```
def ford_lowest_cost_walk(given_graph,vertex1,vertex2):
    """
    Function that computes the lowest cost of a walk from source to
target and builds the path between them,using
    Bellman Ford algorithm. If there exists a negative cost cycle the
cost of the path will be None.
    """
    distance={}
    predecessor = {}

    if vertex1 == vertex2:
        return 0, [vertex1]

    # initialization for the distance and predecessor dictionaries
    for vertex in given_graph.vertices:
        distance[vertex] = 99999999
        predecessor[vertex] = -1

    # we relax the edges vertices_number - 1 times
    distance[vertex1]=0
    changed = True
    iteration = 1
    while changed and iteration < given_graph.get_number_of_vertices:
        # if the distances array doesn't change for the next iteration,
it means we have reached the end
        changed = False
        # we parse through the edges and update the distances array
        for edge in given_graph.get_edges_costs.keys():
            # if the newly calculated distance is smaller than the
existing one, we update it
            # distance[edge[1]] - the current distance
            # distance[edge[0]] - the distance so far
            if distance[edge[1]] > distance[edge[0]] +
given_graph.get_edges_costs[edge]:
                distance[edge[1]] = distance[edge[0]] +
given_graph.get_edges_costs[edge]
                predecessor[edge[1]] = edge[0]  # we need to retain for
each vertex the 'best' predecessor
                changed = True
        iteration += 1

    # Then, we check for negative cost cycle
    # if the iteration number is smaller than the number of vertices
then there can't be a negative cost cycle
    # otherwise do one more iteration over the edges and if a change was
found then there is a negative cost cycle

    if iteration == given_graph.get_number_of_vertices:
        changed = False
        for edge in given_graph.get_edges_costs.keys():
            if distance[edge[1]] > distance[edge[0]] +
given_graph.get_edges_costs[edge]:
                distance[edge[1]] = distance[edge[0]] +
given_graph.get_edges_costs[edge]
                predecessor[edge[1]] = edge[0]
                changed = True
```

```
        if changed is True:
            return None, []

    # if there is no path between source_vertex and end_vertex
    if distance[vertex2] == 99999999:
        return 0, []

    # construct the path by going from the destination to the source
using the predecessor of every vertex
    reversed_path = []
    vertex = vertex2
    reversed_path.append(vertex2)
    while vertex != vertex1:
        reversed_path.append(predecessor[vertex])
        vertex = predecessor[vertex]

    return distance[vertex2], reversed_path[::-1]
```