```
import itertools

def calculate_total_distance(path, distances):

    total_distance = 0
    for i in range(len(path) - 1):
        total_distance += distances[path[i]][path[i + 1]]
    total_distance += distances[path[-1]][path[0]]  # Return to the starting city
    return total_distance

def traveling_salesman_bruteforce(distances):
    cities = list(distances.keys())
    # Generate all possible permutations of cities
    all_permutations = itertools.permutations(cities)
    # Initialize variables to track the best solution
    best_path = None
    best_distance = float('inf')
    i = 1
    # Iterate through all permutations and calculate total distance
    for permutation in all_permutations:
        current_distance = calculate_total_distance(permutation, distances)
        print("Permutation:",i, " : ", permutation, "back to ", permutation[0], ":", current_distance)
        i+=1
        if current_distance < best_distance:
            best_distance = current_distance
            best_path = permutation

    return best_path, best_distance

# Your distances dictionary
distances = {
    'City1': {'City2': 10, 'City3': 15, 'City4': 20},
    'City2': {'City1': 10, 'City3': 35, 'City4': 25},
    'City3': {'City1': 15, 'City2': 35, 'City4': 30},
    'City4': {'City1': 20, 'City2': 25, 'City3': 30}
}


best_path, best_distance = traveling_salesman_bruteforce(distances)
print("Best Path:", best_path, " back to ", best_path[0])
print("Best Distance:", best_distance)
```

```
Permutation: 1  :  ('City1', 'City2', 'City3', 'City4') back to  City1 : 95
Permutation: 2  :  ('City1', 'City2', 'City4', 'City3') back to  City1 : 80
Permutation: 3  :  ('City1', 'City3', 'City2', 'City4') back to  City1 : 95
Permutation: 4  :  ('City1', 'City3', 'City4', 'City2') back to  City1 : 80
Permutation: 5  :  ('City1', 'City4', 'City2', 'City3') back to  City1 : 95
Permutation: 6  :  ('City1', 'City4', 'City3', 'City2') back to  City1 : 95
Permutation: 7  :  ('City2', 'City1', 'City3', 'City4') back to  City2 : 80
Permutation: 8  :  ('City2', 'City1', 'City4', 'City3') back to  City2 : 95
Permutation: 9  :  ('City2', 'City3', 'City1', 'City4') back to  City2 : 95
Permutation: 10  :  ('City2', 'City3', 'City4', 'City1') back to  City2 : 95
Permutation: 11  :  ('City2', 'City4', 'City1', 'City3') back to  City2 : 95
Permutation: 12  :  ('City2', 'City4', 'City3', 'City1') back to  City2 : 80
Permutation: 13  :  ('City3', 'City1', 'City2', 'City4') back to  City3 : 80
Permutation: 14  :  ('City3', 'City1', 'City4', 'City2') back to  City3 : 95
Permutation: 15  :  ('City3', 'City2', 'City1', 'City4') back to  City3 : 95
Permutation: 16  :  ('City3', 'City2', 'City4', 'City1') back to  City3 : 95
Permutation: 17  :  ('City3', 'City4', 'City1', 'City2') back to  City3 : 95
Permutation: 18  :  ('City3', 'City4', 'City2', 'City1') back to  City3 : 80
Permutation: 19  :  ('City4', 'City1', 'City2', 'City3') back to  City4 : 95
Permutation: 20  :  ('City4', 'City1', 'City3', 'City2') back to  City4 : 95
Permutation: 21  :  ('City4', 'City2', 'City1', 'City3') back to  City4 : 80
Permutation: 22  :  ('City4', 'City2', 'City3', 'City1') back to  City4 : 95
Permutation: 23  :  ('City4', 'City3', 'City1', 'City2') back to  City4 : 80
Permutation: 24  :  ('City4', 'City3', 'City2', 'City1') back to  City4 : 95
Best Path: ('City1', 'City2', 'City4', 'City3')  back to  City1
Best Distance: 80
```

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, start, visited):
        visited[start] = True
        print(start, end=" ")

        for neighbor in self.graph[start]:
            if not visited[neighbor]:
                self.dfs(neighbor, visited)

# Example usage for a graph
graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)

visited = [False] * len(graph.graph)
print("DFS starting from vertex 0:")
graph.dfs(0, visited)
```

```
DFS starting from vertex 0:
0 1 2 3
```

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

def dfs_tree(node):
    if node is None:
        return

    print(node.value, end=" ")

    for child in node.children:
        dfs_tree(child)

# Example usage for a tree
root = TreeNode(1)
root.children = [TreeNode(2), TreeNode(3)]
root.children[0].children = [TreeNode(4), TreeNode(5)]

print("DFS starting from the root:")
dfs_tree(root)
```

```
<__main__.TreeNode object at 0x7fa2baf1f490>
DFS starting from the root:
1 2 4 5 3
```

```
from collections import deque

class PuzzleNode:
    def __init__(self, state, parent=None, action=None):
        self.state = state
        self.parent = parent
        self.action = action

    def __eq__(self, other):
        return self.state == other.state

    def __hash__(self):
        return hash(tuple(self.state))

def get_blank_position(state):
```

```
def get_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def get_neighbors(node):
    i, j = get_blank_position(node.state)
    neighbors = []

    for move in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
        ni, nj = i + move[0], j + move[1]

        if 0 <= ni < 3 and 0 <= nj < 3:
            new_state = [list(row) for row in node.state]
            new_state[i][j], new_state[ni][nj] = new_state[ni][nj], new_state[i][j]
            neighbors.append(PuzzleNode(tuple(map(tuple, new_state)), node, move))

    return neighbors

def print_solution(node):
    path = []
    while node is not None:
        path.append((node.state, node.action))
        node = node.parent

    path.reverse()

    for step in path:
        print("Move:", step[1])
        print_state(step[0])
        print("")

def print_state(state):
    for row in state:
        print(row)

def dfs(start_state, goal_state):
    start_node = PuzzleNode(tuple(map(tuple, start_state)))
    goal_node = PuzzleNode(tuple(map(tuple, goal_state)))

    stack = [start_node]
    visited = set()

    while stack:
        current_node = stack.pop()

        if current_node == goal_node:
            print("DFS Solution:")
            print_solution(current_node)
            return True

        visited.add(current_node)

        neighbors = get_neighbors(current_node)
        for neighbor in neighbors:
            if neighbor not in visited:
                stack.append(neighbor)

    print("No solution found.")
    return False

def bfs(start_state, goal_state):
    start_node = PuzzleNode(tuple(map(tuple, start_state)))
    goal_node = PuzzleNode(tuple(map(tuple, goal_state)))

    queue = deque([start_node])
    visited = set()

    while queue:
        current_node = queue.popleft()

        if current_node == goal_node:
            print("BFS Solution:")
            print_solution(current_node)
            return True

        visited.add(current_node)
```

```
        neighbors = get_neighbors(current_node)
        for neighbor in neighbors:
            if neighbor not in visited:
                queue.append(neighbor)

    print("No solution found.")
    return False

# Example usage:
initial_state = [
    [1, 2, 3],
    [4, 0, 6],
    [7, 5, 8]
]

goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

dfs(initial_state, goal_state)
bfs(initial_state, goal_state)
```

```
DFS Solution:
Move: None
(1, 2, 3)
(4, 0, 6)
(7, 5, 8)

Move: (-1, 0)
(1, 0, 3)
(4, 2, 6)
(7, 5, 8)

Move: (0, -1)
(0, 1, 3)
(4, 2, 6)
(7, 5, 8)

Move: (1, 0)
(4, 1, 3)
(0, 2, 6)
(7, 5, 8)

Move: (1, 0)
(4, 1, 3)
(7, 2, 6)
(0, 5, 8)

Move: (0, 1)
(4, 1, 3)
(7, 2, 6)
(5, 0, 8)

Move: (-1, 0)
(4, 1, 3)
(7, 0, 6)
(5, 2, 8)

Move: (-1, 0)
(4, 0, 3)
(7, 1, 6)
(5, 2, 8)

Move: (0, -1)
(0, 4, 3)
(7, 1, 6)
(5, 2, 8)

Move: (1, 0)
(7, 4, 3)
(0, 1, 6)
(5, 2, 8)

Move: (1, 0)
(7, 4, 3)
(5, 1, 6)
(0, 2, 8)
```

```
Move: (0, 1)
(7, 4, 3)
```