Practical Work Number 3 Documentation

Group 913 - Geanovu Medeea-Elena

Write a program that, given a graph with costs that has no negative cost cycles and two vertices, finds a lowest cost walk between the given vertices. The program shall use the Floyd-Warshall algorithm.

Function in the DirectedGraph class:

```
def floyd_warshall(self, start, end):

    # infinity will depict the lack of a lowest-cost path between two vertices
    infinity = 9999999
    number_of_vertices = self.count_number_of_vertices()
    # lowest_walk[][] will be the matrix containing the previously passed vertices in the case a
lowest-cost path exists between two vertices
    previous_vertex_matrix = [[-1 for first_vertex in range(number_of_vertices)] for
second_vertex in range(number_of_vertices)]
    # dist_matrix[][] will contain the lowest cost walk between two vertices (one represented by
line index, the other by the column index
    dist_matrix = [[infinity for first_vertex in range(number_of_vertices)] for second_vertex in
range(number_of_vertices)]
    for i in range(number_of_vertices):
        for j in range(number_of_vertices):
            if (i, j) in self._cost:
                dist_matrix[i][j] = self._cost[(i, j)]
                previous_vertex_matrix[i][j] = i
            if i == j:
                dist_matrix[i][j] = 0

print("Following matrices show the shortest distances between every pair of vertices\n")

    print("\nIteration: ", 0)
    for i in range(number_of_vertices):
        for j in range(number_of_vertices):
            if dist_matrix[i][j] == infinity:
                print("INF", end=" ")
             else:
                print(dist_matrix[i][j], end=" ")
        print(" ")


    for k in range(number_of_vertices):
        # each vertex will be seen as an intermediate vertex in the path between any two vertices
(represented by k)
        for i in range(number_of_vertices):
            # each vertex will be considered a destination point for the previously picked vertex k
            for j in range(number_of_vertices):
                # if k is on the path between i and j, and the cost would be lower than the already existing one the value of dist_matrix[i][j] will be updated to the new cost. Checks to avoid negative cost cycles and omits computing paths with any negative values
                 previous = copy.deepcopy(dist_matrix[i][j])
                 if dist_matrix[i][k] + dist_matrix[k][j] >= 0:
                     dist_matrix[i][j] = min(dist_matrix[i][j], dist_matrix[i][k] + dist_matrix[k][j])
                # if the value of the lowest cost walk was changed, then the previous vertex in the path will be changed to k
                if previous != dist_matrix[i][j] and dist_matrix[i][k] + dist_matrix[k][j] >= 0:
                     previous_vertex_matrix[i][j] = k

        # following is an algorithm to print the contents of the matrix is an easy to read way. the lack of a lowest cost path is marked by "INF"

          print("\nIteration: ", k+1)
          for i in range(number_of_vertices):
            for j in range(number_of_vertices):
              if dist_matrix[i][j] == infinity:
                print("INF", end=" ")
              else:
                print(dist_matrix[i][j], end=" ")
            print(" ")

          # algorithm to print the matrix of all previous vertices in the lowest cost walks for all 2 vertices
          print("\nThe previous vertices matrix is:")
          for i in range(number_of_vertices):
            for j in range(number_of_vertices):
                 print(previous_vertex_matrix[i][j], end=" ")
            print(" ")


        # return the distance matrix and previous vertex matrix
        return dist_matrix, previous_vertex_matrix
```

Function in the UI:

```
def floyd_warshall(self):

        # infinity represents that there is no lowest cost walk between certain vertices
        infinity = 9999999
        continue_program = 1
        start_vertex = -1
        ending_vertex = -1
        while not self.__graph.is_vertex(start_vertex):
            start_vertex = int(input("Starting vertex: "))
        while not self.__graph.is_vertex(ending_vertex):
            ending_vertex = int(input("Ending vertex: "))
        dist, lowest_walk = self.__graph.floyd_warshall(start_vertex, ending_vertex)

        while continue_program != 0:

        if dist[start_vertex][ending_vertex] == infinity or dist[start_vertex][ending_vertex] < 0
                print("There is no lowest-cost walk between the vertices", start_vertex, "and",
ending_vertex)
            else:
                # creates the lowest-cost walk between two vertices in the matrix by using the matrix
with previously parsed vertices, skipping the cases where negative costs may exist -> starts from
ending vertex and stops at start vertex. All the needed vertices for the path are found on the line
defined by the start vertex as line index
                print("The lowest-cost walk between the vertices", start_vertex, "and", ending_vertex,
"is:", dist[start_vertex][ending_vertex])
                path = []
                current_vertex = ending_vertex
                path.append(current_vertex)
                while current_vertex != -1 and current_vertex != start_vertex:
                    current_vertex = lowest_walk[start_vertex][current_vertex]
                    path.append(current_vertex)
                # need to reverse the path as it is computed from end to start
                path.reverse()
                print(path)

            continue_program = int(input("Continue? (1 - yes/ 0 - no) >> "))
            if continue_program == 0:
                break
            start_vertex = int(input("Starting vertex: "))
            ending_vertex = int(input("Ending vertex: "))
```