```python
#Write a program that, given a graph with costs and two vertices, finds a
#lowest cost walk between the given vertices, or prints a message if there are negative cost cycles accessible from
#the starting vertex. The program will use a matrix defined as w[x,k]=the cost of the lowest cost walk from s to x and of
#length equal to k, where s is the starting vertex

from service import GraphServices, DirectedGraph
INF = float('inf')

def init_distances(d:list, graph:DirectedGraph):
    for v1, v2, w in graph.edges_iterator():
        d[v2].append((v1, w))

# computes and returns the lowest cost between 2 vertices
def lowest_cost_walk(d:list, n:int, v1:int, v2:int) -> tuple:
    # w[k][x] the cost of minimum cost walk of length k from s to x, or ∞ if no such walk exists.
    w = [[INF] * n for _ in range(2)]  # only keep 2 rows (current row and previous one) for optimization,
                                       #since we only need to know the information of the last row at the end

    w[0][v1] = 0  # starting node has cost 0 at step 0
    prev_node = [[-1] * n for _ in range(2)] # prev_node[cur][x] = y means: best path to x at length k comes from y
    prev_node[0][v1] = -1

    for k in range(1, n + 1):
        curr = k % 2          # 0 or 1 — the "current" row
        prev = (k - 1) % 2  # previous row

        for x in range(n):
            w[curr][x] = w[prev][x]
            prev_node[curr][x] = prev_node[prev][x]
            for y, cost in d[x]:  # inbound edges y → x
                if w[prev][y] < INF and w[curr][x] > w[prev][y] + cost:
                    w[curr][x] = w[prev][y] + cost
                    prev_node[curr][x] = y

        if k == n:
            for x in range(n):
                if w[curr][x] < w[prev][x]:
                    print(f"Relaxed at step {k}: node {x}, prev={w[prev][x]}, curr={w[curr][x]}")
                    raise Exception("Negative cost cycle detected.\n")

    if w[curr][v2] == INF:
        raise Exception(f"No path found {v1} from to {v2}.\n")

    return (w[curr][v2], prev_node)

# finds the lowest cost path between 2 vertices
def find_min_path(v2:int, n:int, prev_node:list) -> list:
    path = []
    node = v2
    step = n % 2  # final DP step used
    visited = set()

    while node != -1:
        if node in visited:
            raise Exception("Cycle detected in path reconstruction.\n")
        visited.add(node)
        path.append(node)
        node = prev_node[step][node]

    path.reverse()
    return path

# used to solve the entire problem using the functions above
def solve():
    try:
        file_name = "graph1k.txt"
        serv = GraphServices()
        serv.read_graph_from_file(file_name)
        n = serv.graph.get_nr_of_vertices()
        d = [[] for _ in range(n)] # d[x] = list of inbound edges of x (as tuples of form (y, w), meaning that I can go from y to x with a cost of w)
        init_distances(d, serv.graph)

        v1, v2 = tuple(input("Insert both nodes here: ").split())
        v1 = int(v1)
        v2 = int(v2)

        lcw, lcw_path = lowest_cost_walk(d, n, v1, v2)
        print(f"Minimum cost from {v1} to {v2} is {lcw}.")
        print(f"Minimum cost path: {find_min_path(v2, n, lcw_path)}")
    except Exception as e:
        print(str(e))

solve()
```