PRACTICAL 1

Objective

WAP to implement DFS and BFS for traversing a graph from source node (S) to goal node (G), where source node and goal node is given by the user as an input.

Program

```
from collections import deque
import timeit

def calculate_distance_bfs(graph, path, state, end):
    visited = set()
    distance = 0
    count = 0
    while state != end:
        for key in list((graph[state]).keys()):
            if key not in visited:
                distance = distance + graph[state][key]
                visited.add(key)
            if key == end:
                break
        if key == end:
            break
        count = count + 1
        state = path[count]
    return distance

def tsp_bfs(graph, start, end):
    visited = set()
    path = []
    distance = 0
    queue = deque([start])
    visited.add(start)
```

```
    while queue:
        vertex = queue.popleft()
        path.append(vertex)

        if vertex == end:
            distance = calculate_distance_bfs(graph, path, start, end)
            return path, distance

        for adj in graph[vertex]:
            if adj not in visited:
                visited.add(adj)
                queue.append(adj)
    return path, distance

def calculate_distance_dfs(graph, path):
    distance = 0
    for i in range(len(path) - 1):
        distance += graph[path[i]][path[i + 1]]
    return distance

def tsp_dfs(graph, start, stop):
    visited = set()
    stack = [start]
    path = []
    distance = 0

    while stack:
        vertex = stack.pop()
        path.append(vertex)
        visited.add(vertex)
        if vertex == stop:
            distance = calculate_distance_dfs(graph, path)
            return path, distance
```

```
        temp_stack = []
        for adj in graph[vertex]:
            if adj not in visited:
                temp_stack.append(adj)
        stack.extend(temp_stack[::-1])
    return path, distance

if __name__ == "__main__":
    graph_1 = {
        "A": {"B": 22, "C": 48, "D": 28},
        "B": {"A": 22, "C": 20, "D": 18},
        "C": {"A": 48, "B": 20, "D": 32},
        "D": {"A": 28, "B": 18, "C": 32},
    }

    graph_2 = {
        "A": {"B": 2, "G": 6},
        "B": {"A": 2, "C": 7, "E": 2},
        "C": {"B": 7, "D": 3, "F": 3},
        "D": {"C": 3, "H": 2},
        "E": {"B": 2, "F": 2, "G": 1},
        "F": {"C": 3, "E": 2, "H": 2},
        "G": {"A": 6, "E": 1, "H": 4},
        "H": {"D": 2, "F": 2, "G": 4},
    }
    start = input("Enter the starting node: ")
    end = input("Enter the ending node: ")

    # DFS
    start_time_dfs = timeit.default_timer()
    path, dist = tsp_dfs(graph_2, start, end)
    execution_time_dfs = timeit.default_timer() - start_time_dfs
```

```
# BFS
start_time_bfs = timeit.default_timer()
path, dist = tsp_bfs(graph_2, start, end)
execution_time_bfs = timeit.default_timer() - start_time_bfs

print("\nDFS Path:", "".join(path))
print("DFS Cost:", dist)
print("DFS Execution Time:", execution_time_dfs)

print("\nBFS Path:", "".join(path))
print("BFS Cost:", dist)
print("BFS Execution Time:", execution_time_bfs)
```

Output

```
Enter the starting node: A
Enter the ending node: F

DFS Path: ABGCEHDF
DFS Cost: 29
DFS Execution Time: 4.05999890062958e-05

BFS Path: ABGCEHDF
BFS Cost: 29
BFS Execution Time: 3.200001083314419e-05
```

Results

According to the results the **DFS** traversing **takes more time** than **BFS** traversing. Hence in the given example BFS outperforms DFS.