Documentation – Practical Work 4

UndirectedGraph:

- **Properties:**
  - **get_edges_costs**: Returns a dictionary containing the costs of all edges in the graph.
  - **get_number_of_vertices**: Returns the number of vertices in the graph.
  - **get_number_of_edges**: Returns the number of edges in the graph.
- **Vertex operations:**
  - **add_vertex(self, vertex_to_add)**: Adds a vertex to the graph. Returns **True** if the vertex was added successfully, or **False** if it already exists.
  - **remove_vertex(self, vertex_to_remove)**: Removes a vertex from the graph. Returns **True** if the vertex was removed successfully, or **False** if it does not exist.
  - **parse_vertices(self)**: Returns an iterator that yields all vertices in the graph.
  - **get_vertex_degree(self, vertex)**: Returns the degree of a vertex (the number of edges incident to it).
- **Edge operations:**
  - **add_edge(self, vertex1, vertex2, cost)**: Adds an edge between two vertices with the specified cost. Returns **True** if the edge was added successfully, or **False** if it already exists.
  - **remove_edge(self, vertex1, vertex2)**: Removes an edge between two vertices. Returns **True** if the edge was removed successfully, or **False** if it does not exist.
  - **get_edge_cost(self, vertex1, vertex2)**: Returns the cost of the edge between the two vertices.
  - **update_edge_cost(self, vertex1, vertex2, cost)**: Updates the cost of the edge between the two vertices. Returns **True** if the update was successful, or **False** if the edge does not exist.
  - **parse_edges(self)**: Returns an iterator that yields all edges in the graph.
- **Graph properties:**
  - **parse_neighbours_for_vertex(self, x)**: Returns an iterator that yields all neighbors of a given vertex.
  - **find_if_edge_exists(self, vertex1, vertex2)**: Checks if an edge exists between two vertices. Returns **True** if the edge exists, or **False** otherwise.

- **find_if_vertex_exists(self, vertex)**: Checks if a vertex exists in the graph. Returns **True** if the vertex exists, or **False** otherwise.

- **Graph copying:**
  - **make_copy_of_current_graph(self)**: Creates and returns a deep copy of the current graph.

Minimum cost tree (Kruskal's algorithm):

- **minimum_cost_tree_Kruskal(self, minimum_tree)**: Calculates the minimum cost spanning tree of the graph using Kruskal's algorithm. Returns the minimum cost and modifies the **minimum_tree** list to store the edges in the minimum spanning tree.

Source code:

```
def minimum_cost_tree_Kruskal(self,minimum_tree):
    minimum_cost = 0
    count = 0
    # the array of trees to which each vertex belongs to
    tree = [0] * self._nr_vertices
    for x in self.vertices:
        tree[x] = x
    # sort edges by cost
    ordered_edges = sorted(self.edges_costs.items(),
key=lambda x: x[1])
    x = 0
    while x < self._nr_edges and count < self._nr_vertices:
        # edge is the current edge in the array of ordered
edges
        edge = ordered_edges[x]
        # v and w represent the trees to which each vertex of
the current edge belongs to
        v = tree[edge[0][0]]
        w = tree[edge[0][1]]
        # if adding the current edge doesn't form a cycle
        if v != w and (edge[0][1], edge[0][0]) not in
minimum_tree:
            minimum_cost += edge[1]
            count += 1
            minimum_tree.append(edge[0])
            # now they are part of the same tree
            for y in self.vertices:
                if tree[y] == w:
                    tree[y] = v
        x += 1
    return minimum_cost
```

Explanation:

The method modifies the minimum_tree list passed as a parameter by adding the edges of the minimum spanning tree to it.

1. Initialize minimum_cost and count variables to 0. These will be used to keep track of the total cost of the minimum spanning tree and the number of edges added to the tree so far.
2. Create an array tree of size self._nr_vertices to represent the trees to which each vertex belongs. Initially, each vertex is its own tree. This array will be used to check if adding an edge creates a cycle.
3. Iterate over each vertex in self.vertices and set its corresponding entry in the tree array to the vertex itself. This ensures that each vertex belongs to its own tree initially.
4. Sort the edges of the graph by their costs in ascending order using the sorted function and a lambda function as the key for sorting. The edges_costs.items() method returns a list of key-value pairs representing the edges and their costs.
5. Initialize a variable x to 0. This will be used as an index to traverse the sorted edges.
6. Enter a while loop that continues as long as x is less than the number of edges (self._nr_edges) and the number of edges added to the minimum spanning tree (count) is less than the number of vertices (self._nr_vertices).
7. Get the current edge from the sorted edges using the index x. The edge is represented as a key-value pair, where the key is a tuple (vertex1, vertex2) and the value is the cost of the edge.
8. Retrieve the trees to which the vertices of the current edge belong (v and w) from the tree array.
9. Check if adding the current edge to the minimum spanning tree creates a cycle. This is done by comparing the trees to which the vertices belong. If the trees are different and the reverse edge (vertex2, vertex1) is not already in the minimum_tree list, then the edge can be added.
10. If the condition is satisfied, update the minimum_cost by adding the cost of the current edge, increment the count by 1, and append the current edge to the minimum_tree list.
11. Update the tree array to merge the trees of vertices w into the tree of vertex v. This ensures that the vertices of the current edge are now part of the same tree.
12. Increment the index x to move to the next edge in the sorted edges.
13. Once the while loop ends, return the minimum_cost, which represents the total cost of the minimum spanning tree.