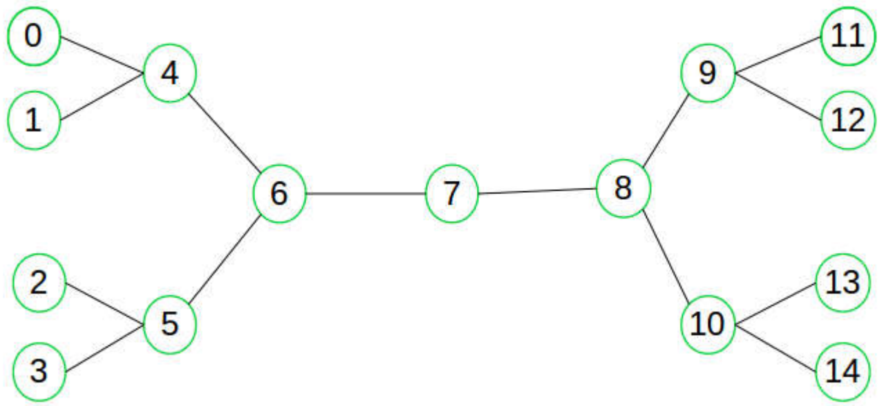
Assignment Group – 60

- **Gupta Kunal Anilbhai**
- **Sachin Baburao Shelke**
- **Vinayak Vaid**
- **Khushi Raj**
- **Ramita Sengupta**

Q4. Explain your strategy in bi-directional search.

Ans:

1. Bidirectional search is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph. It runs two simultaneous searches: one forward from the initial state, and one backward from the goal, stopping when the two meet. The reason for this approach is that in many cases it is faster: for instance, in a simplified model of search problem complexity in which both searches expand a tree with branching factor b, and the distance from start to goal is d, each of the two searches has complexity O(bd/2) (in Big O notation), and the sum of these two search times is much less than the O(bd) complexity that would result from a single search from the beginning to the goal.
2. We can consider bidirectional approach when- Both initial and goal states are unique and completely defined. And the branching factor is exactly the same in both directions.
3. Bidirectional search is complete if BFS is used in both searches. Also, it is optimal if BFS is used for search and paths have uniform cost. Time and space complexity is O(bd/2).
4. For below figure, Suppose we want to find if there exists a path from vertex 0 to vertex 14. Here we can execute two searches, one from vertex 0 and other from vertex 14. When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now. We can clearly see that we have successfully avoided unnecessary exploration.



5. Pseudo Code:

```
BIDIRECTIONAL_SEARCH
Q_I.Insert(x_I) and mark x_I as visited
Q_G.Insert(x_G) and mark x_G as visited
while Q_I not empty and Q_G not empty do
    if Q_I not empty
        x ← Q_I.GetFirst()
        if x = x_G or x ∈ Q_G
            return SUCCESS
        forall u ∈ U(x)
            x' ← f(x, u)
            if x' not visited
                Mark x' as visited
                Q_I.Insert(x')
            else
                Resolve duplicate x'
    if Q_G not empty
        x' ← Q_G.GetFirst()
        if x' = x_I or x' ∈ Q_I
            return SUCCESS
        forall u^-1 ∈ U^-1(x')
            x ← f^-1(x', u^-1)
            if x not visited
                Mark x as visited
                Q_G.Insert(x)
            else
                Resolve duplicate x
return FAILURE
```

6. Conceptual implementation:

```
# Python3 program for Bidirectional BFS
# Search to check path between two vertices

# Class definition for node to
# be added to graph
class AdjacentNode:

    def __init__(self, vertex):
```

```
        self.vertex = vertex
        self.next = None

# BidirectionalSearch implementation
class BidirectionalSearch:

    def __init__(self, vertices):

        # Initialize vertices and
        # graph with vertices
        self.vertices = vertices
        self.graph = [None] * self.vertices

        # Initializing queue for forward
        # and backward search
        self.src_queue = list()
        self.dest_queue = list()

        # Initializing source and
        # destination visited nodes as False
        self.src_visited = [False] * self.vertices
        self.dest_visited = [False] * self.vertices

        # Initializing source and destination
        # parent nodes
        self.src_parent = [None] * self.vertices
        self.dest_parent = [None] * self.vertices

    # Function for adding undirected edge
    def add_edge(self, src, dest):

        # Add edges to graph

        # Add source to destination
        node = AdjacentNode(dest)
        node.next = self.graph[src]
        self.graph[src] = node

        # Since graph is undirected add
        # destination to source
        node = AdjacentNode(src)
        node.next = self.graph[dest]
```

```
        self.graph[dest] = node

    # Function for Breadth First Search
    def bfs(self, direction = 'forward'):

        if direction == 'forward':

            # BFS in forward direction
            current = self.src_queue.pop(0)
            connected_node = self.graph[current]

            while connected_node:
                vertex = connected_node.vertex

                if not self.src_visited[vertex]:
                    self.src_queue.append(vertex)
                    self.src_visited[vertex] = True
                    self.src_parent[vertex] = current

                connected_node = connected_node.next
        else:

            # BFS in backward direction
            current = self.dest_queue.pop(0)
            connected_node = self.graph[current]

            while connected_node:
                vertex = connected_node.vertex

                if not self.dest_visited[vertex]:
                    self.dest_queue.append(vertex)
                    self.dest_visited[vertex] = True
                    self.dest_parent[vertex] = current

                connected_node = connected_node.next

    # Check for intersecting vertex
    def is_intersecting(self):

        # Returns intersecting node
        # if present else -1
        for i in range(self.vertices):
```

```
            if (self.src_visited[i] and
                    self.dest_visited[i]):
                    return i

        return -1

    # Print the path from source to target
    def print_path(self, intersecting_node,
                    src, dest):

        # Print final path from
        # source to destination
        path = list()
        path.append(intersecting_node)
        i = intersecting_node

        while i != src:
            path.append(self.src_parent[i])
            i = self.src_parent[i]

        path = path[::-1]
        i = intersecting_node

        while i != dest:
            path.append(self.dest_parent[i])
            i = self.dest_parent[i]

        print("*****Path*****")
        path = list(map(str, path))

        print(' '.join(path))

    # Function for bidirectional searching
    def bidirectional_search(self, src, dest):

        # Add source to queue and mark
        # visited as True and add its
        # parent as -1
        self.src_queue.append(src)
        self.src_visited[src] = True
        self.src_parent[src] = -1
```

```
        # Add destination to queue and
        # mark visited as True and add
        # its parent as -1
        self.dest_queue.append(dest)
        self.dest_visited[dest] = True
        self.dest_parent[dest] = -1

        while self.src_queue and self.dest_queue:

            # BFS in forward direction from
            # Source Vertex
            self.bfs(direction = 'forward')

            # BFS in reverse direction
            # from Destination Vertex
            self.bfs(direction = 'backward')

            # Check for intersecting vertex
            intersecting_node = self.is_intersecting()

            # If intersecting vertex exists
            # then path from source to
            # destination exists
            if intersecting_node != -1:
                print(f"Path exists between {src} and {dest}")
                print(f"Intersection at : {intersecting_node}")
                self.print_path(intersecting_node,
                                                src, dest)
                exit(0)
        return -1

# Driver code
if __name__ == '__main__':

    # Number of Vertices in graph
    n = 15

    # Source Vertex
    src = 0

    # Destination Vertex
    dest = 14
```

```
# Create a graph
graph = BidirectionalSearch(n)
graph.add_edge(0, 4)
graph.add_edge(1, 4)
graph.add_edge(2, 5)
graph.add_edge(3, 5)
graph.add_edge(4, 6)
graph.add_edge(5, 6)
graph.add_edge(6, 7)
graph.add_edge(7, 8)
graph.add_edge(8, 9)
graph.add_edge(8, 10)
graph.add_edge(9, 11)
graph.add_edge(9, 12)
graph.add_edge(10, 13)
graph.add_edge(10, 14)

out = graph.bidirectional_search(src, dest)

if out == -1:
    print(f"Path does not exist between {src} and {dest}")
```

7. **Output:**