

Objective: **Implementation and analysis of Kruskal's Algorithm**

Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

Kruskal's algorithm

The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Code:

```

parent = dict()
rank = dict()
def make_set(vertex):
    parent[vertex] = vertex
    rank[vertex] = 0
def find(vertex):
    if parent[vertex] != vertex:
        parent[vertex] = find(parent[vertex])
    return parent[vertex]
def union(vertex1, vertex2):
    root1 = find(vertex1)
    root2 = find(vertex2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
        if rank[root1] == rank[root2]: rank[root2] +=
1
def kruskal(graph):
    for vertex in graph['vertices']:
        make_set(vertex)
    minimum_spanning_tree = set()
    edges = list(graph['edges'])
    edges.sort()
    # print edges
    for edge in edges:
        weight, vertex1, vertex2 = edge
        if find(vertex1) != find(vertex2):
            union(vertex1, vertex2)
            minimum_spanning_tree.add(edge)
    return sorted(minimum_spanning_tree)

graph = {
    'vertices': ['A', 'B', 'C', 'D', 'E', 'F', 'G'],
    'edges': set([
        (7, 'A', 'B'),
        (5, 'A', 'D'),
        (7, 'B', 'A'),
        (8, 'B', 'C'),
        (9, 'B', 'D'),
        (7, 'B', 'E'),
        (8, 'C', 'B'),
        (5, 'C', 'E'),
        (5, 'D', 'A'),
        (9, 'D', 'B'),
        (7, 'D', 'E'),
        (6, 'D', 'F'),
        (7, 'E', 'B'),

```

```

        (5, 'E', 'C'),
        (15, 'E', 'D'),
        (8, 'E', 'F'),
        (9, 'E', 'G'),
        (6, 'F', 'D'),
        (8, 'F', 'E'),
        (11, 'F', 'G'),
        (9, 'G', 'E'),
        (11, 'G', 'F'),
    ])

}

print(kruskal(graph))

```

The screenshot shows a code editor with the following Python code for Kruskal's Algorithm:

```

def kruskal(graph):
    for vertex in graph['vertices']:
        make_set(vertex)
    minimum_spanning_tree = set()
    edges = list(graph['edges'])
    edges.sort()
    # print edges
    for edge in edges:
        weight, vertex1, vertex2 = edge
        if find(vertex1) != find(vertex2):
            union(vertex1, vertex2)
            minimum_spanning_tree.add(edge)
    return sorted(minimum_spanning_tree)

graph = {
    'vertices': ['A', 'B', 'C', 'D', 'E', 'F', 'G'],
    'edges': set([
        (7, 'A', 'B'),
        (9, 'A', 'D'),
        (7, 'B', 'A'),
        (8, 'B', 'C'),

```

The IDE interface includes a sidebar with a project tree, a main editor window, and a bottom status bar showing the current file, line numbers, and various settings like encoding and indentation.

This screenshot shows the implementation of the `find` function in a Python file named `13_Kruskal's Algorithm.py`. The function is designed to find the root of a given vertex in a disjoint-set structure. It uses a dictionary `parent` to store the parent of each vertex. The function recursively finds the parent until it reaches a vertex that is its own parent, which is the root. The `rank` dictionary is also used to store the rank of each vertex, which is used in the `union` function to ensure the tree remains balanced.

```
parent = dict()
rank = dict()

def make_set(vertex):
    parent[vertex] = vertex
    rank[vertex] = 0

def find(vertex):
    if parent[vertex] != vertex:
        parent[vertex] = find(parent[vertex])
    return parent[vertex]

def union(vertex1, vertex2):
    root1 = find(vertex1)
    root2 = find(vertex2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
            if rank[root1] == rank[root2]:
                rank[root1] += 1
    return root1
```

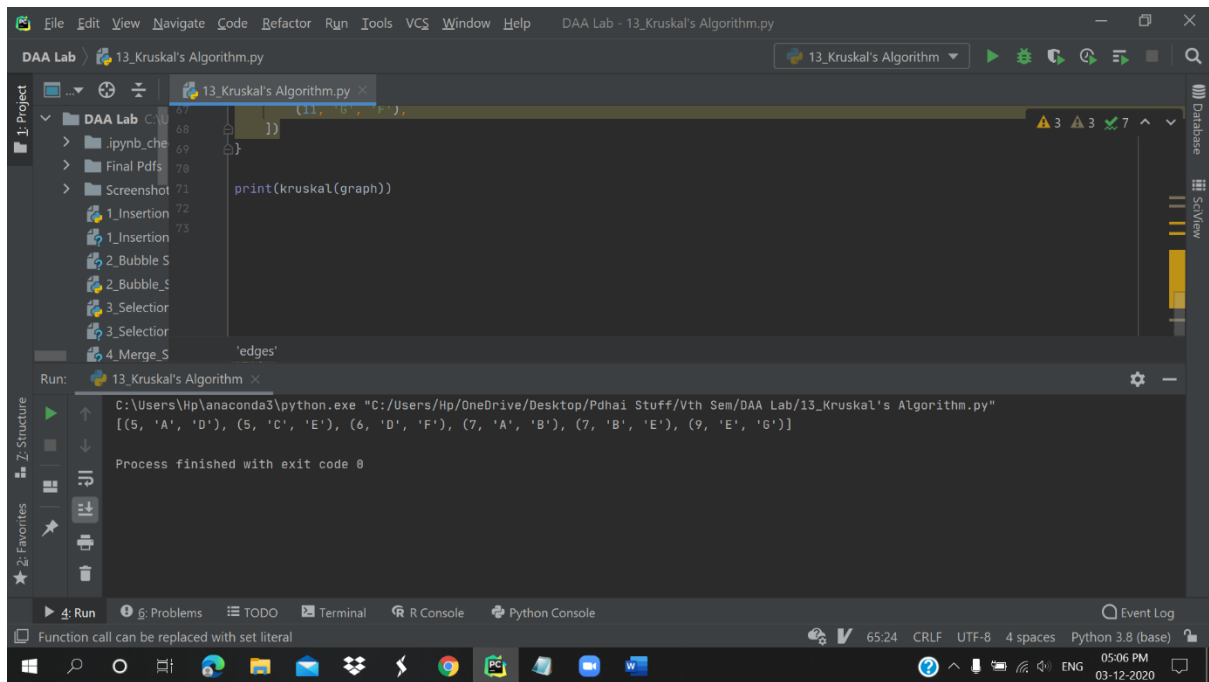
This screenshot shows the implementation of the `union` function and the final output of the Kruskal's Algorithm. The `union` function is used to merge two vertices into a single set. The final output of the algorithm is a list of edges that form a minimum spanning tree. The edges are printed as a list of tuples, where each tuple contains the weight of the edge and the two vertices it connects.

```
(8, 'B', 'C'),
(9, 'B', 'D'),
(7, 'B', 'E'),
(8, 'C', 'E'),
(5, 'C', 'E'),
(5, 'D', 'A'),
(9, 'D', 'B'),
(7, 'D', 'E'),
(6, 'D', 'F'),
(7, 'E', 'B'),
(5, 'E', 'C'),
(15, 'E', 'D'),
(8, 'E', 'F'),
(9, 'E', 'G'),
(6, 'F', 'D'),
(8, 'F', 'E'),
(11, 'F', 'G'),
(9, 'G', 'E'),
(11, 'G', 'F'),
])

print(kruskal(graph))
```

Output:

```
[ (5, 'A', 'D'), (5, 'C', 'E'), (6, 'D', 'F'), (7, 'A', 'B'),
  (7, 'B', 'E'), (9, 'E', 'G') ]
```



Time Complexities:

The time complexity of Kruskal's algorithm is $O(E \log E)$.

Kruskal's Algorithm Applications:

- In order to layout electrical wiring
- In computer network (LAN connection)