Lab 8

1. Coin Change Problem - All combinations

def a(c, s):

result = []

def backtrack(t, start, path):

if t == 0:

result.append(path)

return

for i in range(start, len(s)):

if t >= s[i]:

backtrack(t - s[i], i, path + [s[i]])

backtrack(c, 0, [])

return result

c = 6

s = [1, 3,4]

print("All combinations of coins:", a(c, s))

Output:

2. Knapsack Problem

def b(v, w, m):

i = [(v[i] / w[i], v[i], w[i]) for i in range(len(v))]

i.sort(reverse=True)

k = 0

for p, v, w in i:

if m - w >= 0:

m -= w

k += v

else:

k += p \* m

break

return k

v = [60, 100, 120]

w = [10, 20, 30]

m = 50

print("Maximum value:", b(v, w, m))

Output:

3. Job Sequencing with Deadlines

def c(j, d):

n = len(j)

j.sort(key=lambda x: x[1], reverse=True)

t = [False] \* n

p = ['-1'] \* n

for k in range(n):

for l in range(min(n, j[k][2]) - 1, -1, -1):

if not t[l]:

t[l] = True

p[l] = j[k][0]

break

return [x for x in p if x != '-1']

j = [('a', 100, 2), ('b', 19, 1), ('c', 27, 2), ('d', 25, 1), ('e', 15, 3)]

print("Job sequence:", c(j, len(j)))

Output:

4. Single Source Shortest Paths: Dijkstra's Algorithm

import heapq

def d(g, s):

q, seen, d = [(0, s)], set(), {}

while q:

(cost, v) = heapq.heappop(q)

if v not in seen:

seen.add(v)

d[v] = cost

for next, w in g[v]:

if next not in seen:

heapq.heappush(q, (cost + w, next))

return d

g = {

'a': [('b', 1), ('c', 4)],

'b': [('a', 1), ('c', 2), ('d', 5)],

'c': [('a', 4), ('b', 2), ('d', 1)],

'd': [('b', 5), ('c', 1)]

}

s = 'a'

print("Shortest paths:", d(g, s))

Output:

5. Optimal Tree Problem: Huffman Trees and Codes

from heapq import heappop, heappush, heapify

def e(s):

f = {ch: s.count(ch) for ch in set(s)}

h = [[wt, [ch, ""]] for ch, wt in f.items()]

heapify(h)

while len(h) > 1:

l = heappop(h)

r = heappop(h)

for pair in l[1:]:

pair[1] = '0' + pair[1]

for pair in r[1:]:

pair[1] = '1' + pair[1]

heappush(h, [l[0] + r[0]] + l[1:] + r[1:])

return sorted(heappop(h)[1:], key=lambda p: (len(p[-1]), p))

s = "huffman coding"

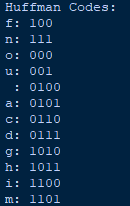
h = e(s)

print("Huffman Codes:")

for p in h:

print(f"{p[0]}: {p[1]}")

Output:



6. Container Loading

def f(w, c):

w.sort(reverse=True)

k = []

for weight in w:

if c >= weight:

c -= weight

k.append(weight)

return k

w = [10, 20, 30, 40, 50]

c = 100

print("Containers loaded:", f(w, c))

Output:

7. Minimum Spanning Tree - Kruskal's Algorithm

class G:

def \_\_init\_\_(self, vertices):

self.v = vertices

self.g = []

def a(self, u, v, w):

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

def f(self, parent, i):

if parent[i] == i:

return i

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

def u(self, parent, rank, x, y):

xroot = self.f(parent, x)

yroot = self.f(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

def kruskal(self):

result = []

i = 0

e = 0

self.g = sorted(self.g, 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.g[i]

i = i + 1

x = self.f(parent, u)

y = self.f(parent, v)

if x != y:

e = e + 1

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

self.u(parent, rank, x, y)

return result

g = G(4)

edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

for u, v, w in edges:

g.a(u, v, w)

print("Edges in MST:", g.kruskal())

Output:

8. Prim's Algorithm

import heapq

def h(g, v):

mst = []

used = {node: False for node in g}

min\_e = [(0, v, v)]

while min\_e:

w, u, v = heapq.heappop(min\_e)

if not used[v]:

used[v] = True

if u != v:

mst.append((u, v, w))

for to, weight in g[v]:

if not used[to]:

heapq.heappush(min\_e, (weight, v, to))

return mst

g = {

'a': [('b', 1), ('c', 4)],

'b': [('a', 1), ('c', 2), ('d', 5)],

'c': [('a', 4), ('b', 2), ('d', 1)],

'd': [('b', 5), ('c', 1)]

}

s = 'a'

print("Edges in MST:", h(g, s))

Output:

9. Boruvka's Algorithm

class I:

def \_\_init\_\_(self, vertices):

self.v = vertices

self.graph = []

def a(self, u, v, w):

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

def find(self, parent, i):

if parent[i] == i:

return i

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

def 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

def boruvka(self):

parent = []

rank = []

cheapest = []

numTrees = self.v

MSTweight = 0

result = []

for node in range(self.v):

parent.append(node)

rank.append(0)

cheapest.append(None)

while numTrees > 1:

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

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

set1 = self.find(parent, u)

set2 = self.find(parent, v)

if set1 != set2:

if cheapest[set1] is None or cheapest[set1][2] > w:

cheapest[set1] = [u, v, w]

if cheapest[set2] is None or cheapest[set2][2] > w:

cheapest[set2] = [u, v, w]

for node in range(self.v):

if cheapest[node] is not None:

u, v, w = cheapest[node]

set1 = self.find(parent, u)

set2 = self.find(parent, v)

if set1 != set2:

MSTweight += w

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

self.union(parent, rank, set1, set2)

numTrees -= 1

cheapest = [None] \* self.v

return result

g = I(4)

edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

for u, v, w in edges:

g.a(u, v, w)

print("Edges in MST:", g.boruvka())

Output: