Genetic Algorithms

Aim:
To apply genetic algorithm for a given problem of Flow Shop Scheduling

Problem Statement: *Examining various genetic operators to design a genetic algorithm for the flowshop scheduling problem with an objective to minimize the makespan of the jobs.*

Tool/Language: Python 3

Report Summary:

1. **Problem that the paper solves:**

Flowshop scheduling for minimizing the makespan is one of the most well-known problems in the area of scheduling. Therefore, this paper aims at examining various genetic operators to design an efficient genetic algorithm for solving this flowshop scheduling problem by minimising the makespan of the jobs. This paper indicates that two-point crossover and shift change mutation schemes prove to be an effective solution to solving this problem. The efficiency of this process is compared with many search algorithms as well to figure out the best possible algorithm.

2. **Components involved in Genetic Algorithm**
    a. **Type of Encoding Scheme used:**

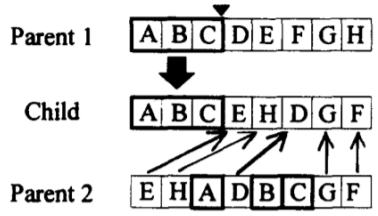
In this paper, each solution of the optimization problem is encoded as a string. A sequence of jobs is represented as a string in this paper for the flowshop scheduling problem. For example, a string 'ABCDEF' represents a sequence of jobs where job A is processed first, followed by job B and so on. If a combination of string is 'ABCADE' then it cannot be counted as a valid solution to the problem as it contains 2 occurrences of job A and no occurrence of job F. We denote the sequence of n jobs by an n-dimensional vector x = (xl ..... xj ..... xn) where xj denotes the j-th processing job. That is, the n-dimensional vector x is handled as a string in genetic algorithms.

    b. **Type of Selection, Mutation and Crossover Schemes used:**

Selection:
Selection is an operation to select two parent strings for generating a new string (i.e., child). Let Npop be the number of solutions in each population in genetic algorithms, i.e., Npop is the population size.
Conventionality calls for the random selection of parents, the beneficial effects of genetic search being gained by the replacement of "poor" chromosomes. But it is also possible to allow at least one parent to be selected according to its fitness, with lower-valued chromosomes being the favoured ones. The selection of parents was then made in accordance with the probability distribution **p([k]) = 2k/(M*(M+1))**, where [k] is the kth chromosome in ascending order of fitness (i.e. descending order of makespan). This implies that the median value has a chance of **1/M** of being selected, while the Mth (the fittest) has a chance of **2/(M + 1)**, roughly twice that of the median.

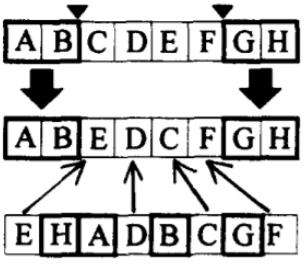
Crossover:
Crossover is an operation to generate a new string (i.e., child) from two parent strings. Since our flowshop scheduling problem is a sequencing problem of n jobs, various crossover operators proposed for traveling salesman problems and scheduling problems are applicable.
There were 10 crossover operators analysed in this paper and the following 6 turned out to be the most effective.

i. *One-point crossover:* One point is randomly selected for dividing one parent. The set of jobs on one side (each side is chosen with the same probability) is inherited from one parent to the child, and the other jobs are placed in the order of their appearance in the other parent.

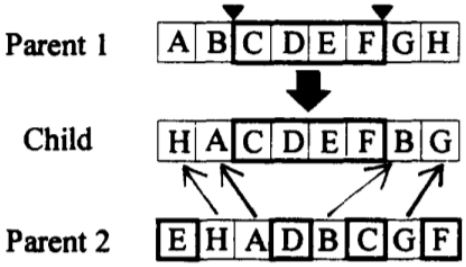


(a) One-point crossover

ii. *Two-point crossover:* There are three versions of this crossover. At first two points are randomly selected for dividing one parent. In the first version, jobs outside the parent are inherited from parent 1 and other jobs are arranged similar to one-point crossover. In version 2, the set of jobs between two randomly selected points is always inherited from one parent to the child, and the other jobs are placed in the same manner as the one-point crossover. Version 3 is a mixture of the version 1 and 2.

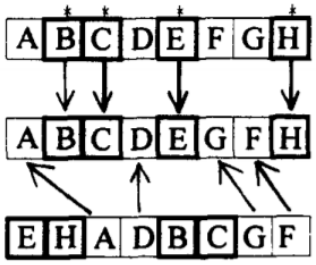


(b) Two-point crossover Ver. I



(c) Two-point crossover Ver. II

iii. *Position based crossover:* There are two versions in this type of crossover. The jobs at randomly selected positions marked by "*" are inherited from one parent to the child, and the other jobs are placed in the order of their appearance in the other parent. In version 1, the number of those positions is first determined as a random integer in [1, n], then positions are randomly selected. In version 2, each position is independently marked with the probability of 0.5, while the number of marked positions is the same as that defined in version 1.

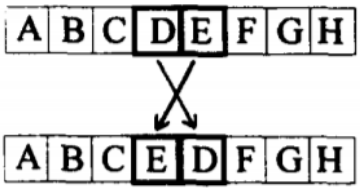


(d) Position based crossover

Mutation:

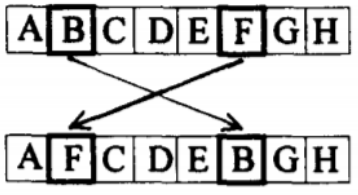
Mutation is an operation to change the order of n jobs in each string generated by a crossover operator. Such a mutation operation can be viewed as a transition from a current solution to its neighborhood solution in local search algorithms. There are four types of mutations reviewed in this paper which are as follows:

i. *Adjacent two-job change:* In this type of mutation, the adjacent two jobs to be changed are randomly selected.



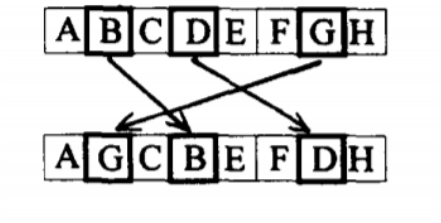
(a) Adjacent two-job change

ii. *Arbitrary two-job change:* In this type of mutation, the two jobs to be changed are arbitrary and randomly selected. This mutation includes the adjacent two-job change as a special case.



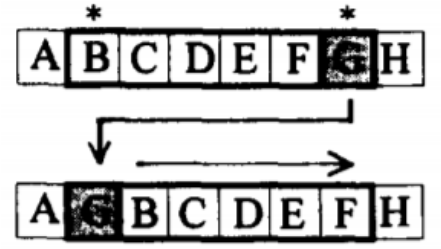
(b) Arbitrary two-job change

iii. *Arbitrary three-job change:* In this type of mutation, the three jobs to be changed are arbitrary and randomly selected, and the order of the selected jobs after the mutation is randomly specified. This mutation includes the above two mutation operators as a special case.



(c) Arbitrary three-job change

iv. *Shift change:* In this mutation, a job at one position is removed and put at another position. The two positions are randomly selected. This mutation includes the adjacent two-job change as a special case and has an intersection with the arbitrary three-job change.



(d) Shift

c. Fitness Function Used:

In minimization problems, the low valued chromosomes are the "good" chromosomes. One approach is to define fitness as (Vmax- V), but of course we are unlikely to know what Vmax is. We could use the largest value found so far as a surrogate for Vmax, but in practice this method was not very good at distinguishing between chromosomes. In essence, if the population consists of many chromosomes whose fitness values are relatively close, the resolution between "good" and "bad" ones is not of a high order. This approach was therefore abandoned, in favour of a simple ranking mechanism. Sorting the chromosomes is a simple task, and updating the list is also straightforward.

The selection of parents was then made in accordance with the probability distribution ***p([k]) = 2k/(M*(M+1))***, where [k] is the kth chromosome in ascending order of fitness (i.e. descending order of makespan). This implies that the median value has a chance of 1/M of being selected, while the Mth (the fittest) has a chance of 2/(M + 1), roughly twice that of the median.

3. Results observed by the authors:

The authors have applied the genetic algorithm, local search algorithm, the taboo search algorithm and the simulated annealing algorithm to 100 test problems with 20 jobs and 10 machines.For comparison, we also applied a random sampling technique with the same computation load as the other algorithms. It was observed that the average makespans obtained by each algorithm are normalized using the results by the simulated annealing algorithm with

200,000 evaluations. We can also see that the genetic algorithm, which is much superior to the random sampling technique, is a bit inferior to the other search algorithms.

4. **Observation and Conclusion:**

In this paper, we examined the performance of genetic algorithms in order to specify some genetic operators (e.g., crossover and mutation) and parameters (e.g., population size, crossover probability and mutation probability) for the flowshop scheduling problem. By computer simulations, we showed that the genetic algorithm was much superior to a random sampling technique but it was a bit inferior to other search algorithms such as local search, taboo search and simulated annealing. In order to improve the performance of the genetic algorithm, we examined two hybrid algorithms: genetic local search and genetic simulated annealing. We also examined some variants of their hybrid algorithms. By computer simulations, we showed that the hybrid algorithms outperformed the other algorithms (i.e., the genetic algorithm, the local search algorithm, the taboo search algorithm and the simulated annealing algorithm).

Algorithm:
Step 1: Initialization - Randomly generate an initial population Wt of Npop strings (i,e., Npop solutions).
Step 2: Selection - Select Npop pairs of strings from a current population according to the selection probability.
Step 3: Crossover - Apply one of the above crossover operators to each of the selected pairs in Step 2 to generate Npop solutions with the crossover probability Pc. If the crossover operator is not applied to the selected pair, one of the selected solutions remains as a new string.
Step 4: Mutation - Apply one of the above mutation operators to each of the generated Npop strings with the mutation probability Pm (we assign the mutation probability not to each bit but to each string).
Step 5: Elitist - Randomly remove one string from the current population and add the best string in the previous population to the current one.
Step 6: Termination - If a prespecified stopping condition is satisfied, stop this algorithm. Otherwise, return to Step 2

Code:

```
import numpy as np
import math
import time
import random
import itertools
import queue
import pandas as pd
from IPython.display import display, Markdown
# Dataset number. 1, 2 or 3
dataset = "2"
```

```
if dataset == "1":
   optimalObjective = 4534
elif dataset == "2":
   optimalObjective = 920
else:
   optimalObjective = 1302

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

# number of jobs
n = int(l[0])

# number of machines
m = int(l[1])

# ith job's processing time at jth machine
cost = []

for i in range(n):
   temp = []
   for j in range(m):
       temp.append(0)
   cost.append(temp)

for i in range(n):
   line = f.readline().split()
   for j in range(int(len(line)/2)):
       cost[i][j] = int(line[2*j+1])

f.close()
def initialization(Npop):
   pop = []
   for i in range(Npop):
       p = list(np.random.permutation(n))
```

```
        while p in pop:
            p = list(np.random.permutation(n))
        pop.append(p)

    return pop

def calculateObj(sol):
    qTime = queue.PriorityQueue()

    qMachines = []
    for i in range(m):
        qMachines.append(queue.Queue())

    for i in range(n):
        qMachines[0].put(sol[i])

    busyMachines = []
    for i in range(m):
        busyMachines.append(False)

    time = 0

    job = qMachines[0].get()
    qTime.put((time+cost[job][0], 0, job))
    busyMachines[0] = True

    while True:
        time, mach, job = qTime.get()
        if job == sol[n-1] and mach == m-1:
            break
        busyMachines[mach] = False
        if not qMachines[mach].empty():
                j = qMachines[mach].get()
                qTime.put((time+cost[j][mach], mach, j))
                busyMachines[mach] = True
        if mach < m-1:
            if busyMachines[mach+1] == False:
```

```
                qTime.put((time+cost[job][mach+1], mach+1, job))
                busyMachines[mach+1] = True
            else:
                qMachines[mach+1].put(job)

    return time


def selection(pop):
    popObj = []
    for i in range(len(pop)):
        popObj.append([calculateObj(pop[i]), i])

    popObj.sort()

    distr = []
    distrInd = []

    for i in range(len(pop)):
        distrInd.append(popObj[i][1])
        prob = (2*(i+1)) / (len(pop) * (len(pop)+1))
        distr.append(prob)

    parents = []
    for i in range(len(pop)):
        parents.append(list(np.random.choice(distrInd, 2, p=distr)))

    return parents

def crossover(parents):
    pos = list(np.random.permutation(np.arange(n-1)+1)[:2])

    if pos[0] > pos[1]:
        t = pos[0]
        pos[0] = pos[1]
        pos[1] = t
```

```
    child = list(parents[0])

    for i in range(pos[0], pos[1]):
        child[i] = -1

    p = -1
    for i in range(pos[0], pos[1]):
        while True:
            p = p + 1
            if parents[1][p] not in child:
                child[i] = parents[1][p]
                break

    return child


def mutation(sol):
    pos = list(np.random.permutation(np.arange(n))[:2])

    if pos[0] > pos[1]:
        t = pos[0]
        pos[0] = pos[1]
        pos[1] = t

    remJob = sol[pos[1]]

    for i in range(pos[1], pos[0], -1):
        sol[i] = sol[i-1]

    sol[pos[0]] = remJob

    return sol

def elitistUpdate(oldPop, newPop):
    bestSolInd = 0
    bestSol = calculateObj(oldPop[0])
```

```
    for i in range(1, len(oldPop)):
        tempObj = calculateObj(oldPop[i])
        if tempObj < bestSol:
            bestSol = tempObj
            bestSolInd = i

    rndInd = random.randint(0,len(newPop)-1)

    newPop[rndInd] = oldPop[bestSolInd]

    return newPop

# Returns best solution's index number, best solution's objective value
and average objective value of the given population.
def findBestSolution(pop):
    bestObj = calculateObj(pop[0])
    avgObj = bestObj
    bestInd = 0
    for i in range(1, len(pop)):
        tObj = calculateObj(pop[i])
        avgObj = avgObj + tObj
        if tObj < bestObj:
            bestObj = tObj
            bestInd = i

    return bestInd, bestObj, avgObj/len(pop)
popSize1 = [3,3,3,3,3,5,5,5,5,5,10,10,10,10,10]
objVal1                                                                =
[4534,4534,4534,4534,4534,4534,4534,4534,4534,4534,4534,4534,4534,4534,453
4]
avgObj1                                                                =
[4929.33,4840.33,5002.00,5154.33,4836.00,5066.80,5356.20,5663.20,4850.20,5
050.60,5530.40,5839.50,5671.60,5725.70,5459.10]
gap1                                                                   =
[0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.0
0]
cpuTime1                                                               =
```

```
[24.80,25.36,26.22,26.35,25.44,43.37,43.84,43.72,44.11,43.64,86.21,86.90,8
2.90,80.21,68.50]

popSize2 = [3,3,3,3,3,5,5,5,5,5,10,10,10,10,10]
objVal2 = [920,929,920,920,937,931,937,937,926,920,937,931,929,938,937]
avgObj2                                                                 =
[1001.00,1002.33,1055.00,964.67,1028.67,1052.40,1185.20,1074.00,1076.00,10
63.60,1177.00,1166.00,1139.80,1182.50,1181.40]
gap2                                                                    =
[0.00,0.98,0.00,0.00,1.85,1.20,1.85,1.85,0.65,0.00,1.85,1.20,0.98,1.96,1.8
5]
cpuTime2                              =                              [41.74,42.15,
45.23,40.75,39.75,64.36,64.04,64.76,67.21,66.68,123.27,122.97,122.32,
122.29,122.42]

popSize3 = [3,3,3,3,3,5,5,5,5,5,10,10,10,10,10]
objVal3                                                                 =
[1302,1302,1302,1302,1302,1302,1302,1302,1302,1323,1302,1302,1302,1302,132
3]
avgObj3       =       [1363.00,       1390.00,1341.00,1302.00,       1346.33,
1521.20,1403.40,1398.40,1497.00,1453.00,                               1610.50,
1619.90,1654.90,1600.30,1678.80]
gap3                                                                    =
[0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,1.61,0.00,0.00,0.00,0.00,1.6
1]
cpuTime3        =        [57.17,        57.09,56.90,57.42,56.91,94.06,93.96,
93.88,93.97,93.81,182.90,189.32,199.76,200.02,193.59]

#dSet = dSet1 + dSet2 + dSet3
popSize = popSize1 + popSize2 + popSize3
objVal = objVal1 + objVal2 + objVal3
avgObj = avgObj1 + avgObj2 + avgObj3
gap = gap1 + gap2 + gap3
cpuTime = cpuTime1 + cpuTime2 + cpuTime3


dfDict1 = {'Pop Size':popSize1, 'Obj Val':objVal1, 'Pop Avg Obj':avgObj1,
```

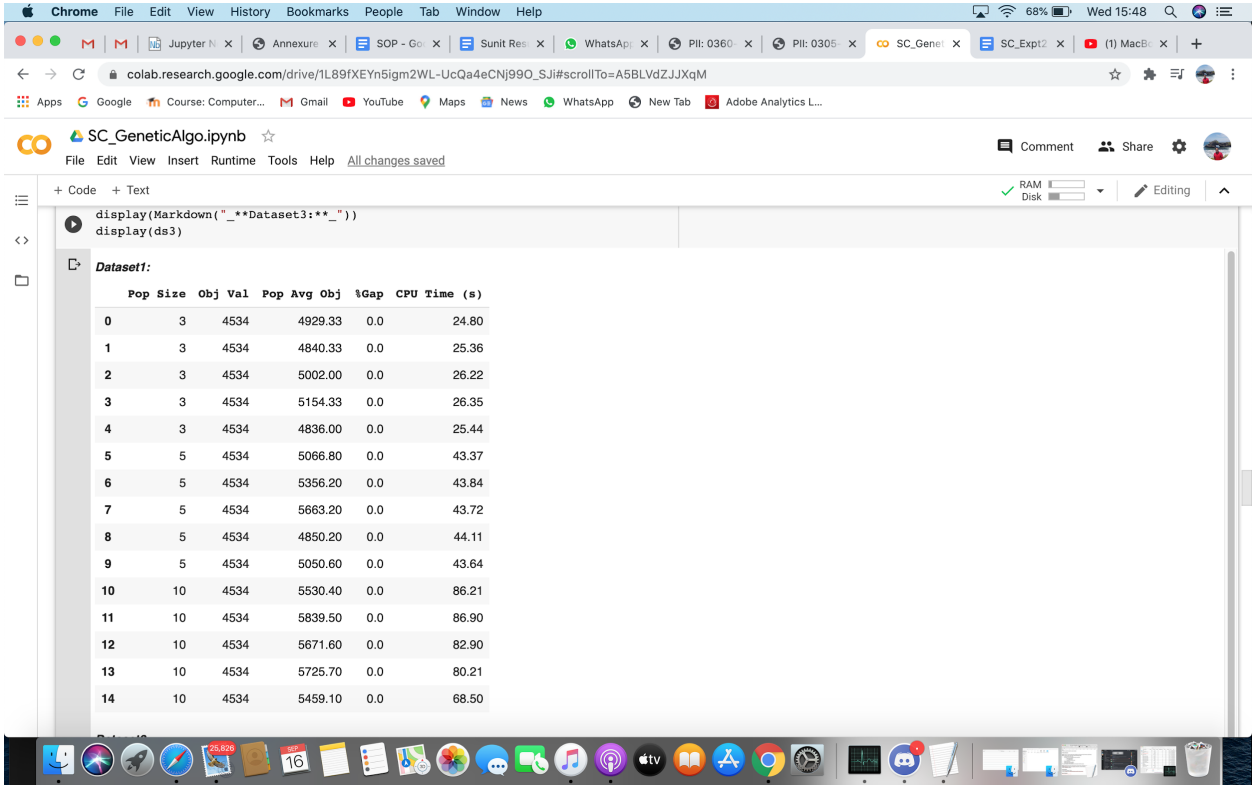
```
'%Gap':gap1, 'CPU Time (s)':cpuTime1 }
ds1 = pd.DataFrame(dfDict1)
ds1 = ds1[['Pop Size', 'Obj Val', 'Pop Avg Obj', '%Gap', 'CPU Time (s)']]

dfDict2 = {'Pop Size':popSize2, 'Obj Val':objVal2, 'Pop Avg Obj':avgObj2,
'%Gap':gap2, 'CPU Time (s)':cpuTime2 }
ds2 = pd.DataFrame(dfDict2)
ds2 = ds2[['Pop Size', 'Obj Val', 'Pop Avg Obj', '%Gap', 'CPU Time (s)']]

dfDict3 = {'Pop Size':popSize3, 'Obj Val':objVal3, 'Pop Avg Obj':avgObj3,
'%Gap':gap3, 'CPU Time (s)':cpuTime3 }
ds3 = pd.DataFrame(dfDict3)
ds3 = ds3[['Pop Size', 'Obj Val', 'Pop Avg Obj', '%Gap', 'CPU Time (s)']]

dfDict = {'Pop Size':popSize, 'Obj Val':objVal, 'Pop Avg Obj':avgObj,
'%Gap':gap, 'CPU Time (s)':cpuTime }
ds = pd.DataFrame(dfDict)
ds = ds[['Pop Size', 'Obj Val', 'Pop Avg Obj', '%Gap', 'CPU Time (s)']]

display(Markdown("_**Dataset1:**_"))
display(ds1)
print()

display(Markdown("_**Dataset2:**_"))
display(ds2)
print()

display(Markdown("_**Dataset3:**_"))
display(ds3)
# Number of population
Npop = 3
# Probability of crossover
Pc = 1.0
# Probability of mutation
Pm = 1.0
# Stopping number for generation
stopGeneration = 500
```

```
# Start Timer
t1 = time.clock()

# Creating the initial population
population = initialization(Npop)
s = []
# Run the algorithm for 'stopGeneration' times generation
for i in range(stopGeneration):
    # Selecting parents
    parents = selection(population)

    childs = []

    # Apply crossover
    for p in parents:
        r = random.random()
        if r < Pc:
            childs.append(crossover([population[p[0]], population[p[1]]]))
        else:
            if r < 0.5:
                childs.append(population[p[0]])
            else:
                childs.append(population[p[1]])

    # Apply mutation
    for c in childs:
        r = random.random()
        if r < Pm:
            c = mutation(c)

    # Update the population
    population = elitistUpdate(population, childs)

    print("Generation "+str(i)+":"+ str(population))
    # print(population)
                print("Time      of      the      Best      Solution:
```

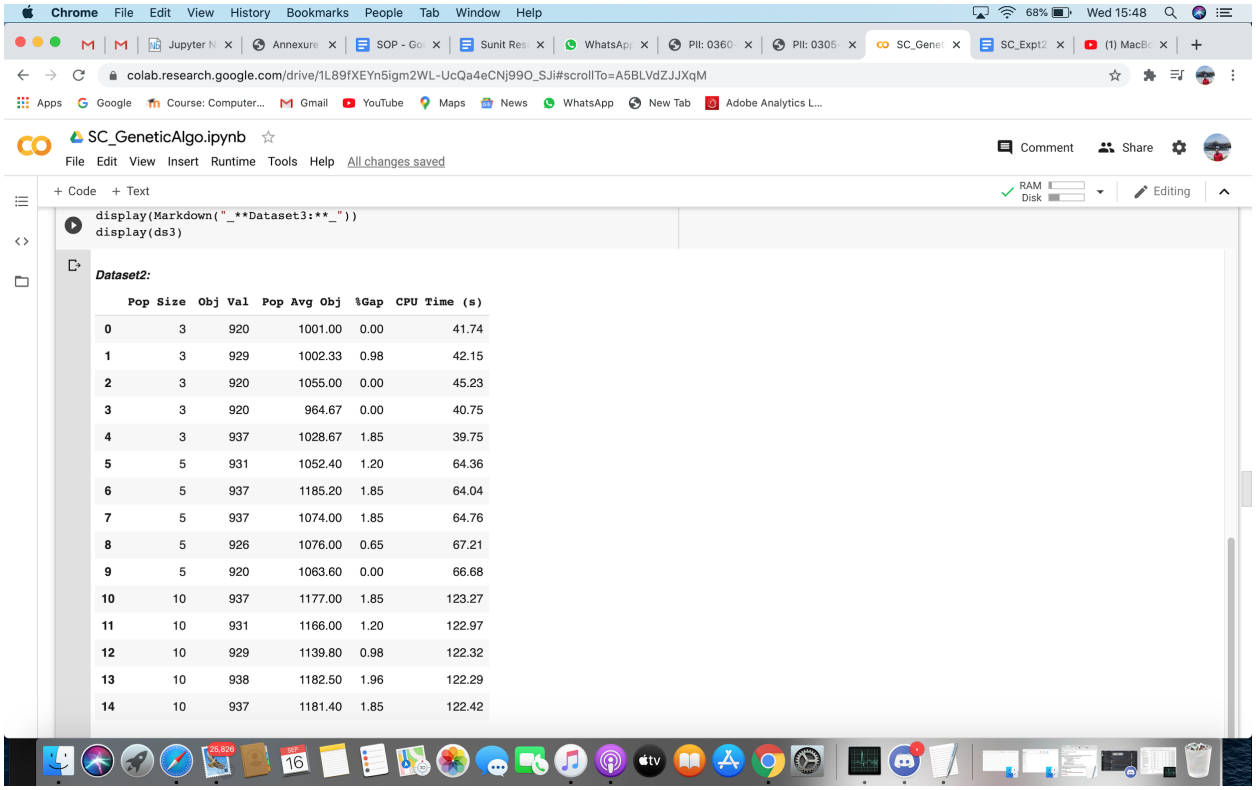
```
"+str(findBestSolution(population)[1]))
   s.append(findBestSolution(population)[1])

# Stop Timer
t2 = time.clock()

# Results Time

bestSol, bestObj, avgObj = findBestSolution(population)

print("Population:")
print(population)
print()

print("Solution:")
print(population[bestSol])
print()

print("Time(In secs):")
print(bestObj)
print()

print("Average Objective Value of Population:")
print("%.2f" %avgObj)
print()

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

print("CPU Time (s)")
timePassed = (t2-t1)
print("%.2f" %timePassed)

import matplotlib.pyplot as plt
```

```
def get_plot(s): #input all variables

    plt.plot(s, color='green', linewidth = 3)
    plt.xlabel('Number of Generations')
    plt.ylabel('Time (in seconds)')
    plt.title('Flow Shop Scheduling')
    plt.show()
get_plot(s)
```

Results:



	Pop Size	Obj Val	Pop Avg Obj	%Gap	CPU Time (s)
0	3	4534	4929.33	0.0	24.80
1	3	4534	4840.33	0.0	25.36
2	3	4534	5002.00	0.0	26.22
3	3	4534	5154.33	0.0	26.35
4	3	4534	4836.00	0.0	25.44
5	5	4534	5066.80	0.0	43.37
6	5	4534	5356.20	0.0	43.84
7	5	4534	5663.20	0.0	43.72
8	5	4534	4850.20	0.0	44.11
9	5	4534	5050.60	0.0	43.64
10	10	4534	5530.40	0.0	86.21
11	10	4534	5839.50	0.0	86.90
12	10	4534	5671.60	0.0	82.90
13	10	4534	5725.70	0.0	80.21
14	10	4534	5459.10	0.0	68.50



SC_GeneticAlgo.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

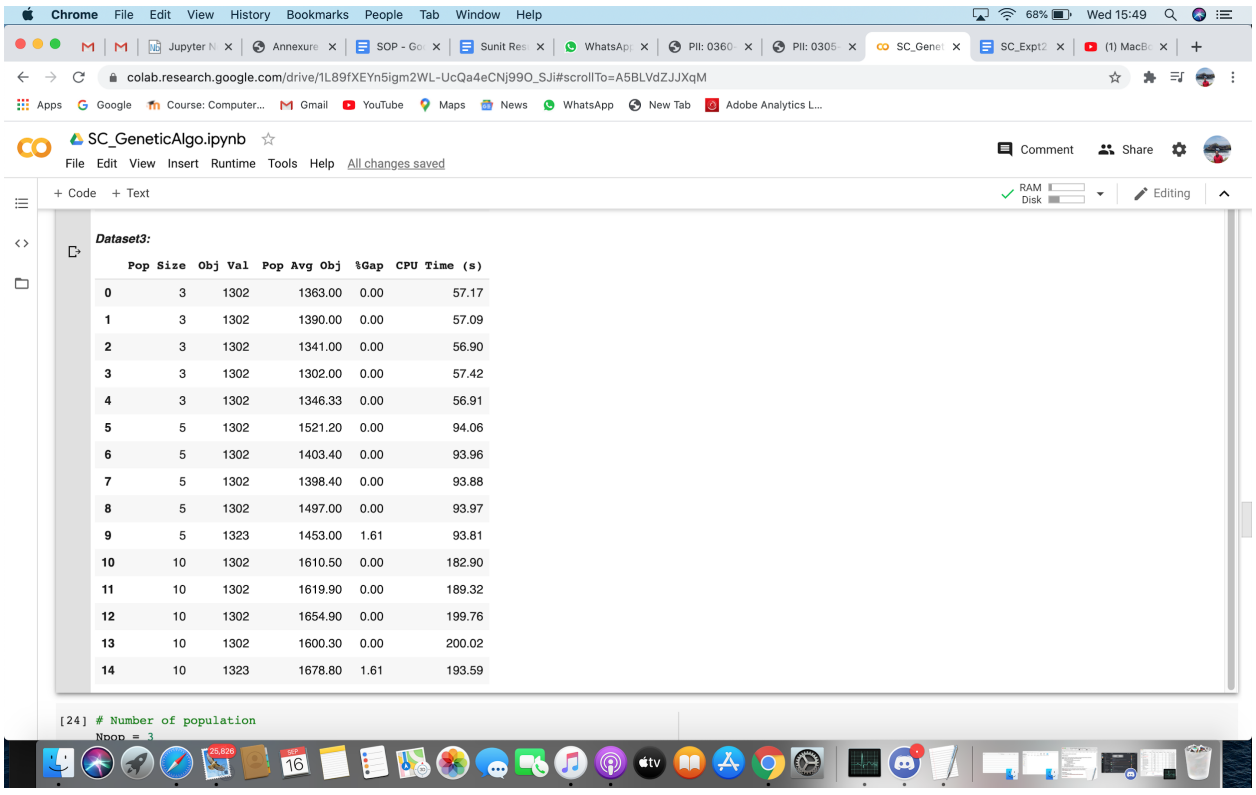
+ Code + Text

RAM Disk Editing

```
display(Markdown("_**Dataset3:**_"))
display(ds3)
```

Dataset2:

	Pop Size	Obj Val	Pop Avg Obj	%Gap	CPU Time (s)
0	3	920	1001.00	0.00	41.74
1	3	929	1002.33	0.98	42.15
2	3	920	1055.00	0.00	45.23
3	3	920	964.67	0.00	40.75
4	3	937	1028.67	1.85	39.75
5	5	931	1052.40	1.20	64.36
6	5	937	1185.20	1.85	64.04
7	5	937	1074.00	1.85	64.76
8	5	926	1076.00	0.65	67.21
9	5	920	1063.60	0.00	66.68
10	10	937	1177.00	1.85	123.27
11	10	931	1166.00	1.20	122.97
12	10	929	1139.80	0.98	122.32
13	10	938	1182.50	1.96	122.29
14	10	937	1181.40	1.85	122.42



SC_GeneticAlgo.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

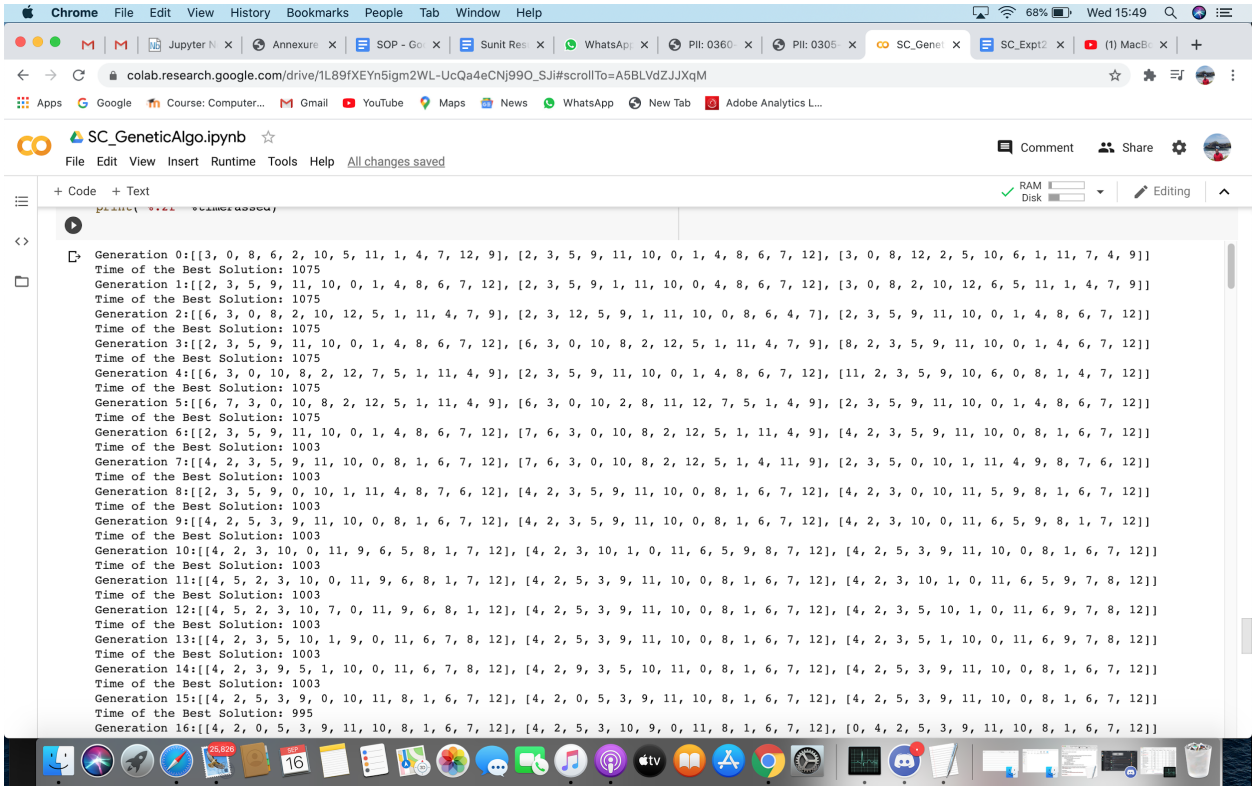
+ Code + Text

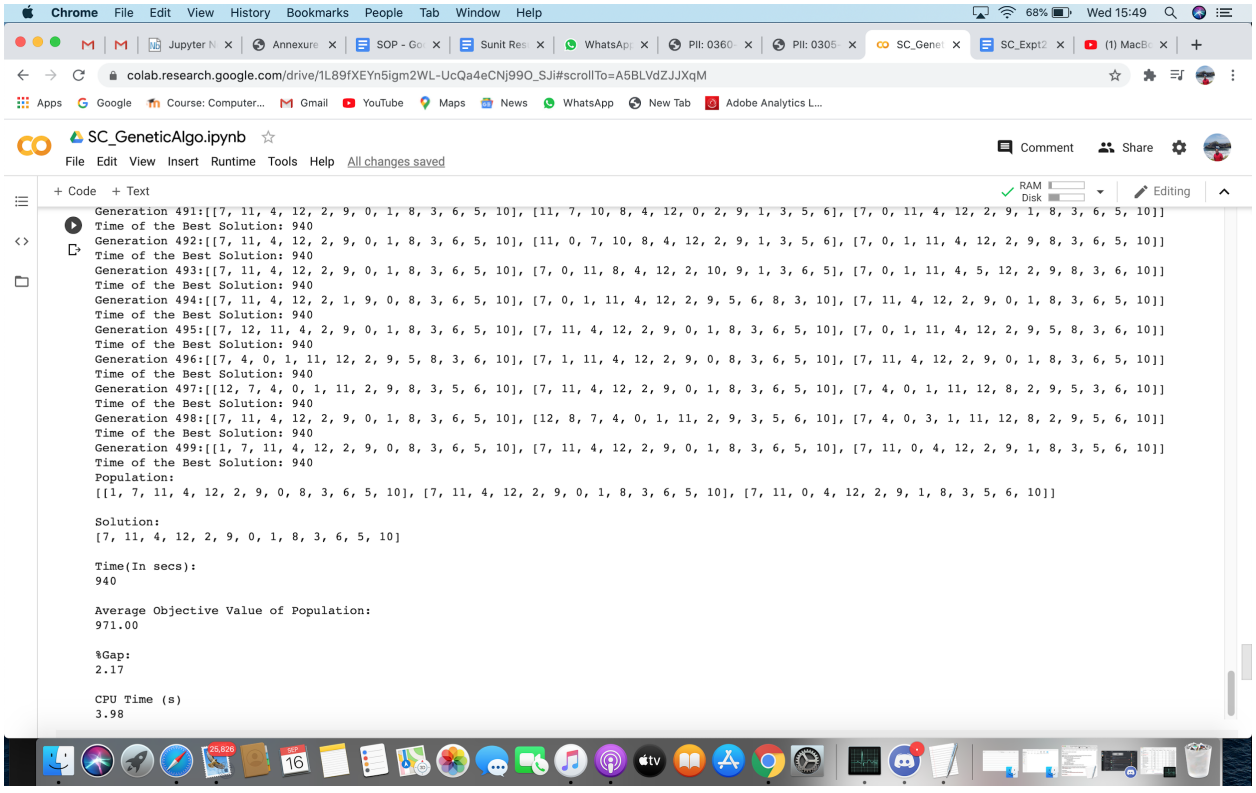
RAM Disk Editing

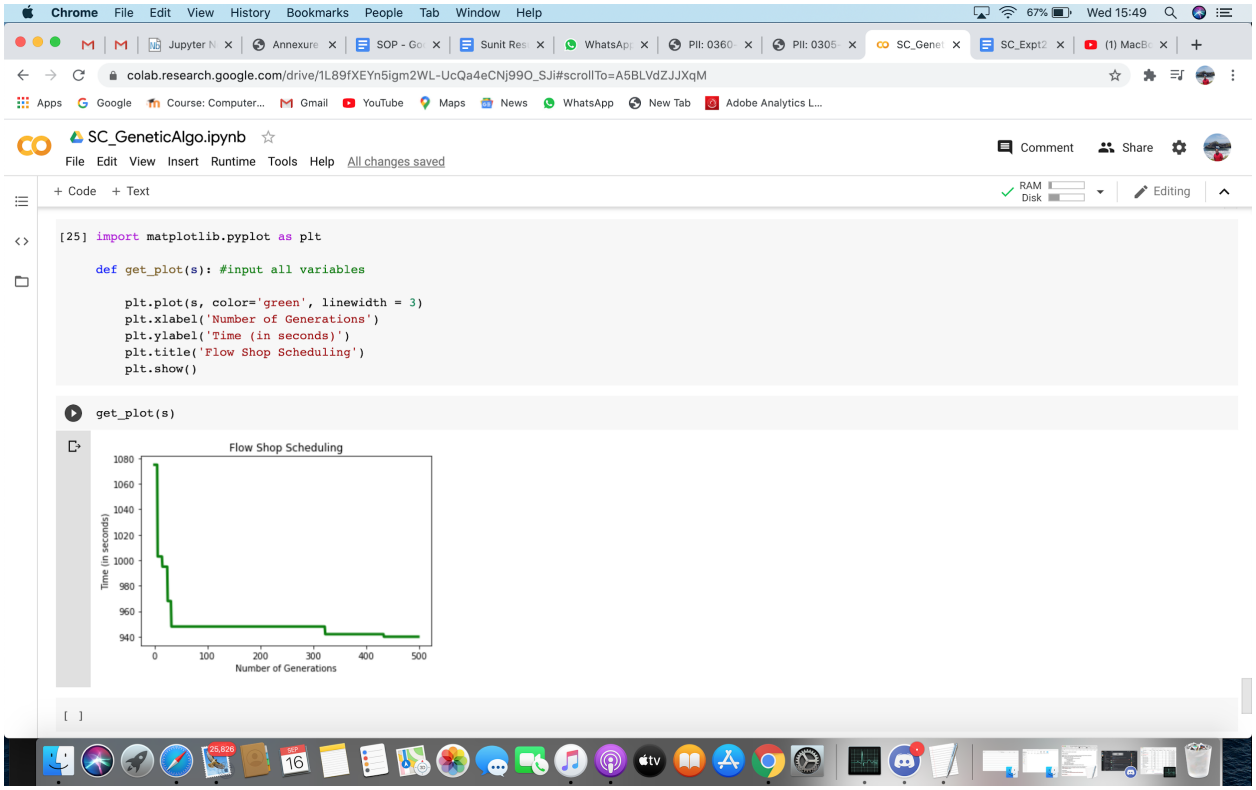
Dataset3:

	Pop Size	Obj Val	Pop Avg Obj	%Gap	CPU Time (s)
0	3	1302	1363.00	0.00	57.17
1	3	1302	1390.00	0.00	57.09
2	3	1302	1341.00	0.00	56.90
3	3	1302	1302.00	0.00	57.42
4	3	1302	1346.33	0.00	56.91
5	5	1302	1521.20	0.00	94.06
6	5	1302	1403.40	0.00	93.96
7	5	1302	1398.40	0.00	93.88
8	5	1302	1497.00	0.00	93.97
9	5	1323	1453.00	1.61	93.81
10	10	1302	1610.50	0.00	182.90
11	10	1302	1619.90	0.00	189.32
12	10	1302	1654.90	0.00	199.76
13	10	1302	1600.30	0.00	200.02
14	10	1323	1678.80	1.61	193.59

```
[24] # Number of population
     Npop = 3
```


SC_GeneticAlgo.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Editing
Generation 0:[[3, 0, 8, 6, 2, 10, 5, 11, 1, 4, 7, 12, 9], [2, 3, 5, 9, 11, 10, 0, 1, 4, 8, 6, 7, 12], [3, 0, 8, 12, 2, 5, 10, 6, 1, 11, 7, 4, 9]]
Time of the Best Solution: 1075
Generation 1:[[2, 3, 5, 9, 11, 10, 0, 1, 4, 8, 6, 7, 12], [2, 3, 5, 9, 1, 11, 10, 0, 4, 8, 6, 7, 12], [3, 0, 8, 2, 10, 12, 6, 5, 11, 1, 4, 7, 9]]
Time of the Best Solution: 1075
Generation 2:[[6, 3, 0, 8, 2, 10, 12, 5, 1, 11, 4, 7, 9], [2, 3, 12, 5, 9, 1, 11, 10, 0, 8, 6, 4, 7], [2, 3, 5, 9, 11, 10, 0, 1, 4, 8, 6, 7, 12]]
Time of the Best Solution: 1075
Generation 3:[[2, 3, 5, 9, 11, 10, 0, 1, 4, 8, 6, 7, 12], [6, 3, 0, 10, 8, 2, 12, 5, 1, 11, 4, 7, 9], [8, 2, 3, 5, 9, 11, 10, 0, 1, 4, 6, 7, 12]]
Time of the Best Solution: 1075
Generation 4:[[6, 3, 0, 10, 8, 2, 12, 7, 5, 1, 11, 4, 9], [2, 3, 5, 9, 11, 10, 0, 1, 4, 8, 6, 7, 12], [11, 2, 3, 5, 9, 10, 6, 0, 8, 1, 4, 7, 12]]
Time of the Best Solution: 1075
Generation 5:[[6, 7, 3, 0, 10, 8, 2, 12, 5, 1, 11, 4, 9], [6, 3, 0, 10, 2, 8, 11, 12, 7, 5, 1, 4, 9], [2, 3, 5, 9, 11, 10, 0, 1, 4, 8, 6, 7, 12]]
Time of the Best Solution: 1075
Generation 6:[[2, 3, 5, 9, 11, 10, 0, 1, 4, 8, 6, 7, 12], [7, 6, 3, 0, 10, 8, 2, 12, 5, 1, 11, 4, 9], [4, 2, 3, 5, 9, 11, 10, 0, 8, 1, 6, 7, 12]]
Time of the Best Solution: 1003
Generation 7:[[4, 2, 3, 5, 9, 11, 10, 0, 8, 1, 6, 7, 12], [7, 6, 3, 0, 10, 8, 2, 12, 5, 1, 4, 11, 9], [2, 3, 5, 0, 10, 1, 11, 4, 9, 8, 7, 6, 12]]
Time of the Best Solution: 1003
Generation 8:[[2, 3, 5, 9, 0, 10, 1, 11, 4, 8, 7, 6, 12], [4, 2, 3, 5, 9, 11, 10, 0, 8, 1, 6, 7, 12], [4, 2, 3, 0, 10, 11, 5, 9, 8, 1, 6, 7, 12]]
Time of the Best Solution: 1003
Generation 9:[[4, 2, 5, 3, 9, 11, 10, 0, 8, 1, 6, 7, 12], [4, 2, 3, 5, 9, 11, 10, 0, 8, 1, 6, 7, 12], [4, 2, 3, 10, 0, 11, 6, 5, 9, 8, 1, 7, 12]]
Time of the Best Solution: 1003
Generation 10:[[4, 2, 3, 10, 0, 11, 9, 6, 5, 8, 1, 7, 12], [4, 2, 3, 10, 1, 0, 11, 6, 5, 9, 8, 7, 12], [4, 2, 5, 3, 9, 11, 10, 0, 8, 1, 6, 7, 12]]
Time of the Best Solution: 1003
Generation 11:[[4, 5, 2, 3, 10, 0, 11, 9, 6, 8, 1, 7, 12], [4, 2, 5, 3, 9, 11, 10, 0, 8, 1, 6, 7, 12], [4, 2, 3, 10, 1, 0, 11, 6, 5, 9, 7, 8, 12]]
Time of the Best Solution: 1003
Generation 12:[[4, 5, 2, 3, 10, 7, 0, 11, 9, 6, 8, 1, 12], [4, 2, 5, 3, 9, 11, 10, 0, 8, 1, 6, 7, 12], [4, 2, 3, 5, 10, 1, 0, 11, 6, 9, 7, 8, 12]]
Time of the Best Solution: 1003
Generation 13:[[4, 2, 3, 5, 10, 1, 9, 0, 11, 6, 7, 8, 12], [4, 2, 5, 3, 9, 11, 10, 0, 8, 1, 6, 7, 12], [4, 2, 3, 5, 1, 10, 0, 11, 6, 9, 7, 8, 12]]
Time of the Best Solution: 1003
Generation 14:[[4, 2, 3, 9, 5, 1, 10, 0, 11, 6, 7, 8, 12], [4, 2, 9, 3, 5, 10, 11, 0, 8, 1, 6, 7, 12], [4, 2, 5, 3, 9, 11, 10, 0, 8, 1, 6, 7, 12]]
Time of the Best Solution: 1003
Generation 15:[[4, 2, 5, 3, 9, 0, 10, 11, 8, 1, 6, 7, 12], [4, 2, 0, 5, 3, 9, 11, 10, 8, 1, 6, 7, 12], [4, 2, 5, 3, 9, 11, 10, 0, 8, 1, 6, 7, 12]]
Time of the Best Solution: 995
Generation 16:[[4, 2, 0, 5, 3, 9, 11, 10, 8, 1, 6, 7, 12], [4, 2, 5, 3, 10, 9, 0, 11, 8, 1, 6, 7, 12], [0, 4, 2, 5, 3, 9, 11, 10, 8, 1, 6, 7, 12]]


SC_GeneticAlgo.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Editing
Generation 491:[[7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10], [11, 7, 10, 8, 4, 12, 0, 2, 9, 1, 3, 5, 6], [7, 0, 11, 4, 12, 2, 9, 1, 8, 3, 6, 5, 10]]
Time of the Best Solution: 940
Generation 492:[[7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10], [11, 0, 7, 10, 8, 4, 12, 2, 9, 1, 3, 5, 6], [7, 0, 1, 11, 4, 12, 2, 9, 8, 3, 6, 5, 10]]
Time of the Best Solution: 940
Generation 493:[[7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10], [7, 0, 11, 8, 4, 12, 2, 10, 9, 1, 3, 6, 5], [7, 0, 1, 11, 4, 5, 12, 2, 9, 8, 3, 6, 10]]
Time of the Best Solution: 940
Generation 494:[[7, 11, 4, 12, 2, 1, 9, 0, 8, 3, 6, 5, 10], [7, 0, 1, 11, 4, 12, 2, 9, 5, 6, 8, 3, 10], [7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10]]
Time of the Best Solution: 940
Generation 495:[[7, 12, 11, 4, 2, 9, 0, 1, 8, 3, 6, 5, 10], [7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10], [7, 0, 1, 11, 4, 12, 2, 9, 5, 8, 3, 6, 10]]
Time of the Best Solution: 940
Generation 496:[[7, 4, 0, 1, 11, 12, 2, 9, 5, 8, 3, 6, 10], [7, 1, 11, 4, 12, 2, 9, 0, 8, 3, 6, 5, 10], [7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10]]
Time of the Best Solution: 940
Generation 497:[[12, 7, 4, 0, 1, 11, 2, 9, 8, 3, 5, 6, 10], [7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10], [7, 4, 0, 1, 11, 12, 8, 2, 9, 5, 3, 6, 10]]
Time of the Best Solution: 940
Generation 498:[[7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10], [12, 8, 7, 4, 0, 1, 11, 2, 9, 3, 5, 6, 10], [7, 4, 0, 3, 1, 11, 12, 8, 2, 9, 5, 6, 10]]
Time of the Best Solution: 940
Generation 499:[[1, 7, 11, 4, 12, 2, 9, 0, 8, 3, 6, 5, 10], [7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10], [7, 11, 0, 4, 12, 2, 9, 1, 8, 3, 5, 6, 10]]
Time of the Best Solution: 940
Population:
[[1, 7, 11, 4, 12, 2, 9, 0, 8, 3, 6, 5, 10], [7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10], [7, 11, 0, 4, 12, 2, 9, 1, 8, 3, 5, 6, 10]]
Solution:
[7, 11, 4, 12, 2, 9, 0, 1, 8, 3, 6, 5, 10]
Time(In secs):
940
Average Objective Value of Population:
971.00
%Gap:
2.17
CPU Time (s)
3.98



Conclusion:

Successfully implemented Flow Shop Scheduling using the genetic algorithm. The type of crossover and mutation used in this code are two point crossover and shift-change mutation which provide the best results in solving this problem.