Job scheduling for a complex space factory

Janek Zimoch, jz16935

29th September 2020

1 Introduction

Problem presented in this coursework is a complex variation of a job shop scheduling problems [1]. The objective is to find an optimal sequence of jobs s^*, such that the makespan for all jobs $C_{max}(s^*)$ is minimised as presented in Equation 1, where X is a set of all possible job sequences.

$$C_{max}(s^*) = \min_{s \in X} C_{n,m}(s) \tag{1}$$

For the presented problem, which involves offline ordering of $n = 30$ jobs, set X contains as many as $n! = 30! = 2.65 \times 10^{32}$ permutations of job sequences. Moreover, the computational time required to solve such problem grows with number of workstations m as $(n!)^m$ [2]. It has been proven that job shop scheduling problems with more than 3 machines are NP-hard [3]. This means, that due to their computational complexity, it is impossible to find an optimal solution in polynomial time, but only to test fitness of some solutions in set X.

Therefore, the mixed-integer programming model which we develop in Section 2 (Q1), won't be applied to solve the problem directly. Instead, in Section 3 (Q2), we use it to create a fitness function to enable evaluation of makespan for any sequence of jobs. Further, this fitness function will be used alongside a Hybrid Genetic Algorithm metaheuristic method to search for a (near-)optimal solution.

1.1 Nomenclature used

- n - number of jobs to be manufactured, $n = 30$
- m - number of workstations, $m = 7$
- i - job number (i.e. "name" of a job)
- j - job's position in a job sequence
- k - workstation's position in a factory
- $C_{j,k}$ - handling finish time of j^{th} job in sequence at k^{th} workstation
- $J_{i,j}$ - binary decision matrix; element value of 1 indicates that i^{th} job will be manufactured as j^{th} in the sequence
- $D_{i,k}$ - description of a k^{th} operation to be performed on job i
- $Handling_{j,k}$ - a minimum time j^{th} job has to stay at k^{th} workstation
- $Align_{j,k}$ - time it takes for k^{th} workstation to prepare for loading of j^{th} job

2 Q1 - Mixed-Integer programming model

Intricacy of this coursework problem is in the fact that order of workstations is set (Flow Shop Scheduling Problem, FSSP), each job has a different sequence of workcells to visit (Job Shop Scheduling Problem, JSSP), and workcells within a workstation can operate simultaneously (Parallel-Machine Problem) [2]. This entanglement of workstations and workcells, makes it difficult to derive a complete mathematical programming model. Therefore, we begin with simplified version of the problem before attempting to model a complete scenario.

2.1 Zero-buffer permutation flow shop scheduling problem

Decision Variables To reduce complexity, we assume that each workstation can process only a single job at a time, thus simplyfing problem to that of Zero-Buffer Permutation FSSP. To map job sequence s to a binary representation we define a binary decision variable matrix J, where columns j specify a job's position in the sequence s and rows i represent a specific job. Presence of 1 in a cell indicates that a job i should enter factory as j^{th} in sequence. Sparsity and binary from of matrix J is defined with constraints in Equations 3, 20, and 5.

$$\underset{n\times n}{J} = \begin{bmatrix} J_{1,1} & J_{1,1} & . & . & J_{1,n} \\ J_{2,1} & . & & & . \\ . & & J_{i,j} & & . \\ . & & & . & . \\ J_{n,1} & . & . & . & J_{n,n} \end{bmatrix} \tag{2}$$

$$\sum_{j=1}^{n} J_{i,j} = 1\,,\ \forall\, i \in \{1, ..., n\} \tag{3}$$

$$\sum_{i=1}^{n} J_{i,j} = 1\,,\ \forall\, j \in \{1, ..., n\} \tag{4}$$

$$J_{i,j} \in \{0, 1\}\,,\ \forall\, j \in \{1, ..., n\}\,,\ and\, i \in \{1, ..., n\} \tag{5}$$

Jobs' description In order to optimise job scheduling we need information about operations to be performed for each job i. For this we use matrix D where each job i is represented by a separate row of operations k. For matrix $\underset{n\times m}{D}$ see Appendix A.

Constraints on Start Time Job handling at k^{th} workstation starts from loading a job. For j^{th} job to be loaded, workstations $(k-1)^{th}$ and k^{th} have to be aligned (i.e. correct workcells have to be in contact with loading and offloading modules). Workcell holding a job at $(k-1)^{th}$ workstation, will be aligned with off-loading transfer module at the end of $C_{j,k-1}$. Workcell receiving a job at k^{th} workstation, will be aligned with loading transfer module once k^{th} workstation finishes handling $(j-1)^{th}$ item $(C_{j-1,k})$ and workstation rotates $(Align_{j,k})$. Thus, following two constraints can be identified:

$$C_{j,k} \geq C_{j-1,k} + Handling_{j,k} + Align_{j,k} \tag{6}$$

$$C_{j,k} \geq C_{j,k-1} + Handling_{j,k} \tag{7}$$

Exact definitions for $Handling_{j,k}$ and $Align_{j,k}$ will be presented in later paragraphs.

Constraints on End Time Job handling at k^{th} workstation finishes when a job can be offloaded to $(k+1)^{th}$ workstation at a next time unit. For this to be possible, relevant workcell in workstation k^{th} has to be aligned for offloading, and workstation $(k+1)^{th}$ has to be aligned for receiving. First condition is already satisfied by the two Start Time Constraints defined in Equations 6 and 7. Second, condition is satisfied when workstation $(k+1)^{th}$ finishes handling $(j-1)^{th}$ job and rotates to align a correct workcell with loading transfer module.

$$C_{j,k} \geq C_{j-1,k+1} + Align_{j,k+1} \tag{8}$$

This constraint defines the 'Zero-buffer' property of the assumed FSSP system, which says that a job cannot be finished until, next workstation is ready to pick it up. Additionally, this constraint makes Start Time constraint expressed as Equation 6 above, redundant, for all, but jobs at first workstation. This is because inequality $C_{j,k} \geq C_{j-1,k+1} + Align_{j,k+1}$ also ensures that $C_{j,k-1} \geq C_{j-1,k} + Align_{j,k}$ is always true (these are the same inequalities).

Handling$_{j,k}$ In this model, $Handling_{j,k}$ consists of loading, which takes 1 time unit, traveling, which takes 3 time units, and job manufacturing which is specific for each job-workstation combination. Loading happens first, and traveling happens in parallel with job manufacturing, with the more time consuming one defining length of $Handling_{j,k}$. Start Time constraint from Equation 7 can be expanded to account for duality of Handling.

$$C_{j,k} \geq C_{j,k-1} + 1 + 3 \tag{9}$$

$$C_{j,k} \geq C_{j,k-1} + 1 + \sum_{i=1}^{n} D_{i,k} J_{i,j} \tag{10}$$

Align$_{y,z}$ This term defines time workstation z will take to align appropriate workcell with the loading transfer module. It also accounts for one time unit which workstation z has to spend to offload job $(y-1)$. Workstation, could be rotating clockwise or counterclockwise to align the workcell, thus both directions are considered and shorter is selected as defined by Equation 11. z and y are general matrix element indexes.

$$Align_{y,z} = 1 + min\left(a_{y,z},\ b_{y,z}\right) \tag{11}$$

$$a_{y,z} = \left| \left(\sum_{i=1}^{n=30} D_{i,z} J_{i,y-1} - 3 \right) - \sum_{i=1}^{n=30} D_{i,z} J_{i,y} \right| \tag{12}$$

$$b_{y,z} = \left| 8 - \left| \left(\sum_{i=1}^{n=30} D_{i,z} J_{i,y-1} - 3 \right) - \sum_{i=1}^{n=30} D_{i,z} J_{i,y} \right| \right| \tag{13}$$

Additional notes and assumptions (1) It is assumed that all workstations are correctly aligned at time zero to pickup the first job. (2) It is necessary to add an extra 1 to the final $C_{n,m}(s)$ to obtain makespan $C_{max}(s)$, as it accounts for offloading of last job from the factory (This won't affect optimisation process). (3) Calculation of makespan for the first job at first workstation $C_{1,1}$ is unique and defined as:

$$C_{1,1} \geq 1 + 3 \tag{14}$$

$$C_{1,1} \geq 1 + \sum_{i=1}^{n=30} D_{i,1} J_{i,1} \tag{15}$$

2.2 Complete job scheduling problem

By allowing to process more than one job at a workstation, the scheduling problem becomes complicated. Now we do not only have to decide what is the better input sequence to the factory, but also how jobs should overtake each other and in what order, whether it is optimal for some jobs to wait to let other pass etc. We believe that in such scenario even checking fitness of some input sequence is computationally infeasible. Therefore to reduce complexity we need to expand definition of solution from a 1-D vector to a 2-D matrix, where each column is a separate sequence of jobs for each workstation. Ordering of those sequences is interdependent on each other, due to zero-buffer constraint of the factory.

Decision Variables It is now required to extend Decision Variables into 3 dimensions, to account for different job sequencing for each workstation.

$$\sum_{j=1}^{n} J_{i,j,k} = 1\,,\ \forall\, i \in \{1,...,n\}\,,\ k \in \{1,...,m\} \tag{16}$$

$$\sum_{i=1}^{n} J_{i,j,k} = 1\,,\ \forall\, j \in \{1,...,n\}\,,\ i \in \{1,...,m\} \tag{17}$$

$$J_{i,j,k} \in \{0,1\}\,,\ \forall\, j \in \{1,...,n\}\,,\ i \in \{1,...,n\}\,,\ k \in \{1,...,m\} \tag{18}$$

However, not every reordering of jobs across workstations is possible. For example 30^{th} job at $k-1$ cannot become 1^{st} job at k, because there are only 8 workcells per workstation and one transfer unit. Thus the following constraint is created to prevent clogging.

$$\sum_{j_{k+1}=j_k-7}^{j_k+7} J_{i,j_{k+1},k+1} = 1\,,\ \forall\, i \in \{1,...,n\}\,,\ k \in \{1,...,\text{m-1}\} \tag{19}$$

One job per workcell Each workcell can physically process only a single job at a time. This could be accounted for by extending Decision Variable matrix J into 4 dimensions. For the constraint below $h=8$ and it is a number of workcells, and q is workcell number. (Note, constraints 16, 17, and 18 would have to be extended accordingly).

$$\sum_{q=1}^{h} J_{i,j,k,q} = 1\,,\ \forall\, i \in \{1,...,n\}\,,\ k \in \{1,...,m\}\,,\ j \in \{1,...,n\} \tag{20}$$

Start Time constraints It is yet unclear how to exactly formulate constraints affecting start time of i^{th} job at k^{th} workstation. Such constraints would have to account for:

- whether job i has finished at workstation $k-1$
- whether it is i^{th} job's turn in sequence to use transfer module.
- whether required workcell q at workstation k is free

End Time constraints Similarly to determine the end time of job i at k^{th} workstation we would need to account for:

- whether job i can enter workstation $(k+1)$
- whether workcell q at $(k+1)$ is free

2.3 Q1 - Part 2

Zero-buffer permutation FSSP Under assumption of Zero-buffer permutation FSSP problem there is no advantage for a job to move backwards. There is no advantage of using workcells from previous workstations over those from the next. Such operation would not only prolong manufacturing of a job which moves backwards but also delay all other jobs later in the sequence, resulting in a worse makespan overall.

Complete job scheduling problem In this scenario moving backwards could be feasible. Depending on a sequence of jobs, movement backward of some job i wouldn't have to halt subsequent jobs, and on contrary could actually reduce makespan if desired workcell on a next workstation wasn't currently available.

To implement this addition we could combine scheduling with traveling salesmen problem. Every workcell would be a node and each job would have to find a path which, given other constraints, would minimise its manufacturing time. With such approach job could move to a node/workcell at previous workstations if such path was happened to be advantageous.

Zero-Bufer FSSP - Final mixed-integer programming model

Objective function

$$C_{max}(s^*) = \min_{s \in X} C_{n,m}(s)$$

Decision Variable Constraints

$$\sum_{j=1}^{n} J_{i,j} = 1\,, \qquad \forall\, i \in \{1,...,n\}$$

$$\sum_{i=1}^{n} J_{i,j} = 1\,, \qquad \forall\, j \in \{1,...,n\}$$

$$J_{i,j} \in \{0,1\}\,, \qquad \forall\, j \in \{1,...,n\}\,,\ \ i \in \{1,...,n\}$$

Start Time Constraints

$$C_{j,k} \geq C_{j,k-1} + 1 + 3\,, \qquad \forall\, j \in \{1,...,n\}\,,\ \ k \in \{2,...,m\}$$

$$C_{j,k} \geq C_{j,k-1} + 1 + \sum_{i=1}^{n} D_{i,k} J_{i,j}\,, \qquad \forall\, j \in \{1,...,n\}\,,\ \ k \in \{2,...,m\}$$

$$C_{j,k} \geq C_{j-1,k} + 1 + Align_{j,k} + 3\,, \qquad \forall\, j \in \{2,...,n\}\,,\ \ k \in \{1\}$$

$$C_{j,k} \geq C_{j-1,k} + 1 + Align_{j,k} + \sum_{i=1}^{n} D_{i,k} J_{i,j}\,, \qquad \forall\, j \in \{2,...,n\}\,,\ \ k \in \{1\}$$

End Time Constraints

$$C_{j,k} \geq C_{j-1,k+1} + Align_{j,k+1}\,, \qquad \forall\, j \in \{2,...,n\}\,,\ \ k \in \{1,...,m-1\}$$

Align Function

$$Align_{y,z} = 1 + min\,(a_{y,z},\ b_{y,z})$$

$$a_{y,z} = \left| \left(\sum_{i=1}^{n} D_{i,z} J_{i,y-1} - 3 \right) - \sum_{i=1}^{n} D_{i,z} J_{i,y} \right|$$

$$b_{y,z} = \left| 8 - \left| \left(\sum_{i=1}^{n} D_{i,z} J_{i,y-1} - 3 \right) - \sum_{i=1}^{n} D_{i,z} J_{i,y} \right| \right|$$

3 Q2 - Metaheuristics

3.1 Fitness function

Psudo code in Algorithm 1 describes a makespan (fitness) function. For Python implementation see Appendix B.

Algorithm 1 Fitness function

```
1: function ALIGN($y, z$)                          ▷ roate $z^{th}$ workstation to pickup $y^{th}$ job
2:     off-loading $\leftarrow 1$
3:     clockwise $\leftarrow |(D_{y-1,z} - travel) - D_{y,z}|$
4:     counter-clockwise $\leftarrow |8 - |(D_{y-1,z} - travel) - D_{y,z}||$
5:     Align$_{y,z} \leftarrow$ off-loading $+ min($clockwise, counter-clockwise$)$
6: function GET MAKESPAN($s[\,]$)                  ▷ input: job sequence $s[\,]$
7:     $n \leftarrow$ number of jobs
8:     $m \leftarrow$ number of workstations
9:     $travel \leftarrow 3$
10:    $loading \leftarrow 1$
11:    $\underset{n\times m}{C} \leftarrow$ empty makespan matrix
12:    $\underset{n\times m}{D} \leftarrow$ job manufacturing plan
Ensure: reorder rows of $D$ by sequence $s$
1st cell $C_{1,1}$
13:    a $\leftarrow travel + loading$             ▷ time it takes to load and rotate to offloading module
14:    b $\leftarrow D_{1,1} + loading$            ▷ time it takes to load and manufacture
15:    $C_{1,1} \leftarrow max(a,\ b)$
1st row $C_{1,k}$
16:    for $k := 2$ to $m$ do
17:        a $\leftarrow C_{1,k-1} + travel + loading$
18:        b $\leftarrow C_{1,k-1} + D_{1,1} + loading$
19:        $C_{1,k} \leftarrow max(a, b)$
Bulk of $C_{j,k}$
20:    for $j := 2$ to $n$ do
21:        a $\leftarrow C_{j-1,1} + travel + loading$
22:        b $\leftarrow C_{j-1,1} + D_{j,1} + loading$
23:        c $\leftarrow C_{j-1,2} + Align(j, 2)$
24:        $C_{j,1} \leftarrow max(a,\ b,\ c)$      ▷ handling time for jobs at $k = 1$
25:        for $k := 2$ to $m$ do
26:            a $\leftarrow C_{j,k-1} + travel + loading$
27:            b $\leftarrow C_{j,k-1} + D_{j,k} + loading$
28:            $C_{j,k} \leftarrow max(a,\ b)$
29:            if $k < m$ then                 ▷ there is no $k + 1$ for last workstation
30:                c $\leftarrow C_{j,k} + Align(j, k + 1)$
31:                $C_{j,k} \leftarrow max(C_{j,k},\ c)$
32:    $C_{max} \leftarrow C_{n,m} +$ off-loading
33:    return $C_{max}$                          ▷ return final makespan
```

3.2 Hybrid Genetic Algorithm

We decided to use Hybrid Genetic Algorithm, because evolutionary algorithms provide means of escaping local minimum, a major problem of constructive approaches and local search algorithms [7]. However, Evolutionary algorithms are rather slow at converging for job shop problems [9]. Therefore, we use Hybrid Genetic Algorithm, in which standard GA is enhanced with Tabu Search [10], which improved convergence. Figure 1 shows a flow chart of the Hybrid Genetic Algorithm developed for this coursework.

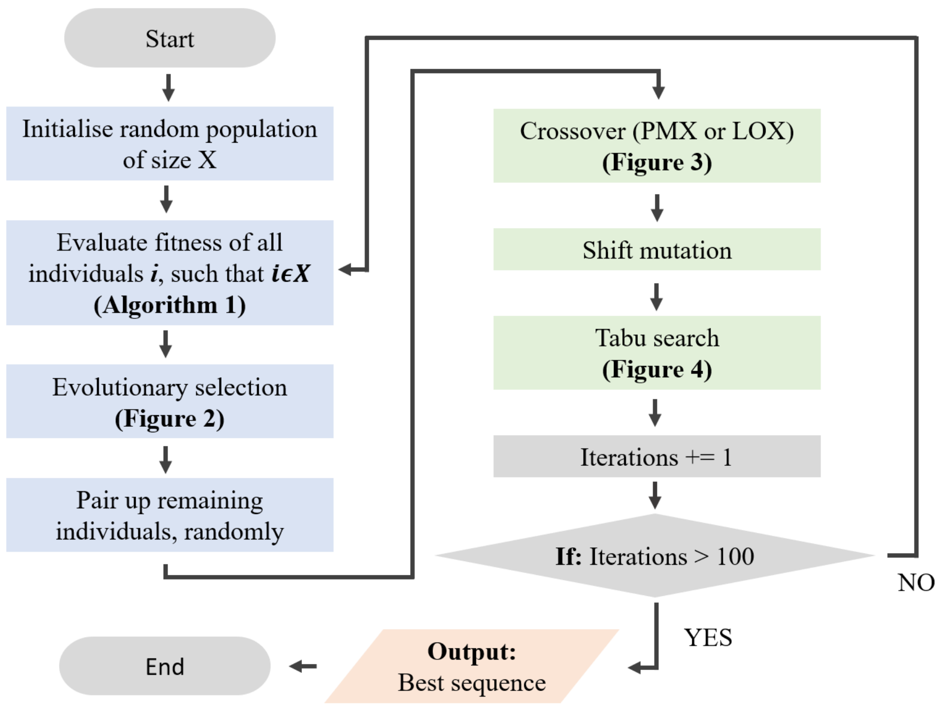

Start
Initialise random population of size X
Evaluate fitness of all individuals *i*, such that *i*ϵ*X* **(Algorithm 1)**
Evolutionary selection **(Figure 2)**
Pair up remaining individuals, randomly
Crossover (PMX or LOX) **(Figure 3)**
Shift mutation
Tabu search **(Figure 4)**
Iterations += 1
If: Iterations > 100
NO
YES
Output: Best sequence
End

Figure 1: Hybrid Genetic Algorithm used in this coursework

Representation of a solution For FSSP the standard way of representing a solution is in form of a job sequence (permutation code) where j^{th} gene is a j^{th} job in that sequence.

Initialisation of population Initial population is generated randomly, which guarantees the best coverage of search space. We start with population of 100 individuals and reduce to 20 after first iteration.

Survival criterion For this coursework we explore an idea of accounting for, both, fitness and diversity when selecting individuals for survival. We believe that by selecting individuals which are simultaneously most fit and most diverse, we will favour search of good solutions outside of local neighborhood [4]. Figure 2 shows a flow chart describing this algorithm. In vector $\bar{e}$ each element i indicates position j of job i in a sequence for that individual. In vector $\bar{p}$ each element at index i represents mean position in sequence of each job i evaluated for all parents. For Python implementation see Appendix C

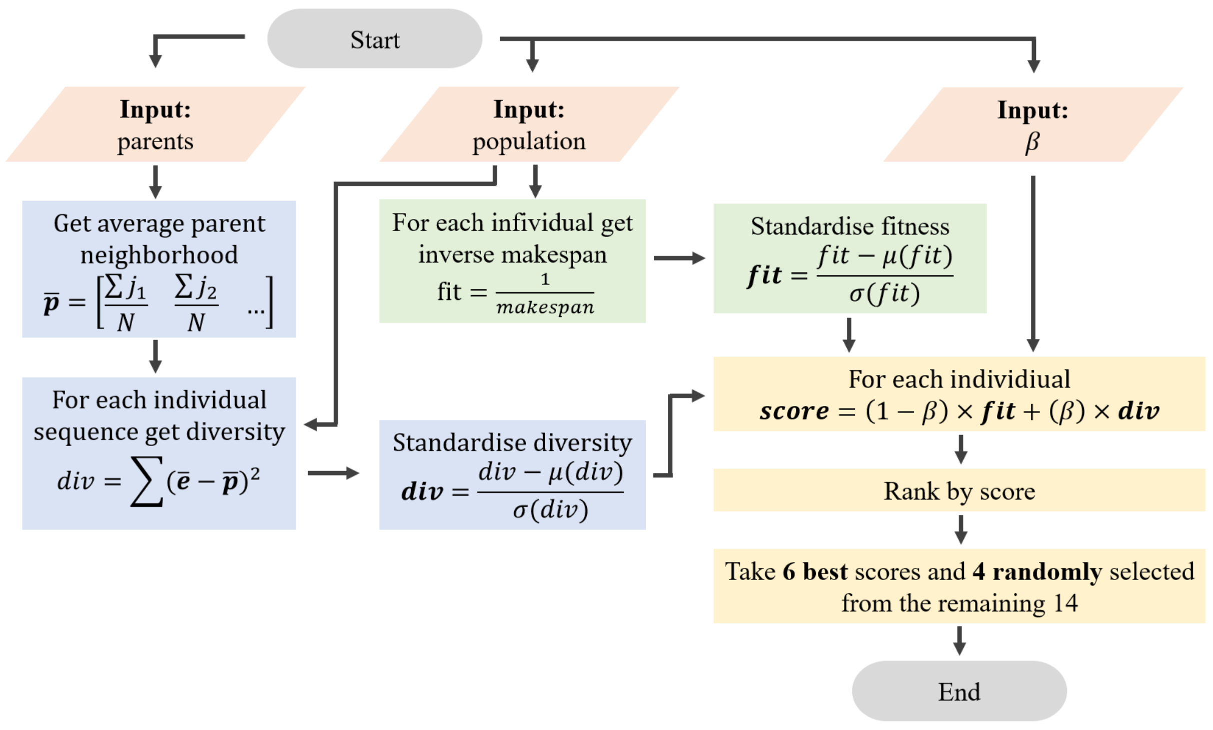

Start
Input: parents
Input: population
Input: β
Get average parent neighborhood
$\bar{\boldsymbol{p}} = \left[\frac{\sum j_1}{N} \quad \frac{\sum j_2}{N} \quad \dots\right]$
For each infividual get inverse makespan
fit $= \frac{1}{makespan}$
Standardise fitness
$\boldsymbol{fit} = \frac{fit - \mu(fit)}{\sigma(fit)}$
For each individual sequence get diversity
$div = \sum(\bar{e} - \bar{\boldsymbol{p}})^2$
Standardise diversity
$\boldsymbol{div} = \frac{div - \mu(div)}{\sigma(div)}$
For each individiual
$\boldsymbol{score} = (1-\beta) \times \boldsymbol{fit} + (\beta) \times \boldsymbol{div}$
Rank by score
Take **6 best** scores and **4 randomly** selected from the remaining 14
End

Figure 2: Algorithm for diversity aware population selection

Crossover When performing crossover with permutation codes, we often produce infeasible offspring, which then have to be corrected. There are many existing crossover operators which account for this: PMX [5], CX [6], LOX [6], OBX [5], PBX [5], and many others. In this implementation we tested PMX and LOX operators, for their overview see Figure 3. For the rate of crossover we used $P_c = 1$ as the survival of best parents is already ensured through elitist survival strategy. For Python implementation see Appendix D.

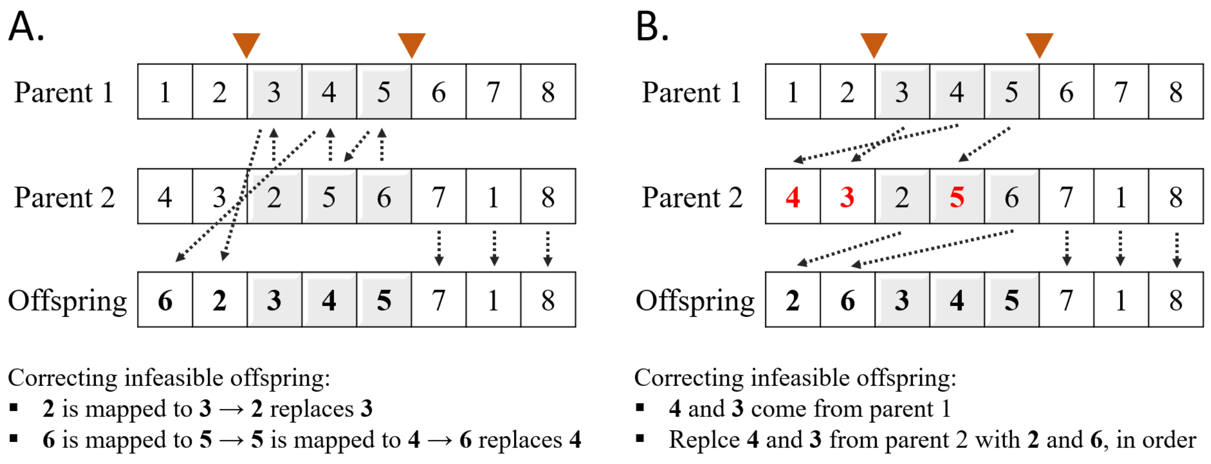

A.
Parent 1 1 2 3 4 5 6 7 8
Parent 2 4 3 2 5 6 7 1 8
Offspring 6 2 3 4 5 7 1 8
Correcting infeasible offspring:
▪ 2 is mapped to 3 → 2 replaces 3
▪ 6 is mapped to 5 → 5 is mapped to 4 → 6 replaces 4
B.
Parent 1 1 2 3 4 5 6 7 8
Parent 2 4 3 2 5 6 7 1 8
Offspring 2 6 3 4 5 7 1 8
Correcting infeasible offspring:
▪ 4 and 3 come from parent 1
▪ Replce 4 and 3 from parent 2 with 2 and 6, in order

Figure 3: (A) PMX crossover and (B) LOX crossover. Note, to create a second offspring we need to swap places/roles of parent 1 with parent 2

Mutation Among many mutation methods we choose shift mutation [1] with mutation rate parameter $P_m = 0.05$. Similar approach was used by Etiler et al. [11] and Ruiz et al. [12]. For Python implementation see Appendix D.

Tabu search Tabu Search was implemented to improve convergence of Genetic Algorithm through a local search technique. Algorithm flow chart is presented in Figure 4 and for Python implementations see Appendix E. Number of iterations n was kept low, below 10, and each generation only 6 out of 20 individuals were selected for Tabu search optimisation, to reduce computational cost.

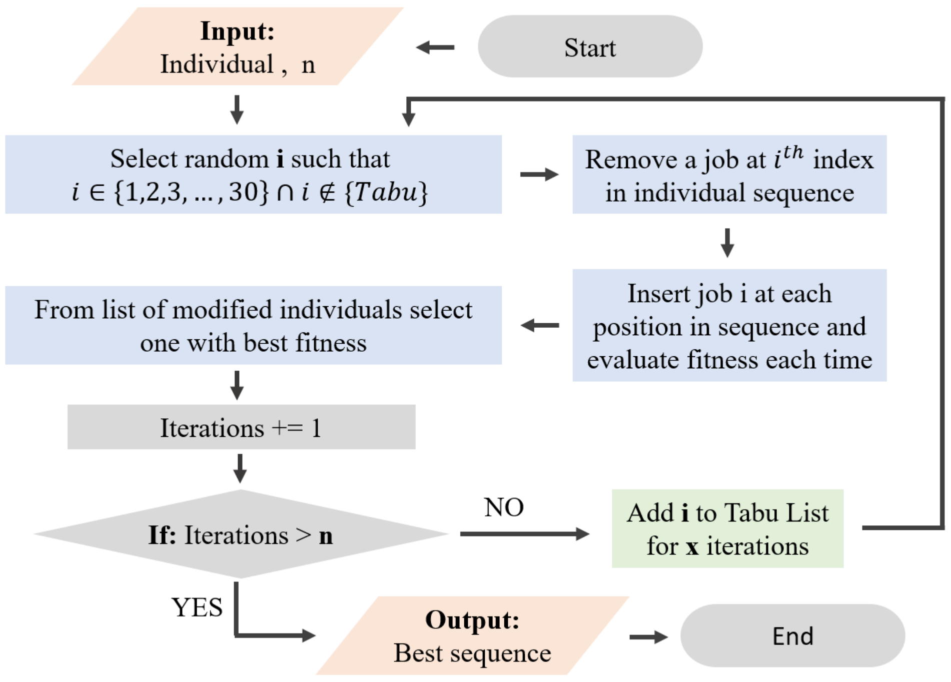

Input:
Individual , n
Start
Select random **i** such that
$i \in \{1,2,3,\ldots,30\} \cap i \notin \{Tabu\}$
Remove a job at i^{th} index in individual sequence
Insert job i at each position in sequence and evaluate fitness each time
From list of modified individuals select one with best fitness
Iterations += 1
If: Iterations > **n**
NO
Add **i** to Tabu List for **x** iterations
YES
Output:
Best sequence
End

Figure 4: Tabu Search flow chart

Stopping criterion There are numerous ways to terminate Genetic Algorithm based on: convergence, number of iterations, or other conditions. For simplicity we decided to terminate algorithm after 100 generations, as by then solutions have already converged.

Best sequence Under an assumption of Zero-buffer permutation FSSP the best sequence s^* found using Hybrid Genetic Algorithm has makespan of 326.

$$s^* = [9, 29, 2, 4, 1, 26, 28, 3, 25, 18, 22, 10, 19, 5, \\ 7, 20, 8, 30, 16, 11, 27, 13, 14, 21, 23, 15, 6, 12, 24, 17] \tag{21}$$

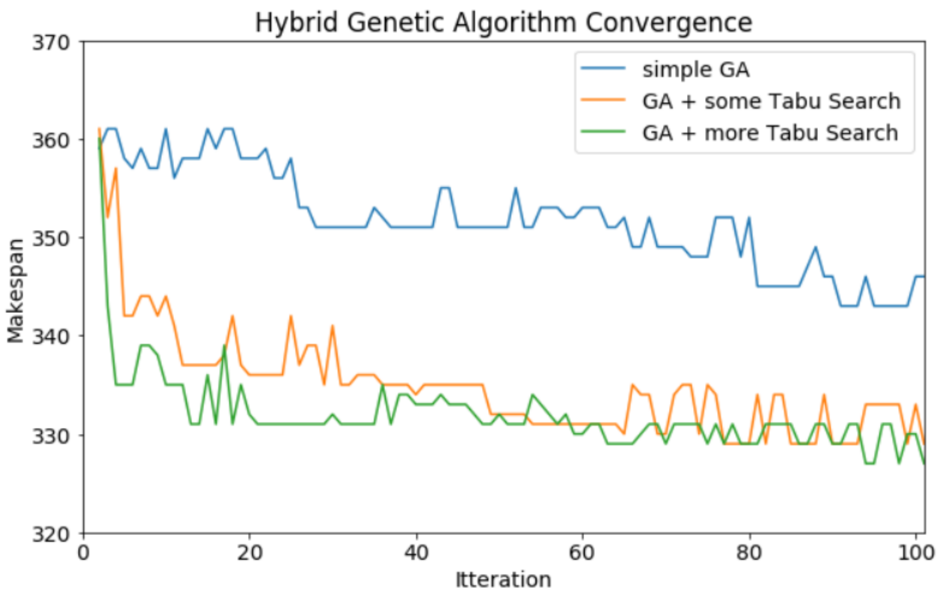


Figure 5: Makespan of best indiviudal across 100 generations for 3 variations of Genetic algorithm. (Blue) no Tabu search applied; (Orange) each generation 3 individuals are optimised with 6 neighborhood insertions each; (Green) each generation 5 individuals are optimised with 8 neighborhood insertions each.

Sequence evaluation - Lower Bound A lower bound (LB) can be used to evaluate how close to optimal solution a given solution is. One way to compute lower bound is through relaxation of no-preemption constraint. In such case no workstation stays idle as jobs are now flexible to use machines at 100% capacity.

To compute LB we replace all 1s and 2s in matrix D (see appendix A) with 3s to account for travel time, and add 2 to each element to account for loading and offloading time. Now we compute mean value of all elements in the resultant matrix and obtain $\mu = 6.90$. This is an average handling time for a job, assuming that no job is ever delayed (relaxed no-preemption constraint). Equation 22 gives $LB = 249$. Comparing this lower bound to $C_{max}(s^*) = 326$ we find that average delay per job per workstation for presented (sub-)optimal sequence is $\frac{326-249}{36} = 2.14$.

$$LB = \lceil (6 + 1 \times 30) \times 6.90 \rceil = 249 \tag{22}$$

References

[1] Werner F. Genetic Algorithms For Shop Scheduling Problems: a Survey; Otto-von-Guericke-Universität, Fakultät fur Mathematik, 39106 Magdeburg, Germany;

[2] Fatos Xhafa and Ajith Abraham. 2008. Metaheuristics for Scheduling in Industrial and Manufacturing Applications (1st. ed.). Springer Publishing Company, Incorporated.

[3] Lawler E.L., Lenstra J.K., Rinnoy Kan A.H.G. and Shmoys D.B., "Sequencing and scheduling: algorithms and complexity", Chapter 9 in Logistics of production and inventory, Graves S.C., Rinnooy Kan A.H.G. and Zipkin P.H. (Eds.), Elsevier Science Publishers, p. 445-522, 1993

[4] MIT 6.034 Artificial Intelligence, Fall 2010, online lecture, 13. Learning: Genetic Algorithms; https://www.youtube.com/watch?v=kHyNqSnzP8Y

[5] Syswerda, G. Schedule optimization using genetic algorithms. In: L. Davis (ed.), Handbook of Genetic Algorithms, 1990, 332 – 349.

[6] Werner, F. On the solution of special sequencing problems. Ph.D. Thesis, TU Magdeburg, 1984 (in German).

[7] Lopez, P., & Roubellat, F. (Eds.). (2008). Production Scheduling. ISTE. https://doi.org/10.1002/9780470611050

[8] Glover F., "Heuristics for integer programming using surrogate constraints", Decision Sciences, vol. 8, p. 156-166, 1977.

[9] Grefenstette J.J., "Incorporating problem specific knowledge into genetic algorithms", in Genetic Algorithms and Simulated Annealing, Davis. L. (ed.), Morgan Kaufmann Publishers, p. 42-60, 1987.

[10] Santos, N., Rebelo, R., &amp; Pedroso, J. P. (2014). A tabu search for the permutation flow shop problem with sequence dependent setup times. International Journal of Data Analysis Techniques and Strategies, 6(3), 275. doi:10.1504/ijdats.2014.063062

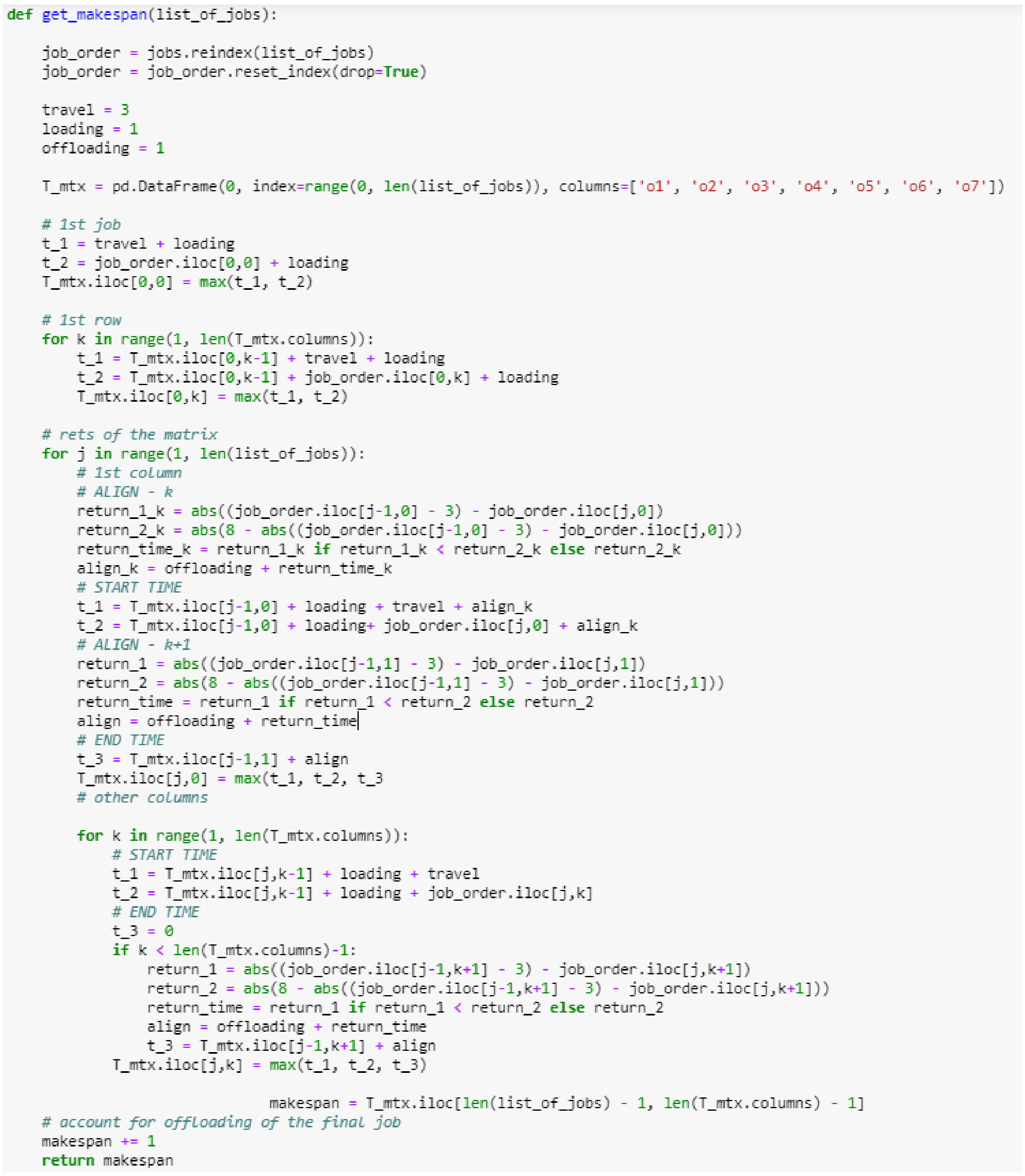
[11] Etiler, O.; Toklu, B.; Atak, M.; Wilson J. A genetic algorithm for flow shop scheduling problem. J Oper. Res. Soc., 2004, 55, 830–835.

[12] Ruiz, R.; Maroto, C.; Alcaraz; J. Two new robust genetic algorithms for the flowshop problem. Omega, 2006, 34, 461 – 476.

A Jobs' description matrix $\underset{n\times m}{D}$

$$\underset{n\times m}{D} = \begin{bmatrix}
6 & 2 & 5 & 7 & 3 & 5 & 3 \\
3 & 7 & 3 & 8 & 4 & 2 & 5 \\
8 & 1 & 3 & 2 & 3 & 4 & 5 \\
4 & 6 & 6 & 4 & 3 & 1 & 1 \\
4 & 2 & 8 & 1 & 2 & 8 & 6 \\
2 & 5 & 5 & 6 & 5 & 5 & 1 \\
3 & 8 & 5 & 4 & 7 & 5 & 3 \\
1 & 6 & 3 & 5 & 5 & 8 & 1 \\
1 & 4 & 1 & 6 & 2 & 8 & 3 \\
7 & 3 & 3 & 7 & 4 & 7 & 3 \\
5 & 8 & 1 & 6 & 8 & 1 & 6 \\
4 & 7 & 5 & 8 & 7 & 2 & 4 \\
4 & 3 & 2 & 6 & 7 & 4 & 8 \\
7 & 7 & 6 & 1 & 4 & 2 & 5 \\
7 & 7 & 1 & 8 & 7 & 8 & 7 \\
5 & 5 & 7 & 7 & 1 & 7 & 5 \\
7 & 7 & 4 & 6 & 2 & 1 & 1 \\
2 & 7 & 2 & 4 & 3 & 7 & 8 \\
1 & 7 & 3 & 6 & 6 & 1 & 8 \\
4 & 3 & 2 & 8 & 2 & 2 & 8 \\
8 & 3 & 2 & 5 & 8 & 5 & 2 \\
1 & 4 & 1 & 8 & 7 & 3 & 4 \\
5 & 7 & 1 & 3 & 6 & 7 & 5 \\
8 & 7 & 4 & 4 & 4 & 3 & 3 \\
5 & 3 & 2 & 4 & 3 & 1 & 4 \\
1 & 7 & 2 & 5 & 2 & 4 & 4 \\
6 & 7 & 7 & 8 & 5 & 8 & 4 \\
1 & 5 & 7 & 4 & 5 & 1 & 2 \\
6 & 4 & 5 & 6 & 7 & 4 & 8 \\
8 & 1 & 3 & 8 & 1 & 7 & 1
\end{bmatrix} \tag{23}$$

B Fitness function in Python

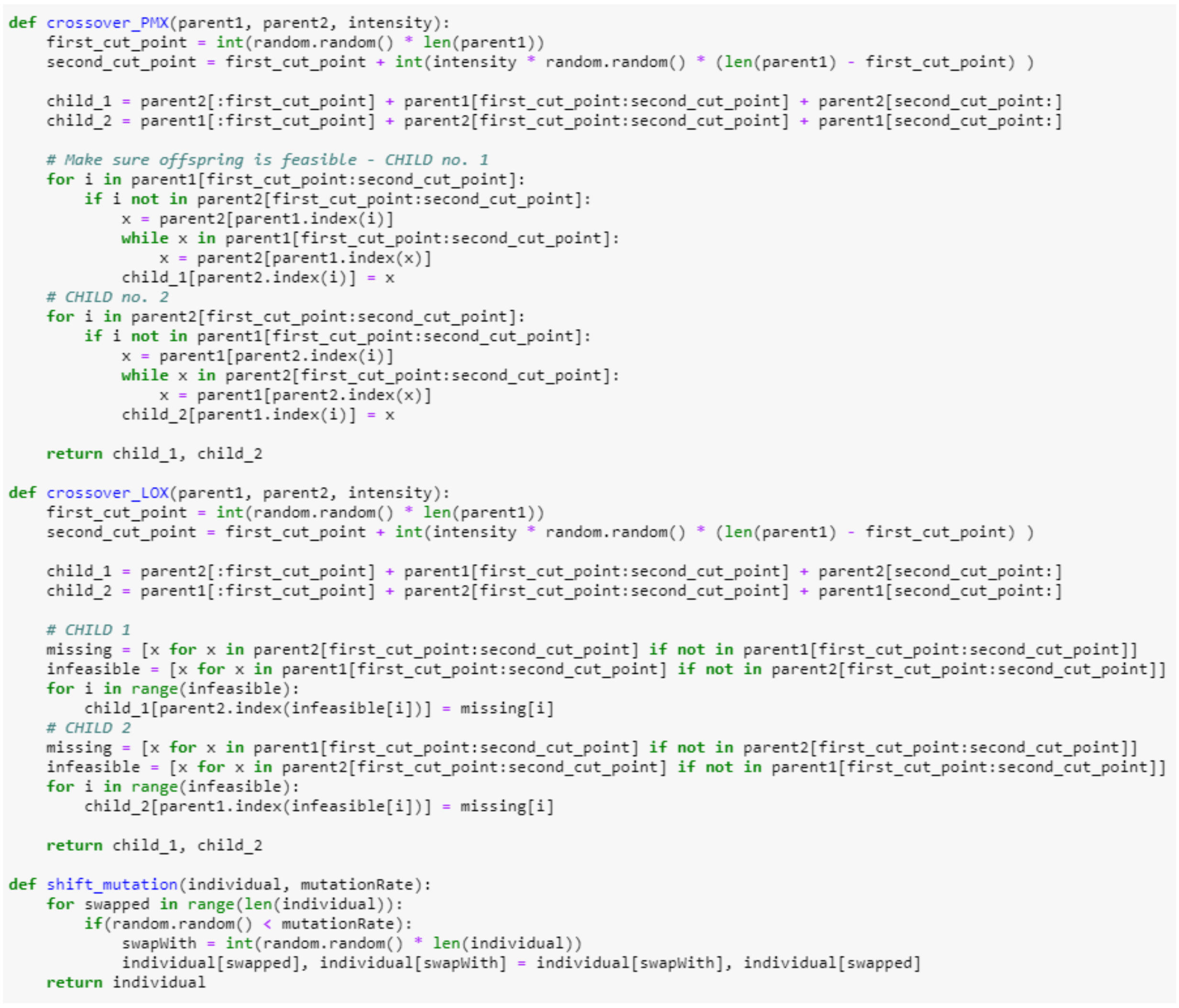


```
def get_makespan(list_of_jobs):

    job_order = jobs.reindex(list_of_jobs)
    job_order = job_order.reset_index(drop=True)

    travel = 3
    loading = 1
    offloading = 1

    T_mtx = pd.DataFrame(0, index=range(0, len(list_of_jobs)), columns=['o1', 'o2', 'o3', 'o4', 'o5', 'o6', 'o7'])

    # 1st job
    t_1 = travel + loading
    t_2 = job_order.iloc[0,0] + loading
    T_mtx.iloc[0,0] = max(t_1, t_2)

    # 1st row
    for k in range(1, len(T_mtx.columns)):
        t_1 = T_mtx.iloc[0,k-1] + travel + loading
        t_2 = T_mtx.iloc[0,k-1] + job_order.iloc[0,k] + loading
        T_mtx.iloc[0,k] = max(t_1, t_2)

    # rets of the matrix
    for j in range(1, len(list_of_jobs)):
        # 1st column
        # ALIGN - k
        return_1_k = abs((job_order.iloc[j-1,0] - 3) - job_order.iloc[j,0])
        return_2_k = abs(8 - abs((job_order.iloc[j-1,0] - 3) - job_order.iloc[j,0]))
        return_time_k = return_1_k if return_1_k < return_2_k else return_2_k
        align_k = offloading + return_time_k
        # START TIME
        t_1 = T_mtx.iloc[j-1,0] + loading + travel + align_k
        t_2 = T_mtx.iloc[j-1,0] + loading+ job_order.iloc[j,0] + align_k
        # ALIGN - k+1
        return_1 = abs((job_order.iloc[j-1,1] - 3) - job_order.iloc[j,1])
        return_2 = abs(8 - abs((job_order.iloc[j-1,1] - 3) - job_order.iloc[j,1]))
        return_time = return_1 if return_1 < return_2 else return_2
        align = offloading + return_time
        # END TIME
        t_3 = T_mtx.iloc[j-1,1] + align
        T_mtx.iloc[j,0] = max(t_1, t_2, t_3
        # other columns

        for k in range(1, len(T_mtx.columns)):
            # START TIME
            t_1 = T_mtx.iloc[j,k-1] + loading + travel
            t_2 = T_mtx.iloc[j,k-1] + loading + job_order.iloc[j,k]
            # END TIME
            t_3 = 0
            if k < len(T_mtx.columns)-1:
                return_1 = abs((job_order.iloc[j-1,k+1] - 3) - job_order.iloc[j,k+1])
                return_2 = abs(8 - abs((job_order.iloc[j-1,k+1] - 3) - job_order.iloc[j,k+1]))
                return_time = return_1 if return_1 < return_2 else return_2
                align = offloading + return_time
                t_3 = T_mtx.iloc[j-1,k+1] + align
            T_mtx.iloc[j,k] = max(t_1, t_2, t_3)

                              makespan = T_mtx.iloc[len(list_of_jobs) - 1, len(T_mtx.columns) - 1]
    # account for offloading of the final job
    makespan += 1
    return makespan
```

C Evolutionary selection in Python

```
def parents_diversity(list_of_ind):
    parent_div = []
    for i in zip(*list_of_ind):
        parent_div.append(sum(i) / len(list_of_ind))

    return parent_div

def diversity(individual, parent_div):
    a = np.array(individual)
    b = np.array(parent_div)
    c = np.power((a - b),2)
    return np.mean(c)

def evaluate_population(population, parents_div, beta):

    population_div = []
    population_fit = []
    for individual in population:
        population_div.append(diversity(individual, parents_div))
        population_fit.append(test_solution(individual))

    population_fit_inv = []
    for i in range(len(population_fit)):
        population_fit_inv.append(1 / population_fit[i])

    population_fit_mean = np.mean(population_fit_inv)
    population_fit_std = np.std(population_fit_inv)
    population_div_mean = np.mean(population_div)
    population_div_std = np.std(population_div)

    population_score = []

    for i in range(len(population_fit)):
        population_fit_norm = (population_fit_inv[i] - population_fit_mean) / population_fit_std
        population_div_norm = (population_div[i] - population_div_mean) / population_div_std + 0.0001
        if population_div_std < 1:
            population_score.append(population_fit_norm)
        else:
            population_score.append((1-beta)*population_fit_norm + beta*population_div_norm)

    return population_score, population_fit
```

D PMX and LOX Crossover and Shift Mutation



```
def crossover_PMX(parent1, parent2, intensity):
    first_cut_point = int(random.random() * len(parent1))
    second_cut_point = first_cut_point + int(intensity * random.random() * (len(parent1) - first_cut_point) )

    child_1 = parent2[:first_cut_point] + parent1[first_cut_point:second_cut_point] + parent2[second_cut_point:]
    child_2 = parent1[:first_cut_point] + parent2[first_cut_point:second_cut_point] + parent1[second_cut_point:]

    # Make sure offspring is feasible - CHILD no. 1
    for i in parent1[first_cut_point:second_cut_point]:
        if i not in parent2[first_cut_point:second_cut_point]:
            x = parent2[parent1.index(i)]
            while x in parent1[first_cut_point:second_cut_point]:
                x = parent2[parent1.index(x)]
            child_1[parent2.index(i)] = x
    # CHILD no. 2
    for i in parent2[first_cut_point:second_cut_point]:
        if i not in parent1[first_cut_point:second_cut_point]:
            x = parent1[parent2.index(i)]
            while x in parent2[first_cut_point:second_cut_point]:
                x = parent1[parent2.index(x)]
            child_2[parent1.index(i)] = x

    return child_1, child_2

def crossover_LOX(parent1, parent2, intensity):
    first_cut_point = int(random.random() * len(parent1))
    second_cut_point = first_cut_point + int(intensity * random.random() * (len(parent1) - first_cut_point) )

    child_1 = parent2[:first_cut_point] + parent1[first_cut_point:second_cut_point] + parent2[second_cut_point:]
    child_2 = parent1[:first_cut_point] + parent2[first_cut_point:second_cut_point] + parent1[second_cut_point:]

    # CHILD 1
    missing = [x for x in parent2[first_cut_point:second_cut_point] if not in parent1[first_cut_point:second_cut_point]]
    infeasible = [x for x in parent1[first_cut_point:second_cut_point] if not in parent2[first_cut_point:second_cut_point]]
    for i in range(infeasible):
        child_1[parent2.index(infeasible[i])] = missing[i]
    # CHILD 2
    missing = [x for x in parent1[first_cut_point:second_cut_point] if not in parent2[first_cut_point:second_cut_point]]
    infeasible = [x for x in parent2[first_cut_point:second_cut_point] if not in parent1[first_cut_point:second_cut_point]]
    for i in range(infeasible):
        child_2[parent1.index(infeasible[i])] = missing[i]

    return child_1, child_2

def shift_mutation(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))
            individual[swapped], individual[swapWith] = individual[swapWith], individual[swapped]
    return individual
```

E Scatter Search and Tabu Search in Python

```python
def scatter_search_procedure(list_of_individuals):
    change = 5
    start = random.random()
    for j in range( int(start * 2), len(list_of_individuals), 2):

        for i in range( int(start *4), len(list_of_individuals[j]), 3):
            first = list_of_individuals[j].copy()
            second = list_of_individuals[j].copy()
            second[i-1], second[i] = second[i], second[i-1]

            if test_solution(second) < test_solution(first):
                list_of_individuals[j] = second
            else: list_of_individuals[j] = first

    return list_of_individuals


def tabu_search(individual, n_iterations):

    tabu_list = []

    # get non-Tabu List
    for x in range(n_iterations):
        available_jobs = [job for job in individual if job not in tabu_list]

        # select job
        i = np.random.choice(available_jobs)
        makespan = np.zeros((len(individual)), dtype=int)
        individual.remove(i)
        for j in range(len(individual)+1):
            ind_temp = individual.copy()
            ind_temp.insert(j, i)
            makespan[j] = test_solution(ind_temp)

        index = np.where( makespan == min(makespan))
        individual.insert(index[0][0], i)

        # add job i to tabu list
        tabu_list.append(i)

    return individual
```