1. Dijikstrs algorithm

function Dijkstra(Graph, source):

dist[] := array of distances initialized to infinity

dist[source] := 0

sptSet[] := array of booleans initialized to false

for each vertex v in Graph:

u := vertex in dist[] with minimum distance value not yet in sptSet[]

sptSet[u] := true

for each neighbor v of u:

if v is not in sptSet[]:

if dist[u] + weight(u, v) < dist[v]:

dist[v] := dist[u] + weight(u, v)

return dist[]

1. Huffman codes

Algorithm Huffman (c)

{

n= |c|

Q = c

for i<-1 to n-1

do

{

temp <- get node ()

left (temp] Get\_min (Q) right [temp] Get Min (Q)

a = left [templ b = right [temp]

F [temp]<- f[a] + [b]

insert (Q, temp)

}

return Get\_min (0)

}

1. Container loading

def num\_of\_containers(n, x):

count = 0

# Container on top level

cont[1][1] = x

for i in range(1, n + 1):

for j in range(1, i + 1):

# If container gets filled

if (cont[i][j] >= 1):

count += 1

# Dividing the liquid

# equally in two halves

cont[i + 1][j] += (cont[i][j] - 1) / 2

cont[i + 1][j + 1] += (cont[i][j] - 1) / 2

print(count)

# Driver code

n = 3

x = 5

num\_of\_containers(n, x)

# This code is contributed by yatinagg

1. Minimum spanning tree

from collections import defaultdict

def minimum\_spanning\_tree(graph):

def dfs(node):

visited.add(node)

for neighbor, weight in graph[node]:

if neighbor not in visited:

edges.append((node, neighbor, weight))

dfs(neighbor)

visited = set()

edges = []

start\_node = next(iter(graph))

dfs(start\_node)

return edges

# Example Usage

graph = {

'A': {('B', 2), ('C', 3)},

'B': {('A', 2), ('C', 1), ('D', 1)},

'C': {('A', 3), ('B', 1), ('D', 2)},

'D': {('B', 1), ('C', 2)}

}

minimum\_spanning\_tree(graph)

1. Kruskals algorithm

# Kruskal's algorithm in Python

class Graph:

def \_init\_(self, vertices):

self.V = vertices

self.graph = []

def add\_edge(self, u, v, w):

self.graph.append([u, v, w])

# Search function

def find(self, parent, i):

if parent[i] == i:

return i

return self.find(parent, parent[i])

def apply\_union(self, parent, rank, x, y):

xroot = self.find(parent, x)

yroot = self.find(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

# Applying Kruskal algorithm

def kruskal\_algo(self):

result = []

i, e = 0, 0

self.graph = sorted(self.graph, key=lambda item: item[2])

parent = []

rank = []

for node in range(self.V):

parent.append(node)

rank.append(0)

while e < self.V - 1:

u, v, w = self.graph[i]

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

if x != y:

e = e + 1

result.append([u, v, w])

self.apply\_union(parent, rank, x, y)

for u, v, weight in result:

print("%d - %d: %d" % (u, v, weight))

g = Graph(6)

g.add\_edge(0, 1, 4)

g.add\_edge(0, 2, 4)

g.add\_edge(1, 2, 2)

g.add\_edge(1, 0, 4)

g.add\_edge(2, 0, 4)

g.add\_edge(2, 1, 2)

g.add\_edge(2, 3, 3)

g.add\_edge(2, 5, 2)

g.add\_edge(2, 4, 4)

g.add\_edge(3, 2, 3)

g.add\_edge(3, 4, 3)

g.add\_edge(4, 2, 4)

g.add\_edge(4, 3, 3)

g.add\_edge(5, 2, 2)

g.add\_edge(5, 4, 3)

g.kruskal\_algo()

6. Boruvkas algorithm

def add\_edge(self, src, dest, weight):

self.graph.append([src, dest, weight])

def boruvka\_mst(self):

parent = {}

cheapest = {}

num\_trees = self.V

result = []

def find(node):

if parent[node] == node:

return node

return find(parent[node])

def union(node1, node2):

root1 = find(node1)

root2 = find(node2)

if root1 != root2:

result.append([node1, node2])

num\_trees -= 1

if cheapest[root1] is None or cheapest[root1][2] > cheapest[root2][2]:

cheapest[root1] = cheapest[root2]

parent[root2] = root1

while num\_trees > 1:

for i in range(self.V):

cheapest[i] = None

parent[i] = i

for i in range(len(self.graph)):

src, dest, weight = self.graph[i]

root\_src = find(src)

root\_dest = find(dest)

if root\_src != root\_dest:

if cheapest[root\_src] is None or cheapest[root\_src][2] > weight:

cheapest[root\_src] = [src, dest, weight]

if cheapest[root\_dest] is None or cheapest[root\_dest][2] > weight:

cheapest[root\_dest] = [src, dest, weight]

for node1, node2, weight in result:

union(node1, node2)

return result