Programming in Python Coursework

Introduction

The flow shop problem is an optimization topic crucial for many production companies who wish to maximize efficiency on numerous jobs across numerous machines. In this short report, we compare the results from multiple algorithms to provide a solution for BWM's flow shop problem. We will review all of the companies' five different instances, each with 8-20 jobs and 5-10 machines using three different algorithms: Random algorithm, Heuristic algorithm and the Genetic algorithm.

Random Selection Algorithm

Initially, we executed a Random Search Algorithm, which chooses a sequence at random (typically in the form of a pseudo-random number generator) (Zabinsky., 2009). The sequence is replaced each time when a sequence with a lower makespan is found. Random search algorithms are relatively easy to implement. It is important to recognize that as the algorithm is random, the output may be different each time. Hence, the algorithm is not the most consistent method. We further generated 30 best time spans from 30 random seeds in hope of seeing how varied the results would be. Table 1 shows the results of the 30 sequences for each car and the distribution of them. As the number of random searches increases, we expect to see lower minimum time span. However, it is unrealistic to run the algorithm infinitely considering both the constraint of time and result saturation. We chose to perform 1000*n permutations of the algorithm for each random seed (where n is the number of jobs for each different car).

From Table 1, we can see that the program time cost is positively correlated to both the number of jobs of a car and the number of machines the jobs have to pass through (See Appendix 1 for properties of each instance). For example, both car 3 and car 4 have the same number of jobs 20, whereas the number of machines for car 4 is twice as big as that of car 3; program time indicates, it took longer to complete a Random Search for car 4 (191.62 seconds) than car 3 (96.58 seconds).

The makespan results from Random Search, for example, for car 2, range from 8505.00 to 8773.00 with a standard deviation of 86.99. The results are highly variable this suggests it is highly unlikely to find "best" job sequence by performing Random Search Algorithm for just one random seed.

Car Type	Program Time cost (seconds 2.d.p)	Minimum Makespan (2.d.p)	Maximum Makespan (2.d.p)	Mean Makespan (2.d.p)	Standard Deviation (2.d.p)
1	29.34	7038.00	7057.00	7039.93	5.69
2	34.43	8505.00	8773.00	8667.26	86.99
3	96.58	1276.00	1303.00	1290.97	7.44
4	191.62	1659.00	1700.00	1679.63	10.21
5	426.12	2304.00	2376.00	2349.90	15.88

Table 1. Results of Random Search for all 5 cars

NEH Algorithm

Unlike Random Search Algorithm, NEH Algorithm does not involve randomness. Details of the algorithm can be found in Nawaz *et al.,* (1982); the basic principle of the algorithm is that it gives priority to the jobs with longer total processing times on all machines.

Table 2 shows the results we got by performing NEH Algorithm on all five cars. Comparing the general program time cost of the two algorithms, NEH appears to be much more efficient than Random Search. However, the minimum makespans (MMS) for car 2 and car 3 resulting from NEH are greater than those MMS resulting from Random Search. Nevertheless, the significant improvement in efficiency needs to be highlighted.

Car Type	Program Time cost (seconds 2.d.p)	Minimum Makespan (2.d.p)
1	0.06	7038.00
2	0.04	8773.00
3	0.16	1281.00
4	0.30	1626.00
5	1.02	2185.00

Table 2. Results of NEH for all 5 Cars

Genetic Algorithm (GA)

The GA algorithm has an aspect of randomization. Therefore, the results we get from performing the algorithm with different seeds could be slightly different. We implemented 30 different random seeds to replicate the possible variations of calculating the MMS. The algorithm and the parameters ($P = 30, P_c = 1, P_m = 0.8, D = 0.95$) we used were taken from Reeves., (1995). Note that instead of using the version of a 2-point crossover presented in the paper, we performed the algorithm with the version presented in Murata *et al*., (1996) as the version in Reeves., (1995) is noted to excessively disrupt and prolongs the convergence of solutions. Figure 1 illustrates the version we used in our implementation.

The program time for the genetic algorithm is not recorded as in Table 1 and 2 as the codes were run on different computers. It is also essential to note that we did 1000n iterations as we had a time constraint and the GA has a considerably longer run time.

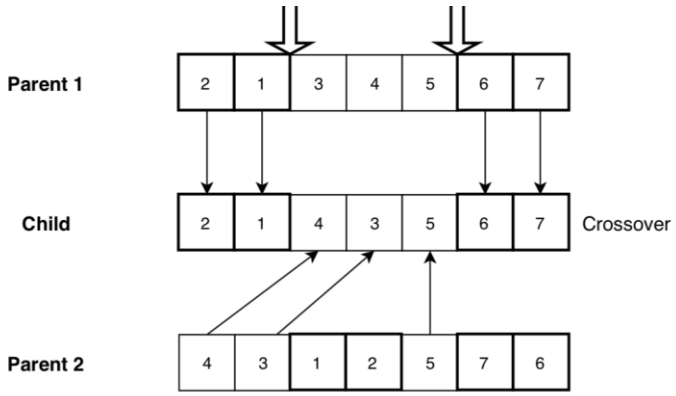


Figure 1. Crossover operation performed Murata *et al*., (1996).

Car Type	Minimum Makespan (2.d.p)	Maximum Makespan (2.d.p)	Mean Makespan (2.d.p)	Standard Deviation (2.d.p)
1	7038.00	7038.00	7038.00	0.00
2	8570.00	8773.00	8765.60	36.63
3	1266.00	1281.00	1279.00	3.74
4	1626.00	1626.00	1626.00	0.00
5	2185.00	2185.00	2185.00	0.00

Table 3. Results of GA for all 5 Cars

Conclusion of the three methods

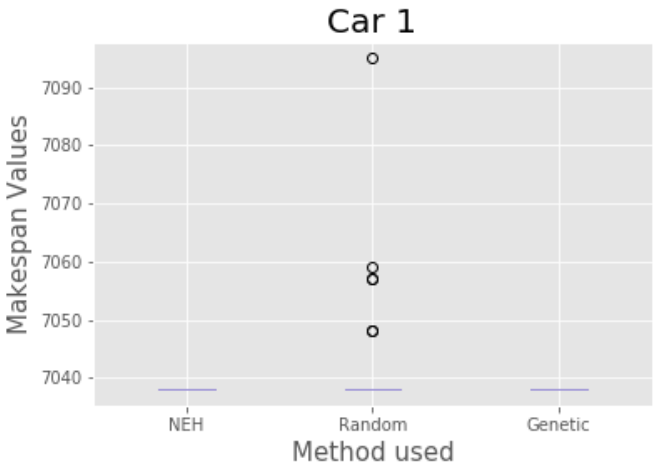


Figure 2. Car 1's makespans for 3 different algorithms

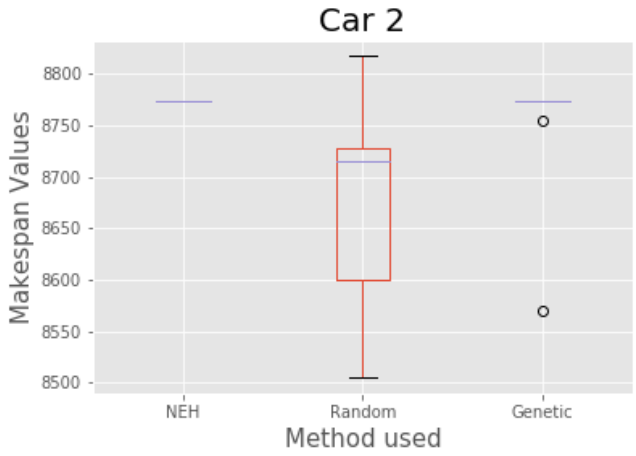


Figure 3. Car 2's makespans for 3 different algorithms

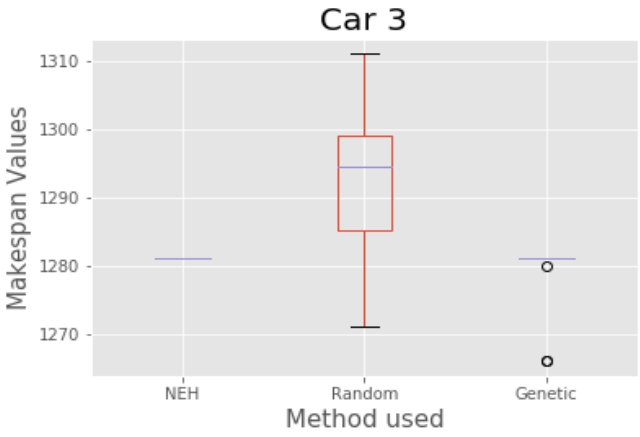


Figure 4. Car 3's makespans for 3 different algorithms

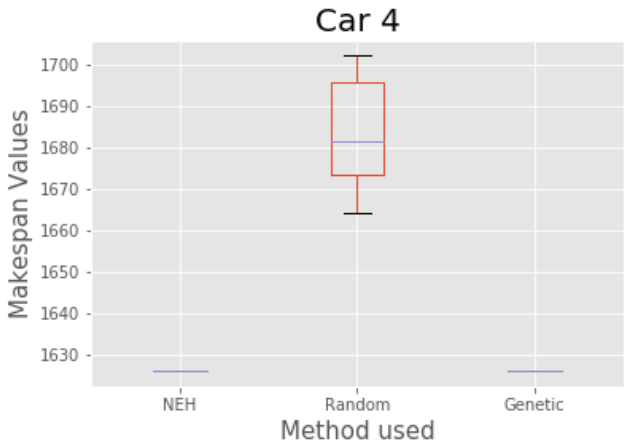


Figure 5. Car 4's makespans for 3 different algorithms

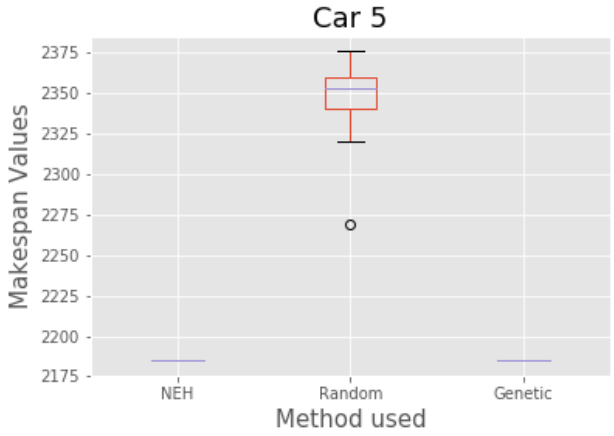


Figure 6. Car 5's makespans for 3 different algorithms

We can see from Table 5 that different algorithms performed better on different car types. With the exception of Car 4, GA produced the optimal makespans for all cars. It is also shown in table 3 and the results boxplots for each car how little variation present within GA results. However, there are a few outliers present in Figures 4 & 5 but these outliers only give lower makespan values.

Although Random search (RS) has been said to be inferior to GA (Murata *et al.*,1996), it performed competitively well on Car 1 and especially Car 2. However, both cars have considerably fewer jobs than the other cars (Appendix 1). For Car 2, more than 95% of the makespans produced by Random Search were less than those produced by NEH and GA (Figure 4). Whereas for Car 1 there was very little variation in results, the standard deviation of 5.69 is mainly the product of the 4 outliers that can be identified in Figure 3. It is noticeable that RS presented a job sequence with lower makespan times of those produced of GA and NEH but as over 95% of the RS's results were higher than those of NEH and GA we have not recommended it. Our reasoning being that those low results are unlikely to occur when if you run the algorithm once (>5%). NEH also created a makespan that was as low as the other two algorithms for Car 1 and Car 2.

Car Type	Recommended Algorithm	Total minimum makespan (2d.p)
1	Random Search NEH Genetic Algorithm	7038.00
2	Random Search Genetic Algorithm	8505.00
3	Genetic Algorithm	1262.00
4	NEH Genetic Algorithm	1626.00
5	Genetic Algorithm	2167.00

Table 5. Summary table of which algorithm presented the best makespan time for each car

When you wish to choose the best algorithm to use in the future it would be wise to first state the scope of the problem and what factor is limited in the scenario; program time or the minimum makespan time.

Changing parameters in the Genetic Algorithm (GA)

To obtain a more comprehensive assessment of GA, we decided to vary the algorithm's parameters and see if the change has an impact on either the algorithm's efficiency or the results.

Figure 7 is a collection of box plots showing the distribution of all resulting makespans for Car 1 from Random Search, NEH, GA and GA implemented with various parameters. The results of the GA implemented with different crossover probability can be noticed immediately. Similar phenomena are also found when we change the value of P_c for the other cars as can be seen in Appendix (2,3,4,5). However, these results are considerably weaker than those calculated initially with $P_c = 1$. Thus, we suggest BWM to set $P_c = 1$when using Genetic Algorithm to get the optimal(lowest) makespan.

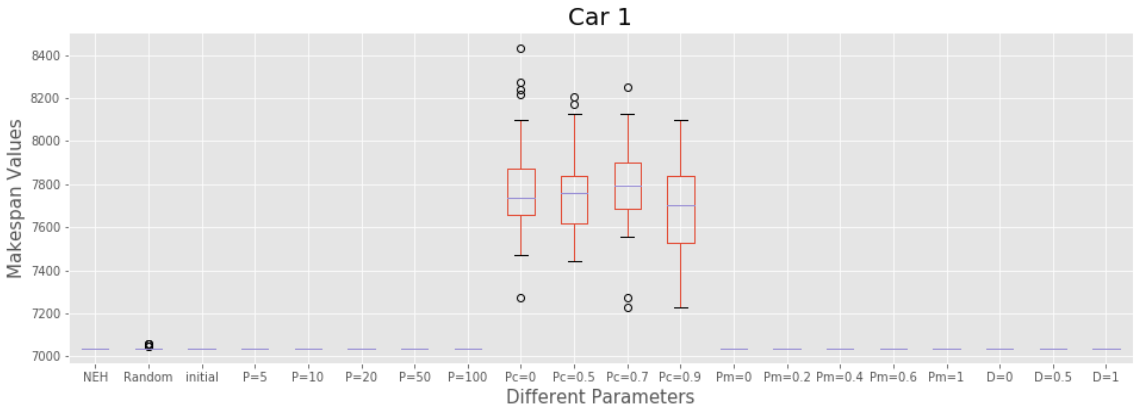


Figure 7. Car 1's results for all three algorithms and when parameters are changed for GA

Summary

In conclusion, we recommend that BWM use the genetic algorithm (GA) to schedule their job order in the future. The reasoning behind our suggestions is that:

- Random Search is simple but computationally taxing. The results are varied, and the quality is only comparable to GA when there are few jobs.
- NEH only performs as well as GA for two cars.
- GA has been shown to produce the best results consistency, with little variation. This is despite differences in job and machine numbers.

Conditions of using GA at the optimal level:

- When using GA parameter in Pc setting is essential and should always = 1 or the results will be significantly worse.
- Other parameters do not make as of significant difference as pc however suggested settings are …. P= 30, P_m=0.8 and D=0.95.
- GA must be run numerous times in order to ensure the sequence obtained is not an outlier, which does not represent the optimal makespan.
- BWM must have the time/hardware capacity to run the algorithm. If the instance specified has more complex (has a high number of jobs).

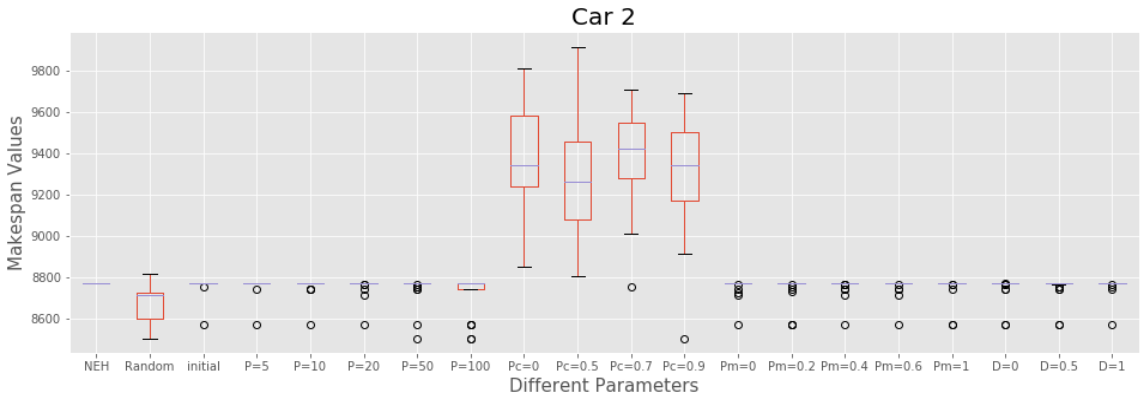
Bibliography

1. Murata,T., Ishibuchi,H., Tanaka,H., (1996) ‘Genetic Algorithms for Flowshop Scheduling Problems’ . *Computers & Industrial Engineering*, 30(4):1061-1071, https://doi.org/10.1016/0360-8352(96)00053-8
2. Nawaz,M., Enscore Jr, E., Ham,I., (1983). ‘A heuristic algorithm for the *m*-machine, *n*-job flow-shop sequencing problem’ , *Omega*, 11(1):91-95, https://doi.org/10.1016/0305-0483(83)90088-9
3. Reeves,C.W., (1995). ‘A genetic algorithm for flowshop sequencing’ , *Computers & Operations Research*, 22(1):5-13, https://doi.org/10.1016/0305-0548(93)E0014-K
4. Zabinsky, Z. B., (2011) “Random Search Algorithms.” *Wiley Encyclopedia of Operations Research and Management Science*, doi:10.1002/9780470400531.eorms0704.

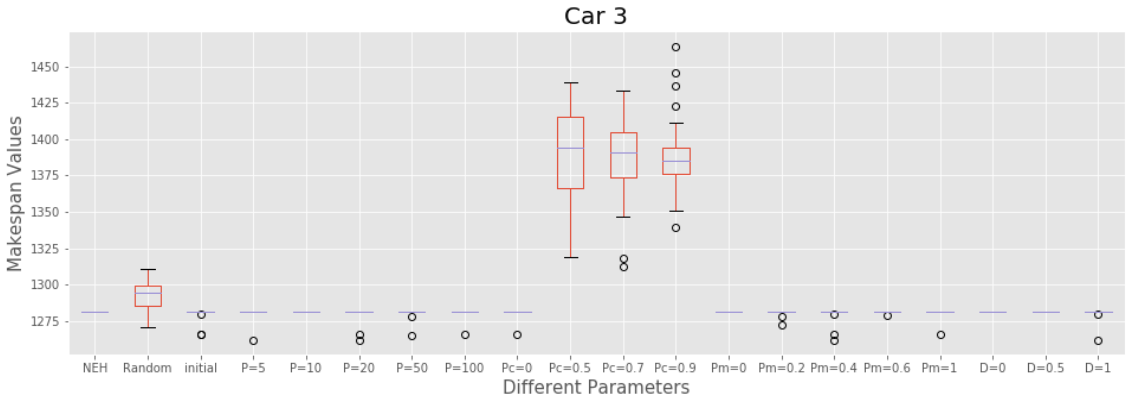
Appendix

Car	Number of jobs	Number of machines
1	11	5
2	8	9
3	20	5
4	20	10
5	30	10

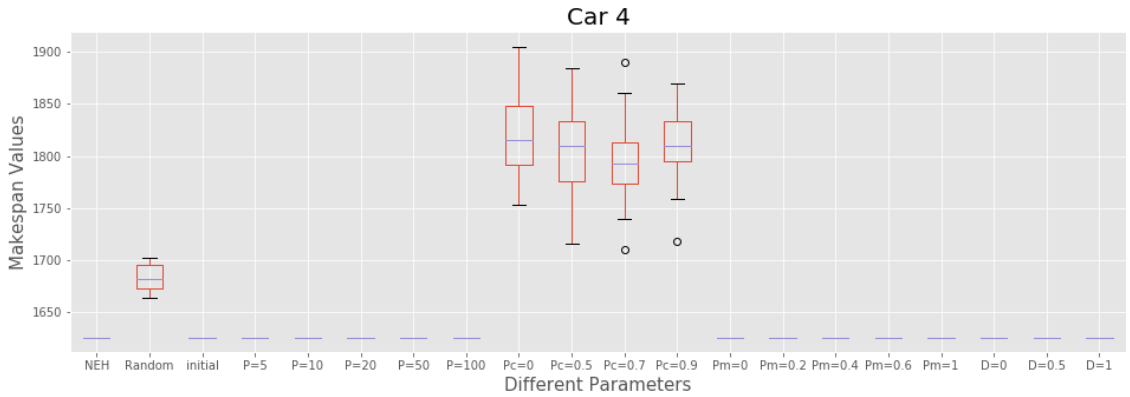
Appendix 1- Table showing properties (number of jobs and number of machines) for each car.



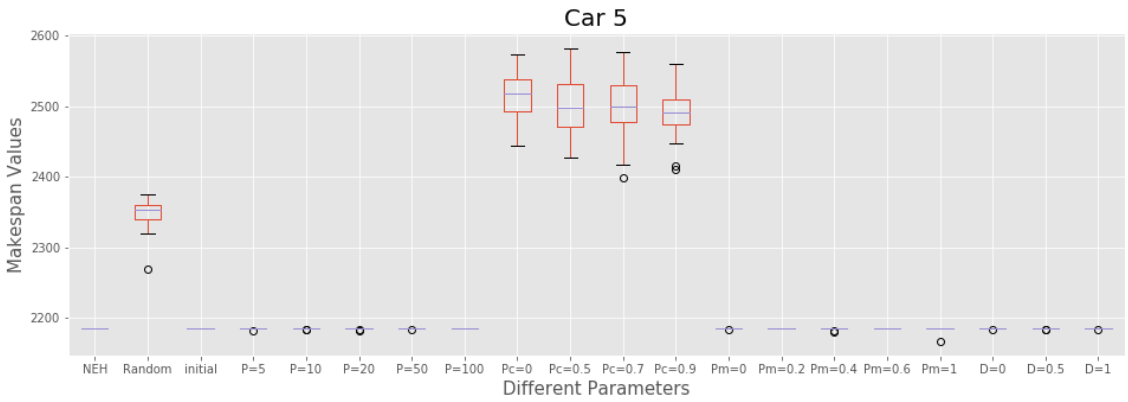
Appendix 2 - Car 2 results for all three algorithms and when parameters are changed for GA



Appendix 3 - Car 3 results for all three algorithms and when parameters are changed for GA



Appendix 4 - Car 4 results for all three algorithms and when parameters are changed for GA



Appendix 5 - Car 5 results for all three algorithms and when parameters are changed for GA

Appendix 6 - Codes for 3 algorithms and visualization

```
#########################################################
# algorithm for random search
#########################################################

# Load the libraries needed
import time
import numpy as np
import pandas as pd
from copy import deepcopy
import random

#read 5 datasets
car1 = pd.read_csv('car1.csv')
car1 = np.array(car1)
car2 = pd.read_csv('car2.csv')
car2 = np.array(car2)
car3 = pd.read_csv('car3.csv')
car3 = np.array(car3)
car4 = pd.read_csv('car4.csv')
car4 = np.array(car4)
car5 = pd.read_csv('car5.csv')
car5 = np.array(car5)

time_start=time.time() #recode the program start time

#calculate the makespan
#p_ij=dataset ,nbm=machine number, my_seq=current job sequence

def makespan(current_seq, p_ij, nbm):
    c_ij = np.zeros((nbm, len(current_seq) + 1))
    for j in range(1, len(current_seq) + 1):
        c_ij[0][j] = c_ij[0][j - 1] + p_ij[0][current_seq[j - 1]]

    for i in range(1, nbm):
        for j in range(1, len(current_seq) + 1):
            c_ij[i][j] = max(c_ij[i - 1][j], c_ij[i][j - 1]) + p_ij[i][current_seq[j - 1]]
    return current_seq,c_ij

#Apply the random search with 1000*n solution evaluation and calculate their makespan
def random_search(car_num):
    p_ij=car_num
    nbm=len(p_ij)
    nbj=len(p_ij[0])
    #original job sequence
    seq_origin=list(range(len(p_ij[0])))
    current_seq = []
    result_time=[]
    min_seq=[]
    loop_result={}
    for i in range(1000*nbj):
        seq = deepcopy(seq_origin)
        random.shuffle(seq)
        current_seq,c=makespan(seq,p_ij,nbm)
        mp=c[len(c)-1][len(c[0])-1]
        result_time.append(mp)
        loop_result[i]={'seq':current_seq,'makespan_values':mp}
    min_mp=min(result_time)
    for loop_time in loop_result.values():
        if loop_time['makespan_values']==min_mp:
```

```
            min_seq.append(loop_time['seq'])
    return min_seq,min_mp

#best_time has 30 makespan values , best_seq 30 seqs
def main(n,car_num):
    best_time=[]
    best_seq=[]
    seed_result={}
    for times in range(n):
        np.random.seed(n*times)
        seq,min_mp = random_search(car_num)
#         best_seq.append(seq)
        best_time.append(min_mp)
        seed_result[times]={'seq':seq,'makespan_values':min_mp}
        #print 30 seeds best sequence and makespan for each 1000*n solution evaluations
        print('Seed {0} best sequence:{1} makespan values:{2}\n'.format(times+1,seq,min_mp))
    print('Makespan table:\n mininum:{0}, maximum:{1}, mean:{2}, standard deviation:{3}'.\
          format(np.min(best_time),np.max(best_time),np.mean(best_time),np.std(best_time)))
    for times in seed_result.values():
        if times['makespan_values']==np.min(best_time):
            best_seq.extend(times['seq'])

# remove the same best_seq,not-repeating
    NP_bestseq=[]
    [NP_bestseq.append(i) for i in best_seq if not i in NP_bestseq]
    NP_bestseq.sort()
    return best_time,NP_bestseq

# set 30 seeds
def min_mp(car_num):
    n = 30
    best_time=main(n,car_num)
    return best_time

#chose the car number here
best_time=min_mp(car2)

#recode the program running time
time_end=time.time()
print(' Program Time Cost:',time_end-time_start,'s')
```

```
#########################################################
# algorithm for neh
#########################################################

# Load the libraries needed
import pandas as pd
import numpy as np

# read the 5 datasets
car1 = pd.read_csv('car1.csv')
car1 = np.array(car1)
car2 = pd.read_csv('car2.csv')
car2 = np.array(car2)
car3 = pd.read_csv('car3.csv')
car3 = np.array(car3)
car4 = pd.read_csv('car4.csv')
car4 = np.array(car4)
car5 = pd.read_csv('car5.csv')
car5 = np.array(car5)

#chose the car number
p_ij=car2

nbm=len(p_ij)
nbj=len(p_ij[0])

#calculate makespan
def makespan_neh(current_seq, p_ij, nbm):
    c_ij = np.zeros((nbm, len(current_seq) + 1))
    for j in range(1, len(current_seq) + 1):
        c_ij[0][j] = c_ij[0][j - 1] + p_ij[0][current_seq[j - 1]]

    for i in range(1, nbm):
        for j in range(1, len(current_seq) + 1):
            c_ij[i][j] = max(c_ij[i - 1][j], c_ij[i][j - 1]) + p_ij[i][current_seq[j - 1]]
    return c_ij[nbm - 1][len(current_seq)]

#calculate the job's total processing time in all machines
def sum_processing_time(index_job, data, nb_machines):
    sum_p = 0
    for i in range(nb_machines):
        sum_p += data[i][index_job]
    return sum_p


#Sort the current sequence by job's total processing time(descending)
def order_neh(data, nb_machines, nb_jobs):
    my_seq = []
    for j in range(nb_jobs):
        my_seq.append(j)
    return sorted(my_seq,key=lambda x:sum_processing_time(x, data, nb_machines),reverse=True)

#insert the new job and obtain the new sequence
def insertion(sequence, index_position, value):
    new_seq = sequence[:]
    new_seq.insert(index_position, value)
    return new_seq

#run the neh
```

```python
#calculate the new makespan after insert the new job
#Compare the makespan of each sequence, retaining the best sequence and makespan
def neh(data, nb_machines, nb_jobs):
    order_seq = order_neh(data, nb_machines, nb_jobs)
    seq_current = [order_seq[0]]
    #obtain partial and complete sequences [n(n+1)/2]-1 times
    for i in range(1, nb_jobs):
        min_cmax = float("inf")
        for j in range(0, i + 1):
            tmp_seq = insertion(seq_current, j, order_seq[i])
            cmax_tmp = makespan_neh(tmp_seq, data, nb_machines)
            print(tmp_seq, cmax_tmp)
            if min_cmax > cmax_tmp:
                best_seq = tmp_seq
                min_cmax = cmax_tmp
        seq_current = best_seq
    return seq_current,makespan_neh(seq_current, data, nb_machines)

# print the NEH seq and it's makespan
seq, cmax = neh(p_ij, nbm, nbj)
print('Number of Machines:{0},Number of Jobs:{1}'.format(nbm,nbj))
print("NEH sequence:", seq)
print("Makespan:",cmax)
```

```
#########################################################
# algorithm for GA
#(Algo4:2-point crossover(C1) with shift mutation(SM))
#########################################################

import numpy as np
import pandas as pd
import random

# read the 5 datasets
car1 = pd.read_csv('car1.csv')
car1 = np.array(car1)
car2 = pd.read_csv('car2.csv')
car2 = np.array(car2)
car3 = pd.read_csv('car3.csv')
car3 = np.array(car3)
car4 = pd.read_csv('car4.csv')
car4 = np.array(car4)
car5 = pd.read_csv('car5.csv')
car5 = np.array(car5)

#choose the car number
p_ij=car2

#define the NEH_SEQ
NEH_seq=[4, 7, 5, 6, 2, 0, 3, 1]

#5 cars NEH_seq
#car1 [7, 0, 4, 8, 2, 10, 3, 6, 5, 1, 9]
#car2 [4, 7, 5, 6, 2, 0, 3, 1]
#car3 [18, 7, 11, 15, 19, 4, 0, 9, 12, 2, 1, 17, 8, 6, 5, 10, 3, 14, 13, 16]
#car4 [17, 12, 9, 0, 16, 18, 8, 1, 11, 2, 7, 3, 4, 14, 10, 15, 5, 6, 19, 13]
#car5 [13, 19, 28, 4, 17, 10, 16, 12, 5, 8, 1, 0, 2, 20, 6, 22, 9, 23, 7, 3,
#      15, 29, 25, 26, 14, 11, 24, 21, 18, 27]

nbm=len(p_ij)
nbj=len(p_ij[0])

print('Number of Machines:{0},Number of Jobs:{1}'.format(nbm,nbj))

#set the parameters
Npop = 30    # Number of population
Pc = 1       # Probability of crossover
Pm = 0.8     # Probability of mutation
D=0.95       #Threshold parameter
sig=0.99     #sigema=0.99

print('The parameters we chosen:\n''Population size:{0}\nCrossover probability:{1}\n\
Initial mutation probability:{2}\nThreshold parameter:{3}\n'.format(Npop,Pc,Pm,D))

#Number of evaluations
stopGeneration = 1000*nbj
#(it will take a long time to run the algorithm)
#(to test the algrithm we can set stopGeneration small)
#stopGeneration = 100

#calculate the makespan
def makespan_GA(current_seq, p_ij, nbm):
    c_ij = np.zeros((nbm, len(current_seq) + 1))
```

```
    for j in range(1, len(current_seq) + 1):
        c_ij[0][j] = c_ij[0][j - 1] + p_ij[0][current_seq[j - 1]]

    for i in range(1, nbm):
        for j in range(1, len(current_seq) + 1):
            c_ij[i][j] = max(c_ij[i - 1][j], c_ij[i][j - 1]) + p_ij[i][current_seq[j - 1]]
    return c_ij[nbm - 1][nbj]

#initialize the population and append the neh_seq to the initial population
def initialization(Npop):
    pop = []
    for i in range(Npop):
        p = list(np.random.permutation(nbj))
        while p in pop:
            p = list(np.random.permutation(nbj))
        pop.append(p)
    pop.append(NEH_seq)
    return pop

#select the population
def selection(pop):
    #popobj[0]=makespans ,popobj[1]=order from 0 to 30 ,totally 31
    popObj = []
    for i in range(len(pop)):
        popObj.append([makespan_GA(pop[i],p_ij,nbm),i])
#sort by makespan,so the plpobj[1]will not be in order
    popObj.sort()
    distr = []
    distrInd = []

    for i in range(len(pop)):
        #append the makespan's order in the distrInd
        #(while the makespan in pop are in order ) ascending in makespan index
        distrInd.append(popObj[i][1])
        #Select parent 1 using 2k/M(M+1) fitness_rank distribution
        prob = (2*(len(pop)-i)) / (len(pop) * (len(pop)+1))
        distr.append(prob)

    parents = []
    for i in range(len(pop)):
        #Select parent 2 using uniform distribution
        parents.append(list(np.random.choice(distrInd, 1, p=distr)))
        parents[i].append(np.random.choice(distrInd))
    return parents

#2-point crossover (C2)
def crossover(parents):
    pos = list(np.random.permutation(np.arange(nbj-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

#shift mutation
def mutation(sol):
    pos = list(np.random.permutation(np.arange(nbj))[: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

#Update the population
def elitistUpdate(oldPop, newPop):
    bestSolInd = 0
    bestSol = makespan_GA(oldPop[0],p_ij,nbm)

    for i in range(1, len(oldPop)):
        tempObj = makespan_GA(oldPop[i],p_ij,nbm)
        if tempObj < bestSol:
            bestSol = tempObj
            bestSolInd = i
    rndInd = random.randint(0,len(newPop)-1)
    newPop[rndInd] = oldPop[bestSolInd]
    return newPop

# find the best solution
def findBestSolution(pop):
    bestObj = makespan_GA(pop[0],p_ij,nbm)
    avgObj = bestObj
    bestInd = 0
    for i in range(1, len(pop)):
        tObj = makespan_GA(pop[i],p_ij,nbm)
        avgObj = avgObj + tObj
        if tObj < bestObj:
            bestObj = tObj
            bestInd = i

    return bestInd, bestObj, avgObj/len(pop)

# Run the algorithm for 'stopGeneration'(1000*n) times generation
def Loop(Npop):

    Pm = 0.8
    # Creating the initial population
    population = initialization(Npop)
```

```
    for i in range(stopGeneration):
        # Selecting parents
        parents = selection(population)
        childs = []

        # Apply crossover
        for p in parents:
            r = random.random()
            y = random.random()
            if r < Pc:
                childs.append(crossover([population[p[0]], population[p[1]]]))
            else:
                if y < 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_new = elitistUpdate(population, childs)
        # Results Time
        bestSol, minObj, avgObj = findBestSolution(population_new)
        mpset=[]
        mpset.append(minObj)
        if minObj/avgObj > D:
            Pm=sig*Pm
    return min(mpset)

# recode 30 seeds results for each run
def ran_seed(carnum):
    result=[]
    for times in range(30):
        np.random.seed(30*times)
        min_makespan=(Loop(Npop))
        result.append(min_makespan)
        print('seed {0}: makespan value {1}'.format(times+1,min_makespan))
    return result

#print the makespan table
result=ran_seed(p_ij)
print('\nmakespan table:\n''mininum:{0}, maximum:{1}, mean:{2}, standard deviation:{3}'.\
format(np.min(result),np.max(result),np.mean(result),np.std(result)))
```

```
############################################################
# Visualization
############################################################
import matplotlib.pyplot as plt
import pandas as pd
plt.style.use("ggplot")

###Car 1
df=pd.DataFrame()
df["NEH"]=[7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0]

df["Random"]=[7038.0, 7038.0, 7038.0, 7057.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7059.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7048.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7057.0, 7038.0, 7048.0, 7038.0, 7038.0]

df["initial"]=[7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0]

df["P=5"]=[7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0]

df["P=10"]=[7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0]

df["P=20"]=[7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0]

df["P=50"]=[7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0]

df["P=100"]=[7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0]

df["Pc=0"]=[8220.0, 7938.0, 7655.0, 7829.0, 7590.0, 7792.0, 7828.0, 7472.0, 7991.0, 7689.0,
7626.0, 7700.0, 8098.0, 7272.0, 7565.0, 7844.0, 7685.0, 7740.0, 7626.0, 7677.0, 8435.0, 7708.0,
8243.0, 7797.0, 7730.0, 7804.0, 7885.0, 8277.0, 7685.0, 7573.0]
```

df["Pc=0.5"]=[7648.0, 8171.0, 7589.0, 7890.0, 7817.0, 8092.0, 7971.0, 8130.0, 7612.0, 8206.0, 7542.0, 7732.0, 7677.0, 7687.0, 7652.0, 7847.0, 7706.0, 7548.0, 7781.0, 7797.0, 7843.0, 7779.0, 7563.0, 7593.0, 7820.0, 7530.0, 7803.0, 7753.0, 7768.0, 7444.0]

df["Pc=0.7"]=[7689.0, 7899.0, 7689.0, 7892.0, 8009.0, 7689.0, 7834.0, 7974.0, 7276.0, 7904.0, 7618.0, 7726.0, 7788.0, 7816.0, 8125.0, 7961.0, 7974.0, 7965.0, 7689.0, 7554.0, 7789.0, 7689.0, 7803.0, 7707.0, 7881.0, 7230.0, 7697.0, 8254.0, 7709.0, 7828.0]

df["Pc=0.9"]=[8098.0, 7502.0, 7689.0, 7832.0, 7689.0, 7276.0, 7643.0, 7370.0, 7519.0, 7677.0, 7978.0, 7935.0, 7760.0, 7839.0, 7793.0, 7472.0, 7689.0, 7712.0, 7828.0, 7853.0, 7229.0, 7554.0, 7358.0, 7712.0, 7894.0, 7491.0, 7566.0, 8062.0, 7880.0, 7802.0]

df["Pm=0"]=[7038.0, 7038.0]

df["Pm=0.2"]=[7038.0, 7038.0]

df["Pm=0.4"]=[7038.0, 7038.0]

df["Pm=0.6"]=[7038.0, 7038.0]

df["Pm=1"]=[7038.0, 7038.0]

df["D=0"]=[7038.0, 7038.0]

df["D=0.5"]=[7038.0, 7038.0]

df["D=1"]=[7038.0, 7038.0]

```
plt.figure(figsize=(16,5))
df.boxplot()
plt.title('Car 1', size=20)
plt.xlabel("Different Parameters", size=15)
plt.ylabel("Makespan Values", size=15)
plt.show()

#boxplot 3 algorithms
#car1
df=pd.DataFrame()

df["NEH"]=[7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0]
df["Random"]=[7038.0, 7038.0, 7038.0, 7057.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7059.0,
7038.0, 7038.0, 7095.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7048.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7057.0, 7038.0, 7048.0, 7038.0, 7038.0]
df["Genetic"]=[7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0,
7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0, 7038.0]
df.boxplot()
plt.title('Car 1', size=20)
plt.xlabel("Method used", size=15)
plt.ylabel("Makespan Values", size=15)
plt.show()
```