Chapter 7

Scheduling Optimization

In the previous chapters, we described prognostics and health management (PHM), discussed its role and significance in the industry, and dove into its methods and shortcomings. We then emphasized one specific type of prognostic method within the PHM sphere, called data-driven PHM, that, as the name suggests, relies heavily on past and current data sources to estimate the RUL of an asset. In this direction, we considered the difficulties and challenges of data-driven RUL estimation, discussed possible solutions, and showcased a type of explainable AI for PHM in the context of aerospace.

In this and the following final chapter we will change direction. It might, initially, seem that we are distancing ourselves from PHM and AI-based time-series applications. However, that is not the case. This chapter[1] will deal with the next logical step that arises in predictive maintenance (PdM) which is scheduling. After determining the RUL of a set of assets, how can we *optimally* schedule their maintenance to satisfy specific criteria? In the next chapter, we will deal solely with AI-based time-series applications in the medical domain and show that tools developed for industry can lend themselves to other fields as well.

We will start this chapter by introducing the so-called multi-objective flexible job-shop scheduling problem (FJSSP), discuss its inherent difficulties, and present a method that combines global and local search to solve it. Our proposed method yields competitive results to the state-of-the-art. It can be extended and be used on top of an RUL estimation method.

7.1 Introduction

The estimation of the RUL (and any other prognostics measure for that matter) lies at the heart of PHM and PdM. However, determining the RUL is only part of the overall promise

[1]Contents of this chapter are based on [115]; Marios Kefalas, Steffen Limmer, Asteris Apostolidis, Markus Olhofer, Michael Emmerich, and Thomas Bäck. A tabu search-based memetic algorithm for the multi-objective flexible job shop scheduling problem. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO ’19, page 1254–1262, New York, NY, USA, 2019. Association for Computing Machinery.

of PdM (albeit a pivotal one). As its name suggests, PdM uses prognostics for the sake of maintenance planning. Therefore, in principle, having the estimated RUL values, one can plan the maintenance of the assets through scheduling.

Scheduling operations is one of the most important industrial activities, especially in planning and managing manufacturing processes, such as maintenance. Already in the 1950s, this led to the formulation of one of the classical operations research problems [38], the job-shop scheduling problem (JSSP). The JSSP can be described as a set of jobs that must be processed on a set of *pre-determined* machines uninterruptedly, where each job is a sequence of consecutive operations. Each operation requires exactly one machine, and machines are continuously available and can process one operation in a *given* duration. A solution to this problem is a schedule that sequences these operations on the available machines in a way that satisfies predefined performance indicators. A typical performance indicator for the solution is the maximum completion time of all operations, also called the makespan. The usual objective, to find a schedule with minimum length (minimum makespan), was proven to be NP-hard [77] and belongs to the most intractable instances of NP-hard problems [131]. Instead of just the makespan, though, several other performance measures can be used as well, such as the maximum machine workload or the total machine workload [173]. In this case, the problem automatically becomes a multi-objective optimization (MOO) problem, in which a variety of incomparable solutions exist. The set of such solutions, which cannot be improved with respect to one objective without making another objective worse, is called the Pareto set [62].

An extension of the JSSP is the so-called flexible job-shop scheduling problem (FJSSP). The difference between the FJSSP and the JSSP, is that the JSSP has the list of machines on which the operations of the jobs will be processed on, already pre-determined. This is in contrast to the FJSSP, where the machine assignment (or routing subproblem) is also to be determined. Therefore, given that in the FJSSP, we deal with the sequencing of operations and the machine assignments to the operations, it is by nature more complex than the classic JSSP. The FJSSP is, thus, also NP-hard since it is an extension of the NP-hard JSSP. This work will focus on the FJSSP as it resembles more closely dynamic, real-world environments where operations can be processed on different sets of machines.

To deal with the combinatorial complexity, meta-heuristic techniques, such as evolutionary algorithms (EA) [182], particle swarm optimization (PSO) [130] and tabu search (TS) [135] can be used. Specifically, EA is proven to be a successful candidate for multi-objective optimization problems as they are capable of finding a good approximation to the Pareto front [188]. EA comprises a class of direct, probabilistic search and optimization algorithms inspired from the model of organic structure evolution [34, 91]. TS is a metaheuristic, developed by Glover [81], that guides a local heuristic search procedure to explore the solution space beyond local optimality in mathematical optimization by directing (stochastic) local search (LS) methods away from suboptimal regions of the search space. Memetic algorithms combine the two, EA with lo-

cal search operators, and have been widely used in combinatorial optimization [163]. Recently, Yuan et al. [251] have successfully employed memetic algorithms for the multi-objective FJSSP. In this view, our main research question considers the usage of TS as the local search method for a multi-objective evolutionary algorithm (MOEA) to solve the multi-objective FJSSP. The selection of TS as the local search method is based on its versatility (see also Section 7.4), as well as on the lack of sufficient work that uses TS in the context of memetic algorithms for the multi-objective FJSSP.
Our contributions lie in the following:

- Tabu search (TS) is used in two ways: As a local search method, as well as part of the mutation operator.
- Stagnation avoidance based on the hypervolume indicator [61].
- We evaluate our algorithm on the widely used Brandimarte datasets [32] and we compare ourselves to the state-of-the-art algorithms by Yuan et al. [251].

7.2 Problem Definition

An FJSSP instance can be described as a set $\mathcal{N}$ of N jobs that need to be processed on a set $\mathcal{M}$ of M machines. Each job $i \in \{1, ..., N\}$ consists of a tuple of N_i operations O_{ij}, with $j \in \{1, ..., N_i\}$, which have a predetermined execution sequence. This means that for job i to be completed, its N_i operations O_{ij} must be processed in their given order. This is called a precedence constraint. Furthermore, each operation O_{ij} has a predetermined set of machines $\mathcal{M}_{ij} \subseteq \mathcal{M}$ which can process this operation. The processing time p_{ijk} of the process O_{ij} on machine $k \in \mathcal{M}_{ij}$ is also known a-priori. Furthermore, the following assumptions are made:

- All machines are available at time 0.
- All jobs are released at time 0.
- Each machine can process one operation at a time.
- Jobs are independent of each other each other, i.e., there are no precedence constraints among the operations of different jobs.
- No interruption is allowed once a process has started (no pre-emption of operations is allowed).
- The setup times of machines and transfer times of operations are considered negligible.

The FJSSP consists of two subproblems:

1. **The routing subproblem**, i.e., assigning each operation O_{ij} to a machine $k \in \mathcal{M}_{ij}$.
2. **The sequencing subproblem** that determines a sequence of operations on all machines, to obtain a feasible schedule which satisfies predefined objectives.

As mentioned in Section 7.1 the difference between the FJSSP and the JSSP is that the JSSP has the machine assignment (the routing subproblem) already pre-determined, as opposed to the FJSSP, where we not only deal with the sequencing of operations, but also with the machine assignments to the operations. Therefore, as an extension of the NP-hard JSSP, the FJSSP is also NP-hard.
Regarding the flexibility of the problem, there are two classifications based on [108]. These are:

1. Total flexibility, where each operation can be processed by any of the M machines ($\mathcal{M}_{ij} = \mathcal{M}, \forall i \in \{1, .., n\}$ and $j \in \{1, .., N_i\}$).
2. Partial flexibility where some operations can only be processed on a subset of the available M machines in the shop.

Finally, let C_i be the completion time of job i. W_k is the sum of the processing times of operations on machine k. In this work, the three objectives makespan C_{max}, total workload W_T and maximum or critical workload W_{max} are to be minimized. These are defined as follows:

$$\begin{aligned} C_{max} &= \max\{C_i \,|\, i \in \{1, .., N\}\} \;, && (7.1)\\ W_T &= \sum_{k=1}^{M} W_k \;, && (7.2)\\ W_{max} &= \max\{W_k \,|\, k \in \{1, .., M\}\} \;. && (7.3) \end{aligned}$$

7.3 Related Work

Due to its high relevance, the last three decades have seen extensive development of efficient techniques to solve the FJSSP [38]. Between 2010 and 2013, a considerable increase in the number of publications addressing the problem can be observed, with almost 50% of those contributions using multi-objective performance measures [38]. Regarding the latter, the performance measures mostly used are makespan, total workload, and critical workload. Moreover, emphasis has been given to the use of hybrid techniques, i.e., techniques that combine one or more heuristics or metaheuristics [38]. The most common form of hybridization is local search [9]. The term memetic algorithm (MA) is often used synonymously for hybrid evolutionary algorithms [162, 163]. Memetic algorithms combine evolutionary algorithms with local search operators and are widely used in combinatorial optimization. In this view, in [42] the authors introduce a multi-objective memetic algorithm (MA) with an embedded variable neighborhood descent procedure, and in [251] the authors propose new memetic algorithms for the multi-objective flexible job-shop scheduling problem (MO-FJSSP) with the objectives to minimize the makespan, total workload, and critical workload, by adapting the NSGA-II optimizer [51] through a well-designed chromosome encoding/decoding scheme and genetic operators. They also develop a novel local search method based on critical operations, using a hierarchical strategy to handle multiple objectives, emphasizing makespan. To the best of our knowledge, these two papers are the most recent in the field that deal with multiple objectives using a memetic approach. Most researches with a hybrid/memetic structure usually deal with one objective, most often makespan (i.e., see [250, 37]), or other forms of hybridization such as in [246, 164].

7.4 Tabu Search

TS is based on the assumption that problem-solving, to qualify as intelligent, must incorporate adaptive memory and responsive exploration [38].
Local search methods tend to become stuck in suboptimal regions (local optima) or on plateaus where many solutions are equally fit. Tabu search overcomes this pitfall of local search by relaxing its basic rule. First, a worse move can be accepted at each step if no improving move is available. In addition, prohibitions (hence the term tabu) are introduced to discourage the search from coming back to previously visited solutions. These prohibitions are facilitated through a memory structure called the tabu list. In its simplest form, a tabu list is a short-term set of the solutions that have been visited in the recent past, i.e., within less than a certain number of iterations which is called the tabu list size $|T|$ or tenure. In this list, one can alternatively store characteristics or attributes of the forbidden moves [89]. In this approach, we used solutions instead of attributes or moves. Furthermore, these memory structures can be divided into three categories:

1. Short-term: The list of solutions recently considered. If a potential solution appears on the tabu list, it cannot be revisited until it reaches an expiration point, which usually means $|T|$ iterations. This is the approach used in this work.

2. Intermediate-term: Intensification rules which intend to bias the search towards promising areas of the search space.

3. Long-term: Diversification rules that drive the search into new regions.

An aspiration criterion can also be used in tabu search to determine when the tabu restriction can be overridden, thus removing a tabu classification. The aspiration criterion is useful when the tabu list stores solutions' attributes rather than the solutions themselves. We did not use an aspiration criterion in this work since we used entire solutions in the tabu list.

The main decisions to be made, *in general* for TS, are:

- The specification of a neighborhood structure.
- The move attributes (if used).
- The tabu list length (or tenure).
- The aspiration criterion (if used).
- The stopping criteria.

In Algorithm 7.1[2] we show the pseudocode of the TS algorithm used in this work. The algorithm considers the minimization of an objective function f. In detail, given an initial solution x_0 and the tabu tenure T, the TS starts in line 4. Line 6 finds the neighbors of the current candidate, and line 7 checks if the generated neighborhood is empty. This can be the case if, for example, the number of blocks that generate the moves is not larger than 1 [168] (see also Section 7.5.4 for the notion of blocks.) If that is the case, the search stops and returns the best solution found so far. Line 10 checks if all elements of the neighborhood belong in the tabu list, and if they do, the search stops and returns the best solution found so far. On line 13, the best neighbor in the generated neighborhood is found. If the best neighbor is in the tabu list (line 14), we find a new best neighbor, discarding the previous one (lines 15 and 16). On line 21, the tabu list is updated by the best neighbor, and on line 23, it is checked whether the best neighbor is better than the best solution found so far. Lines 26 to 33 check whether there is no progress in discovering a new best solution, in which case the search terminates after a specific number of consecutive tries. We continue like this until the termination criterion of a maximum number of iterations is met.

7.5 A New Memetic Genetic Algorithm

Memetic algorithms combine evolutionary algorithms with local search operators and are widely used in combinatorial optimization [163]. Our algorithmic approach to the multi-objective nature of this problem combines a genetic algorithm (GA) [70] with local search (here, with TS). GA is likely the most widely known type of EA. As such, our approach can be considered a memetic multi-objective algorithm. A high-level outline of the proposed approach, memetic genetic algorithm (TSM), is shown in Algorithm 7.2[3]. Details of each step are given in the following subsections.

[2]Please note that some of the notations in the pseudocode of Algorithm 7.1 differ from the notations in the pseudocode of the original publication [115]. We did this for clarity, as well as for consistency among the chapters of this thesis.

[3]Please note that some of the notations in the pseudocode of Algorithm 7.2 differ from the notations in the pseudocode of the original publication [115]. We did this for clarity, as well as for consistency among the chapters of this thesis.

Algorithm 7.1: Tabu search.

```
Data: $x_0, T, maxIter, noProg, f$ ;    # Initial solution, Tabu tenure, maximum
      number of iterations, maximum number of no progress, objective
      function
Result: $bestSolution$ ;                # Best solution at the end of the search
1  $bestSolution \leftarrow x_0$; $bestSolution\_old \leftarrow x_0$;
2  $bestCandidate \leftarrow bestSolution$; $tabu\_list \leftarrow [\,]$;
3  $counter \leftarrow 0$; $count \leftarrow 0$ ;   # Counter monitoring maximum iterations, Counter
     monitoring no progress
4  while $counter \leq maxIter$ do
5      $counter \leftarrow counter + 1$;
6      $sNeighborhood \leftarrow getNeighbors(bestCandidate)$ ;          # Generating the
        neighborhood from current candidate
7      if not $sNeighborhood$ then
8          Return $bestSolution$;
9      end
10     if all $x$ in $sNeighborhood$ is in $tabu\_list$ then
11         Return $bestSolution$
12     else
13         $bestNeighbor \leftarrow getBestNeighbor(sNeighborhood)$ ;        # Get the best
            neighbor w.r.t. the objective $f$
14         while $bestNeighbor$ in $tabu\_list$ do
15             $sNeighborhood.remove(bestNeighbor)$;
16             $bestNeighbor \leftarrow getBestNeighbor(sNeighborhood)$;
17         end
18     end
19     if $|tabu\_list| = T$ then
20         $tabu\_list.pop(0)$;
21     end
       # Appending best neighbor to the tabu list
22     $tabu\_list \leftarrow tabu\_list\ bestNeighbor$ ;   # Appending best neighbor to the tabu
        list
23     $bestCandidate \leftarrow bestNeighbor$;
24     if $f(bestCandidate) < f(bestSolution)$ then
25         $bestSolution \leftarrow bestCandidate$;
26     end
       # Checking for stagnation
27     if $bestSolution = bestSolution\_old$ then
28         $count \leftarrow count + 1$; $bestSolution\_old \leftarrow bestSolution$;
29     else
30         $count \leftarrow 0$; $bestSolution\_old \leftarrow bestSolution$;
31     end
32     if $count > noProg$ then
33         Return $bestSolution$;
34     end
35 end
```

Algorithm 7.2: TSM.

```
Data: $max\_Solutions, tour, M, m\_Pr, c\_Pr, Pr, hv\_Ref$;       # Maximum number of
      examined solutions, tournament size, population size, mutation
      probability, crossover probability, hypervolume reference point
Result: $pop$ ;                                                  # Best individuals found
1  $pop \leftarrow init\_Pop(M)$ ;                          # Initialize population of size $M$
2  $pop \leftarrow local\_Search(pop, Pr)$ ;          # Apply local search on population with
     probability $Pr$
3  $fit_{pop} \leftarrow eval\_Pop(pop)$ ;                                # Evaluate population
4  $hv_{pop} \leftarrow HVI(pop, hv\_Ref)$ ;    # Hypervolume calculation of the Pareto front
5  $solutions\_Count \leftarrow M$ ;                             # Initializing solution counter
6  while $solutions\_Count \leq max\_Solutions$ do
7      $pop \leftarrow mut\_Correct(pop)$ ;  # Mutate individuals mapping to the same value
         and evaluate them
8      $offspring \leftarrow create\_Offspr(pop, tour, c\_Pr, m\_Pr)$ ;         # Create offspring
         (parent selection($tour$), reproduction($c\_Pr$), mutation($m\_Pr$))
9      $offspring \leftarrow local\_Search(offspring, Pr)$ ;      # Apply local search on the
         offspring with probability $Pr$
10     $fit_{offspring} \leftarrow eval\_Pop(offspring)$ ;                     # Evaluate offspring
11     $pop_{pool} \leftarrow pop : offspring$ ;       # Merge parent population with offspring
         population
12     $pop \leftarrow pop\_Select(pop_{pool}, M)$ ;    # Select new parent population of size $M$
         for next generation
13     $hv_{pop} \leftarrow HVI(pop, hv\_Ref)$ ; # Hypervolume calculation of the Pareto front
14     $stagn\_Check(hv_{pop})$ ;          # Check for stagnation and adjust parameters
         accordingly
15     $solutions\_Count \leftarrow solutions\_Count + M$
16 end
```

7.5.1 Representation

For the chromosome representation we follow the approach presented in [241]. In this representation each individual is a tuple (u, v), where u represents the operation sequences and v the machine assignment for operations. In detail, u is a vector of integers in which the operations of each job is denoted by the corresponding job number. Thus, the $k-$th occurrence of a job number refers to the $k-$th operation in the sequence of this job. For example, the operation sequence u =[$1, 2, 1, 2, 1$] represents the operation sequence [$0_{11}0_{21}0_{12}0_{22}0_{13}$] for jobs 1 and 2 with operation sequences $0_{11}, 0_{12}, 0_{13}$ and $0_{21}, 0_{22}$, respectively. For the machine assignment vector v, each number represents the machine assigned for each operation successively. For instance, for a two job problem with 3 and 2 operations, respectively, and 3 machines, the vector v =[[132][12]], means that 0_{11} is sequenced on machine 1, 0_{12} on machine 3 and operation 0_{13} on machine 2 and similarly for the other job. In Figure 7.1 we see an illustration of the example above, which represents the following operation sequence and their assigned machines:

$(0_{11}, M_1)$, $(0_{21}, M_1)$, $(0_{12}, M_3)$, $(0_{22}, M_2)$, $(0_{13}, M_2)$.

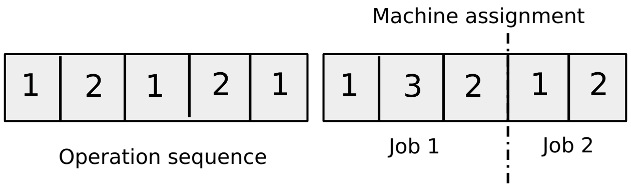

Figure 7.1: Individual representation. The vertical dashed line in the machine assignment vector is used to *visually* distinguish between the machine assignment of the two jobs.

7.5.2 Initialization

For the initialization of the initial population (line 1 in Algorithm 7.2), we follow the procedure introduced in [178] for both the machine assignment sequence and the operation sequence. For the machine assignment, we switch between two assignment approaches. Assignment rule 1 starts from the operation corresponding to the minimum in the processing timetable. Assignment rule 2 permutes the jobs randomly in the timetable before applying the approach by *localization*, described in [108]. This approach considers both the processing times and the workload of the machines, i.e., the sum of the processing times of all the operations assigned to each of the machines. The procedure then consists of finding, for each operation, the machine with the minimum processing time, fixing that particular assignment, and then adding this minimum processing time to every subsequent entry in the same column (machine workload update) [178]. Based on [178] the initialization with the minimum method has a rate of 10% and the initialization with permutation 90%. After the machine assignment is settled, we move on to the operation sequencing. The sequencing of the initial assignments is obtained by a mix of three known dispatching rules:

- Randomly select a job. In this method, a job is randomly selected to be put into the chromosome.
- Most work remaining. In this method, before selecting an operation, the remaining processing times of all jobs are calculated respectively, and the first unselected operation sequence of the job with the highest remaining processing time is placed into the chromosome.
- Most number of operations remaining. In this method, before selecting an operation, the number of succeeding operations of all jobs are calculated respectively, and the first unselected operation sequence of the job with the highest number is placed into the chromosome.

The three dispatching rules above, are used interchangeably with rates 20%, 40%,40%, respectively, following [178].

7.5.3 Parent Selection and Offspring Generation

For the parent selection, we use tournament selection, i.e., the individual for reproduction is chosen to be the one with the smallest makespan among a particular number q of randomly selected individuals. Once the individuals for reproduction have been selected, the crossover and mutation operators are applied to produce the offspring (line 8 in Algorithm 7.2). The crossover operator is applied to pairs of chromosomes, while the mutation operator is applied to single individuals. We distinguish between two kinds of operators:

- Assignment operators, referring to the machine assignment of individuals.
- Sequencing operators, referring to the sequencing of operations of individuals.

Assignment Operators Assignment operators only change the machine assignment of the individuals, i.e., the sequencing of operations is preserved in the offspring. Assignment crossover (or crossover) generates the offspring by exchanging the assignment of a subset of operations between the two parents. On the other hand, assignment mutation only exchanges the assignment of a single operation in a single parent.
In this work, for the machine assignment operators, we used the recombination operator as in [241], originally suggested by Zhang et al. [254] and called multipoint preservative crossover (MPX). This entails the following steps (see also Figure 7.2 for an example):

1. Let $P_1^{\text{mach}}, P_2^{\text{mach}}$ be the operation sequence vectors of the parents P_1, P_2.
2. Generate a random bit-string of 0 and 1 with the same length as the previous selected chromosomes.
3. Exchange the machine assignment of P_1^{mach} to C_2^{mach} (representing the second offspring) and of P_2^{mach} to C_1^{mach} (representing the first offspring) at the same positions where the random bit-string has the *value* 1. Copy the remaining machine assignments of $P_1^{\text{mach}}, P_2^{\text{mach}}$ to $C_1^{\text{mach}}, C_2^{\text{mach}}$ in the same *position.*

For the mutation operator for the assignment, we used a mixture of the approach of [241] and TS. For the approach suggested in [241] we perform the following [241] (see also Figure 7.3 for an example):

- Choose two genes randomly from the machine assignment sequence from the chromosome of an individual, and then change each number in these genes randomly with another machine from the set of capable machines for these two operations.

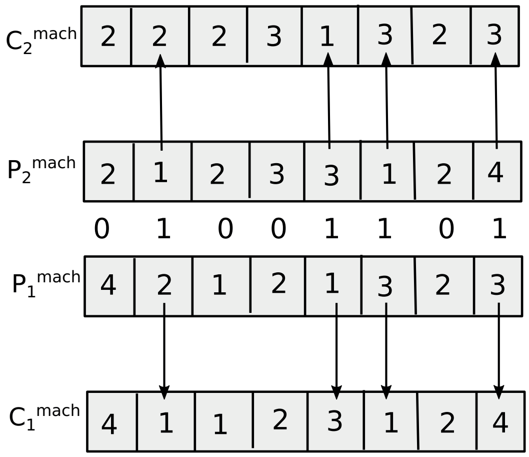

Figure 7.2: MPX crossover operation for the machine assignment sequence. Adapted from [241].

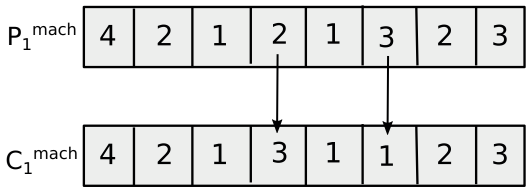

Figure 7.3: Mutation of the machine assignment sequence. "New" machines, $3, 1$, belong in the set of capable machines for the operations that have the same position in the respective operation sequence, as the position of the exchanged machines. Adapted from [241].

In the TS used here, we used as neighborhood structure of the assignment of the individual the use of a random selection of two operations. With these operations in hand, we randomly exchange the machine already assigned to these operations with another one from the set of available machines. We did this 10 times to define the neighborhood around the current seed (i.e., a neighborhood of size 10). For efficiency, we only selected operations that have more than one machine available. We used a neighborhood of steady size equal to 10 and used as the tabu list tenure the closest integer to the squared root of the size of the neighborhood. We also used as a stopping rule 20 repetitions of the TS and a limit of 5 repetitions without improvement. Moreover, we used a 50% probability for the switch between these two mutation methods. The parameters selected here were based on preliminary results.

Sequencing Operators On the other hand, sequencing operators change the sequence of the operations in the parent chromosomes, i.e., the assignment of operations to machines is preserved in the offspring. In applying the sequencing operators, we must respect the precedence constraints among operations of the same job. We followed the suggestion of [241] for both the

crossover and mutation for the sequencing operator.
For the crossover operator of the operation sequence, the authors use an improved version of the precedence operation crossover (IPOX). POX was originally developed by Zhang et al. [252]. This works as follows [241] (see also Figure 7.4 for an example):

1. Let $P_1^{\text{op}}, P_2^{\text{op}}$ be the operation sequence vectors of the parents P_1, P_2. Then we randomly divide the jobs in the two operation sequences into two sets J_1, J_2.

2. We then copy the elements of P_1^{op} that are included in J_1 to C_1^{op} (representing the first offspring) in the same *position* and similarly, copy the elements of P_2^{op} that are included in J_2 to C_2^{op} (representing the second offspring) in the same *position.*

3. Lastly, we copy the elements of P_2^{op} that are included in J_2 to C_1^{op} in the same *order* and copy the elements of P_1^{op} that are included in J_1 to C_2^{op} in the same *order.*

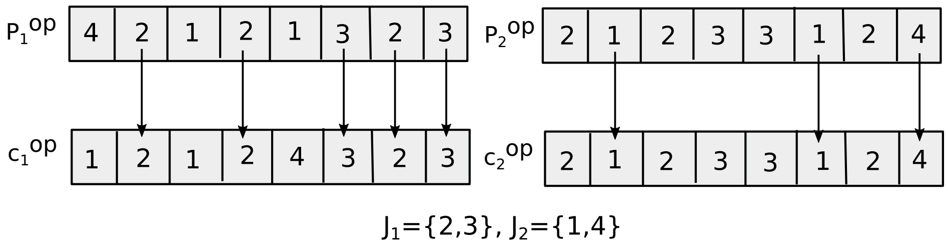

Figure 7.4: IPOX crossover operation for the operation sequence. Adapted from [241].

For the mutation operator of the operation sequence, we perform the following [241] (see also Figure 7.5 for an example):

- Choose a gene randomly from the operation sequence chromosome of an individual, and insert it in a position before a random operation.

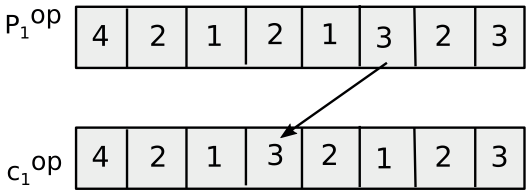

Figure 7.5: Mutation of the operation sequence vector. Adapted from [241].

Finally, non-dominated sorting and the crowding distance operator from NSGA-II [51] are applied for parent selection for the next generation after merging the offspring with the current parent population (i.e., elitism).

7.5.4 Local Search

We decided to hybridize genetic algorithms with TS since in many combinatorial optimization problems, TS can be locally more exhaustive than genetic search [100], as it prevents premature convergence in sub-optimal regions, such as local optima. The local search we decided to use is a TS. For the local search we focused on the minimization of the makespan.
In detail, in every iteration, we applied TS to 10% of the individuals after the genetic operators are applied and before the merging between the parent population and offspring population takes place (line 9 in Algorithm 7.2). Since we focused on minimizing the makespan, we employed the notion of the disjunctive graph. The disjunctive graph is a directed acyclic graph (DAG) used as a compact representation of a schedule [218, 19]. Figure 7.6 shows an example of a disjunctive graph for a schedule of 3 jobs and 3 machines. The operations of every job are nodes of the graph. The solid lines show the precedent constraints between the operations in each job, and the dashed lines (disjunctive arcs) show the sequencing of the operations on the machines. For example, operation $O_{1,1}$ is scheduled before operation $O_{2,2}$ on machine M_1. The numbers outside of each operation-node inform about the processing time of that operation on the specific machine. Nodes S and E are the source and sink nodes, respectively, and are on the graph for completeness reasons. Based on [168] we made adjustments only on particular parts of the critical path of the individual-schedule, called blocks. The critical path is the longest path on the disjunctive graph. Blocks can be considered as the maximal subset of the critical path, which contains operations processed on the same machine. The TS parameters used were: 7 for the tabu list size, 5 for the maximum number of iterations without improvement, and 20 for the maximum overall number of iterations (see also Table 7.2). The neighborhood structure was based on the notion of critical paths and blocks, and as a result, the neighborhood size varied per iteration.

7.5.5 Problem Specific Hypervolume Calculation

In each iteration of TSM, we calculate the hypervolume indicator (HVI) [61] of the Pareto front of the solutions. For more details on the HVI we refer the reader to Section 5.4.3. A possible stagnation of the HVI would mean that TSM is not able to produce solutions that increasingly dominate the objective space. In turn, this means, that there are no solutions which are “better” compared to the non-dominated solutions of the previous generation in at least one of the objectives. To counter this we entered a switch (line 14 in Algorithm 7.2). Suppose the hypervolume is stagnant for more than 3 generations. In that case, we increase the number of the possible solutions entering the local search to 50% (instead of 10%). A problem-specific conservative choice of a reference point is used, as follows: The basic idea is to find a point that will bound from above the Pareto front, and as such, we decided to go with the summation of the predefined processing times on all capable machines of all processes

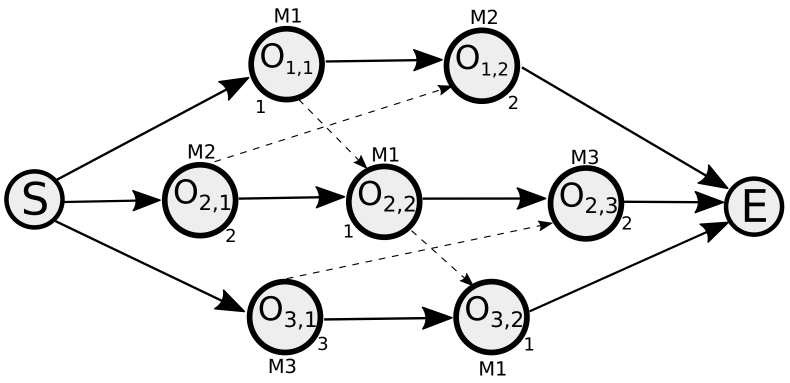

Figure 7.6: Example of a disjunctive graph representation of a schedule with 3 jobs, with 2, 3, and 2 operations respectively and 3 machines. Operations are depicted as nodes. The solid lines show the precedent constraints between the operations in each job and the dashed lines (disjunctive arcs) show the sequencing of the operations on machines. Nodes S and E are the source and sink nodes, respectively, and are on the graph for completeness reasons. Adapted from [251].

overall the jobs. In detail, we used as a reference point the triple (x, x, x), where:

$$x = \sum_{i=1}^{N} \sum_{j=1}^{N_i} \sum_{k \in \mathcal{M}_{ij}} p_{ijk} \quad , \tag{7.4}$$

where p_{ijk} is the process time of 0_{ij} on machine k from its set of machines able to process it. Obviously, this point varies per problem instance, as it is directly related to its input data.

7.5.6 Solution Redundancy

One issue that is common for the FJSSP is solution redundancy. With this, we mean that more than one solution in the decision space maps to the same value in the objective space. That is, the mapping is not injective. There is also the chance that some individuals in the decision space are duplicates after several generations. In this work we tackled the first matter by inserting (line 7 in Algorithm 7.2) an operator which determines the individuals that map to the same objective values. Subsequently, it selects the largest subset of individuals which map to the same value and mutates them through the mutation process we described earlier in Section 7.5.3. The second point, which in essence regards monitoring solution diversity, will be considered in future work.

7.6 Experimental Setup and Results

The main research question is whether the use of memetic algorithms with TS as the local search and as a mutation operator can compete, extend or outperform the state-of-the-art algorithms in the context of the FJSSP. Experiments, datasets, and comparisons to state-of-the-art methods are described in this section.

7.6.1 Data

In this work, we used the famous Brandimarte datasets [32]. The dataset consists of 10 FJSSP problem-instances with the number of jobs ranging from 10 to 20, number of machines ranging from 4 to 15 and number of operations for each job ranging from 3 to 15.
In Table 7.1 we give an overview of the problem instances.

Table 7.1: Brandimarte dataset [32] characteristics.

Dataset	**# Jobs**	**# Machines**	**# Operations**
Mk01	10	6	5-7
Mk02	10	6	5-7
Mk03	15	8	10-10
Mk04	15	8	3-10
Mk05	15	4	5-10
Mk06	10	15	15-15
Mk07	20	5	5-5
Mk08	20	10	10-5
Mk09	20	10	10-15
Mk10	20	15	10-15

7.6.2 Experimental Setup

The experiments[4] were executed on the DAS-4 (Distributed ASCII Computer) [18], with 16 dual quad-core at 2.4GHz with 48GB RAM. Source code has been developed in `Python` Version 3.0. To the best of our knowledge, this is the first research written in Python on multi-objective scheduling.

Benchmark Data and Setup We tested the performance of our algorithm on the 10 benchmark instances Mk01-10 taken from Brandimarte [32] (see also Section 7.6.1). Table 7.2 summarizes the parameter settings of our new algorithm as used for these runs. We ran our algorithm on each benchmark 30 times and merged the results keeping, in the end, the non-dominated

[4]The source code of the experiments can be found at `https://moda.liacs.nl/code/KefalasEtAl2019-Supplement.zip`.

solutions from the merged collection. We used as a termination criterion 500.000 examined solutions, as proposed in [251]. We furthermore used the `DEAP` [71] framework to build TSM.

Table 7.2: Parameter settings of the TSM.

Parameter	**Value**
Population size	300
Crossover probability	0.5
Mutation probability	1
Tournament size	3
Local search probability	0.1
Mutation tabu tenure	3
Mutation tabu search maximum number of *no* progress	5
Mutation tabu search maximum number of iterations	20
Local search tabu tenure	7
Local search maximum number of *no* progress	5
Local search maximum number of iterations	20

Baselines We compared our algorithm to the state-of-the-art algorithms MA-1, MA-2, MA-1-NH, MA-2-NH, MRLS-1, MRLS-2, and NSGA-II variant, from Yuan et al., [251]. Their parameter settings and their solutions and reference set can be found in the same paper. We compared our results with the aggregated results over 30 runs of each of their algorithms, and we did this for each benchmark. We report the results found between our algorithm and the algorithms from Yuan et al. and the hypervolume indicator difference between their reference set (after having added our solutions) and our solutions for each benchmark.
We should note here that, in this work, we did not intend to create a reference Pareto set for each benchmark, as done by Yuan et al. [251], but instead enrich this field by doing additional research on ways of determining new or even better solutions to this problem.

7.6.3 Experimental Results

The results are summarized in Tables 7.3 - 7.8. Specifically, on Tables 7.3 and 7.4 we show our generated solutions, and on Tables 7.5 - 7.8 we compare our solutions to those of [251]. We report our results for each benchmark and the *dominated* solutions of the algorithms we compare to, if they exist, otherwise there is a '–'. On Tables 7.5 - 7.8 the reader can compare the results by identifying on which benchmarks TSM is able to dominate solutions from each of the baselines and on which of these benchmarks can TSM extend the solutions returned by the baselines. Furthermore, we also report on the new solutions found, indicating this with the phrase *Extended by.* Identifying new solutions means determining solutions that increase the diversity of the already-found non-dominated solutions by the other methods. This is important for creating and/or extending reference sets that approximate the Pareto front, as well as identifying strengths of one algorithmic method compared to another. For some benchmarks, specifically for Mk06, Mk07 on Table 7.4 we report only part of the results of TSM due to their large number. We do the same for the new solutions found against MRLS-1 on Mk06 and Mk10 on Table 7.6[5].

New and/or Dominating Solutions From the results on Tables 7.3 and 7.4 the non-dominated solutions from 30 runs of TSM resulted in the following cardinality distribution, Mk01 (14), Mk02 (7), Mk03 (25), Mk04 (23), Mk05 (20), Mk06 (47), Mk07 (50), Mk08 (9), Mk09 (30), Mk10(44). Moreover, on Tables 7.5 - 7.8 we see that TSM performs well against both MRLS-1 and MRLS-2 in all instances by dominating some of their solutions or determining new points that extend the Pareto front found by MRLS-1 and MRLS-2. Specifically, on Tables 7.5 and 7.6 we that TSM partially dominates the solutions of MRLS-1 in 8 out of 10 benchmarks (Mk01, Mk02, Mk03, Mk04, Mk05, Mk07, Mk08, Mk09) and finds new Pareto solutions in 2 out of 10 benchmarks (Mk06 and Mk10). Regarding MRLS-2, from Tables 7.7 and 7.8 we see that TSM partially dominates the solutions returned by MRLS-2 in 7 out of 10 cases (Mk01, Mk03, Mk04, Mk05, Mk06, Mk08, Mk10) and identifies new solutions in 3 out of 10 cases (Mk02, Mk07, Mk09). Furthermore, we see that for instance Mk06, TSM identifies a new solution to the results returned by MA-1, MA-2, and NSGA-II on Tables 7.6 and 7.8, respectively. However in most cases MA-1, MA-2, NSGA-II dominate our solutions. Similarly, we did not find any new or dominating solutions compared to the solutions returned by MA-1-NH and MA-2-NH in any case.

Hypervolume Indicator We, furthermore, computed the difference between the HVI of the reference set (*after having added our solutions*) and the HVI of the non-dominated solutions found by *all* algorithms (including TSM) in *all* 30 runs, for each benchmark. We normalized the reference sets and the obtained sets by using the nadir point of the reference set multiplying by 1.1 [166]. The results can be seen in Table 7.9. It is clear from the results that algorithms MA-1 and MA-2 show the best performance. Nevertheless, our approach gives (see TSM results in bold), when compared to MRLS-1 and MRLS-2, competitive results on Mk01, better results in Mk03 and Mk04, Mk05 (compared to MRLS-2), Mk07, Mk08 and Mk09. In Mk08, our algorithm is able to determine the reference set.

In Table 7.10 we present the median of the difference between the HVI of the reference set

[5]The full list of solutions is available at `https://moda.liacs.nl/code/KefalasEtAl2019-Supplement.zip`

Table 7.3: Solutions identified by TSM for benchmarks Mk01-Mk05.

Algorithm	**Mk01**	**Mk02**	**Mk03**	**Mk04**	**Mk05**
TSM	(40, 168, 37), (41, 167, 36), (41, 162, 39), (41, 165, 37), (42, 159, 39), (42, 160, 38), (40, 174, 36), (41, 164, 38), (43, 155, 40), (42, 163, 37), (42, 165, 36), (43, 158, 39), (44, 154, 40), (46, 153, 42)	(29, 150, 26), (29, 144, 28), (29, 145, 27), (30, 143, 29), (31, 141, 31), (31, 142, 30), (33, 140, 33)	(204, 864, 204), (206, 857, 204), (210, 855, 204), (213, 852, 204), (215, 849, 213), (216, 848, 213), (222, 847, 222), (223, 847, 213), (224, 851, 204), (226, 843, 222), (230, 842, 222), (234, 846, 213), (237, 844, 213), (240, 850, 204), (246, 841, 231), (247, 849, 210), (248, 848, 210), (249, 840, 249), (256, 838, 249), (256, 840, 222), (262, 838, 231), (274, 839, 222), (275, 838, 222), (282, 837, 231), (297, 843, 221)	(68, 355, 68), (68, 376, 60), (69, 360, 60), (69, 351, 63), (71, 353, 62), (72, 347, 66), (72, 357, 61), (73, 342, 72), (73, 348, 63), (75, 344, 66), (75, 347, 65), (77, 340, 72), (78, 337, 78), (79, 343, 67), (84, 334, 84), (90, 331, 90), (98, 330, 98), (106, 329, 106), (114, 328, 114), (122, 327, 122), (130, 326, 130), (138, 325, 138), (146, 324, 146)	(174, 687, 173), (176, 686, 173), (177, 685, 173), (178, 683, 175), (178, 682, 176), (179, 684, 174), (179, 680, 179), (180, 682, 175), (180, 681, 178), (181, 684, 173), (181, 679, 179), (181, 680, 178), (182, 683, 173), (182, 687, 172), (183, 677, 183), (185, 676, 185), (191, 675, 191), (197, 674, 197), (203, 673, 203), (209, 672, 209)

(*after having added our solutions*) and the HVI of each algorithm on each benchmark over 30 trials. We used the Wilcoxon rank sum test (Mann-Whitney U test), with a significance level of $\alpha = 0.01$, to see whether the hypervolume difference values obtained with the TSM strategy are significantly different than those obtained with one of the other strategies[6]. We selected this test to take into account the non-normality of the data and the independence between the samples. We also used the Bonferroni correction, which means that for each individual test, the significance level α is divided by the number of tests per test instance. For us, this is 7 and, thus, $\alpha \approx 0.001$. We made the Bonferroni correction, to counteract the increased likelihood of incorrectly rejecting the null hypothesis (type I error or false positive), which exists because of the *multiple* hypotheses tested *at once*. In Table 7.10 the superscripts in the bold TSM results, indicate from which of the other 7 algorithms the TSM returned significantly lower hypervolume difference values. From the table, we see that TSM performed significantly better than both MRLS-1 and MRLS-2 on Mk03, Mk04, and Mk08[7].

Finally, in Figure 7.7 we report the average execution time (in seconds) that TSM took on each of the 10 benchmarks.

[6]This means that the null hypothesis is that the hypervolume difference values obtained with the TSM strategy come from the same distribution as those obtained with one of the other strategies.

[7]Note that in the original publication [115], it was erroneously noted that TSM performed significantly better in Mk09 as well. This error has been corrected and noted in the remainder of the chapter and in Table 7.10 as well.

Table 7.4: Solutions identified by TSM for benchmarks Mk06-Mk10.

Algorithm	**Mk06**	**Mk07**	**Mk08**	**Mk09**	**Mk10**
	(91, 474, 57),	(144, 690, 144),			(300, 2157, 224),
	(91, 453, 66),	(148, 685, 144),			(311, 2128, 256),
	(92, 436, 60),	(150, 690, 143),			(313, 2190, 220),
	(93, 480, 54),	(150, 684, 149),			(313, 2127, 242),
	(95, 456, 55),	(153, 680, 150),			(313, 2132, 241),
	(96, 434, 60),	(153, 683, 147),			(314, 2133, 230),
	(96, 428, 61),	(154, 673, 150),			(315, 2156, 220),
	(99, 432, 60),	(156, 682, 147),		(369, 2711, 328),	(316, 2128, 220),
	(99, 427, 71),	(157, 683, 145),		(372, 2493, 310),	(317, 2127, 211),
	(100, 476, 54),	(157, 691, 142),		(373, 2452, 299),	(318, 2113, 239),
	(100, 450, 57),	(158, 670, 156),		(377, 2415, 300),	(318, 2125, 230),
	(102, 455, 54),	(158, 679, 145),		(379, 2396, 299),	(321, 2101, 259),
	(103, 452, 54),	(158, 690, 140),		(386, 2375, 320),	(322, 2122, 223),
	(103, 446, 59),	(160, 675, 147),		(389, 2387, 299),	(323, 2113, 224),
	(104, 451, 54),	(160, 671, 150),		(393, 2365, 315),	(324, 2112, 217),
	(105, 449, 55),	(160, 677, 144),		(394, 2376, 299),	(325, 2094, 220),
	(106, 420, 74),	(161, 673, 144),		(396, 2368, 299),	(326, 2090, 221),
	(107, 423, 63),	(162, 668, 156),		(399, 2364, 307),	(331, 2109, 214),
	(108, 421, 69),	(163, 666, 162),	(523, 2524, 523),	(401, 2336, 331),	(332, 2171, 210),
	(108, 447, 56),	(163, 667, 157),	(524, 2519, 524),	(401, 2364, 299),	(333, 2137, 210),
	(109, 421, 66),	(166, 664, 157),	(533, 2514, 533),	(410, 2340, 316),	(335, 2106, 218),
	(109, 441, 59),	· · ·	(542, 2509, 542),	(414, 2361, 315),	(336, 2087, 233),
TSM	(110, 442, 55),	(172, 687, 143),	(551, 2504, 551),	(419, 2352, 304),	(336, 2112, 208),
	(110, 421, 60),	(174, 688, 140),	(560, 2499, 560),	(424, 2361, 299),	(339, 2082, 229),
	(112, 417, 67),	(175, 686, 140),	(569, 2494, 569),	(427, 2359, 300),	(343, 2109, 213),
	(113, 411, 74),	(176, 660, 174),	(578, 2489, 578),	(427, 2360, 299),	(345, 2107, 216),
	(115, 414, 68),	(178, 668, 152),	(587, 2484, 587)	(432, 2341, 299),	(353, 2105, 215),
	(115, 415, 63),	(179, 657, 170),		(448, 2331, 328),	(357, 2082, 220),
	(122, 439, 56),	(182, 684, 143),		(468, 2322, 307),	(358, 2111, 212),
	· · ·	(185, 665, 156),		(493, 2339, 299),	(359, 2069, 253),
	(129, 437, 57),	(191, 660, 169),		(507, 2338, 303),	(359, 2091, 208),
	(130, 434, 58),	(192, 661, 162),		(523, 2338, 299),	(362, 2080, 250),
	(131, 440, 55),	(193, 659, 162),		(534, 2335, 301),	(362, 2081, 236),
	(131, 413, 63),	(194, 655, 190),		(543, 2311, 320),	(363, 2057, 242),
	(136, 449, 54),	(197, 655, 176),		(559, 2321, 310),	(364, 2054, 210),
	(139, 407, 69),	(206, 653, 202),		(563, 2335, 299),	(364, 2128, 205),
	(140, 444, 54),	(220, 658, 166),		(567, 2327, 299)	(368, 2115, 206),
	(141, 438, 56),	(221, 654, 190),			(390, 2092, 205),
	(141, 439, 55),	(227, 653, 187),			(397, 2050, 248),
	(142, 411, 65),	(241, 652, 209),			(416, 2084, 206),
	(143, 402, 82),	(244, 657, 166),			(427, 2127, 204),
	(144, 406, 67),	(265, 651, 209),			(452, 2082, 206),
	(154, 434, 54),	(268, 651, 205),			(460, 2078, 209),
	(158, 473, 53)	(277, 652, 202)			(515, 2132, 202)

Table 7.5: *Dominated* solutions and *extended* solutions by TSM on the MA-1, MA-2, MA-1-NH, MA-2-NH, MRLS-1 algorithms on the Mk01-Mk05 benchmark datasets. Shown solutions are Pareto *dominated* solutions by TSM. "-" indicates no Pareto dominated solutions. "Extended by" marks TSM solutions that extend the solutions returned by the algorithms in [251] on the benchmark data.

Algorithm	**Mk01**	**Mk02**	**Mk03**	**Mk04**	**Mk05**
MA-1	-	-	-	-	-
MA-2	-	-	-	-	-
MA-1-NH	-	-	-	-	-
MA-2-NH	-	-	-	-	-
MRLS-1	(43, 163, 37), (43, 156, 40), (42, 166, 36), (46, 153, 46)	(33, 142, 30)	(212, 932, 204), (204, 956, 204), (207, 947, 204)	(79, 338, 78), (84, 335, 84), (78, 339, 78)	(186, 676, 186), (192, 675, 192), (181, 679, 181)

Table 7.6: *Dominated* solutions and *extended* solutions by TSM on the MA-1, MA-2, MA-1-NH, MA-2-NH, MRLS-1 algorithms on the Mk06-Mk10 benchmark datasets. Shown solutions are Pareto *dominated* solutions by TSM. "-" indicates no Pareto dominated solutions. "Extended by" marks TSM solutions that extend the solutions returned by the algorithms in [251] on the benchmark data.

Algorithm	**Mk06**	**Mk07**	**Mk08**	**Mk09**	**Mk10**
MA-1	Extended by: (158, 473, 53)	-	-	-	-
MA-2	Extended by: (158, 473, 53)	-	-	-	-
MA-1-NH	-	-	-	-	-
MA-2-NH	-	-	-	-	-
MRLS-1	Extended by: (91, 474, 57), (92, 436, 60), (93, 480, 54), (95, 456, 55), ... (139, 407, 69), (140, 444, 54), (141, 438, 56), (141, 439, 55), (142, 411, 65), (144, 406, 67), (154, 434, 54), (158, 473, 53)	(157, 673, 150), (150, 688, 144), (149, 689, 144)	(555, 2531, 542), (523, 2542, 523), (524, 2541, 524), (533, 2532, 533), (530, 2540, 524)	(387, 2382, 320)	Extended by: (300, 2157, 224), (313, 2190, 220), (314, 2133, 230), (315, 2156, 220), (316, 2128, 220), (317, 2127, 211), (318, 2113, 239), (318, 2125, 230), (322, 2122, 223), (323, 2113, 224), (324, 2112, 217), ... (390, 2092, 205), (397, 2050, 248), (416, 2084, 206), (427, 2127, 204), (452, 2082, 206), (460, 2078, 209), (515, 2132, 202)

Table 7.7: *Dominated* solutions and *extended* solutions by TSM on the MRLS-2 and NSGA-II variant algorithms on the Mk01-Mk05 benchmark datasets. Shown solutions are Pareto *dominated* solutions by TSM. "-" indicates no Pareto dominated solutions. "Extended by" marks TSM solutions that extend the solutions returned by the algorithms in [251] on the benchmark data.

Algorithm	Mk01	Mk02	Mk03	Mk04	Mk05
MRLS-2	(47, 153, 42), (48, 165, 36), (43, 163, 37), (46, 166, 36), (46, 153, 46)	Extended by: (33, 140, 33)	(204, 931, 204)	(78, 339, 78), (84, 336, 84), (72, 360, 61)	(198, 674, 198), (186, 676, 186)
NSGA-II	-	-	-	-	-

7.7 Discussions and Conclusions

The flexible job-shop scheduling problem (FJSSP) is an NP-hard optimization problem, which can be considered a subsequent step to the RUL estimation and is a pivotal part of PHM. In this chapter, we presented a memetic multi-objective algorithm for the FJSSP, called tabu search memetic (TSM) algorithm, which jointly minimizes the makespan, the total machine workload and the critical machine workload. Our main contributions lie in the usage of tabu search (TS) as the local search method and mutation operator and the employment of the hypervolume indicator (HVI) as a stagnation avoidance switch. Although there have been many publications on the FJSSP, to our knowledge, there has not been any work that uses TS in the context of multi-objective FJSSP. We evaluated our approach against the reference solutions set returned by the state-of-the-art algorithms on the multi-objective FJJSP by Yuan et al. [251] on the widely used Brandimarte dataset [32].

The experimental results show that TSM can outperform 2/7 competing algorithms on the majority of the benchmark datasets by dominating certain of their solutions. Furthermore, TSM is able to extend the set of the Pareto solutions returned from 5/7 competing methods on 5/10 benchmark datasets. We, further, extended the reference sets per benchmark introduced in [251] by adding our solutions to them and showed that TSM can better approximate the extended reference set compared to 2/7 competing algorithms on 5/10 benchmarks. Finally, the results showed that TSM could approximate this extended reference set statistically significantly better than 2/7 algorithms on 3/10 benchmarks.

In summary, the results suggest that the TSM algorithm is an interesting alternative to the state-of-the-art algorithms introduced by Yuan et al. [251], in terms of quality. More generally, our work shows that combining local search with global search can have a significant positive impact for heuristic solvers for the FJSSP. A limitation of this study lies in the small number of tested benchmarks. Besides, TSM is made available as an open-source Python implementation, making multi-objective FJSSP available to the big community of Python programmers.

Finally, this method can, in principle, be extended to account for RUL information. For

Table 7.8: *Dominated* solutions and *extended* solutions by TSM on the MRLS-2 and NSGA-II variant algorithms on the Mk06-Mk10 benchmark datasets. Shown solutions are Pareto *dominated* solutions by TSM. "-" indicates no Pareto dominated solutions. "Extended by" marks TSM solutions that extend the solutions returned by the algorithms in [251] on the benchmark data.

Algorithm	Mk06	Mk07	Mk08	Mk09	Mk10
MRLS-2	(92, 439, 62), (92, 477, 61), (94, 475, 61), (99, 486, 60)	Extended by: (150, 690, 143), (157, 691, 142), (158, 679, 145), (158, 690, 140), (160, 675, 147), (160, 671, 150), (160, 677, 144), (161, 673, 144), (166, 670, 150), (168, 689, 142), (169, 688, 141), (169, 663, 162), (170, 662, 157), (171, 661, 169), (172, 667, 156), (172, 687, 143), (174, 688, 140), (175, 686, 140), (176, 660, 174), (178, 668, 152), (179, 657, 170), (182, 684, 143), (185, 665, 156), (191, 660, 169), (192, 661, 162), (193, 659, 162), (194, 655, 190), (197, 655, 176), (206, 653, 202), (220, 658, 166), (221, 654, 190), (227, 653, 187), (241, 652, 209), (244, 657, 166), (265, 651, 209), (268, 651, 205), (277, 652, 202)	(560, 2528, 560), (523, 2537, 523), (524, 2532, 524), (543, 2530, 542), (569, 2525, 569)	Extended by: (373, 2452, 299), (377, 2415, 300), (379, 2396, 299), (386, 2375, 320), (389, 2387, 299), (393, 2365, 315), (394, 2376, 299), (396, 2368, 299), (399, 2364, 307), (401, 2336, 331), (401, 2364, 299), (410, 2340, 316), (414, 2361, 315), (419, 2352, 304), (424, 2361, 299), (427, 2359, 300), (427, 2360, 299), (432, 2341, 299), (448, 2331, 328), (468, 2322, 307), (493, 2339, 299), (507, 2338, 303), (523, 2338, 299), (534, 2335, 301), (543, 2311, 320), (559, 2321, 310), (563, 2335, 299), (567, 2327, 299)	(330, 2100, 239)
NSGA-II	Extended by: (158, 473, 53)	-	-	-	-

example, knowing the time-to-maintenance of a set of assets, one can introduce these assets in the job set to the job-shop at the appropriate times each (e.g., the RUL) and adjust the routing and the sequencing operations appropriately.

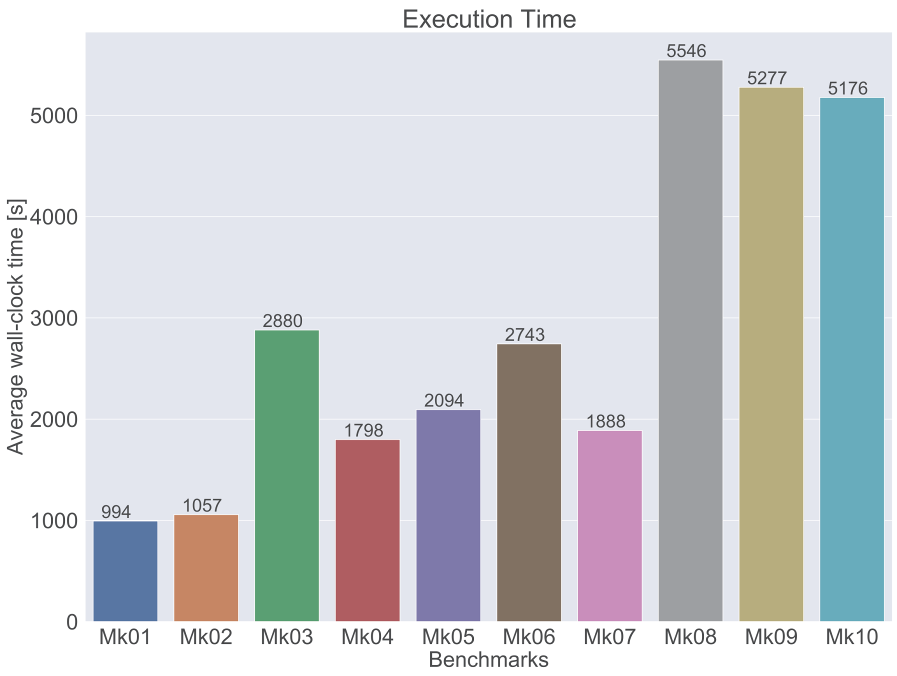

Figure 7.7: Average (over 30 runs) TSM execution wall-clock time (in seconds) per benchmark.

Table 7.9: HVI difference from reference set (lower is better). TSM solutions that outperform a competing algorithm from [251] are shown in **bold**. Superscript numbers indicate the index of the outperformed algorithm.

Algorithms	**Mk01**	**Mk02**	**Mk03**	**Mk04**	**Mk05**	**Mk06**	**Mk07**	**Mk08**	**Mk09**	**Mk10**
TSM	0.000263	0.002660	**0.002046**^{5,6}	**0.004647**^{5,6}	**0.000247**^{5}	0.056141	**0.002001**^{5,6}	**0.000000**^{5,6}	**3.509746e-03**^{5,6}	0.021632
[1] MA-1	0.000000	0.000000	0.000000	0.000000	0.000000	0.003675	0.000000	0.000000	5.414126e-07	0.000915
[2] MA-2	0.000000	0.000000	0.000000	0.000000	0.000000	0.002644	0.000000	0.000000	1.598030e-09	0.000412
[3] MA-1-NH	0.000040	0.000043	0.000000	0.000189	0.000000	0.002143	0.000000	0.000000	2.404332e-05	0.002283
[4] MA-2-NH	0.000000	0.000047	0.000000	0.000076	0.000000	0.002596	0.000000	0.000000	2.520381e-05	0.002037
[5] MRLS-1	0.000131	0.000226	0.016691	0.004766	0.000266	0.042805	0.003620	0.000409	9.736424e-03	0.019498
[6] MRLS-2	0.000113	0.000217	0.020501	0.004812	0.000219	0.043104	0.003403	0.000355	9.138569e-03	0.019221
[7] NSGA-II	0.000040	0.000138	0.000000	0.000564	0.000000	0.007663	0.000000	0.000000	3.174112e-04	0.003024

Table 7.10: Median of the HVI difference from reference set. TSM solutions that *statistically significantly* outperform a competing algorithm from [251] are shown in **bold**. Superscript numbers indicate the index of the *significantly* outperformed algorithm. Note that in the original publication [115], it was erroneously noted that TSM performed significantly better in Mk09 as well. This error has been corrected in the Table below.

Algorithms	**Mk01**	**Mk02**	**Mk03**	**Mk04**	**Mk05**	**Mk06**	**Mk07**	**Mk08**	**Mk09**	**Mk10**
TSM	0.000998	0.003774	**0.003641**^{5,6}	**0.007081**^{5,6}	0.000772	0.061426	0.005147	**0.000020**^{5,6}	0.010655	0.022215
[1] MA-1	0.000000	0.000050	0.000000	0.000295	0.000000	0.005438	0.000000	0.000000	0.000005	0.001809
[2] MA-2	0.000000	0.000045	0.000000	0.000342	0.000000	0.005230	0.000000	0.000000	0.000003	0.001432
[3] MA-1-NH	0.000042	0.000185	0.000000	0.000400	0.000000	0.007017	0.000000	0.000000	0.000112	0.003695
[4] MA-2-NH	0.000040	0.000142	0.000000	0.000381	0.000000	0.006897	0.000000	0.000000	0.000110	0.003174
[5] MRLS-1	0.000504	0.001232	0.020065	0.010253	0.000375	0.049451	0.005101	0.000571	0.010895	0.020479
[6] MRLS-2	0.000420	0.001167	0.021041	0.009804	0.000326	0.048765	0.004778	0.000521	0.010474	0.020179
[7] NSGA-II	0.000108	0.000905	0.000000	0.001873	0.000105	0.010133	0.000715	0.000000	0.000359	0.004454

.