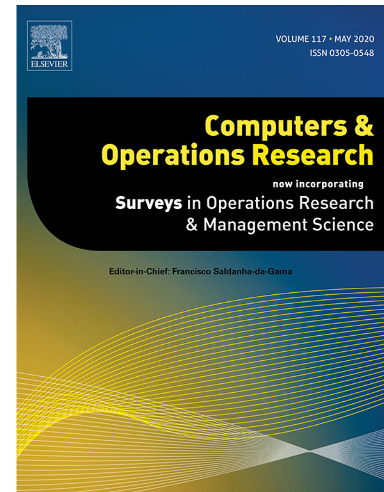
Journal Pre-proofs



Machine Learning based Algorithm Selection and Genetic Algorithms for serial-batch scheduling

Aykut Uzunoglu, Christian Gahm, Axel Tuma

PII: S0305-0548(24)00299-5
DOI: https://doi.org/10.1016/j.cor.2024.106827
Reference: CAOR 106827

To appear in: *Computers and Operations Research*

Received Date: 29 December 2023
Revised Date: 11 June 2024
Accepted Date: 27 August 2024

Please cite this article as: A. Uzunoglu, C. Gahm, A. Tuma, Machine Learning based Algorithm Selection and Genetic Algorithms for serial-batch scheduling, *Computers and Operations Research* (2024), doi: https://doi.org/10.1016/j.cor.2024.106827

This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2024 The Author(s). Published by Elsevier Ltd.

Journal Pre-proofs

Machine Learning based Algorithm Selection and Genetic Algorithms for serial-batch scheduling

Aykut Uzunoglu[a*], Christian Gahm[a], Axel Tuma[a]

[a] Chair of Business Administration, Production & Supply Chain Management, Augsburg University, D-86135 Augsburg, Germany

* Corresponding author:
E-mail: aykut.uzunoglu@wiwi.uni-augsburg.de; Phone: +49-821-598-4041

Abstract:

Whenever combinatorial optimization problems cannot be solved by exact solution methods in reasonable time, tailor-made algorithms (heuristics, meta-heuristics) are developed. Often, these heuristics exploit structural properties and perform well on selected subsets of the problem space. For example, this is how the two best-known construction heuristics solve the scheduling problem investigated in this study (i.e., the scheduling of parallel serial-batch processing machines with incompatible job families, restricted batch capacities, arbitrary batch capacity demands, and sequence-dependent setup times). However, when the properties change, the performance of one algorithm might decrease, and another algorithm might have been the better choice. To resolve this issue, we propose using Machine Learning to exploit the strengths of different algorithms and to select the probably best-performing algorithm for each problem instance individually. To that, we investigate a variety of methods from the "learning-to-rank" literature and propose several adaptations. Furthermore, because there is no algorithm for the considered scheduling problem that is capable to explore the entire solution space, we developed two Genetic Algorithms for the improvement of initial solutions computed by the selected algorithms. Here, we put special emphasis on ensuring that the solution representation (encoding) reflects the entire solution space and that the operators (e.g., for recombination and mutation) are appropriate to explore and exploit this space completely. Our computational experiments show an average increase of 39.19% in solution quality.

Keywords: serial batching, total weighted tardiness, algorithm selection, genetic algorithm, machine learning

Journal Pre-proofs

1 Introduction

Whenever combinatorial optimization problems of industrial scale have to be solved, exact solution methods (algorithms in the narrower sense) need prohibitively long computation times due to the "combinatorial explosion" of the solution space (i.e., the set of all feasible solutions). Therefore, experts design heuristics or meta-heuristics (algorithms in a broader sense) incorporating knowledge about the specific problem to leverage prior information about structures or properties of good solutions. These algorithms return good results for the subset of problem instances they were designed for but fail to find good results for other subsets of the problem space comprising "all" problem instances of a specific problem. This is why, for most combinatorial problems, a variety of algorithms can be found in the literature. This situation also holds for the application case studied in this paper, a serial-batch scheduling problem often found in the metal processing industry (cf., e.g., Helo et al., 2019). It involves the decisions to group metal pieces into batches and to schedule the resulting batches, and can be summarized as the scheduling of parallel serial-batch processing machines with incompatible job families, restricted batch capacities, arbitrary batch capacity demands, and sequence-dependent setup times (abbreviated PSBIJF and classified as "P | if, crJ, sFS, sb | wT, F, Lex"-problem according to Wahl et al., 2023). To solve the PSBIJF, several (learning-augmented) construction heuristics (LACH) can be found in literature (most recent ones in Uzunoglu et al., 2023a and Uzunoglu et al., 2023b), for which none of them clearly dominates the other.

The dominance of algorithms on a narrow subset of problem instances is more than a mere empirical observation but rather an underlying property of combinatorial optimization problems and their algorithms. In some sense, researchers refer to this phenomenon as the "No Free Lunch theorem in optimization and search" (NFL; see Wolpert & Macready, 1997). Like other "negative" theoretical results, such as Gödel's incompleteness theorem in mathematics or Arrow's impossibility theorem in social choice theory, NFL shows the theoretical boundaries of optimization, namely that no general algorithm can dominate other algorithms on the whole problem space or even all combinatorial optimization problems. Since NFL is not the focus of our paper, we refer the reader to Droste et al. (2002) for a more practical reframing of NFL, Ho & Pepyne (2002) for a version of finite and discrete input and output spaces, and Adam et al. (2019) for a review. Apart from the theoretical discussion, the NFL points towards an essential question for optimization problems: which algorithm to use when there is no clear winner?

Long before the formalization of NFL, Rice (1976) presented a framework that deals with the fact that algorithms dominate (in terms of performance) for certain problem subsets and get dominated by other algorithms for other problem subsets. In the presence of a variety of algorithms for a problem, the so-called "algorithm selection problem" (ASP) aims to find a selector that, depending on a given problem instance, chooses an algorithm performing best according to a performance metric. In its basic form, the ASP comprises a problem space P containing problem instances $i \in P$, an algorithm space A (finite or infinite), and a performance measure indicating the quality of an algorithm to solve a problem instance (e.g., objective value or computation time). Given that, the goal of the ASP is to find a selection function $S: P \rightarrow A$ mapping every problem instance to the algorithm that achieves the best value according to the performance measure. Rice proposes using a feature extraction step that converts a problem instance $i \in P$ to a feature representation $f(i) \in F$, in the feature space F, which is often $\mathbf{R}^m$. The framework introduced by Rice (1976) is very versatile and applicable to a broad range of problems and applications. Since then, many approaches have been published in Operations Research and other fields like "meta-learning" (a subfield of Machine Learning) that have successfully applied the idea of ASP (see Soares et al., 2004 or Feurer & Hutter, 2019). Some of them incorporate Machine Learning (ML) models in their selection procedure that predict an algorithm's performance metric (e.g., objective value or computation time) on a specific problem

instance. Others even aim to design more sophisticated methods that create an "algorithm schedule" for a specific problem instance (e.g., Streeter et al., 2007 or Kadioglu et al., 2011). In such approaches, computational resources are not allocated to a single algorithm but are distributed among several algorithms according to their expected performance. Information from preceding algorithm runs, or even a pre-solving phase, can be used to make more intelligent decisions. Overall, the contributions to ASPs have evolved massively, helping to improve solution quality by combining algorithms and adding their strengths. However, as Smith-Miles (2009) argues in her review on ASP and "meta-learning", both research areas could benefit from each other, but those connection have been missed in the past. Algorithm selection could use meta-learning methods and ML models if the following requirements are fulfilled (Smith-Miles, 2009):

i. A large collection of problem instances with various complexity is available.

ii. Many diverse algorithms to solve the problem exist.

iii. Performance metrics to evaluate the algorithm's performance are known.

iv. Suitable features to describe the properties of problem instances can be computed.

With those requirements, the ASP can be stated as a learning problem: given the feature vector representation of the problem instances and objective values computed by the algorithms as training data, learn an ML model that predicts the best-performing algorithm. In this representation, the ML model corresponds to the function *S* in Rice's framework. In his review on algorithm selection applied to combinatorial optimization problems, Kotthoff (2016) emphasized the role of ML for performance prediction models. The ongoing progress suggests that "free lunches" are getting closer. With their growing prediction performance, ML models already play a significant role in the current literature and will gain importance in developing ASP solutions. They are capable of recognizing patterns between algorithms, problem instances, and their performance. Nevertheless, to achieve the most benefit from ML models, the question to be asked (i.e., what should be predicted) and the application of the model must be carefully designed.

In this paper, we propose using an ASP method to select the most appropriate algorithm for a single problem instance offline (i.e., before the solution process starts) and to use this algorithm to compute initial solutions to be improved by two Genetic Algorithms (GA)s.

We make two major contributions to the literature:

- First, we thoroughly discuss and analyze ML models (from different research areas) and present several adaptations (e.g., an adapted loss function) for the ASP task at hand. In this context, we put special emphasis on the "question to ask" the ML model and on finding suitable models to answer it. As we show, "learning-to-rank" models are better suited for our ASP than pure performance predictions.
- Second, as no meta-heuristic has been developed so far for the PSBIJF, we present two GAs as improvement procedures. The two GAs mainly differ in their solution representation: the first one uses a single integer chromosome newly designed for the PSBIJF (but not restricted to it) and the second one, an adaptation from literature, combines an integer chromosome with a random key chromosome and is enhanced by new recombination and mutation operators.

The structure of the paper is as follows. In section 2, the PSBIJF is formally described and existing algorithms for solving are described. In section 3, we discuss literature related to

learning-to-rank models and GAs developed to solve problems similar to the PSBIF. Section 4 presents the methods applied to solve the ASP and section 5 presents the two GAs. The basic experimental setup and the (hyper-)parameter tuning are described in section 6 and 7, respectively. The final experimental results are presented in section 8. In the closing section 9, we summarize the findings and give an outlook on future research topics.

2 Problem description (PSBIJF) and existing algorithms

To provide a common understanding of the investigated scheduling problem, the analyzed serial-batch scheduling problem, called PSBIJF (Parallel Serial-Batch scheduling with Incompatible Job Families), is described in the next section. In the subsequent section, we describe existing algorithms for solving the PSBIJF. These algorithms make up the finite algorithm space *A* of the ASP under consideration.

2.1 The PSBIJF

Basic task of the PSBIJF considered in this paper is the grouping of n jobs ($J = \{J_j \mid j = 1, ..., n \in \mathbb{N}^{>}\}$) into o batches ($B = \{B_b \mid b = 1, ..., o \in \mathbb{N}^{>}\}$) and the scheduling (machine allocation and sequencing) of those batches on a set of *m* identical parallel machines ($M = \{M_l \mid l = 1, ..., m \in \mathbb{N}^{>}\}$). The maximum batch capacity *bc* is identical for all machines and the sum of the individual (arbitrary) batch capacity requirement cr_j of each job assigned to a batch must be lower. Generally, $cr_j \leq bc$ must hold. Furthermore, each job has individual weights w_j, processing times p_j, due dates d_j, and each job belongs to a job family f ($F = \{F_f \mid f = 1, ..., q \in \mathbb{N}^{>}\}$), whereby job families are "incompatible" (that means jobs of different families cannot be processed together in one batch, e.g., due to technical or material restrictions). Because setups between the processing of two batches (jobs) are required, the batching of jobs of the same family is done to reduce setup efforts. Hereby, the setup times are family- and sequence-dependent: $s_{f,g}$ defined the setup time for a setup from a family f batch to a family g batch. Note that $s_{0,f}$ depicts initial setup times for family *f* at the beginning of a schedule. Other assumptions include that each machine can process no more than one batch at a time, that a batch can only be processed by one machine at a time, that all jobs are available for processing at the start (i.e. no release dates), that batch processing cannot be interrupted (i.e. no preemption), that jobs cannot be added or removed once processing of a batch has started (i.e. batch availability), and that the completion time of a job is the completion time of the batch to which a job is assigned.

The primary objective is the minimization of the total weighted tardiness. Additionally, we aim to minimize the total flow time whenever all jobs can be delivered in time (i.e., tardiness is equal to zero). This objective function is writes as follows:

$$ov = \sum_{j \in J} w_j \cdot T_j + {TF}/{\tilde{C}_{\max} \cdot n \cdot 10}. \qquad (1)$$

with tardiness of job T_j and total flow time *TF*. The ordering of the objectives is achieved by defining the (constant) denominator to be greater than the nominator (i.e., the total flow time) in the second part. To assure that the denominator is greater than the TF, the upper bound of the

Journal Pre-proofs

maximum flow time of one job is estimated by the approximated makespan $\tilde{C}_{max}$ and multiplied by the number of jobs. Because the makespan is approximated, we use a "safety" factor of 10.

2.2 Existing algorithms

The first time, the previously defined PSBIJF was considered in Gahm et al. (2022). The authors developed a mixed-integer linear program (MILP), multi-start construction heuristics based on existing, adapted, and new priority rules, and a local search mechanism. Multi-starts are used to perform a grid-search with different heuristic parameter configuration. Their results show that the BATCS-b heuristic (originally abbreviated ATCS-BATCS(β)) with a "controlled batch utilization" outperforms all other approaches.

Based on these results Uzunoglu et al. (2023b) developed a similar multi-start heuristic with "controlled batch urgency" (called BATCS-d) and use the same MILP and local search for improving initial solutions. Furthermore, to improve solution efficiency, the authors proposed learning-augmented heuristics using ML methods (i.e., neural networks, NNs) to minimize the number of starts by predicting most suitable parameters for the heuristics. To that, not only the single predicted parameter configuration is used but a reduced parameter grid is computed, and a parameter is introduced to control the size of the reduced grid (and thus, can be used to balance between solution quality and computation time). Their results show that the best variants with ML-reduced grids (BATCS-b-ML(PC_8, GS3) and BATCS-b-ML(PC_6, GS3)) are very competitive regarding solution quality and clearly outperform BATCS-b and BATCS-d in terms of computation time. The results also show that the "d"-variants generally outperform the "b"-variants but that for some problem instances, only a "b"-variant is capable to compute the best solution.

A similar approach to increase solution efficiency was proposed in Uzunoglu et al. (2023a). The authors also used ML to reduce the parameter grid searched by the multi-start heuristic BATCS-b but do not predict parameters. Instead, they used NNs to predict the performance of a certain parameter configuration and compute a ranking based on the predictions. After computing this ranking, which can be done efficiently due to the very low response time of NNs, several ranking application strategies are available to create a reduced parameter grid. Also, a parameter controls the size of the reduced grid. Their results show that their BATCS-b-MLGS variants are competitive regarding solution quality and clearly outperform BATCS-b in terms of computation time. Similar to the findings in Uzunoglu et al. (2023b), also Uzunoglu et al. (2023a) observe that not one specific BATCS-b-MLGS variant performs best for all problem instances but that different ones achieve best results in relation to some problem instance characteristics. Therefore, we conclude that selecting the most promising algorithm among the best available for the PSBIJF before starting the solution process has the potential to improve the solution quality without increasing the computation time.

Table 1 lists the 13 most efficient algorithms from the literature. Note that we do not consider the algorithms presented in Uzunoglu et al. (2023b), which use improvement procedures, since the ASP task considered here is to select the algorithm that efficiently computes initial solutions.

Table 1: Algorithms from the literature

Algorithm based on parameter predictions (see tables 21 and 22 in Uzunoglu et al., 2023b)	Algorithm based on ranking predictions (see table 6 in Uzunoglu et al., 2023a)

Journal Pre-proofs

BATCS-b-MLPP(PC_4, GS3)	BATCS-b-MLRP ([0,5,5]-AF, Bx)
BATCS-b-MLPP(PC_8, GS3)	BATCS-b-MLRP ([0,5,5]-AF, B(9)-G)
BATCS-b-MLPP(PC_{10}, GS3)	BATCS-b-MLRP ([1,2,7]-AF, Bx)
BATCS-d-MLPP(PC_1, GS3)	BATCS-b-MLRP ([1,2,7]-AF, B(9)-G)
BATCS-d-MLPP(PC_4, GS3)	BATCS-b-MLRP ([1,0,9]-AF, Bx)
BATCS-d-MLPP(PC_6, GS3)	BATCS-b-MLRP ([1,0,9]-AF, B(9)-G)
BATCS-d-MLPP(PC_{10}, GS3)	

Because Uzunoglu et al. (2023a) did not consider "d"-variants in their analysis, we performed a preliminary study to close this gap (see Table 12 in Appendix A-1). Based on the results of this study, we added the four most robust algorithms (BATCS-d-MLRP ([1,2,7]-AF, B(5)-G), BATCS-d-MLRP ([1,2,7]-AF, B(9)-G), BATCS-d-MLRP ([1,4,5]-CF, B(5)-G), BATCS-d-MLRP ([1,4,5]-CF, B(9)-G)) to the algorithm space A (with $|A| = 17 =: n_A$). Note that the MLRP-variants in their original version use larger grids (compared to the MLPP-variants), resulting in higher computation times. To align both variants, we adjust the grid sizes for the MLRP-variants in a way that all 17 considered algorithms perform similar grid searches. The following grid sizes are used (cf. Uzunoglu et al., 2023b): 228 if $n < 100$, 76 if $n \in [100, 1000)$, and 20 if $n \geq$ 1,000. Note that for the "d"-variants, the grid sizes are 1 less because the set of considered "d"-values only contains 10 elements (the set of "b" values contains 11 values).

3 Related literature

Since the literature on the PSBIJF and its solution methods has already been presented in the previous section (more details can be found in the referenced literature), and the basic literature on ASP has been presented in the introduction, we will focus on the learning-to-rank related literature in the next section (along with some notes and references on basic ML methods). The second part of the literature review is dedicated to existing GAs that have been developed to solve problems similar to PSBIJF.

3.1 Learning-to-rank literature

Concerning the problem at hand, we know that the four requirements i. to iv. (see section 1) for using meta-learning methods are met. Therefore, we conclude that an offline trained ML model is suitable to fulfill the ASP task. For using ML models, one should carefully consider what level of information is necessary for algorithm selection, i.e., what the model's response should be. For example, is it necessary to predict the objective value of an algorithm, or is an ordering of algorithms (i.e., ranking) sufficient to select an algorithm? The latter question might be easier to answer when the objective values differ immensely between problem instances (like it is the case for given PSBIJF). Note that a ranking suffices to select an algorithm with a fixed set of algorithms but is not applicable in other cases. Because such a ranking better reflects its application for the algorithm selection, it could also be seen as the "more natural" question to be asked. However, we will analyze both approaches in this paper.

For predicting objective values of algorithms, ML models and methods capable to perform regressions are required. As such ML models are widespread in literature, we are not going to discuss detail here but refer the reader to some standard literature (e.g., Murphy, 2013 or Goodfellow et al., 2016). In contrast, ML techniques that return a ranking are less common and thus will be discussed in detail in the following.

As pointed out, a ranking of algorithms better reflects the application of the ML model output than a mere objective value prediction. ML models returning a ranking of objects are known as "learning-to-rank" models and are most often found in the field of recommender systems (e.g., recommending products to customers based on purchase history). Also, the terminologies "collaborative filtering" and "information retrieval" are used in the literature. For such problems, the ranking problem is stated as follows: given a query q_i (e.g., a user's purchase history) and objects $x_1, \dots, x_n$ (e.g., documents or products), find a machine learning model f that returns a ranking of x_a according to their relevance to the query. This assumes a numerical representation of the query q_i and the objects x_a. In the literature, the mode of operation of learning-to-rank models are defined according to these three options:

- Pointwise ranking: every object gets a relevance score independent of other objects and is sorted according to the relevance score.
- Pairwise ranking: the model predicts which one to prefer for each pair of objects (the final ranking results from multiple preference relations predicted by the model).
- Listwise ranking: given a set of objects, the model outputs a permutation of the objects, considering the dependency in between.

Pointwise ranking computes a relevance score for each object and sorts accordingly. In this setting, standard loss functions, such as mean squared error (MSE) (if different levels of relevance exist) or binary cross entropy (if only levels "relevant" and "not relevant" exist), can be used directly. However, they fail to capture the interdependency between objects related to a query (Freund et al., 2003), which is better incorporated in the last two options – pairwise and listwise approaches. Regarding pairwise ranking approaches, a common drawback is that the learning objective minimizes the error in classifying preferences on pairs of objects rather than the error in ranking the objects (Cao et al., 2007). In consequence, we concluded that listwise ranking approaches are more suitable for solving the ASP as it accounts for the interdependencies between objects and loss function compare complete rankings instead of pairs of objects. Listwise ranking calculates the loss function on the predicted ranked list of objects and compares it to the ranked list given as ground truth. However, a major challenge in this approach is to find an appropriate loss function that can be used in common learning mechanisms (e.g., the loss function must be differentiable for using gradient descent). Note that we refer to the lowest index in the ranking as highest position (indicating the probably best algorithm in the ASP) and the highest index in the ranking as lowest position (indicating the probably worst algorithm in the ASP).

Järvelin & Kekäläinen (2000) proposed the metric normalized discounted cumulative gain (NDCG), taking into account that objects with high relevance in a higher position in the predicted ranking return a higher value (gain) for the user but objects in lower positions in the predicted ranking are less likely to be used. So, for example, misplacing the highest two objects in the ranking results in a higher loss than misplacing the lowest two objects. To define the NDCG metric, we first need a function $g : X \to$ ¡ returning the gain (relevance or value) of objects for a query and the cumulative gain function $cg :$ ¥ $\to$ ¡ $, p$ a $\sum_{k=1}^{p} g(\sigma(k))$ that sums up the gains obtained by the ranking σ up to position p. Then, the cumulative gain is discounted

Journal Pre-proofs

according to the ranking position, for example, by dividing each gain in the sum by $\log_2(\cdot)$ of its rank: *dcg*: $p \mapsto \sum_{k=1}^{p} g(\sigma(k)) / \log_2(\sigma(k))$. Also other monotonically increasing transformations can be used for the discounting. Lastly, the achieved discounted cumulative gain (DCG) is *normalized* by the highest DCG achievable by the optimal ranking (i.e., NDCG) to allow comparing predicted rankings between queries with different resulting rankings (Järvelin & Kekäläinen, 2002). The summation can also be truncated to a specific position *k* such that NDCG only considers the *k* highest ranked positions of the prediction (called NDCG(k)). As NDCG depends on the ranking σ, which returns the position of object x_a and hence is non-differentiable, the NDCG metric itself is non-differentiable and thus cannot be used in machine learning mechanisms relying on the loss function's gradient.

Burges et al. (2006) circumvent the non-differentiability of the NDCG metric in their algorithm "LambdaRank" by defining a smooth approximative loss function. During training, it is known which properties the gradient should have to achieve a better ranking outcome. To derive an appropriate loss function, the goal is to choose the gradient of the desired loss function λ that fulfils this property. In their evaluation, they analyzed different λ functions and compared them against RankNet using the NDCG metric. Their analysis revealed that neural nets trained with LambdaRank clearly outperform a competitive pairwise approach (RankNet; see Burges et al., 2005) regarding accuracy (NDCG) and training time. Its boosted decision tree equivalent was named "LambdaMART" and published in Burges (2010).

Also, with a focus on treating the ranking problem in a listwise manner, Cao et al. (2007) present "ListNet". ListNet is a neural network operating on probability distributions over permutations or the probability of objects being ranked on the top *k* positions (called top *k* probability). The computational experiments show the superiority of ListNet over other pairwise competitive algorithms on almost every test data set, concluding the advantage of listwise approaches over pairwise approaches.

Another line of research takes a more direct approach to find functions that are easier to optimize and behave like the previous metrics (e.g., NDCG). Some of the works try to optimize functions that are (upper) bounds to a ranking metric (see Chapelle et al., 2007 for a 'structured output' based approach and Xu et al., 2008 for a framework on upper bounds and applications). Another approach by Taylor et al. (2008) (called "SoftRank") tries to smooth the metric by assuming randomness and using a Gaussian distribution as ranking score. Qin et al. (2010) proposed an approximation framework that reformulates the metrics needing a sorting, and therefore, positions as indices (i.e., indexed by positions), to metrics that derive indices from their objects. In addition, they approximate the position function with a logistic function based on the object's ranking score and apply their approximation in an optimization technique called "ApproxNDCG" (and "ApproxAP" for average precision), which also needs a hyperparameter to be determined. They compared the accuracy of ApproxNDCG to SoftRank, ListNet, and others (but without including LambdaRank and LambdaMART in their analysis). They showed that ApproxNDCG outperforms all other methods on a variety of data sets.

The literature on learning-to-rank presents a rich body that differs in how they treat the dependency between objects: pointwise approaches neglect dependencies between objects, pairwise approaches consider preferences between pairs of objects, and listwise approaches work on the complete list of objects. Listwise approaches best reflect how the prediction is applied for solving the ASP, but loss functions of interest are not differentiable and prevents their direct usage in gradient-based methods. Plenty of works attempt to overcome this issue and the more

recent approaches, such as LambdaRank/LambdaMART or ApproxNDCG, have shown to be superior in computational studies.

In consequence, these approaches are of special interest regarding the ASP task at hand and their application, adaption (if necessary), and enhancements is elaborated in detail in section 4.

3.2 Genetic algorithm related literature

The following part of the literature review analyses publications using GAs for solving batch scheduling problems that are closely related to the problem at hand, i.e., batch scheduling problems with bounded batch capacities, batch availability, and incompatible job families. Using the knowledge base provided by Wahl et al. (2023), we identified 13 relevant articles using GAs (note that we also use their notation scheme to specify batch scheduling problems in the following). Since our goal is to design a GA for the PSBIJF, we put special emphasis in the analysis on the used evolutionary mechanisms (e.g., mutation or recombination). Another important aspect for each GA is whether the chromosome decoding procedure (or any other batching procedure) follows a ”full-batch-policy” (i.e., batch are always filled with jobs until batch capacity does not exceed) or not. As discussed in Gahm et al. (2022) and Uzunoglu et al. (2023b), forced full batches may lead to suboptimal decisions for the PSBIJF (but, of course, might have been appropriate for the originally addressed problem). Furthermore, our GAs are designed to improve the quality of pre-computed solutions. To represent these pre-computed solutions and to not miss regions that could further improve them, we require our GA to operate on the entire solution space. Two aspects that are contrary to this goal (even if they have other benefits) are: decomposition approaches (instead of solving all subproblems jointly) and exclusion of certain parts of the solution space. Therefore, we report if the GA solves the complete problem or a decomposition approach is used, and if the solution methods cover the entire solution space of the problem. In Table 2, the developed GAs of the relevant literature are summarized according to the discussed aspects, if a full-batch-policy was used (F) or not (L), if the entire problem was solved by the GA (C) or a decomposition approach was used (D), the type of representation/chromosomes (bin := binary, int := integer; rk := random key, and obj := objects), and the information represented by the chromosomes.

Since GAs using a full-batch-policy are not suitable for the PSBIJF, we concentrate on the other approaches in the following detailed analysis.

Table 2: Overview of related GA developments

Reference	Problem specification	(F) or (L)	(C) or(D)	Chromosome type(s)	Chromosome information (chromosome size)
Balasubramanian et al. (2004)	P | pF, if, cr1, pb | wT’	F F	D D	int int	Job to batch assignment (*n*) Batch to machine assignment (*n*)

Koh et al. (2004)	P | pF, if, crJ, pb | C, wC, Cmax’	F F	C D	rk + rk rk	Job to batch assignment + batch to machine assignment with batch sequence (2*n*) Job to batch assignment (*n*)
Koh et al. (2005)	S | pF, if, crJ, pb | C, wC, Cmax’	F	D	rk	Job to batch assignment (*n*)
Mönch et al. (2005)	P | pF, rJ, if, cr1, pb | wT’	L L	D D	int int	Job to batch assignment (*n*) Batch to machine assignment (*n*)
Malve & Uzsoy (2007)	P | rJ, if, cr1, pb | Lmax	F	C	rk	Job sequence (*n*)
Mönch et al. (2007)	P | rJ, net, if, cr1, sFS, elig, pb | wT	L	D	int	Batch to machine assignment (*n*)
Dauzère-Pérès & Mönch (2013)	S | pF, dJ, if, cr1, pb | wU	F	C	rk	Job sequence (*n*)
Jia et al. (2013)	P | pF, rJ, re, if, cr1, maxL, pb, bLb, on | wT, TP, cIh	F	D	int	Batch to machine assignment (*n*)
Castillo & Gazmuri (2015)	HJ | if, crJ, sFMS, cb, sb, bF, bLb | Cmax	L	C	objects	Batch sequence, size, machine assignment, family (*n*)
Huynh & Chien (2018)	P | pF, dlJ, if, crJ, sFS, pb | Cmax	L	C	int + rk	Job to batch assignment + batch to machine assignment with batch sequence (2*n*)
Huang et al. (2020)	S, aFlex | rJ, if, cr1, pb | Cmax’	F	D	rk + binary	Job sequence + preventive maintenance (2*n*)
Kim et al. (2021)	S, aFlex | pF, pDet, if, crJ, pb | Cmax	F F F	D D D	rk rk rk + rk	Job to batch assignment (*n*) Batch sequence (up to *n*) Job to batch assignment + Batch sequence (up to 2*n*)
Wu et al. (2022)	HJ, aFlex | pF, rO, dlO, re, if, cr1, elig, pb, bFM | TP’	L	C	objects	Job to batch assignment, batch to machine assignment, and batch sequence (up to *n*)

Mönch et al. (2005) proposed using GAs to solve the “P | pF, rJ, if, cr1, pb | wT”-problem. They used the same two decomposition approaches first presented in Balasubramanian et al. (2004) but combined the GAs with other heuristics for batching and sequencing. Their modified ATC dispatching rule considered multiple batch combinations, and thus, also batches with “free” capacity are considered (no full-batch-policy). In their GAs’ implementations, the roulette wheel selection mechanisms and operators (one-point crossover and a flip mutation) were the same as in Balasubramanian et al. (2004). However, due to their decomposition approach and the applied

Journal Pre-proofs

dispatching rules for batching and sequencing, not the entire solution space is covered. We therefore do not consider their approach to be an appropriate solution method to the PSBIJF.

To solve a complex job shop scheduling problem, Mönch et al. (2007) proposed a decomposition approach, which needs to solve the batch-scheduling sub-problem "P | rJ, net, if, cr1, sFS, elig, pb | wT". For solving this sub-problem, a GA allocated and sequenced already formed batches to and on machines. To that, the integer chromosome represents a batch's machine allocation, and the sequence of genes (representing batches) in the chromosome is equal to the sequence of batches on the allocated machine. The GA used a one-point crossover, flip mutation, and overlapping populations as evolutionary mechanisms. Computations terminated after reaching a given number of generations. The batching procedure considered multiple batch combinations and thus did not aim at full batches, but due to the decomposition approach, not the entire solution space is covered. As this GA has the same characteristics as that of Mönch et al. (2005), we do not consider it appropriate either.

Castillo & Gazmuri (2015) proposed three GAs with different types of crossovers for solving the "HJ | if, crJ, sFMS, cb, sb, bF, bLb | Cmax"-problem. For solution representation, the authors used an ordered set of batches with additional information such as batch size and assigned machine. Three crossover mechanisms were developed: "edge recombination-based crossover", "batch position-based crossover" (similar to a one-point crossover), and "guided mutation crossover" (a local search around the clone parent, towards the guide parent). For mutation, the authors proposed four mechanisms that partially use some kind of local search to keep all chromosomes feasible: "mutate amount of batches", "mutate batch sizes", "mutate machine assignments", and "mutate batch sequence" (i.e., batch swapping). The experiments showed that the guided mutation crossover (with the integrated local search) had the fastest convergence of all three crossovers. The applied chromosome representation allows any batch size, the complete problem is solved, and the entire solution space is covered. However, solution representation, recombination and mutation operators are highly problem specific and thus, we do not directly follow their approach. Nevertheless, their developed mutation mechanisms influenced our mutation operator development.

For the "P | pF, dlJ, if, crJ, sFS, pb | Cmax"-problem, Huynh & Chien (2018) proposed a multi-subpopulation GA combined with heuristics. Because our second GA uses the same solution representation and similar operators, we are not going into detail here but in section 5.2. The multi-subpopulation GA used three parallel subpopulations that evolved independently and were coordinated at certain points to prevent any single subpopulation from converging too quickly or slowly. To coordinate the three subpopulations, new subpopulations containing new chromosomes were created after a certain number of generations by interchanging a given number of best solutions. The results showed that the multi-subpopulation GA outperformed the conventional GAs by about 5% in terms of solution quality, while the CPU time was about the same. Unfortunately, the authors did not report results without the integration of the two local search heuristics. Therefore, it is not possible to isolate their multi-subpopulation-strategy's contribution to the result. However, since the representation by two chromosomes allows arbitrary batch sizes and the complete problem is solved, the entire solution space for the problem at hand is covered and makes their approach interesting for solving the PSBIJF. As already mentioned, this GA directly inspired the development of our second GA, but we did not integrate their local search heuristics as those are dedicated to problem specific characteristics (e.g., the makespan objective) and therefore not appropriate for the PSBIJF. Furthermore, we made several adaptations regarding recombination and mutation operators.

To solve the "HJ, aFlex | pF, rO, dlO, re, if, cr1, elig, pb, bFM | TP"-problem, Wu et al. (2022) developed a GA where each chromosome represented a sequence of "batch objects"

defining the machine (assigned tool), the job family (recipe), the set of assigned operations, and the start time of the batch. The authors developed several problem-specific properties to reduce the solution space for the GA. These properties were also used by the procedure to randomly generate the initial population. Based on a fitness-proportional parent selection, the offspring were generated through representation-specific crossover and mutation operations. The authors explicitly stated that they did not follow a full-batch-policy because smaller batches may lead to better schedules. The proposed GA covers the entire solution space of the problem considered here. However, due to the large difference regarding the basic problem settings compared to the PSBIJF and the GA's high problem specificity, particularly because of the used properties, we do not consider its application to be expedient.

Summarizing this analysis of GAs, we come to the conclusion that the GA developed by Huynh & Chien (2018) provides the most suitable aspects for developing GAs for the PSBIJF which we integrated in our second GA. Both GAs are elaborated in detail in section 5.

4 Algorithm selection by learning-to-rank

As the more recent literature in algorithm selection suggests, ML models can be powerful tools to select good algorithms for a problem instance to solve. Empirical hardness models or models to predict the performance of an algorithm solving a problem instance are popular choices. However, predicting the actual performance value is unnecessary but a mere ordering of algorithms is sufficient for algorithm selection. Furthermore, evaluating the performance of an ML model for algorithm selection based on the error (e.g., the MSE) between the actual and predicted algorithm performance does not coincide with its later use. Loosely speaking, predicting an algorithm's (continuous) objective values is more challenging than predicting the algorithm performance on an ordinal scale (e.g., ”good” or ”mediocre” performance) or a listing according to the performance (which implies the latter two cases). This becomes particularly important when objective values can vary by several orders of magnitude from problem instance to problem instance, as is the case for the PSBIJF with the total weighted tardiness objective. In consequence, learning-to-rank methods seem the better choice. Nevertheless, one should be aware that many learning-to-rank algorithms were developed with use cases very different from algorithm selection. They stem from applications like recommender systems or search engines with vast sets of objects explained by numerical feature vectors and deal with problems like bias in human-generated data. Therefore, thoroughly analyzing the applicability of different learning-to-rank methods is essential to find the most suitable technique for our ASP.

To perform the ASP task by learning-to-rank ML models, three components must be determined, considering their interdependencies: the basic ML model type, the labeling strategy, and the loss function used during training, validation, and testing. These components and combinations are discussed in the following paragraphs,

Because NNs are commonly used by recent ranking methods (and regression tasks in general) and gradient-boosting decision trees (GBDT) have shown their superior performance as learning-to-rank methods, we use them as basic ML model types in our ASP methods.

In the context of ranking ML tasks like the ASP, the way of labeling the training data plays an important role. Therefore, we investigate five new or adapted “labeling strategies” in our analysis: In the first strategy called “BIN”, all algorithms from A that have computed the best solution for a problem instance are indicated by 1 and otherwise by 0. That means, each sample has n_A binary labels indicating the best algorithm. In strategy “OV”, n_A real-value labels depict the objective values computed by each algorithm (ov_a) and in “OV-N”, the objective values

have been normalized on the interval [0,1] with $ov_a = \frac{ov_{\max} - ov_a}{ov_{\max} - ov_{\min}}$ (regarding a single problem instance). Furthermore, we use the labeling strategy "RANK", which has n_A labels with integer from the interval $\{1, \ldots, n_A\}$. Here n_A marks the best algorithm and 1 the worst. If two algorithms achieved the best objective values, both are labeled with 17 and the next is labeled with 15. We also use a version of RANK with scaled labels from $\{1/n_A, 2/n_A, \ldots, 1\}$ called RANK-SC, to analyze the impact of scaling on the training strategies. Here, 1 marks the best algorithm and $1/n_A$ the worst.

In addition to the labelling strategy, the loss function used during training, validation, and testing is essential. Therefore, we investigate different loss functions that are coupled to the labeling strategy and model type (i.e., not all loss functions are applicable to all labeling strategies or model types). For the labeling strategy BIN, the loss function binary cross entropy (BCE) and for the labeling strategies OV, OV-N, and RANK-SC the MSE is used as recommended in the literature. A newly developed loss function first proposed in this paper is the "importance weighted MSE" (iwMSE(α)). The idea of this loss function is to emphasize better (higher) ranked algorithms, so in a sense, the ML model also must understand which algorithms should be ranked higher. It is basically the MSE with a weight depending on the "true" ranking of algorithm a: $iwMSE(y, \hat{y}) = 1/n_A \sum_{i=1}^{n_A} \left(y_a - \hat{y}_a\right)^2 \left(\alpha y_a + 1\right)$ (with y_a representing the label and $\hat{y}_a$ the predicted ranking score). We use the parameter $\alpha \in \mathbb{R}^{>0}$ to control the effect of the weighting. Of course, this biases a model to predict higher ranking scores $\hat{y}_a$ since missing higher-ranked algorithms is punished stronger than missing lower-ranked algorithms. Therefore, the control parameter should not be exaggerated and be tuned with caution. Theoretically, an "optimal" model would not mind this effect, but approximation and estimation errors (and the fact that gradient descent could converge in a local optimum) hinder us from finding such models in practice. In our experiments, we analyze the impacts for $\alpha \in \{1, 2, 3\}$. The following "loss functions" also affect the learning process (algorithm) and are therefore not applicable to both ML model types (see section 3.1 for further details): ApproxNDCG is a loss function particularly developed for NNs (Qin et al., 2010), LambdaRankNDCG is the learning method published in Burges et al. (2006) for the NDCG loss, and LambdaRankNDCG(1) is its truncated version to rank 1 only. For GBDTs and the labeling strategy BIN, the LambdaBinary trains according to the mean average precision (Donmez et al., 2009). RankPairwise uses an GBDT-adaptation of the pairwise loss function RankNet developed by Burges et al. (2005) and LambdaMART is the GBDT-equivalent to LambdaRankNDCG presented in Burges (2010).

All appropriate ML model types, labeling strategy, and loss function combinations forming "ASP-models" are summarized in Table 3 in section 7.1. Note that all the presented ML model types, labeling strategy, and loss function combinations can be considered as listwise ranking approaches as we have a finite and fixed object space (algorithm space).

For the training, validation, and testing of the ASP-models, we use a large instance set from the literature (see section 6.1) and solve each of the 71,040 instances with the 17 available algorithms. This leads to a set of 1,207,680 data points when each problem instance and algorithm combination is processed individually.

To convert a problem instance $i \in P$ into a feature representation, we use the AF-vector (aggregated feature-vector) proposed in Uzunoglu et al. (2023b), as it has shown to be suitable for adequately representing the properties of PSBIJF problem instances.

Journal Pre-proofs

Since with the objective function in eq. (1) also a metric to evaluate an algorithm's performance exists, all four requirements to use ML models (see i. to iv. in section 1) for solving the ASP Algorithm are fulfilled.

5 The Genetic Algorithms

To solve the present serial-batch scheduling problem, we propose to improve the solutions computed by the LACH with two GAs that mainly differ in their problem representation. The first one uses an integer representation where each job is assigned a batch-position (representing a specific position on one of the machines). We abbreviate this approach as GA-J2P (job-to-position assignment). In the language of genetics, this means that a phenotype (schedule) is encoded by a genotype (individual) consisting of a single chromosome of n genes (one for each job), and the allele is an integer value representing one of the given batch-positions (for definitions and details of GA-related terms used in this paper see Eiben & Smith, 2015). The second GA uses a genotype consisting of two chromosomes: The first chromosome consists of n integer genes representing the job-to-batch assignment, and the second chromosome consists of n real-valued chromosomes representing the batch-to-machine assignment and the sequence of batches on a machine. This one is called GA-J2BRK (job-to-batch assignment with random keys). Note that both GAs are capable to represent all feasible solutions of interest because we do not integrate any heuristics for batching or scheduling and both representations and their decoding (genotype-phenotype mapping) produce non-delay schedules (no idle time is inserted) and the objective function (1) is regular (cf., Pinedo, 2016). This means that both mappings are complete (cf., Eiben & Smith, 2015).

For both GAs (individual aspects are described in sections 5.1 and 5.2, respectively), we implemented the following basic GA scheme (cf. Eiben & Smith, 2015) with the parameters "population size" (μ), "initial population composition" (*ipc*), "elitism rate" (*esr*; fraction of best individuals that are certain to be transferred to the next generation), "survival rate" (*svr*; fraction of individuals that survive, i.e., that are transferred unaltered to the next generation), "survivor selection mechanism" (*svs*; how individuals are selected for survival), "parent selection mechanism" (*pas*; how individuals are selected to compute offspring individuals), "recombination probability" (P_{rec}; probability that two parents are recombined into two offspring individuals), "mutation probability" (P_{mut}; probability that a gene of an offspring individual is mutated), and the "termination condition setting" (*tcs*).

General GA scheme

---

pop(0){} **:= getInitialPopulation(** μ , *ipc***)**
evaluate(*pop(0){}***)** // calculate fitness of all initial individuals
Do
  g := *g*+1 // increment generation
  s{} **:= selectSurvivor(***pop(g-1){}*, *esr*, *svr*, *svs***)**
   // survivors are directly transferred to the next generation
  p{} **:= selectParents(** *pop(g-1){}*, 1 - *svr, pas***)**
   // for recombination
  o{} **:= recombine(***p{}*, P_r**)** // recombination of two parents
  m{} **:= mutate(***o{}*, P_m**)** // mutation of the offspring
  evaluate(*m{}***)** // calculate fitness of all offspring individuals
  pop(g) := *s{}* + *m{}*
Until (*tcs* is satisfied**)**

The corresponding GA-specific components are described in the following sections. However, we use the same termination criterion "maximum execution times in seconds" (*maxS*) for both GAs. This value is defined in relation to the number of jobs of an instance, as this is the major characteristic influencing the computation time. During different stages of the development and final testing of our GAs, we would like to use different settings to keep computation times manageable. Therefore, two settings are defined:

- TS1: *maxS*=30 if $n \leq 600$, *maxS*=60 if $n \in (600, 1{,}200]$, *maxS*=90 if $n \in (1{,}200, 2{,}400]$, and *maxS*=120 if $n > 2{,}400$.
- TS2: *maxS*=45 if $n \leq 100$, *maxS*=90 if $n \in (100, 200]$, *maxS*=180 if $n \in (200, 400]$, *maxS*=360 if $n \in (400, 800]$, *maxS*=720 if $n \in (800, 1{,}600]$, and *maxS*=1,440 if $n > 1{,}600$.

5.1 GA-J2P

The basic idea of GA-J2P is the new genotype representation of a schedule by a single chromosome containing the complete information. The proposed representation can model the complete solution space, which is particularly important with respect to the problem characteristics combination of serial batching, arbitrary batch capacity requirements, and weighted tardiness. At the same time, we wanted to avoid an overly specific representation by complex genes (see. e.g., Castillo & Gazmuri, 2015 and Wu et al., 2022) to make the proposed GA applicable to a wider range of batch scheduling problems. Another benefit of this genotype representation lies in its close relation to the phenotype schedule as such "natural" representations combined with applicable genetic operators are "quite useful in the approximation of solutions of many problems" (Michalewicz, 1996, p. 4).

5.1.1 Genotype representation and decoding

The genotype consists of a single integer chromosome with one gene for each of the *n* jobs (gene indices are therefore simultaneously job indices *j*), and the allele values represent batch-positions. Hereby, batch-positions simultaneously encode the machine and the sequence of batches on a machine, and therefore, each batch-position simultaneously represents one batch.

This representation idea is very similar to the main decision variable (*X*) used in the mixed-integer linear program presented in Gahm et al. (2022). In general, the number of batch-positions on each machine must not be less than n. However, since we observed that the number of batches created in the initial solutions is much smaller than this value and Gahm et al., 2022 also report a similar observation (see their table 14), we introduce the parameter "batch-positions per machine" (*bpm*) to control the total number of batch-positions. Here, we assume that smaller *bpm* values are beneficial for the solution process, but it is important to use values that do not restrict the solution space in a way that prevents good or optimum solutions. The effectiveness and increase in efficiency of different *bpm* values will be analyzed in detail in the experimental section. Batch-positions are numbered as follows: bp=1 for the first batch on machine i=1, bp=2 for the second batch on machine i=1, bp =bpm for the last batch on machine i=1, $bp=(i-1)bpm+1$ for the first batch on machine i=2, $bp=(i-1)bpm+2$ for the second batch on machine i=2, $bp=(i-1)bpm+bpm$ for the last batch on machine i=2, and so on. Fig. 1 illustrates in part a) the genotype chromosome assigning a batch-position to each job and in part b) the resulting phenotype

schedule after decoding, i.e., after assigning jobs (italic) to the batch-positions (in the squares) on the machines. Note that we do not have any ordering or sorting (e.g., by job families) in the integer chromosome to avoid any positional bias (see the associated families *f* per job in Fig. 1 a)). This is in contrast to the family-wise ordering of integer chromosomes proposed by Huynh & Chien (2018).

a)

f =	*X*	*Y*	*X*	*Z*	*Y*	*Z*	*Z*	*X*	*X*	*Y*	*X*
j =	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
Chromosome A	2	5	2	3	8	1	3	6	6	7	9

b)

Batch per machine

	1	2	3	4	*bpm*=5
i=1	[1] *6*	[2] *1, 3*	[3] *4, 7*	[4]	[5] *2*
i=2	[6] *8, 9*	[7] *10*	[8] *5*	[9] *11*	[10]

Fig. 1: Integer chromosome with job to batch-position assignment and decoding

The decoding follows the gene sequence of the integer chromosome and adds one job after the other to the corresponding batches: e.g., job 1 to batch-position 2 → (then) job 2 to batch-position 5 → job 3 to batch-position 2 and so on (cf. Fig. 1 b)). Note that empty batches (e.g., batch-position 4) are just ignored during the evaluation of the schedule.

5.1.2 Close and least impact insertion procedure (CLIP)

Several operations in GA-J2P (e.g., random solution generation or recombination) require the insertion of a job into "new" batch-positions if the desired batch-position is already occupied and an insertion is not feasible (due to job family or batch capacity requirements). In this case, our proposed insertion procedure aims to insert the jobs into batch-positions closest to the initially desired batch-position (e.g., near on the same machine or at a similar position on a different machine) and that have the least impact on the overall schedule. The latter aspect is important to allow the transfer of "good" partial schedules from one genotype to the other. To achieve this insertion in a most efficient manner, we define 20 insertion sections, as exemplarily depicted in Fig. 2.

For the definition of these sections, we use several bounds in relation to *bpm*: sections 1 to 8 are defined by lb_1=1 and ub_1=0.1·*bpm*, sections 9 to 14 are defined by lb_2=ub_1+1 and

Batch per machine (with *bpm* = 20)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…	20
i=1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		20
i=2	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	…	40
i=3	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	…	60
i=4	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	…	80
i=5	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	…	100
i=6	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	…	120

Fig. 2: Illustration of CLIP with a job insertion at the already occupied batch-position 48 (numbers in circles refer to insertion sections)

Journal Pre-proofs

ub_2=3·ub_1, and sections 15 to 20 are defined by lb_3=ub_2+1 and ub_3=bpm. Given these definitions, Fig. 2 illustrates the procedure to find a new batch-position for all the jobs with the desired batch-position 48: we first try to insert the job in a batch-position in section 1 (49→50), then in a batch-position in section 2 (47→46), then in a batch-position in section 3 (28→8), then in a batch-position in section 4 (68→88→108), then in a batch-position in section 5 (29→9→3→10), then in a batch-position in section 6 (69→89→109→70→90→ 110), then in a batch-position in section 7 (27→7→26→6), then in a batch-position in section 8 (67→87→107→66→86→106), and so on.

Note that the sections 3, 5, 7, 11, 13, 17, and 19 contain the corresponding batch-positions from the current machine (here 3) to machine one (i=1) and that the sections 4, 6, 8, 12, 14, 18, and 20 contain the corresponding batch-positions from the current machine (here 3) to machine i=m.

5.1.3 Initial population composition

To generate the initial population for GA-J2P, we use three approaches controlled by the parameter "initial population composition" (*ipc*): the first approach uses only randomly generated solutions (*ipc*=*ran*), the second one uses 30% initial solutions computed by the LACH and 70% randomly generated solutions (*ipc*=*icr*), and the third one uses 30% initial solutions computed by the LACH and 70% randomly modified initial solutions (*ipc*=*iri*; by the recombination and mutation variation operators described below). The latter two approaches significantly differ in the diversity of the initial population, and we will investigate the effect on the solution process in our experimental study. Depending on the population size and since the learning-augmented heuristics provide only a limited number of solutions related to the number of jobs (i.e., 227 or 228 if n < 100, 75 or 76 if $n \in [100, 1,000)$, and 19 or 20 if n > 1,000; cf. section 2.2), the number of available initial solutions may not be sufficient with respect to the given proportion. In this case, additional random solutions are added to the initial population according to the population size μ. Because of the limited number of available initial solutions, we are not going to tune the fraction of initial solutions during parameter tuning.

For the fully randomly generated solutions, we draw batch-positions from $\{1, \ldots, m \cdot bpm\}$ for each job (note that if not stated otherwise, random numbers are always drawn from restricted uniform distributions). To ensure feasible solutions in the initial population, we decode all generated genotypes after generation. If we identify an infeasible batch in terms of incompatible job family or batch capacity during decoding, we store the job in an additional list instead of adding it to the batch. After adding all jobs to the schedule or the list, we sort the list by non-decreasing batch positions and add the jobs to the schedule in that order using CLIP.

For generating randomly modified initial solutions, we randomly pick two of the initial solutions, apply all five mutation operators to them, use all two recombination operators to generate four offsprings, and then apply all five mutation operators to the offspring. This leads to 10 + 4 + (4·10) = 54 randomized initial solutions. These actions are repeated with a gradually increasing chance of mutation until the necessary quantity of initial solutions is reached. The probability of mutation begins at 0.02 and rises by 0.005 per cycle. As all operators compute feasible solutions, an additional feasibility checking is not required here.

5.1.4 Elitism and survivor selection

Since GA-J2P is basically designed to improve initial solutions, we use an elitism mechanism that directly transfers a certain number ε of best genotypes to the next population. This number is controlled by the parameter "elitism rate" *esr* and defines the number of transferred best

genotypes as $\varepsilon = \lfloor esr \cdot \mu \rfloor$. For the same reason, we follow a steady-state population management model, i.e., we do not replace the entire population in each generation, but only a part of it (cf., Eiben & Smith, 2015). In literature, the number of individuals replaced by its offspring is named λ and thus, the number of surviving individuals is $\mu - \lambda$. As the number of surviving genotypes is controlled by the "survival rate" parameter *svr*, $\lambda = \mu - \lfloor svr \cdot \mu \rfloor$. To preserve the diversity of populations, not the best $\mu - \lambda$ genotypes are selected but more advanced selection mechanisms, controlled by the "survivor selection mechanism" parameter *svs* are applied. Here, we will study the fitness-based selection mechanisms "Tournament Selection" (TOS) and "Stochastic-universal Sampling" (SUS). Note that we do not study "Roulette wheel selection", as it has been recognized that it does not actually provide a good sample of the distribution when more than one sample is drawn (Eiben & Smith, 2015). In TOS, a genotype wins a "tournament" if its fitness is greater than the fitness of the other $s - 1$ competing genotypes, whereat the competing genotypes are drawn randomly. To vary selection pressure, one can adjust the tournament size s. When s is larger, genotypes with lower fitness have a reduced chance of surviving. Note that the worst genotype never survives, and the fittest genotype is the winner of every tournament in which it competes. The tournament selector is commonly used in practice due to its lack of stochastic noise in comparison to fitness proportional selectors. In contrast, SUS chooses individuals based on a given probability (related to the fitness) to minimize the chance of fluctuations. It can be seen as a form of a roulette wheel game with evenly spaced points that we spin. SUS uses a single random value to select individuals at equally spaced intervals. This fitness-based selection method grants a better chance of the selection of weaker individuals, thus reducing the "unfairness" associated with other fitness-based selection methods. Because both survivor selection mechanism are intended to preserve diversity, we use a tournament size of two (s=2) in all experiments. The parameters *esr*, *svr*, and *svs* will be tuned in section 7.2.

5.1.5 Parent selection and recombination

To randomly select the parents that are used for offspring generation, we also study the two selection mechanisms TOS (with s=2) and SUS from section 5.1.4. The parameter "parent selection mechanism" *pas* is used to differentiate between them.

For the recombination of two genotypes, we adapted two standard operators from literature: one-point crossover and two-point crossover. The first operator, "J2P adapted one-point crossover" (J2PaOPX), begins with randomly selecting a crossover point from {2, … $m \cdot bpm$-1} (e.g., 6 in Fig. 3 a)). This point partitions both the chromosomes A and B into head and tail (see Fig. 3 a)). The J2P specific adaptation takes place with the exchanging of the tails. Here, the batch-positions are transferred by CLIP in non-decreasing order of batch-positions and job indices (see the small numbers in Fig. 3 a)) to the two offspring A* and B*, respectively. Note that all following examples assume m=1, bc=2 (jobs), and bpm=11. Furthermore, actualized batch-positions are marked bold and that gray shaded fields signal batch-position updates by CLIP.

Journal Pre-proofs

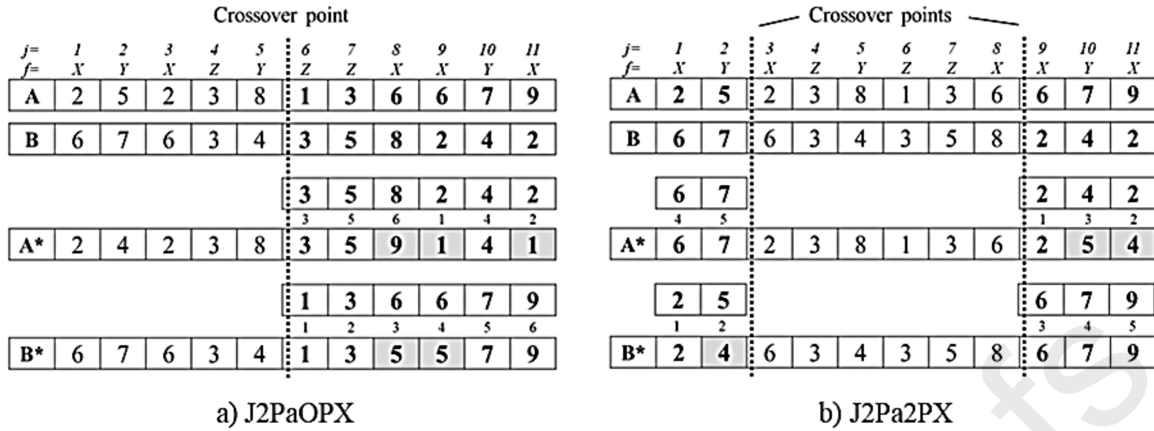


Fig. 3: **Recombination operators for GA-J2P**

The second recombination operator "J2P adapted two-point crossover" (J2Pa2PX) follows the same procedure but uses two randomly selected crossover points from $\{2,\ldots,m \cdot bpm\}$ (e.g., 3 and 9 in Fig. 3 b)) and first transfers the middle parts and then CLIP inserts head and tail in non-decreasing order of batch-positions and job indices (see Fig. 3 b)).

Both recombination operators have individual recombination probabilities (P_{aOPX}^{J2P} and P_{a2PX}^{J2P}) that must be tuned, whereby a value of 0 indicates that the operator is not in use (as also for the following mutation operators).

5.1.6 Mutation

Mutation operators have two distinct (conflicting) roles in the evolution process of a GA: Exploitation, i.e., intensification of the search in promising regions of the solution space by making small changes, and exploration, i.e., maintaining population diversity to prevent a premature convergence to a local optimum. In this context we developed five operators for mutating the offspring resulting from recombination: "J2P random resetting" (J2Prr), "J2P batch swap" (J2Pbsw), "J2P job swap" (J2Pjsw), "J2P batch insert" (J2Pbin), and "J2P job insert" (J2Pjin). The latter four mutation operators are design to achieve similar mutations as the mutation operators presented in Castillo & Gazmuri (2015).

Random resetting is a standard mutation operator for integer chromosomes that mutates each gene independently (with a given probability P_{rr}^{J2P}) and a new allele value (batch-position) is drawn from $\{1,\ldots,m \cdot bpm\}$. The new batch positions are added by CLIP in non-decreasing order of batch-positions and job indices.

Fig. 4 shows an example where the jobs 2, 5, 6, 10, and 11 have been selected, new randomly chosen batch-positions have been drawn, and added to the new chromosome A* using

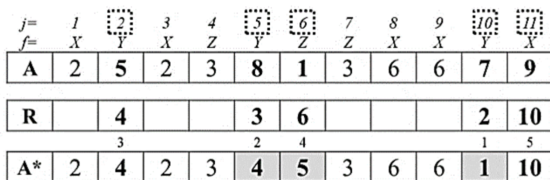


Fig. 4: **J2P random resetting with CLIP**

Journal Pre-proofs

CLIP. The number of mutated genes in a population can be approximated by $\mu \cdot n \cdot P_{rr}^{J2P}$ (where n represents the number of genes in the chromosome). Therefore, J2Prr combined with a high mutation probability P_{rr}^{J2P} can be used to maintain population diversity as many jobs to batch-positions assignments may be affected and batches may be "opened" or "closed".

The mutation operator J2Pbsw swaps two complete batches with arbitrary families. This is done by randomly selecting two genes with probability P_{bsw}^{J2P} and swapping batch-positions for all jobs assigned to one of the batch-positions (see Fig. 5 a)). Since both batches (batch-positions 3 and 6, respectively) were feasible in terms of job family and batch capacity, the swapping must be feasible.

j=	1	2	3	4	5	6	7	8	9	10	11
f=	X	Y	X	Z	Y	Z	Z	X	X	Y	X
A	2	5	2	3	8	1	3	6	6	7	9
A*	2	5	2	6	8	1	6	3	3	7	9

a) J2Pbsw

j=	1	2	3	4	5	6	7	8	9	10	11
f=	X	Y	X	Z	Y	Z	Z	X	X	Y	X
A	2	5	2	3	8	1	3	6	6	7	9
A*	2	5	6	3	8	1	3	6	2	7	9

b) J2Pjsw

Fig. 5: J2P batch swap and J2P job swap

In contrast, infeasibilities may occur when J2Pjsw tries to swap two jobs between batches (by interchanging the batch-positions). Therefore, we first randomly select a single gene with probability P_{jsw}^{J2P} (indicated by 1 in Fig. 5 b)), then determine all genes with jobs of the same family (e.g., 1, 3, and 11 in Fig. 5 b)) and that are not in the selected batch-position (since otherwise the probability of swapping jobs from batches with many jobs is lower compared to other batches). Then we randomly select one of these genes (indicated by 2 in Fig. 5 b)). If swapping is feasible in terms of batch capacity, the batch-positions are interchanged, otherwise, the chromosome remains unchanged. In general, J2Pbsw has a greater impact on a chromosome (because multiple jobs change their batch-position) than J2Pjsw, which only changes the batch-position of two jobs. In both cases, the number of batches remains the same.

The mutation operator J2Pbin inserts a complete batch at a different batch-position. First, a single gene j is randomly selected with probability P_{bin}^{J2P} and the assigned batch-position bpA is determined (e.g., j=9 with pbA=6; cf., Fig. 6 a)). Then, the new batch-position bpB is drawn from $\{1,\ldots,m \cdot bpm\}$. If bpB is empty (i.e., no job is assigned to it), all jobs at bpA are inserted into bpB by updating the corresponding genes (see for example chromosome A^{*1} with bpB=4). If bpB is not empty (e.g., bpB=2), all jobs at bpA are also inserted into bpB, and the batch-positions of jobs formerly assigned to bpB are inserted into bpB+1. If bpB+1 is not empty, its jobs are moved to bpB+2 and so on (see chromosome A^{*2} in Fig. 6 a); additionally actualized batch-positions are marked in italic). If $bpB + x = m \cdot bpm$ holds, the "search" for empty batch-positions starts with batch-position one and ends at the latest when bpA is reached (which must now be empty). This operator can have a high impact on a chromosome if many batch-positions are occupied because many jobs may change their batch-position.

j=	1	2	3	4	5	6	7	8	9	10	11
f=	X	Y	X	Z	Y	Z	Z	X	X	Y	X
A	2	5	2	3	8	1	3	6	6	7	9
A*1	2	5	2	3	8	1	3	4	4	7	9
A*2	3	5	3	4	8	1	4	2	2	7	9

a) J2Pbin

j=	1	2	3	4	5	6	7	8	9	10	11
f=	X	Y	X	Z	Y	Z	Z	X	X	Y	X
A	2	5	2	3	8	1	3	6	6	7	9
A*1	2	5	2	3	8	1	3	6	4	7	9
A*2	2	5	2	3	8	1	3	6	9	7	9

b) J2Pjin

Fig. 6: J2P batch insert and J2P job insert

The fifth mutation operator J2Pjin extracts a single job from one batch and inserts it into another batch that can already contain jobs (of the same family) or is empty. Again, we first randomly select a single gene *j* with probability P_{jin}^{J2P} , and the associated job family *jfA* and the assigned batch-position *bpA* is derived from that gene (e.g., *j*=9 with *jfA*=X and *bpA*=6; cf. , Fig. 6 b)). Then, all batch-positions associated with *jfA* and different from *bpA* are determined and combined with all empty batch-positions (e.g., 2 and 9 combined with 4, 10, and 11). From this set of batch-positions, the new batch-position *bpB* is randomly selected. If *bpB* is empty (e.g., *bpB*=4), the batch-position of *j* is updated (cf., chromosome A^{*1} in Fig. 6 b)). If *bpB* is not empty, a capacity feasibility check is required. If the check is positive, job *j* is inserted into *bpB* (see chromosome A^{*2} in Fig. 6 b)); otherwise, the chromosome remains unchanged. The impact of this operator on a chromosome is relatively low, as only a single job changes its batch-position. However, it can ”open” a new batch or “close” a batch (if *j* was the only job in batch *bpA* and is inserted into a batch already containing jobs).

5.2 GA-J2BRK

The schedule representation of GA-J2BRK requires two chromosomes and was also used by Huynh & Chien (2018) for a problem very similar to the PSBIJF. However, we propose several adaptations and enhancements.

5.2.1 Genotype representation and decoding

The first chromosome (INT) of the J2BRK genotype consists of *n* integer genes with one gene for each of the *n* jobs (gene indices are therefore simultaneously job indices *j*) and the alleles define the job-to-batch assignment by batch indices. In contrast to Huynh & Chien (2018), we do not group jobs by job families to avoid positional biases. The second chromosome (RK) consists of *n* real-valued genes (one for each possible batch) and the allele represent random keys from $[0, 1]$ indicating the batch-to-machine assignment and the sequence of batches on a machine. Fig. 7 shows the two chromosomes in part a).

Journal Pre-proofs

a)

	f =	X	Y	X	Z	Y	Z	Z	X	X	Y	X
	j =	1	2	3	4	5	6	7	8	9	10	11
INT		2	7	2	3	6	1	1	4	4	7	5
RK		.74	.43	.23	.53	.67	.17	.91	.34	.12	.83	.61
	b =	1	2	3	4	5	6	7	8	9	10	11
	f =	Z	X	Z	X	X	Y	Y				

b)

	Sequenced batches per machine (with jobs)				
***i*=1**	*9 ()*	*6 (5)*	*3 (4)*		
***i*=2**	*11 ()*	*4 (8, 9)*	*2 (1, 3)*		
***i*=3**	*5 (11)*	*1 (6, 7)*	*10 ()*	*7 (2, 10)*	

Fig. 7: Integer (INT) and random key (RK) chromosomes and decoding

The batch-to-machine assignment is decoded by dividing $[0,1]$ into *m* equal ranges (e.g., for *m*=3: $[0,1/m)$, $[1/m, 2/m)$, and $[2/m, 1]$) and batches with random keys in the first range are assigned to machine one, batches with random keys in the second range are assigned to machine two, and so on. The sequencing of batches on the machines is done by sorting the batches by their random keys. To avoid that small random key changes leading to machine assignment changes also completely changes the position of the changed batch and all other batches on the "new" machine, we change the batch sorting according to the machine index: On machines with an odd index, batches are sequencing in non-decreasing order of their random keys, whereas on machines with an even index, batches are sequencing in non-increasing order of their random keys. The decoding of the example is shown in Fig. 7 b).

A general advantage of this representation compared to the representation of GA-J2P is the smaller chromosome as it consists of only 2*n* genes compared to n^2 genes in the chromosome of GA-J2P. However, due to the "gene-reducing" parameter *bpm*, this advantage becomes smaller, and the handling of a single chromosome might be more efficient compared to the handling of two.

5.2.2 Initial population composition

To generate the initial population for GA-J2BRK, we use the same three approaches as before: *ipc* = *ran, icr*, or *iri*. For generating completely random initial solutions, we first randomly draw *n* integers from $\{1,\ldots,n\}$ for chromosome INT and *n* random keys from $[0, 1]$ for chromosome RK. As this will likely result in infeasible solutions regarding job families and batch capacities, we need a repair mechanism. For repairing, we check if the insertion of a job into a batch is feasible and if not, the corresponding allele in RK is incremented by one as long as insertion becomes feasible. Since this may lead to a "chain" of allele updates, the resulting genotype is very different from the initial one. However, as we just want to compute random initial solutions here, it does not matter.

The procedure for generating randomized initial solutions remains the same as before. We apply mutation and recombination operators iteratively, gradually increasing the mutation probabilities.

5.2.3 Elitism and survivor selection

For elitism and survivor selection, we use the same mechanisms as for GA-J2P with the control

parameters *esr*, *svr*, and *svs* to be tuned for GA-J2BRK.

5.2.4 Parent selection and recombination

To select the parents that are used for offspring generation, we also study the two selection mechanisms TOS (with s=2) and SUS, controlled by the parameter *pas*.

For recombing to genotypes A and B, Huynh & Chien (2018) used a job family related approach. They selected all genes of a randomly selected job family in genotype A (B), transferred these genes to offspring A* (B*), and transferred the genes of all other job families from B (A) to A* (B*). For the random key chromosome, they used a one-point crossover. In contrast to this job family related approach, we propose using adapted standard recombination mechanisms: "J2BRK adapted one-point crossover" (J2BRKaOPX) and "J2BRK adapted two-point crossover" (J2BRKa2PX).

J2BRKaOPX starts with the random selection of the crossover point from $\{1,\ldots,n\}$ to partition both chromosomes of genotypes A and B into head and tails (e.g., 6 in Fig. 8). First, the heads of the INT chromosomes and the complete RK chromosomes are transferred to the offsprings A* and B*. Next, the tail of both chromosomes of B (A) are transferred to A* (B*) in non-decreasing order of assigned batches (already existing random keys are overwritten). If a batch index to be transferred is already in use, the next unused batch index is used, and the random key is set accordingly. Note that this procedure does not affect job-to-batch and batch-to-machine assignments but only the batch sequence on a machine might change. Thus, J2BRKaOPX is able to preserve promising parts of both parent genotypes. In Fig. 8, allele values resulting from tail transfers are marked bold and gray shaded fields indicate values affected by the batch renumbering.

j=	1	2	3	4	5	6	7	8	9	10	11
f=	X	Y	X	Z	Y	Z	Z	X	X	Y	X
A.INT	2	4	2	3	8	**1**	**3**	**6**	**6**	**7**	**9**
A.RK	.74	.43	.23	.53	.67	.17	.91	.34	.12	.83	.61
						3	**5**	**8**	**2**	**4**	**2**
						3	5	6	1	4	2
A*.INT	2	4	2	3	8	**5**	**7**	**9**	**1**	**6**	**1**
A*.RK	**.23**	.43	.23	.53	**.45**	**.73**	**.27**	.34	**.21**	.83	.61
b=	1	2	3	4	5	6	7	8	9	10	11
f=	X	X	Z	Y	Z	Y	Z	Y	X		

j=	1	2	3	4	5	6	7	8	9	10	11
f=	X	Y	X	Z	Y	Z	Z	X	X	Y	X
B.INT	6	7	6	3	4	**3**	**5**	**8**	**2**	**4**	**2**
B.RK	.14	.23	.45	.73	.27	.92	.37	.21	.11	.96	.83
						1	**3**	**6**	**6**	**7**	**9**
						1	2	3	4	5	6
B*.INT	6	7	6	3	4	**1**	**5**	**8**	**8**	**9**	**10**
B*.RK	**.74**	.23	.45	.73	.53	.92	.37	.17	.83	.12	.83
b=	1	2	3	4	5	6	7	8	9	10	11
f=	Z		Z	Y	Z	X	Y	X	Y	X	

Fig. 8: The J2BRKaOPX recombination operator

The J2BRKa2PX recombination operator follows the same procedure except using two randomly selected crossover points from $\{1,\ldots,n\}$, first transfers the middle parts, and then completes the offspring genotypes accordingly.

5.2.5 Mutation

For mutation we propose using three standard operators, "J2BRK random resetting" (J2BRKrr), "J2BRK uniform" (J2BRKuni), and "J2BRK Gaussian" (J2BRKgau).

J2BRKrr is an adaptation of the standard random resetting mutation operator for integer chromosomes. It mutates each gene with given probability P_{rr}^{J2BRK} by drawing new allele values for the INT chromosome from $\{1,\ldots,n\}$. If one of the drawn batch indices is already in use, renumbering takes place. The random keys of chromosome RK remain unchanged.

Journal Pre-proofs

In contrast, J2BRKuni (also used by Huynh & Chien, 2018) and J2BRKgau keep job-to-batch assignments unchanged but random keys change. They mutate each gene with probability P_{uni}^{J2BRK} and P_{gau}^{J2BRK}, respectively. As the names suggest, they differ in the distribution new allele values are drawn from: J2BRKuni draws from a uniform distribution restricted by [0, 1], whereas J2BRKgau draws from a Gaussian (normal) distribution with the mean equal to the old random key and a standard deviation of $0.25/m$. The adjustment by m is made to make the probability that this mutation operator leads to a machine swap of a batch independent of the number of machines m. Obviously, the drawn values must be clamped to [0, 1]. The INT chromosome remains unchanged when J2BRKuni and J2BRKgau are applied.

In addition to these standard mutation operators, we developed four problem specific mutation operators like those for GA-J2P: "J2BRK batch swap" (J2BRKbsw), "J2BRK job swap" (J2BRKjsw), "J2BRK batch insert" (J2BRKbin), and "J2BRK job insert" (J2BRKjin).

J2BRKbsw swaps two complete batches of arbitrary job families by randomly selecting two genes (with probability P_{bsw}^{J2BRK}) and swapping the batch indices for all jobs assigned to one of these indices. Since both batches were feasible in terms of job family and batch capacity, the swapping must be feasible. RK remains unchanged.

As for J2Pjsw (cf., Fig. 5 b)), J2BRKjsw first randomly selects a single gene from chromosome INT with probability P_{jsw}^{J2BRK}, then determines all genes from INT with jobs of the same family and that are not in the same batch, and finally randomly selects one of these genes. If swapping the two jobs between the assigned batches is feasible in terms of batch capacity, the batch indices are swapped, otherwise, the chromosome remains unchanged. Random keys are always left unchanged. A similar mutation operator is also used by Huynh & Chien (2018).

The mutation operator J2BRKbin inserts a complete batch at a new position. To this, a gene from INT is selected with probability P_{bin}^{J2BRK} and the assigned batch bA is determined. Then, the new batch bB is drawn from $\{1,\ldots,n\}$. If bB equals bA, no insertion is done. If batch bB is empty, all jobs are inserted into bB by updating the corresponding genes of chromosome INT. If bB is not empty, batch bA is inserted before bB by setting the random key of bA to the random key of bB decremented by a very small number. This operator can be very influential as a new batch may be created and batch positions on a machine may change.

J2BRKjin extracts a single job from one batch and inserts it into another batch that can already contain jobs (of the same family) or is empty. We first randomly select a gene (job) j from INT (with probability P_{jin}^{J2BRK}) and determine its job family jfA. Then, all batches also containing jobs of family jfA are combined with all empty batches. From this set of batches, the new batch bB is randomly selected. If bB is empty, the batch index of j is updated to bB. If bB is not empty, a capacity feasibility check is required. and if it is positive, job j is inserted into bB. Otherwise, the chromosome INT remains unchanged (RK remains unchanged in any case). The effect of this operator on INT is relatively small because only a single job-to-batch assignment is changed. However, it can "open" a new batch or "close" a batch if j was the only job in batch bA and is inserted into a batch that already contains jobs.

5.3 Task parallel implementation

As the task parallel implementation of the learning augmented heuristics proposed in Uzunoglu et al. (2023b) resulted in a remarkably speedup, we follow this approach not only in the

implementation of the heuristics computing initial solutions but also for our GAs. Similar to the multi-start of the heuristics, we start several evolution engines in parallel threads. This means, that several identically parametrized GA are executed and the best result of all of them is returned. As the results of Uzunoglu et al. (2023b) showed that the greatest efficiency gains are achieved with four parallel threads, we also use four threads in our experiments.

6 Experimental setup

To perform training, validation (hyper-parameter tuning), GA parameter tuning, and testing, we need to prepare the required data and ML models. Note that all used data sets are either already available for download or are added to a new data set on Mendeley Data (*reference to be added*). In addition, this section describes the performance metric used to assess the quality of the solution and some implementation details.

6.1 Problem instances and data sets

For training good-performing ML models with a good generalization and prediction accuracy, it is vital to have a sufficiently large set of problem instances. Furthermore, to validate models' applicability to real-world problems, the instances should also be realistic, diverse (i.e., representing different scenarios), and present a challenging learning task. The data set provided by Gahm (2022) fulfills these requirements. It contains three sets of instances that differ in the number of jobs n and machines m. As we are interested realistic-large problem instances, we use the sets L (containing 57,600 large instances with $n \in \{100, 200, 400\}$ and $m \in \{1, 3, 4, 5, 10\}$) and XL (containing 13,440 large instances with $n \in \{800, 1,600, 3,200\}$ and $m \in \{5, 10, 20\}$) to define the problem space P for this paper ($|P| = 71{,}040$). All generated instances base on nine attributes (like n or m) and for each attribute combination, five instances are randomly generated (thus, each instance has a marker $s \in \{1, 2, \ldots, 5\}$). Based on the marker, we can define different data set for training, validation (hyper-parameter tuning), GA parameter tuning, and testing. To represent these subsets, we use the notation D_s (e.g., $D_{1,2}$ consists of all instances with the markers 1 and 2).

6.2 Model training for the learning augmented heuristics

Since all 17 algorithms from algorithm space A use ML models to predict parameters in one way or another, we must train these models. Because hyper-parameter tuning and validation was already done in the referenced publications, we can use the instance subset $D_{1,2,3,4}$ for training the models. For future use, we provide all 17 models (and scaling data) for download (*reference to be added*).

6.3 Performance metric

Comparing the performance of algorithms relative to the best solution could be misleading if objective values tend to get close (or equal to) zero (like for the weighted tardiness criterion of the PSBIJF). In these cases, small absolute deviations to zero result in huge relative deviations, which may distort the analysis even with a large number of experiments. To get a robust metric for our analysis, we use the key figure "mean relative improvement versus the worst objective function value" (MRIW; cf., Valente & Schaller, 2012 and Gahm et al., 2022). The relative improvement to the worst objective value for a problem instance $RIW_{a,i}$ bases on the following definitions: let $A' \subseteq A$ be the (sub)set of all algorithms to be analyzed, let $ov_{a,i}$ be the objective

value computed by algorithm $a \in A'$ for problem instance $i \in P'$ (with $P' \subseteq P$), and let $ov_i^{worst} = \max_{a \in A'} ov_{a,i}$ be the worst achieved objective value by one of the algorithms (note that ov_i^{worst} is not an intermediate solution or the worst candidate objective value observed during the execution of an algorithm but the worst final solution of one of the considered algorithms). With these definitions, we define $RIW_{a,i} = (ov_i^{worst} - ov_{a,i}) / ov_i^{worst}$ if $ov_i^{worst} > 0$ and otherwise, $RIW_{a,i} = 0$. Aggregating these values using the mean over all instances of interest (P') for an algorithm a gives us the $MRIW_a$. Here, it is fundamentally important that $MRIW_a$-values can only be compared to each other if the same (sub)set of algorithms A' (and instances) is used for their computation.

6.4 Implementation

The training, hyper-parameter tuning, validation, and testing of all ML models is implemented in Python and uses "Keras" (keras.io), "TensorFlow" and "TensorFlow Ranking (tensorflow.org), "scikit-learn" (Pedregosa et al., 2011), and XGBoost (xgboost.ai). All heuristics, the GAs, and a tool for the management of the experiments are implemented in Java 10. For the GA implementations, we use the Jenetics framework (jenetics.io) and for task-parallel execution of the learning-augmented heuristics and the GAs, we use the "Parallel Java 2 PJ2" API (available at http://jimihford.github.io/pj2/).

The machine learning and all experiments have been executed on workstations with an Intel® XEON® CPU E5-2690 with 3.0 GHz and 64 GB RAM.

7 Parameter tuning

Parameter tuning is a very important step for algorithms to achieve their best performance. This is particularly true for ML models where this process is called hyper-parameter tuning. The hyper-parameter tuning comes along with the validation of ASP models, i.e., the selection of the best performing ML model type, labeling strategy, and loss function combination. This is followed by the parameter tuning for GA-J2P and GA-J2BRK.

7.1 ASP model hyper-parameter tuning

The hyper-parameter tuning of the ASP models uses the dataset $D_{1,2}$ for training and D_3 for validation (i.e., to determine the most suitable one). For the 15 NN-based ASP models, we use NNs with three hidden layers and evaluated the following hyper-parameters: 256, 512, and 1,024 neurons for all three layers independently, the dropout rates 0.1, 0.2, and 0.3, and the L2 weight regularization rates 0.001, 0.002, and 0.003. In total, 243 ($3^3 \cdot 3 \cdot 3$) hyper-parameter combinations have to be evaluated to determine the best configuration for each NN-based ASP model. For the 14 GBDT-based ASP models, we evaluated the following hyper-parameters: number of estimators 250, 350, and 450, learning rates 0.01 and 0.001, maximum depth 8, 9, and 10, minimal child weights 1 and 2, subsampling rates 0.5 and 0.7, column sampling rates 0.5 and 0.7. In total, 144 (3·2·3·2·2·2) hyper-parameter combinations must be evaluated.

Because the ASP models use different loss functions, we cannot use them for the validation of the most suitable ASP model. Therefore, we introduce the metric "hit rate", quantifying the ratio of how often the model's proposed (best) algorithm was in fact one of the best algorithms for a problem instance. Table 3 depicts the best hit scores for every ML model, labeling strategy, and loss function after tuning the hyper-parameters (see Appendix A-2 Table

13 and Table 14 for best hyper-parameters for each ML model, labeling strategy, and loss function combination).

Table 3: Hit rates by ML models, labeling strategies, and loss functions

ML model type	Labeling strategy	Loss function	Hit rate [%]	MRIW [%]
NN	BIN	BCE	70.16	40.33
	OV	MSE	57.09	36.38
	OV-N	MSE	64.92	39.61
	RANK	MSE	66.31	39.89
		iwMSE(1)	68.94	40.31
		iwMSE(2)	68.98	40.39
		iwMSE(3)	61.77	38.98
		ApproxNDCG	68.28	39.45
		LambdaRankNDCG	67.46	39.27
		LambdaRankNDCG(1)	67.91	39.34
	RANK-SC	MSE	65.13	39.71
		iwMSE(1)	65.70	39.72
		iwMSE(2)	68.00	40.04
		iwMSE(3)	64.23	38.46
		ApproxNDCG	64.75	39.34

Journal Pre-proofs

Journal Pre-proofs

GBDT	BIN	BCE	73.44	41.14
	BIN	LambdaBinary	69.93	40.59
	OV	MSE	67.37	39.84
	OV-N	MSE	71.90	41.22
	RANK	MSE	72.99	41.12
		iwMSE(1)	73.11	41.10
		iwMSE(2)	73.07	41.07
		iwMSE(3)	72.99	41.08
		RankPairwise	71.13	40.77
		LambdaMARTNDCG	70.26	40.57
	RANK-SC	MSE	72.99	41.12
		iwMSE(1)	73.08	41.13
		iwMSE(2)	73.18	41.15
		iwMSE(3)	73.24	41.14

The results in Table 3 show that GBDT as basic ML model type generally outperforms NN, which is not surprising since GBDT is known to outperform other models on tabular data (in contrast to image data or text documents; Shwartz-Ziv & Armon, 2022). For both model types, representing the ranking problem as a binary classification achieves the highest hit scores. It is also evident, that using (pure) objective value labeling strategies (see OV rows in Table 3) perform worse than all other labeling strategies for NN and GBDT, respectively. Consequently, we conclude to ask ML models for a ranking of algorithms is better than to ask for objective values for each of the algorithm in A. For the NN, the two best-performing settings use the labeling strategies BIN with training strategy BCE and RANK combined with the custom loss function iwMSE(α). Interestingly, the custom loss function outperforms MSE and the scaling

factor α seems to play an important role as the hit rate decreases from 68.98% to 61.77% when α changes from 2 to 3. This indicates that tuning the α-value is important. For GBDT, the two best-performing settings are BINARY labels with BCE and RANK-SC labels with the custom loss function iwMSE(α). In contrast to the NN, the GBDT seems to be less sensitive to changes in the α-value. However, tuning the α-value is still important. Overall, the GBDT model type achieves more consistent hit rates throughout the different methods. The results also show us that the scaled RANK version RANK-SC performs better for GBDT, while this is not the case for NN. Somewhat surprisingly, the "simple" binary encoding BIN combined with BCE for both GBDT and NN outperforms more sophisticated alternatives (such as LambdaRankNDCG or ApproxNDCG) with respect to the ASP at hand. The only loss function that achieves a similar performance is the newly introduced iwMSE(α) function. An advantage of iwMSE(α) is its less complex implementation compared to the more sophisticated ones. However, further studies have to examine if iwMSE(α) is generally suitable for ranking predictions.

The best determined hyper-parameters for the four best ASP models show that both model types prefer higher capacity models in almost every experiment. To achieve a maximum prediction performance, we performed a second hyper-parameter tuning on the best four ASP models with extended capacities but omitted certain hyper-parameters for the GBDT since they did not lead to accurate models (e.g., 250 and 350 as number of estimators or 8 as maximum depth) For GBDT models we define the hyper-parameter space as: 450 or 550 for number of estimators, 9 or 10 for maximal depth, 0.001 or 0.01 for the learning rate, 1 or 2 for minimum child weight, and 0.5 or 0.7 for subsampling rates for data points and columns. For the NN, we increase the number of neurons in the hidden layers (all three hidden layers with 256, 512, 1,024, or 2,048 neurons, dropout rate of 0.1 or 0.2, and regularization rates of 0.001, 0.002, or 0.003). To further improve prediction accuracy, in the second hyper-parameter tuning, training is performed the data set $D_{1,2,3}$ and validation on D_4.

Table 4 Results of second hyper-parameter tuning

ML model type	Labeling strategy	Loss function	Hit rate [%]	MRIW [%]
NN	BIN	BCE	64.75	40.95
NN	RANK	iwMSE(2)	61.87	40.54
GBDT	BIN	BCE	67.67	41.56
GBDT	RANK-SC	iwMSE(3)	66.57	41.43

Table 4 shows that overall, the hit rate in the second hyper-parameter tuning slightly decreases due to the new validation data set D_4. Nevertheless, the results confirm that GBDT-RANK-SC-iwMSE(3) is competitive and that GBDT-BIN-BCE outperforms the other ASP models (see Appendix A-3 Table 15 and Table 16 for best-performing hyper-parameters). In consequence, *GBDT-BIN-BCE* is used to select the learning-augmented heuristics for initial solution computation in all following experiments.

7.2 GA-J2P parameter tuning

Journal Pre-proofs

Because of the large number of parameters and values for GA-2JP, we perform a two-step tuning to keep experiments at a manageable level. For the parameter tuning of GA-J2P (and also GA-J2BRK), we use a reduced set D_{PT} of 30 instances, i.e., five randomly selected instances for each $n \in \{100, 200, 400, 800, 1{,}600, 3{,}200\}$ from D_4.

In parameter tuning step one, we are interested in the general behavior of GA-J2P for example regarding the initial population composition or the effect of the elitism mechanism. The parameters and their investigated values as listed in Table 5 are used to setup 51,840 experiments in the first study GA-J2P-PT1 (1,728 parameter configurations·30 problem instances from D_{PT}). For termination, we use setting TS1 (see Section 5). Because the MRIW of GA-J2P without initial (*ipc*=*ran*) solutions is so much worse compared to the other ones (33.8% vs. 93.7% and 93.7%), we remove these results from further considerations (and MRIW computations) to make the differences between parameter values more traceable. To simplify readings we introduce $P^{J2P}_{cust.}$ for representing identical probabilities for P^{J2P}_{bsw}, P^{J2P}_{jsw}, P^{J2P}_{bin}, and P^{J2P}_{jin}. In Table 5, bold values indicate values to be further investigated in step two and bold and underlined values indicate values that are fixed based on the results.

Table 5: Parameters and results of study GA-J2P-PT1

Parameters and values		# values	MRIW [%]
bpm	*n* vs. **1.2(*n/m*)**	2	19.9520.88
ipc	(*ran* vs.) *icr* vs. ***<u>iri</u>***	3	19.98 20.85
μ	**200** vs. 400	2	20.6020.23
esr	0 vs. **<u>0.05</u>**	2	19.2621.57
svr	**0.2** vs. 0.4	2	19.8820.94
svs	TOS(2) vs. **<u>SUS</u>**	2	20.4020.43
pas	TOS(2) vs. **<u>SUS</u>**	2	20.2420.59
rec.	P^{J2P}_{aOPX}=0.2, P^{J2P}_{a2PX}=0.0 vs. P^{J2P}_{aOPX}=**<u>0.0</u>**, P^{J2P}_{a2PX}=**0.2** vs. P^{J2P}_{aOPX}=0.2, P^{J2P}_{a2PX}=0.2	3	20.6421.08 19.53
mut.	P^{J2P}_{rr}=0.02, P^{J2P}_{cust}=0.00 vs. P^{J2P}_{rr}=**<u>0.00</u>**, P^{J2P}_{cust}=**0.02** vs. P^{J2P}_{rr}=0.02, P^{J2P}_{cust}=0.02	3	17.1522.25 21.85
Total number of parameter configurations:		1,728	

The results in Table 5 show that reducing the number of batch positions has a remarkable effect on the solution quality and thus, we conclude that different parameter values are worth

Journal Pre-proofs

exploring in the second parameter tuning step. As already mentioned, the standalone execution of GA-J2P without initial solutions is so much worse that we do not investigate this approach further. Furthermore, since the randomized initial solutions outperform initial populations with fully randomized solutions, we fix the parameter *ipc* to *iri*. We also fix the parameters *esr* to 0.05 as larger values are not used in literature and therefore perceived as unfavorable, and *svs* to SUS and *pas* to SUS as results are almost similar. Because smaller populations sizes (leading to more generations) seem to be favorable, we will explore different settings for μ in the next tuning step. Regarding recombination, we see that J2Pa2PX outperforms J2PaOPX, and thus, we only tune P_{a2PX}^{J2P} in the second step. For mutation, we can see that the problem specific mutation operators J2Pbsw, J2Pjsw, J2Pbin, and J2Pjin perform best when solely applied. As the survival rate (*svr*) is directly related to recombination and mutation probabilities, we investigate both *esr* values in step two.

In the second parameter tuning study (GA-J2P-PT2), we use 540 parameter configurations to define 16,200 experiments (see Table 6). To ensure a feasible transfer of initial solutions to the genotype representation, we must guarantee that the number of batches on each machine is not smaller than in any initial solution. Therefore, we calculate the maximum value of used batches per machine of all solution bpm_m^{MAX} and adjust the finally available batches-positions per machine accordingly. In addition, we use this value to define two additional settings.

Table 6: Parameters and results of study GA-J2P-PT2

	Parameters and values	#values	MRIW [%]
bpm	**max{n/m, bpm_m^{MAX}}** vs. max{$1.1n/m$, bpm_m^{MAX}} vs. max{$1.2n/m$, bpm_m^{MAX}} vs. $1.1\,bpm_m^{MAX}$, $1.2\,bpm_m^{MAX}$	5	8.07 7.62 7.91 7.88 7.75
μ	**150** vs. 200 vs. 250	3	7.92 7.89 7.72
svr	0.2 vs. **0.4**	2	7.43 8.27
rec.	P_{a2PX}^{J2P} = **0.2** vs. 0.3 vs. 0.4	3	9.44 7.76 6.33
mut.	P_{cust}^{J2P} = 0.02 vs. 0.04 vs. **0.08** vs. 0.12 vs. 0.16 vs. 0.20	6	6.91 7.68 8.42 8.30 8.08 7.69
	Total number of parameter configurations:	540	

Based on the results of the parameter tuning studies GA-J2P-PT1 and GA-J2P-PT2, we fix the parameters of GA-J2P as follows: batch-positions per machine *bpm* = max{n/m, bpm_m^{MAX}}, initial population composition *ipc*=*iri*, population size μ =*150,* elitism rate *esr* =0.05, survival rate *svr*=0.4, survivor selection mechanism *svs*=SUS, parent selection

mechanism *pas*=SUS, recombination probability P^{J2P}_{a2PX}=0.2, and mutation probabilities P^{J2P}_{bsw} = $P^{J2P}_{jsw} = P^{J2P}_{bin} = P^{J2P}_{jin}$ =0.08 (all other operator probabilities are set to 0).

7.3 GA-J2BRK parameter tuning

We also perform a two-step parameter tuning for GA-J2BRK and instance set D_{PT}. The parameters and their investigated values as listed in Table 7 are used to setup 60,480 experiments in study GA-J2BRK-PT1. For termination, we use setting TS1 (see the beginning of section 5). The version of GA-J2BRK without initial solutions (*ipc*=*ran*) is much worse compared to the other ones (22.7% vs. 89.1% and 85.1%) and therefore, we remove the corresponding results from further considerations and MRIW computations. In Table 7, bold values indicate values to be further investigated in step two and bold and underlined values that are fixed based on the results. To simplify readings we introduce $P^{J2BRK}_{cust.}$ for representing identical probabilities for P^{J2BRK}_{bsw}, P^{J2BRK}_{jsw}, P^{J2BRK}_{bin}, and P^{J2BRK}_{jin}. Similarly, $P^{J2BRK}_{uni,gau}$ represents identical probabilities for P^{J2BRK}_{uni} and P^{J2BRK}_{gau}.

Table 7: Parameters and results of study GA-J2BRK-PT1

	Parameters and values	# values	MRIW [%]
ipc	(*cr* vs.) *icr* vs. ***iri***	3	23.59 24.15
μ	200 vs. **400**	2	23.8223.92
esr	0 vs. **0.05**	2	23.0024.74
svr	0.2 vs. **0.4**	2	23.4824.26
svs	TOS(2) vs. **SUS**	2	23.8023.94
pas	**TOS(2)** vs. SUS	2	23.9123.83
rec.	P^{J2BRK}_{aOPX} =0.2 P^{J2BRK}_{a2PX} =0.0 vs. P^{J2BRK}_{aOPX} **=0.0** P^{J2BRK}_{a2PX} **=0.2** vs. P^{J2BRK}_{aOPX} =0.2 P^{J2BRK}_{a2PX} =0.2	3	23.6124.33 23.67
mut.	P^{J2BRK}_{rr} =0.02 $P^{J2BRK}_{cust.}$ =0.00 $P^{J2BRK}_{uni,gau}$ =0.00 vs. P^{J2BRK}_{rr} =0.00 $P^{J2BRK}_{cust.}$ **=0.02** $P^{J2BRK}_{uni,gau}$ =0.00 vs. P^{J2BRK}_{rr} =0.00 $P^{J2BRK}_{cust.}$ =0.00 $P^{J2BRK}_{uni,gau}$ **=0.02**	3	23.8923.94 24.03

Journal Pre-proofs

vs.	P^{J2BRK}_{rr} =0.02 $P^{J2BRK}_{cust.}$ =0.02 $P^{J2BRK}_{uni,gau}$ =0.00	vs.	P^{J2BRK}_{rr} =0.02 $P^{J2BRK}_{cust.}$ =0.00 $P^{J2BRK}_{uni,gau}$ =0.02	2	23.7923.88
vs.	P^{J2BRK}_{rr} =0.00 $P^{J2BRK}_{cust.}$ =0.02 $P^{J2BRK}_{uni,gau}$ =0.02	vs.	P^{J2BRK}_{rr} =0.02 $P^{J2BRK}_{cust.}$ =0.02 $P^{J2BRK}_{uni,gau}$ =0.02	2	23.9223.64
Total number of parameter configurations: 2,016					

The results in Table 7 support the previous observation that randomized initial solutions are preferable compared to completely randomly generated solutions for initial population composition. In contrast to GA-J2P, GA-J2BRK performs better with larger populations and μ will be tuned in the second step. For the parameters *esr*, *svr*, *svs*, and *pas*, the results are very similar to those from GA-J2P and thus, we proceed as before. The recombination by J2BRKa2PX leads to better results compared to J2BRKaOPX, accordingly, we tune the corresponding probability P^{J2BRK}_{a2PX} in the second step. The results for the mutation operator are not as clear as for GA-J2P because the MRIWs are close together. However as the setting with P^{J2BRK}_{rr} = 0.00, $P^{J2BRK}_{cust.}$ =0.02, and $P^{J2BRK}_{uni,gau}$ =0.0 (23.94%) and the setting with P^{J2BRK}_{rr} =0.00, $P^{J2BRK}_{cust.}$ =0.00, and $P^{J2BRK}_{uni,gau}$ =0.02 (24.03%) perform best, we are going to tune $P^{J2BRK}_{cust.}$ and $P^{J2BRK}_{uni,gau}$ independently of each other in tuning step two. The investigated parameter values are summarized in Table 8.

Table 8: Parameters and results of study GA-J2BRK-PT2

Parameters and values		# values	MRIW [%]
μ	300 vs. **350** vs. 400 vs. 450	4	1.291.311.281.26
svr	0.2 vs. **0.4**	2	1.261.31
rec.	P^{J2BRK}_{aOPX} = **0.2** vs. 0.3 vs 0.4	3	1.361.301.20

Journal Pre-proofs

	P_{cust}^{J2BRK} = 0.02 vs. 0.04 vs. 0.08 vs. 0.12 vs. 0.16 (with P_{rr}^{J2BRK} =0.00 and $P_{uni,gau}^{J2BRK}$ =0.00)	10	1.271.301.231.27 1.24
mut.	$P_{uni,gau}^{J2BRK}$ = **<u>0.02</u>** vs. 0.04 vs. 0.08 vs. 0.12 vs. 0.16 (with P_{rr}^{J2BRK} =0.00 and P_{cust}^{J2BRK} =0.00)		1.371.311.291.31 1.26
	Total number of parameter configurations:	240	

The results in Table 8 clearly indicate most suitable parameters. The better performance of $P_{uni,gau}^{J2BRK}$ = 0.02 compared to all other mutation settings can be traced back to be the most efficient one and leads to the highest number of generations in the given time limit.

Based on the overall results of the parameter tuning studies GA-J2BRK-PT1 and GA-J2BRK-PT2, we fix the parameters of GA-J2BRK as follows: initial population composition *ipc*=*iri*, population size μ *=350,* elitism rate *esr*=0.05, survival rate *svr*=0.4, survivor selection mechanism *svs*=SUS, parent selection mechanism *pas*=TOS, recombination probability P_{a2PX}^{J2BRK} =0.2, and mutation probability $P_{uni,gau}^{J2BRK}$ =0.08 (all other operator probabilities are set to 0).

8 Experimental results

For the final testing of our developed algorithms, we reserved instance set D_5 (20% of the instances, which was not used in either of the previous training or tuning phases) for assessing the solution quality on unseen problem instances. We use the MRIW metric to compare the following six solution methods:

- **BATCS-d-MLPP(PC$_6$, GS3)** has the highest MRIW score among the 17 solution methods forming the base for the ASP (see section 2.2). We use this solution method as a baseline to compare the effectiveness of our newly developed solution methods.

- **AlgSel(GBDT, BIN, BCE)** has the highest hit rate according to our analysis in section 7.1. We trained the model on $D_{1,2,3,4}$ to predict the best performing LACH for each problem instance from section 2.2 and used that algorithm to solve it.

- **BATCS-d-ML(PC$_6$, GS3)-LS(TS2)** is the most efficient heuristic from the literature for solving the PSBIJF (Uzunoglu et al., 2023b). In contrast to Uzunoglu et al. (2023b), we use the same time-oriented termination setting as for our GAs (TS2) to make a fair comparison (combined with the two parameters $\varepsilon^{WT} = 1.0 \times 10^{-8}$ and $\varepsilon^{FT} = 1.0 \times 10^{-8}$ to avoid needles computation times; see Uzunoglu et al., 2023b for more details).

- **GA-J2BRK** with the parameters *ipc*=*iri*, μ *=350,* *esr*=0.05, *svr*=0.4, *svs*=SUS, *pas*=TOS, P_{a2PX}^{J2BRK} =0.2, and $P_{uni,gau}^{J2BRK}$ =0.08 (cf., section 7.3).

- **GA-J2BRK-HC** is our GA-J2BRK implementation with parameters making our GA most similar to GA1 of Huynh & Chien (2018) (without implementing the makespan

related improvement heuristics). Parameters are as follows: ipc=iri, μ =150, esr=0.3, svr=0.4, svs=TOS, pas=TOS, P_{a2PX}^{J2BRK} =0.9, and $P_{jsw}^{J2BRK} = P_{uni}^{J2BRK}$ =0.1.

- **GA-J2P** with the parameters batch-positions per machine bpm = max{n/m, bpm_m^{MAX}}, ipc=iri, μ =150, esr=0.05, svr=0.4, svs=SUS, pas=SUS, P_{a2PX}^{J2P} =0.2, and $P_{bsw}^{J2P} = P_{jsw}^{J2P} = P_{bin}^{J2P} = P_{jin}^{J2P}$ =0.08 (cf., section 7.2).

Both GAs used the AlgSel(GBDT, BIN, BCE) to create their initial population and terminated their computations after reaching the time limits as defined in TS2 (see the beginning of section 5). So, our algorithm set A' consists of six solution methods for the computation of the MRIW. Table 9 presents the MRIWs of all six methods grouped according to the number of jobs (n) and number of machines (m). Certain (n, m) groups do not exist in the used data set and hence are left blank in the table.

Our ML-based algorithm selection method outperforms the “statically” chosen best solution method in every group (n, m). Averaged over all instances, it achieves an MRIW of 6.42% compared to the static best solution method’s MRIW of 2.23%. The performance improvement is higher for instances with fewer jobs and decreases for very large instances, but nonetheless exists. This justifies the efforts of training a model to rank solution methods if several are available for the problem at hand. Both GAs create their initial population using the AlgSel(GBDT-BIN-BCE) and hence achieve, as expected, substantially higher performances in each group (n, m) than the algorithm selection method without post-optimization.

Table 9: MRIWs [%] per solution method, number of jobs, and machines.

n=	100	200	400	800	1,600	3,200	MEAN
BATCS-d-MLPP (PC_6, GS3)							
m=1	2.02	0.85	1.76				1.54
m=3	2.55	2.29	1.78				2.21
m=4	2.60	2.47	1.80				2.29
m=5	2.19	2.24	2.05	1.63			2.05
m=10	2.33	2.32	2.61	2.45	1.95		2.37
m=20				3.13	4.25	3.48	3.56
MEAN	2.34	2.03	2.00	2.40	3.10	3.48	2.23
AlgSel(GBDT, BIN, BCE)							
m=1	6.71	6.60	6.92				6.74

m=3	4.62	5.55	4.93				5.03
m=4	5.99	6.53	5.72				6.08
m=5	5.92	5.96	5.25	4.51			5.47
m=10	9.48	9.22	8.01	5.68	5.63		7.95
m=20				9.79	5.25	3.65	7.25
MEAN	6.54	6.77	6.17	6.66	5.44	3.65	6.42

BATCS-d-MLPP (PC$_6$, GS3)-LS(TS2)

m=1	10.85	14.71	19.37				14.98
m=3	14.16	16.39	18.72				16.43
m=4	15.92	18.27	19.58				17.92
m=5	17.97	18.37	19.79	17.92			18.55
m=10	22.90	24.68	25.28	20.44	20.85		23.20
m=20				26.45	23.97	18.90	24.37
MEAN	16.36	18.48	20.55	21.60	22.41	18.90	19.07

GA-J2BRK

m=1	11.31	10.00	10.17				10.49
m=3	13.19	12.06	9.70				11.65
m=4	15.80	14.40	11.40				13.86
m=5	16.92	14.11	12.72	8.66			13.40
m=10	22.24	20.05	16.65	12.48	11.55		17.43
m=20				19.09	13.87	9.66	15.78
MEAN	15.89	14.12	12.13	13.41	12.71	9.66	13.84

GA-J2BRK-HC

m=1	12.33	10.20	10.59				11.04
m=3	14.44	12.50	9.98				12.31
m=4	16.92	14.59	11.18				14.23
m=5	18.38	14.59	12.50	8.52			13.83

Journal Pre-proofs

m=10	23.28	20.84	16.46	11.84	10.13		17.53
m=20				18.79	12.01	4.72	14.19
MEAN	17.07	14.54	12.14	13.05	11.07	4.72	14.07

GA-J2P

m=1	47.42	49.87	47.56				48.28
m=3	41.51	40.35	34.57				38.81
m=4	43.07	40.60	33.43				39.03
m=5	44.83	39.05	31.66	30.61			36.93
m=10	51.72	45.63	36.71	28.70	24.57		39.50
m=20				32.69	23.18	16.52	26.83
MEAN	45.71	43.10	36.79	30.67	23.88	16.52	39.19

Interestingly, a clear winner exists between three GAs. GA-J2P outperforms GA-J2BRK and GA-J2BRK-CH in each group. The greatest difference in performance improvement between the GAs can be observed for small instances (e.g., n = 100 or n = 200) but applies in principle to all instances as the difference of mean improvements of 25.35 and 25.12 percentage points show. Across the different machine settings (m), the three GAs performed (rather) similarly. One exception to that is the performance of GA-J2P for m = 20. From m = 10 to m = 20, the MRIW severely drops from 39.50% to 26.83%. The overall better performance of GA-J2P compared to the other GAs may be partly due to the higher number of generations performed by GA-J2P within the time limit. This higher number of generations in turn results from the "smaller" genotype GA-J2P (1.04n on average) compared to GA-J2BRK (2n) and GA-J2BRK-CH (2n) and the resulting lower computational effort for encoding, decoding, and operator execution.

Also interesting is that BATCS-d-MLPP (PC6, GS3)-LS(TS2) is able to outperform GA-J2P for large instances with 3,200 jobs and 20 machines. In these instances, we can see that the GA has not yet converged, and the evolution is still going on. This observation is confirmed by the fact that the minimum objective value found by GA-J2P is computed after 92.97% of the time limit (on average for instances with 3,200 jobs; see Table 10) and exemplarily illustrated by the courses of objective values illustrated in Fig. 9 for two instances. The courses also show that the evolution mechanisms of GA-J2P (with the tuned parameters) are capable to avoid premature convergence.

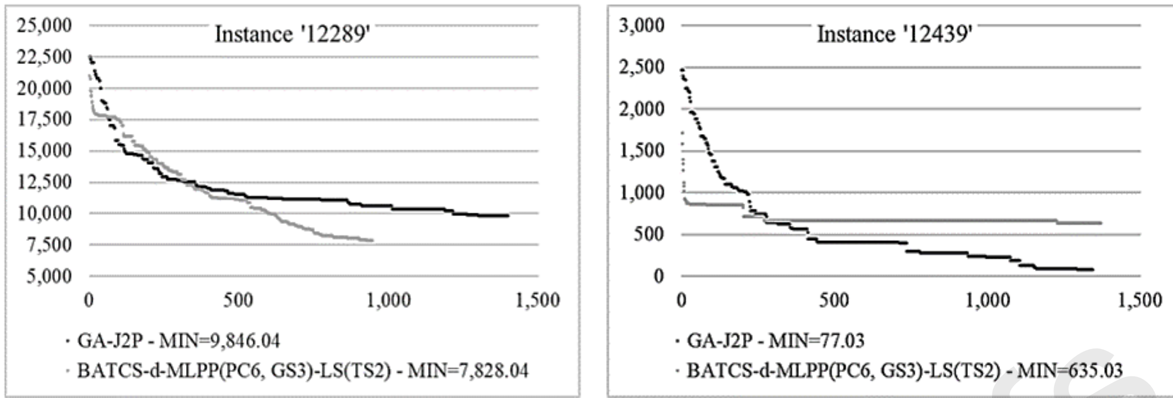


Fig. 9: Exemplary courses of objective values for two instances with 3,200 jobs

On the left side of Fig. 9, we can exemplarily see that BATCS-d-MLPP (PC6, GS3)-LS(TS2) is able to solve some large instances with a higher solution quality in less computation time (976 seconds vs. 1,442 seconds) compared to GA-J2P. However, as can be seen by comparing the MRIWS of Table 9 and the values in Table 10, BATCS-d-MLPP (PC6, GS3)-LS(TS2) often gets stuck in a local optimum and is unable to explore the complete solution space.

Table 10: Mean relative percentage of the time limit in which minimum objective values are computed

n=	100	200	400	800	1,600	3,200
BATCS-d-MLPP(PC6, GS3)-LS(TS2)	0.15	3.92	23.09	35.02	66.61	77.80
GA-J2P	59.52	79.67	87.06	88.92	95.02	92.97

Journal Pre-proofs

In consequence, the overall performance of GA-J2P is much better (see Table 9). From these results it can also be concluded that a combination of the two solution methods to a Memetic Algorithm is worth investigating in the future.

Error! Not a valid bookmark self-reference. shows the MRIW grouped for each instance characteristic (see Gahm et al., 2022 or Uzunoglu et al., 2023b for detailed description of instance characteristics).

Table 11: Results of GA-J2P by instance characteristics

Instance characteristic		MRIWs [%]					
n	100, 200, 400, 800, 1600, 3200	45.71	43.10	36.79	30.67	23.88	16.52
m	1, 3, 4, 5, 10, 20	48.28	38.81	39.03	36.93	39.50	26.83

q	3, 5, 10, 20, 40	37.05 39.54 41.55 42.74 24.75
jtfam	ND, UD	39.08 39.29
crs	1, 2, 3, 4	49.36 40.40 33.97 33.02
st	AR, AE, SE	38.45 40.27 38.84
eta	0.25, 0.75	39.70 38.67
tf	0.3, 0.6	40.66 37.72
rdd	0.5, 2.5	41.62 36.75

The job to family assignment mode (*jtfam*), setup time allocation (*st*), setup time severity(*eta*), tightness factor (*tf*), and due date range factor (*rdd*) seem to have minor effects on the performance of GA-J2P. The capacity requirement scenarios (*crs*) define the ranges of a job's capacity demand and have a noticeable effect on the MRIW. Also, the MRIW for the number of job families (q) drops immensely for $q = 40$. The reason for this is that this number of job families only exists for $n = 3200$ and the lower performance of GA-J2P for these instances has already been discussed. Due to its coupling to the job size, it is unclear how much of this effect should be attributed to the characteristic itself. An explanation for the trend of decreasing improvement with increasing job size might be the coupling between job size and the "complexity" of the problem: large instances need more time for their mutation and repair mechanisms, and therefore, GAs can search for solution improvements for these instances less intensely. However, this situation needs a deeper understanding of its causes and must be analyzed more nuancedly in future research.

9 Conclusions and further research directions

In this paper, we presented two learning-based contributions to both ends of using a heuristic to solve a complex scheduling problem. First, we addressed an issue appearing before using a heuristic – selecting the most suitable method. The ASP is an active topic in research and has come a long way since its initial formulation in 1976 by Rice. Recent contributions incorporate ML models, for example, by predicting the performance of an algorithm on a given instance. In our approach, we used learning-to-rank methods from different ML research fields like information retrieval and recommender systems that best suit the needs of the algorithm selection task. Our computational results show that asking "the right questions" does matter: Asking for the (probably) best algorithm (or a ranking of algorithms) instead of asking for a performance prediction results in a better hit rate. Besides this finding, we can confirm that GBDTs perform better on the tabular data of our ASP than NNs, and we can report that the importance weighted MSE with weighting parameter α (iwMSE(α)) is competitive and worth to be studied in more detail in the future. Overall, the application of the ASP model GBDT-BIN-BCE to dynamically select an algorithm for solving an instance has outperformed the static selection of the best algorithm (BATCS-d-MLPP(PC_6, GS3)) (MRIW of 6.42% vs. 2.23%).

The second major contribution of this paper addressed what happens after solving the problem instance with a selected construction heuristic – improving the initial solution(s). To that, we presented the two GAs GA-J2P and GA-J2BRK with different representations (encodings). The random keys approach (GA-J2BRK) represents feasible solutions with every representation and,

therefore, does not need any costly repair mechanism during its computation like the GA-J2P approach. However, a great advantage of GA-J2P is that we can easily include knowledge about reasonable solutions. For example, we know the number of used batches (i.e., batch positions per machine) from the initial solutions and use it to limit the number of available batches. In consequence, the computational effort for processing a single generation is much lower and thus, more generations can be computed and the overall solution quality increases (compared to GA-J2BRK). As the experimental results have shown, the GA-J2P approach achieves remarkably higher improvements when compared to the other GAs. Comparing GA-J2P with the most efficient heuristic from literature BATCS-d-ML(PC_6, GS3)-LS(TS2), we see that the latter one is only competitive for very large instances with 1,600 and 3,200 jobs. In conclusion, we demonstrated in this paper how to integrate different (machine) learning related components in a scheduling method to get better solutions faster. We strongly believe that having a more holistic view on the solution method(s) will lead to very fruitful results for practical application.

Furthermore, we advocate shifting the goal of ASP from a mere, isolated selection of the best algorithm to decisions involving the allocation of computational resources, interleaving of algorithms, or use of information gathered online during the solving. Kotthoff, 2016 presented an interesting survey on algorithm selection and referred to the topic in a broader sense with the term algorithm portfolio. The algorithm portfolio technique outputs several (probably) good algorithms for a problem instance to achieve more robust results. Their online variants may even change the currently used algorithm if the solution quality misses the expectation to mitigate wrong decisions made at the beginning. One next challenge for our solution method(s) would be to include every part of our solution concept into the ASP. This could help us to use the potentials of the GA(s) even for the largest instances. Fundamental questions implied by that would be: how much time should we invest in finding the initial population compared to the GA? Or which GA-variant to choose, and which parameter configuration to select, could be included in the ASP. Taking that idea even further, information from a pre-solving phase of several GA variants could be used to further improve the solution quality. These ideas and questions will definitely need sophisticated ML models tailored to give the right answers. Researchers must, however, carefully consider how to incorporate the feedback from these ML models into their decision-making.

References

Adam, S. P., Alexandropoulos, S.-A. N., Pardalos, P. M., & Vrahatis, M. N. (2019). No Free Lunch Theorem: A Review. In I. C. Demetriou & P. M. Pardalos (Eds.), *Springer Optimization and Its Applications. Approximation and Optimization* (, 57–82), Cham: Springer International Publishing.

Balasubramanian, H., Mönch, L., Fowler, J. W., & Pfund, M. E. (2004). Genetic algorithm based scheduling of parallel batch machines with incompatible job families to minimize total weighted tardiness. *International Journal of Production Research*, *42*(*8*), *1621–1638.* doi:10.1080/00207540310001636994.

Burges, C., Ragno, R., & Le, Q. (2006). Learning to Rank with Nonsmooth Cost Functions. In B. Schölkopf, J. Platt, & T. Hoffman (Eds.), *Advances in Neural Information Processing Systems*, MIT Press.

Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., & Hullender, G. (2005). Learning to rank using gradient descent. In S. Dzeroski (Ed.), *Proceedings of the*

Journal Pre-proofs

22nd international conference on Machine learning - ICML '05, 89–96, New York, New York, USA: ACM Press.

Burges, C. J. C. (2010). From ranknet to lambdarank to lambdamart: An overview. *Learning*, *11*(*23-581)*, *81*.

Cao, Z., Qin, T., Liu, T.-Y., Tsai, M.-F., & Li, H. (2007). Learning to rank: from pairwise approach to listwise approach. In Z. Ghahramani (Ed.), *ICML 2007. Proceedings of the twenty-fourth International Conference on Machine Learning*, 129–136, New York: ACM.

Castillo, F., & Gazmuri, P. (2015). Genetic algorithms for batch sizing and production scheduling. *The International Journal of Advanced Manufacturing Technology*, *77*(*1-4)*, *261–280.* doi:10.1007/s00170-014-6456-5.

Chapelle, O., Le, Q., & Smola, A. (2007). Large margin optimization of ranking measures. In *NIPS workshop: Machine learning for Web search*.

Dauzère-Pérès, S., & Mönch, L. (2013). Scheduling jobs on a single batch processing machine with incompatible job families and weighted number of tardy jobs objective. *Computers & Operations Research*, *40*(*5)*, *1224–1233.* doi:10.1016/j.cor.2012.12.012.

Donmez, P., Svore, K. M., & Burges, C. J. (2009). On the local optimality of LambdaRank. In J. Allan, J. Aslam, M. Sanderson, C. Zhai, & J. Zobel (Eds.), *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, 460–467, New York, NY, USA: ACM.

Droste, S., Jansen, T., & Wegener, I. (2002). Optimization with randomized search heuristics—the (A)NFL theorem, realistic scenarios, and difficult functions. *Theoretical Computer Science*, *287*(*1)*, *131–144.* doi:10.1016/S0304-3975(02)00094-4.

Eiben, A. E., & Smith, J. E. (2015). *Introduction to evolutionary computing* (Second Edition). *Natural computing series*. Berlin, Heidelberg, New York, Dordrecht, London: Springer.

Feurer, M., & Hutter, F. (2019). Hyperparameter Optimization. In F. Hutter, L. Kotthoff, & J. Vanschoren (Eds.), *The Springer Series on Challenges in Machine Learning. Automated Machine Learning* (, 3–33), Cham: Springer International Publishing.

Freund, Y., Iyer, R., Schapire, R. E., & Singer, Y. (2003). An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research*, *4*(*Nov)*, *933–969*.

Gahm, C. (2022). *Extended instance sets for the parallel serial-batch scheduling problem with incompatible job families, sequence-dependent setup times, and arbitrary sizes* (V1). Mendeley Data. doi:10.17632/rxc695hj2k.1.

Gahm, C., Wahl, S., & Tuma, A. (2022). Scheduling parallel serial-batch processing machines with incompatible job families, sequence-dependent setup times and arbitrary sizes. *International Journal of Production Research*, *60*(*17)*, *5131–5154.* doi:10.1080/00207543.2021.1951446.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. *Adaptive computation and machine learning*. Cambridge, Massachusetts, London, England: The MIT Press.

Helo, P., Phuong, D., & Hao, Y. (2019). Cloud manufacturing – Scheduling as a service for sheet metal manufacturing. *Computers & Operations Research*, *110*, *208–219.* doi:10.1016/j.cor.2018.06.002.

Ho, Y. C., & Pepyne, D. L. (2002). Simple Explanation of the No-Free-Lunch Theorem and Its Implications. *Journal of Optimization Theory and Applications*, *115*(*3*), *549–570.* doi:10.1023/A:1021251113462.

Huang, J., Wang, L., & Jiang, Z. (2020). A method combining rules with genetic algorithm for minimizing makespan on a batch processing machine with preventive maintenance. *International Journal of Production Research*, *58*(*13*), *4086–4102.* doi:10.1080/00207543.2019.1641643.

Huynh, N.-T., & Chien, C.-F. (2018). A hybrid multi-subpopulation genetic algorithm for textile batch dyeing scheduling and an empirical study. *Computers & Industrial Engineering*, *125*, *615–627.* doi:10.1016/j.cie.2018.01.005.

Järvelin, K., & Kekäläinen, J. (2000). IR evaluation methods for retrieving highly relevant documents. In N. J. Belkin, P. Ingwersen, & M.-K. Leong (Eds.): *Vol. v. 34. SIGIR forum, SIGIR 2000. Proceedings of the 23rd annual international ACM SIGIR conference on research and development in information retrieval held in Athens, Greece, July 24-28, 2000*. Athens Greece, 41–48, New York: ACM Press.

Järvelin, K., & Kekäläinen, J. (2002). Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, *20*(*4*), *422–446.* doi:10.1145/582415.582418.

Jia, W., Jiang, Z., & Li, Y. (2013). Closed loop control-based real-time dispatching heuristic on parallel batch machines with incompatible job families and dynamic arrivals. *International Journal of Production Research*, *51*(*15*), *4570–4584.* doi:10.1080/00207543.2013.774505.

Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2011). Algorithm selection and scheduling. In : *CP'11, Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, 454–469, Berlin, Heidelberg: Springer-Verlag.

Kim, Y. J., Jang, J. W., Kim, D. S., & Kim, B. S. (2021). Batch loading and scheduling problem with processing time deterioration and rate-modifying activities. *International Journal of Production Research*, *1–21.* doi:10.1080/00207543.2020.1866783.

Koh, S.-G., Koo, P.-H., Ha, J.-W., & Lee, W.-S. (2004). Scheduling parallel batch processing machines with arbitrary job sizes and incompatible job families. *International Journal of Production Research*, *42*(*19*), *4091–4107.* doi:10.1080/00207540410001704041.

Koh, S.-G., Koo, P.-H., Kim, D.-C., & Hur, W.-S. (2005). Scheduling a single batch processing machine with arbitrary job sizes and incompatible job families. *International Journal of Production Economics*, *98*(*1*), *81–96.* doi:10.1016/j.ijpe.2004.10.001.

Kotthoff, L. (2016). Algorithm Selection for Combinatorial Search Problems: A Survey. In C. Bessiere, L. de Raedt, L. Kotthoff, S. Nijssen, B. O'Sullivan, & D. Pedreschi (Eds.), *LNCS sublibrary. SL 7, Artificial intelligence: Vol. 10101. Data Mining and Constraint Programming. Foundations of a Cross-Disciplinary Approach* (2016th ed., 149–190), Cham: Springer International Publishing; Imprint; Springer.

Journal Pre-proofs

Journal Pre-proofs

Malve, S., & Uzsoy, R. (2007). A genetic algorithm for minimizing maximum lateness on parallel identical batch processing machines with dynamic job arrivals and incompatible job families. *Computers & Operations Research*, *34*(*10*), *3016–3028.* doi:10.1016/j.cor.2005.11.011.

Michalewicz, Z. (1996). *Genetic algorithms + data structures: = evolution programs ; with 36 tables* (3., rev. and extended ed.). Berlin, Heidelberg: Springer.

Mönch, L., Balasubramanian, H., Fowler, J. W., & Pfund, M. E. (2005). Heuristic scheduling of jobs on parallel batch machines with incompatible job families and unequal ready times. *Computers & Operations Research*, *32*(*11*), *2731–2750.* doi:10.1016/j.cor.2004.04.001.

Mönch, L., Schabacker, R., Pabst, D., & Fowler, J. W. (2007). Genetic algorithm-based subproblem solution procedures for a modified shifting bottleneck heuristic for complex job shops. *European Journal of Operational Research*, *177*(*3*), *2100–2118.* doi:10.1016/j.ejor.2005.12.020.

Murphy, K. P. (2013). *Machine learning: A probabilistic perspective* (4th ed.). *Adaptive computation and machine learning series*. Cambridge, Mass.: MIT Press.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., … (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*, *2825–2830.*

Pinedo, M. L. (2016). *Scheduling: Theory, Algorithms, and Systems* (Fifth Edition). Cham: Springer International Publishing.

Qin, T., Liu, T.-Y., & Li, H. (2010). A general approximation framework for direct optimization of information retrieval measures. *Information Retrieval*, *13*(*4*), *375–397.* doi:10.1007/s10791-009-9124-x.

Rice, J. R. (1976). The Algorithm Selection Problem. In *Advances in Computers. Advances in Computers Volume 15* (, 65–118), Elsevier.

Shwartz-Ziv, R., & Armon, A. (2022). Tabular data: Deep learning is not all you need. *Information Fusion*, *81*, *84–90.* doi:10.1016/j.inffus.2021.11.011.

Smith-Miles, K. A. (2009). Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys*, *41*(*1*), *1–25.* doi:10.1145/1456650.1456656.

Soares, C., Brazdil, P. B., & Kuba, P. (2004). A Meta-Learning Method to Select the Kernel Width in Support Vector Regression. *Machine Learning*, *54*(*3*), *195–209.* doi:10.1023/B:MACH.0000015879.28004.9b.

Streeter, M. J., Golovin, D., & Smith, S. F. (2007). Combining Multiple Heuristics Online. In *AAAI Conference on Artificial Intelligence*.

Taylor, M., Guiver, J., Robertson, S., & Minka, T. (2008). SoftRank. In M. Najork, A. Broder, & S. Chakrabarti (Eds.), *Proceedings of the international conference on Web search and web data mining - WSDM '08*,p. 77–77, New York, New York, USA: ACM Press.

Uzunoglu, A., Gahm, C., & Tuma, A. (2023a). A machine learning enhanced multi-start heuristic to efficiently solve a serial-batch scheduling problem. *Annals of Operations Research.* doi:10.1007/s10479-023-05541-w.

Uzunoglu, A., Gahm, C., Wahl, S., & Tuma, A. (2023b). Learning-augmented heuristics for scheduling parallel serial-batch processing machines. *Computers & Operations Research*, *151*, *106122.* doi:10.1016/j.cor.2022.106122.

Valente, J. M., & Schaller, J. E. (2012). Dispatching heuristics for the single machine weighted quadratic tardiness scheduling problem. *Computers & Operations Research*, *39*(*9*), *2223–2231.* doi:10.1016/j.cor.2011.11.005.

Wahl, S., Gahm, C., & Tuma, A. (2023). *Knowledge base for batch-processing machine scheduling research*, Mendeley Data (V3). doi:10.17632/7cv58py5hk.3.

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*(*1*), *67–82.* doi:10.1109/4235.585893.

Wu, K., Huang, E., Wang, M., & Zheng, M. (2022). Job scheduling of diffusion furnaces in semiconductor fabrication facilities. *European Journal of Operational Research*, *301*(*1*), *141–152.* doi:10.1016/j.ejor.2021.09.044.

Xu, J., Liu, T.-Y., Lu, M., Li, H., & Ma, W.-Y. (2008). Directly optimizing evaluation measures in learning to rank. In T.-S. Chua, M.-K. Leong, S. H. Myaeng, D. W. Oard, & F. Sebastiani (Eds.), *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, 107–114, New York, NY, USA: ACM.

Appendix

A-1. Preliminary study of "d-MLRP" variants

Table 12: Performance of "d-MLRP" variants by pipeline configuration and ranking application strategy

Pipeline configuration	Ranking application strategy						Mean
	B1	*B1-G*	*Bx*	*B(3)-G*	*B(5)-G*	*B(9)-G*	
[1,9,0]-CF	34,42	76,41	76,75	76,86	76,86	76,90	69,70
[1,9,0]-AF	34,68	76,24	76,44	76,42	76,44	76,54	69,46
[0,5,5]-CF	51,82	77,38	77,55	77,59	77,64	77,46	73,24
[0,5,5]-AF	54,80	77,43	77,55	77,53	77,55	77,56	73,74
[1,4,5]-CF	56,80	77,55	77,62	77,53	**77,63**	**77,55**	**74,12**
[1,4,5]-AF	53,23	77,39	77,73	77,56	77,58	77,66	73,52

[0,2,8]-CF	52,56	77,02	77,46	77,38	77,44	77,53	73,23
[0,2,8]-AF	55,07	77,43	77,52	77,49	77,47	77,51	73,75
[1,2,7]-CF	56,05	77,15	77,33	77,30	77,38	77,34	73,76
[1,2,7]-AF	57,99	77,40	77,74	77,72	**77,69**	**77,73**	**74,38**
[0,1,9]-CF	46,56	76,95	76,90	77,14	77,11	76,99	71,94
[0,1 9]-AF	51,68	77,07	77,12	77,08	77,03	77,13	72,85
[1,0,9]-CF	49,32	77,06	77,00	77,06	77,02	76,99	72,41
[1,0,9]-AF	52,50	77,24	77,43	77,34	77,52	77,42	73,24
[0,0,10]-CF	47,44	76,74	75,97	76,74	76,72	76,52	71,69
[0,0,10]-AF	47,81	77,07	76,84	77,26	77,29	77,32	72,26
	52.87	67.94	68.75	68.55	**68.71**	**68.76**	

A-2. Best-performing hyper-parameters for tuning phase 1

Table 13 Best-performing hyper-parameter for NN tuning phase 1

Labeling strategy	Loss function	Layer 1	Layer 2	Layer 3	Dropout rate	Reg. rate
BIN	BCE	1024	1024	1024	0.1	0.010
OV	MSE	256	1024	256	0.3	0.001
OV-N	MSE	1024	1024	512	0.1	0.010
RANK	MSE	1024	1024	512	0.1	0.001
	iwMSE(1)	1024	1024	512	0.1	0.001
	iwMSE(3)	1024	512	512	0.1	0.002
	iwMSE(3)	1024	1024	256	0.1	0.002
	ApproxNDCG	1024	1024	256	0.1	0.002
	LambdaRankNDCG	512	1024	1024	0.1	0.002
	LambdaRankNDCG(1)	1024	256	256	0.1	0.002
RANK-SC	MSE	1024	1024	1024	0.1	0.001
	iwMSE(1)	1024	1024	1024	0.1	0.002
	iwMSE(2)	512	512	1024	0.1	0.010
	iwMSE(3)	1024	512	512	0.1	0.001
	ApproxNDCG	1024	256	512	0.1	0.002

Table 14 Best-performing hyper-parameter for GBDT tuning phase 1

Labeling strategy	Loss function	Number of estimators	Learning rate	Maximum depth	Min. child weight	Subsampling rate	Column sample
BIN	BCE	450	0.01	9	1	0.7	0.5
BIN	LambdaBinary	350	0.10	8	2	0.7	0.5

OV	MSE	450	0.01	10	2	0.7	0.7
OV-N	MSE	450	0.01	10	1	0.7	0.7
RANK	MSE	350	0.01	10	1	0.7	0.7
	iwMSE(1)	450	0.01	10	2	0.7	0.7
	iwMSE(2)	450	0.01	10	2	0.7	0.7
	iwMSE(3)	350	0.01	10	2	0.7	0.7
	RankPairwise	450	0.10	9	2	0.7	0.5
	LambdaMARTNDCG	250	0.10	9	1	0.7	0.5
RANK-SC	MSE	350	0.01	10	1	0.7	0.7
	iwMSE(1)	450	0.01	10	1	0.7	0.5
	iwMSE(2)	450	0.01	9	2	0.7	0.5
	iwMSE(3)	450	0.01	10	1	0.7	0.5

A-3. Best-performing hyper-parameters for tuning phase 2

Table 15 Best performing hyper-parameters for NN phase 2

Labeling strategy	Loss function	Layer 1	Layer 2	Layer 3	Dropout rate	Reg. rate
BIN	BCE	2048	256	2048	0.1	0.001
RANK	iwMSE(2)	2048	2048	2048	0.1	0.002

Table 16 Best-performing hyperparameters for GBDT phase 2

Labeling strategy	Loss function	Number of estimators	Learning rate	Maximum depth	Min. child weight	Subsampling rate	Column sample
BIN	BCE	550	0.01	10	2	0.7	0.5
RANK	iwMSE(2)	550	0.01	10	2	0.7	0.5

Highlights

Journal Pre-proofs

- A serial-batch scheduling problem is the application case of our analysis.
- “Learning-to-rank” ML models select the best-performing heuristic.
- Two Genetic Algorithms are introduced with different encoding schemes.
- Both Genetic Algorithms can use solutions from the heuristics as initial populations.
- Our method substantially improves the solution quality.

Journal Pre-proofs