BFS Implementation

```
graph = {
    '5': ['3', '7'],
    '3': ['2', '4'],
    '7': ['8'],
    '2': [],
    '4': ['8'],
    '8': []
}
def bfs(visited, graph, node):
 visited = []  # List to keep track of visited nodes
  queue = []  # Initialize a queue
   visited.append(node)
    queue.append(node)
    print("Following is the Breadth-First Search:")
    while queue:
        m = queue.pop(0)  # Dequeue a node
        print(m, end=" ")
        for neighbor in graph[m]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
# Driver Code
bfs([], graph, '5')
# DFS Implementation
graph = {
    '5': ['3', '7'],
    '3': ['2', '4'],
```

```python
    '7': ['8'],
    '2': [],
    '4': ['8'],
    '8': []
}

visited = set()  # Set to keep track of visited nodes

def dfs(visited, graph, node):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("\nFollowing is the Depth-First Search:")
dfs(visited, graph, '5')
```

```
from queue import PriorityQueue

v = 14

graph = [[] for _ in range(v)]

# Function for Best First Search

def best_first_search(actual_Src, target, n):

  visited = [False] * n

  pq = PriorityQueue()

  pq.put((0, actual_Src))  # (cost, node)

  visited[actual_Src] = True

  print("Best First Search Path:")

  while not pq.empty():

    u = pq.get()[1]

    print(u, end=" ")

    if u == target:

      break

    for v, c in graph[u]:

      if not visited[v]:

        visited[v] = True

        pq.put((c, v))

  print()

# Function for adding edges to the graph

def add_edge(x, y, cost):

  graph[x].append((y, cost))

  graph[y].append((x, cost))

# Adding edges
```

```
add_edge(0, 1, 3)
add_edge(0, 2, 6)
add_edge(0, 3, 5)
add_edge(1, 4, 9)
add_edge(1, 5, 8)
add_edge(2, 6, 12)
add_edge(2, 7, 14)
add_edge(3, 8, 7)
add_edge(8, 9, 5)
add_edge(8, 10, 6)
add_edge(9, 11, 1)
add_edge(9, 12, 10)
add_edge(9, 13, 2)
# Running Best First Search
source = 0
target = 9
best_first_search(source, target, v)
import numpy as np
import heapq
class Graph:
    def __init__(self, adjacency_matrix):
        self.adjacency_matrix = adjacency_matrix
        self.num_nodes = len(adjacency_matrix)
    def get_neighbors(self, node):
        return [neighbor for neighbor, is_connected in enumerate(self.adjacency_matrix[node]) if is_connected]
```

```
def memory_bounded_a_star(graph, start, goal, memory_limit):
    visited = set()
    priority_queue = []
    heapq.heappush(priority_queue, (0, start))  # (cost, node)
    memory_usage = 0
    while priority_queue:
        memory_usage = max(memory_usage, len(visited))
        if memory_usage > memory_limit:
            print("Memory limit exceeded!")
            return None
        cost, current_node = heapq.heappop(priority_queue)
        if current_node == goal:
            print("Goal found!")
            return cost
        if current_node not in visited:
            visited.add(current_node)
            for neighbor in graph.get_neighbors(current_node):
                heapq.heappush(priority_queue, (cost + 1, neighbor))
    print("Goal not reachable!")
    return None
if __name__ == "__main__":
    # Define the adjacency matrix of the graph
    adjacency_matrix = np.array([
        [0, 1, 1, 0, 0, 0, 0],
        [1, 0, 0, 1, 1, 0, 0],
```

```
    [1, 0, 0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0, 1, 1],
    [0, 1, 0, 0, 0, 0, 1],
    [0, 0, 1, 1, 0, 0, 1],
    [0, 0, 0, 1, 1, 1, 0]
])
graph = Graph(adjacency_matrix)
start_node = 0
goal_node = 6
memory_limit = 10  # Set the memory limit
result = memory_bounded_a_star(graph, start_node, goal_node, memory_limit)
print("Shortest path cost:", result)
```