Iterative beam search algorithms for the permutation flowshop

Vincent Jost, Luc Libralesso, Pablo Andres Focke, Aurélien Secardin

Version of Record: https://www.sciencedirect.com/science/article/pii/S0377221721008602
Manuscript_7b30f1bb6fbb37b3a5385607648ea398

Iterative beam search algorithms for the permutation flowshop

Luc Libralesso[a,b,*], Pablo Andres Focke[a], Aurélien Secardin[a] and Vincent Jost[a]

[a] Univ. Grenoble Alpes, CNRS, Grenoble INP, G-SCOP, 38000 Grenoble, France
[b] University Clermont Auvergne, LIMOS, CNRS UMR 6158, Aubière France

ARTICLE INFO

Keywords:
Heuristics
Iterative beam search
Permutation flowshop
Makespan
Flowtime

ABSTRACT

We study an iterative beam search algorithm for the permutation flowshop (makespan and flowtime minimization). This algorithm combines branching strategies inspired by recent branch-and-bounds and a guidance strategy inspired by the LR heuristic. It obtains competitive results on large instances compared to the state-of-the-art algorithms, reports many new-best-so-far solutions on the VFR benchmark (makespan minimization) and the Taillard benchmark (flowtime minimization) without using any NEH-based branching or iterative-greedy strategy.
The source code is available at: `https://github.com/librallu/dogs-pfsp`.

1. Introduction

In the flowshop problem, one has to schedule jobs, where each job has to follow the same route of machines. The goal is to find a job order that minimizes some criteria. The Permutation FlowShop Problem (PFSP) is a common (and fundamental) variant that imposes the machines to process jobs in the same order (thus, a permutation of jobs is enough to describe a solution). The permutation flowshop has been one of the most studied problems in the literature [36, 32] and has been considered on various industrial applications [17, 43]. We may also note that the permutation flowshop is at the origin of multiple other variants, for instance, the blocking permutation flowshop [46], the multiobjective permutation flowshop [21], the distributed permutation flowshop [12], the no-idle permutation flowshop [33], the permutation flowshop with buffers [29] and many others. Regarding the criteria to minimize, in this paper, we study two of the most studied objectives: the makespan (minimizing the completion time of the last job on the last machine) and the flowtime (equivalent to minimizing the sum of completion times of each job on the last machine if we do not consider the releasing date of the jobs). According to the scheduling notation introduced by Graham, Lawler, Lenstra, and Rinnooy Kan [14], the makespan criterion is denoted $F_m|prmu|Cmax$ and the flowtime criterion $F_m|prmu|\sum C_i$.

Consider the following example instance with $m = 3$ machines with $n = 4$ jobs (j_1, j_2, j_3, j_4) with the job processing time matrix P defined as follows where $P_{j,m}$ indicates the processing time of job j on machine m:

$$P = \begin{pmatrix} 3 & 2 & 1 & 3 \\ 3 & 4 & 3 & 1 \\ 2 & 1 & 3 & 2 \end{pmatrix}$$

One possible solution can be described in Figure 1. This solution has a makespan (completion time of the last job on the last machine) of 18 and a flowtime (sum of completion times on the last machine) of $8 + 11 + 16 + 18 = 53$.

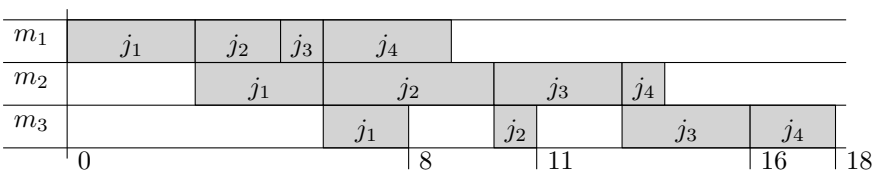


Figure 1: A solution for the example instance with a job order $\sigma = j_1, j_2, j_3, j_4$

luc.libralesso@uca.fr (L. Libralesso); pablofocke@gmail.com (P.A. Focke); aurelien.secardin@icloud.com (A. Secardin); vincent.jost@grenoble-inp.fr (V. Jost)
ORCID(s):

Regarding resolution methods, the makespan minimization permutation flowshop problem has been massively studied over the last 50 years and numerous numerical methods have been applied.

In 1983, Nawaz, Enscore, Ham proposed an insertion-based heuristic (later called NEH) [28]. This heuristic sorts jobs by some criterion (usually by a non-decreasing sum of processing times), then, it adds them one by one at the position that minimizes the objective function. The NEH, obtained, at the time, excellent results compared to other heuristics and can be used to perform greedy algorithms and perturbation-based algorithms as well. It has been largely considered as an essential component producing excellent solutions for large-scale permutation flowshop instances, and multiple methods have been built using it. One of the most famous ones is Taillard's acceleration [40]. It reduces the cost of inserting a job at all possible positions from $O(n^2.k)$ to $O(n.k)$. Considering these results, multiple works aim to improve the NEH heuristic [11, 27, 15, 4, 37, 45, 26] to quote a few.

The (meta-)heuristics state-of-the-art methods for the makespan minimization usually perform an iterated-greedy algorithm [39, 8]. Such algorithms start with a NEH heuristic to build an initial solution. Then, destroy a part of it and reconstruct it using again a NEH heuristic. To the best of our knowledge, the current state-of-the-art algorithms for the makespan minimization criterion are: the variable block insertion heuristic [16], the best-of-breed Iterated-Greedy [8], and, an automatically designed algorithm using the EMILI framework [30]. We may note that other algorithms exist to solve the makespan minimization. To quote a few, we can find some hybrid algorithms [47] (a combination of the NEH heuristic as a part of the initial population, a genetic algorithm, and simulated annealing to replace the mutation), memetic algorithms [18], an automatically designed local-search scheme [30].

The (meta-)heuristics methods for the flowtime minimization also involve the NEH heuristic, but some other constructive methods as well. For instance, the Liu and Reeve's method (LR) [25] performs a forward search (*i.e.* appending jobs at the end of the partial schedule). It was later improved to reduce its complexity from $O(n^3m)$ to $O(n^2m)$, later called the FF algorithm [6]. Later, this scheme was integrated into a beam search algorithm (more on that later) that obtained state-of-the-art performance [7]. Recently, this beam search was integrated within a biased random-key genetic algorithm as a warm-start procedure [1]. In parallel, the authors of the EMILI framework also proposed an efficient algorithm for the flowtime minimization. These are, to the best of our knowledge, the state-of-the-art methods for the flowtime minimization alongside the algorithms proposed in [31].

Regarding exact-methods, a recent branch-and-bound [13] brought light on a bi-directional branching (*i.e.* constructing the candidate solution from the beginning and the end at the same time) combined with a simple yet efficient bounding scheme to solve the makespan minimization criterion. The resulting branch-and-bound obtained excellent performance and was even able to solve to optimality almost all large VFR instances with 20 machines.

Moreover, recently, an iterative beam search has been proposed and, successfully applied to various combinatorial optimization problems as guillotine 2D packing problems [23, 10], the sequential ordering problem [22] and the longest common subsequence problem [24]. This iterative beam search scheme, at the beginning of the search, behaves as a greedy algorithm and then, more and more, as a branch-and-bound algorithm as time goes (it performs a series of beam search iterations with a geometric growth). It naturally combines search-space reductions from branch-and-bounds and guidance strategies from classical (meta-)heuristics. Considering the success of recent branch-and-bound branching schemes and the performance of greedy-like algorithms to solve the permutation flowshop, it would be a natural idea to combine them. However, to the best of our knowledge, it has not been studied before. This paper aims to fill this gap. For the makespan criterion, we implemented a bi-directional branching scheme and combined it with a variant of the LR [25] guidance strategy and use an iterative beam-search algorithm to perform the search. We report competitive results compared to the state-of-the-art algorithms and find new best-known solutions on many large VFR instances (we improve the best-known solution for almost all instances with 400 jobs or more and 40 machines or more orders of magnitude faster than previous works). Note that these results are interesting and new, as almost all the efficient algorithms in the literature are based on the NEH heuristic or the iterated greedy algorithm. This is not the case for our algorithm, as it is based on a variant of the LR heuristic and an exact-method branching scheme (bi-directional branching).

Regarding the flowtime criterion, the bi-directional branching cannot be directly applied (the bounding procedure is less efficient than for the makespan criterion). However, we show that an iterative beam search with a simple forward search (modified LR algorithm) is efficient, outperforms the current state-of-the-art algorithms, and, reports new best-solutions for the Taillard's benchmark orders of magnitude faster than previous works (almost all solutions for instances with 100 jobs or more were improved).

This paper is structured as follows: Section 2 presents the iterative beam search strategy. Section 3 presents the

branching schemes we implement (the forward and bi-directional search). Section 4 present the guides we implement (the bound guide, the idle-time guide, and mixes between these two first guides) and Section 5 presents the results obtained by running all variants described in this paper, showing that an iterative beam search combined with a simple variant of the LR heuristic can outperform the state-of-the-art.

2. The search strategy: Iterative beam search

Beam Search is a tree search algorithm that uses a parameter called the beam size (D). Beam Search behaves like a truncated *Breadth First Search (BrFS)*. It only considers the best D nodes on a given level. The other nodes are discarded. Usually, we use the bound of a node to choose the most promising nodes. It generalizes both a greedy algorithm (if $D = 1$) and a BrFS (if $D = \infty$). Figure 2 presents an example of beam search execution with a beam width $D = 3$.

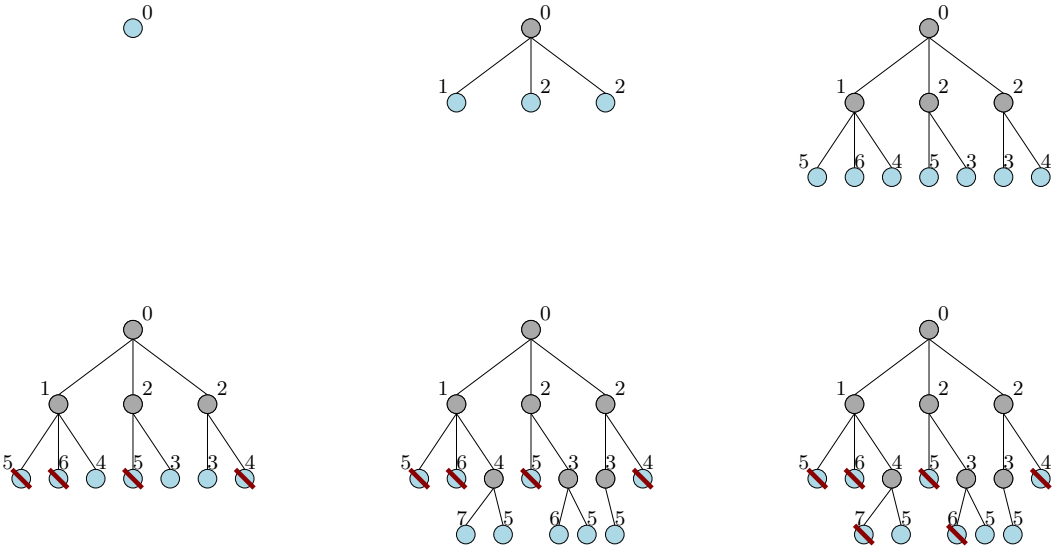


Figure 2: Beam Search Iterations with a beam width $D = 3$

Beam Search was originally proposed in [35] and used in speech recognition. It is an incomplete (*i.e.* performing a partial tree exploration and can miss optimal solutions) tree search parametrized by the beam width D. Thus, it is not an anytime algorithm. The parameter D allows controlling the quality of the solutions and the execution time. The larger D is, the longer it will take to reach feasible solutions, and the better these solutions will be.

Recently, a variant of beam search, called iterative beam search, was proposed and obtained state-of-the-art results on various combinatorial optimization problems [22, 23, 24, 10]. Iterative beam search performs a series of restarting beam search with geometrically increasing beam size until the time limit is reached. Algorithm 2.1 shows the pseudo-code of an iterative beam search. The algorithm runs multiple beam-searches starting with $D = 1$ (line 1) and increases the beam size (line 8) geometrically. Each run explores the tree with the given parameter D. In the pseudo-code, we increase geometrically the beam size by 2. This parameter can be tuned, however, we did not notice a significant variation in the performance while adjusting this parameter. This parameter (that can be a real number) should be strictly larger than 1 (for the beam to expand) and should not be too large, say less than 3 or 5 (otherwise, the beam grows too fast and when the time limit is reached, most of the computational time was possibly wasted in the last incomplete beam, without providing any solution).

Notes about the implementation: We use two mechanisms to limit the memory usage and speed-up the iterative beam search:

Algorithm 2.1: Iterative Beam Search algorithm

Input: root node

```
1 D ← 1
2 while stopping criterion not met do
3 |   Candidates ← {root}
4 |   while Candidates ≠ ∅ do
5 |   |   nextLevel ← ⋃_{n∈Candidates} children(n)
6 |   |   Candidates ← best D nodes among nextLevel
7 |   end
8 |   D ← D× 2
9 end
```

1. A naive way to store the partial job order in each node would be to use an array within each node. However, this can be costly in memory on larger runs. We propose instead to store a decision tree that stores each decision taken at each iteration. All nodes contain a pointer towards a node of this tree. One can obtain a solution by backtracking from a node to the root of this decision tree. This mechanism limit the number of copies performed within the iterative beam search.
2. When a level of the beam search is expanded, it generates in average $n.D$ nodes but only D will be expanded. We propose a "lazy node compute" that only computes the bounds and guides for a given node, and only computes the remaining parts (*i.e.* the fronts) if the node is expanded. This mechanism allows reducing even further the memory requirements and improve the running times.

The Section 3 presents the branching schemes used to generate children (Algorithm 2.1, line 5) and the Section 4 presents ways to identify the best nodes (Algorithm 2.1, line 6).

3. Branching schemes

We present in this section the two branching schemes we use (*i.e.* the search tree structure): the forward search (*i.e.* constructing the solution from the beginning) and the bi-directional search (*i.e.* constructing the solution from the beginning and the end).

3.1. Forward branching

The forward branching assigns jobs at the first free position in the partial sequences (it constructs the solutions from the beginning). The root corresponds to a situation where the candidate solution contains no job (*i.e.* c.STARTING $= \emptyset$). Each of the search-tree nodes correspond to the first jobs in the resulting solution. Children of a given node correspond to a possible insertion of each job that is not scheduled yet at the end of the schedule. Each node stores information about the partial candidate solution (jobs already added), the release time of each machine, and the partial makespan (resp. flowtime). A candidate solution (or node) c is considered as "goal" or "feasible" if all jobs are inserted (*i.e.* c.STARTING $= J$) and contains the following information:

- STARTING: vector of jobs inserted that lead to the candidate c (first jobs of the sequence we want to generate).
- FRONTSTARTING: vector of times when machines are first available after appending STARTING jobs.

Before presenting the forward children-generation, we present how to insert a job $j \in J$ in a candidate solution c (Algorithm 3.1). This insertion can be done in $O(m)$ where m is the number of machines.

Algorithm 3.2 presents the forward branching pseudo-code (how to generate all children of a candidate solution c).

3.2. Bi-directional branching

To the best of our knowledge, bi-directional branching was first introduced in 1980 [34]. The bi-directional search appends jobs at the beginning and the end of the candidate solution. It aims to exploit the property of the inverse problem (job order inverted and machine order inverted). Since then, the efficiency of this scheme has been largely

Algorithm 3.1: Forward search: insertion of job j in candidate solution c (INSERTFORWARD(c, j))

Input: candidate solution (or node) c
Input: job to be inserted $j \in J$

```
1  c.FRONTSTARTING_1 ← c.FRONTSTARTING_1 + P_{j,1}
2  for i ∈ {2, ... m} do
3      if c.FRONTSTARTING_{i-1} > c.FRONTSTARTING_i then
           /* there is some idle time on machine i */
4          idle ← c.FRONTSTARTING_{i-1} − c.FRONTSTARTING_i
5          c.FRONTSTARTING_i ← c.FRONTSTARTING_{i-1} + P_{j,i}
6      else
           /* no idle time on machine i */
7          c.FRONTSTARTING_i ← c.FRONTSTARTING_i + P_{j,i}
8      end
9  end
10 c.STARTING ← c.STARTING ∪ {j}
```

Algorithm 3.2: Forward search children generation from a candidate solution c (CHILDREN(c))

Input: candidate solution (or node) c

```
1  children ← ∅
2  for j ∈ unscheduled jobs do
3      children ← children ∪ INSERTFORWARD(Copy(c), j)
4  end
5  return children
```

recognized to solve the makespan minimization optimally [2, 19, 20, 5, 3, 38]. Recently, a parallel branch-and-bound was successfully used to solve the makespan minimization criterion [13] using this bi-directional scheme. Multiple ways to decide if the algorithm performs a forward or backward insertion were studied (for instance, alternating between a forward insertion and backward insertion). This study found out that the best way is selecting the insertion type that has the less remaining children after the bounding pruning step. Ties are broken by selecting the type of insertion that maximizes the sum of the lower bounds, as large lower bounds are usually a more precise estimation.

A candidate solution (or node) c is considered as "goal" or "feasible" if all jobs are inserted (*i.e.* c.STARTING $\cup$ c.FINISHING $= J$) and contains the following information:

- STARTING: vector of jobs inserted at the beginning of the partial permutation that lead to the candidate c (first jobs of the sequence we want to generate).
- FRONTSTARTING: vector of times when machines are first available after appending STARTING jobs.
- FINISHING: (inverted) vector of jobs inserted at the end of the partial permutation that lead to the candidate c (last jobs of the sequence we want to generate).
- FRONTFINISHING: vector of times when machines are no more available after appending STARTING jobs.

Algorithm 3.3 presents the bi-directional branching pseudo-code. We use INSERTFORWARD (Algorithm 3.1) to insert a job within the STARTING vector and INSERTBACKWARD that inserts a job within the FINISHING vector. This procedure is almost similar to INSERTFORWARD but iterates over machines in an inverted order ($m \rightarrow 2$ instead of $2 \rightarrow m$). It generates children of both the forward and backward search (lines 1-6), prunes nodes that are dominated by the best-known solution (or upper-bound, lines 7-8). Then, it chooses the scheme that has fewer children (thus, usually a smaller search-space) and breaks ties by selecting the scheme having the more precise lower bounds (sum of lower bounds).

Algorithm 3.3: Bi-directional search children generation from a candidate solution c (CHILDREN(c))

Input: candidate solution (or node) c

```
F ← ∅                                          /* F correspond to the children obtained by forward search */
B ← ∅                                          /* B correspond to the children obtained by backward search */
for j ∈ unscheduled jobs do
    F ← F ∪ INSERTFORWARD(Copy(c), j)
    B ← B ∪ INSERTBACKWARD(Copy(c), j)
end
F ← {c|c ∈ F if BOUND(c) < best known solution}      /* removing forward nodes dominated by the UB */
B ← {c|c ∈ B if BOUND(c) < best known solution}      /* removing backward nodes dominated by the UB */
if |F| < |B| ∨ (|F| = |B| ∧ ∑_{c∈F} BOUND(c) > ∑_{c∈B} BOUND(c)) then
    return F /* choosing the forward search */
else
    return B /* choosing the backward search */
end
```

4. Guides

In the previous section, we discussed the branching rules that define a search tree. As such trees are usually large, a way to tell which node is apriori more desirable is needed. In branch-and-bounds, this mechanism is called "bound" and also constitutes an optimistic estimate of the best solution that can be achieved in a given sub-tree. In constructive meta-heuristics, the guidance strategy is usually not an optimistic estimate, which often allows finding better solutions (for instance the LR [25] greedy guidance strategy). In this section, we present several guidance strategies for both the makespan and flowtime criteria.

4.1. Bound

We define the bound guidance strategy for the forward search and makespan minimization as follows. It measures the first time the last machine (machine m) is available and assumes that each remaining job can be scheduled without any idle time.

$$\mathrm{F}g_{\text{bound}} = Cmax_{f,m} + R_m$$

The bound guidance strategy for the bi-directional search and makespan minimization is defined as follows. It generalizes the bound for the forward search by also taking into account the backward front. We may note that the bi-directional branching allows computing a better bound as all machines are relevant for this bound (compared to the forward branching bound in which only the last machine is used to compute a bound).

$$\mathrm{FB}g_{\text{bound}} = \max_{i \in M}(Cmax_{f,i} + R_i + Cmax_{b,i})$$

The flowtime bound is defined as the sum of end times for each job scheduled in the forward search. Each time a job is added to the candidate solution, the flowtime value is modified.

4.2. Idle time

The bound guide is an effective guidance strategy, but is known to be imprecise at the beginning of the search (*i.e.* the first levels of the search tree). Another guide that is usually considered as a part of effective greedy strategies (for instance, the LR heuristic) is to use the idle time of the partial solution. Usually, a solution with a small idle time reaches good performance on both the makespan or flowtime criteria.

The idle time can be defined as follows:

$$\mathrm{FB}g_{\text{idle}} = \sum_{i \in M} I_{f,i} + I_{b,i}$$

4.3. Bound and idle time

As it is noted in many works [25, 7], another interesting guidance strategy is to combine both guidance strategies discussed earlier (*i.e.* the bound and idle time guides). Indeed, while the bound guide is usually ineffective to guide the search close to the root, it is very precise close to feasible solutions. Inversely, the idle time is an efficient guide close to the root but relatively inefficient close to feasible solutions. We study the *bound and idle time guide* that linearly reduces the contribution of the idle time to favor the bound, depending on the completion level of the candidate solution.

The bound and idle time guide can be defined as follows, where C is a value used to make the idle time and bound comparable ($C = \frac{\alpha|J|}{m}$):

$$g_{\text{alpha}} = \alpha \,.\, g_{\text{bound}} + (1-\alpha) \,.\, C \,.\, g_{\text{idle}}$$

α corresponds to the proportion of jobs added (*i.e.* 0 if no jobs are added, 1 if all jobs are added). It is defined as follows: $\alpha = \frac{|F|+|B|}{|J|}$ for the bi-directional branching or $\alpha = \frac{|F|}{|J|}$ for the forward branching.

4.4. Bound and weighted idle time

Another useful remark found in greedy algorithms for the permutation flowshop problem [25] is to add additional weight to the idle time produced by the first machines at the beginning of the search (as it will have a greater impact on the objective function than the others). However, the LR heuristic cannot be directly applied in a general tree search context. Indeed, it is sometimes noted [7] that algorithms like the beam search usually compare nodes from different parents, thus, it is needed to adapt the LR heuristic guidance that only compares nodes with the same parent. We propose a simple yet efficient ways to implement similar ideas. The search is guided by a combination of a weighted idle time and by the bounding procedure.

We present a new weighted idle time guidance strategy that considers the sum of idle time percentage divided by the position of each front. Doing this, it allows making idle time on the first machines more important to the forward search and the idle time on the last machines more important to the backward search. The bound and weighted idle time guide for the bi-directional search is defined as follows:

$$g_{\text{walpha}} = \alpha.g_{\text{bound}} + (1-\alpha).\left(\sum_{i\in M} \frac{I_{f,i}}{Cmax_{f,i}} + \frac{I_{b,i}}{Cmax_{b,i}}\right).g_{\text{bound}}$$

Notice that, during the bi-directional search, if only one direction is used (all jobs are inserted in the forward part (resp. backward part)), g_{walpha} is not defined. We choose to consider that $g_{\text{walpha}} = \infty$ in this case. Indeed, using both fronts allows better bounds and guides, thus nodes using only one front should be not chosen over nodes that use both.

4.5. Bound and gap

While solving some instances using a bi-directional branch-and-bound, we may notice that sometimes, the bound is very tight (thus is also a good guide). We propose a new guide that uses the gap between the best solution found, and the node bound ($\frac{UB-LB}{UB}$). If the gap is small (close to 0) the bound will be used more as a guide. If the gap is large, the idle time will be more considered. The "gap" guide is defined as follows:

$$g_{\text{gap}} = \frac{UB}{UB-LB}.g_{\text{bound}} + \frac{UB-LB}{UB}.\left(\sum_{i\in M} \frac{I_{f,i}}{Cmax_{f,i}} + \frac{I_{b,i}}{Cmax_{b,i}}\right)$$

Similarly to g_{walpha}, $g_{\text{gap}} = \infty$ if only one direction has been taken by the node.

5. Numerical results

In this section, we perform various experiments to evaluate the efficiency of the algorithms discussed in the previous sections. In Subsection 5.1, we present numerical results obtained in the makespan minimization version and Subsection 5.2, the results obtained in the flowtime minimization version. All algorithms have been implemented in

rust (IGbob is available online [8]) and executed on an Intel Xeon Gold 5118 @ 2.30 GHz with 32 GB RAM. As the CPU has multiple physical cores, we ran 20 tests in parallel. For both objectives, we study the ARPD (Average Relative Percentage Deviation), defined as follows:

$$ARPD_{Ia} = \sum_{i \in I} \frac{M_{ai} - M_i^*}{M_i^*} \cdot \frac{100}{|I|}$$

where I is a set of instances with similar characteristics, M_{ai} corresponds to the objective obtained by algorithm a on instance i. And M_i^* the reference solution objective for the instance i. The ARPD describes the performance of a given algorithm on a given instance type. For the makespan minimization (Taillard benchmark), we used the best upper-bounds provided on Taillard's website[1]. For the makespan minimization (VFR benchmark, we used the best-results provided by [30][2]). For the flowtime minimization, we used the best solutions reported in [31].

Some algorithms in the literature do not provide their implementation but provide their ARPDs. To make a fair comparison, we regularize the CPU times using the PassMark database [3]. This database allows to compare the relative speed of CPU and provide an estimation of the running time of the literature algorithms on our CPU using the CPU frequency, the caches, *etc*. As all algorithms we compare with are single-threaded, we use the single-thread performance. More precisely, we used the following conversions:

- For the results presented for IGirms, IGall, ALGirtct and IGA [30], the authors used an AMD Opteron 6272 @ 2.1 GHz. Our CPU is estimated to run 2.4 times faster, thus we artificially slow down our CPU by this amount while comparing with these algorithms. For instance, for a 1000 seconds run of either IGirms, IGall, ALGirtct or IGA, we compare them to an iterative beam search run with time limit set to $\frac{1000}{2.4} = 416$ seconds.
- For the results presented for MRSILS and BSCH [7], the authors used an Intel Core i7-3770 @ 3.40GHz. Their CPU is estimated to run 1.17 times faster than our CPU, thus, we artificially speed up our CPU by a factor of 1.17.
- For the results presented for VBIH [16], the authors used an Intel Core i5 @ 3.40GHz. Their CPU is estimated to run 1.16 times faster than our CPU, thus, we artificially speed up our CPU by a factor of 1.16.

For each instance and each criterion, we ran our algorithms for $n.m.45$ milliseconds, where n is the number of jobs and m the number of machines as it is usually done in the literature. We evaluate our algorithms on the famous Taillard benchmark [41] (makespan and flowtime minimization) and on the famous VFR benchmark [44]. The first consists of sets of 10 instances with a job number $n \in \{20, 50, 100, 200, 500\}$ and machine number $m \in \{5, 10, 20\}$. The latter consists of sets of 10 instances with a job number $n \in \{100, 200 \dots 800\}$, a machine number $m \in \{20, 40, 60\}$. For each variant, we compare our algorithms with state-of-the-art algorithms.

5.1. Makespan minimization

5.1.1. Iterative beam search performance comparison

In Sections 3,4, we presented multiple variants of the Iterative beam search (forward and bi-directional search, 5 different guides). Figure 3 presents a performance comparison of the different iterative beam search algorithms we proposed.

Discussions: Regarding the forward branching procedures, we observe a significant improvement by including the idle time in the guide and obtain the best results by including a weighted idle time within the guide (similarly to the principles presented in the LR heuristic [25]). Indeed, ARPD ranges from 17% to 25% for the bound guide, and goes down between 1% to 5% for the idle and gap guides on the VFR instances. We note that the *walpha* and *gap* guides do not contribute much to the algorithm performance, and, surprisingly, simple guides (*idle, alpha*) perform better.

Regarding the bi-directional branching procedures, we observe that the bound guide performs well in most cases, from 0.16% to 8% ARPD. This can be explained as the bound gets tighter when the number of machines is low. Using the idle time in the guide (idle time only or idle time combined with the bound) decreases the performance of the algorithm (performances ranging from 2% to 17%). It seems to indicate that the idle time is a less efficient guide than

[1] http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/flowshop.dir/best_lb_up.txt
[2] http://iridia.ulb.ac.be/supp/IridiaSupp2018-002/
[3] https://www.cpubenchmark.net/

	Forward search					bi-directional search				
instance sets	bound	idle	alpha	walpha	gap	bound	idle	alpha	walpha	gap
TAI20_5	1.92	1.45	1.31	1.13	1.53	0.0	0.0	0.0	0.0	0.0
TAI20_10	2.12	0.52	0.6	0.1	2.08	0.0	0.27	0.27	0.0	0.0
TAI20_20	3.47	0.8	0.85	0.93	3.79	0.67	2.04	2.02	0.28	0.64
TAI50_5	0.54	0.86	0.97	2.0	1.47	0.0	0.0	0.0	0.06	0.0
TAI50_10	5.6	3.4	3.35	2.61	4.38	0.09	4.42	3.49	0.64	0.09
TAI50_20	9.79	2.32	2.43	2.25	9.59	2.54	7.14	7.36	0.45	2.5
TAI100_5	0.36	0.26	0.29	0.99	1.02	0.0	0.09	0.1	0.08	0.0
TAI100_10	2.79	1.32	1.38	1.55	1.9	0.0	2.42	1.97	0.86	0.0
TAI100_20	11.68	3.29	2.93	2.29	10.01	1.97	8.65	8.62	1.52	1.14
TAI200_10	1.88	1.07	1.13	1.66	1.22	0.0	1.42	1.38	1.74	0.87
TAI200_20	9.62	2.29	2.53	2.04	3.58	1.13	7.79	7.34	1.81	0.59
TAI500_20	3.97	1.6	1.62	1.69	1.87	0.46	4.56	4.58	1.43	**0.06**
VFR100_20	12.02	3.07	2.94	1.23	11.98	1.79	10.41	9.66	0.43	1.78
VFR100_40	15.5	4.33	4.71	5.55	15.56	4.96	8.87	8.49	1.32	4.99
VFR100_60	15.54	4.59	4.59	8.23	14.92	5.84	8.89	8.24	2.55	5.72
VFR200_20	9.48	2.57	2.4	1.52	6.76	0.9	10.99	10.01	0.97	0.55
VFR200_40	17.86	3.7	3.79	2.15	17.5	4.73	14.79	13.94	**-0.26**	4.75
VFR200_60	17.84	4.65	5.12	6.68	17.76	6.99	15.04	15.07	1.19	7.06
VFR300_20	6.04	1.72	1.8	1.26	2.52	0.42	7.71	7.29	1.15	**0.01**
VFR300_40	17.3	3.68	4.14	1.36	17.06	5.2	16.95	15.91	**-0.69**	5.11
VFR300_60	18.13	5.0	5.49	5.26	18.05	7.56	12.47	12.0	0.41	7.34
VFR400_20	4.88	1.41	1.46	1.25	2.35	0.32	4.31	4.0	0.83	**-0.17**
VFR400_40	15.31	2.78	3.32	0.61	15.19	4.75	15.87	15.88	**-1.04**	5.16
VFR400_60	17.5	4.45	4.89	3.5	17.62	7.66	17.34	17.46	**-0.45**	7.42
VFR500_20	4.11	1.0	1.06	1.14	1.68	0.22	3.18	3.14	0.88	**-0.14**
VFR500_40	14.69	2.84	3.16	0.35	14.63	5.04	13.67	13.93	**-0.69**	4.91
VFR500_60	17.08	5.4	5.97	2.22	17.21	8.09	17.08	16.84	**-1.04**	7.63
VFR600_20	3.33	1.0	0.78	1.06	1.42	0.28	3.53	3.34	0.87	**-0.12**
VFR600_40	13.89	2.58	2.58	0.53	13.44	5.05	16.17	16.81	**-0.57**	4.72
VFR600_60	17.47	5.02	5.6	2.01	16.91	8.09	12.78	12.34	**-1.15**	7.38
VFR700_20	3.05	0.87	0.89	1.34	1.34	0.16	2.45	2.23	0.95	**-0.12**
VFR700_40	12.82	2.15	2.38	0.0	12.7	4.6	12.59	11.9	**-0.44**	4.11
VFR700_60	16.22	4.9	4.64	1.2	16.17	7.93	15.99	15.51	**-1.38**	7.46
VFR800_20	2.8	0.81	0.84	1.19	1.07	0.21	2.12	2.05	0.78	**-0.11**
VFR800_40	11.05	2.2	2.25	0.16	11.0	4.08	14.59	13.62	**-0.46**	3.9
VFR800_60	15.45	4.61	4.99	0.36	15.87	7.84	16.04	16.35	**-1.42**	7.67

Figure 3: Average Relative Percentage Deviation (ARPD) of all the presented algorithms on the Taillard and VFR instances for the makespan minimization version. Bold values indicate that the algorithm obtained significantly better results than the other 9 algorithms according to the Wilcoxon signed-rank test with a 95% confidence interval (time limit: $n.m.90/2$).

the bound for this branching strategy. Finally, using the weighted idle time proves to be a significant bonus and largely improves the quality of the solutions, from -1.42% to 2.55% ARPD. The *gap* guide also allows improving the results obtained by the *bound* guide. These results show that the bi-directional search with the *walpha* guide performs well on most instances (especially those with a high number of machines) and the *gap* guide performs well on instances with fewer machines. Thus, we use these algorithms to compare with the state-of-the-art algorithms.

5.1.2. Comparison with the state-of-the-art algorithms

The best performing algorithms in the literature are: The Variable Block Insertion Heuristic (VBIH) [16], the Best-of-Breed Iterated Greedy algorithm (IGbob) [8], and, the Iterated Greedy designed using the EMILI framework (IGirms) [30]. Figure 4 compares the performance of our algorithms with the VBIH algorithm. VBIH results are obtained from the supplementary materials of [16]. CPU times are regularized to make a fair comparison. We do not include results on the Taillard dataset as the authors of VBIH only compared it using the VFR benchmark. Figure 5 compares the performance of our algorithms with the IGirms algorithm. IGirms results are obtained from the supple-

mentary materials of [30][4]. IGirms authors provide their ARPD values but not the solutions obtained for each instance (thus, we cannot apply the Wilcoxon signed-rank test). Figure 6 compares the performance of our algorithms compared to the IGbob algorithm [8]. As the authors provide their source-code, we executed their algorithm on our machine. Figure 7 presents Pareto diagrams showing the time/performance trade off of our algorithms and the state-of-the-art algorithms for 2 of the largest instance families (VFR800_20, VFR800_60).

	*n.m.*30/2 CPU-regularized ms			*n.m.*60/2 CPU-regularized ms			*n.m.*90/2 CPU-regularized ms		
instance set	VBIH	IBS walpha	IBS gap	VBIH	IBS walpha	IBS gap	VBIH	IBS walpha	IBS gap
VFR100_20	0.31	0.56	1.92	0.26	0.43	1.78	0.06	0.43	1.78
VFR100_40	**0.52**	1.66	5.57	**0.46**	1.43	5.29	**0.24**	1.32	4.99
VFR100_60	**0.65**	3.04	6.59	**0.58**	2.75	6.15	**0.29**	2.55	5.72
VFR200_20	0.22	1.08	0.72	0.2	1.03	0.55	0.09	0.97	0.55
VFR200_40	0.57	**0.07**	5.47	0.52	**-0.16**	5.12	0.24	-0.26	4.75
VFR200_60	**0.61**	1.52	7.45	**0.58**	1.23	7.28	**0.29**	1.19	7.06
VFR300_20	0.21	1.2	0.12	0.17	1.15	0.01	0.12	1.15	0.01
VFR300_40	0.53	**-0.6**	5.41	0.49	**-0.69**	5.11	0.31	**-0.69**	5.11
VFR300_60	0.66	0.66	7.76	0.62	0.41	7.34	0.33	0.41	7.34
VFR400_20	0.16	0.91	-0.01	0.13	0.86	-0.09	0.04	0.83	-0.17
VFR400_40	0.47	**-0.84**	5.81	0.43	**-0.93**	5.31	0.21	**-1.04**	5.16
VFR400_60	0.58	**-0.11**	8.04	0.54	**-0.24**	8.04	0.2	**-0.45**	7.42
VFR500_20	0.13	0.91	-0.04	0.12	0.88	**-0.14**	0.05	0.88	-0.14
VFR500_40	0.55	**-0.66**	5.12	0.49	**-0.68**	4.91	0.3	**-0.69**	4.91
VFR500_60	0.41	**-0.61**	7.92	0.37	**-0.79**	7.68	0.15	**-1.04**	7.63
VFR600_20	0.12	0.89	**-0.1**	0.11	0.87	**-0.12**	0.07	0.87	-0.12
VFR600_40	0.5	**-0.52**	4.83	0.38	**-0.57**	4.72	0.23	**-0.57**	4.72
VFR600_60	0.64	**-1.01**	7.38	0.5	**-1.15**	7.38	0.27	**-1.15**	7.38
VFR700_20	0.1	1.01	**-0.11**	0.06	0.98	**-0.12**	0.04	0.95	-0.12
VFR700_40	0.42	**-0.39**	4.14	0.29	**-0.44**	4.11	0.17	**-0.44**	4.11
VFR700_60	0.56	**-1.2**	7.7	0.41	**-1.38**	7.46	0.31	**-1.38**	7.46
VFR800_20	0.07	0.82	**-0.08**	0.06	0.8	**-0.1**	0.03	0.78	-0.11
VFR800_40	0.32	**-0.38**	3.99	0.3	**-0.4**	3.9	0.22	**-0.46**	3.9
VFR800_60	0.41	**-1.29**	7.76	0.37	**-1.37**	7.76	0.28	**-1.42**	7.67

Figure 4: Comparison of Average Relative Percentage Deviation (ARPD) with VBIH. Bold values indicate that the algorithm obtained significantly better results than the others according to the Wilcoxon signed-rank test with a 95% confidence interval (makespan minimization).

Discussions: From Tables 4,5,6, we remark that the *walpha iterative beam search* perform significantly better on large instances (more than 500 jobs and 40 machines). It often reports negative ARPD (meaning that it was able to consistently report new-best-known solutions compared to IGirms), even on short computation times. Notice, on the Pareto diagrams 7, that the iterative beam searches find better solutions in shorter computation times on large instances than all the reported state-of-the-art results. We also notice that the CPU time to strictly improve the best-known solutions by Iterative Beam Search has a median of 25 seconds on VFR_800_60 class (83 times faster than the time limit), and 13 seconds on the VFR_800_40 class (106 times faster than the time limit).

5.2. Flowtime minimization

5.2.1. Comparison with the state-of-the-art algorithms

The best performing algorithms in the literature are: IGA [31], ALGirtct [30], MRSILS(BSCH) [7] and Shake-LS [1]. Figure 8 compares our algorithms with ALGirtct and IGA with the results presented in [30]. Figure 9 compares our algorithms with MRSILS(BSCH). For both tables, the authors provide their ARPD values but not the solutions obtained for each instance (thus, we cannot apply the Wilcoxon signed-rank test). Figure 7 presents Pareto diagrams to evaluate our algorithms with state-of-the-art algorithms. Finally, the authors of Shake-LS report the best results obtained by their algorithm (30 independent runs of 1 hour). We do not directly compare our running times (that

[4] http://iridia.ulb.ac.be/supp/IridiaSupp2018-002/

	n.m.60/2 CPU-regularized ms			*n.m*.120/2 CPU-regularized ms			*n.m*.240/2 CPU-regularized ms		
instance set	IGirms	IBS walpha	IBS gap	IGirms	IBS walpha	IBS gap	IGirms	IBS walpha	IBS gap
TAI20_5	0.03	0.0	0.0	0.02	0.0	0.0	0.01	0.0	0.0
TAI20_10	0.01	0.0	0.07	0.01	0.0	0.03	0.01	0.0	0.0
TAI20_20	0.01	0.45	0.75	0.01	0.3	0.66	0.01	0.28	0.64
TAI50_5	0.0	0.1	0.0	0.0	0.08	0.0	0.0	0.06	0.0
TAI50_10	0.3	1.02	0.11	0.28	0.89	0.1	0.25	0.64	0.09
TAI50_20	0.47	0.62	2.92	0.39	0.52	2.81	0.34	0.45	2.5
TAI100_5	0.0	0.09	0.0	0.0	0.08	0.0	0.0	0.08	0.0
TAI100_10	0.03	1.18	0.0	0.03	1.07	0.0	0.02	0.86	0.0
TAI100_20	0.62	1.65	1.42	0.52	1.52	1.15	0.44	1.52	1.14
TAI200_10	0.03	2.24	1.05	0.03	1.95	0.94	0.03	1.74	0.87
TAI200_20	0.66	1.88	0.92	0.57	1.85	0.7	0.51	1.81	0.59
TAI500_20	0.29	1.49	0.18	0.26	1.47	0.09	0.24	1.43	0.06
VFR100_20	0.57	0.56	2.06	0.42	0.43	1.88	0.29	0.43	1.78
VFR100_40	0.67	1.66	5.57	0.49	1.43	5.29	0.35	1.32	4.99
VFR100_60	0.64	3.04	6.59	0.48	2.75	6.15	0.34	2.55	5.72
VFR200_20	0.45	1.08	0.72	0.32	1.03	0.55	0.22	0.97	0.55
VFR200_40	0.79	0.07	5.47	0.52	-0.16	5.12	0.3	-0.26	4.75
VFR200_60	0.74	1.82	7.85	0.5	1.49	7.45	0.28	1.19	7.06
VFR300_20	0.35	1.24	0.13	0.24	1.2	0.1	0.17	1.15	0.01
VFR300_40	0.7	-0.48	5.75	0.48	-0.6	5.41	0.24	-0.69	5.11
VFR300_60	0.77	1.08	7.97	0.53	0.66	7.76	0.29	0.41	7.34
VFR400_20	0.21	0.91	-0.01	0.16	0.86	-0.07	0.12	0.83	-0.17
VFR400_40	0.62	-0.84	5.81	0.42	-0.93	5.31	0.22	-1.04	5.16
VFR400_60	0.68	0.07	8.37	0.46	-0.19	8.04	0.23	-0.45	7.42
VFR500_20	0.17	0.95	-0.0	0.13	0.91	-0.14	0.09	0.88	-0.14
VFR500_40	0.54	-0.66	5.12	0.37	-0.68	4.91	0.2	-0.69	4.91
VFR500_60	0.61	-0.61	7.92	0.41	-0.79	7.68	0.21	-1.04	7.63
VFR600_20	0.17	0.92	-0.08	0.13	0.89	-0.1	0.09	0.87	-0.12
VFR600_40	0.52	-0.49	5.41	0.34	-0.52	4.83	0.17	-0.57	4.72
VFR600_60	0.62	-0.69	7.59	0.42	-1.01	7.38	0.21	-1.15	7.38
VFR700_20	0.14	1.01	-0.11	0.1	0.98	-0.11	0.07	0.95	-0.12
VFR700_40	0.48	-0.37	4.33	0.31	-0.39	4.14	0.15	-0.44	4.11
VFR700_60	0.57	-1.06	7.95	0.37	-1.2	7.7	0.19	-1.38	7.46
VFR800_20	0.15	0.82	-0.08	0.1	0.8	-0.1	0.07	0.78	-0.11
VFR800_40	0.46	-0.38	3.99	0.29	-0.4	3.9	0.14	-0.46	3.9
VFR800_60	0.51	-1.11	7.94	0.32	-1.29	7.76	0.15	-1.42	7.67

Figure 5: Comparison of Average Relative Percentage Deviation (ARPD) with IGirms using the ARPD presented in the literature [30]. CPU times are regularized as described in Section 5 (makespan minimization).

are much shorter), but we are still able to report many new-best-known solutions in Appendix B showing that our algorithm can compete with Shake-LS. All the algorithms presented above perform their experiments on the Taillard dataset [41].

Discussion: We observe that the iterative beam search performs well for many instances and finds new-best-known solutions. Compared to existing beam search in the literature [7], our beam search has a simpler guidance strategy that only considers the flowtime of the partial solution and an estimation of the idle time. It also has a geometric growth that allows regularly improving the best-so-far solution during the search (anytime performance).

It is worth noticing that IBS reports better ARPDs on all instance classes compared to MRSILS(BSCH), IGA, or ALGirtct (Figures 9,8). IBS also provides better ARPD on all instance classes with 100 jobs or more than BSCH, with similar running times (Figures 9,10). We may also note that there exists some work that provide some acceleration to insertion-based algorithms (thus IGA or MRSILS could benefit from it) [9]. This work could provide a speed-up up to 3 times faster. However, despite this speed-up, it is unlikely that either IGA or MRSILS can outperform the iterative beam search. Indeed, for IGA, on instances with 100 jobs or more, the iterative beam search provides better

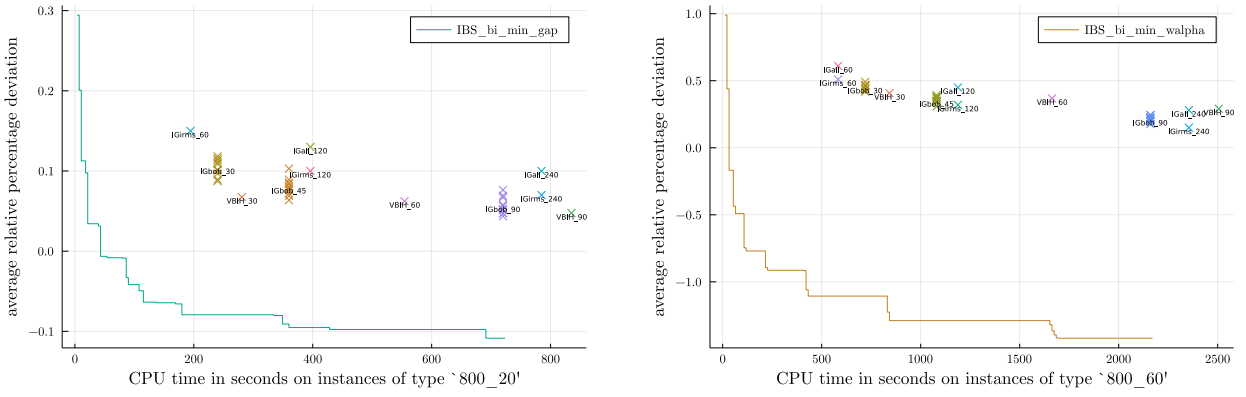
	$n.m.30/2$ ms			$n.m.45/2$ ms			$n.m.90/2$ ms		
instance set	IGbob	IBS walpha	IBS gap	IGbob	IBS walpha	IBS gap	IGbob	IBS walpha	IBS gap
TAI20_5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
TAI20_10	0.0	0.0	0.07	0.0	0.0	0.03	0.0	0.0	0.0
TAI20_20	**0.0**	0.34	0.72	**0.0**	0.34	0.66	**0.0**	0.28	0.64
TAI50_5	0.0	0.08	0.0	0.0	0.08	0.0	0.0	0.06	0.0
TAI50_10	0.34	1.02	0.11	0.32	0.89	0.1	0.3	0.64	0.09
TAI50_20	0.46	0.55	2.81	0.43	0.55	2.81	0.33	0.45	2.5
TAI100_5	0.0	0.09	0.0	0.0	0.09	0.0	0.0	0.08	0.0
TAI100_10	0.03	1.18	0.0	0.02	1.07	0.0	0.02	0.86	0.0
TAI100_20	0.63	1.65	1.24	0.58	1.65	1.24	0.54	1.52	1.14
TAI200_10	0.03	2.22	1.05	0.03	1.95	0.94	0.03	1.74	0.87
TAI200_20	0.67	1.88	0.92	0.64	1.85	0.92	0.54	1.81	0.59
TAI500_20	0.27	1.47	**0.09**	0.27	1.47	0.09	0.24	1.43	**0.06**
VFR100_20	0.61	0.56	1.92	0.55	0.56	1.92	0.47	0.43	1.78
VFR100_40	**0.73**	1.66	5.57	**0.68**	1.43	5.29	**0.52**	1.32	4.99
VFR100_60	**0.8**	3.04	6.59	**0.65**	2.75	6.15	**0.48**	2.55	5.72
VFR200_20	0.45	1.08	0.72	0.42	1.05	0.63	0.28	0.97	0.55
VFR200_40	0.85	**0.07**	5.47	0.69	**-0.16**	5.12	0.52	**-0.26**	4.75
VFR200_60	**0.8**	1.82	7.85	**0.65**	1.49	7.45	**0.43**	1.19	7.06
VFR300_20	0.44	1.24	**0.13**	0.35	1.2	0.12	0.27	1.15	**0.01**
VFR300_40	0.73	**-0.6**	5.41	0.6	**-0.6**	5.41	0.43	**-0.69**	5.11
VFR300_60	0.79	0.66	7.76	0.67	0.66	7.76	0.48	0.41	7.34
VFR400_20	0.27	0.91	**-0.01**	0.2	0.86	**-0.07**	0.16	0.83	**-0.17**
VFR400_40	0.61	**-0.84**	5.81	0.49	**-0.93**	5.31	0.28	**-1.04**	5.16
VFR400_60	0.64	**0.07**	8.37	0.53	**-0.19**	8.04	0.32	**-0.45**	7.42
VFR500_20	0.21	0.91	**-0.04**	0.19	0.91	**-0.1**	0.14	0.88	**-0.14**
VFR500_40	0.6	**-0.66**	5.12	0.49	**-0.68**	5.12	0.32	**-0.69**	4.91
VFR500_60	0.45	**-0.61**	7.92	0.36	**-0.75**	7.68	0.2	**-1.04**	7.63
VFR600_20	0.17	0.92	**-0.09**	0.14	0.89	**-0.1**	0.1	0.87	**-0.12**
VFR600_40	0.4	**-0.52**	4.83	0.3	**-0.52**	4.83	0.16	**-0.57**	4.72
VFR600_60	0.55	**-1.01**	7.38	0.44	**-1.01**	7.38	0.27	**-1.15**	7.38
VFR700_20	0.12	1.01	**-0.11**	0.1	0.98	**-0.11**	0.08	0.95	**-0.12**
VFR700_40	0.39	**-0.37**	4.33	0.31	**-0.39**	4.14	0.16	**-0.44**	4.11
VFR700_60	0.45	**-1.06**	7.95	0.36	**-1.2**	7.7	0.2	**-1.38**	7.46
VFR800_20	0.1	0.82	**-0.08**	0.08	0.8	**-0.1**	0.06	0.78	**-0.11**
VFR800_40	0.36	**-0.38**	3.99	0.27	**-0.4**	3.9	0.15	**-0.46**	3.9
VFR800_60	0.45	**-1.11**	7.94	0.35	**-1.29**	7.76	0.21	**-1.42**	7.67

Figure 6: Comparison of Average Relative Percentage Deviation (ARPD) with IGbob. Bold values indicate that the algorithm obtained significantly better results than the others according to the Wilcoxon signed-rank test with a 95% confidence interval (makespan minimization).

ARPDs than the IGA more than 200 times faster (see 10). For MRSILS, the iterative beam search outperforms its ARPDs even before its initialization (BSCH) finishes (see 9). Moreover, IBS only requires a fraction of the time limit to report new-best-known solutions. Indeed, for the TAI_500_20 class, the median time required to report new-best-known solutions is of 25 seconds while the time limit is of 450 seconds, thus 18 times faster (56 times faster for the TAI_200_20 class). For these reasons, this speed-up mechanism is certainly not enough for insertion-based algorithms to outperform iterative beam search.

6. Conclusions & perspectives

In this paper, we present some iterative beam search algorithms applied to the permutation flowshop problem (makespan and flowtime minimization). These algorithms use branching strategies inspired by the LR heuristic (forward branching) and recent branch-and-bound schemes [13] (bi-directional branching). We compare several guidance strategies (starting from the bound as commonly done in most branch-and-bounds) to more advanced ones (LR-inspired



(a) Aggregated results for VFR800_20_X instances

(b) Aggregated results for VFR800_60_X instances

Figure 7: solution-quality/regularized-CPU time Pareto diagram comparing IBS algorithms with the state-of-the-art meta-heuristics on the largest VFR instances (makespan minimization).

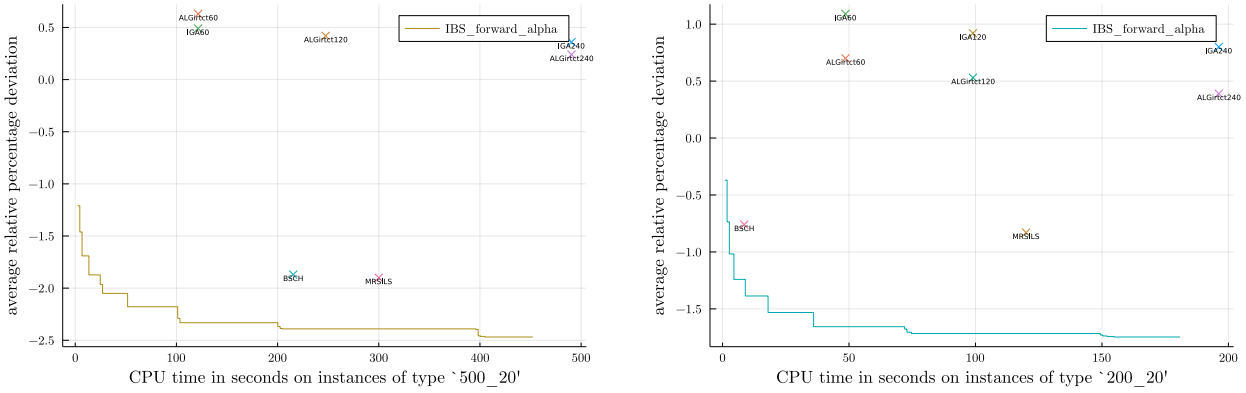
	*n.m.*30			*n.m.*60			*n.m.*120		
instance set	**IGA**	**ALGirtct**	**IBS alpha**	**IGA**	**ALGirtct**	**IBS alpha**	**IGA**	**ALGirtct**	**IBS alpha**
TAI20_5	0.15	0.0	0.0	0.15	0.0	0.0	0.15	0.0	0.0
TAI20_10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
TAI20_20	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
TAI50_5	0.64	0.47	0.13	0.54	0.38	0.12	0.48	0.31	0.11
TAI50_10	1.1	0.51	0.1	1.04	0.41	0.07	0.99	0.35	0.07
TAI50_20	0.72	0.45	0.17	0.66	0.35	0.13	0.61	0.29	0.12
TAI100_5	1.17	0.99	-0.14	1.08	0.89	-0.15	0.99	0.81	-0.16
TAI100_10	1.49	1.03	-0.33	1.37	0.9	-0.38	1.29	0.79	-0.4
TAI100_20	1.54	1.15	-0.21	1.4	0.97	-0.3	1.3	0.83	-0.3
TAI200_10	1.27	0.86	-1.01	1.17	0.73	-1.07	1.09	0.64	-1.1
TAI200_20	1.09	0.7	-1.66	0.92	0.53	-1.72	0.8	0.39	-1.75
TAI500_20	0.49	0.63	-2.33	0.42	0.42	-2.39	0.36	0.24	-2.47

Figure 8: Comparison of Average Relative Percentage Deviation (ARPD) with ALGirtct and IGA for the flowtime minimization variant (Taillard benchmark, flowtime minimization)

guidance). We show that the combination of all of these components obtains state-of-the-art performance on large instances. Furthermore, we report 111 new-best-so-far solutions for the permutation flowshop (makespan minimization) on the open instances of the VFR benchmark and 58 new-best-so-far solutions for the permutation flowshop (flowtime minimization) on the open instances of the Taillard benchmark 1 to 2 orders of magnitude faster than previous works. These algorithms compare, and sometimes perform better, than the algorithms based on the NEH branching scheme (which is usually considered as "the most efficient constructive heuristic for the problem" [8]) and the iterated greedy algorithm (again considered as "the most efficient approximate algorithm for the problem" [8]). We believe that the performance of the bi-directional branching combined to the iterative beam search highlighted in this paper could draw the interest of the community for these techniques as they are rather unexplored, although simple and efficient. On the makespan minimization, we recommend using an iterative beam search on large instances (300 jobs or more) and an iterated greedy algorithm on instances with less than 300 jobs to obtain state-of-the-art performance on all the benchmark instances. Studying these techniques leads to a few other questions: we considered the iterative beam search and showed that it is competitive with classical meta-heuristics for the permutation flowshop. However, many others exist. For instance Iterative Memory Bounded A* [10, 23], Beam Stack Search [48], Anytime Column Search [42]. To the best of our knowledge, they have not been tested yet for the permutation flowshop. In this paper, we studied the makespan and flowtime minimization criteria and achieved competitive results. Many more flowshop variants have been studied. For instance, the blocking flowshop, the distributed permutation flowshop and many others. It could be interesting to assess the performance of the LR-based beam search on these variants.

instance set	BSCH	IBS alpha	MRSILS(BSCH)	IBS alpha
TAI20_5	1.25	0.91	0.01	0.0
TAI20_10	0.75	1.27	0.0	0.0
TAI20_20	0.75	1.44	0.0	0.0
TAI50_5	0.75	0.63	0.28	0.12
TAI50_10	1.04	1.66	0.47	0.07
TAI50_20	1.48	1.73	0.63	0.12
TAI100_5	0.3	-0.0	0.22	-0.15
TAI100_10	0.57	0.06	0.27	-0.38
TAI100_20	1.14	0.58	0.83	-0.3
TAI200_10	-0.61	-0.92	-0.71	-1.07
TAI200_20	-0.76	-1.29	-0.83	-1.72
TAI500_20	-1.87	-2.39	-1.9	-2.39

Figure 9: Comparison of Average Relative Percentage Deviation (ARPD) with BSCH and MRSILS(BSCH) (flowtime minimization). We compare IBS with BSCH using the same time when BSCH finishes. We compare IBS with MRSILS(BSCH) with the same running times ($n.m.30$ CPU-regularized ms)



(a) Aggregated results for TAI500_20_X instances

(b) Aggregated results for TAI200_20_X instances

Figure 10: solution-quality/regularized-CPU time Pareto diagram comparing IBS algorithms with the state-of-the-art meta-heuristics on the largest Taillard instances (flowtime minimization).

References

[1] Andrade, C.E., Silva, T., Pessoa, L.S., 2019. Minimizing flowtime in a flowshop scheduling problem with a biased random-key genetic algorithm. Expert Systems with Applications 128, 67–80.

[2] Carlier, J., Rebaï, I., 1996. Two branch and bound algorithms for the permutation flow shop problem. European Journal of Operational Research 90, 238–251.

[3] Chakroun, I., Melab, N., Mezmaz, M., Tuyttens, D., 2013. Combining multi-core and gpu computing for solving combinatorial optimization problems. Journal of Parallel and Distributed Computing 73, 1563–1577.

[4] Dong, X., Huang, H., Chen, P., 2008. An improved neh-based heuristic for the permutation flowshop problem. Computers & Operations Research 35, 3962–3968.

[5] Drozdowski, M., Marciniak, P., Pawlak, G., Płaza, M., 2011. Grid branch-and-bound for permutation flowshop, in: International Conference on Parallel Processing and Applied Mathematics, Springer. pp. 21–30.

[6] Fernandez-Viagas, V., Framinan, J.M., 2015. A new set of high-performing heuristics to minimise flowtime in permutation flowshops. Computers & Operations Research 53, 68–80.

[7] Fernandez-Viagas, V., Framinan, J.M., 2017. A beam-search-based constructive heuristic for the pfsp to minimise total flowtime. Computers & Operations Research 81, 167–177.

[8] Fernandez-Viagas, V., Framinan, J.M., 2019. A best-of-breed iterated greedy for the permutation flowshop scheduling problem with makespan objective. Computers & Operations Research 112, 104767.

[9] Fernandez-Viagas, V., Molina-Pariente, J.M., Framinan, J.M., 2020. Generalised accelerations for insertion-based heuristics in permutation flowshop scheduling. European Journal of Operational Research 282, 858–872.

[10] Fontan, F., Libralesso, L., 2020. Packingsolver: a tree search-based solver for two-dimensional two-and three-staged guillotine packing problems .

[11] Framinan, J.M., Leisten, R., 2003. An efficient constructive heuristic for flowtime minimisation in permutation flow shops. Omega 31, 311–317.
[12] Gao, J., Chen, R., 2011. A hybrid genetic algorithm for the distributed permutation flowshop scheduling problem. International Journal of Computational Intelligence Systems 4, 497–508.
[13] Gmys, J., Mezmaz, M., Melab, N., Tuyttens, D., 2020. A computationally efficient branch-and-bound algorithm for the permutation flow-shop scheduling problem. European Journal of Operational Research .
[14] Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.R., 1979. Optimization and approximation in deterministic sequencing and scheduling: a survey, in: Annals of discrete mathematics. Elsevier. volume 5, pp. 287–326.
[15] Kalczynski, P.J., Kamburowski, J., 2008. An improved neh heuristic to minimize makespan in permutation flow shops. Computers & Operations Research 35, 3001–3008.
[16] Kizilay, D., Tasgetiren, M.F., Pan, Q.K., Gao, L., . A variable block insertion heuristic for solving permutation flow shop scheduling problem with makespan criterion 12, 100. URL: https://www.mdpi.com/1999-4893/12/5/100, doi:10.3390/a12050100. number: 5 Publisher: Multidisciplinary Digital Publishing Institute.
[17] Krajewski, L.J., King, B.E., Ritzman, L.P., Wong, D.S., 1987. Kanban, mrp, and shaping the manufacturing environment. Management science 33, 39–57.
[18] Kurdi, M., 2020. A memetic algorithm with novel semi-constructive evolution operators for permutation flowshop scheduling problem. Applied Soft Computing , 106458.
[19] Ladhari, T., Haouari, M., 2005. A computational study of the permutation flow shop problem based on a tight lower bound. Computers & Operations Research 32, 1831–1847.
[20] Lemesre, J., Dhaenens, C., Talbi, E.G., 2007. An exact parallel method for a bi-objective permutation flowshop problem. European Journal of Operational Research 177, 1641–1655.
[21] Li, B.B., Wang, L., Liu, B., 2008. An effective pso-based hybrid algorithm for multiobjective permutation flow shop scheduling. IEEE transactions on systems, man, and cybernetics-part A: systems and humans 38, 818–831.
[22] Libralesso, L., Bouhassoun, A.M., Cambazard, H., Jost, V., 2019. Tree search algorithms for the sequential ordering problem. arXiv preprint arXiv:1911.12427 .
[23] Libralesso, L., Fontan, F., 2020. An anytime tree search algorithm for the 2018 roadef/euro challenge glass cutting problem. arXiv preprint arXiv:2004.00963 .
[24] Libralesso, L., Secardin, A., Jost, V., 2020. Longest common subsequence: an algorithmic component analysis. URL: https://hal.archives-ouvertes.fr/hal-02895115. working paper or preprint.
[25] Liu, J., Reeves, C.R., 2001. Constructive and composite heuristic solutions to the $p//\sum c_i$ scheduling problem. European Journal of Operational Research 132, 439–452.
[26] Liu, W., Jin, Y., Price, M., 2017. A new improved neh heuristic for permutation flowshop scheduling problems. International Journal of Production Economics 193, 21–30.
[27] Nagano, M., Moccellin, J., 2002. A high quality solution constructive heuristic for flow shop sequencing. Journal of the Operational Research Society 53, 1374–1379.
[28] Nawaz, M., Enscore Jr, E.E., Ham, I., 1983. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. Omega 11, 91–95.
[29] Nowicki, E., 1999. The permutation flow shop with buffers: A tabu search approach. European Journal of Operational Research 116, 205–219.
[30] Pagnozzi, F., Stützle, T., 2019. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. European Journal of Operational Research 276, 409–421.
[31] Pan, Q.K., Ruiz, R., 2012. Local search methods for the flowshop scheduling problem with flowtime minimization. European Journal of Operational Research 222, 31–43.
[32] Pan, Q.K., Ruiz, R., 2013. A comprehensive review and evaluation of permutation flowshop heuristics to minimize flowtime. Computers & Operations Research 40, 117–128.
[33] Pan, Q.K., Ruiz, R., 2014. An effective iterated greedy algorithm for the mixed no-idle permutation flowshop scheduling problem. Omega 44, 41–50.
[34] Potts, C., 1980. An adaptive branching rule for the permutation flow-shop problem. European Journal of Operational Research 5, 19–25.
[35] Reddy, D.R., et al., 1977. Speech understanding systems: A summary of results of the five-year research effort. department of computer science.
[36] Reza Hejazi*, S., Saghafian, S., 2005. Flowshop-scheduling problems with makespan criterion: a review. International Journal of Production Research 43, 2895–2929.
[37] Ribas, I., Mateo, M., 2009. Improvement tools for neh based heuristics on permutation and blocking flow shop scheduling problems, in: IFIP International Conference on Advances in Production Management Systems, Springer. pp. 33–40.
[38] Ritt, M., 2016. A branch-and-bound algorithm with cyclic best-first search for the permutation flow shop scheduling problem, in: 2016 IEEE International Conference on Automation Science and Engineering (CASE), IEEE. pp. 872–877.
[39] Ruiz, R., Stützle, T., 2007. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. European Journal of Operational Research 177, 2033–2049.
[40] Taillard, E., 1990. Some efficient heuristic methods for the flow shop sequencing problem. European journal of Operational research 47, 65–74.
[41] Taillard, E., 1993. Benchmarks for basic scheduling problems. european journal of operational research 64, 278–285.
[42] Vadlamudi, S.G., Gaurav, P., Aine, S., Chakrabarti, P.P., 2012. Anytime column search, in: Australasian Joint Conference on Artificial Intelligence, Springer. pp. 254–265.
[43] Vakharia, A.J., Wemmerlov, U., 1990. Designing a cellular manufacturing system: a materials flow approach based on operation sequences. IIE transactions 22, 84–97.

[44] Vallada, E., Ruiz, R., Framinan, J.M., 2015. New hard benchmark for flowshop scheduling problems minimising makespan. European Journal of Operational Research 240, 666–677.
[45] Vasiljevic, D., Danilovic, M., 2015. Handling ties in heuristics for the permutation flow shop scheduling problem. Journal of Manufacturing Systems 35, 1–9.
[46] Wang, L., Pan, Q.K., Tasgetiren, M.F., 2011. A hybrid harmony search algorithm for the blocking permutation flow shop scheduling problem. Computers & Industrial Engineering 61, 76–83.
[47] Zheng, D.Z., Wang, L., 2003. An effective hybrid heuristic for flow shop scheduling. The International Journal of Advanced Manufacturing Technology 21, 38–44.
[48] Zhou, R., Hansen, E.A., 2005. Beam-stack search: Integrating backtracking with beam search., in: ICAPS, pp. 90–98.

A. Notations

- J: all the jobs
- M: all the machines
- n: job number ($n = |J|$)
- m: machine number ($m = |M|$)
- F (resp. B): all the jobs scheduled in the prefix (resp. suffix)
- $Cmax_{F,i}$: first availability of machine i in the forward search
- $Cmax_{B,i}$: first availability of machine i in the backward search
- R_i: remaining processing time on machine i. $R_i = \sum_{j \in J \setminus \{F \cup B\}} p_{ij}$
- $I_{f,i}$: total idle time on machine i in the forward search ($I_{f,i} = Cmax_{f,i} - \sum_{j \in F} p_{i,j}$)
- $I_{b,i}$: total idle time on machine i in the backward search ($I_{b,i} = Cmax_{b,i} - \sum_{j \in B} p_{i,j}$)
- α: proportion of scheduled jobs. $\alpha = \frac{|F|+|B|}{|J|}$ on bi-directional branching or $\alpha = \frac{|F|}{|J|}$ on forward branching.
- g_{bound}: guidance function based on the bound (makespan or flowtime)
- g_{idle}: guidance function based only by the idle time
- g_{alpha}: guidance function based on both the bound and idle time
- g_{walpha}: guidance function based on both the bound and the proportion of idle time in the partial solution
- g_{gap}: guidance function based on both the gap, bound, and weighted idle time

B. Detailed numerical results

instance	best-known	IGrms	VBIH	GmysBB	IGbob	IBS_bimin_walpha	IBS_bimin_gap	time_to_improve_bks (s)
VFR100_20_1	6.121	6.176	6.173	**6.121**	6.188	6.163	6.229	-
VFR100_20_2	6.119	**6.119**	6.221	6.224	6.282	6.282	6.350	-
VFR100_20_3	6.157	6.267	6.227	**6.157**	6.221	6.231	6.296	-
VFR100_20_4	6.173	6.210	6.264	**6.173**	6.237	6.250	6.287	-
VFR100_20_5	6.221	6.223	6.285	**6.221**	6.268	6.319	6.404	-
VFR100_20_6	6.247	6.260	6.401	**6.247**	6.301	6.333	6.445	-
VFR100_20_7	6.074	6.274	**6.074**	6.358	6.419	6.394	6.561	-
VFR100_20_8	6.023	6.411	6.328	**6.023**	6.091	6.091	6.149	-
VFR100_20_9	6.074	**6.074**	6.125	6.286	6.345	6.345	6.443	-
VFR100_20_10	6.048	6.324	6.267	**6.048**	6.136	6.175	6.266	-
VFR100_40_1	7.840	7.840	7.846	-	**7.836**	7.947	8.223	-
VFR100_40_2	7.894	**7.894**	**7.894**	-	7.981	8.014	8.294	-
VFR100_40_3	7.913	7.957	7.913	-	**7.893**	7.963	8.261	-
VFR100_40_4	7.889	**7.889**	7.997	-	7.920	7.981	8.240	-
VFR100_40_5	7.895	**7.895**	7.993	-	8.002	8.045	8.357	-
VFR100_40_6	7.968	**7.968**	7.980	-	8.013	8.096	8.415	-
VFR100_40_7	7.957	7.988	**7.957**	-	7.982	8.117	8.369	-
VFR100_40_8	7.888	7.956	**7.888**	-	7.951	8.084	8.355	-
VFR100_40_9	7.917	7.936	7.917	-	**7.899**	7.963	8.344	-
VFR100_40_10	7.853	**7.853**	7.976	-	7.912	8.014	8.266	-
VFR100_60_1	9.326	**9.326**	9.353	-	9.366	9.618	9.898	-
VFR100_60_2	9.349	9.547	**9.349**	-	9.550	9.698	10.071	-
VFR100_60_3	9.403	9.513	9.403	-	**9.315**	9.596	9.802	-
VFR100_60_4	9.316	**9.316**	9.431	-	9.394	9.521	9.912	-
VFR100_60_5	9.366	**9.366**	9.630	-	9.422	9.628	9.861	-
VFR100_60_6	9.346	9.391	**9.346**	-	9.642	9.867	10.111	-
VFR100_60_7	9.523	9.622	9.523	-	**9.367**	9.609	9.944	-
VFR100_60_8	9.326	**9.326**	9.488	-	9.530	9.766	10.055	-
VFR100_60_9	9.507	9.507	9.572	-	**9.502**	9.730	10.046	-
VFR100_60_10	9.480	**9.480**	9.567	-	9.555	9.755	10.083	-
VFR200_20_1	11.181	11.271	11.272	**11.181**	11.289	11.405	11.260	-
VFR200_20_2	11.254	11.286	11.294	11.254	11.227	11.424	**11.161**	87.95
VFR200_20_3	11.188	11.227	**11.188**	11.233	11.311	11.362	11.509	-
VFR200_20_4	11.090	11.297	11.143	**11.090**	11.193	11.243	11.220	-
VFR200_20_5	11.076	11.175	11.310	**11.076**	11.168	11.254	11.274	-
VFR200_20_6	11.152	**11.152**	11.365	11.208	11.321	11.355	11.339	-
VFR200_20_7	11.128	11.301	**11.128**	11.266	11.364	11.432	11.438	-
VFR200_20_8	11.041	11.347	11.091	**11.041**	11.118	11.250	**11.041**	-
VFR200_20_9	11.008	11.107	11.294	**11.008**	11.085	11.243	11.278	-
VFR200_20_10	11.069	**11.069**	11.240	11.193	11.276	11.352	11.322	-
VFR200_40_1	13.077	13.077	13.124	-	13.125	**13.049**	13.653	297.70
VFR200_40_2	13.134	13.134	13.222	-	13.072	**12.968**	13.552	9.99
VFR200_40_3	13.027	**13.027**	13.163	-	13.201	13.135	13.740	-
VFR200_40_4	12.974	13.197	**12.974**	-	13.140	13.102	13.709	-
VFR200_40_5	13.061	13.111	13.061	-	12.964	**12.899**	13.518	9.39
VFR200_40_6	12.927	**12.927**	13.220	-	13.056	13.026	13.672	-
VFR200_40_7	13.023	**13.023**	13.132	-	13.190	13.156	13.825	-
VFR200_40_8	13.033	13.188	**13.033**	-	13.136	13.067	13.782	-
VFR200_40_9	13.089	13.089	13.146	-	13.078	**13.024**	13.703	18.29
VFR200_40_10	13.042	**13.042**	13.049	-	13.166	13.049	13.865	-
VFR200_60_1	14.861	**14.861**	14.906	-	14.893	14.946	15.931	-
VFR200_60_2	14.881	**14.881**	15.134	-	14.900	14.965	15.771	-
VFR200_60_3	14.890	**14.890**	14.968	-	15.129	15.446	16.410	-
VFR200_60_4	15.042	15.103	15.042	-	**14.969**	15.162	15.989	-
VFR200_60_5	14.918	**14.918**	14.996	-	15.028	15.249	16.138	-
VFR200_60_6	15.006	15.020	15.006	-	**14.948**	14.951	15.939	-
VFR200_60_7	14.894	14.909	**14.894**	-	14.982	15.063	16.094	-
VFR200_60_8	14.925	14.956	14.925	-	**14.901**	15.094	15.831	-
VFR200_60_9	14.852	**14.852**	14.908	-	14.926	15.026	15.908	-
VFR200_60_10	14.867	**14.867**	14.909	-	14.908	15.122	15.777	-

Figure 11: Makespan minimization, all VFR instances with 100 and 200 jobs

instance	best-known	IGrms	VBIH	GmysBB	IGbob	IBS_bimin_walpha	IBS_bimin_gap	time_to_improve_bks (s)
VFR300_20_1	15.996	16.092	16.089	**15.996**	16.097	16.230	16.153	-
VFR300_20_2	16.129	16.794	**16.129**	16.409	16.491	16.641	16.470	-
VFR300_20_3	16.010	16.465	16.168	**16.010**	16.136	16.336	16.136	-
VFR300_20_4	16.052	16.115	16.307	**16.052**	16.167	16.307	**16.052**	-
VFR300_20_5	16.095	16.125	**16.095**	21.399	16.320	16.520	16.278	-
VFR300_20_6	16.021	16.293	16.244	**16.021**	16.086	16.271	**16.021**	-
VFR300_20_7	16.062	**16.062**	16.369	16.188	16.224	16.343	16.188	-
VFR300_20_8	16.228	**16.228**	16.324	16.287	16.372	16.498	16.407	-
VFR300_20_9	16.203	16.363	16.798	**16.203**	16.339	16.489	16.347	-
VFR300_20_10	16.298	**16.298**	16.483	16.780	16.794	17.063	16.780	-
VFR300_40_1	18.127	18.127	18.199	-	18.124	**18.009**	19.157	41.01
VFR300_40_2	18.253	18.253	18.348	-	18.360	**18.175**	19.199	20.50
VFR300_40_3	18.227	18.341	18.227	-	18.342	**18.201**	19.176	330.15
VFR300_40_4	18.276	18.276	18.343	-	18.240	**18.095**	19.146	10.71
VFR300_40_5	18.181	18.181	18.340	-	18.357	**18.127**	19.124	165.08
VFR300_40_6	18.320	18.320	18.396	-	18.336	**18.157**	19.323	10.27
VFR300_40_7	18.250	18.250	18.290	-	18.306	**18.171**	19.204	81.82
VFR300_40_8	18.261	18.283	18.261	-	18.209	**18.119**	19.148	10.18
VFR300_40_9	18.238	18.238	18.286	-	18.274	**18.073**	19.323	10.13
VFR300_40_10	18.226	18.226	18.373	-	18.240	**18.107**	19.023	40.76
VFR300_60_1	20.397	**20.397**	20.483	-	20.417	20.409	21.826	-
VFR300_60_2	20.290	20.290	20.328	-	20.249	**20.193**	21.667	469.89
VFR300_60_3	20.224	**20.224**	20.293	-	20.305	20.468	21.751	-
VFR300_60_4	20.200	20.244	**20.200**	-	20.311	20.309	21.538	-
VFR300_60_5	20.235	20.235	20.280	-	20.183	**20.176**	21.624	240.90
VFR300_60_6	20.156	**20.156**	20.358	-	20.272	20.244	21.600	-
VFR300_60_7	20.180	**20.180**	20.319	-	20.351	20.414	21.754	-
VFR300_60_8	20.285	**20.285**	20.405	-	20.314	20.490	21.959	-
VFR300_60_9	20.291	**20.291**	20.385	-	20.397	20.339	21.715	-
VFR300_60_10	20.249	20.326	**20.249**	-	20.355	20.407	22.067	-
VFR400_20_1	20.952	21.027	21.042	**20.952**	21.054	21.167	20.994	-
VFR400_20_2	21.346	21.524	21.428	**21.346**	21.429	21.578	21.375	-
VFR400_20_3	21.237	21.411	**21.237**	21.379	21.426	21.668	21.379	-
VFR400_20_4	21.125	21.426	21.528	**21.125**	21.247	21.482	21.167	-
VFR400_20_5	16.245	21.231	21.188	**16.245**	21.513	21.671	21.413	-
VFR400_20_6	21.075	21.497	21.599	**21.075**	21.183	21.274	21.168	-
VFR400_20_7	21.165	**21.165**	21.264	21.507	21.606	21.832	21.507	-
VFR400_20_8	21.198	21.580	21.293	**21.198**	21.258	21.441	21.202	-
VFR400_20_9	21.236	21.264	21.526	**21.236**	21.297	21.460	21.379	-
VFR400_20_10	21.301	**21.301**	21.411	21.456	21.527	21.618	21.456	-
VFR400_40_1	23.362	23.362	23.393	-	23.380	**23.085**	24.464	9.00
VFR400_40_2	23.467	23.504	23.467	-	23.293	**23.004**	24.523	1.50
VFR400_40_3	23.257	23.257	23.269	-	23.439	**23.200**	24.538	145.30
VFR400_40_4	23.213	23.405	23.213	-	23.255	**22.893**	24.762	2.70
VFR400_40_5	23.220	23.220	23.298	-	23.166	**22.901**	24.563	2.28
VFR400_40_6	23.141	23.141	23.415	-	23.305	**23.034**	24.456	71.52
VFR400_40_7	23.290	23.292	23.290	-	23.391	**23.157**	24.672	17.94
VFR400_40_8	23.364	23.364	23.424	-	23.243	**23.012**	24.207	2.23
VFR400_40_9	23.266	23.266	23.606	-	23.492	**23.265**	24.507	581.74
VFR400_40_10	23.380	23.457	23.380	-	23.578	**23.278**	24.597	72.10
VFR400_60_1	25.392	25.392	25.395	-	25.458	**25.214**	27.437	51.26
VFR400_60_2	25.618	25.618	25.707	-	25.533	**25.436**	27.475	210.50
VFR400_60_3	25.498	**25.498**	25.638	-	25.626	25.516	27.751	-
VFR400_60_4	25.590	25.590	25.669	-	25.683	**25.554**	27.361	835.99
VFR400_60_5	25.407	25.608	**25.407**	-	25.636	25.587	27.565	-
VFR400_60_6	25.415	25.615	25.415	-	25.331	**25.246**	26.753	207.96
VFR400_60_7	25.358	25.358	25.603	-	25.410	**25.173**	27.114	435.55
VFR400_60_8	25.372	25.372	25.673	-	25.616	**25.364**	27.632	826.64
VFR400_60_9	25.541	25.541	25.658	-	25.639	**25.474**	27.443	831.93
VFR400_60_10	25.549	25.622	25.549	-	25.626	**25.494**	27.624	832.37

Figure 12: Makespan minimization, all VFR instances with 300 and 400 jobs

instance	best-known	IGrms	VBIH	GmysBB	IGbob	IBS_bimin_walpha	IBS_bimin_gap	time_to_improve_bks (s)
VFR500_20_1	26.253	26.355	26.374	**26.253**	26.380	26.560	26.276	-
VFR500_20_2	26.043	**26.043**	26.359	26.555	26.620	26.846	26.575	-
VFR500_20_3	26.080	26.631	**26.080**	26.268	26.363	26.591	26.269	-
VFR500_20_4	25.994	26.357	26.759	**25.994**	26.086	26.307	25.996	-
VFR500_20_5	26.058	**26.058**	26.411	26.703	26.733	27.099	26.703	-
VFR500_20_6	26.325	26.729	26.409	**26.325**	26.428	26.555	**26.325**	-
VFR500_20_7	26.305	26.395	**26.305**	26.313	26.401	26.666	26.413	-
VFR500_20_8	26.217	26.401	26.430	**26.217**	26.305	26.401	26.327	-
VFR500_20_9	26.034	26.302	**26.034**	26.345	26.416	26.646	26.368	-
VFR500_20_10	26.345	26.410	26.641	26.345	26.052	26.292	**26.024**	0.10
VFR500_40_1	28.362	28.362	28.402	-	28.367	**28.110**	30.049	7.04
VFR500_40_2	28.526	28.585	28.526	-	28.574	**28.306**	29.752	14.09
VFR500_40_3	28.503	28.503	28.615	-	28.436	**28.234**	29.620	3.57
VFR500_40_4	28.374	28.374	28.579	-	28.562	**28.327**	29.824	224.73
VFR500_40_5	28.432	28.477	28.432	-	28.530	**28.276**	29.762	28.21
VFR500_40_6	28.543	28.543	28.553	-	28.347	**28.111**	29.535	1.72
VFR500_40_7	28.248	**28.248**	28.488	-	28.545	28.294	29.900	-
VFR500_40_8	28.486	28.486	28.640	-	28.481	**28.305**	30.130	28.13
VFR500_40_9	28.435	28.435	28.644	-	28.629	**28.404**	30.344	56.22
VFR500_40_10	28.613	28.640	28.613	-	28.592	**28.322**	29.697	7.14
VFR500_60_1	30.609	30.609	30.682	-	30.683	**30.263**	33.168	82.11
VFR500_60_2	30.828	30.828	30.852	-	30.623	**30.328**	32.593	40.86
VFR500_60_3	30.597	30.597	30.793	-	30.775	**30.494**	33.437	322.57
VFR500_60_4	30.763	30.823	30.763	-	30.817	**30.512**	32.712	88.27
VFR500_60_5	30.788	30.796	30.788	-	30.751	**30.371**	33.339	80.75
VFR500_60_6	30.700	30.700	30.826	-	30.836	**30.507**	33.488	161.27
VFR500_60_7	30.829	30.829	30.837	-	30.715	**30.406**	32.988	20.69
VFR500_60_8	30.733	30.733	30.805	-	30.751	**30.389**	32.901	81.03
VFR500_60_9	30.729	30.729	30.866	-	30.754	**30.393**	33.224	160.51
VFR500_60_10	30.664	30.785	30.664	-	30.833	**30.552**	33.044	1292.38
VFR600_20_1	31.303	31.359	31.372	**31.303**	31.361	31.523	**31.303**	-
VFR600_20_2	31.107	**31.107**	31.429	31.281	31.386	31.684	31.281	-
VFR600_20_3	31.372	**31.372**	31.487	31.374	31.414	31.670	31.374	-
VFR600_20_4	31.407	31.412	**31.407**	31.417	31.491	31.679	31.440	-
VFR600_20_5	31.323	31.480	31.696	**31.323**	31.396	31.657	31.476	-
VFR600_20_6	31.387	**31.387**	31.527	31.613	31.685	31.973	31.613	-
VFR600_20_7	31.461	31.668	31.523	**31.461**	31.527	31.885	**31.461**	-
VFR600_20_8	31.414	31.483	31.532	**31.414**	31.489	31.700	31.425	-
VFR600_20_9	31.107	31.465	**31.107**	31.473	31.528	31.931	31.477	-
VFR600_20_10	31.021	31.514	31.397	**31.021**	31.107	31.264	**31.021**	-
VFR600_40_1	33.618	33.618	33.683	-	33.598	**33.337**	35.264	5.24
VFR600_40_2	33.396	33.396	33.713	-	33.362	**33.155**	34.765	5.08
VFR600_40_3	33.356	**33.356**	33.584	-	33.591	33.404	35.364	-
VFR600_40_4	33.401	33.612	33.401	-	33.522	**33.231**	34.777	10.23
VFR600_40_5	33.477	33.477	33.626	-	33.307	**33.157**	34.809	2.53
VFR600_40_6	33.307	**33.307**	33.545	-	33.598	33.420	35.006	-
VFR600_40_7	33.298	33.552	**33.298**	-	33.502	33.320	35.204	-
VFR600_40_8	33.492	33.492	33.567	-	33.303	**33.052**	34.590	1.21
VFR600_40_9	33.282	33.282	33.473	-	33.445	**33.268**	35.346	81.09
VFR600_40_10	33.405	33.422	33.405	-	33.417	**33.257**	35.186	20.35
VFR600_60_1	35.863	35.863	35.976	-	35.867	**35.450**	38.094	29.20
VFR600_60_2	35.804	35.804	35.917	-	35.796	**35.450**	38.503	58.71
VFR600_60_3	35.791	35.791	36.000	-	35.958	**35.584**	38.501	234.12
VFR600_60_4	35.896	35.896	36.004	-	35.855	**35.454**	38.457	29.13
VFR600_60_5	35.883	35.883	35.943	-	35.888	**35.466**	38.104	58.15
VFR600_60_6	35.929	35.929	35.965	-	35.844	**35.370**	38.848	58.26
VFR600_60_7	35.828	35.828	35.894	-	36.009	**35.500**	38.517	117.99
VFR600_60_8	35.882	35.882	35.987	-	35.932	**35.368**	38.558	28.91
VFR600_60_9	35.784	35.784	35.943	-	35.906	**35.466**	39.184	118.13
VFR600_60_10	35.923	35.935	35.923	-	35.926	**35.366**	38.297	14.48

Figure 13: Makespan minimization, all VFR instances with 500 and 600 jobs

instance	best-known	IGrms	VBIH	GmysBB	IGbob	IBS_bimin_walpha	IBS_bimin_gap	time_to_improve_bks (s)
VFR700_20_1	36.285	36.354	36.388	**36.285**	36.355	36.759	36.294	-
VFR700_20_2	36.220	36.376	36.519	**36.220**	36.316	36.721	36.226	-
VFR700_20_3	36.303	**36.303**	36.380	36.419	36.509	36.829	36.534	-
VFR700_20_4	36.361	36.487	36.556	**36.361**	36.385	36.782	**36.361**	-
VFR700_20_5	36.379	**36.379**	36.645	36.496	36.578	36.867	36.496	-
VFR700_20_6	36.547	**36.547**	36.597	36.556	36.607	36.899	36.556	-
VFR700_20_7	36.492	36.610	**36.492**	36.540	36.608	36.901	36.540	-
VFR700_20_8	36.315	36.609	**36.315**	36.418	36.484	36.775	36.418	-
VFR700_20_9	36.212	36.481	36.386	**36.212**	36.296	36.576	36.215	-
VFR700_20_10	36.290	**36.290**	36.316	36.362	36.374	36.800	36.362	-
VFR700_40_1	38.720	38.720	38.767	-	38.707	**38.501**	40.141	6.94
VFR700_40_2	38.460	38.647	38.460	-	38.521	**38.246**	40.378	6.92
VFR700_40_3	38.499	38.499	38.597	-	38.371	**38.127**	40.543	3.36
VFR700_40_4	38.393	38.393	38.490	-	38.611	**38.386**	40.352	446.18
VFR700_40_5	38.440	38.593	38.440	-	38.449	**38.250**	39.973	6.87
VFR700_40_6	38.355	38.430	38.355	-	38.388	**38.203**	39.721	55.24
VFR700_40_7	38.336	38.336	38.817	-	38.312	**38.056**	40.051	13.58
VFR700_40_8	38.287	**38.287**	38.569	-	38.773	38.691	40.102	-
VFR700_40_9	38.712	38.766	38.712	-	38.527	**38.411**	39.661	1.62
VFR700_40_10	38.452	**38.452**	38.560	-	38.632	38.564	40.012	-
VFR700_60_1	41.125	41.125	41.192	-	41.107	**40.438**	44.623	39.30
VFR700_60_2	41.093	41.093	41.173	-	41.028	**40.588**	43.069	39.34
VFR700_60_3	41.008	41.008	41.120	-	41.038	**40.450**	43.595	20.02
VFR700_60_4	40.961	40.961	41.167	-	41.068	**40.472**	43.830	84.80
VFR700_60_5	41.070	41.070	41.159	-	41.057	**40.367**	44.522	40.20
VFR700_60_6	40.734	41.022	40.734	-	40.970	**40.396**	44.039	78.92
VFR700_60_7	40.994	40.994	41.305	-	40.629	**40.028**	43.977	9.84
VFR700_60_8	40.572	**40.572**	41.111	-	41.170	40.650	44.205	-
VFR700_60_9	41.121	41.121	41.186	-	40.980	**40.367**	44.611	10.14
VFR700_60_10	40.930	40.930	41.002	-	41.100	**40.480**	43.981	79.81
VFR800_20_1	41.413	41.477	41.479	**41.413**	41.501	41.764	41.415	-
VFR800_20_2	41.282	41.561	41.399	**41.282**	41.337	41.611	**41.282**	-
VFR800_20_3	41.319	41.337	41.426	**41.319**	41.389	41.577	**41.319**	-
VFR800_20_4	41.362	**41.362**	41.705	41.375	41.426	41.892	41.433	-
VFR800_20_5	41.426	**41.426**	41.961	41.626	41.705	41.939	41.626	-
VFR800_20_6	41.395	41.702	**41.395**	41.919	41.953	42.334	41.919	-
VFR800_20_7	41.342	41.959	41.435	**41.342**	41.379	41.666	41.352	-
VFR800_20_8	41.379	**41.379**	41.783	41.390	41.420	41.950	41.394	-
VFR800_20_9	41.429	**41.429**	41.568	41.697	41.783	42.033	41.697	-
VFR800_20_10	41.345	41.753	**41.345**	41.489	41.564	41.844	41.489	-
VFR800_40_1	43.456	43.456	43.466	-	43.446	**43.219**	45.354	18.08
VFR800_40_2	43.592	43.592	43.596	-	43.557	**43.324**	45.309	4.42
VFR800_40_3	43.483	43.483	43.743	-	43.465	**43.233**	45.066	4.40
VFR800_40_4	43.512	43.512	43.794	-	43.675	**43.430**	44.981	287.31
VFR800_40_5	43.557	43.557	43.638	-	43.657	**43.510**	45.670	1152.49
VFR800_40_6	43.484	43.635	43.484	-	43.575	**43.226**	45.450	8.90
VFR800_40_7	43.549	43.549	43.666	-	43.445	**43.307**	45.455	2.18
VFR800_40_8	43.458	43.458	43.643	-	43.572	**43.335**	44.615	143.53
VFR800_40_9	43.548	43.548	43.630	-	43.505	**43.387**	45.269	8.94
VFR800_40_10	43.497	43.497	43.575	-	43.588	**43.321**	45.091	36.06
VFR800_60_1	46.130	46.130	46.279	-	46.126	**45.402**	49.927	51.59
VFR800_60_2	46.004	46.004	46.258	-	46.176	**45.551**	49.520	51.86
VFR800_60_3	46.164	46.164	46.261	-	46.160	**45.494**	49.057	12.82
VFR800_60_4	46.108	46.108	46.164	-	46.148	**45.454**	49.133	25.92
VFR800_60_5	46.035	46.035	46.288	-	46.110	**45.382**	49.600	25.74
VFR800_60_6	46.061	46.101	46.061	-	46.151	**45.384**	50.056	25.74
VFR800_60_7	46.110	46.110	46.257	-	46.060	**45.423**	49.273	12.75
VFR800_60_8	45.986	45.986	46.279	-	46.207	**45.465**	49.679	54.92
VFR800_60_9	46.136	46.136	46.211	-	46.254	**45.591**	50.036	102.95
VFR800_60_10	46.226	46.226	46.232	-	46.026	**45.304**	50.099	12.84

Figure 14: Makespan minimization, all VFR instances with 700 and 800 jobs

instance	best-known	ALGirtct	shake-LS	IBS_alpha	time_to_improve_bks
TA20_5_0	14.033	**14.033**	**14.033**	**14.033**	-
TA20_5_1	15.151	**15.151**	**15.151**	**15.151**	-
TA20_5_2	13.301	**13.301**	**13.301**	**13.301**	-
TA20_5_3	15.447	**15.447**	**15.447**	**15.447**	-
TA20_5_4	13.529	**13.529**	**13.529**	**13.529**	-
TA20_5_5	13.123	**13.123**	**13.123**	**13.123**	-
TA20_5_6	13.548	**13.548**	**13.548**	**13.548**	-
TA20_5_7	13.948	**13.948**	**13.948**	**13.948**	-
TA20_5_8	14.295	**14.295**	**14.295**	**14.295**	-
TA20_5_9	12.943	**12.943**	**12.943**	**12.943**	-
TA20_10_0	20.911	**20.911**	**20.911**	**20.911**	-
TA20_10_1	22.440	**22.440**	**22.440**	**22.440**	-
TA20_10_2	19.833	**19.833**	**19.833**	**19.833**	-
TA20_10_3	18.710	**18.710**	**18.710**	**18.710**	-
TA20_10_4	18.641	**18.641**	**18.641**	**18.641**	-
TA20_10_5	19.245	**19.245**	**19.245**	**19.245**	-
TA20_10_6	18.363	**18.363**	**18.363**	**18.363**	-
TA20_10_7	20.241	**20.241**	**20.241**	**20.241**	-
TA20_10_8	20.330	**20.330**	**20.330**	**20.330**	-
TA20_10_9	21.320	**21.320**	**21.320**	**21.320**	-
TA20_20_0	33.623	**33.623**	**33.623**	**33.623**	-
TA20_20_1	31.587	**31.587**	**31.587**	**31.587**	-
TA20_20_2	33.920	**33.920**	**33.920**	**33.920**	-
TA20_20_3	31.661	**31.661**	**31.661**	**31.661**	-
TA20_20_4	34.557	**34.557**	**34.557**	**34.557**	-
TA20_20_5	32.564	**32.564**	**32.564**	**32.564**	-
TA20_20_6	32.922	**32.922**	**32.922**	**32.922**	-
TA20_20_7	32.412	**32.412**	**32.412**	**32.412**	-
TA20_20_8	33.600	**33.600**	**33.600**	**33.600**	-
TA20_20_9	32.262	**32.262**	**32.262**	**32.262**	-

Figure 15: Flowtime minimization, all Taillard instances TAI1 to TAI40

instance	best-known	ALGirtct	shake-LS	IBS_alpha	time_to_improve_bks
TA50_5_0	64.802	**64.802**	**64.802**	64.886	-
TA50_5_1	68.051	**68.051**	**68.051**	68.074	-
TA50_5_2	63.162	**63.162**	**63.162**	**63.162**	-
TA50_5_3	68.226	**68.226**	**68.226**	**68.226**	-
TA50_5_4	69.351	**69.351**	**69.351**	69.490	-
TA50_5_5	66.841	**66.841**	**66.841**	**66.841**	-
TA50_5_6	66.253	**66.253**	**66.253**	66.287	-
TA50_5_7	64.332	**64.332**	**64.332**	64.386	-
TA50_5_8	62.981	**62.981**	**62.981**	63.317	-
TA50_5_9	68.770	**68.770**	**68.770**	68.834	-
TA50_10_0	87.114	**87.114**	**87.114**	87.140	-
TA50_10_1	82.820	**82.820**	**82.820**	**82.820**	-
TA50_10_2	79.931	**79.931**	**79.931**	79.987	-
TA50_10_3	86.446	**86.446**	**86.446**	**86.446**	-
TA50_10_4	86.377	**86.377**	**86.377**	86.388	-
TA50_10_5	86.587	**86.587**	**86.587**	86.650	-
TA50_10_6	88.750	**88.750**	**88.750**	89.046	-
TA50_10_7	86.727	**86.727**	**86.727**	**86.727**	-
TA50_10_8	85.441	**85.441**	**85.441**	85.548	-
TA50_10_9	87.998	**87.998**	**87.998**	88.077	-
TA50_20_0	125.831	**125.831**	**125.831**	**125.831**	-
TA50_20_1	119.247	**119.247**	**119.247**	119.270	-
TA50_20_2	116.459	**116.459**	**116.459**	116.536	-
TA50_20_3	120.261	**120.261**	**120.261**	120.923	-
TA50_20_4	118.184	**118.184**	**118.184**	118.379	-
TA50_20_5	120.586	**120.586**	**120.586**	**120.586**	-
TA50_20_6	122.880	**122.880**	**122.880**	123.120	-
TA50_20_7	122.489	**122.489**	**122.489**	122.583	-
TA50_20_8	121.872	**121.872**	**121.872**	**121.872**	-
TA50_20_9	123.954	**123.954**	**123.954**	124.158	-
TA100_5_0	253.167	253.167	253.167	**252.687**	0.12
TA100_5_1	241.925	241.989	241.925	**241.593**	0.95
TA100_5_2	237.832	237.832	237.832	**237.289**	0.06
TA100_5_3	227.522	227.738	227.522	**227.345**	0.56
TA100_5_4	240.301	240.301	240.301	**240.138**	0.91
TA100_5_5	232.247	232.247	232.342	**231.973**	0.56
TA100_5_6	240.366	240.366	240.366	**240.111**	0.94
TA100_5_7	230.866	230.866	230.945	**230.290**	0.03
TA100_5_8	247.526	247.771	247.526	**247.362**	1.88
TA100_5_9	242.933	**242.933**	**242.933**	243.209	-
TA100_10_0	298.385	298.385	298.385	**296.990**	0.30
TA100_10_1	273.674	273.674	273.674	**273.014**	0.57
TA100_10_2	288.114	288.114	288.114	**287.420**	2.07
TA100_10_3	301.044	301.044	301.044	**299.467**	0.29
TA100_10_4	284.148	284.148	284.233	**283.260**	0.29
TA100_10_5	269.686	269.686	269.686	**268.324**	1.98
TA100_10_6	279.463	**279.463**	**279.463**	279.565	-
TA100_10_7	290.703	290.703	290.908	**289.334**	0.12
TA100_10_8	301.970	301.970	301.970	**301.005**	1.05
TA100_10_9	291.283	291.283	291.283	**290.038**	0.54

Figure 16: Flowtime minimization, all Taillard instances TAI41 to TAI80

instance	best-known	ALGirtct	shake-LS	IBS_alpha	time_to_improve_bks
TA100_20_0	365.463	365.463	365.463	**365.333**	43.03
TA100_20_1	372.001	372.001	372.449	**370.605**	2.43
TA100_20_2	370.027	370.027	370.027	**368.971**	2.47
TA100_20_3	372.393	372.393	372.393	**371.738**	43.56
TA100_20_4	368.915	368.915	368.915	**367.702**	4.80
TA100_20_5	370.908	370.908	370.908	**369.821**	1.19
TA100_20_6	373.408	373.408	373.408	**372.176**	9.85
TA100_20_7	384.525	384.525	384.525	**382.766**	4.99
TA100_20_8	374.423	374.423	374.423	**372.817**	2.42
TA100_20_9	379.296	379.296	379.296	**378.566**	20.59
TA200_10_0	1.041.023	1.042.452	1.041.023	**1.035.999**	1.90
TA200_10_1	1.028.775	1.028.775	1.028.828	**1.024.752**	1.82
TA200_10_2	1.042.357	1.043.631	1.042.357	**1.038.814**	7.20
TA200_10_3	1.023.188	1.023.188	1.025.564	**1.019.215**	1.85
TA200_10_4	1.028.506	1.028.506	1.028.963	**1.024.759**	1.76
TA200_10_5	998.340	998.686	998.340	**994.661**	0.96
TA200_10_6	1.042.570	1.042.570	1.042.570	**1.038.357**	0.43
TA200_10_7	1.035.915	1.035.945	1.035.915	**1.033.303**	1.94
TA200_10_8	1.015.280	1.015.560	1.015.280	**1.011.878**	1.86
TA200_10_9	1.021.633	1.021.633	1.021.865	**1.017.386**	0.91
TA200_20_0	1.219.341	1.221.768	1.219.341	**1.205.091**	0.63
TA200_20_1	1.231.880	1.231.880	1.233.161	**1.224.536**	4.28
TA200_20_2	1.254.822	1.254.822	1.259.605	**1.248.190**	4.22
TA200_20_3	1.226.654	1.226.654	1.228.027	**1.217.648**	8.63
TA200_20_4	1.215.411	1.215.411	1.215.854	**1.203.033**	1.06
TA200_20_5	1.218.757	1.219.698	1.218.757	**1.207.770**	2.12
TA200_20_6	1.234.330	1.237.014	1.234.330	**1.224.492**	1.06
TA200_20_7	1.233.257	1.233.257	1.240.105	**1.222.559**	1.05
TA200_20_8	1.220.058	1.222.431	1.220.058	**1.212.081**	8.53
TA200_20_9	1.234.864	1.234.864	1.235.113	**1.229.039**	4.33
TA500_20_0	6.558.109	6.562.522	6.558.109	**6.529.840**	50.11
TA500_20_1	6.678.713	6.678.713	6.679.339	**6.642.805**	24.67
TA500_20_2	6.624.644	6.632.299	6.624.644	**6.585.806**	12.42
TA500_20_3	6.633.622	6.633.622	6.646.006	**6.601.961**	24.71
TA500_20_4	6.587.110	6.609.322	6.587.110	**6.556.492**	24.96
TA500_20_5	6.602.685	6.605.982	6.602.685	**6.563.223**	6.12
TA500_20_6	6.576.047	6.576.412	6.576.047	**6.530.456**	12.44
TA500_20_7	6.628.915	6.628.915	6.629.065	**6.594.903**	25.15
TA500_20_8	6.569.013	6.569.013	6.587.638	**6.532.742**	50.58
TA500_20_9	6.614.629	6.614.629	6.623.849	**6.589.096**	49.59

Figure 17: Flowtime minimization, all Taillard instances TAI81 to TAI120