**Task 1:**

**READING IN THE INSTANCE**

The first line of the txt file was loaded in as m and n, where n is the number of rows and n is the number of columns. A was first initialized as a (m\*n) zero matrix. The C was first created an n zeros vector. The dataset is in a format that, the costs are the next n elements after the first line. The n elements were extracted and used to replace the C vector. The next number after that represents a row index, say r. The next r values are the columns to be changes in a row. The first list of columns corresponds to the first row of the A matrix. This is repeated until the end of the dataset with the A matrix being updated as such, to treat this as a binary optimization problem. This asks to ask the user to input the filename. This is done, however has been commented, due to the running of task 5. Regardless, the task 5 has also been made to ask for the list of test instances.

Task 2:

CONSTUCTIVE HEURISTICS

This was built using the greedy algorithm approach. Started off by setting a solution vector of zeros with the length of the cost variable and named it x. The problem is to:

Minimize C^Tx; subject to: Ax>=1. From this, the aim of this algorithm is to start off with the entire matrix and taking out rows which satisfy the conditions Ax>=1. Which means, once the row has a 1 value present, it satisfies the constraints. This is done by selecting the lowest cost per unsatisfied constrained ratio. This is done iteratively until the final optimal solution is realized. Tested this on a small data which had a row full of zeros and got stuck in an infinite loop, because of unsatisfiable constraints. To address this issue, the algorithm was modified to terminate when no more improvement can be made, even if not all constraints are satisfied. This can however lead to suboptimal optimization.

**Pseudocode for this algorithm:**

Function Constructive\_heuristics(C, A):

1. Initialize variables:

* n is the length of C (number of variables)
* x is a list of zeros with length n (solution vector)
* satisfied\_const is a list of zeros which has a length equal to the number of constraints.

2. Start a loop that continues until all constraints are satisfied:

* ratios list object is created as an empty object to store the cost to unsatisfied constraint ratio.
* Calculate the cost to unsatisfied constraint ratio for each variable i in range(n) and store in ratios if it is not yet covered.
* If the variable is not selected, set it to infinity.
* Find the index of the variable with the minimum ratio.
* Terminate the loop if the minimum ratio is infinity, which means no further improvements can be made.
* Update the x vector by selecting the variable with the minimum ratio.
* Update the satisfied\_const vector.

3. Calculate the final cost and then return the final cost and the solution associated with it.

**TASK 3**

**Local Search**

Started off with a simple hill climbing local search algorithm. This involved defining a local search with the cost, constraint matrix, initial solution, and the number of iterations as parameters. This function is to call another function to make a small change in the neighbourhood by swapping 0 and 1 values in random positions. This is to be done iteratively the number of iterations given to get a better solution. Hill climbing does not accept worse solutions than the previous one. This was tested on several small instances, but no improvement was realized. Several techniques were used to try improving this algorithm, including swapping multiple 0s and 1s. This did not result in any improvements also. The partial neighoburhood local search technique was implemented. This was done by incorporating a neighbourhood size parameter in the local search function. This defines the number of neighbouring solutions to consider at each step of the search process. A bigger size allows exploring broader areas but computationally expensive. A success rate (0-1) was introduced in the neighbourhood function. This is to measure how a particular neighbourhood leads to improvement in solution. To do this, an improvement history must be kept which keep the indexes of neighbourhoods that have improved solution. This is used to update the success rate, that is, the ratio of times that neighbourhood has led to an improvement to the total length of the improvement history.

This task asks to ask the user to put the number of iterations and save solution and cost in a csv. These are done but commented to not disturb the running of the codes during task 5. Regardless, saving the csv has also been incorporated into the task 5 to perform this task whilst iteratively finding solutions to the test instances.

**Pseudocode**

The local\_search function code:

1. Copy starting solution and call it curr\_best. Calculate cost using dot product of costs and starting solution, store in curr\_best\_cost.
2. Make list neigh\_success\_rates with equal chance for each neighborhood (1 divided by neighborhood size) and create empty list improve\_history.
3. Do steps for the number of iterations given(num\_searches):
   * Use get\_neighbour function with current\_best, A, and neigh\_success\_rates as inputs. Store the results as new solution and neighbourhood respectively.
   * Calculate cost of new solution with dot product of costs and new solution, and save in new\_solution\_cost.
   * If new solution cost less than current best cost, update curr\_best and curr\_best\_cost to take the new solution and the new cost.
   * Add current neighbourhood to improvement history list.If list is getting too long, that is the list is greater than the neighbourhood size(neigh\_size), remove the oldest history.
   * Update success rates for each neighborhood by counting how many times each neighborhood in improvement history list and divide by list length.

4. Return best solution (curr\_best) and best cost (curr\_best\_cost).

The get\_neighbour function pseudocode is as follows:

1. Define the function get\_neighbour that takes the following inputs: sol, constraint\_matrix which is the A matrix, and \_success\_rate.
2. Create a copy of the given solution called new\_sol.
3. Find the positions of 0's and 1's in the solution and store them as zeros and ones.
4. If there are no 0's or 1's, return the original solution unchanged.
5. Randomly select index based on probability provided by \_success\_rate and store in neighbourhood.
6. Randomly choose a position with a 0 (random0) and a position with a 1 (random1).
7. Swap the values at the chosen positions in the new\_sol.
8. Check if the new\_sol satisfies the constraints(Ax>=1):
   * If the new\_sol satisfies the constraints, return new\_sol.
   * b. Otherwise, return the original solution (sol).

**TASK 4**

**SIMULATED ANNEALING**

To further try and improve solution, simulated annealing is going to be used to try improving from the solution of the local search. This combines the local search with momentary jumps in an attempt to escape the local minima. This is done by accepting better solutions and even sometimes worse solutions during the search process according to some standards. This algorithm is implemented in this work by first defining a simulated annealing function. The function these parameters; cost, constraints, initial solution, number of iterations, initial temperature, and the cooling rate. The initial solution will be the solution from the local search results. The number of iterations defines the number of times to make a move in attempt to get a better solution, the initial temperature controls the chances of accepting a worse solution in the beginning of a search. A higher temperature results in high exploration, which however is computationally expensive. The cooling rate defines the level at which the temperature decreases during the search. I higher cooling rate means, slower cooling (0-1).

This algorithm is executed by making a small change, which enhances the chances of solution improvement and new possibilities. A help function (make\_small\_change) is created to make the small change by swapping the positions of random 1s and 0s as in the local search implementation. This is to get a new solution, then the Boltzmann distribution will be used to calculate an acceptance probability which sometimes help the algorithm to accept worse solutions. A threshold is set, since there were some warnings received during running to prevent an overflow if the acceptance probability if the ratio of the relative improvement between the current cost and the new cost to the temperature is very large in the Boltzmann distribution. The temperature is updated in each iteration according to the cooling rate, which influences the acceptance probability to accept worse solutions. This happens until the number of iterations is finally reached.

**Pseudocode**

1. **make\_small\_change** function:

* Input: sol, constraint\_matrix
* Make copy of sol, and call it new\_sol
* Create empty lists zeros and ones that take the positions of 0’s and 1’s in the solution.
* Find the positions of the 0's and 1's in solution
* If there are no 0's or 1's, return the unmodified solution
* Choose random positions to swap 0’s and 1’s
* Interchange values at chosen positions.
* Check if new solution satisfies the constraints.
* Return new solution if it satisfies the constraints, else return the original solution.

1. **simulated\_annealing** function:
   * Input: costs, constraints, initial\_solution, max\_iter, initial\_temp, alpha
   * Copy initial\_solution and store it as curr\_solution.
   * Find its cost, by finding the dot product of the costs and initial solution and store it as curr\_cost
   * Set best\_solution to initial\_solution, best\_cost to curr\_cost
   * Set a threshold to a very small number to avoid overflow during the calculation of the acceptance probability.

For each iteration from 1 to max\_iter, do following steps:

* + Calculate current temperature using alpha, initial temperature and the current iteration number and store as temp.
  + Make new\_solution by calling make\_small\_change function with curr\_solution and constraints .
  + Calculate new\_cost by multiplying costs and new\_solution
  + Find the acceptance probability using the idea from the Boltzmann distribution.
  + If new\_cost is less than current\_cost or a random number less than the acceptance probability, update curr\_solution, curr\_cost with the new solution and the new cost.
  + If new\_cost is less than best\_cost, update best\_solution and best\_cost
  + Return best\_solution, best\_cost.

TASK 5:

This task is to find results from the various heuristics. This was done by using a loop to run through the test instances, iterating over certain parameters and selecting the best solution based on those parameters.

RESULTS

Due to time consumption and computational complexities, the least large files (SCP\_01.txt, SCP\_02.txt, SCP\_03.txt, SCP\_07.txt, SCP\_08.txt and SCP\_09.txt) were put in a loop, where for local search, considered number of iterations of 10000 and 5000, and neighbourhood sizes of 3 and 5. For simulated annealing, maximum iteration of 5000 and 10000, start temperature of 200 and 100, and a cooling rate(alpha) of 0.95 and 0.99 were considered. The results are shown in the table 1 below. The local search parameters will be in the format (number of iterations, neigbourhood size), and the simulated annealing (number of iterations, start temperature, alpha)

Table 1 results from test instances

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test instances | Constructive heuristics/ optimal value | Local search/ (parameters), optimal value | Simulated annealing/(parameters), optimal value | Number of variables in solution |
| SCP\_01.txt | 505 | (10000,3), 503 | (10000,200,0.95), 503 | 9 |
| SCP\_02.txt | 306 | (10000,3), 306 | (10000,200,0.95), 306 | 6 |
| SCP\_03.txt | 257 | (10000,3), 257 | (10000,200,0.95), 257 | 5 |
| SCP\_07.txt | 1510 | (10000,5), 1497 | (5000,200,0.95), 1497 | 27 |
| SCP\_08.txt | 5596 | (10000,5), 5590 | (5000,200,0.95),5590 | 101 |
| SCP\_09.txt | 11782 | (10000,3), 11780 | (10000,100,0.99), 11780 | 210 |

For the very large datasets (SCP\_04.txt, SCP\_05.txt, SCP\_06.txt, and SCP\_10.txt), constant parameters were set for them to find the optimal solutions. For the local search, a neighbourhood size of 3 and 1000 for the number of searches. The parameters for the simulated annealing were maximum iteration of 5000, initial temperature of 300 and a cooling rate of 0.95. The results are shown in table 2 below.

Table 2 results from test instances

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test instances | Constructive heuristics/ optimal value | Local search/ optimal value | Simulated annealing/ optimal value | Number of variables in solution |
| SCP\_04.txt | 514 | 509 | 509 | 9 |
| SCP\_05.txt | 677 | 677 | 677 | 13 |
| SCP\_06.txt | 972 | 971 | 971 | 19 |
| SCP\_10.txt | 20092 | 20089 | 20072 | 363 |

The variable positions from the local search saved in the csv’s are shown in the table 3 below:

Table 3 variable positions for local search from saved csv’s

|  |  |
| --- | --- |
| Test instance | Variable positions |
| SCP\_01.txt | 120, 563, 825, 1737, 2412, 2510, 2621, 2643, 2820 |
| SCP\_02.txt | 367, 1033, 1041, 3510, 8796, 10468 |
| SCP\_03.txt | 2016, 5910, 11354, 22302, 22660 |
| SCP\_04.txt | 5090, 9672, 10008, 10518, 13440, 15986, 16826, 22252, 26908, 39435 |
| SCP\_05.txt | 1730, 1985, 4272, 6822, 9794, 9960, 12650, 19110, 23016, 24696, 25248, 27696, 37250 |
| SCP\_06.txt | 3697, 11161, 18240, 18750, 19610,21846, 35814, 39604, 40708, 43248, 43345, 44496, 47260, 49254, 68428, 70491, 72482, 73926  ,86020 |
| SCP\_07.txt | 87,188,262,380,658,828,866,1079,1128,1154,1272,1517, 1612,1668,1734,1808,1830,1838,1893,1958,2024,2028, 2086,2253,2268,2466,2749 |
| SCP\_08.txt | 32,176,367,407,456,525,641,704,815,843,898,920,  983,1079,1211,1217,1346,1592,1998,2079,2128, 2148,2185,2218,2364,2597,2670,2788,2792,2904,  3124,3294,3372,3379,3416,3499,3546,3547,3656, 3713,3765,3862,4040,4384,4446,4824,5154,5212, 5220,5424,5480,5494,5533,5537,5801,6011,6157, 6197,6386,6440,6455,6782,6798,6878,6885,6922, 6988,7152,7206,7303,7404,7602,7616,7648,7739, 7747,7756,8120,8124,8229,8260,8408,8424,8504, 8566,8600,8694,8800,8870,8971,9032,9054,9154,9190, 9197,9396,9403,9410,9438,10316,10405 |
| SCP\_09.txt | 18,119,905,942,1097,1148,1240,1412,1531,1714, 1917,2074,2338,2406,2436,2621,2747,2774,2891, 2978,3038,3153,3238,3414,3422,3449,3530,3566, 3569,3604,3655,3804,3850,3883,4286,4317,4371, 4438,4636,4802,4823,4830,4876,4896,5044,5136, 5171,5230,5236,5240,5346,5450,5477,5480,5512, 5666,5844,5962,6012,6102,6228,6418,6442,6595, 6722,6742,6864,6926,6990,7146,7165,7234,7330, 7344,7380,7568,7834,7904,8104,8120,8302,8493, 8687,8734,9142,9216,9254,9276,9329,9576,9582, 9599,9821,9944,10036,10334,10372,10496,10802, 10856,11134,11379,11582,11639,11802,11940,11971, 12046,12115,12172,12185,12252,12281,12490,12538, 12752,12877,13223,13420,13442,13475,13598,13702, 13736,13826,13859,14286,14352,14524,14811,14825, 14916,14923,14929,15002,15322,15335,15414,15511, 15528,15765,15873,16088,16098,16244,16282,16603, 16609,16784,16806,16853,16946,17053,17133,17174, 17340,17366,17508,17626,17651,17698,17932,18151, 18216,18242,18253,18283,18384,18479,18570,18704, 18729,18765,18768,18882,18953,19082,19180,19232, 19446,19496,19559,19670,19919,20099,20120,20133, 20172,20197,20506,20534,20840,20908,21030,21082, 21390,21410,21424,21550,21695,21860,21902,22031, 22046,22186,22684,22739,22860,22912,22931 |
| SCP\_10.txt | 103,112,284,1070,1110,1211,1474,1490,1624, 1861,  1883,2011,2126,2175,2234,2242,2486, 2539,2750,  2755,2887,3033,3164,3177,3240, 3317,3354,3396,3575  ,3613,3626,3720,3838, 3854,3910,3917,4116,4188,4280,  4310,4434, 4505,4536,4676,4754,4762,4836,4852,5012, 5155,5230,5414,5460,5522,5635,5643,5856,6028,6216,  6388,6390,6604,6695,6816,6974,6990,6996,7193,7330,  7348,7503,7518,7522,7762,7932, 8150,8300,8350,8403,  8424,8456,8512,8552,8575, 8828,9022,9076,9144,9468,  9476,9886,9915, 9972,9992,9993,9998,10103,10701,10716,  10761,10904,11006,11253,11586,11626,11783,11835,11906,  11958,12202,12251,12317,12536,12578,12588,12809,12833,  12848,13029,13117,13164,13211,13412,13511,13583,13584,  13637,14018,14044,14134,14180,14297,14335,14635,14731,  15559,15613,15686,15746,15994,16068,16114,16179,16247,  16258,16562,16584,16657,16675,16794,16866,16937,16956,  16990,17046,17102,17120,17237,17364,17399,17428,17475,  17558,17578,17640,17695,17861,17904,18074,18078,18143,  18379,18512,18547,18854,18944,19179,19261,19266,19379,  19678,19698,19760,19814,19830,19902,19986,20028,20033,  20096,20342,20404,20568,20654,20872,21001,21032,21034,  21113,21130,21335,21365,21438,21487,21946,21962,22152,  22400,22546,22633,22788,22795,22977,23042,23057,23170,  23321,23364,23444,23713,23741,23744,23786,23899,23936,  24058,24147,24538,24575,24764,24777,24916,25120,25224,  25347,25651,25734,25807,25872,26040,26182,26287,26443,  26640,26722,26910,26962,26982,27264,27280,27375,27503,  27510,27857,28014,28240,28260,28360,28428,28453,28572,  28648,28894,28937,29013,29205,29354,29606,29774,30030,  30091,30104,30122,30156,30172,30289,30320,30433,30493,  30810,31044,31138,31268,31459,31766,31879,32388,32588,  32897,32919,32993,33058,33084,33398,33420,33521,33538,  33540,33683,33714,33720,33828,33936,33948,34112,34128,  34320,34332,34459,34492,34514,34584,34616,34725,34982,  35024,35050,35200,35236,35388,35552,35558,35567,35637,  35822,35826,36126,36130,36248,36358,36728,36855,36866,  36918,36944,36953,37026,37192,37203,37210,37302,37444,  37542,37550,37721,37910,37916,38127,38316,38345,38560,  38644,38860,38940,39194,39285,39358,39406,39484,39636,  40084,40100,40166 |

From the results, it is noticeable that, the results from the local search is mostly the same to the simulated annealing and some of the constructive heuristics results remained the same throughout. Every dataset is different and therefore may require different parameters to find a better solution. Not all parameters were explored. In the end, it is up to the dataset and the person performing the task to select the parameters based on the details.

The codes for the entire work is saved in the txt document which submitted in addition to this file.

RECOMMENDATIONS

This implementation however works fine, can undergo massive improvement, especially to improve the computational time. Limiting knowledge on some library functions resulted to the use of several lists and loops which affects the running speed. Also, due to the large datasets, it was difficult to explore larger iteration numbers and higher temperatures which may have improved solution even further.

REFERENCES

Task 1

* Evolutionary Search Techniques to Solve Set Covering Problems (link:https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=65188cb77acc0d1072f00d56aad81c9837c198c6)

Task 2

* Design and supply chains (link: <https://youtu.be/MJ9uK2NtWxw>)
* Python pool (<https://www.pythonpool.com/knapsack-problem-python/>)
* [AndreaRubbi](https://github.com/AndreaRubbi) (<https://github.com/AndreaRubbi/Set-Cover-problem-solution-Python/blob/master/Greedy.py>)

Task3:

* Machine Learning Mastery (<https://machinelearningmastery.com/iterated-local-search-from-scratch-in-python/>)
* Partial Neighborhood local searches (<https://onlinelibrary.wiley.com/doi/10.1111/itor.12983>) – Sara Tari, Matthieu Basseur, Adrien Goëffon
* [rafia37](https://github.com/rafia37) (<https://github.com/rafia37/Local-Search/blob/master/guided_local_search.py>)

Task 4

* Machine Learning Mastery (<https://machinelearningmastery.com/simulated-annealing-from-scratch-in-python/>)
* Re-heat simulated annealing algorithm for rough set attribute reduction(https://academicjournals.org/journal/IJPS/article-full-text-pdf/D6CF1F025890)
* Nathan (<https://nathanrooy.github.io/posts/2020-05-14/simulated-annealing-with-python/>)
* [perrygeo](https://github.com/perrygeo/simanneal/commits?author=perrygeo) (<https://github.com/perrygeo/simanneal>)

**CODES USED**

import numpy as np  
import math  
import random  
import csv  
  
# i would want to acknowledge the creators of all the libraries used in this work  
  
# file\_name = input("input the test instance: ")  
#Task 1  
#Read in problem instance file  
  
def read\_file(file\_name):  
 with open(file\_name, "r") as file:  
 contents = file.read()  
 #splitting the characters into individual characters  
 char = contents.split()  
 # first two numbers, rows(m) and columns(n)  
 m,n = int(char[0]), int(char[1])  
 #creating A matix and C vector  
 A= np.zeros((m,n))  
 C = np.zeros(n)  
 #iterating to fill the C vector  
 i = 0  
 for num in char[2:]: # numbers after the first two characters  
 C[i] = int(num)  
 i += 1  
 if i == n:  
 break  
  
 # iterating over the remaining of the numbers and filling up the elements of a  
 i = n + 2 # after the cost characters  
 rows = [] # empty lsit to store the row indexes  
 cols = [] # empty list to store the corresponding columns  
  
 while i < len(char):  
 # getting the row index  
 row\_ind = int(char[i])  
 rows.append(row\_ind)  
 i += 1  
 # getting the columns for the row index  
 col\_ind = char[i:i + row\_ind]  
 cols.append(col\_ind)  
 i+= row\_ind  
 # stopping loop when there are no more characters  
 if i == len(char):  
 break  
 # convert the strings to integers  
 cols = [[int(col) for col in row] for row in cols]  
 #updating the A matrix  
 for row\_A, col\_A in zip(A, cols):  
 for c in col\_A:  
 row\_A[c-1]= 1  
 #print (rows)  
 print(rows)  
 print((sum([1 for val in A[3] if val ==1])))  
 print(A[16])  
  
 return A, C  
# A, C = read\_file("SCP\_01.txt")  
  
# task 2  
# Calculate the solution using a greedy algorithm  
def Constructive\_heuristics(C, A):  
 #initialise variables  
 n = len(C)  
 x = [0] \* n #solution vector x, set to zeros with equal length as length of C  
 satisfied\_const = [0] \* len(A) # constraint satisfaction  
  
 # iterating until all constraints are satisfied  
 while not all(satisfied\_const):  
 # calculate the ratio of cost to unsatisfied variables with at least on unsatisfied constraint  
 ratios = [] # list to store the ratios  
 for i in range(n):  
 # considering only yet uncovered variables with at least one unsatisfied constraint  
 sum\_var = np.sum(A[:, i] \* ( 1 - np.array(satisfied\_const)))  
 if x[i] == 0 and sum\_var > 0:  
 ratio = C[i]/sum\_var  
 else:  
 ratio = float("inf") # set the ratio to infinity if the variable cannot be selected  
 ratios.append(ratio)  
 # selecting variable with minimum ratio  
 min\_idx = ratios.index(min(ratios))  
 # end the loop if no further improvements can be made  
 if ratios[min\_idx] == float("inf"):  
 break  
 # updating the solution and constraint vector  
 x[min\_idx] = 1  
 for j in range(len(A)):  
 satisfied\_const[j] = max(satisfied\_const[j], A[j][min\_idx])  
  
 value = sum([C[i] \* x[i] for i in range(n)]) # final cost (C^Tx)  
 # return the solution and the cost  
 return x, value  
# call the greedy heuristics and print results  
# solution, value = Constructive\_heuristics(C,A)  
# print("Solution:" , solution)  
# print ("Cost: ", value)  
# print(sum([1 for val in solution if val == 1]))  
  
  
# Task 3  
def local\_search(costs, A, start\_solution, num\_searches, neigh\_size = 3):  
 curr\_best = np.copy(start\_solution) #copy the start solution as current best  
 curr\_best\_cost = np.dot(costs, start\_solution) # current best cost  
 # initialising the neighbourhood success rates and history  
 neigh\_success\_rates = [1/ neigh\_size] \* neigh\_size  
 improve\_history = [] # improvement history  
 # loop the "num\_searches" times to find a better solution  
 for w in range(num\_searches):  
 # calling the get\_ neighbour function to get a new solution and neighbourhoood  
 new\_solution, neighbourhood = get\_neighbour(curr\_best, A , neigh\_success\_rates)  
 # calculating the cost of the new solution  
 new\_solution\_cost = np.dot(costs, new\_solution)  
 # checking if the new solution has a better cost than the current solution  
 if new\_solution\_cost < curr\_best\_cost:  
 #updating the current best solution with the new solution if the condition is met  
 curr\_best = new\_solution  
 curr\_best\_cost = new\_solution\_cost  
 #updating improvement history and success rates  
 improve\_history.append(neighbourhood)  
 # remove the oldest history if the history becomes too long  
 if len(improve\_history)> neigh\_size:  
 improve\_history.pop(0)  
 # update success rate for each neighbourhood  
 for i in range(neigh\_size):  
 neigh\_success\_rates[i] = improve\_history.count(i)/ len(improve\_history)  
  
 return curr\_best, curr\_best\_cost  
  
# the neighbourhood function to make small change  
def get\_neighbour(sol, constraint\_matrix, \_success\_rates):  
 new\_sol = np.copy(sol)  
 # finding the positions of 0's and 1's in the solution  
 zeros = [] # zero positions  
 ones = [] #one positions  
 # iterating to store the positions of 0's and 1's  
 for i in range(len(sol)):  
 if sol[i] == 0:  
 zeros.append(i)  
 elif sol[i] == 1:  
 ones.append(i)  
 # if there are no more 0's and 1's, return the unchanged solution  
 if len(zeros) == 0 or len(ones) ==0:  
 return new\_sol  
 # selecting neighbourhood based on success rates  
 neighbourhood = np.random.choice(len(\_success\_rates), p = \_success\_rates)  
 # interchanging the values in the neighbourhood randomly and picking the positions from the zeros and ones lists  
 random0 = np.random.choice(zeros)  
 random1 = np.random.choice(ones)  
 # interchanging the values at the random positions in the new solution  
 new\_sol[random0], new\_sol[random1] = new\_sol[random1], new\_sol[random0]  
 #checking if the solution the constraints (Ax>=1)  
 if np.all(np.dot(constraint\_matrix, new\_sol) >=1):  
 # return the new solution and the neighbourhood if condition is satisfied  
 return new\_sol, neighbourhood  
 else:  
 # else return the original solution  
 return sol, None  
  
# # get the number of searches from the user  
# num\_searches = int(input("How many times do you want to search: "))  
#  
# #using solution from the constructive heuristics as intial solution  
# n = len(C) # number of variables  
# start\_solution = np.copy(solution)  
# #perform local search  
# LS\_solution, LS\_cost = local\_search(C, A, start\_solution, num\_searches)  
#  
# # print the results  
# print("Final solution: ", LS\_solution)  
# print("final cost: ", LS\_cost)  
# print(sum([1 for val in LS\_solution if val == 1]))  
  
# # saving best solution to csv  
# with open(f"{file\_name}.csv", "w", newline="") as csvfile:  
# csvwriter = csv.writer(csvfile)  
# for i in range(len(LS\_solution)):  
# csvwriter.writerow([i + 1, LS\_solution[i]])  
# csvwriter.writerow(["Cost", LS\_cost])  
  
  
# Task 4  
# Modified get\_neighbour function without neighborhood\_success\_rates parameter  
def make\_small\_change(sol, constraint\_matrix):  
 new\_sol = np.copy(sol)  
 # Finding the positions of 0's and 1's in the solution  
 zeros = [] # zero positions  
 ones = [] # one positions  
 for i in range(len(sol)):  
 if sol[i] == 0:  
 zeros.append(i)  
 elif sol[i] == 1:  
 ones.append(i)  
 # return unmodified soultion if there are no 1's and 0's present  
 if len(zeros) == 0 or len(ones) == 0:  
 return new\_sol  
 #Select random zero and one positions to swap  
 random\_zero = np.random.choice(zeros)  
 random\_one = np.random.choice(ones)  
 # interchanging the values at the chosen positions  
 new\_sol[random\_zero], new\_sol[random\_one] = new\_sol[random\_one], new\_sol[random\_zero]  
 # Check if the new solution satisfies the constraints  
 if np.all(np.dot(constraint\_matrix, new\_sol) >= 1):  
 return new\_sol  
 #return original solution if condition is not satisfied  
 else:  
 return sol  
  
  
  
# Simulated annealing function  
def simulated\_annealing(costs, constraints, initial\_solution, num\_iter, initial\_temp, alpha):  
 # getting the initial solution and calculates its cost  
 curr\_solution =initial\_solution.copy()  
 curr\_cost = np.dot(costs, initial\_solution)  
  
 # Keep track of the best solution and the best cost storing the initial solution and the current cost as best so far.  
 best\_solution = initial\_solution.copy()  
 best\_cost =curr\_cost  
 # Adding threshold to avoid overflow when calculating acceptance probability  
 threshold = -1e308 # to prevent an overflow if the exponent in the acceptance probabilty is too large, that is  
 # if "(C(s) - C(s')) / T" is to large from the Boltzmann distribution: P = exp((C(s) - C(s')) / T).  
  
 # Iterating for the given number of iterations  
 for iteration in range(num\_iter):  
 # Calculate the current temperature using alpha and the iteration it is on  
 temp = initial\_temp \* (alpha \*\* iteration)  
  
 # Creating a new solution by making a small change to the current solution  
 new\_solution = make\_small\_change(curr\_solution, constraints)  
 new\_cost = np.dot(costs, new\_solution)  
  
 # Calculating the probability of accepting the new solution  
 exponent = min((curr\_cost - new\_cost) / temp, threshold) #relative improvement compared to previous soln  
 accept\_prob = math.exp(exponent) #probability acceptance to escape local minima (boltzmann distribution)  
  
 # If new soltuin is better or random number is less than the acceptance probability  
 if new\_cost < curr\_cost or random.random() < accept\_prob:  
 curr\_solution = new\_solution  
 curr\_cost = new\_cost  
 # Update the best solution and cost if the new solution is better  
 if new\_cost < best\_cost:  
 best\_solution = new\_solution  
 best\_cost = new\_cost  
  
 return best\_solution,best\_cost  
  
#  
# # Parameters for simulated annealing  
# num\_iter = 10000 #maximum number of iteration  
# initial\_temp = 300  
# alpha = 0.95  
#  
# # Call simulated annealing and print the results  
# sa\_solution, sa\_cost = simulated\_annealing(C, A, LS\_solution, num\_iter, initial\_temp, alpha)  
# print("Simulated Annealing Solution:", sa\_solution)  
# print("Simulated Annealing Cost:", sa\_cost)  
# print(sum([1 for val in sa\_solution if val == 1]))  
  
  
#Task 5  
  
# List of test instances  
#get instances from user  
get\_input = input("Enter test file(separate different files with commas): ")  
test\_instances = get\_input.split(",")  
  
# defining parameters for local search and simulated annealing  
# local search  
num\_iter = [10000,5000]  
neighbour\_size = [3,5]  
# for simulated annealiing  
max\_iter = [5000,10000]  
start\_temp = [200, 100]  
alpha = [0.95, 0.99]  
  
  
#load the files and run them  
for file in test\_instances:  
 # loading the txt file  
 A, C = read\_file(file)  
 # Run the constructive heuristics  
 greedy\_soln, greedy\_value = Constructive\_heuristics(C,A)  
 print(f" test {file} results for constructive heuristics: {greedy\_value} ")  
  
 # run local search  
 # to find the best results and corresponding parameters  
 # initialising  
 best\_ls\_results = float("inf")  
 best\_ls\_param = None  
 best\_ls\_soln = None  
  
 for size in neighbour\_size:  
 for iter in num\_iter:  
 ls\_solution, ls\_cost\_Value = local\_search(C, A, greedy\_soln,num\_searches= iter, neigh\_size=size)  
 # finding the best solution out of the combinations  
 if ls\_cost\_Value < best\_ls\_results:  
 best\_ls\_results = ls\_cost\_Value  
 best\_ls\_param = (size,iter)  
 best\_ls\_soln = ls\_solution  
  
 print(f"test {file} results for local search: {best\_ls\_results} with parameters {best\_ls\_param}")  
 # Saving the best solution to a CSV file  
 with open(f"{file}.csv", "w", newline="") as csvfile:  
 csvwriter = csv.writer(csvfile)  
 for i in range(len(best\_ls\_soln)):  
 csvwriter.writerow([i + 1, best\_ls\_soln[i]])  
 csvwriter.writerow(["Cost", best\_ls\_results])  
 #for simulateed annealing  
 #finding best results using best results from the local search as initial solution  
 best\_SA\_results = float("inf")  
 best\_SA\_paramters = None  
 best\_SA\_soln = None  
  
 for search in max\_iter:  
 for temp in start\_temp:  
 for a in alpha:  
 SA\_solution, SA\_best\_value = simulated\_annealing(C, A, best\_ls\_soln, num\_iter=search, initial\_temp=temp,  
 alpha=a)  
 # finding the best solution out of the combinations  
 if SA\_best\_value < best\_SA\_results:  
 best\_SA\_results = SA\_best\_value  
 best\_SA\_paramters = (search, temp, a)  
 #best\_SA\_soln = SA\_solution  
 print(  
 f"Best simulated annealing result for {file}: {best\_ls\_results} with parameters {best\_SA\_paramters}")