Documentation Practical Work no. 4

Code added

Repository

```
    def TopoSortDFS(self, vertex: int, sorted: list, fullyProcessed: set, inProcess: set) -> bool:
        '''
        This function performs a depth-first search on the graph, topologically sorting the vertices.
        It verifies the existence of the provided vertex within the graph.
        The function returns True if the graph is acyclic, False otherwise.
        '''
        inProcess.add(vertex)
        for neighbor in self.get_inbounds_of_vertex(vertex):
            if neighbor in inProcess:
                return False
            if neighbor not in fullyProcessed:
                if not self.TopoSortDFS(neighbor, sorted, fullyProcessed, inProcess):
                    return False
        inProcess.remove(vertex)
        sorted.append(vertex)
        fullyProcessed.add(vertex)
        return True

    def topological_sort(self) -> list:
        '''
        This function performs a topological sort on the graph.
        It verifies the existence of the provided vertices within the graph.
        The function returns a list with the topologically sorted vertices.
        '''
        sorted = []
        fullyProcessed = set()
        inProcess = set()
        for vertex in self.get_vertices():
            if vertex not in fullyProcessed:
                if not self.TopoSortDFS(vertex, sorted, fullyProcessed, inProcess):
                    return None
        return sorted
```

Service

```
    def topological_sort(self) -> list:
        """
        Returns a topological sort of the graph
        """
        return self.repo.topological_sort()
```

UI

```
if command == "1":
        topological_order = self.service.topological_sort()
        if topological_order is None:
            print("\nGraph is not a DAG\n")
        else:
            print("\nTopological order: ", end="")
            for vertex in topological_order:
                print(vertex, end=" | ")
            print()
```

Bonus 2

Repository

```
    def count_paths(self, start: int, end: int) -> int:
        '''
        This function calculates the number of 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 function returns the number of paths between the two vertices.
        '''
        if not ((self.is_vertex(start) and self.is_vertex(end))):
            raise RepoError("\nVertices do not exist in the graph\n")

        if start == end:
            return 1

        sorted_vertices = self.topological_sort()
        if sorted_vertices is None:
            raise RepoError("\nGraph contains a cycle\n")

        num_paths = [0] * self.number_of_vertices()
        num_paths[start] = 1

        for vertex in sorted_vertices:
            for neighbor in self.get_outbounds_of_vertex(vertex):
                num_paths[neighbor] += num_paths[vertex]

        return num_paths[end]
```

Service

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

UI

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

Bonus 3

Domain

```
    def get_cost(self, i: int, j: int) -> int:
        '''
        Returns the cost of the edge (i, j)
        '''
        return self.edges[(i, j)]
```

Repository

```
    def count_lowest_cost_paths(self, start: int, end: int) -> tuple:
        '''
        This function calculates the number of lowest 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 function returns a tuple consisting of the number of lowest cost paths and the lowest cost.
        '''
        if not ((self.is_vertex(start) and self.is_vertex(end))):
            raise RepoError("\nVertices do not exist in the graph\n")

        if start == end:
            return 1, 0

        sorted_vertices = self.topological_sort()
        if sorted_vertices is None:
            raise RepoError("\nGraph contains a cycle\n")

        num_paths = [0] * self.number_of_vertices()
        lowest_cost = [float('inf')] * self.number_of_vertices()
        num_paths[start] = 1
        lowest_cost[start] = 0

        for vertex in sorted_vertices:
            for neighbor in self.get_outbounds_of_vertex(vertex):
                cost = self.graph.get_cost(vertex, neighbor)
                if lowest_cost[vertex] + cost < lowest_cost[neighbor]:
                    lowest_cost[neighbor] = lowest_cost[vertex] + cost
                    num_paths[neighbor] = num_paths[vertex]
                elif lowest_cost[vertex] + cost == lowest_cost[neighbor]:
                    num_paths[neighbor] += num_paths[vertex]

        return num_paths[end], lowest_cost[end]
```

Service

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

UI

```
elif command == "3":
    start = int(input("Enter start vertex: "))
    end = int(input("Enter end vertex: "))
    number_of_paths, cost = self.service.count_lowest_cost_paths(start, end)
    print(f"\nNumber of lowest cost paths between {start} and {end} is: {number_of_paths}\nAnd the cost of the path is: {cost}\n")
```