```
def is_dag_util(graph, vertex, visited, stack):
    visited[vertex] = True
    stack[vertex] = True

    for i in graph.outbound[vertex]:
        if not visited[i]:
            if is_dag_util(graph, i, visited, stack):
                return True
        elif stack[i]:
            return True

    stack[vertex] = False
    return False


def is_dag(graph):
    visited = {vertex: False for vertex in graph.inbound.keys()}
    stack = {vertex: False for vertex in graph.inbound.keys()}

    for i in graph.inbound.keys():
        if not visited[i]:
            if is_dag_util(graph, i, visited, stack):
                return False  #not acyclic

    return True


def topo_sort(graph):
    if is_dag(graph):
        visited = {v: False for v in graph.inbound.keys()}
        stack = []

        for i in graph.inbound.keys():
```

```
            if not visited[i]:
                visit(graph, i, visited, stack)

        start = 0
        end = len(stack) - 1

        while start < end:
            stack[start], stack[end] = stack[end], stack[start]
            start += 1
            end -= 1

        return stack
    else:
        return None


def visit(graph, vertex, visited, stack):
    visited[vertex] = True

    for neighbor in graph.outbound[vertex]:
        if not visited[neighbor]:
            visit(graph, neighbor, visited, stack)

    stack.append(vertex)


def highest_cost_path(graph, start, end):
    if is_dag(graph):
        topological_vertices = topo_sort(graph)

        dist = {v: float('-inf') for v in graph.inbound.keys()}
        prev = {v: None for v in graph.inbound.keys()}
        dist[start] = 0
```

```
        for vertex in topological_vertices:
            if dist[vertex] != float('-inf'):
                for neighbour in graph.iterate_out(vertex):
                    if dist[neighbour] < dist[vertex] + graph.get_cost(vertex,
neighbour):
                        dist[neighbour] = dist[vertex] + graph.get_cost(vertex,
neighbour)
                        prev[neighbour] = vertex

        if dist[end] == float('-inf'):
            return None

        path = [end]
        end2 = end
        while prev[end2] is not None:
            path.append(prev[end2])
            end2 = prev[end2]

        path.reverse()
        return [path, dist[end]]
    else:
        print("The graph is not acyclic, we can not find the highest cost
path!")
```