Permutation FS

André Couto and Andrei Teixeira

Métodos Heurísticos - MECD - DEI - FCT/UC

Abstract. The abstract should briefly summarize the contents of the paper in 15–250 words.

Keywords: FS · Permutation Flow Show · Makespan

1 Problem description

The Permutational Flow Shop problem, also known simply as "Flow Shop" (FS), is a challenging and essential issue in the field of combinatorial optimization, especially when applied to production scheduling contexts. In modern industry, optimizing the sequence of processing jobs on machines is crucial to improving efficiency, reducing production times and minimizing operating costs, for example, downtime costs, empty conveyors, or producing all the demand in a long space of time. So many strongly correlated problems challenge the ordering of tasks on a series of machines, especially when maintaining a fixed order of processing activities, which can somewhat restrict the optimal production sequence.

In a FS scenario, multiple jobs (also called "jobs" or "activities") must follow the same processing sequence on all the machines involved. This fixed constraint implies that the processing order cannot be changed, making optimization a complex task. In practical terms, this means that once the order of operations has been defined for a set of tasks on the first machine, this order must be maintained for all jobs on the following machines.

Solving the FS efficiently is a continuous search for algorithms that can find optimal or approximate solutions to this problem. Hence the heuristic nature of the problem, as there are many ways of starting from an initial or empty set of jobs and having an idiosyncratic sequence to reach a so-called optimal point. The permutational nature of the FS implies substantial challenges in determining a processing sequence that minimizes the total completion time of all activities on all machines, from the start of the first job on machine 1 to the end of the last job on machine M, also known as "makespan". This critical factor is the fundamental metric in the context of production scheduling for guaranteeing operational efficiency.

Also very common is the minimization of the "total flowtime", which is the sum of the times that each task spends in the production system, from the moment it enters until the moment it is completed. It involves the time spent by each task at each machine or workstation, including waiting times. In other words, makespan is a global measure that focuses on the total time needed to complete all tasks, while total flowtime is a more detailed measure that considers the total time each task spends in the system.

The application of FS covers a wide range of industrial sectors, including manufacturing, logistics, production scheduling, task optimization and even service queues. Solving this problem efficiently offers businesses competitive advantages and has therefore been widely explored for decades. Ultimately, the challenge posed by the FS problem motivates the continuous search for innovative optimization algorithms and strategies that can offer viable and efficient solutions.

Multiple approaches have been proposed to deal with the challenge and an overview will be presented in the next section, including heuristic algorithms, genetic algorithms, local search techniques and methods based on integer programming. These strategies aim to find optimal or admissible solutions, given the computationally challenging nature of a set of machines with fixed constraints. There is no single optimal solution and there are many possible combinations of solutions, because you can work with a different number of machines, with a different number of j activities on each machine, seeking to optimize the desired number and combination of N objective functions. For each of these combinations, you can apply different neighborhood sizes and different methodologies of steps to be taken at each iteration of the algorithm.

The second section looks at the literature specific to the FS problem and gives an overview of how it has been possible to tackle increasingly complex problems, as exemplified above. In the third section, the modeling characteristics of the problem to be studied in this research are given, namely the objective function, the decision space, the limits of interest (in this case the lower limit because it is a minimization problem), the construction rules and the neighborhood structure to be used. In the fourth section, details of the implementation are given, for example, whether it starts with an empty or given solution and if it is given, how it was chosen, the other solutions, the components of the problem and characteristics of the moves, such as representation, move generation and evaluation. This is followed by the fifth implementation section focusing on the characteristics of the code, followed by the evaluation section and the final section discussing the results. XXXXX → voltar no final para descrever tudo o que aconteceu, tipo mais de um modelo, mais de uma avaliação, etc...

2 State of the Art

The FS problem is already quite vast, but it has not stopped being explored, as there are many ways of representing the components, the choice between starting an empty set or not and the options for making movements between jobs in order to minimize makespan.

The work of Minella (2008) [5] emerges as a valuable reference, providing a comprehensive historical summary and concentrating on compiling the main articles to date on the FS problem. The author conducts a detailed analysis of exact, heuristic and metaheuristic methods, presenting a comparison of 23 approaches accompanied by a Pareto evaluation. He also points out that minimizing makespan is the most explored objective, followed by total flowtime and, in third place, machine delay time. Reading Minella (2008) provided a solid basis for a clearer understanding of the FS problem and the first, simpler heuristic applications. Subsequent works advance in complexity and hence highlight the importance of all the literature up to Minella (2008), given that this research aims to address a problem of greater simplicity.

Nagar et al. (1995a) [6] proposed a Branch and Bound (B&B) procedure, which creates subsets to reject non-optimal regions and reduce the search space. The authors approach the FS with 2 machines and the objective of minimizing a weighted combination of 2 criteria: makespan and total flowtime. The algorithm starts building the branching tree with an initial feasible solution and an upper limit, both derived from a greedy heuristic. This method has proved effective in identifying optimal solutions for problems involving two machines and up to 500 jobs. However, it is important to note that this optimized performance occurs under certain strict assumptions and specific data distributions.

Sivrikaya-Serifoglu (1998) [7] compared a model based on the B&B method to a fully heuristic one. The problem of 2 machines and 2 criteria is maintained, but now the objective is the minimization of a weighted combination of the average working time and the duration of the schedule. In the method developed by the author, he seeks to minimize the idle time between jobs on machine 2, a method called "FSH1". When comparing different quantities of jobs - namely 10, 14 and 18 - it was observed that adding more jobs ended up improving the results of the minimization problem, which at first may seem counterintuitive, but which shows the quality of FSH1.

An important paper was Framinan et al. (2002) [3], as it was the first to test the makespan problem in a generalized way, i.e. for M machines. The objective function was still composed of 2 minimization objectives, the weighted sum between makespan and total flowtime. To do this, the author followed a heuristic model for two separate analyses, one with a priori data, i.e. a non-empty solution, and the other with a posteriori data, i.e. a randomly created initial solution. The latter proved to be better than the former, but at a higher computational cost. The heuristic called "NEH" consists of two phases. In the first, the jobs are ordered by decreasing sum of processing times. In the second, the sequence is built by evaluating partial results: for each job, candidate sequences are generated by inserting it into the possible slots of the current sequence until the sequence with the lowest makespan is chosen. The process is repeated until all jobs are sequenced.

Allahverdi (2003)[2] goes one step further and tests a heuristic method for two scenarios: one with 2 machines and the other with M machines. The problem is bicriteria and slightly different, aiming to minimize the weighted sum of makespan and average total flowtime. The heuristic method used consists of three distinct phases. In the first, an initial sequence is generated. In the second, the initial order is adjusted by moving jobs to different positions in the sequence to improve performance metrics. Finally, in the third phase, a definitive pair swapping procedure is carried out on the sequence obtained in the second phase, seeking a further improvement in the solution.

Comparing different models is a challenging task due to the different methodologies adopted. For this reason, Knowles et al. (2006) [4] propose three reliable approaches to this task. The first is based on Pareto dominance relations between sets of solutions. It is possible to rank a particular algorithm against another based on the number of times the ”dominance fronts” come close to the Pareto optimal results. The second approach uses quality indicators, mainly hypervolume IH and epsilon indicators, which were introduced by Zitzler and Thiele (1999) [8] and Zitzler et al. (2003) [9], respectively. Quality indicators generally transform a set of optimal solutions into a real number. Finally, the third approach is based on ”attainment functions”, which provide, in terms of the objective space, the relative frequency with which each region is reached by the approximation set provided by an algorithm.

3 Scope

In the previous section, it was observed how the modeling of the FS problem, which emerged in the 1990s, has evolved from 2 to M machines, initially using B&B methods and moving on to different heuristic methods, with the capacity for problems of tens to hundreds of jobs. In general, the objective functions mostly sought to minimize a weighting between makespan and total flowtime, the metrics most observed in the synthesis carried out by Minella (2008) [5].

Once we know the characteristics of the FS (FS) problem and present different strategies for solving simple problems involving 2 or M machines, we can define the objective of this work: the minimization of a one-dimensional function composed exclusively of ”makespan” in a context of M machines and N jobs.

The aim of this work is to create and test a code that works as a model and houses all the functions needed to run a set of heuristic methods already consolidated in the literature, namely: First Improvement, Best Improvement, Greedy Construction, Greedy Construction with Random Tie-Breaking, Greedy Randomize Adaptive Construction randomness and GRASP.

4 Modeling

In the permutation FS problem, a set of n jobs $J = J_1, J_2, \ldots, J_n$ needs to be processed on a series of m machines $M = M_1, M_2, \ldots, M_m$ in a specific order. Each machine can work in parallel but the sequence of jobs on every machine must be the same. The processing time of job J_i on machine M_j is denoted by $P_{ij}, i = 1, \ldots, n,\ j = 1, \ldots, m$. As each machine can only handle one job at a time, the first job in the sequence begins processing on the first machine. Once completed, it moves to the second machine, allowing the next job to start on the first machine, and so on. If a machine is still occupied when a job is ready to move, it waits until the machine is available, leaving the previous machine empty. The goal is to find a sequence of jobs that minimize the makespan $C_{\max}$, i.e, the time when all jobs have been processed. Figure 1 shows an example of this problem with 3 jobs and 3 machines.

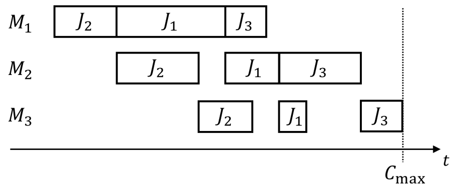

Fig. 1. An example of the permutation FS with $n = 3$ jobs and $m = 3$ machines.

Let $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$[1] denote a permutation of n jobs, where n jobs will be sequenced through m machines (for example, in Figure 1 we have $\pi = \{\pi_1 = J_2, \pi_2 = J_1, \pi_3 = J_3\}$). Let $C(\pi_i, m)$ denote the completion time of job π_i on machine m. The completion time of the permutation FS problem according to the processing sequence $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ is shown as follows:

$$C(\pi_1, 1) = p_{\pi_1,1}, \tag{1}$$
$$C(\pi_i, 1) = C(\pi_{i-1}, 1) + p_{\pi_i,1}, i = 2, \ldots, n \tag{2}$$
$$C(\pi_1, j) = C(\pi_1, j-1) + p_{\pi_1,j}, j = 2, \ldots, m \tag{3}$$
$$C(\pi_i, j) = \max[C(\pi_{i-1}, j), C(\pi_i, j-1)] + p_{\pi_i,j}, i = 2, \ldots, n,\ j = 2, \ldots, m \tag{4}$$

Then the makespan can be defined as

$$C_{\max}(\pi) = C(\pi_n, m) \tag{5}$$

4.1 Objective function

The objective function is the one that minimizes the makespan:

$$\min \quad C_{\max}(\pi) \tag{6}$$

4.2 Decision space

In the FS, the decision space refers to the set of all possible solutions that can be achieved by introducing new jobs and/or making changes to them. It represents the universe of all existing solutions, regardless of whether or not they are visited by any heuristics. A program that performs the FS task could store all the possible combinations and then choose one of them, but this is very computationally expensive the greater the number M of machines and N of jobs.

[1] Note that there will be $n!$ different permutations so the rigorous notation should be $\pi^{(k)} = \{\pi_1^{(k)}, \pi_2^{(k)}, \ldots, \pi_n^{(k)}\}$, $k = 1, \ldots, n!$. However, to simplify the description we will use $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$.

For purely constructive methods, the decision space in this problem is composed of all complete or incomplete solutions, that is, the empty solution, solutions with only one job, solutions with two jobs, and so on, until we reach the admissible solutions with n jobs. Therefore, the cardinality of the decision space is:

$$1 + n + \frac{n!}{(n-2)!} + \frac{n!}{(n-3)!} + \cdots + n! \tag{7}$$

Heuristics consisting solely of local search moves, i.e. starting with a given sequence of jobs, have a decision space equal to the number of possible permutations between them. If we are using a k-opt move during local search, this means that we are swapping the position of k jobs at once. The number of possible permutations for k elements in a set of n elements is given by:

$$P(n,k) = \frac{n!}{(n-k)!} \tag{8}$$

4.3 Lower Bound

Given that the FS is a typical minimization problem, then regardless of the variation of the problem, the bound of interest is always the Lower Bound (LB). It acts as an essential benchmark for evaluating the performance of the algorithms being tested and, by comparing the solutions obtained by the algorithms with the LB, it is possible to measure how close they are to the theoretical optimal solution. In this report, we will work with two different main groups of heuristic methods: local search algorithms and constructive ones.

For the first group, we assume the LB value as the makespan after a swap between two jobs. This decision is justified as an approach aimed at the continuous improvement of solutions. The lower bound represents a conservative estimate of the objective value and is calculated based on partial information about the current solution. By synchronizing the makespan with the lower bound after each change, the local search aims to incorporate a more optimistic view of the solution's performance, directing exploration towards promising regions of the solution space.

This strategy directs the search towards solutions that have the potential to be substantially superior, promoting the intensification of the search towards optimized solutions. The makespan update frequency can be adjusted experimentally, providing flexibility to adapt the strategy to the specific characteristics of the problem and optimizing the performance of the local search algorithm. This approach is in line with the search for efficiency in heuristic methods, seeking to optimize solutions in the context of complex scheduling problems.

So, for local search algorithms, the applied rule is the one it follows: for each swap move i:

$$LB_i = y_i = makespan_i \tag{9}$$

Now, for constructive algorithms, following the work developed in [1] we examine four simple LBs and one slightly more complex for constructive heuristics, which will be all detailed subsequently. The first bound, L_1, is simply obtained by computing the load of each machine and picking the largest:

$$L_1 = \max_{1\leq i\leq m} \left\{ \sum_{j=1}^{n} p_{ij} \right\} \tag{10}$$

L_2 is an improvement of L_1. Let's consider a particular machine i. If i is greater than 1, then machine i cannot start processing its first job until the same has been completed on the machines that come before it in the sequence. Similarly, if $i < m$, which means it's not the last machine in the sequence, then after machine i has finished processing its last job, that job must be processed on the subsequent machines. So the makespan must be at least:

$$P_i = \sum_{j=1}^{n} p_{ij} + \min_{1\leq j\leq n} \left\{ \sum_{r<i} p_{rj} \right\} + \min_{1\leq j\leq n} \left\{ \sum_{r>i} p_{rj} \right\} \tag{11}$$

So, L_2 is:

$$L_2 = \max_{1\leq i\leq m} \{P_i\} \tag{12}$$

The third bound, also a simple one is obtained by computing the total processing time of each job and picking the largest:

$$L_3 = n \max_{1\leq j\leq n} \left\{ \sum_{i=1}^{m} p_{ij} \right\} \tag{13}$$

Just like before, one way to improve L_3 is to use L_4. Consider a particular job j. The improvement involves considering the total processing time of a specific job, j, and the minimum processing time required before and after job j on the first and last machines, respectively because every other job either comes before or after j Thus, the makespan must be at least:

$$Q_j = \sum_{i=1}^{m} p_{ij} + \sum_{s\neq j} \min\{p_{1s}, p_{ms}\} \tag{14}$$

This enables us to increase L_3 to:

$$L_4 = \max_{1\leq i\leq n} \{Q_j\} \tag{15}$$

A fifth LB was studied, and the implementation was carried out based on the following steps: For each machine i, the p_{ij} values are sorted in non-decreasing order and denoted by $\tau_{i1}, \ldots, \tau_{in}$. The minimum time required for machine i to process k jobs is given by σ_{ik}, which denotes the sum $\sum_{k'=1}^{k} \tau_{i,k'}$. Thus, the time at which each machine i finishes processing job k is given by γ_{1k}:

$$\gamma_{ik} = \begin{cases} \sigma_{1k}, & \text{if } i = 1 \\ \min_j \left\{ \sum_{i'=1}^{i} p_{i'j} \right\}, & \text{if } k = 1 \\ \max\{\beta_{ik}^1, \beta_{ik}^2, \beta_{ik}^3, \beta_{ik}^4\}, & \text{if } i \neq 1 \text{ and } k \neq 1 \end{cases} \tag{16}$$

for

$$\beta_{ik}^1 = \sigma_{ik} + \gamma_{i-1,1}, \tag{17}$$

$$\beta_{ik}^2 = \sigma_{i,k-1} + \gamma_{i1}, \tag{18}$$

$$\beta_{ik}^3 = \max_{1 \leq i' \leq i} \left\{ \gamma_{i',k-1} + \min_j \left\{ \sum_{i''=i'}^{i} p_{i''j} \right\} \right\}, \tag{19}$$

$$\beta_{ik}^4 = \max_{1 \leq i' < i} \left\{ \gamma_{i'k} + \min_j \left\{ \sum_{i''=i'+1}^{i} p_{i''j} \right\} \right\} \tag{20}$$

At the end of the procedure, γ_{mn} is a LB (L5).

The evaluation of the algorithms with different LBs is discussed in the Evaluation section.

4.4 Construction algorithms and its rules

In constructive algorithms, it is important to know which rule will be used to add elements to the solution in order to start from the empty solution and complete it. The job can be chosen in an ordered way (by job index or by job time, ascending or descending), randomly or based on a function that tells it which characteristics of the job or the objective function are important for defining what the next job will be.

Once a job has been added, it is crucial to calculate the value of the objective function in order to see whether you are closer or further away from the solution. This information can be stored in memory or not for future decisions. If it is stored and if the value of the objective function does not change in relation to the two jobs, the choice of job for the tie-breaker can follow the same rules as above: randomly, in order or by means of a function.

In this project, for the constructive approach, the first algorithm tested was the Greedy Construction with Random Tie-Breaking (GRTB). The algorithm begins with an empty solution, represented by the Solution class in the model. Initially, no jobs are assigned (the *used* list is empty), meaning all jobs are available to be added (all job IDs are in the *unused* set). The *add_moves()* method generates all possible components (jobs) that can be added to the solution. The selection of components is performed using the *lower_bound_incr_add()* method, which changes according to the LB being tested, as defined by the *_lb_update_add()* function. In the event of a tie, where multiple components result in the same LB increment, the choice between components is made randomly.

This randomness can lead to different solutions each time the algorithm is run, potentially avoiding convergence to a local optimum.

Another algorithm examined in this project was the Greed Randomize Adaptive Construction (GRAC). The distinction between this algorithm and the previous one lies in its adaptive selection process. The algorithm calculates the minimum (*c_min*) and maximum (*c_max*) LB increments from the candidate list. It then defines a threshold (*thresh*) based on a parameter alpha and the range between *c_min* and *c_max*. The alpha parameter controls the greediness of the algorithm; a smaller alpha leads to a more greedy selection. A restricted candidate list (*rcl*) is created, containing only those components whose LB increment is less than or equal to *thresh*. From the *rcl*, the algorithm randomly selects a component to add to the solution. This randomness, similar to the previous algorithm, helps to decrease the likelihood of becoming trapped in a local optimum.

To conclude the application of constructive algorithms, the Greedy Randomized Adaptive Search Procedure (GRASP) was implemented. This algorithm includes a constructive phase where components (jobs) are iteratively added to the solution, a process managed by the *greed_randomize_adaptive_construction()* function. The algorithm then repeats the construction and local search phases multiple times, each iteration introducing a different level of greediness and randomness controlled by the *alpha* parameter. The *grasp()* function oversees this iterative process, keeping track of the best solution found within a specified time budget.

4.5 Local search algorithms and Neighborhood structure

While constructive algorithms have construction rules to work, local search algorithms have a neighborhood structure to define. Since the initial solution is made up of all the elements, it is important to define how to change this order in order to find better combinations.

For example, you could change just one element at a time and pass it through all the possible positions in the job sequence, which is easier to control. The choice of position can be made randomly or according to a rule, and positions may or may not be repeated. Sometimes there is a risk of loops, so some mechanism can be used to avoid repeating paths. It is also possible to change more than one element at a time, which adds complexity to the movements. As with construction rules, the neighborhood structure can be stored in memory or not and, once stored, can and should be saved for later changes.

In this project, for the local search approach, the first algorithm tested was the First Improvement (FI). The algorithm starts with a initial complete solution represented by the data that is tested with all jobs and its respective times. The *local_moves()* function do swaps in jobs and, for each swap, some evaluation of the makespan is done in *step()* function with the help of an iterator *vi* and the difference between two maskepans is captured by *objective_incr_local()* method after each step. This algorithm is very simples and at the moment there is a improvement in the objective function, that here in this project for a local search heuristic is equal to the makespan, it stops.

Another algorithm examined in this project was the Best Improvement (BI) and uses the same functions as FI. The difference is that this algorithm keeps a record of the best move and the corresponding improvement. And after going through all the local moves, the algorithm performs the best move found. Otherwise, when it finds a move that improves the objective function, it performs that move and restarts the local search. Both algorithms use a iterator *vi* to move from one solution to another.

5 Implementation

5.1 Solutions

In our solution, we focused on representing five important elements for the functioning of the functions and, consequently, of the algorithms, namely: (i) the job_ids that were already added to the solution at each step of the heuristic; (ii) the job_ids not yet selected at each step; (iii) the processing time of each machine, to help calculate the makespan; (iv) the value of the makespan itself; and (v) the LB value.

For constructive and mixed methods, the value of the first will always start before the first step and the second will always be full before the first step.

For local search methods, the first element of the solution will always be complete, the second will always be empty and the last two will be equal due to their own construction decision.

5.2 Components

In our modeling, the main choosen component was the job, that was represented by its identification *job_id* in *Component()* class.

Because of this, for the execution of the movements, the component is now divided between the job that is added (*jobid_add*) and the one that is removed (*jobid_rem*). These components are used in all methods. The only thing that changes, however, is their interpretation. For pure constructive methods, adding or removing is literally placing or removing jobs from the sequence, whereas for pure local search, "adding" means moving a job to a new position and "removing" is moving a job from its initial position.

5.3 Local move representation

For local search, the function *generate_swap_moves*() is used. This function reads the job's position within the machine as if it were an index and creates an ordered pair. The first element of the pair is the position where the new job will be added, and the second is where it is removed from: (arrival index, departure index). All ordered pairs are stored in a list, which is later accessed for the execution of job swaps

5.4 Local move generation

For constructive and mixed algorithms, the *add_moves()* function is used to add a job belonging to the unused group to the solution (used). So this jobs component is saved in the *Component* class for subsequent evaluation.

For local search, the *local_moves()* function is used. This function utilizes the auxiliary function *_can_swap()*, which always allows the swapping of two numbers. For each ordered pair present in the list generated by the *_swap_moves()* function, a swap is made, and a new job sequence is generated. This sequence is stored in the *LocalMove* class for subsequent evaluation.

5.5 Local move evaluation

For the constructive and mixed methods, the functions responsible for generating the movement are *_add_job()*, which adds a component to the solution and updates the LB after the movement. Next, the *_lb_update_add()* function updates the previously calculated bound and finally the *lower_bound_incr_add()* function calculates the delta between two bounds to be used later by the algorithms.

For local search, the *step()* function is used. This function recreates each sequence generated by the addition and removal of jobs (swaps between two jobs) and calculates the makespan of each sequence. The makespan, in turn, is used to update the value of the LB with the help of the *_lb_update_local()* function. The difference between the previous LB and the current one is sequentially calculated by the *lower_bound_incr_local()* function. This process ensures efficient job sequencing, makespan calculation, and LB updating.

6 Evaluation and Discussion

6.1 Constructive and mixed models results

To evaluate the performance of the algorithms, we primarily consider the quality of the solution within a limited time frame (in this case, 2 seconds for GRASP). It is evident that this performance is highly dependent on the size of the instance, the structure of the algorithm, the cost of evaluation in the model, and the number of steps the algorithm can take within this time. As such, various tests were conducted with the 5 different LBs mentioned above, for two different instances (20 jobs and 5 machines, and 500 jobs on 20 machines). The folder containing the codes and the report also includes a table with results for different instances analyzed.

To assess the quality of the solution, we used the relative error (RE), which considers the absolute error (the difference between the measured value and the benchmark value) divided by the benchmark value.

$$RE = \frac{absolute\_error}{benchmark} = \frac{measured\_value - benchmark}{benchmark} \tag{21}$$

The benchmark consists on the one found in the first line of the dataset, classified as the best solution found for this instance. For example, for 20_5 dataset, the benchmark is equal to 1278 and for 500_20, 26189.

So, *RE* results are presented in Table 1. It is important to note that these results consider the average of the errors obtained when the code was run 10 times for each algorithm and for each instance.

Table 1. Constructive algorithms quality evaluation

	L1		**L2**		**L3**		**L4**		**L5**	
RE	20_5	500_20	20_5	500_20	20_5	500_20	20_5	500_20	20_5	500_20
GRTB	0.1917	0.1724	0.1533	0.1386	0.0923	0.1403	0.1002	0.1389	0.1345	0.1232
GRAC	0.2387	0.1555	0.1439	0.1570	0.1134	0.1448	0.1494	0.1237	0.1181	0.1368
GRASP	0.0446	0.1534	0.0602	0.1526	0.0391	0.1373	0.0359	0.1463	**0.0148**	**0.1233**

The results obtained under these conditions were quite interesting, especially for GRASP, which provided more accurate solutions for most of the instances observed, primarily with the fifth LB due to its robustness (this better solution is bolded in Table 1). Another observation that is important to note is that the simpler LBs, which are the first and third, had the worst performance across all algorithms.

However, it is not sufficient to only analyze the quality of the solutions. Therefore, we calculated the execution time (ET) of the algorithms under the same circumstances. With the support of Python's time library, it was possible to mark the start and end times of each algorithm's execution:

$$ET = end\_time - begin\_time \tag{22}$$

It is not necessary to present the values for GRASP since it was trained with an execution limit of 2 seconds, meaning the solutions it provided were within this time period. Thus, the results of the other two algorithms are presented in Table 2.

Table 2. Constructive algorithms execution time evaluation

	L1		**L2**		**L3**		**L4**		**L5**	
ET	20_5	500_20	20_5	500_20	20_5	500_20	20_5	500_20	20_5	500_20
GRTB	0.0005	294.7553	0.0156	407.64	0.0062	133.06	0.009	2.6137	0.0042	**0.0663**
GRAC	**0.0004**	295.460	0.0094	229.023	0.0063	118.91	0.0065	2.6763	0.0072	0.0782

The results displayed in these two first tables were obtained using an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 2.00 GHz with 8.00 GB RAM processor. Pypy was used to run results. It is noteworthy that for the largest instance, the first LBs were not the best, as the weight of the evaluation was very high in the calculation of the LB. However, for the fourth and fifth LBs, both algorithms ran quite quickly.

Now that we have the errors and the times it took for the algorithms to run, it is worth noting that in terms of the best solution with the least execution time cost, the fifth LB, followed by the fourth, were the best choices in terms of modeling. Among these, GRASP achieved the best performance.

6.2 Pure local search models results

Similarly to evaluations made for the construction algorithms, the same were used for local search algorithms studied in this report: the relative error RE and the execution time TE, for different number of machine and jobs. Because we assume that LB is equal to the makespan, we had just one LB measure, so to bring more information, five datasets were tested: 20_5, 20_10, 50_10, 100_20 and 500_20.

Knowing that LB benchmark values for each set of data above are, respectively, 1232, 1448, 2907, 5851 and 25922, Table 3 presents RE results for local search algorithms. For both FI and BI algorithms, numbers were calculated using the average of 10 experiments and the results are very interesting in term of variability, from a good result using many machines and many jobs and bad results while decreasing them.

Table 3. Local search algorithms quality evaluation

RE	20_5	20_10	50_10	100_20	500_20
FI	0.1862	0.3008	0.2370	0.2508	0.1389
BI	0.1862	0.3008	0.2370	0.2508	0.1389

Table 4 presents TE results for local search algorithms.

Table 4. Local search algorithms execution time evaluation

ET	20_5	20_10	50_10	100_20	500_20
FI	**0.0163**	0.0309	0.516	7.72	862.8
BI	0.0186	**0.0332**	**0.513**	**7.60**	**818.2**

Between FI and BI, results are exactly the same for RE, but the main difference was in terms of ET, when BI was faster in four of the five tests. So Best Improvement shows a slight superiority compared to First Improvement, especially when you consider the fact that their solution is not the first one chosen, but one of several previously observed.

The results displayed in these two last tables were obtained using an Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz 2.80 GHz with 16.00 GB RAM processor. Pypy was not used to run these results.

7 Conclusion

This work was able to achieve the main objective of creating a code with functions that could solve the Permutation FS's makespan minimization problem. After carrying out some tests to ensure that the code only worked with 20 jobs on 5 machines, it was possible to extend the code to an M number of machines and an N number of jobs.

The set of functions created sought to address two major groups of heuristic algorithms: constructive and mixed, and local search. In the first group, the main concern was the definition and testing of different LBs, since it is important to update their value at each step. The number of tests required the use of Pypy, which helped speed up obtaining the final results. In the second group, since the initial solution is the same as the given sequence of jobs, the concern was to test the algorithms for more different numbers of machines and jobs, and no training was carried out with Pypy. The advantage of this approach showed how the processing time grows exponentially with each new dataset.

In terms of evaluating the results, all the algorithms in the two groups can be compared in terms of the error metric. As the best constructive LB was obtained with L5, the model that best minimized the makespan of the Permutation FS problem was GRASP, with less than 1.5% difference from the benchmark already found in the literature for the smallest dataset and just over 12% for the largest one. The other algorithms, for both the 20_5 dataset and the 500_20 dataset, are GRAC, GRTB, BI and FI. In these two datasets, BI and FI had the lowest errors, while for intermediate values of machines and jobs, errors were up to 3x higher.

It is important to note that this division between model and algorithm is quite significant in problems like this, as it allows for the testing of different algorithms that may already be programmed. Good modeling can lead to better and faster results from the algorithms, depending of course on the weight of the evaluation of the movements. This division inherently leads to a better understanding of the problem, and hence a more efficient resolution.

References

1. Sebastian Cáceres Gelvez, Thu Huong Dang, Adam N. Letchford, On some LBs for the permutation flowshop problem, Computers & Operations Research, Volume 159, 2023, 106320, ISSN 0305-0548, https://doi.org/10.1016/j.cor.2023.106320

2. Allahverdi, A. 2003. The two- and m-machine flowshop scheduling problems with bicriteria of makespan and mean flowtime. Eur. J. Oper. Res. 147 373—396.
3. Framinan, J. M., R. Leisten, R. Ruiz-Usano. 2002. Efficient heuristics for flowshop sequencing with the objectives of makespan and flowtime minimization. Eur. J. Oper. Res. 141 559—569.
4. Knowles, J., L. Thiele, E. Zitzler. 2006. A tutorial on the performance assessment of stochastic multiobjective optimizers. Technical Report 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich. [Revised version.]
5. Minella, G., Ruiz, R., & Ciavotta, M. (2008). A Review and Evaluation of Multi-objective Algorithms for the Flowshop Scheduling Problem. INFORMS Journal on Computing**20**(3), 451—471. https://doi.org/10.1287/1070.0258
6. Nagar, A., S. S. Heragu, J. Haddock. 1995a. A branch-and-bound approach for a two-machine flowshop scheduling problem. J. Oper. Res. Soc. 46, 721-–734.
7. Sivrikaya- ¸Serifo ˘glu, F., G. Ulusoy. 1998. A bicriteria two-machine permutation flowshop problem. Eur. J. Oper. Res. 107 414—430.
8. Zitzler, E., L. Thiele. 1999. Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. IEEE Trans. Evolutionary Comput. 3 257-–271.
9. Zitzler, E., L. Thiele, M. Laumanns, C. M. Fonseca, V. G. da Fonseca. 2003. Performance assessment of multiobjective optimizers: An analysis and review. IEEE Trans. Evolutionary Comput. 7 117–132.