

## 1. Dijkstra's algorithm

```
dijkstra's.cpp
1  #include <iostream>
2  using namespace std;
3  #include <limits.h>
4  #define V 9
5  int minDistance(int dist[], bool sptSet[])
6  {
7      int min = INT_MAX, min_index;
8
9      for (int v = 0; v < V; v++)
10         if (sptSet[v] == false && dist[v] <= min)
11             min = dist[v], min_index = v;
12
13     return min_index;
14 }
15 void printSolution(int dist[])
16 {
17     cout << "Vertex \t Distance from Source" << endl;
18     for (int i = 0; i < V; i++)
19         cout << i << " \t\t\t\t" << dist[i] << endl;
20 }
21 void dijkstra(int graph[V][V], int src)
22 {
23     int dist[V];
24
25     bool sptSet[V];
26     for (int i = 0; i < V; i++)
27         dist[i] = INT_MAX, sptSet[i] = false;
```

```
C:\Users\sirra\OneDrive\Belgr... X + -
Vertex    Distance from Source
0          0
1          4
2         12
3         19
4         21
5         11
6          9
7          8
8         14

-----
Process exited after 0.1359 seconds with return value 0
Press any key to continue . . . |
```

## 2. Huffman codes

```
Project
  closet pair C:\Users\sirra\PycharmProjects\closet pair
    .venv library root
    main.py
  External Libraries
  Scratches and Consoles

main.py
1  import heapq
2
3
4  2 usages
5  class node:
6      def __init__(self, freq, symbol, left=None, right=None):
7          self.freq = freq
8          self.symbol = symbol
9          self.left = left
10         self.right = right
11         self.huff = ''
12
13     def __lt__(self, nxt):
14         return self.freq < nxt.freq
15
16  3 usages
17  def printNodes(node, val=''):
18      newVal = val + str(node.huff)
19      if (node.left):
20          printNodes(node.left, newVal)
21      if (node.right):
22          printNodes(node.right, newVal)
23
24  Run main
25
26  "C:\Users\sirra\PycharmProjects\closet pair\.venv\Scripts\python.exe" "C:\Users\sirra\PycharmProjects\closet pair\main.py"
27
28  f -> 0
29  c -> 100
30  d -> 101
31  a -> 1100
32  b -> 1101
33  e -> 111
```

### 3.Container loading

```
jikstra's.cpp
1 #include <bits/stdc++.h>
2 using namespace std;
3 double cont[1000][1000];
4 void num_of_containers(int n,
5                        double x)
6 {
7     int count = 0;
8     cont[1][1] = x;
9     for (int i = 1; i <= n; i++) {
10         for (int j = 1; j <= i; j++) {
11             if (cont[i][j] >= (double)1) {
12                 count++;
13                 cont[i + 1][j]
14                     += (cont[i][j]
15                        - (double)1)
16                       / (double)2;
17
18                 cont[i + 1][j + 1]
19                     += (cont[i][j]
20                        - (double)1)
21                       / (double)2;
22             }
23         }
24     }
25     cout << count;
26 }
27 int main()
```

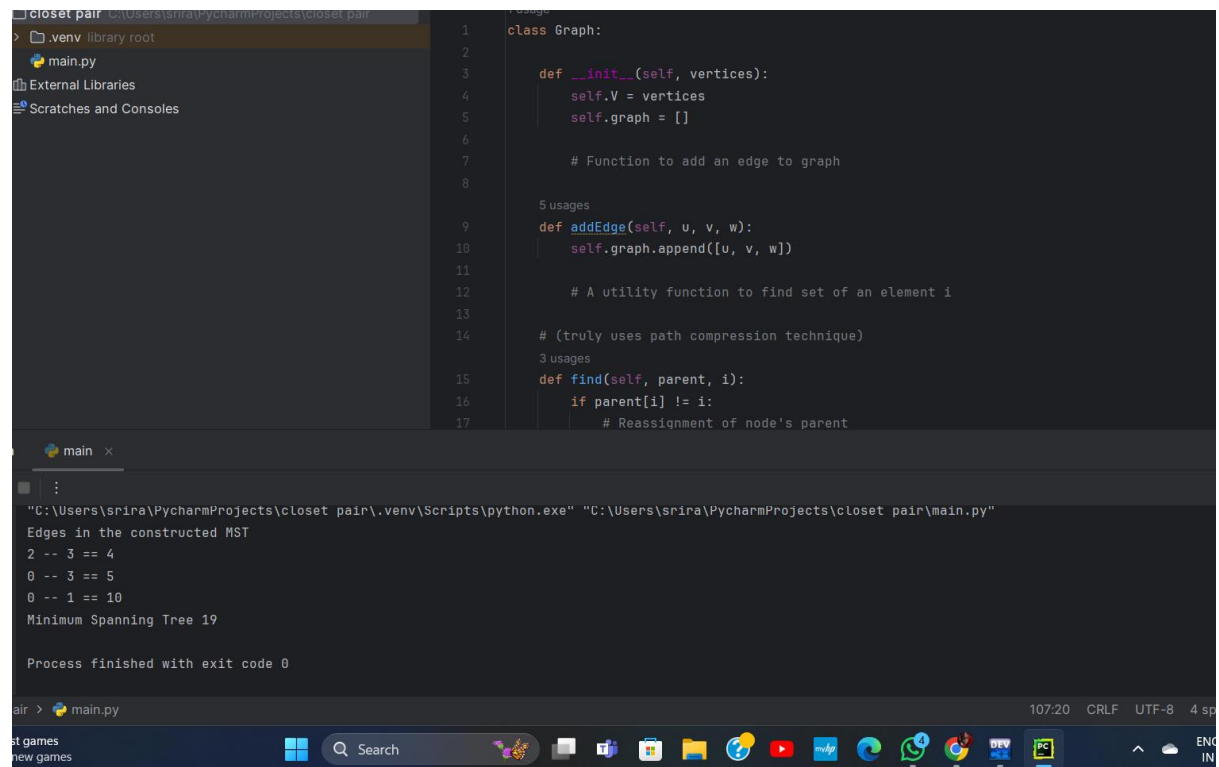
```
C:\Users\sriya\OneDrive\Belgr
4
-----
Process exited after 0.1547 seconds with return
Press any key to continue . . .
```

### 4.Minimum spanning tree

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAX 30
5
6 typedef struct edge {
7     int u, v, w;
8 } Edge;
9
10 typedef struct graph {
11     int V, E;
12     Edge edges[MAX];
13 } Graph;
14
15 int parent[MAX];
16
17 int find(int i) {
18     while (parent[i])
19         i = parent[i];
20     return i;
21 }
22
23 void union_ij(int i, int j) {
24     parent[i] = j;
25 }
26
27 void kruskalMST(Graph graph) {
28     int minCost = 0;
29     for (int i = 0; i < MAX; i++)
30         parent[i] = 0;
31
32     int edgeCount = 0;
33     for (int i = 0; edgeCount < graph.V - 1; i++) {
34         int x = find(graph.edges[i].u);
35         int y = find(graph.edges[i].v);
```

```
Select E:\DAA\LAB\LAB PROGRAMS\Untitled1545.exe
0 ~ 1
0 ~ 2
0 ~ 3
Minimum Cost: 21
-----
Process exited after 0.1878 seconds with return value 0
Press any key to continue . . .
```

## 5. Kruskal's algorithm



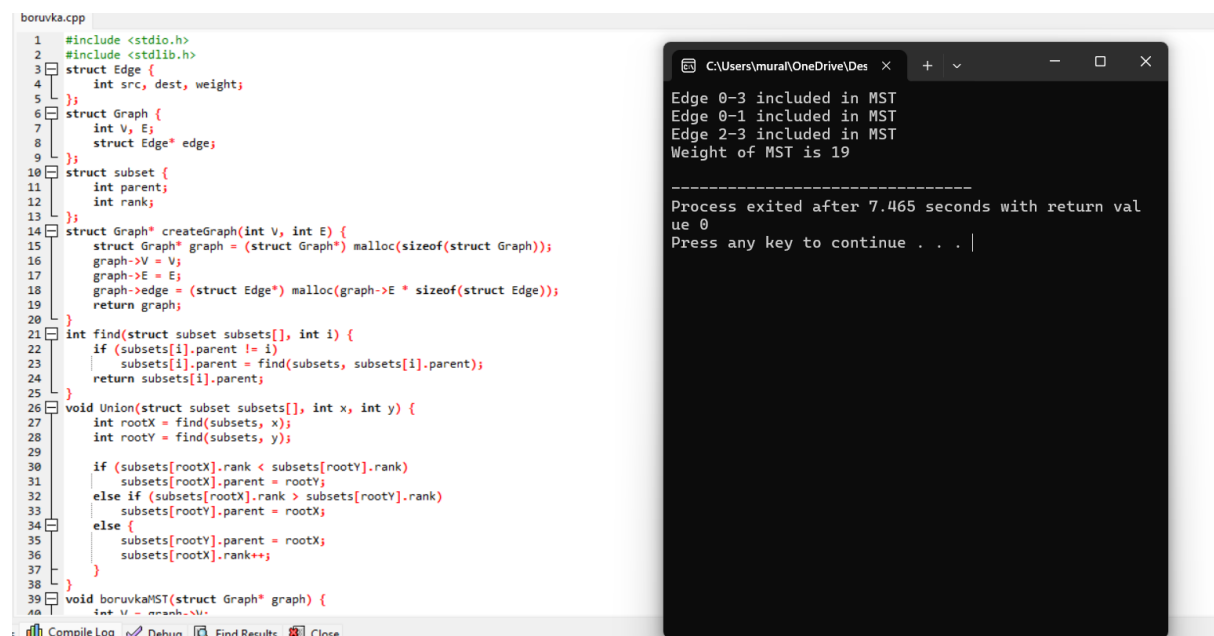
The screenshot shows the PyCharm IDE with a Python file named `main.py`. The code implements Kruskal's algorithm for finding a Minimum Spanning Tree (MST) in a graph. The `Graph` class has methods `__init__`, `addEdge`, and `find`. The `main` function reads the number of vertices and edges, adds the edges to the graph, and then prints the edges in the constructed MST and the total weight of the MST.

```
1 class Graph:
2
3     def __init__(self, vertices):
4         self.V = vertices
5         self.graph = []
6
7         # Function to add an edge to graph
8
9     5 usages
10    def addEdge(self, u, v, w):
11        self.graph.append([u, v, w])
12
13        # A utility function to find set of an element i
14
15    # (truly uses path compression technique)
16    3 usages
17    def find(self, parent, i):
18        if parent[i] != i:
19            # Reassignment of node's parent
```

The output in the console shows the edges in the constructed MST and the total weight of the MST.

```
Edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Spanning Tree 19
Process finished with exit code 0
```

## 6. Boruvka's algorithm



The screenshot shows the Visual Studio Code IDE with a C++ file named `boruvka.cpp`. The code implements Boruvka's algorithm for finding a Minimum Spanning Tree (MST) in a graph. The `Graph` struct has fields `V`, `E`, and `edge`. The `createGraph` function creates a graph structure. The `find` function finds the root of a subset. The `Union` function unions two subsets. The `boruvkaMST` function finds the MST using Boruvka's algorithm.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct Edge {
4     int src, dest, weight;
5 };
6 struct Graph {
7     int V, E;
8     struct Edge* edge;
9 };
10 struct subset {
11     int parent;
12     int rank;
13 };
14 struct Graph* createGraph(int V, int E) {
15     struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
16     graph->V = V;
17     graph->E = E;
18     graph->edge = (struct Edge*) malloc(graph->E * sizeof(struct Edge));
19     return graph;
20 }
21 int find(struct subset subsets[], int i) {
22     if (subsets[i].parent != i)
23         subsets[i].parent = find(subsets, subsets[i].parent);
24     return subsets[i].parent;
25 }
26 void Union(struct subset subsets[], int x, int y) {
27     int rootX = find(subsets, x);
28     int rootY = find(subsets, y);
29
30     if (subsets[rootX].rank < subsets[rootY].rank)
31         subsets[rootX].parent = rootY;
32     else if (subsets[rootX].rank > subsets[rootY].rank)
33         subsets[rootY].parent = rootX;
34     else {
35         subsets[rootY].parent = rootX;
36         subsets[rootX].rank++;
37     }
38 }
39 void boruvkaMST(struct Graph* graph) {
40     int V = graph->V;
41     int E = graph->E;
42     struct subset subsets[V];
43     for (int i = 0; i < V; i++)
44         subsets[i].parent = i;
45     for (int i = 0; i < E; i++)
46         subsets[graph->edge[i].src].parent = graph->edge[i].dest;
47     for (int i = 0; i < E; i++)
48         Union(subsets, graph->edge[i].src, graph->edge[i].dest);
49     int MSTWeight = 0;
50     for (int i = 0; i < E; i++)
51         if (find(subsets, graph->edge[i].src) != find(subsets, graph->edge[i].dest))
52             MSTWeight += graph->edge[i].weight;
53     printf("Weight of MST is %d", MSTWeight);
54 }
```

The output in the console shows the edges included in the MST and the total weight of the MST.

```
Edge 0-3 included in MST
Edge 0-1 included in MST
Edge 2-3 included in MST
Weight of MST is 19

-----
Process exited after 7.465 seconds with return value 0
Press any key to continue . . .
```