Documentation Practical Work no. 3

Code added

Repository

```
def lowest_cost_walk_floyd_warshall(self, start: int, end: int) -> tuple:
    '''
    This function calculates the lowest cost walk between two vertices in a graph using the Floyd-Warshall algorithm, provided that
    the given graph has no negative cost cycles.
    It verifies the existence of the provided vertices within the graph and returns the starting vertex if both vertices are the same.
    The algorithm initializes a matrix of costs, where the cost of an edge is stored in the corresponding cell.
    It then iterates through the matrix, updating the cost of a path if a shorter path is found.
    The function returns a tuple consisting of a list with the intermediate matrices(as tuples), the path and the cost of the walk.
    '''
    if not ((self.is_vertex(start) and self.is_vertex(end))):
    raise RepoError("\nVertices do not exist in the graph\n")

    no_vertices = self.number_of_vertices()
    cost_matrix = [[float('inf')] * no_vertices for _ in range(no_vertices)] # initialize the matrix of costs
    next_vertex = [[None] * no_vertices for _ in range(no_vertices)] # initialize the matrix for the lowest cost path between two vertices
    intermediate_matrices = []

    for u in self.graph.vertices:
    for v in self.get_outbounds_of_vertex(u):
        cost_matrix[u][v] = self.graph.edges[(u, v)]
      next_vertex[u][v] = v

    copy_matrix = [row.copy() for row in cost_matrix]
    intermediate_matrices.append(cost_matrix.copy())

    for k in self.graph.vertices:
    for i in self.graph.vertices:
        for j in self.graph.vertices:
          if k != i and k != j and i != j:
            if cost_matrix[i][j] > cost_matrix[i][k] + cost_matrix[k][j]:
              cost_matrix[i][j] = cost_matrix[i][k] + cost_matrix[k][j]
              next_vertex[i][j] = next_vertex[i][k]
              copy_matrix = [row.copy() for row in cost_matrix]
              intermediate_matrices.append(copy_matrix)

    if next_vertex[start][end] is None:
      return intermediate_matrices, [], float('inf')
    path = [start]
    cost = cost_matrix[start][end]
    while start != end:
      start = next_vertex[start][end]
      path.append(start)

    return intermediate_matrices, path, cost
```

Service

```
def lowest_cost_walk_floyd_warshall(self, start: int, end: int) -> tuple:
    """
    Returns the lowest cost walk between two vertices using Floyd-Warshall algorithm
    Args:
        start : the starting vertex
        end : the ending vertex
    Preconditions:
        Both vertices must exist in the graph
    """
    return self.repo.lowest_cost_walk_floyd_warshall(start, end)
```

UI

```
if command == "1":
    start = int(input("Enter start vertex: "))
    end = int(input("Enter end vertex: "))
    intermediate_matrices, path, cost = self.service.lowest_cost_walk_floyd_warshall(start, end)
    if len(path) == 0:
        print("\nThere is no path between the two vertices\n")
    else:
        print("\nLength of the path: ", len(path) - 1)
        print("Cost of the path: ", cost)
        print("Path: ", end="")
        for vertex in path[:-1]:
            print(vertex, end=" -> ")
        print(path[-1])

    i = 1
    for matrix in intermediate_matrices:
        row_headers = [f'Vertex {i}' for i in range(len(matrix))]
        col_headers = [f'Vertex {j}' for j in range(len(matrix[0]))]
        print(f"\nIntermediate matrix {i}:")
        i += 1
        print(tabulate(matrix, headers=col_headers, showindex=row_headers, tablefmt="fancy_grid"))
```

Bonus 1

Repository

```
def find_min_cost_paths(self, source: int, target: int) -> int:
  '''
  This function calculates the number of minimum cost paths between two vertices in a graph.
  It verifies the existence of the provided vertices within the graph and returns the starting vertex if both vertices are the same.
  The algorithm initializes a matrix of costs, where the cost of an edge is stored in the corresponding cell.
  It then iterates through the matrix, updating the cost of a path if a shorter path is found.
  The function returns a tuple consisting of the number of minimum cost paths and a list with the paths.
  '''
  n = self.number_of_vertices()
  cost_matrix = [[float('inf')] * n for _ in range(n)]
  num_paths = [[0] * n for _ in range(n)]

  for u in range(n):
      for v in range(n):
      if self.is_edge(u, v):
          cost_matrix[u][v] = self.graph.edges[(u, v)]
          num_paths[u][v] = 1

  for k in range(n):
    for i in range(n):
      for j in range(n):
          if cost_matrix[i][k] + cost_matrix[k][j] < cost_matrix[i][j]:
            cost_matrix[i][j] = cost_matrix[i][k] + cost_matrix[k][j]
            num_paths[i][j] = num_paths[i][k] * num_paths[k][j]
          elif cost_matrix[i][k] + cost_matrix[k][j] == cost_matrix[i][j]:
            num_paths[i][j] += num_paths[i][k] * num_paths[k][j]

  return num_paths[source][target]
```

Service

```
def find_min_cost_paths(self, source: int, target: int) -> int:
  """
  Returns the number of minimum cost paths between two vertices
  Args:
      source : the starting vertex
      target : the ending vertex
  Preconditions:
      Both vertices must exist in the graph
  """
  return self.repo.find_min_cost_paths(source, target)
```

UI

```
elif command == "2":
  start = int(input("Enter start vertex: "))
  end = int(input("Enter end vertex: "))
  number_of_paths = self.service.find_min_cost_paths(start, end);
  print(f"\nNumber of minimum cost paths between {start} and {end}: {number_of_paths}")
```

Bonus 2

Repository

```
 def find_all_possible_paths(self, source: int, target: int) -> int:
    '''
    This function calculates the number of possible paths between two vertices in a graph.
    It verifies the existence of the provided vertices within the graph and returns the starting vertex if both vertices are the same.
    The algorithm initializes a matrix of costs, where the cost of an edge is stored in the corresponding cell.
    It then iterates through the matrix, updating the cost of a path if a shorter path is found.
    The function returns the number of possible paths between the two vertices.
    '''
    n = self.number_of_vertices()
    num_paths = [[0] * n for _ in range(n)]

    for u in range(n):
      for v in range(n):
      if self.is_edge(u, v):
        num_paths[u][v] = 1

    for k in range(n):
    for i in range(n):
      for j in range(n):
        num_paths[i][j] += num_paths[i][k] * num_paths[k][j]

    return num_paths[source][target]
```

Service

```
def find_all_possible_paths(self, start: int, target: int) -> int:
  """
  Returns the number of all possible paths between two vertices
  Args:
      start : the starting vertex
      target : the ending vertex
  Preconditions:
      Both vertices must exist in the graph
```

```
    """
    return self.repo.find_all_possible_paths(start, target)
```

UI

```
elif command == "3":
    start = int(input("Enter start vertex: "))
    end = int(input("Enter end vertex: "))
    number_of_paths = self.service.find_all_possible_paths(start, end);
    print(f"\nNumber of possible paths between {start} and {end}: {number_of_paths}")
```