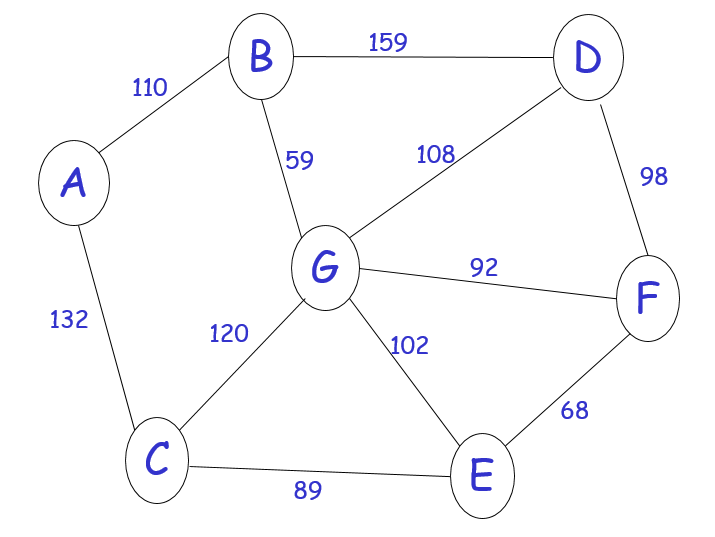
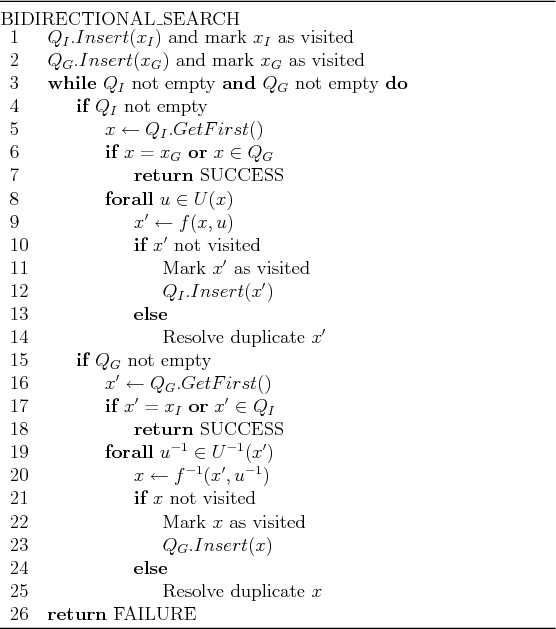
**Assignment Group - 60**

**Justification towards the choices of algorithm/data structures/heuristic/fitness function wherever appropriate written in own words.**

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.



1. **Pseudo Code:**
2. **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}")