AIM:- Implement DFS and BFS for solving travelling salesman problem (graph specified in lab with start and end node.

DFS

```
import time

# Example graph represented as an adjacency matrix
graph = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

# Number of nodes in the graph
N = len(graph)

def tsp_dfs(graph, start):
    visited = [False] * N
    visited[start] = True

    def dfs(v, count, cost, min_cost):
        if count == N and graph[v][start]:
            return min(min_cost, cost + graph[v][start])
        for i in range(N):
            if not visited[i] and graph[v][i]:
                visited[i] = True
                min_cost = dfs(i, count + 1, cost + graph[v][i], min_cost)
                visited[i] = False
        return min_cost

    min_cost = float('inf')
    min_cost = dfs(start, 1, 0, min_cost)
    return min_cost

# Start node for TSP
start_node = 0

# Measure execution time for DFS
start_time = time.time()
dfs_cost = tsp_dfs(graph, start_node)
dfs_time = time.time() - start_time

print("DFS Cost:", dfs_cost)
print("DFS Execution Time:", dfs_time)
```

BFS

```
import collections

def tsp_bfs(graph, start):
    queue = collections.deque([(start, [start], 0)])
    min_cost = float('inf')
    while queue:
        current, path, cost = queue.popleft()
        if len(path) == N:
            if graph[current][start]:
                min_cost = min(min_cost, cost + graph[current][start])
        else:
            for i in range(N):
                if i not in path and graph[current][i]:
                    new_path = path + [i]
                    new_cost = cost + graph[current][i]
                    queue.append((i, new_path, new_cost))
    return min_cost

# Measure execution time for BFS
start_time = time.time()
bfs_cost = tsp_bfs(graph, start_node)
bfs_time = time.time() - start_time

print("BFS Cost:", bfs_cost)
print("BFS Execution Time:", bfs_time)
```

Execution Time

- **DFS**: Tends to be faster in some scenarios because it dives deep into the graph along a single branch before backtracking. This makes it efficient for trees or graphs with lots of depth.
- **BFS**: Can be slower, especially for large graphs, as it explores all neighboring nodes at the current depth before moving on to nodes at the next depth level.

Memory Usage

- **DFS**: Generally uses less memory than BFS because it's not necessary to store all the child pointers at each level.
- **BFS**: Requires more memory, as it keeps track of all the nodes at the current depth level before moving on to the next level.

Complexity and Scalability

- **DFS**: Can get complex and less efficient for large graphs, as it may explore deep, irrelevant branches.
- **BFS**: While more systematic, it can quickly become unfeasible for large graphs due to memory constraints.

Conclusion

In summary, while DFS might be faster and more memory-efficient for certain types of graphs, it is not as thorough as BFS. BFS, though more systematic and thorough, can be slower and less memory-efficient. However, for the TSP, both algorithms are generally inefficient and not commonly used due to the nature of the problem. For TSP, algorithms like the Held-Karp algorithm, branch and bound, or various heuristic methods (like genetic algorithms or simulated annealing) are more commonly employed.