```python
# PB 5: Given a digraph with costs and two vertices,
#  find a minimum cost path between them
# (negative cost cycles may exist in the graph).


def read_graph(filename:str) -> list:
    with open(filename, 'r') as file:
        n, m = file.readline().split(' ')
        n = int(n.strip())
        m = int(m.strip())

        graph = []
        while m > 0:
            v1, v2, cost = file.readline().split(' ')
            v1 = int(v1.strip())
            v2 = int(v2.strip())
            cost = int(cost.strip())
            graph.append((v1, v2, cost))
            m -= 1

    return graph

def min_cost_path(graph:list, v_src:int, v_dest:int) -> int:
    n = len(graph)
    dist = [float('inf')] * n # no vertex has been visited
    dist[v_src] = 0 # distance from the source vertex to itself

    for i in range(n):
        for u, v, w in graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]: # if we find a better path from the source vertex to vertex v (by visiting
                                                                 # vertex u first, then v) then we can update the cost from the source vertex
                                                                 # to vertex v

                if i == n - 1: # if after n iterations we can still find a better path, it means that we have a negative cost cycle because
                               # a path can have at most n - 1 edges, where n = the number of vertices
                    raise Exception("Negative cost cycles detected!\n")

                dist[v] = dist[u] + w

    return dist[v_dest] # the minimum cost of the optimal path going from the source vertex to the destination vertex

def find_path(graph:list, path:list, visited:list, dist:int, v_src:int, v_dest:int, found:list):
    if dist == 0 and v_src != v_dest: # we get the distance we want, but not for the required pair of vertices
        visited[v_src] = False # we might want to revisit this again from some other part of the graph
        return

    if dist == 0 and v_src == v_dest: # we got from the source vertex to the destination vertex with the cost that we wanted => we can stop
        found[0] = True
        return

    visited[v_src] = True # we mark the current vertex as visited so that we avoid cycles
    for u, v, w in graph:
        if u == v_src and not visited[v]: # looking at the edges of from v_src -> v
            path.append(v)
            find_path(graph, path, visited, dist - w, v, v_dest, found) # we go to vertex v, append v in the path and check if it's the path that we want
            if found[0] == True: # if we have found the path that we wanted we can stop
                return

            path.pop() # in case the path is not valid, remove the last element from it and look at other edges v_src -> v
    visited[v_src] = False # in case we return from this vertex, we still might want to revisit this again from some other part of the graph


def solve():
    try:
        digraph = read_graph("graph1.txt")
        v_src, v_dest = input("Insert the vertices here: ").split(' ')
        v_src = int(v_src)
        v_dest = int(v_dest)

        mcp = min_cost_path(digraph, v_src, v_dest)
        path = [v_src]
        found = [False]
        visited = [False] * len(digraph)
        find_path(digraph, path, visited, mcp, v_src, v_dest, found)

        print(f"The minimum cost path from {v_src} to {v_dest} is {path}, with cost of {mcp}.\n")
    except Exception as e:
        print(str(e))

solve()
```