Airplane allocation - A stochastic problem

Benjamin THOMINET
Francisco RESTIVO
Valentin REVERSAT

Computer Science department
Akademia Górniczo-Hutnicza im. Stanisława Staszica
Poland

Fall semester - 2020 / 2021

Contents

1 Problem definition

In a world where everything move fast, airline companies have to adapt their airplane allocation in order to fit as best as they can the passengers demand. To reach that goal, one of the possible method is using a stochastic approach. In fact, many variables must be digest to find one the best possible airplane distribution. A stochastic approach is well fitted for our case because it's allow to search one possible solution among a large set of possible one by trying a lot of airplanes distribution solutions.

In our problem, an airline company wishes to allocate airplanes of various types among its route to satisfy an uncertain passenger demand, in such a way as to minimize operating costs plus the lost revenue from passengers turned away. This problem use *4* different types of aircraft and *5* routes.

This problem is taken from the book *Numerical Technics for Stochastic Optimizations* by Yu. Ermoliev & R. J-B Wets (p.565, A.J. King). Link : http://pure.iiasa.ac.at/id/eprint/3065/1/XB-88-403.pdf#page=565

In the following report, we will introduce us our work and our implementation of this airplane allocation problem

1.1 The modeling

In order to modelize the problem, we use a matrix A containing the number of aircraft use for a given route and a given type :

$$A = (x_{i,j})_{1\leq i\leq 5, 1\leq j\leq 4} = \begin{pmatrix} x_{1,1} & \dots & \dots & x_{1,4} \\ x_{2,1} & \dots & \dots & x_{2,4} \\ x_{3,1} & \dots & \dots & x_{3,4} \\ x_{4,1} & \dots & \dots & x_{4,4} \\ x_{5,1} & \dots & \dots & x_{5,4} \end{pmatrix} \in M_{5,4}(\mathbb{N}) \text{ where } x \in \mathbb{N}$$

Where $x_{i,j}$ represents the number of airplane for the type j on the route i.

Example $x_{3,4} = 23$ means that we have 23 air-crafts of type 4 on the route number 3

1.2 The cost function

In order to find the best solution, we need to minimize the following cost function :

$$f = \sum_{i=1}^{5} \sum_{j=1}^{4} (Cost_{i,j} * x_{i,j}) + \sum_{k=1}^{5} RevenueLost_k * PassengersTA_k$$

Where :

- $Cost_{i,j}$ characterize the operational cost of an aircraft by the route. For our implementation, we chose to use this matrix $C \in M_{5,4}(\mathbb{N})$ of operational costs :

$$C = \begin{pmatrix} 12 & 20 & 20 & 19 \\ 2 & 34 & 10 & 20 \\ 43 & 63 & 40 & 12 \\ 32 & 10 & 6 & 34 \\ 20 & 30 & 10 & 87 \end{pmatrix} \in M_{5,4}(\mathbb{N})$$

Example : $Cost_{1,4} = 19$ and $Cost_{5,2} = 30$

- $RevenueLost_k$ with $k \in [1, 5]$ denote the revenue lost per passenger turned away on the *kth* route. For our implementation, we chose to use this matrix R :

$$R = \begin{pmatrix} 13 & 20 & 7 & 7 & 15 \end{pmatrix} \in M_{1,5}(\mathbb{N})$$

- $PassengersTA_k \in \mathbb{N} = D_{1,k} - \sum_{j=1}^{4} (A_{k,j} * B_{k,j})$ with $k \in [1, 5]$ denote the number of passenger who turned away from the *kth* route. This is obtained by subtracting the route demand by the real number of passenger knew by multiplying the number of aircraft per route by their capacity. The demand is denote by this matrix D:

$$D = \begin{pmatrix} 800 & 900 & 700 & 650 & 380 \end{pmatrix} \in M_{1,5}(\mathbb{N})$$

And the total capacity of each type of aircraft is denoted by the following matrix B :

$$B = \begin{pmatrix} 16 & 10 & 30 & 23 \end{pmatrix} \in M_{1,4}(\mathbb{N})$$

Our goal here will to lower this function in order to fit the demand with the lowest possible cost. To achieve this goal, we choose two approach. The first one is the resolution by a population based algorithm, the genetic one.

2 Implementation

2.1 Goal

The main goal of our project is to determine which of the both stochastic approach (**simulated annealing or genetic**) is the best fitted for this kind of problem. In order to determine that, we established a test protocol by measuring the number of cost function calls of each algorithm with a set a predefined parameters. Both algorithm are quite different and don't take the same input parameters. So we choose the best suited for each.

In the following chapters, we will present you how work the **simulated annealing** and the **genetic**, the problems we encountered during the implementations and finally the tests with the results.

First of all, we will introduce in depth those two algorithms, then the research we did to determine the best arguments for our test runs. Finally, we will compare the both algorithms and conclude

2.2 First proposed solution - Population based algorithm

The main idea behind this type of algorithm is to generate multiple generation and for each one pick the two best members, cross them, mutate the resulted child and generate a new population :

2.2.1 Key features

- **Population size** : We chose to use a single population. The size is kept constant across all generations
- **Mixing operations** : To generate our children, we cross the columns between parental matrices. When our children are generated, we select one them with the uniform probability distribution. Finally, the picked child is mutated by re-generating the entire given column. (more details one below section)
- **Succession scheme** : We use the μ,μ succession scheme. This means that with μ parents we get μ children in our new generation
- **Stopping condition** : As we are looking for the one of the best solution, our stopping condition is determined by the length of our generation.

2.2.2 The algorithm

1. Generate a base population $P = \{p_0, ..., p_n\}$ of size n. The first iteration will generate a random one with one constraint, we have a limited number $T_{1,k}$ of each type k of aircraft :
$$T = \begin{pmatrix} 10 & 19 & 25 & 16 \end{pmatrix} \in M_{1,5}(\mathbb{N})$$
2. Compute the total cost of the generation with the our formula previously define.
3. Compute the probability for each individuals to picked them regarding their cost :
$$P \in [0,1] = \frac{1/Cost}{\sum 1/Cost}$$
4. Pick the to best members $\{p_1, p_2\}$ of the population according to their probability. They will be our two *parents*.
5. Cross both parents according to the continuous uniform distribution $unif(0,1)$ and a threshold $C_T \in [0,1]$. For each column of the parents, if $v \leq C_T$ we swap the column. At the end we have a new child. This is done two times to get two children
6. Pick one of the two children according to the continuous uniform distribution $unif(0,1)$, if $unif(0,1) \geq 0.5$ we pick the first child, then we pick the second.
7. Mutate the picked child according to the continuous uniform distribution $unif(0,1)$ and a threshold $M_T \in [0,1]$. For each column of the child, if $v \leq M_T$ we swipe two random value in the same row.
8. We repeat *4, 5 and 6* until the size of the new generation is equal to the previous one (n).

2.2.3 Set of research

Let's denote $(\Omega, \mathcal{F}, \mathcal{P})$ our probabilistic space :

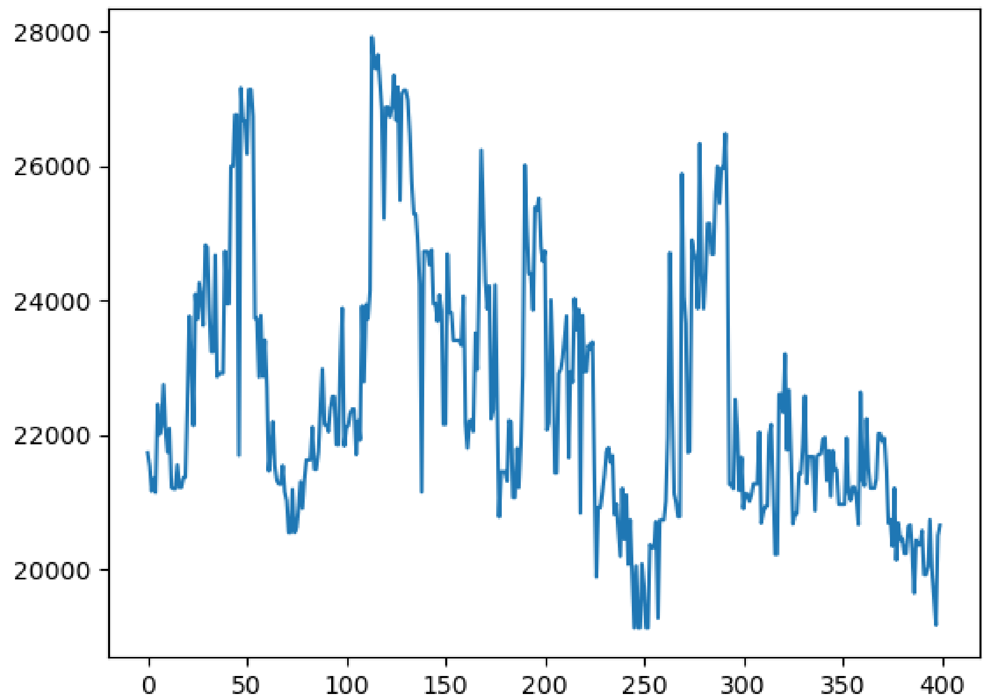
$$\Omega = \{M \in M_{5,4}(\mathbb{N}); \textstyle\sum_{k=1}^{5} M_{k,n} = T_{1,n}, n \in [1,4]\}$$

$$\mathcal{F} = 2^{\Omega}$$

$$Probability\ on\ M\ is\ the\ particular\ [0,1] \ni Pr(M) = \frac{1/Cost(M)}{\sum 1/Cost(M)}\ (cf.Cost\ function\ paragraph)$$

2.2.4 Encountered problems - The non convergence

The first implementation of the genetic algorithm was based as described in the previous part on probabilities. In fact, in order to choose our parents, we computed the cost of each one and then associate the corresponding probability. At the end, the parents was picked up following their associated probability.



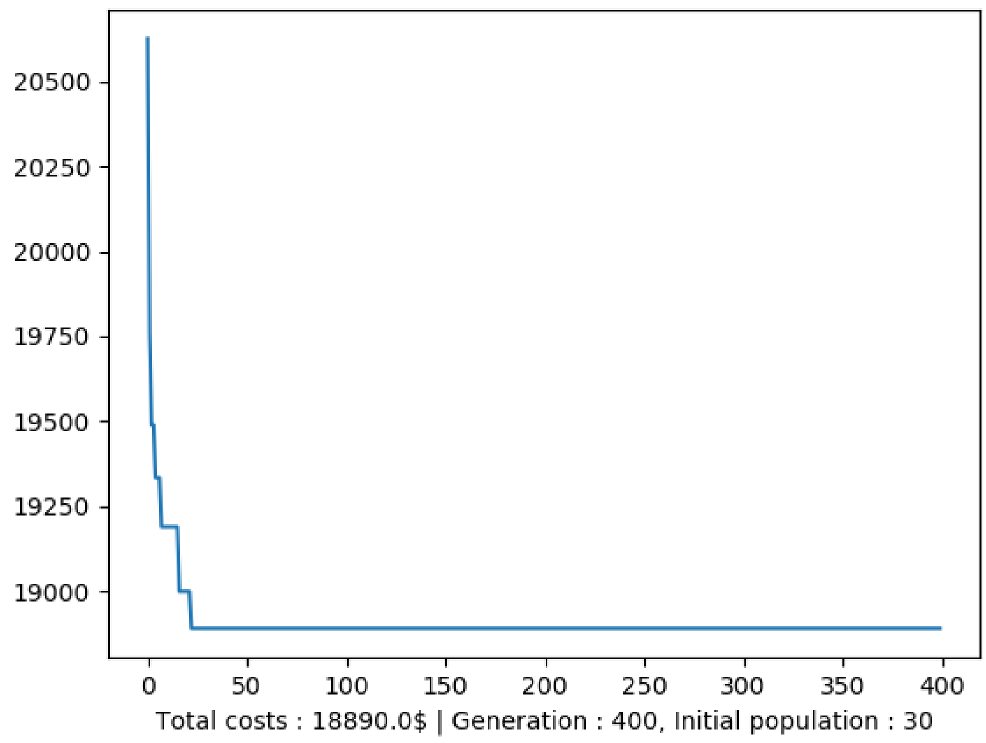
Total costs : 20660.0$ | Generation : 400, Initial population : 30

This graph shows you for every generation, the best cost we can get. In the run, we run *400* generations of *30* individuals.

Unfortunately as you can see, the result is not converging toward a possible solution but it is chaotic. This is because of the way we pick our parents. We try in a first place to If we base this on the probabilities, the results are not good.

In this first run, $C_T = 0.3$ and $M_T = 0.15$ and $min_{cost} = 20660\$$

In order to get around this problem, we decided to change the way of picking the parents responsible to generate the future generation. Instead of picking them according to there probability, we directly pick the two best element of the population. So we change the fourth step of the algorithm by this :



4. Pick the to best members $\{p_1, p_2\}$ of the population according to their cost. The parents with the lowest cost among all the population. They will be our two *parents*: $p_1 = \min_{x \in P} f(x)$ **and** $p_2 = \min_{x \in P - p_1} f(x)$**.**

As you can see we have a convergence toward one of the possible solution and no more chaos.

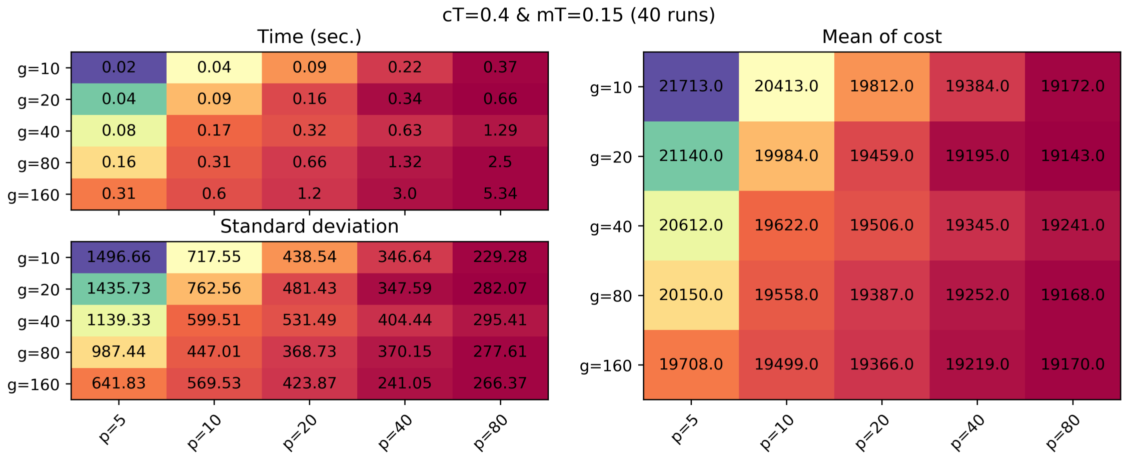
In this second run, $C_T = 0.3$ and $M_T = 0.15$ and $min_{cost} = 18980\$$

2.3 Parameters optimization

In order to pick the appropriates arguments for the genetic stochastic approach, I run multiple times a a range of simulation with different parameters :

2.3.1 Statistic representation

(vertical = number of generation ; horizontal = population size)



cT=0.4 & mT=0.15 (40 runs)

Time (sec.)

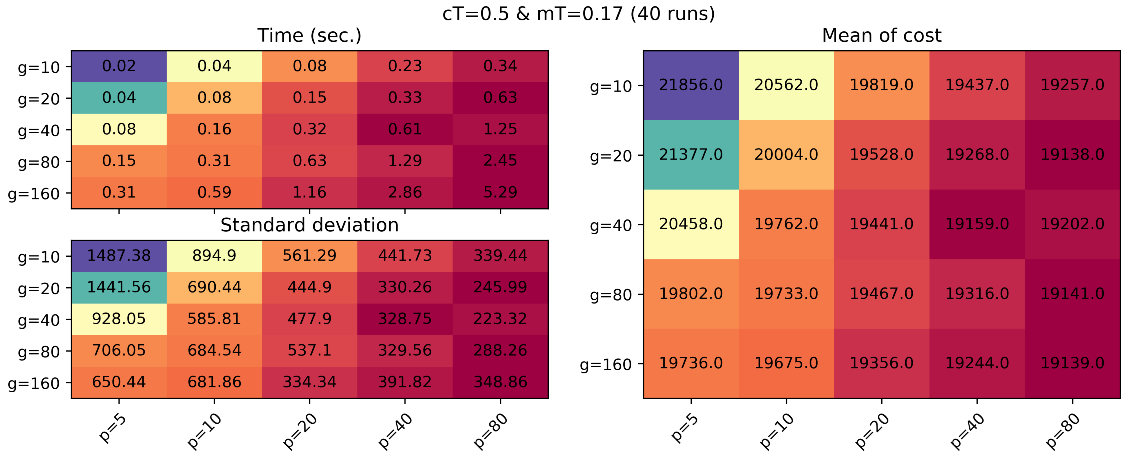
	p=5	p=10	p=20	p=40	p=80
g=10	0.02	0.04	0.09	0.22	0.37
g=20	0.04	0.09	0.16	0.34	0.66
g=40	0.08	0.17	0.32	0.63	1.29
g=80	0.16	0.31	0.66	1.32	2.5
g=160	0.31	0.6	1.2	3.0	5.34

Standard deviation

	p=5	p=10	p=20	p=40	p=80
g=10	1496.66	717.55	438.54	346.64	229.28
g=20	1435.73	762.56	481.43	347.59	282.07
g=40	1139.33	599.51	531.49	404.44	295.41
g=80	987.44	447.01	368.73	370.15	277.61
g=160	641.83	569.53	423.87	241.05	266.37

Mean of cost

	p=5	p=10	p=20	p=40	p=80
g=10	21713.0	20413.0	19812.0	19384.0	19172.0
g=20	21140.0	19984.0	19459.0	19195.0	19143.0
g=40	20612.0	19622.0	19506.0	19345.0	19241.0
g=80	20150.0	19558.0	19387.0	19252.0	19168.0
g=160	19708.0	19499.0	19366.0	19219.0	19170.0



cT=0.5 & mT=0.17 (40 runs)

Time (sec.)

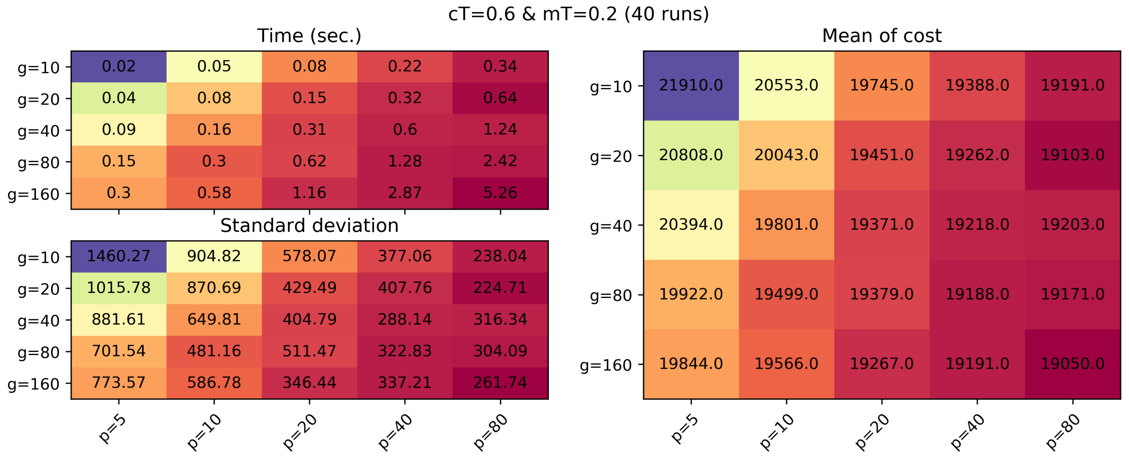
	p=5	p=10	p=20	p=40	p=80
g=10	0.02	0.04	0.08	0.23	0.34
g=20	0.04	0.08	0.15	0.33	0.63
g=40	0.08	0.16	0.32	0.61	1.25
g=80	0.15	0.31	0.63	1.29	2.45
g=160	0.31	0.59	1.16	2.86	5.29

Standard deviation

	p=5	p=10	p=20	p=40	p=80
g=10	1487.38	894.9	561.29	441.73	339.44
g=20	1441.56	690.44	444.9	330.26	245.99
g=40	928.05	585.81	477.9	328.75	223.32
g=80	706.05	684.54	537.1	329.56	288.26
g=160	650.44	681.86	334.34	391.82	348.86

Mean of cost

	p=5	p=10	p=20	p=40	p=80
g=10	21856.0	20562.0	19819.0	19437.0	19257.0
g=20	21377.0	20004.0	19528.0	19268.0	19138.0
g=40	20458.0	19762.0	19441.0	19159.0	19202.0
g=80	19802.0	19733.0	19467.0	19316.0	19141.0
g=160	19736.0	19675.0	19356.0	19244.0	19139.0



cT=0.6 & mT=0.2 (40 runs)

Time (sec.)

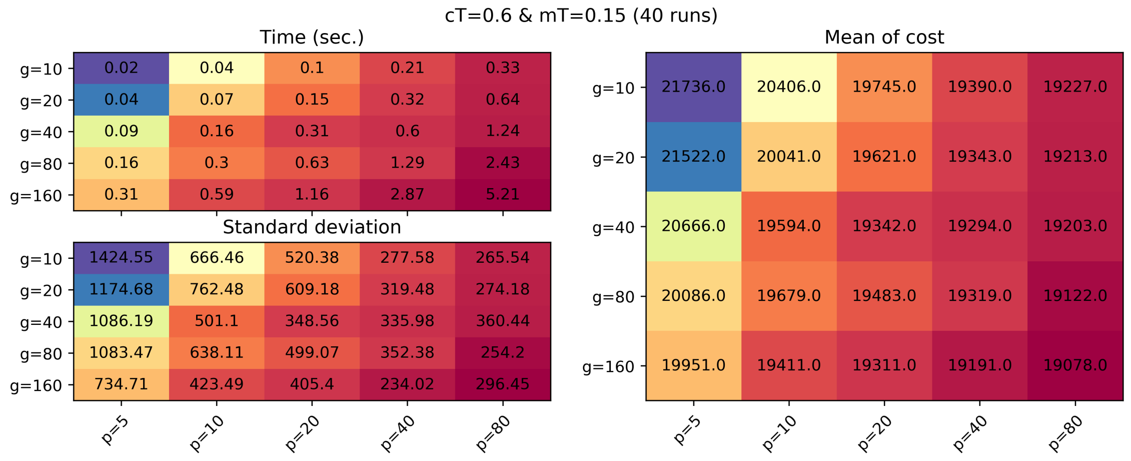
	p=5	p=10	p=20	p=40	p=80
g=10	0.02	0.05	0.08	0.22	0.34
g=20	0.04	0.08	0.15	0.32	0.64
g=40	0.09	0.16	0.31	0.6	1.24
g=80	0.15	0.3	0.62	1.28	2.42
g=160	0.3	0.58	1.16	2.87	5.26

Standard deviation

	p=5	p=10	p=20	p=40	p=80
g=10	1460.27	904.82	578.07	377.06	238.04
g=20	1015.78	870.69	429.49	407.76	224.71
g=40	881.61	649.81	404.79	288.14	316.34
g=80	701.54	481.16	511.47	322.83	304.09
g=160	773.57	586.78	346.44	337.21	261.74

Mean of cost

	p=5	p=10	p=20	p=40	p=80
g=10	21910.0	20553.0	19745.0	19388.0	19191.0
g=20	20808.0	20043.0	19451.0	19262.0	19103.0
g=40	20394.0	19801.0	19371.0	19218.0	19203.0
g=80	19922.0	19499.0	19379.0	19188.0	19171.0
g=160	19844.0	19566.0	19267.0	19191.0	19050.0



Time (sec.)

	p=5	p=10	p=20	p=40	p=80
g=10	0.02	0.04	0.1	0.21	0.33
g=20	0.04	0.07	0.15	0.32	0.64
g=40	0.09	0.16	0.31	0.6	1.24
g=80	0.16	0.3	0.63	1.29	2.43
g=160	0.31	0.59	1.16	2.87	5.21

Standard deviation

	p=5	p=10	p=20	p=40	p=80
g=10	1424.55	666.46	520.38	277.58	265.54
g=20	1174.68	762.48	609.18	319.48	274.18
g=40	1086.19	501.1	348.56	335.98	360.44
g=80	1083.47	638.11	499.07	352.38	254.2
g=160	734.71	423.49	405.4	234.02	296.45

Mean of cost

	p=5	p=10	p=20	p=40	p=80
g=10	21736.0	20406.0	19745.0	19390.0	19227.0
g=20	21522.0	20041.0	19621.0	19343.0	19213.0
g=40	20666.0	19594.0	19342.0	19294.0	19203.0
g=80	20086.0	19679.0	19483.0	19319.0	19122.0
g=160	19951.0	19411.0	19311.0	19191.0	19078.0

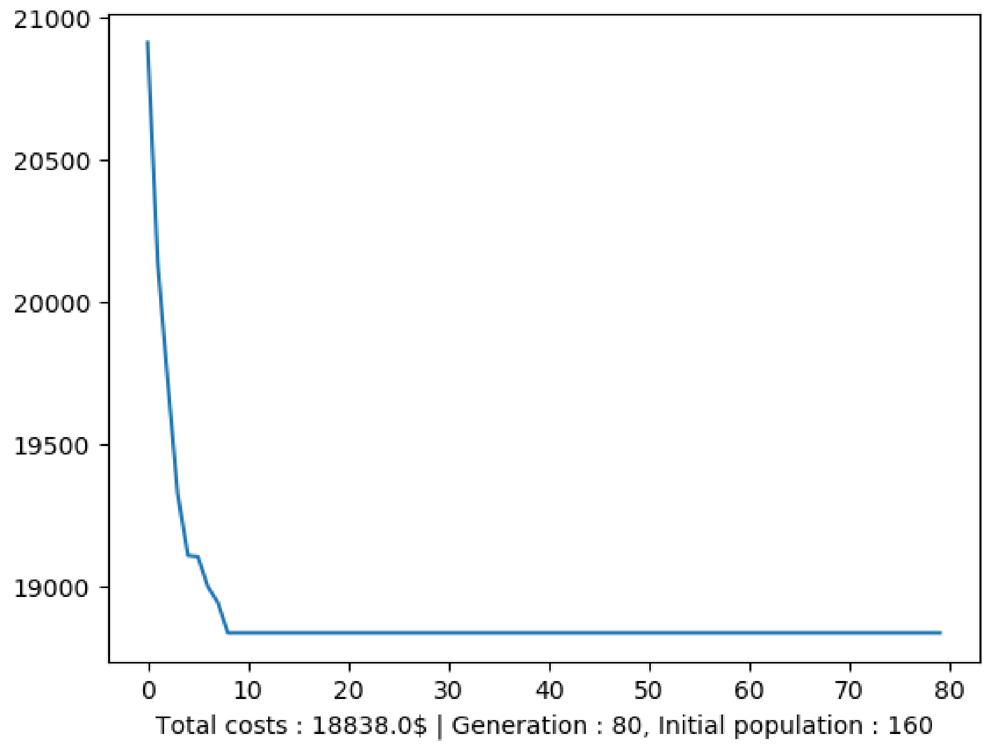
2.3.2 Decision

As you can see, the last runs with $C_T = 0.6$ and $M_T = 0.2$ gave us the best result

2.3.3 Evaluation

When we run the algorithm with the following arguments, we get this graph :

$$n = 160 \; p = 80 \; C_T = 0.6 \; M_T = 0.2$$



As we can see, there is a very quick convergence

2.4 Second proposed solution - Simulated Annealing

The simulated annealing is a good choice when one is trying to find a single best solution to a problem, and this is the case with this aircraft allocation problem.

2.4.1 Key features

- **Temperature** : The main parameters for the simulated annealing algorithm, the temperature allows this algorithm to have a lot of diversity at the beginning to finally converge to a unique solution.
- **Decrease Factor** : At each end of a loop the temperature is multiplied by (1 - Decrease Factor), thus the decrease factor define the speed of convergence of this algorithm.
- **Number of neighbors generated each time** : To implement a successful S.A. algorithm we needed to generate some neighbors from a given solution to create diversity and thus help the algorithm to find the best solution. The number of neighbors generated each time will remain the same : 4.

2.4.2 The algorithm

Pseudo code

```
Algorithm 1: Simulated Annealing
 Input  : Temperature , DecreaseFactor
 Output: CurrentCandidate
1  Initialize randomly FirstCandidate
2  $CurrentCandidate \leftarrow FirstCandidate$
3  while $Temperature < 0.1$ do
4      for step = 0 ; step < 4 ; step++ do
5          Generate 4 Neighbors using CurrentCandidate by only modifying
            column number step
6          Find the BestNeighbor (with minimum cost)
7          $CostDifference \leftarrow BestNeighborCost - CurrentCandidateCost$
8          if $CostDifference < 0$ then
9              $CurrentCandidate \leftarrow BestNeighbor$
10         else if $RandomProbability < exp(-\frac{CostDifference}{Temperature})$ then
11             $CurrentCandidate \leftarrow BestNeighbor$
12     end
13     $Temperature \leftarrow Temperature * (1 - DecreaseFactor)$
14 end
15 Return CurrentCandidate
16
```

(RandomProbability is a number randomly generated between 0 and 1.)

Neighbors generation : Let's explain more precisely how the neighbours are generated. First, as displayed in the pseudo code, there is not only 1 step in the S.A. algorithm before the decreasing of the temperature but 4. We choose this to simplify the neighbors generation and to modify airplane allocation one airplane type at a time, so as there are 4 airplane type it is done 4 times.

There are 3 operations to generate neighbors, and they all keep the same amount of plane as constraint :

- Single Random Permutation
  - Ex : From (2,**1**,3,0,**4**) to (2,**4**,3,0,**1**) (4 and 1 permuted)
- Random +1 / -1
  - ex : from (2,1,3,**0**,**4**) to (2,1,3,**1**,**3**)
- Whole New Column
  - Ex : from (2,1,3,0,4) to (3,0,3,2,2) (whole new allocation)

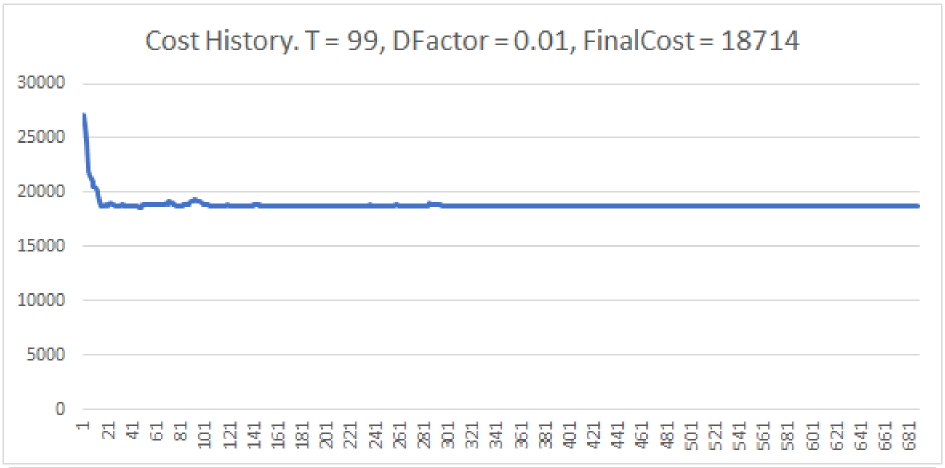
Thanks to this 3 operations the S.A. algorithm is capable of creating enough diversity to converge towards a better and better candidate.

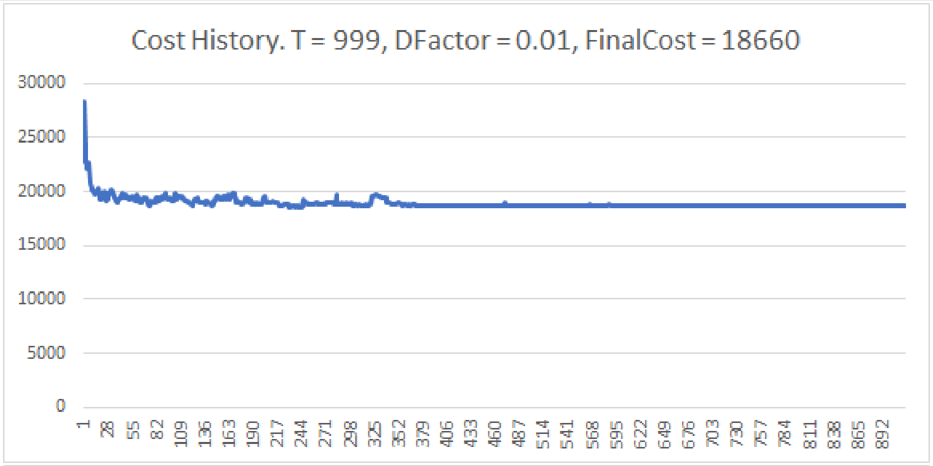
2.5 Parameters Optimization

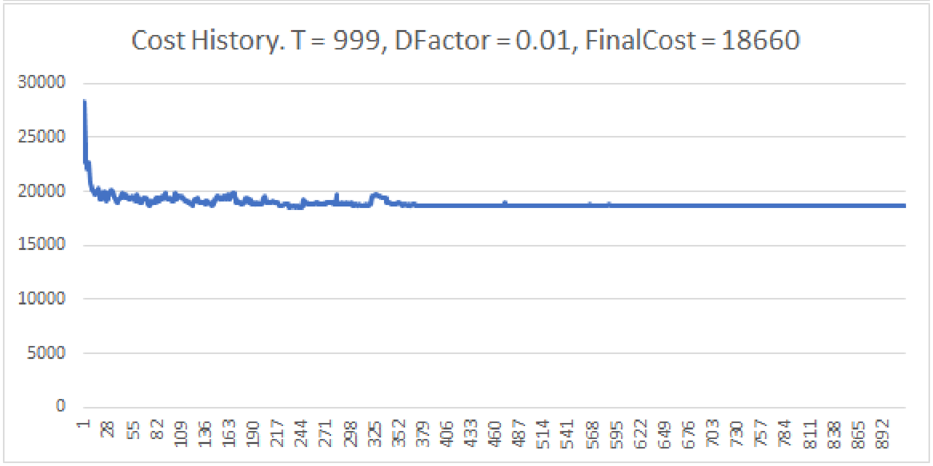
The algorithm work as expected, however we also need to find the good set of parameters for our problem in order to have the best result in the minimum time.

2.5.1 Result

Here is some result with the cost of the current candidate in ordinate and the number of loop in the while loop in abscissa.

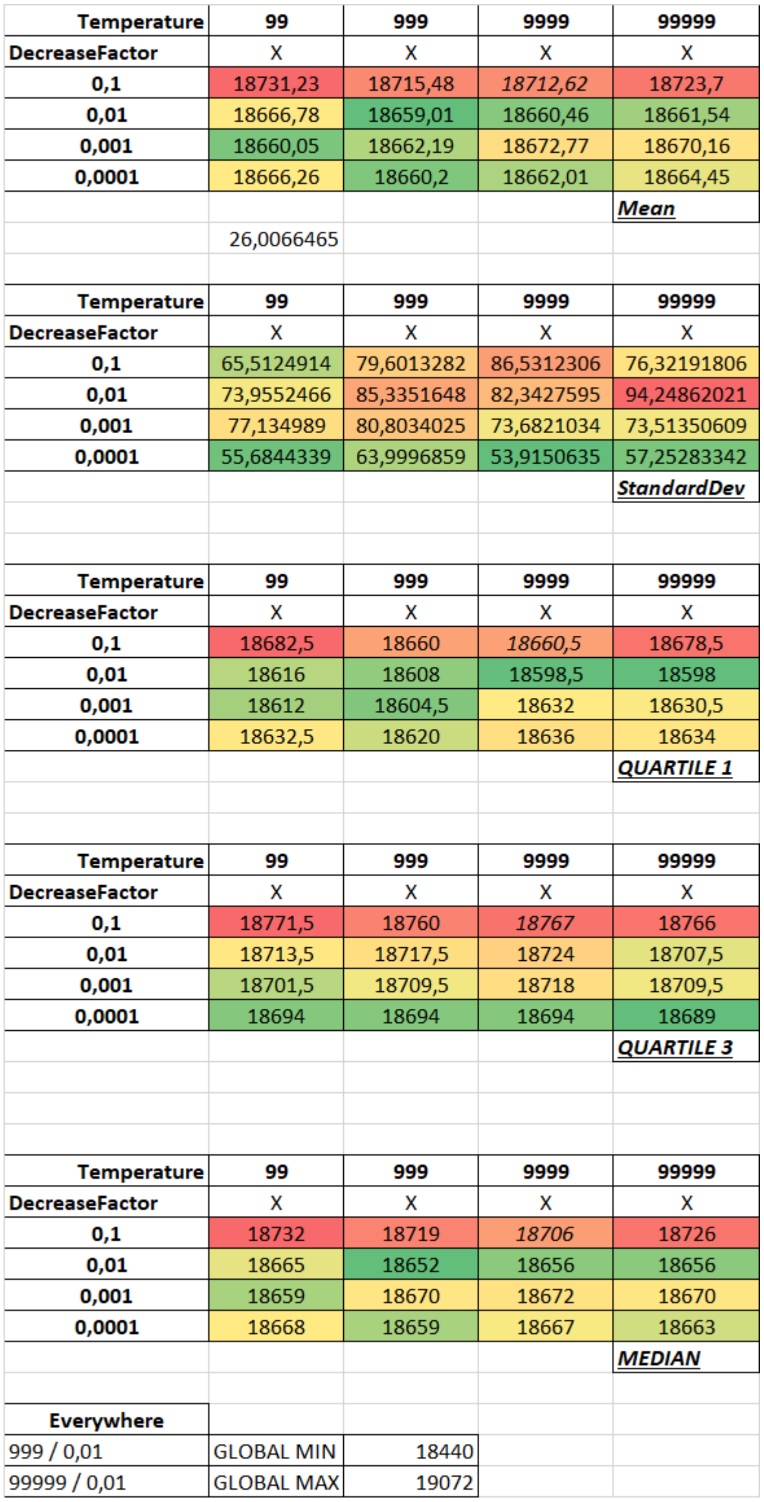






2.5.2 Some statistical datas

In order to understand in a more effective way how the Temperature and the decrease factor influences the result, we realize some statistical computation. There is below some data using 200 output of the algorithm per case (so the SA algorithm ran 3200 times for these data). In green the minimum value and in red the maximum value.



Temperature	99	999	9999	99999
DecreaseFactor	X	X	X	X
0,1	18731,23	18715,48	*18712,62*	18723,7
0,01	18666,78	18659,01	18660,46	18661,54
0,001	18660,05	18662,19	18672,77	18670,16
0,0001	18666,26	18660,2	18662,01	18664,45
				Mean
	26,0066465			

Temperature	99	999	9999	99999
DecreaseFactor	X	X	X	X
0,1	65,5124914	79,6013282	86,5312306	76,32191806
0,01	73,9552466	85,3351648	82,3427595	94,24862021
0,001	77,134989	80,8034025	73,6821034	73,51350609
0,0001	55,6844339	63,9996859	53,9150635	57,25283342
				StandardDev

Temperature	99	999	9999	99999
DecreaseFactor	X	X	X	X
0,1	18682,5	18660	*18660,5*	18678,5
0,01	18616	18608	18598,5	18598
0,001	18612	18604,5	18632	18630,5
0,0001	18632,5	18620	18636	18634
				QUARTILE 1

Temperature	99	999	9999	99999
DecreaseFactor	X	X	X	X
0,1	18771,5	18760	*18767*	18766
0,01	18713,5	18717,5	18724	18707,5
0,001	18701,5	18709,5	18718	18709,5
0,0001	18694	18694	18694	18689
				QUARTILE 3

Temperature	99	999	9999	99999
DecreaseFactor	X	X	X	X
0,1	18732	18719	*18706*	18726
0,01	18665	18652	18656	18656
0,001	18659	18670	18672	18670
0,0001	18668	18659	18667	18663
				MEDIAN

Everywhere		
999 / 0,01	GLOBAL MIN	18440
99999 / 0,01	GLOBAL MAX	19072

2.5.3 Decision

In the statistical data we can see that (99,0.1) is obviously the worst solutions as the algorithm has not enough time to create diversity in order to find the best solutions and move away from local minimum. Looking at the mean and the median the couple (999,0.01) seems to be pretty good, but if we look at the first and third quartile it is possible to see that (9999,0.01) and (9999,0.0001) seem to also give good result. Finally, even if (9999,0.01) and (9999,0.0001) seem to be good candidate, (9999,0.01) takes more than twice loop than (999,0.1) and (9999,0.0001) more than ten times.

So the best set of parameters for the Simulated Annealing is Temperature = 999 and DecreaseFactor = 0.01.

2.6 Solutions evaluation

Intro In order to compare the Genetic Algorithm and the Simulated Annealing solutions, it has been decided to compare them using the same element which will represent the cost of each solutions : the number of times the cost evaluation is done. In the code is will be how many times the cost evaluations function is called.

2.6.1 Simulated Annealing

As the cost of the algorithm was defined as the number of times the cost evaluation is done, here is the mean value and standard deviation for the S.A. algorithm with the previous set of parameters (999,0.01). The evaluation is done by counting the number of times the cost evaluation is done until the algorithm find a candidate which has a cost $<= 18700$. For these values the S.A. algorithm ran more than 300 times.

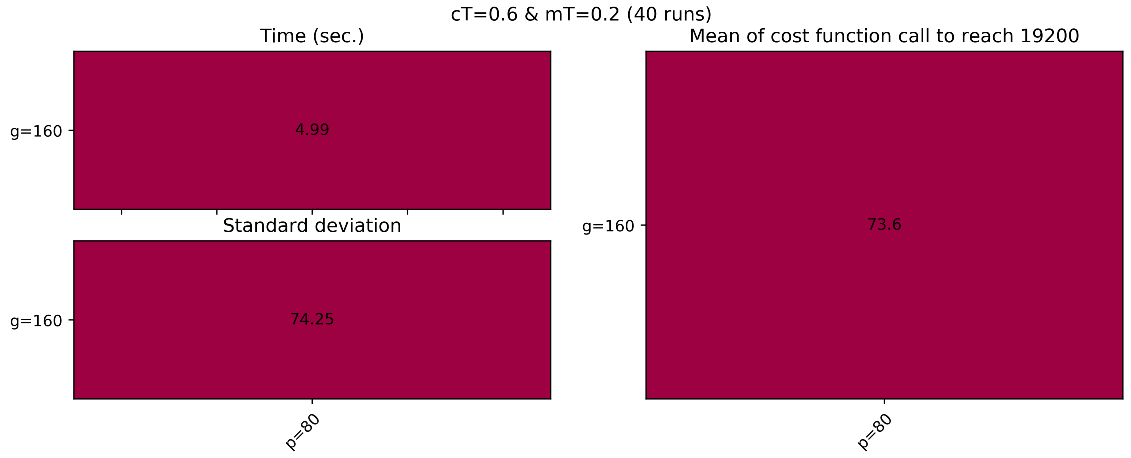
S.A. cost, Mean Value : 3225
S.A. cost, Standard Deviation : 1582

This numbers may seem big, however they don't represent the number in abscissa showed above in the Result section. In the graphics this is the number of loop in the While Loop and in each While Loop 4 new candidate are generated 4 times. So in each while loop the cost evaluation is done 16 times. So if we divided these number by 16 (Mean Value : 201 / Standard Deviation : 98), we have more relevant numbers to compare with the graphic.

2.6.2 Genetic Algorithm

As we can see in the different runs, the best mean value we can get is around 19100. So in order to determine the number of cost function call, I will look how many generations does it take to get get a cost below 19200 with the following parameters :

$n = 160$ $p = 80$ $C_T = 0.6$ $M_T = 0.2$



If we compare these values with the number of the simulated annealing approach, we get much fewer calls. Even if we get a slightly lower cost with the first approach **(18700 vs 19200)**, the genetic one if much cheaper in term of computing cost.

3 Conclusion

The airline sector his one of the most competitive in the world. Because of that, every cost that may be cut have to. In order to do that, many solutions to evaluate the profitability of a model exist. In our, we choose to evaluate a given model using two stochastic algorithm : Simulated Annealing and the genetic population based. From that, we decided to evaluate which of both may be the best fitted for our needs which are being able to get an aircraft distribution with the lowest cost possible.

In order to do that, we run with both methods multiple simulation in order to pick the best parameters for each implementation. When these arguments has been chosen, we fixed then and run the final simulation and counted how much the cost function has been called during the computation. The lower this number, the more effective the algorithm is.

We get these results :

$$Simulated\ annealing = 201\ and\ Genetic\ population\ based = 74\ (mean\ of\ 40\ runs)$$

It can be seen that the genetic population based algorithm require fewer call to the cost function than the simulated annealing one. However, it's important to underline that the second methods one get slightly better results (18652) than the first one (18838)

To conclude, we can say that our researches brought out that **the best algorithm to pick in term of computational cost is the genetic** one. However its important to highlight that we get a slightly lower cost with the simulated annealing and also that other stochastic approach may have be better (ant colony for example)

4 Annex - Codes (Python and C++)

4.1 Python code (Genetic algorithm)

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import sys
from scipy import stats
import statistics
import random
import copy
import math
import time


# Structure to store all the data needed per population member
class PopulationMember:
    def __init__(self, airplane):
        self.airplane = airplane
        self.cost = 0
        self.invert_cost = 0
        self.probability = 0


routes = np.array([0, 1, 2, 3, 4])
revenue_lost_per_passenger_turned_away = np.array([13, 20, 7, 7, 15])

passenger_demand_per_route = np.array([800, 900, 700, 650, 380])

number_of_aircraft_available_per_type = [10, 19, 25, 16]

aircraft_capacity_per_spot = np.matrix([[16, 10, 30, 23], [16, 10, 30, 23],
                                        [16, 10, 30, 23], [16, 10, 30, 23],
                                        [16, 10, 30, 23]])

operational_cost_per_spot = np.matrix([[12, 20, 30, 19], [2, 34, 10, 20],
                                       [43, 63, 40, 12], [32, 10, 6, 34],
                                       [20, 30, 10, 87]])

number_of_aircraft_per_spot = np.matrix([[0, 0, 0, 0], [0, 0, 0, 0],
                                         [0, 0, 0, 0], [0, 0, 0, 0],
                                         [0, 0, 0, 0]])


# Function to generate 1 column of a population member
def generate_column(row_nb, aircraft_type_count):
    column = np.zeros((row_nb))
    local_total = 0
    for index in range(0, row_nb):
        # pick a random number between 0 and the aircraft count
        val = random.randint(0, aircraft_type_count - local_total)
        local_total += val
        # if we are at the end but the total is not yet reached
        if aircraft_type_count != local_total and index == row_nb - 1:
            column[index] = aircraft_type_count - local_total
        # else, we add the random value to the airplane matrix
        else:
            column[index] = val
    return column


# Function to generate a population member
def generate_population_member():
    # get the dimension (row*column) of the number_of_aircraft_per_spot matrix)
    row_nb, column_nb = number_of_aircraft_per_spot.shape
    distribution = np.zeros((row_nb, column_nb))
    for index in range(0, column_nb):
        # get the aircraft count per type
        aircraft_type_count = number_of_aircraft_available_per_type[index]
        # generate the corresponding column and assign
        distribution[:, index] = generate_column(row_nb, aircraft_type_count)
    return distribution


# Function to generate the initial population
def generate_first_population(pop_count):
    all_population = []
    for index in range(0, pop_count):
        # generate a random distribution
        distribution = generate_population_member()
        all_population.append(PopulationMember(distribution))
    return all_population


# Function to computes the total revenu lost of one route
def get_revenu_lost_per_route(airplanes):
    revenu_lost = 0
    for index in routes:
        revenu_lost += max(((passenger_demand_per_route[index] - np.multiply(
            airplanes[index], aircraft_capacity_per_spot[index]).sum()) *
                            revenue_lost_per_passenger_turned_away[index]), 0)
    return revenu_lost


# The cost function
```

```
def cost_function(member):
    total_operating_cost = np.multiply(operational_cost_per_spot,
                                       member.airplane).sum()
    total_revenue_lost = get_revenu_lost_per_route(member.airplane)
    total_lost = total_operating_cost + total_revenue_lost
    return total_lost


# Function to compute the cost of all the population
def compute_costs(population):
    for index in range(0, len(population)):
        cost = cost_function(population[index])
        population[index].cost = cost
        population[index].invert_cost = 1 / cost
    return population


# Function to compute the picking probability according to the cost
def compute_probabilities(all_population):
    sum_cost = sum(p.invert_cost for p in all_population)
    for index in range(0, len(all_population)):
        probability = math.exp(
            math.exp((1 / all_population[index].cost) / sum_cost))
        all_population[index].probability = probability
    return all_population


# Function to pick the two best individual among the population
def pick_parents_elitist(all_population):
    parents = sorted(all_population, key=lambda k: k.cost)
    return [parents[0], parents[1]]


# Function to pick the two best individual among the population
# according to their probability
def pick_parents_proba(all_population):
    all_proba = list(map(lambda x: x.probability, all_population))
    return random.choices(all_population, weights=all_proba, k=2)


# Cross function
def cross(parent1, parent2, cross_threshold):
    # Copy both parents
    children = [
        PopulationMember(copy.deepcopy(parent1.airplane)),
        PopulationMember(copy.deepcopy(parent2.airplane))
    ]
    row_nb, column_nb = number_of_aircraft_per_spot.shape
    for index in range(0, column_nb):
        np.random.seed(10)
        rand = random.uniform(0, 1)
        # If the random is lower than the threshold, the two
        # column are switched
        if rand <= cross_threshold:
            children[0].airplane[:, index] = copy.deepcopy(
                parent2.airplane[:, index])
            children[1].airplane[:, index] = copy.deepcopy(
                parent1.airplane[:, index])
    return children[0], children[1]


# Mutation function by recreating the whole column
def mutation_generate_column(child, mutation_threshold):
    row_nb, column_nb = number_of_aircraft_per_spot.shape
    for index in range(0, column_nb):
        np.random.seed(10)
        rand = random.uniform(0, 1)
        if rand <= mutation_threshold:
            aircraft_type_count = number_of_aircraft_available_per_type[index]
            child.airplane[:, index] = generate_column(row_nb,
                                                       aircraft_type_count)
    return child


# Mutation function by switching two values of the same column
def mutation_switch_one_element(child, mutation_threshold):
    row_nb, column_nb = number_of_aircraft_per_spot.shape
    child_copy = copy.deepcopy(child)
    for index in range(0, column_nb):
        np.random.seed(10)
        rand = random.uniform(0, 1)
        if rand <= mutation_threshold:
            index1 = random.randint(0, row_nb - 1)
            index2 = random.randint(0, row_nb - 1)
            while index1 == index2:
                index2 = random.randint(0, row_nb - 1)
            tmp = child_copy.airplane[index1, index]
            child_copy.airplane[index1, index] = child_copy.airplane[index2,
                                                                      index]
            child_copy.airplane[index2, index] = tmp
    return child_copy


# Function to regenerate a new population
def generate_new_population(all_population, cross_threshold,
                            mutation_threshold):
```

```
    new_pop = []
    for index in range(0, len(all_population)):
        parents = pick_parents_elitist(all_population)
        parent1 = copy.deepcopy(parents[0])
        parent2 = copy.deepcopy(parents[1])
        child1, child2 = cross(parent1, parent2, cross_threshold)
        np.random.seed(10)
        rand_cross = random.uniform(0, 1)
        rand_mutation = random.uniform(0, 1)
        if rand_cross >= 0.5:
            picked_child = child1
        else:
            picked_child = child2
        if rand_mutation >= 0.5:
            picked_child = mutation_switch_one_element(picked_child,
                                                       mutation_threshold)
        new_pop.append(copy.deepcopy(picked_child))
    return new_pop


def generate_plot(generation, all_mins, initial_population_count,
                  cross_threshold, mutation_threshold):
    results = list(map(lambda x: x.cost, all_mins))
    print(all_mins[generation - 1].airplane)
    print(all_mins[generation - 1].cost)
    plt.plot(results)
    title = 'Total␣costs␣:␣' + str(
        all_mins[generation - 1].cost) + '$' + '␣|␣Generation␣:␣' + str(
            generation) + ',␣Initial␣population␣:␣' + str(
                initial_population_count)
    plt.xlabel(title)
    plt.savefig('../outputs/airplanes_g' + str(generation) + '_i' +
                str(initial_population_count) + '_cT' + str(cross_threshold) +
                '_mT' + str(mutation_threshold) + '.png')


def draw_table(mean_of_mins, std_of_mins, times_of_mins, row_labels,
               col_labels, title):

    fig = plt.figure(figsize=(12, 4))
    gs = GridSpec(nrows=2, ncols=2)

    ax1 = fig.add_subplot(gs[:, 1])
    ax2 = fig.add_subplot(gs[1, 0])
    ax3 = fig.add_subplot(gs[0, 0])

    plt.set_cmap('Spectral')
    im = ax1.imshow(mean_of_mins, aspect='auto')
```

```
im2 = ax2.imshow(mean_of_mins, aspect='auto')
im3 = ax3.imshow(mean_of_mins, aspect='auto')

ax1.set_title("Mean␣of␣cost␣function␣call␣to␣reach␣19200")
ax2.set_title("Standard␣deviation")
ax3.set_title("Time␣(sec.)")

# We want to show all ticks...
ax1.set_xticks(np.arange(len(col_labels)))
ax1.set_yticks(np.arange(len(row_labels)))
# We want to show all ticks...
ax2.set_xticks(np.arange(len(col_labels)))
ax2.set_yticks(np.arange(len(row_labels)))
# We want to show all ticks...
ax3.set_yticks(np.arange(len(row_labels)))
# ... and label them with the respective list entries
ax1.set_xticklabels(list(map(lambda x: "p=" + str(x), col_labels)))
ax1.set_yticklabels(list(map(lambda x: "g=" + str(x), row_labels)))
# ... and label them with the respective list entries
ax2.set_xticklabels(list(map(lambda x: "p=" + str(x), col_labels)))
ax2.set_yticklabels(list(map(lambda x: "g=" + str(x), row_labels)))
# ... and label them with the respective list entries
ax3.set_yticklabels(list(map(lambda x: "g=" + str(x), row_labels)))

# Rotate the tick labels and set their alignment.
plt.setp(ax1.get_xticklabels(),
         rotation=45,
         ha="right",
         rotation_mode="anchor")
# Rotate the tick labels and set their alignment.
plt.setp(ax2.get_xticklabels(),
         rotation=45,
         ha="right",
         rotation_mode="anchor")
# Rotate the tick labels and set their alignment.
plt.setp(ax3.get_xticklabels(), visible=False)

# Loop over data dimensions and create text annotations.
for i in range(len(row_labels)):
    for j in range(len(col_labels)):
        text = ax1.text(j,
                        i,
                        mean_of_mins[i][j],
                        ha="center",
                        va="center",
                        color="black")
        text2 = ax2.text(j,
                         i,
                              std_of_mins[i][j],
                              ha="center",
                              va="center",
                              color="black")
            text3 = ax3.text(j,
                              i,
                              times_of_mins[i][j],
                              ha="center",
                              va="center",
                              color="black")
    plt.suptitle(title)
    print('matplotlib-table_' + str(title))
    plt.savefig('../outputs/matplotlib-table_' + str(title) + '.png',
                bbox_inches='tight',
                pad_inches=0.05,
                dpi=400)


def main():
    mins_of_mins = []
    std_of_mins = []
    times_of_mins = []
    calls_of_mins = []
    calls_of_mins_std = []
    generations = [160]
    init_pops = [80]
    reach = 19200
    cross_threshold = float(sys.argv[1])
    mutation_threshold = float(sys.argv[2])
    runs = 2
    # For each generations
    for gen in range(0, len(generations)):
        generation = generations[gen]
        all_mins_mean = []
        all_mins_std = []
        all_mins_times = []
        all_calls_of_pop = []
        all_calls_of_pop_std = []
        # For each population
        for init_p in range(0, len(init_pops)):
            initial_population_count = init_pops[init_p]
            start = time.time()
            mins_of_pop = []
            calls_of_pop = []
            # Run n times
            for n in range(0, runs):
                all_mins_of_the_current_gen = []
                # initial population
```

```
                    all_population_of_the_current_gen = generate_first_population(
                        initial_population_count)
                    compute_cost_call = 0
                    print('Time␣:', n, 'and␣generation␣:', generation, 'for␣pop␣:',
                          initial_population_count)
                    # From 0 to the size of the generation
                    for gen_2 in range(0, generation):
                        # Compute the cost of all the population
                        all_population_of_the_current_gen = compute_costs(
                            all_population_of_the_current_gen)
                        # Get the minimum cost and add it into the list
                        min_cost = min(all_population_of_the_current_gen,
                                       key=lambda x: x.cost)
                        all_mins_of_the_current_gen.append(min_cost)
                        if (min_cost.cost >= reach):
                            compute_cost_call += 1
                        # Compute probabilities
                        all_population_of_the_current_gen = compute_probabilities(
                            all_population_of_the_current_gen)
                        # Pick parents
                        all_population_of_the_current_gen = generate_new_population(
                            all_population_of_the_current_gen, cross_threshold,
                            mutation_threshold)
                    # Pick the last min (the best we get) and put it
                    # into the min of mins
                    mins_of_pop.append(all_mins_of_the_current_gen[generation -
                                                                   1].cost)
                    # Pick the number of time the cost function have
                    # been called to reach the REACH value
                    calls_of_pop.append(compute_cost_call)
                # After 10 tried, only keep the mean of all the mins
                print("##############")
                all_mins_mean.append(round(statistics.mean(mins_of_pop)))
                all_mins_std.append(round(statistics.stdev(mins_of_pop), 2))
                all_calls_of_pop.append(round(statistics.mean(calls_of_pop), 2))
                all_calls_of_pop_std.append(
                    round(statistics.stdev(calls_of_pop), 2))
                all_mins_times.append(round((time.time() - start) / runs, 2))
            # Put the array of mins into the array containing all array of mins
            mins_of_mins.append(all_mins_mean)
            std_of_mins.append(all_mins_std)
            times_of_mins.append(all_mins_times)
            calls_of_mins.append(all_calls_of_pop)
            calls_of_mins_std.append(all_calls_of_pop_std)
            #generate_plot(generation, mins_of_pop, initial_population_count,
            #              cross_threshold, mutation_threshold)
        draw_table(mins_of_mins,
                   std_of_mins,
                    times_of_mins,
                    row_labels=generations,
                    col_labels=init_pops,
                    title='cT=' + str(cross_threshold) + '␣&␣mT=' +
                    str(mutation_threshold) + "␣(" + str(runs) + "␣runs)")


main()
```

4.2 C++ code (Simulated annealing algorithm)

```
#include <iostream>
#include <vector>
#include <random>
#include <math.h>
#include <fstream>
#include <string>
#include <thread>

class Data
{
public:
    Data() {}
    ~Data() {}

    std::vector<int> m_vAircraftAllocation;

    std::vector<float> m_vCosts;
    std::vector<float> m_vRevenueLost;
    std::vector<float> m_vPassengerDemand;

    std::vector<int> m_vAircraftAvailable;
    std::vector<int> m_vAircraftCapacity;

    std::vector<std::vector<int>> m_bAircraftRouteAllocation; // [routeNumber][airc

    std::vector<int> m_vPassengerTurnedAway;

    bool m_bVerbose = false;

    void LoadDataSet_1()
    {
        m_vCosts = {18, 21, 18, 16, 10, 15, 16, 14, 9, 10, 9, 6, 17, 16, 17, 15, 10
        m_vRevenueLost = {13, 13, 7, 7, 1};
        m_vAircraftCapacity = {16, 15, 28, 23, 81, 10, 14, 15, 57, 5, 7, 29, 9, 11,
        m_vAircraftAvailable = {10, 19, 25, 15};

        m_bAircraftRouteAllocation = {{0, 5, 10, 15}, {1, 6, 11, 16}, {2, 7, 12, 17

        m_vPassengerDemand = {250, 100, 180, 100, 600};
    }

    void LoadDataSet_2()
    {
        m_vCosts = {12, 2, 43, 32, 20, 20, 34, 63, 10, 30, 30, 10, 40, 6, 10, 19, 2
        m_vRevenueLost = {13, 20, 7, 7, 15};
        m_vAircraftCapacity = {16, 16, 16, 16, 16, 10, 10, 10, 10, 10, 30, 30, 30,
    m_vAircraftAvailable = {10, 19, 25, 16};

    m_bAircraftRouteAllocation = {{0, 5, 10, 15}, {1, 6, 11, 16}, {2, 7, 12, 1

    m_vPassengerDemand = {800, 900, 700, 650, 380};
}

float CostEvaluation(const std::vector<int> &p_vAircraftAllocation)
{
    if (!CheckAircraftAllocationIntegrity(p_vAircraftAllocation))
    {
        std::cout << "␣Wrong␣Size␣Data␣";
    }

    float l_fResult = 0;

    // Calcul total operating cost
    for (int index = 0; index < p_vAircraftAllocation.size(); ++index)
    {
        l_fResult += m_vCosts[index] * p_vAircraftAllocation[index];
    }

    if (m_bVerbose)
        std::cout << "␣total␣route␣operating␣cost␣:␣" << l_fResult << std::endl

    // calcul number of passenger turned away per route
    std::vector<float> l_vPassengersTurnedAway;

    // for each route
    for (int index = 0; index < m_vRevenueLost.size(); ++index)
    {
        float l_fPassengerTA = m_vPassengerDemand[index];

        // for each aircraft type in route
        for (auto idAircraftRoute : m_bAircraftRouteAllocation[index])
        {
            l_fPassengerTA -= m_vAircraftCapacity[idAircraftRoute] * p_vAircra
        }

        if (l_fPassengerTA < 0) // more place than expected passengers
            l_vPassengersTurnedAway.push_back(0);
        else // less place
            l_vPassengersTurnedAway.push_back(l_fPassengerTA);

        if (m_bVerbose)
            std::cout << "passengers␣TA␣on␣route␣" << index + 1 << "␣:␣" << (l_
    }
    // calcul revenue lost
    for (int route = 0; route < 5; ++route)
    {
        l_fResult += m_vRevenueLost[route] * l_vPassengersTurnedAway[route];

        if (m_bVerbose)
            std::cout << "revenue␣lost␣on␣route␣" << route + 1 << "␣:␣" << m_v
    }

    if (m_bVerbose)
        std::cout << "result␣:␣" << l_fResult << std::endl;

    return l_fResult;
}

std::vector<int> GenerateRandomAllocation()
{
    std::vector<int> l_vRandomAllocation(20, 0);

    // For each aircraft type
    for (int aircraftType = 0; aircraftType < m_vAircraftAvailable.size(); ++ai
    {
        int l_iAircraftLeft = m_vAircraftAvailable[aircraftType];

        int l_iImproveRandom = 2;

        // For each route
        for (auto routeAirplane : m_bAircraftRouteAllocation)
        {
            if (l_iAircraftLeft != 0)
            {
                int l_iAircraftAttribution = GenerateRandomNumber(0, l_iAircra
                l_iAircraftLeft -= l_iAircraftAttribution;
                l_vRandomAllocation[routeAirplane[aircraftType]] = l_iAircraftA
            }
        }

        // if some airplane is still available, put it randomly in one route
        if (l_iAircraftLeft != 0)
            l_vRandomAllocation[m_bAircraftRouteAllocation[GenerateRandomNumber
    }

    //printAllocation(l_vRandomAllocation);

    return l_vRandomAllocation;
}

std::vector<std::vector<int>> GenerateNeighbors(const std::vector<int> &p_vInit
    {
        std::vector<std::vector<int>> l_vNeighbors;

        int rd;

        for (int count = 0; count < p_iNeighborsNumber; ++count)
        {
            rd = GenerateRandomNumber(0, 2);

            if (rd == 0)
            {
                l_vNeighbors.push_back(SingleRandomPermutation(p_vInitAllocation, l
            }
            else if (rd == 1)
            {
                l_vNeighbors.push_back(RandomPlusOneMinusOne(p_vInitAllocation, l_i
            }
            else if (rd == 2)
            {
                l_vNeighbors.push_back(WholeNewColumn(p_vInitAllocation, l_iAircraf
            }
        }

        return l_vNeighbors;
    }

    void printAllocation(const std::vector<int> &p_vAllocation)
    {
        int route = 1;

        // For each aircraft type
        for (int aircraftType = 0; aircraftType < m_vAircraftAvailable.size(); ++ai
        {
            std::cout << "␣Aircraft␣type␣:␣" << aircraftType + 1 << std::endl;

            for (auto routeAirplane : m_bAircraftRouteAllocation)
            {
                std::cout << "x" << routeAirplane[aircraftType] + 1 << "␣:␣" << p_
            }
        }
    }

private:
    bool CheckAircraftAllocationIntegrity(const std::vector<int> &p_vAircraftAlloca
    {
        if (p_vAircraftAllocation.size() != m_vAircraftCapacity.size())
            return false;
    return true;
}

// Generate random number between min and max
float GenerateRandomNumber(int min, int max)
{
    // https://en.cppreference.com/w/cpp/numeric/random
    // Seed with a real random value, if available
    std::random_device r;

    // Choose a random mean between min and max
    std::default_random_engine e1(r());
    std::uniform_int_distribution<int> uniform_dist(min, max);
    float number = ((float)uniform_dist(e1));
    return number;
}

std::vector<int> SingleRandomPermutation(const std::vector<int> &p_vAllocation,
{
    std::vector<int> l_vOutput = p_vAllocation;

    // Compute index
    int l_iIndexToPermuteFirst = GenerateRandomNumber(aircraftType * 5, (aircra
    int l_iIndexToPermuteSecond = l_iIndexToPermuteFirst;

    while (l_iIndexToPermuteFirst == l_iIndexToPermuteSecond)
        l_iIndexToPermuteSecond = GenerateRandomNumber(aircraftType * 5, (aircr

    // Permute
    int temp = l_vOutput[l_iIndexToPermuteFirst];
    l_vOutput[l_iIndexToPermuteFirst] = l_vOutput[l_iIndexToPermuteSecond];
    l_vOutput[l_iIndexToPermuteSecond] = temp;

    return l_vOutput;
}

std::vector<int> WholeNewColumn(const std::vector<int> &p_vAllocation, const i
{
    std::vector<int> l_vOutput = p_vAllocation;

    int l_iAircraftLeft = m_vAircraftAvailable[aircraftType];

    int l_iImproveRandom = 2;

    // For each airplane type in route
    for (auto routeAirplane : m_bAircraftRouteAllocation)
    {
        if (l_iAircraftLeft != 0)
        {
            int l_iAircraftAttribution = GenerateRandomNumber(0, l_iAircraftLe
            l_iAircraftLeft -= l_iAircraftAttribution;
            l_vOutput[routeAirplane[aircraftType]] = l_iAircraftAttribution;
        }
    }

    // if some airplane is still available, put it randomly in one route
    if (l_iAircraftLeft != 0)
        l_vOutput[m_bAircraftRouteAllocation[GenerateRandomNumber(0, 3)][aircra

    return l_vOutput;
}

std::vector<int> RandomPlusOneMinusOne(const std::vector<int> &p_vAllocation, c
{
    std::vector<int> l_vOutput = p_vAllocation;

    // Compute index
    int l_iIndexToPermuteFirst = GenerateRandomNumber(aircraftType * 5, (aircra

    bool isValid = false;
    // Error check
    for (int index = aircraftType * 5; index <= (aircraftType * 5) + 4; ++inde
    {
        if (index != l_iIndexToPermuteFirst)
            if (p_vAllocation[index] > 0)
                isValid = true;
    }

    int l_iIndexToPermuteSecond = l_iIndexToPermuteFirst;

    if (isValid)
    {
        while (l_iIndexToPermuteFirst == l_iIndexToPermuteSecond || l_vOutput[l
            l_iIndexToPermuteSecond = GenerateRandomNumber(aircraftType * 5, (a

        // apply
        l_vOutput[l_iIndexToPermuteFirst] += 1;
        l_vOutput[l_iIndexToPermuteSecond] -= 1;
    }
    else // only one positive value
    {
        while (l_iIndexToPermuteFirst == l_iIndexToPermuteSecond)
            l_iIndexToPermuteSecond = GenerateRandomNumber(aircraftType * 5, (a

        // apply
        l_vOutput[l_iIndexToPermuteFirst] -= 1;
```

```
            l_vOutput[l_iIndexToPermuteSecond] += 1;
        }

        return l_vOutput;
    }
};

class SimulatedAnnealing
{

public:
    SimulatedAnnealing() {}
    SimulatedAnnealing(float p_fTemperature, float p_fDecreaseFactor) : m_fTempera

    ~SimulatedAnnealing() {}

    float m_fTemperature;

    float m_fDecreaseFactor;

    int m_iCountStatic;

    const int m_iTryNumber = 3;
    int m_iTryNumberCount;

    int m_iStepNumber = 4; // equals to the number of aircraft type

    float ApplyAlgorithm(Data &p_Data)
    {
        float l_fTemperature = m_fTemperature; // Set the temperature value

        // initiate first candidate and its cost
        std::vector<int> l_vCurrentCandidate = p_Data.GenerateRandomAllocation();
        float l_fCurrentCost = p_Data.CostEvaluation(l_vCurrentCandidate);

        std::vector<float> l_vCostHistory{l_fCurrentCost}; // to store the cost his

        std::vector<std::vector<int>> l_vNeighbors; // variable declaration

        int l_iLoopCount = 0;
        int l_iCostLimit = 18700;

        // Loop on temperature condition
        while (l_fTemperature > 0.1)
        {
            // Each Step before decreasing the temperature
            for (int step = 0; step < m_iStepNumber; ++step)
            {
```

```
        // Step 1 : Generate Neighbors
        l_vNeighbors = p_Data.GenerateNeighbors(l_vCurrentCandidate, step);

        // Step 2 : Find Minimum cost neighbor
        std::vector<int> l_vMinimunCostNeighbor;
        float l_fCost = FLT_MAX;

        for (auto NeighBorsCandidate : l_vNeighbors)
        {
            float l_fNewCost = p_Data.CostEvaluation(NeighBorsCandidate);
            // Loop Count
            l_iLoopCount++;

            if (l_fNewCost < l_fCost)
            {
                l_vMinimunCostNeighbor = NeighBorsCandidate;
                l_fCost = l_fNewCost;
            }
        }

        // Step 3 : Apply SA condition

        float l_fNeighborsCandidateCost = p_Data.CostEvaluation(l_vMinimun
        float l_fCostDifference = l_fNeighborsCandidateCost - l_fCurrentCo

        // If it's a better value choose it
        if (l_fCostDifference < 0)
        {
            l_vCurrentCandidate = l_vMinimunCostNeighbor;
            l_fCurrentCost = l_fNeighborsCandidateCost;
        }
        // Condition with the temperature value
        else if (GenerateRandomNumber(0, 100) < exp((-l_fCostDifference) /
        {
            l_vCurrentCandidate = l_vMinimunCostNeighbor;
            l_fCurrentCost = l_fNeighborsCandidateCost;
        }
    }

    // Cost History
    l_vCostHistory.push_back(l_fCurrentCost);

    // Decrease the temperature
    l_fTemperature *= (1 - m_fDecreaseFactor);
}

// Generate csv for cost history
std::ofstream myfile;
```

```
        myfile.open("result_.csv");
        myfile << "Cost␣History.\n";
        myfile << "T␣=␣" << m_fTemperature << ",\n";
        myfile << "DFactor␣=␣" << m_fDecreaseFactor << ",\n";
        myfile << "FinalCost␣=␣" << l_vCostHistory.back() << ",\n";
        for (auto cost : l_vCostHistory)
            myfile << cost << ",\n";
        myfile.close();

        // Print Solution
        p_Data.printAllocation(l_vCurrentCandidate);

        return l_vCostHistory.back();
    }

private:
    // Generate random number between min and max and divided it by 100
    float GenerateRandomNumber(int min, int max)
    {
        // https://en.cppreference.com/w/cpp/numeric/random
        // Seed with a real random value, if available
        std::random_device r;

        // Choose a random mean between 0 and 100
        std::default_random_engine e1(r());
        std::uniform_int_distribution<int> uniform_dist(min, max);
        float number = ((float)uniform_dist(e1)) / 100;

        return number;
    }
};

int main()
{
    std::cout << "##########␣Aircraft␣Allocation␣Problem␣##########\n";

    Data l_Data;
    SimulatedAnnealing l_SA(999, 0.01);

    l_Data.LoadDataSet_2();

    l_SA.ApplyAlgorithm(l_Data);

    std::cout << "the␣end";
}
```