```python
def lowest_cost_walk(graph, start_vertex, end_vertex):
    """
    Returns the lowest cost walk in the graph from start_vertex to end_vertex using Bellman-Ford algorithm
    The program will use a matrix defined as d[x,k]=the cost of the lowest cost walk from s to x and of length at most k, where s is the starting vertex.
    :param graph: the graph
    :param start_vertex: the start vertex
    :param end_vertex: the end vertex
    :return: the lowest cost walk in the graph from start_vertex to end_vertex
    """
    # The program will use a matrix defined as d[x,k]=the cost of the lowest cost walk from s to x and of length at most k, where s is the starting vertex.
    # The program will also use a matrix defined as p[x,k]=the predecessor of x in the lowest cost walk from s to x and of length at most k, where s is the starting vertex.

    # initialising the distances with infinity
    infinity = 9999999999
    n = graph.get_nr_of_vertices()
    d = [[infinity for i in graph.vertices_iterator()] for j in graph.vertices_iterator()]
    p = [[None for i in graph.vertices_iterator()] for j in graph.vertices_iterator()]

    # initialising the distances of the start_vertex with 0
    d[start_vertex][0] = 0

    # Bellman-Ford algorithm
    for k in range(1, n):
        for x in graph.vertices_iterator():
            d[x][k] = d[x][k-1]
            p[x][k] = p[x][k-1]
            for y in graph.inbound_iterator(x):
                if d[y][k-1] + graph.get_cost(y, x) < d[x][k]:
                    d[x][k] = d[y][k-1] + graph.get_cost(y, x)
                    p[x][k] = y
    # checking for negative cost cycles
    for x in range(n):
        if d[x][n - 1] != d[x][n - 2]:
            raise NegativeCycleException("The graph contains anegative cost cycle!")
    if d[end_vertex][n - 1] == infinity:
        raise PathDoesNotExistException("There is no path between the given vertices!")

    # reconstructing the path
    path = []
    current_vertex = end_vertex
    k = n - 1
    while current_vertex is not None:
        path.append(current_vertex)
        current_vertex = p[current_vertex][k]
        k -= 1
    # reversing the path
    path.reverse()
    return d[end_vertex][n - 1], path
```