Jorza Ionut lab 3 Documentation

We had to implement an algorithm that will calculate the minimum cost walk from a starting vertex to a destination vertex.

If there is a negative cycle, we have to print a message and stop the program.

So let's start with the first part of the function, the variables.

```
# Performs the walk that has the minimum cost from starting_vertex to destination_vertex.
# It then returns the path and the total cost of that path.
# If no walk can be done, it returns None as a path and the sum 0.
def perform_min_cost_walk(starting_vertex, destination_vertex):
    queue = []
    dist = {starting_vertex: 0}
    prev = {starting_vertex: None}
    heapq.heappush(queue, (0, starting_vertex))
    inf = False
```

Here we see we have 2 parameters, starting_vertex which represents the vertex that we start from, and destination_vertex which represents the vertex where we want to go.

We have:

- dist dictionary where we will calculate all the costs to a vertex,
- prev dictionary where we put the vertices that helps us go to the destination vertex,
- queue list where we put all the vertices and then after we will take the vertices from this queue list.
- Inf is just a bool variable that turns true if we detect an infinite negative cycle.

```
while len(queue) != 0:
    cost, current_vertex = heapq.heappop(queue)

    if current_vertex == starting_vertex and cost < 0:
        inf = True
        break

    if current_vertex == destination_vertex:
        break

    if cost > dist[current_vertex]:
        continue

    for y in graph[current_vertex]:
        if y[0] not in dist or dist[current_vertex] + y[1] < dist[y[0]]:
            dist[y[0]] = dist[current_vertex] + y[1]
            heapq.heappush(queue, (dist[y[0]], y[0]))
            prev[y[0]] = current_vertex
```

This is the proper algorithm. We go through the queue list until there is nothing else left. We use the heapq.heappop() function so that we can take the minimum cost first (heappop takes the lowest numbers first).

The first ‘if’ checks if the cost is < 0 and we are back to the starting vertex, that means we are in a negative cycle and we have to break the program.

The second ‘if’ checks if we are done (we are on the destination vertex), if that is the case, we break.

The third ‘if’ checks if the cost is bigger that we currently have, if that happens, we skip the vertex.

The ‘for’ goes through all the children vertices of the current vertex (y is the edge from the current vertex to all the other vertices) and we check if the edge going from the current vertex to a vertex is lower than what we currently have. If that is the case, we simply modify the dictionaries (dist and prev).

```
if not inf:
    return extract_path(prev, starting_vertex, destination_vertex)
else:
    return None, 0, True
```

After we are done with the algorithm, we check if there was a negative cycle, if there is none, we call the extract_path function, which is used to extract the minimum cost path from the starting vertex to the ending vertex, having the prev dictionary. If there was a negative cycle, we return None as a path, 0 as a cost and True because we had a negative cycle.

```
# This function is used to compute the path and the total sum from the starting vertex to the destination vertex.
# It is used in the function that performs the minimum cost walk after
# we've completed the parents list for the destination vertex.
def extract_path(parent, starting_vertex, destination_vertex):
    min_sum = 0

    if destination_vertex not in parent:
        return None, 0, False
    path = []
    current_vertex = destination_vertex

    while current_vertex != starting_vertex:
        path.append(current_vertex)
        current_vertex = parent[current_vertex]

    path.append(starting_vertex)
    path.reverse()

    for i in range(0, len(path) - 1):
        for edges in graph[path[i]]:
            if edges[0] == path[i + 1]:
                min_sum += edges[1]

    return path, min_sum, False
```

This is the extract path function, having the prev dictionary, the starting vertex and the destination vertex.

We construct the path list based on the prev dictionary. After we construct the path list, we use the min_sum variable to calculate the cost between all the vertices in the path list.

After all, we return the path list, the sum of the costs and False (no infinite cycle detected).