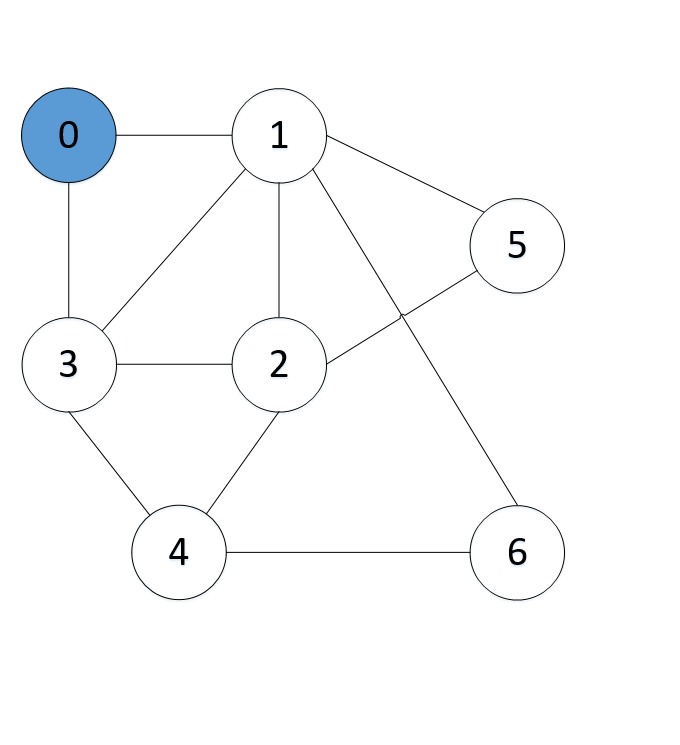
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

Queue

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

**Step2:** Push node 0 into queue and mark it visited.



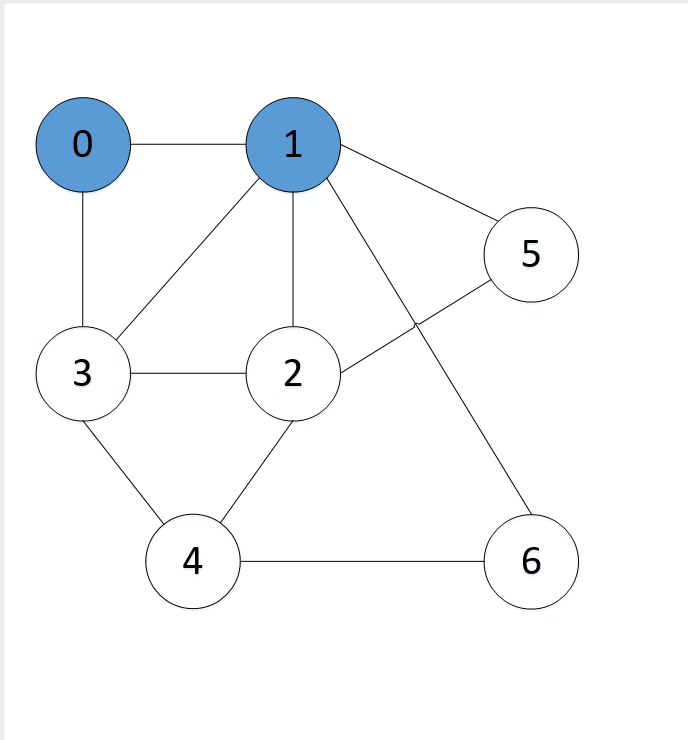
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 |  |  |  |  |  |  |

Queue

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 |  |  |  |  |  |  |

**Step3:** remove node 0 from the front of queue, visit the unvisited neighbors, and push them into queue. In this case, 1 is next up in line as shown below.



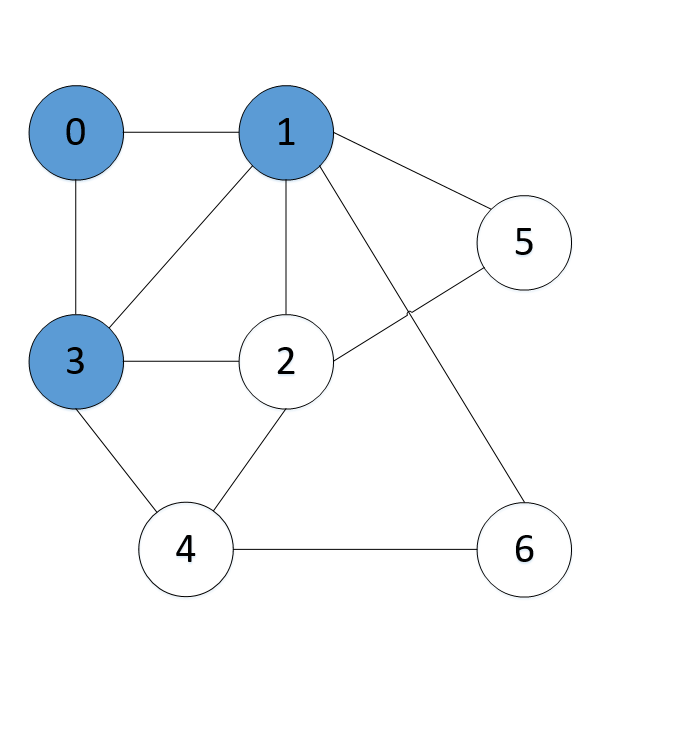
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 |  |  |  |  |  |

Queue

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 |  |  |  |  |  |

**Step4:** remove node 1 from the front of queue, visit the unvisited neighbors, and push them into queue. As we can see that every neighbors of node 1 is visited, so move to the next node that are in the front of the queue.



Visited

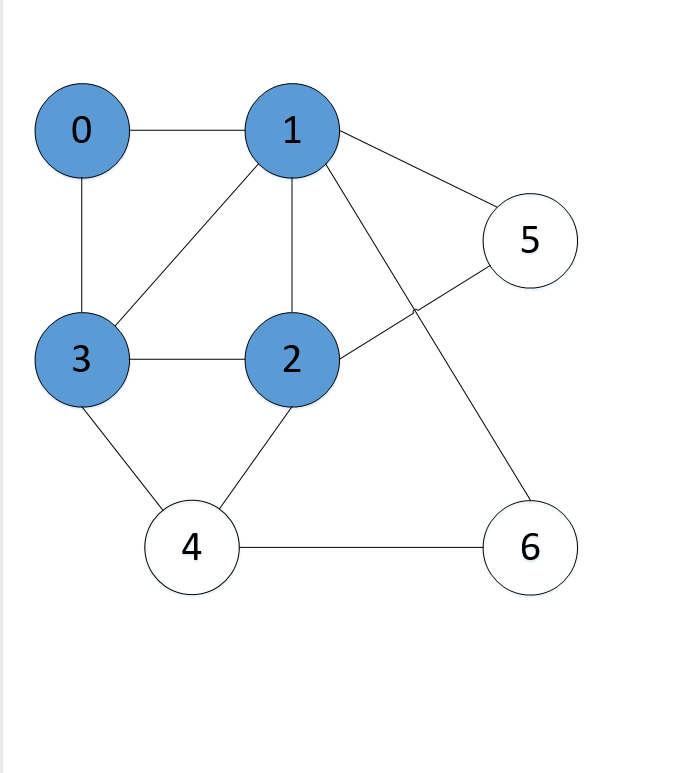
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 |  |  |  |  |

Queue

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 2 | 5 | 6 |  |  |  |

**Step5:** remove node 3 from the front of queue, visit the unvisited neighbors, and push them into queue.

As we can see that every neighbors of node 3 is visited, so move to the next node that are in the front of the queue.



Visited

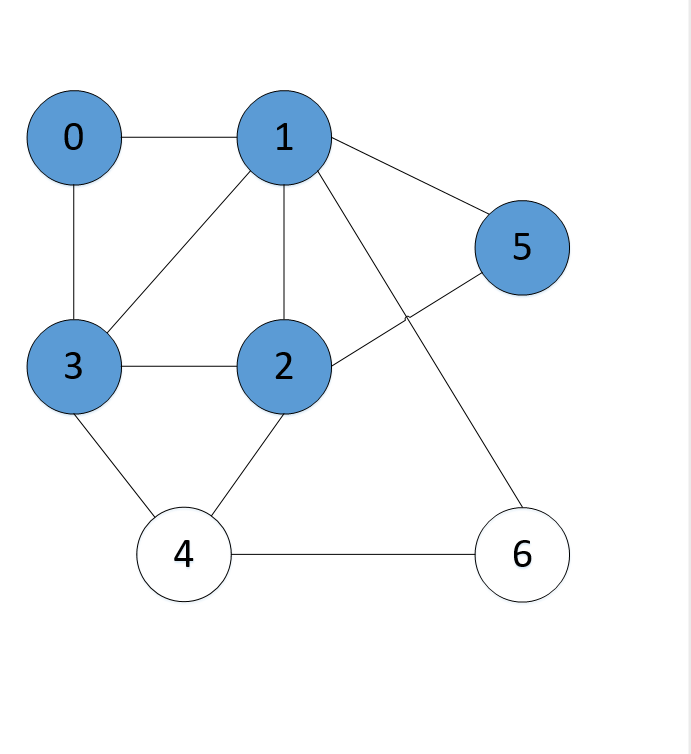
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 |  |  |  |

Queue

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 6 | 4 |  |  |  |

**Step6:** remove node 2 from the front of queue, visit the unvisited neighbors, and push them into queue.

As we can see that every neighbors of node 2 is visited, so move to the next node that are in the front of the queue.



Visited

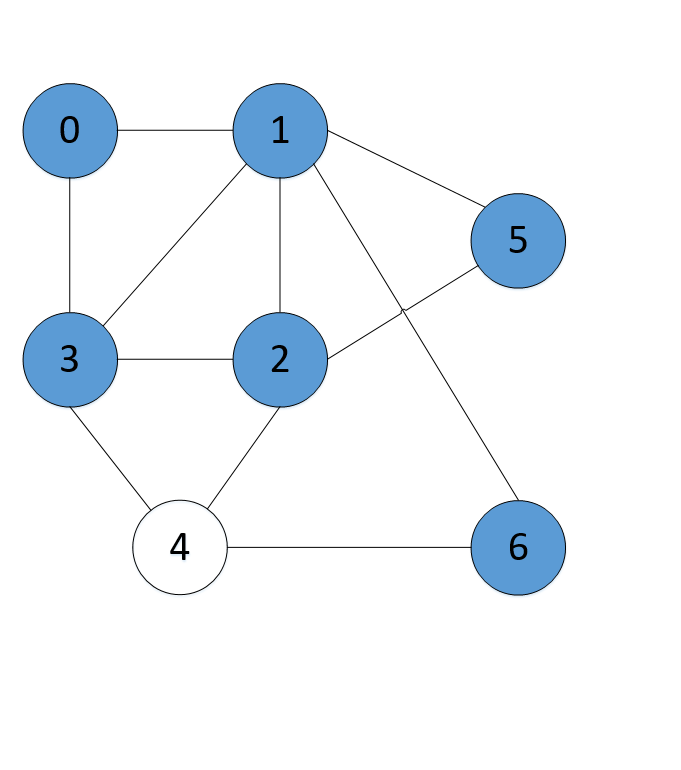
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 5 |  |  |

Queue

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 5 | 6 | 4 |  |  |  |  |

**Step7:** remove node 5 from the front of queue, visit the unvisited neighbors, and push them into queue.

As we can see that every neighbors of node 5 is visited, so move to the next node that are in the front of the queue.



Visited

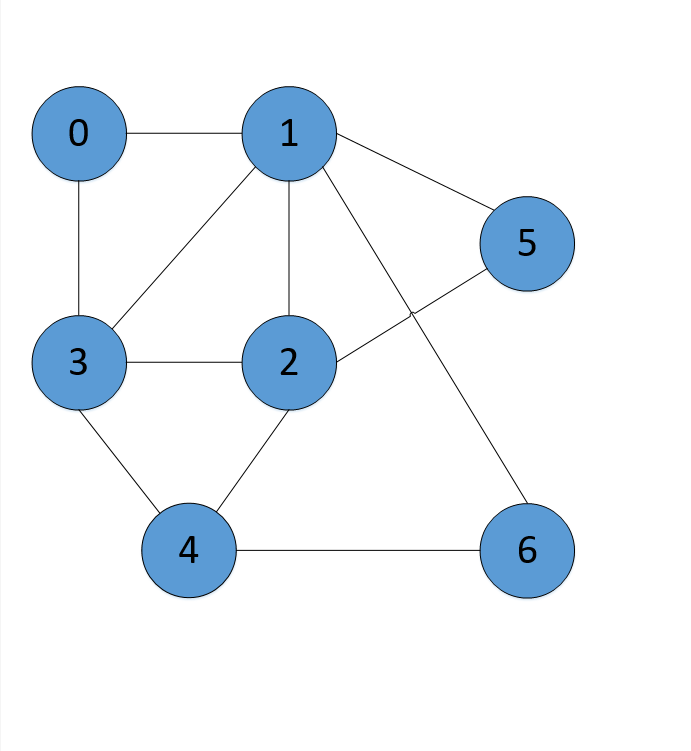
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 5 | 6 |  |

Queue

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 6 | 4 |  |  |  |  |  |

**Step8:** remove node 6 from the front of queue, visit the unvisited neighbors, and push them into queue.

As we can see that every neighbors of node 6 is visited, so move to the next node that are in the front of the queue.



Visited

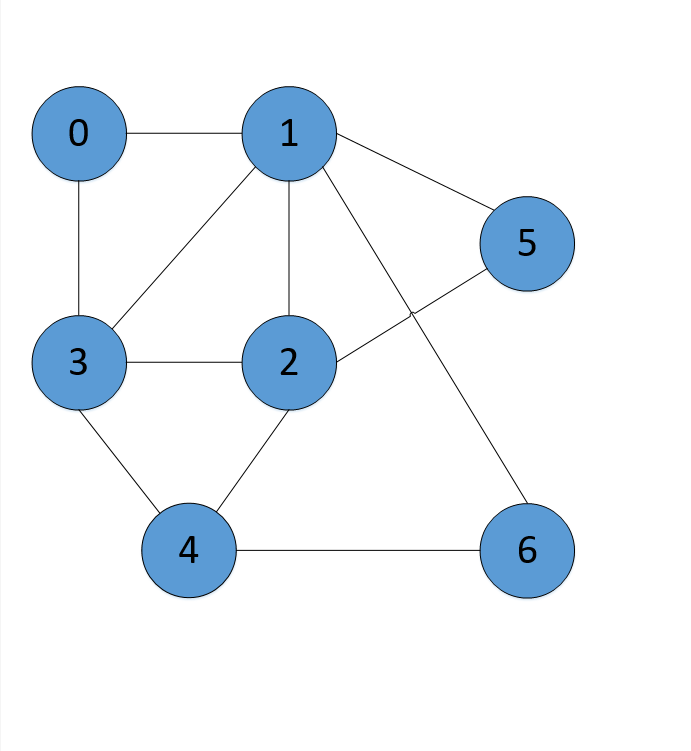
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 5 | 6 | 4 |

Queue

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 |  |  |  |  |  |  |

**Step9:** remove node 4 from the front of queue, visit the unvisited neighbors, and push them into queue.

As we can see that every neighbors of node 4 is visited, so move to the next node that are in the front of the queue.



Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 5 | 6 | 4 |

Queue

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

Now, queue becomes empty, so, terminate this process of iteration.

**Depth-First Search Algorithm**

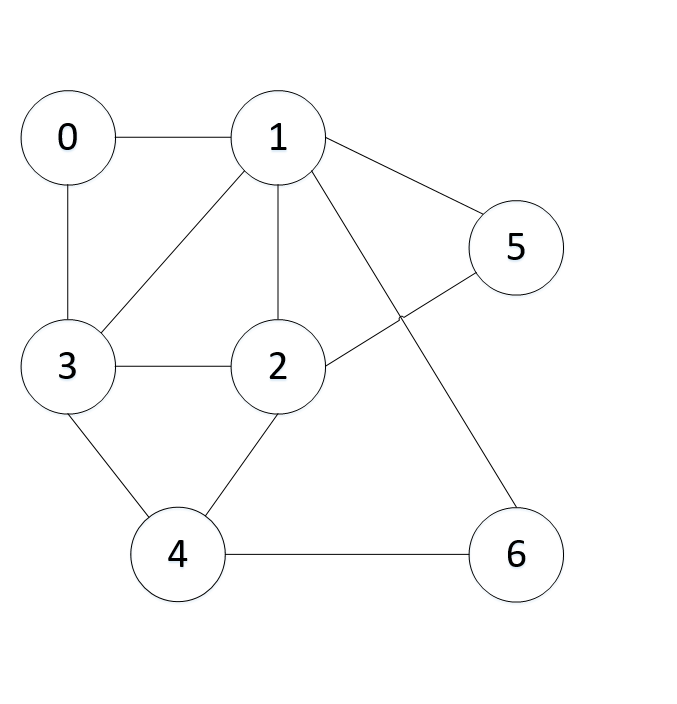
* It is a recursive algorithm for searching tree or graph data structures. It starts at the root node and explores as far as possible along each branch before backtracking.
* DFS uses the stack data structure. It can be implemented easily using recursion and data structures like dictionaries and sets.
* To avoid processing a node more than once, use a Boolean visited array.

**A demonstration on how DFS works:**

* The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

**Illustration:**

**Step1:** initially stack and visited arrays are empty.



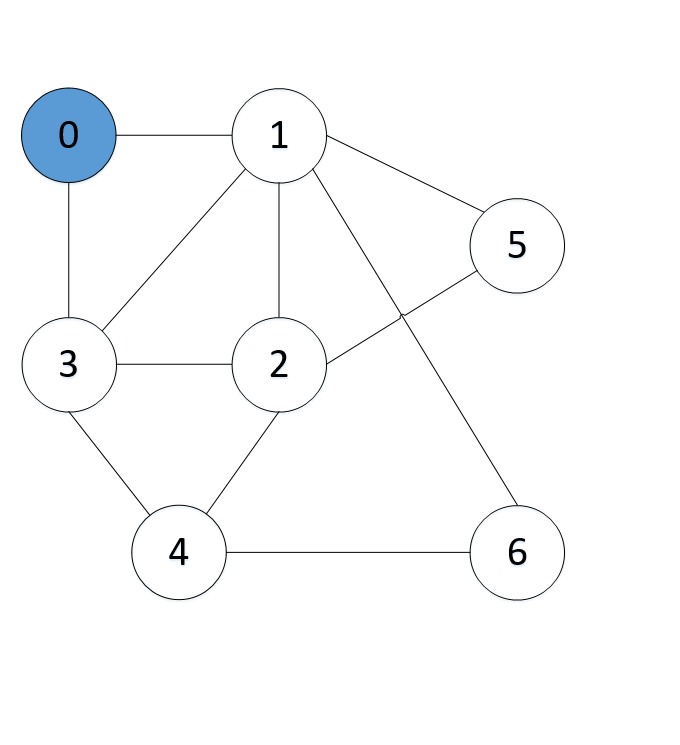
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

**Step2:** visit 0 and put its adjacent nodes that are not visited yet into the stack.



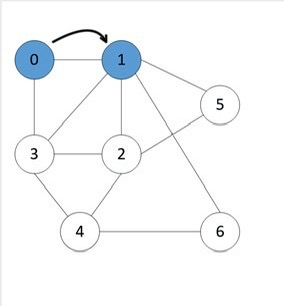
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 |  |  |  |  |  |  |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 |  |  |  |  |

**Step3:** Now, node 1 at the top of the stack, so we visit 1 and put all of its adjacent nodes that are not visited in the stack.



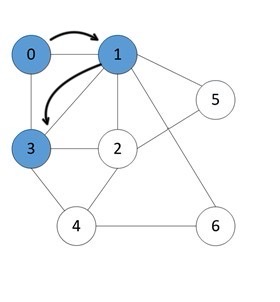
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 |  |  |  |  |  |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 |  |  |  |

**Step4:** Now, node 3 at the top of the stack, so we visit 3, and put all of its adjacent nodes that are not visited, in the stack.



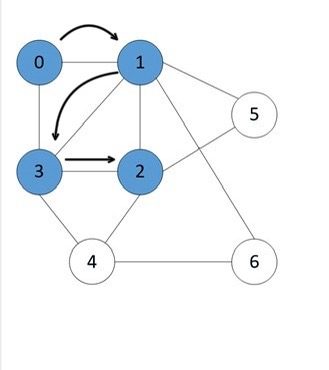
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 |  |  |  |  |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 |  |  |

**Step5:** Now, node 2 at the top of the stack, so we visit 2, and put all of its adjacent nodes that are not visited, in the stack.



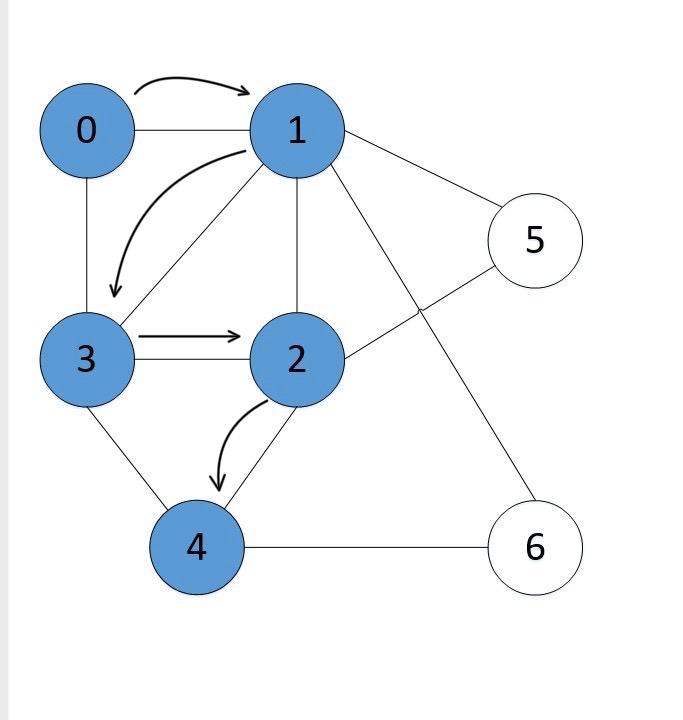
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 |  |  |  |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 |  |

**Step6:** Now, node 4 at the top of the stack, so we visit 4, and put all of its adjacent nodes that are not visited, in the stack.



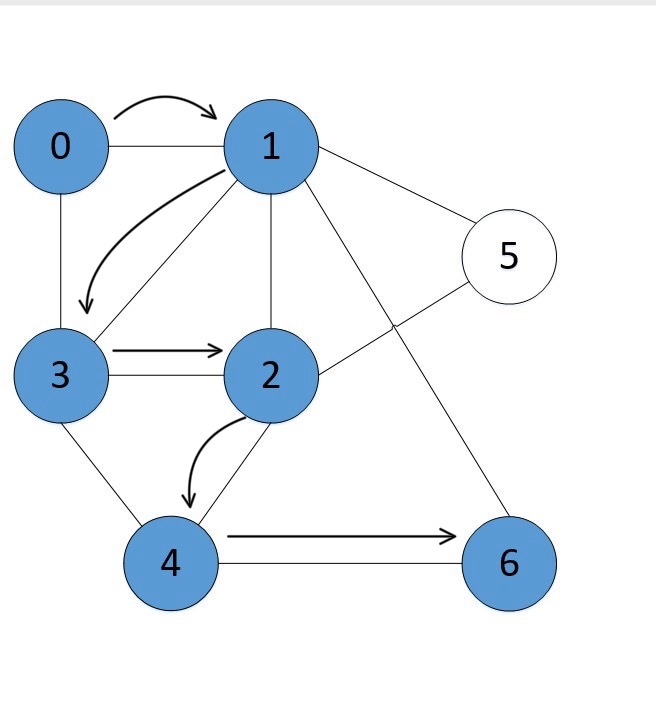
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 |  |  |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 |  |

**Step7:** Now, node 6 at the top of the stack, so we visit 6, and put all of its adjacent nodes that are not visited, in the stack.



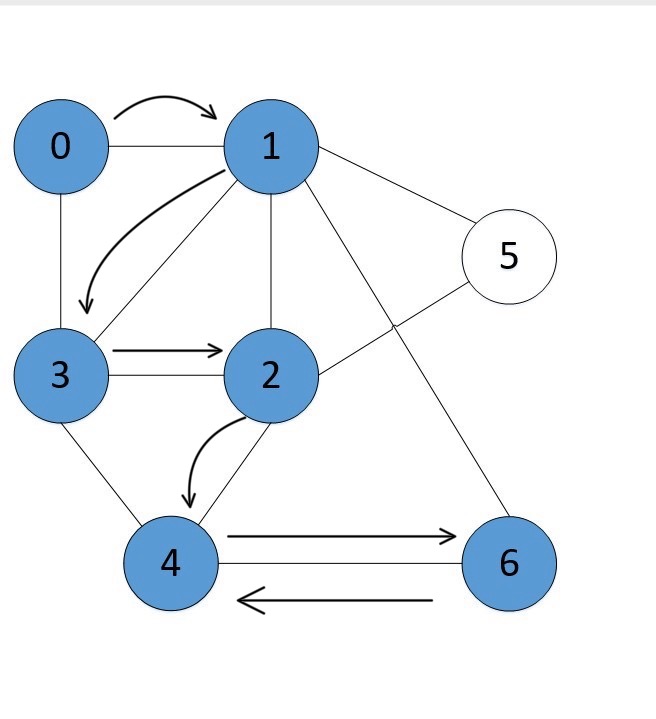
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 |  |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 |  |

**Step8:** Now, at this point node 6 does not have any adjacent nodes that are not visited in the stack, we then begin to backtrack the stack by popping/deleting the nodes from stack starting with 6.



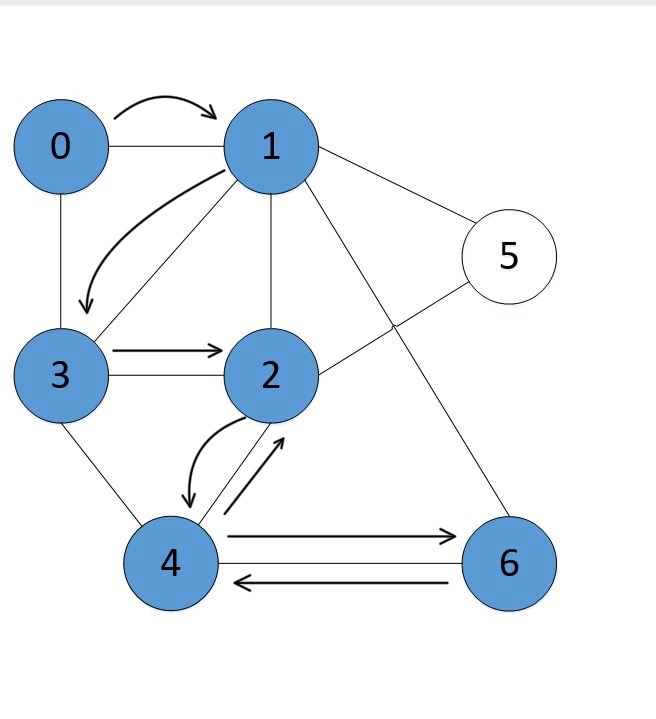
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 |  |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 |  |  |

**Step9:** we continue the back track, pop out 4 from the stack, looking from the unvisited nodes.



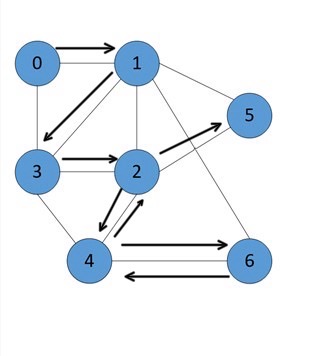
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 |  |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 |  |  |  |

**Step10:** Approach node 2 and check for unvisited nodes, which we then find that, it is node 5 and we add it to stack and visited array.



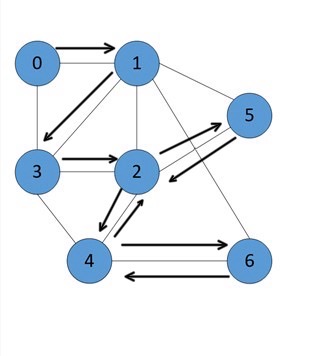
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 | 5 |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 5 |  |  |

**Step11:** we check whether node 5 has unvisited nodes, all the nodes have been visited, we then continue to backtrack the stack until we reach the initial node and the stack becomes empty.



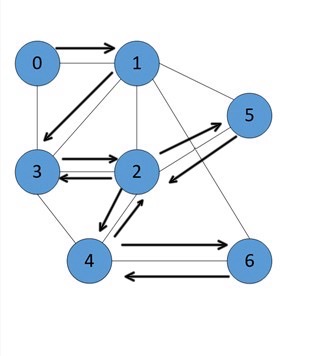
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 | 5 |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 |  |  |  |

**Step12:** pop node 2 from the stack



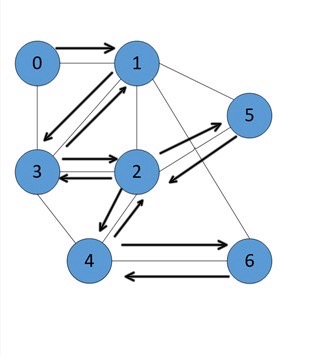
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 | 5 |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 |  |  |  |  |

**Step13:** pop node 3 from the stack



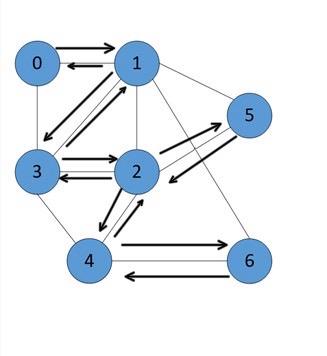
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 | 5 |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 |  |  |  |  |  |

**Step14:** pop node 1



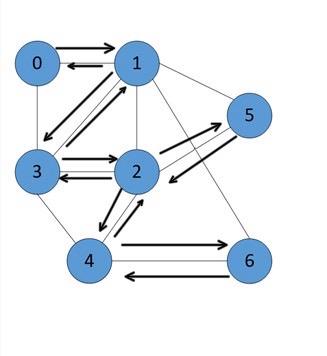
Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 | 5 |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 |  |  |  |  |  |  |

**Step15:** pop 0



Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 | 5 |

Stack

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

**Step16:** the stack is empty, which means we have visited all the nodes and our DFS traversal ends.

Visited

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 3 | 2 | 4 | 6 | 5 |

Stack

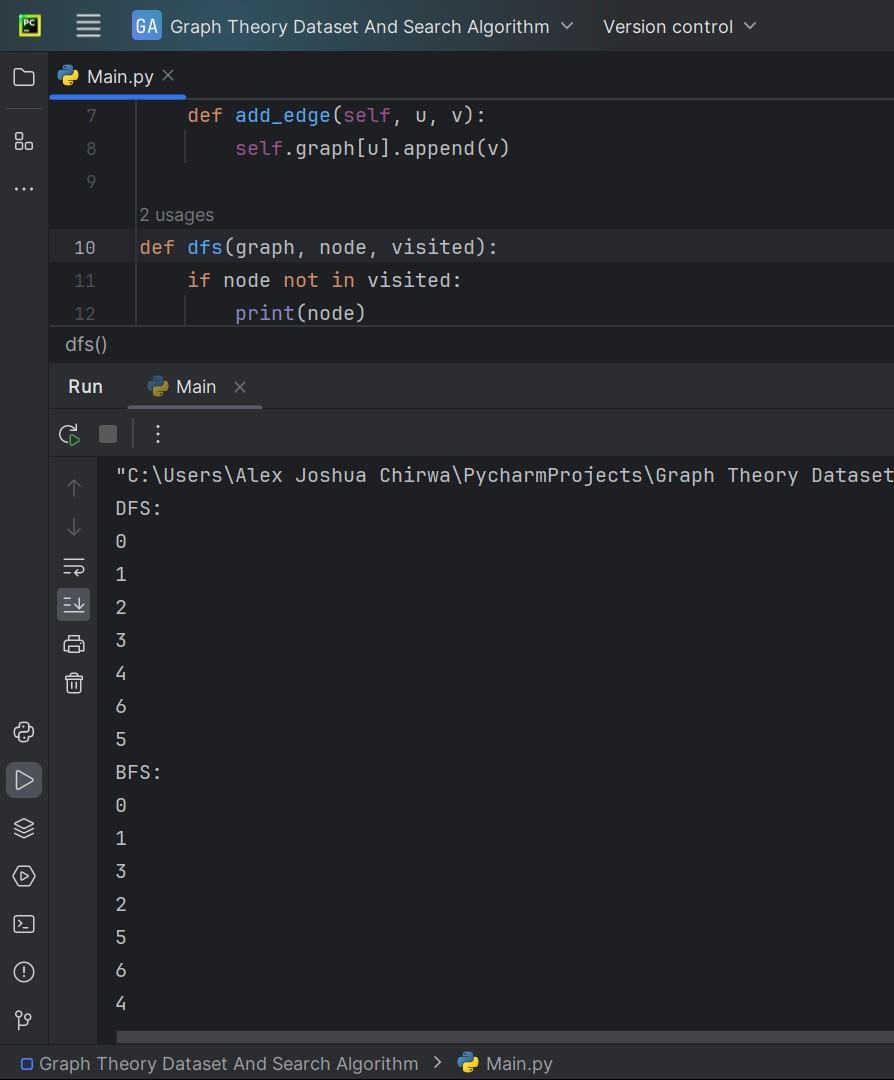
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

**The code of graph theory data set construction**

**Code Implementation:**

from collections import defaultdict  
  
class Graph:  
 def \_\_init\_\_(self):  
 self.graph = defaultdict(list)  
  
 def add\_edge(self, u, v):  
 self.graph[u].append(v)  
  
def dfs(graph, node, visited):  
 if node not in visited:  
 print(node)  
 visited.add(node)  
 for neighbor in graph[node]:  
 dfs(graph, neighbor, visited)  
  
def bfs(graph, start):  
 visited = set()  
 queue = [start]  
  
 while queue:  
 node = queue.pop(0)  
 if node not in visited:  
 print(node)  
 visited.add(node)  
 queue.extend(graph[node])  
  
g = Graph()  
g.add\_edge(0, 1)  
g.add\_edge(0, 3)  
g.add\_edge(1, 2)  
g.add\_edge(1, 5)  
g.add\_edge(1, 6)  
g.add\_edge(2, 3)  
g.add\_edge(2, 4)  
g.add\_edge(2, 5)  
g.add\_edge(3, 2)  
g.add\_edge(3, 4)  
g.add\_edge(4, 2)  
g.add\_edge(4, 6)  
  
  
print('DFS:')  
dfs(g.graph, 0, set())  
  
print('BFS:')  
bfs(g.graph, 0)

**Output:**



**Explanation:**

* The ‘Graph’ class is used to represent an undirected graph using an adjacency list.
* ‘add\_edge’ method is used to add edges to the graph.
* The ‘dfs’ function performs Depth-First Search recursively starting from a given node.
* The ‘bfs’ function performs Breadth-First Search using a queue starting from a given node.
* Usage creates a graph, adds edges, and demonstrates both DFS and BFS starting from node 0.
* The results show the order in which nodes are visited during both DFS and BFS traversals. The order might vary depending on the specific graph structure and the starting node.

**In summary:**

* Depth-First Search and Breadth-First Search are algorithms for searching tree or graph data structures. The choice between the two depends on the problem and graph.
* **Breath-First Search:**
* Method: Starts at the root node and examines all surrounding nodes
* Goal: Finds the shortest path or minimum number of steps to reach a node
* Completeness: complete
* Advantages: Will find a solution if one exists
* **Depth-First Search:**
* Method: Starts with the first node and explores until it finds the desired node or a node with no children
* Goal: Finds all possible paths or exhaustively explores the entire graph
* Completeness: Not guaranteed to find a solution
* Disadvantages: May loop forever if the graph has cycles