```
import gurobipy as gp
import numpy as np
from gurobipy import GRB, quicksum
import csv
import math

def RMPsolver(d_i, a):
    """
    the solver for RMP
    :param d_i: the demand array
    :param a: the patterns we take into account
    :return: the optimal y_bar (shadow price), and optimal x_bar
    """
    num_i = a.shape[0] # number of type we demand
    num_j = a.shape[1]  # number of patterns we initially have
    m = gp.Model()
    m.Params.LogToConsole = 0

    m.ModelSense = 1 # it is a minimization problem
    x = m.addMVar(num_j, obj=1, vtype="C", name="x")  # x is time we use each
pattern

    constr = m.addConstrs(
        ((quicksum(a[i][j] * x[j] for j in list(range(num_j)))) >= d_i[i] for i in
list(range(num_i))))

    m.optimize()
    # find y_bar
    y_bar = [constr[i].pi for i in list(range(num_i))]  # get y_bar

    x_bar = x.X # get x_bar
    return y_bar, x_bar, m.objVal

def knapsacksolver(y, raw, alpha, num_i):
    """
    this is solver for the knapsack problem
    :param y: y is the optimal solution we get from RMP
    :param raw: the length of the raw material
    :param alpha: the length of each final we demand
    :param num_i : the number of type we demand
    :return: pi_star and z
    """
    ks = gp.Model()
    ks.Params.LogToConsole = 0
    ks.ModelSense = -1

    pi = ks.addMVar(num_i, obj=y, vtype="I", name="pi")  # pi is new pattern we
need to find
    ks.addConstr((quicksum(alpha[i] * pi[i] for i in list(range(num_i))) <= raw))
    ks.optimize()

    z = ks.objVal
    pi_star = pi.X
    return z, pi_star

def checkrep(patterns, new_pi):
    """
    check if the new pi is already included in the pattern set
    :param patterns: the set of patterns
```

```
    :param new_pi: the new optimal pi we solved from knapsack
    :return: Boolean
    """
    tmp = new_pi[:, None]
    tmp2 = patterns - tmp  # subtract the value of pi* from every column of
patterns
    for k in range(0, patterns.shape[1]):
        if np.all(tmp2[:, k] == 0): # if there is a column of all zeros, we have a
pattern that is already included
            return True
        else:
            continue
    return False


def ultimatesolver(d_i, a, raw, alpha):
    """
    implement the Delayed Column Generation algorithm
    :param d_i: the demand we have for each final
    :param a: the initial patterns we have
    :param raw: the length of the raw
    :param alpha: the length of each final we demand
    :return: the optimal solution x_bar
    """
    i = a.shape[0]  # total number of types we demand
    z = float('inf')
    newcol = 0

    while z > 1:
        y_bar = RMPsolver(d_i, a)[0] # find the optimal solution y_bar
        x_bar = RMPsolver(d_i, a)[1] # find the optimal solution x_bar
        z, pat_new = knapsacksolver(y_bar, raw, alpha, i) # the objective for
knapsack problem and the new pattern
        # if z > 1 and checkrep(a, pat_new):
        # if z > 1 + 1e-6:
        #     a = np.column_stack((a, pat_new.transpose()))  # add the new pattern
to the set
        # else:
        #     break

        if z <= 1:
            optimalval = RMPsolver(d_i, a)[2]
            break
        # elif checkrep(a, pat_new): # if the new optimal pattern is already
included, then break
        #     break
        else:
            a = np.column_stack((a, pat_new.transpose()))  # add the new pattern to
the set
            newcol += 1  # we generate one new column

    print("Optimal value is: ",optimalval)
    # print(z)
    print("Optimal solution is: ", x_bar)
    print("Number of new columns generated is: ", newcol)
    return x_bar


def datagenerator(filelocat):
```

```
    """
    this function reads the text file and generates the information we need for the
solver
    :param filelocat: the location the text files stored on the computer
    :return: demand_num : num of demand for each final
             patt: the initial set of patterns
             raw_length: the length of the raw materials
             demand_length: the length of each final we need
    """
    with open(filelocat, 'r') as f:
        fread = csv.reader(f, delimiter='\t') # read the input text file
        output = []
        for row in fread:
            output.append(row) # convert txt file to a list

    rawlen = []
    demand_length = []
    demand_num = []

    for line in output:
        if len(line) == 1:
            rawlen.append(line) # identify the raw length
        else:
            demand_length.append(int(line[0])) # identify the length of demand
            demand_num.append(int(line[1])) # identify the number of demand for
each final

    demand_num = np.array(demand_num)
    demand_length = np.array(demand_length)

    raw_length = int((rawlen[1][0]))
    num_type_demand = int((rawlen[0][0])) # identify the total number of finals

    patt = np.zeros((num_type_demand, num_type_demand)) # initialize the matrix for
patterns
    for k in range(0, num_type_demand):
        patt[k][k] = math.floor(raw_length / demand_length[k]) # generate the basic
patterns where only making one type from each

    return demand_num, patt, raw_length, demand_length


d_i1, patt1, raw1, alpha1 = datagenerator('C://Users//admin//Desktop//Rice//2023
spring//caam476//Scholl_CSP//Scholl_CSP//Scholl_1//N1C1W1_A.txt')
ultimatesolver(d_i1, patt1, raw1, alpha1)

# d_i2, patt2, raw2, alpha2 = datagenerator('C://Users//admin//Desktop//Rice//2023
spring//caam476//Scholl_CSP//Scholl_CSP//Scholl_2//N1W1B1R0.txt')
# ultimatesolver(d_i2, patt2, raw2, alpha2)


# dd = np.array([97, 610, 395, 211])  # demand of each type
# aa = np.array([[2, 0, 0, 0],
#                [0, 2, 0, 0],
#                [0, 0, 3, 0],
#                [0, 0, 0, 7]])
# rr = 100  # our length of raw
# alp = np.array([45, 36, 31, 14]) # length of our finals
#
```

```
# print(aa)
# print(ultimatesolver(dd, aa, rr, alp))
```