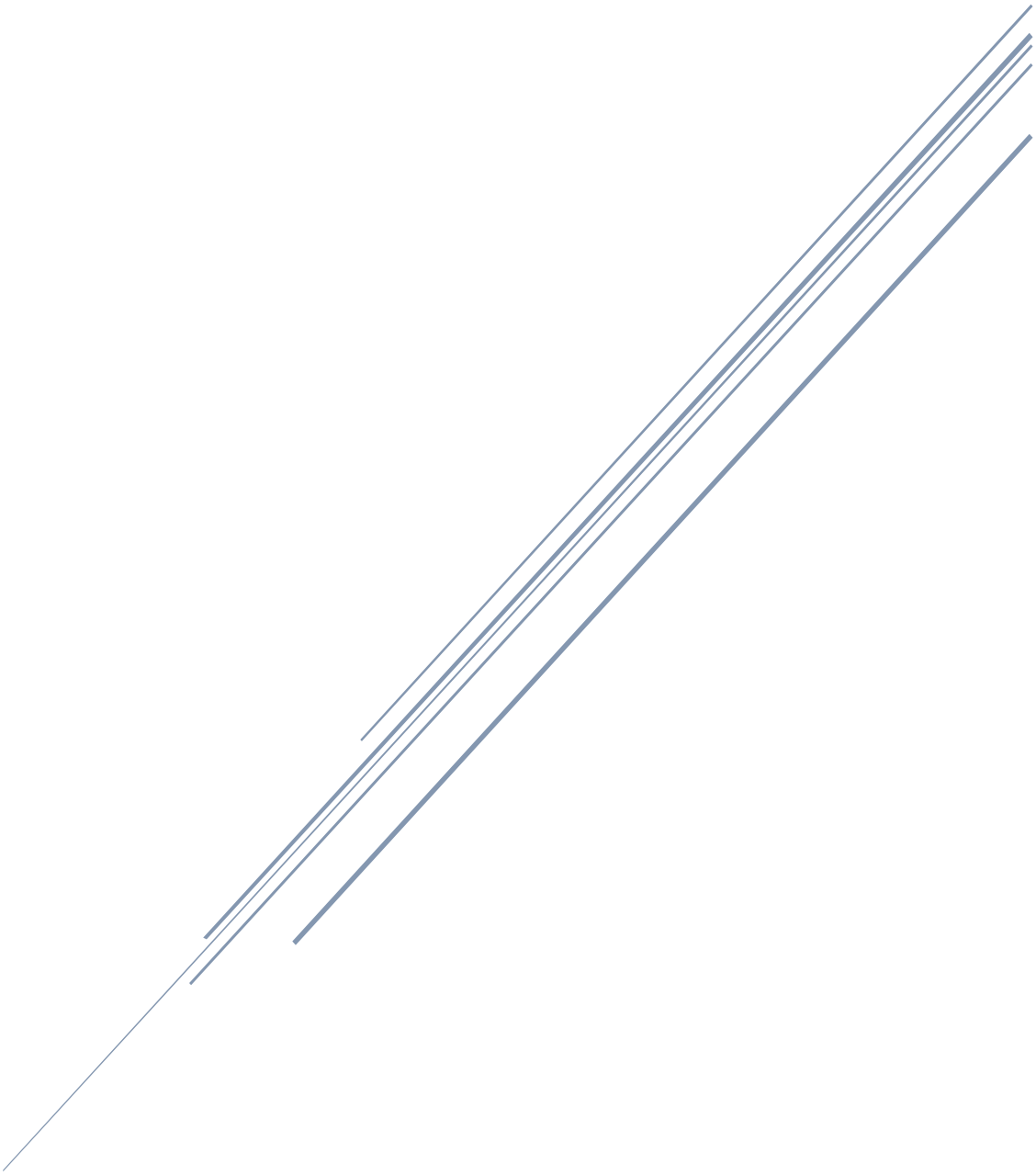
HOMEWORK2 – SIMULATED ANNEALING ON GENERAL ASSIGNMENT PROBLEM

IE 517 - Spring17



Burak Suyunu
2012400156

Important Note: This report is a summary of my detailed work. So, please check the Jupyter Notebook that I have created in https://github.com/suyunu/SA-GAP repository. You can directly view the code from here: http://nbviewer.jupyter.org/github/suyunu/SA-GAP/blob/master/sa-gap.ipynb . You can find the detailed process of the code with heavy commenting. Also, there are lots of explanatory outputs and plots of my analysis.

Introduction

In this project, we tried to solve General Assignment Problem (GAP) with Simulated Annealing (SA).

General Assignment Problem

There are agents and tasks. Any agent can be assigned to perform any task, incurring some cost. Moreover, each agent has a budget and the sum of the costs of tasks assigned to it cannot exceed this budget. It is required to find an assignment in which all agents do not exceed their budget and total cost of the assignment is minimized.

Solution Representation

To represent the solution, I used a simple list structure. Cells of the list are jobs and numbers in the cells are the agents assigned to that job.

Example solution representation:

[7, 1, 0, 4, 4, 2, 4, 1, 5, 6, 3, 4, 5, 5, 1, 5, 2, 1, 4, 0, 7, 7, 5, 7, 5, 2, 3, 0, 2, 1, 7, 6, 6, 7, 3, 4, 3, 6, 3, 6]

Algorithm

We used simulated annealing with some modification to solve General Assignment Problem. First, I will write the pseudocode of my simulated annealing and then try to explain important parts like neighborhood structure and stopping condition

Pseudocode

1. Compute an initial solution X and choose an initial temperature $T > 0$ and a repetition factor L
2. As long as the stopping criterion is not satisfied perform the following steps:
    A. Do the following L_k times
        a. Generate a random solution in the neighborhood $x' \in N(x)$.
        b. Compute $\Delta = \mathrm{f(x')} - \mathrm{f(x)} + \mathrm{unfit(x')}$
        c. Generate a random number $0 < \mathrm{u} < 1$
        d. If $\Delta < 0$ oru $< e^{-\frac{\Delta}{T_k}}$ then set $\mathrm{x} \leftarrow \mathrm{x'}$
        e. If x is feasible and $\mathrm{f(x)} < \mathrm{f}(x^*)$ then set $x^* \leftarrow \mathrm{x}$
    B. Update Tk and Lk
        a. $\mathrm{T}_{k+1} \leftarrow \alpha * \mathrm{T}_k$
        b. $\mathrm{L}_{k+1} \leftarrow \gamma * \mathrm{L}_k$
3. Report the incumbent solution

Important Functionalities

In this part, I will explain important parts of the algorithm. You can also find the corresponding function's name of the code next to each subsection name.

First Solution Generation (generateSolution())

This step generates the first solution for the algorithm. Since SA is an improvement heuristic, we should generate an initial solution. I will explain it with the pseudo code:

1. While no feasible solution found, loop 1000 times:
    A. Create a list of jobs and take a random permutation of it then iterate on the list:
        a. Find the agent with minimum resource consumption on this job and the agent whose resource limit not exceeded

i. Assign the job to that user

b. If there is no viable/feasible agent for a job, then break

2. If no feasible solution found, then create a random assignment list.

Neigborhood Structure (findNeighbor(solution, nSize))

I tried two different neighborhood structure. They are very similar.

1. Randomly change assignees of two randomly chosen jobs
2. Randomly change assignees of three randomly chosen jobs

Actually in the code, the nSize parameter of findNeighbor(solution, nSize) function indicates the number of changes of neighbor structure.

Unfitness of a Solution (calculateUnfit(solution))

My simulated annealing algorithm can also accept infeasible solutions. However, to punish infeasible solutions, I used an unfitness score which I took from the GA on GAP paper. In short, the unfitness score calculated like this:

$$u_k = \sum_{i \in I} \max\left[0, \left(\sum_{j \in J, s_{kj}=i} r_{ij}\right) - b_i\right]$$

Initial Temperature (setInitialTemp(acceptProb, nSize))

I used the method we learnt in the class. I calculated the first temperature from the initial acceptance probability.

First, I calculated 10000 delta value which are difference between objective values of two neighbor solution. Then took the average of delta values. Then:

$$T_1 = \frac{|\bar{\Delta}|}{\ln \frac{1}{P_1}}$$

Stopping Condition (stopCond(allSolutions, L))

- Do not stop while $L < 10000$
- If $L > 10000$ and $L < 50000$then stop if the change in objective function of 10 last solution is less than 0.1
- If $L > 50000$ and $L < 150000$ then stop if the change in objective function of 5 last solution is less than 0.1
- If $L > 150000$ stop

Hyperparameter Optimization

After coding a working and partially efficient simulated annealing algorithm, the process turns into a hyperparameter optimization problem. Because in simulated annealing, there are lots of parameters you need to decide on and all of them can affect both the quality of output and time spent. So, after trying lots of combinations and taking notes on them, I actually got very confused and decided to make a automated system which will run the code several times with different parameters.

I determined some values for each parameter and run the code 5 times for each adjustment for each dataset. After each run I took objective values, %Gap values, number of cycles and run times for statistics.

We ran the hyperparameter optimization on dataset 2. It was going to take more than 8 hours to complete, so to avoid this situation, when we encounter a configuration which gives worse than 2% gap in any of its run then we discard this configuration. With this method, it took nearly 3 hours to complete.

After all calculations and analysis, I chose this two parameters configuration to use in my algorithm:

- nSize: 2 - Acceptance: 0.75 - L: 100 - alpha: 0.95 - gamma: 1.1
    - Expected Values:
        - Objective Value: 648.80
        - %Gap: 0.43
        - No. of Cycles: 66.40
        - CPU Time(s): 20.00

- nSize: 2 - Acceptance: 0.75 - L: 1000 - alpha: 0.9 – gamma: 1.15
    - Expected Values:
        - Objective Value: 649.20
        - %Gap: 0.49
        - No. of Cycles: 33.60
        - CPU Time(s): 33.66

In the code, I used both parameter configurations. First, the code runs twice with the first parameter configuration. Then twice with the second one. In the last step, code reports the best solution among those 4 runs.

Results

Note: Reported "No. of iterations" and "CPU Time(s)" are total of my 4 runs from 2 different parameter configurations that I have mentioned.

	Instance: GAP 5-25				
Run No.	Solution (jobs assigned to each agent)	Obj. Value	% Gap	No. of cycles	CPU Time (s)
1	1: [3, 4, 12, 14, 16, 20] 2: [5, 6, 13, 17, 23, 25] 3: [1, 7, 19, 21, 22] 4: [2, 8, 10, 11] 5: [9, 15, 18, 24]	440	0.46	167	32.04
2	1: [4, 11, 16, 18, 20] 2: [3, 6, 13, 14, 17, 23] 3: [1, 7, 12, 19, 22] 4: [2, 8, 10, 25] 5: [5, 9, 15, 21, 24]	438	0.00	170	41.16
3	1: [4, 11, 16, 18, 20] 2: [3, 6, 13, 14, 17, 23] 3: [1, 7, 12, 19, 22] 4: [2, 8, 10, 25] 5: [5, 9, 15, 21, 24]	438	0.00	161	29.44
4	1: [4, 11, 12, 14, 16, 20] 2: [2, 6, 13, 17, 23, 25] 3: [1, 3, 15, 22, 24] 4: [7, 8, 10, 21] 5: [5, 9, 18, 19]	438	0.00	176	36.32
5	1: [4, 11, 16, 18, 20] 2: [3, 6, 13, 14, 17, 23] 3: [1, 7, 12, 19, 22] 4: [2, 8, 10, 25] 5: [5, 9, 15, 21, 24]	438	0.00	158	28.44

Best Run					
5	1: [4, 11, 16, 18, 20] 2: [3, 6, 13, 14, 17, 23] 3: [1, 7, 12, 19, 22] 4: [2, 8, 10, 25] 5: [5, 9, 15, 21, 24]	438	0.00	166.4	133.92

	Instance: GAP 8-40				
Run No.	Solution (jobs assigned to each agent)	Obj. Value	% Gap	No. of cycles	CPU Time (s)
1	1: [3, 4, 5, 28, 34, 37] 2: [9, 27, 30, 31] 3: [7, 20, 24, 26] 4: [1, 8, 14, 18, 22] 5: [2, 12, 33, 39] 6: [13, 19, 23, 25, 29] 7: [10, 16, 21, 32, 35, 38, 40] 8: [6, 11, 15, 17, 36]	647	0.15	170	80.92
2	1: [3, 4, 14, 21, 28, 37] 2: [9, 27, 30, 31] 3: [7, 20, 24, 26] 4: [1, 5, 8, 18, 22, 25] 5: [2, 12, 33, 39] 6: [13, 19, 23, 29] 7: [10, 16, 32, 34, 35, 38, 40] 8: [6, 11, 15, 17, 36]	647	0.15	197	95.4
3	1: [3, 4, 21, 28, 34, 37] 2: [19, 27, 30, 31] 3: [7, 20, 24, 26] 4: [1, 5, 8, 18, 22, 25] 5: [2, 32, 39] 6: [6, 13, 23, 29, 33] 7: [9, 10, 11, 16, 35, 38, 40] 8: [12, 14, 15, 17, 36]	651	0.77	198	105.68

4	1: [4, 28, 33, 37, 38] 2: [9, 20, 25, 27, 30] 3: [3, 6, 7, 24, 26] 4: [1, 5, 8, 13, 18, 22] 5: [2, 21, 39] 6: [19, 23, 29, 31] 7: [10, 11, 16, 32, 34, 35, 40] 8: [12, 14, 15, 17, 36]	650	0.62	185	79.56
5	1: [3, 4, 16, 21, 28, 37] 2: [9, 27, 30, 31] 3: [7, 20, 24, 26] 4: [1, 5, 8, 18, 22, 25] 5: [2, 19, 39] 6: [6, 13, 23, 29, 33] 7: [10, 11, 32, 34, 35, 38, 40] 8: [12, 14, 15, 17, 36]	650	0.62	196	98.08
Best Run					
1	1: [3, 4, 5, 28, 34, 37] 2: [9, 27, 30, 31] 3: [7, 20, 24, 26] 4: [1, 8, 14, 18, 22] 5: [2, 12, 33, 39] 6: [13, 19, 23, 25, 29] 7: [10, 16, 21, 32, 35, 38, 40] 8: [6, 11, 15, 17, 36]	647	0.15	189.2	459.64

	Instance: GAP 10-50				
Run No.	Solution (jobs assigned to each agent)	Obj. Value	% Gap	No. of cycles	CPU Time (s)
1	1: [24, 25, 29, 34, 39] 2: [1, 17, 19, 20, 27, 28] 3: [9, 13, 21, 36, 37, 50] 4: [2, 3, 38, 46, 48] 5: [4, 11, 12, 14, 15, 26, 31] 6: [5, 7, 32, 40, 49] 7: [23, 43, 45, 47] 8: [6, 22, 30] 9: [8, 10, 35, 41] 10: [16, 18, 33, 42, 44]	575	0.35	216	198.63
2	1: [24, 25, 29, 41] 2: [1, 13, 19, 20, 26, 27, 28] 3: [9, 21, 23, 36, 37, 50] 4: [2, 3, 38, 46, 48] 5: [4, 11, 14, 31, 34] 6: [5, 7, 32, 43, 49] 7: [17, 30, 45, 47] 8: [6, 10, 22, 39] 9: [8, 12, 15, 35, 40] 10: [16, 18, 33, 42, 44]	579	1.05	200	127.43
3	1: [14, 24, 29, 39] 2: [1, 17, 20, 25, 27, 28] 3: [9, 13, 21, 36, 37, 50] 4: [2, 3, 8, 46, 48] 5: [4, 11, 26, 31, 38, 49] 6: [5, 7, 19, 32, 40, 43] 7: [15, 23, 34, 45, 47] 8: [6, 22, 30] 9: [10, 12, 35, 41] 10: [16, 18, 33, 42, 44]	577	0.70	204	146.37

4	1: [9, 24, 25, 29, 39] 2: [1, 17, 19, 20, 27, 28] 3: [13, 21, 23, 36, 37, 50] 4: [2, 3, 8, 46, 48] 5: [4, 11, 14, 15, 26, 31, 38] 6: [5, 7, 32, 43, 49] 7: [30, 34, 47] 8: [6, 10, 22, 44] 9: [12, 35, 40, 45] 10: [16, 18, 33, 41, 42]	576	0.52	198	117.03
5	1: [24, 29, 39, 41] 2: [1, 19, 20, 25, 27, 28] 3: [9, 13, 17, 21, 23, 36, 50] 4: [2, 3, 8, 46, 48] 5: [4, 11, 14, 15, 26, 31, 38] 6: [5, 7, 32, 43, 49] 7: [30, 34, 45, 47] 8: [6, 22, 37] 9: [10, 12, 35, 40] 10: [16, 18, 33, 42, 44]	577	0.70	206	153.96
Best Run					
1	1: [24, 25, 29, 34, 39] 2: [1, 17, 19, 20, 27, 28] 3: [9, 13, 21, 36, 37, 50] 4: [2, 3, 38, 46, 48] 5: [4, 11, 12, 14, 15, 26, 31] 6: [5, 7, 32, 40, 49] 7: [23, 43, 45, 47] 8: [6, 22, 30] 9: [8, 10, 35, 41] 10: [16, 18, 33, 42, 44]	575	0.35	204.8	743.42

	Instance: GAP 5-100				
Run No.	Solution (jobs assigned to each agent)	Obj. Value	% Gap	No. of cycles	CPU Time (s)
1	1: [6, 13, 17, 18, 21, 23, 29, 41, 47, 49, 51, 68, 73, 79, 81, 96] 2: [4, 26, 27, 34, 36, 46, 50, 60, 65, 69, 72, 76, 84, 88, 93, 94, 98] 3: [10, 14, 16, 25, 28, 30, 32, 33, 35, 37, 55, 59, 62, 67, 70, 71, 75, 78, 85, 90, 92, 100] 4: [1, 3, 5, 7, 9, 12, 19, 22, 24, 31, 42, 48, 52, 53, 56, 57, 58, 61, 63, 80, 82, 83, 87, 95, 99] 5: [2, 8, 11, 15, 20, 38, 39, 40, 43, 44, 45, 54, 64, 66, 74, 77, 86, 89, 91, 97]	1698	0.00	215	378.47
2	1: [6, 13, 17, 18, 21, 23, 29, 41, 47, 49, 51, 54, 68, 73, 79, 81, 96, 99] 2: [4, 26, 27, 34, 36, 46, 50, 60, 65, 69, 72, 76, 84, 88, 93, 94, 98] 3: [10, 14, 16, 25, 28, 30, 32, 33, 35, 37, 55, 59, 62, 67, 70, 71, 75, 78, 85, 90, 92, 100] 4: [1, 5, 7, 9, 12, 19, 22, 24, 31, 42, 48, 52, 53, 56, 57, 58, 61, 63, 80, 82, 83, 87, 95] 5: [2, 3, 8, 11, 15, 20, 38, 39, 40, 43, 44, 45, 64, 66, 74, 77, 86, 89, 91, 97]	1700	0.12	212	363.02
3	1: [6, 13, 17, 18, 21, 23, 29, 41, 47, 49, 51, 68, 73, 79, 81, 96] 2: [4, 26, 27, 34, 36, 46, 50, 60, 65, 69, 72, 75, 76, 84, 88, 93, 94, 98] 3: [10, 14, 16, 25, 28, 30, 32, 33, 35, 37, 55, 59, 62, 67, 70, 71, 78, 85, 90, 92, 100] 4: [1, 3, 5, 7, 9, 12, 19, 22, 24, 31, 42, 48, 52, 53, 56, 57, 58, 61, 63, 80, 82, 83, 87, 95, 99] 5: [2, 8, 11, 15, 20, 38, 39, 40, 43, 44, 45, 54, 64, 66, 74, 77, 86, 89, 91, 97]	1698	0.00	210	349.86
4	1: [6, 13, 17, 18, 21, 23, 26, 29, 41, 47, 49, 51, 68, 73, 79, 81, 96]	1699	0.06	207	331.31

	2: [4, 27, 34, 36, 46, 50, 60, 65, 69, 72, 76, 84, 88, 93, 94, 98] 3: [10, 14, 16, 25, 28, 30, 32, 33, 35, 37, 55, 59, 62, 67, 70, 71, 75, 78, 85, 90, 92, 100] 4: [1, 3, 5, 7, 9, 12, 19, 22, 24, 31, 42, 48, 52, 53, 56, 57, 58, 61, 63, 80, 82, 83, 87, 95, 99] 5: [2, 8, 11, 15, 20, 38, 39, 40, 43, 44, 45, 54, 64, 66, 74, 77, 86, 89, 91, 97]				
5	1: [6, 13, 17, 18, 21, 23, 29, 41, 47, 49, 51, 68, 73, 79, 81, 96] 2: [4, 26, 27, 34, 36, 46, 50, 60, 65, 69, 72, 76, 84, 88, 93, 94, 98] 3: [10, 14, 16, 25, 28, 30, 32, 33, 35, 37, 55, 59, 62, 67, 70, 71, 75, 78, 85, 90, 92, 100] 4: [1, 3, 5, 7, 9, 12, 19, 22, 24, 31, 42, 48, 52, 53, 56, 57, 58, 61, 63, 80, 82, 83, 87, 95, 99] 5: [2, 8, 11, 15, 20, 38, 39, 40, 43, 44, 45, 54, 64, 66, 74, 77, 86, 89, 91, 97]	1698	0.00	223	482.12
Best Run					
3	1: [6, 13, 17, 18, 21, 23, 29, 41, 47, 49, 51, 68, 73, 79, 81, 96] 2: [4, 26, 27, 34, 36, 46, 50, 60, 65, 69, 72, 75, 76, 84, 88, 93, 94, 98] 3: [10, 14, 16, 25, 28, 30, 32, 33, 35, 37, 55, 59, 62, 67, 70, 71, 78, 85, 90, 92, 100] 4: [1, 3, 5, 7, 9, 12, 19, 22, 24, 31, 42, 48, 52, 53, 56, 57, 58, 61, 63, 80, 82, 83, 87, 95, 99] 5: [2, 8, 11, 15, 20, 38, 39, 40, 43, 44, 45, 54, 64, 66, 74, 77, 86, 89, 91, 97]	1698	0.00	213.4	1904.78

Code

Note: As I mentioned at the beginning, more detailed work of mine can be found in my GitHub page.

Since my code is too long, I will only put the parts that are directly related with the algorithm.

Reading Data

```
# Dataset number. 1, 2, 3 or 4
dataset = "4"

if dataset == "1":
   optimalObjective = 438
elif dataset == "2":
   optimalObjective = 646
elif dataset == "3":
   optimalObjective = 573
else:
   optimalObjective = 1698

filename = "gap-data-4instances" + dataset + ".txt"
f = open(filename, 'r')
l = f.readline().split()

# number of agents
m = int(l[0])

# number of jobs
n = int(l[1])

# cost of assigning job j to agent i
c = []

# resource required by agent i to perform job j
r = []

# resource capacity of agent i.
b = []


for i in range(m):
   temp = []
   for l in f.readline().split():
      temp.append(int(l))
   c.append(temp)

for i in range(m):
   temp = []
   for l in f.readline().split():
      temp.append(int(l))
```

```
    r.append(temp)

for l in f.readline().split():
    b.append(int(l))

f.close()
```

Helper Funtions

```
def isFeasible(solution):
    caps = [0] * m
    for i in range(len(solution)):
        caps[solution[i]] = caps[solution[i]] + r[solution[i]][i]

    for i in range(len(caps)):
        if b[i] < caps[i]:
            return False

    return True

def calculateCost(solution):
    cost = 0
    for i in range(len(solution)):
        cost = cost + c[solution[i]][i]

    return cost

def calculateUnfit(solution):
    unfitness = 0
    caps = [0] * m

    for i in range(len(solution)):
        caps[solution[i]] = caps[solution[i]] + r[solution[i]][i]

    for i in range(len(caps)):
        unfitness = unfitness + max(0, caps[i] - b[i])

    return unfitness

def generateSolution():
    solution = [0] * n

    for count in range(1000):
        isFound = True
        caps = [0] * m
        for job in np.random.permutation(n):
            minWork = sum(b)
            minWorker = -1
            for i in range(m):
                if caps[i]+r[i][job] <= b[i]:
```

```
                if r[i][job] < minWork:
                    minWork = r[i][job]
                    minWorker = i
            if minWorker == -1:
                isFound = False
                break
            else:
                solution[job] = minWorker
                caps[minWorker] = caps[minWorker] + minWork

        if isFound == True:
            break

    if not isFound:
        for i in range(len(solution)):
            solution[i] = random.randint(0,m-1)

    return solution


def findNeighbor(solution, nSize):
    newSolution = list(solution)

    for i in range(nSize):
        k = random.randint(0,m*n-1)
        agent = int(k / n)
        job = k % n
        newSolution[job] = agent

    return newSolution

def setInitialTemp(acceptProb, nSize):
    deltaSum = 0
    for i in range(10000):
        solution = generateSolution()
        newSolution = findNeighbor(solution, nSize)
        deltaSum = deltaSum + abs(calculateCost(newSolution) - calculateCost(solution))

    return (deltaSum/10000) / math.log(1/acceptProb)


def stopCond(allSolutions, L):
    s = len(allSolutions)
    if L < 10000:
        return False
    if L > 150000:
        return True
    if L > 50000 and ((allSolutions[s-5] - allSolutions[s-1]) / allSolutions[s-5]) * 100 < 0.1:
        return True
    if ((allSolutions[s-10] - allSolutions[s-1]) / allSolutions[s-10]) * 100 > 0.1:
        return False
```

```
    return True


solution = generateSolution()
print(solution)
print(isFeasible(solution))

newSolution = findNeighbor(solution, 2)
print(newSolution)

print(calculateCost(solution))
print(calculateCost(newSolution))


#print(math.exp(-0.1/0.3))
```

Main Simulated Annealing Code

```
# nReheat: How many times the algorithm will run with this parameters from the begining.
# bestSolutions: Holds all the incumbent solutions for each run.
# totalIterations: Number of total iterations for each run.

def SA_GAP(initialTemp, initialL, alpha, gama, nSize, nReheat, bestSolutions, totalIterations):
    Temp = initialTemp
    L = initialL
    allSolutions = []
    solution = generateSolution()

    bestSolution = []
    if isFeasible(solution):
        bestSolution = list(solution)

    reheated = True

    for rh in range(nReheat):
        t11 = time.clock()
        while not stopCond(allSolutions, L):
            if not reheated:
                if bestSolution == []:
                    solution = generateSolution()
                else:
                    solution = list(bestSolution)
            else:
                reheated = False

            for i in range(L):
                newSolution = findNeighbor(solution, nSize)
                u = np.random.uniform(0,1)
```

```
            deltaCost = calculateCost(newSolution) - calculateCost(solution) + calculateUnfit(newSolution)

            if deltaCost < 0 or u < math.exp(-deltaCost/(Temp)):
               solution = list(newSolution)
            if isFeasible(newSolution) and (bestSolution == [] or calculateCost(newSolution) < calculateCost(bestSolution)):
               bestSolution = list(newSolution)


         Temp = Temp * alpha
         L = int(L * gama)
         allSolutions.append(calculateCost(bestSolution))
         #print(str(len(allSolutions)) + ' - ' + str(math.exp(-deltaCost/(Temp))) + ' - ' + str(L) + ' - ' + str(calculateCost(bestSolution)) + ' - ' + str(calculateCost(solution)))

      t22 = time.clock()
      bestSolutions.append(bestSolution)
      totalIterations.append(len(allSolutions))

      bSolVal = calculateCost(bestSolution)
      G = 100 * (bSolVal-optimalObjective) / optimalObjective
      nCycle = len(allSolutions)
      timePassed = t22-t11
      print("Objective Value: " + str(bSolVal) + " - %Gap: " + "%.2f" %G + " - No. of Cycles: " + "%.2f" %nCycle +" - CPU Time(s): " + "%.2f" %timePassed)

      allSolutions = []
      Temp = initialTemp
      L = initialL
      solution = generateSolution()
      bestSolution = []
      if isFeasible(solution):
         bestSolution = list(solution)
      reheated = True
```

Main Solver

```
params1 = [2, 0.75, 100, 0.95, 1.1]
params2 = [2, 0.75, 1000, 0.9, 1.15]

# Start Timer
t1 = time.clock()

# Solve twice with first parameters

# Neighborhood search replace size
nSize = params1[0]

# How many times to reheat
```

```
nReheat = 2

initialTemp = setInitialTemp(params1[1], nSize)
initialL = params1[2]
alpha = params1[3]
gama = params1[4]
reheated = True
bestSolutions = []
totalIterations = []

SA_GAP(initialTemp, initialL, alpha, gama, nSize, nReheat, bestSolutions, totalIterations)

# Solve twice with second parameters


nSize = params2[0]
nReheat = 2
initialTemp = setInitialTemp(params2[1], nSize)
initialL = params2[2]
alpha = params2[3]
gama = params2[4]
reheated = True
SA_GAP(initialTemp, initialL, alpha, gama, nSize, nReheat, bestSolutions, totalIterations)


# Stop Timer
t2 = time.clock()
```

Results

```
bSol = bestSolutions[0]
bSolVal = calculateCost(bestSolutions[0])
for sol in bestSolutions:
    if bSolVal > calculateCost(sol):
        bSolVal = calculateCost(sol)
        bSol = sol


solDict = {}
print("Solution:")
for i in range(len(bSol)):
    if bSol[i]+1 not in solDict:
        solDict[bSol[i]+1] = [i+1]
    else:
        solDict[bSol[i]+1].append(i+1)

for i in range(m):
    print(str(i+1) + ": " + str(solDict[i+1]))
print()
```

```
print("Objective Value:")
print(bSolVal)
print()

print("%Gap:")
G = 100 * (bSolVal-optimalObjective) / optimalObjective
print("%.2f" %G)
print()

print("No. of Cycles: (Total of " + str(len(bestSolutions)) + " runs)")
nCycle = sum(totalIterations)
print("%.2f" %nCycle)
print()

print("CPU Time (s): (Total of " + str(len(bestSolutions)) + " runs)")
timePassed = (t2-t1)
print("%.2f" %timePassed)
```