European Journal of Operational Research 284 (2020) 814–833

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete Optimization

A computationally efficient Branch-and-Bound algorithm for the permutation flow-shop scheduling problem

Jan Gmys[a], Mohand Mezmaz[a], Nouredine Melab[b], Daniel Tuyttens[a,*]

[a] Mathematics and Operational Research Department, University of Mons, Belgium
[b] University of Lille, Inria Lille - Nord Europe, CNRS/CRIStAL, France

ARTICLE INFO

Article history:
Received 14 June 2019
Accepted 16 January 2020
Available online 24 January 2020

Keywords:
Branch-and-Bound
Flowshop
Makespan
Parallel computing

ABSTRACT

In this work we propose an efficient branch-and-bound (B&B) algorithm for the permutation flow-shop problem (PFSP) with makespan objective. We present a new node decomposition scheme that combines dynamic branching and lower bound refinement strategies in a computationally efficient way. To alleviate the computational burden of the two-machine bound used in the refinement stage, we propose an online learning-inspired mechanism to predict promising couples of bottleneck machines. The algorithm offers multiple choices for branching and bounding operators and can explore the search tree either sequentially or in parallel on multi-core CPUs. In order to empirically determine the most efficient combination of these components, a series of computational experiments with 600 benchmark instances is performed. A main insight is that the problem size, as well as interactions between branching and bounding operators substantially modify the trade-off between the computational requirements of a lower bound and the achieved tree size reduction. Moreover, we demonstrate that parallel tree search is a key ingredient for the resolution of large problem instances, as strong super-linear speedups can be observed. An overall evaluation using two well-known benchmarks indicates that the proposed approach is superior to previously published B&B algorithms. For the first benchmark we report the exact resolution – within less than 20 minutes – of two instances defined by 500 jobs and 20 machines that remained open for more than 25 years, and for the second a total of 89 improved best-known upper bounds, including proofs of optimality for 74 of them.

1. Introduction

In this paper, we consider the m-machine permutation flow-shop scheduling problem (PFSP) with makespan criterion. The problem consists in scheduling n jobs on machines $M_1, \ldots, M_m$ in that order and without interruption. Machines can process at most one job at a time and the processing order of jobs is the same on all machines. The goal is to find a permutation (a processing order) that minimizes the termination date of the last job on the last machine, called makespan. The processing time of job $j = 1, 2, \ldots, n$ on machine M_k is given by p_{jk}. In the case $m = 2$ the problem can be solved in $\mathcal{O}(n \log n)$ steps by sorting the jobs according to Johnson's rule (Johnson, 1954). For $m \geq 3$ the problem is shown to be $\mathcal{NP}$-hard (Garey, Johnson, & Sethi, 1976).

Due to the $\mathcal{NP}$-hardness of the problem, research efforts over the last two decades have been focused on approximate methods to find high-quality solutions within a reasonably short amount of computation time (Fernandez-Viagas, Ruiz, & Framinan, 2017; Framinan, Gupta, & Leisten, 2004; Hejazi & Saghafian, 2005). In contrast, exact methods allow to find optimal solution(s) with a proof of optimality, but their execution time is unpredictable and exponential in the worst-case.

Branch-and-Bound (B&B) is the most frequently used exact method to solve combinatorial optimization problems like the PFSP. The algorithm recursively decomposes the initial problem by dynamically constructing and exploring a search-tree, whose root node represents the initial problem, leaf nodes are possible solutions and internal nodes are subproblems of the initial problem. This is done using four operators: branching, bounding, selection and pruning. The branching operator divides the initial problem into smaller disjoint subproblems and a bounding function computes lower bounds on the optimal cost of a subproblem. The pruning operator eliminates subproblems whose lower bound exceeds the cost of the best solution found so far (upper bound on the optimal makespan). The tree-traversal is guided by the selection operator which returns the next subproblem to be processed according to a search strategy (e.g. depth-first search).

* Corresponding author.
E-mail address: daniel.tuyttens@umons.ac.be (D. Tuyttens).

https://doi.org/10.1016/j.ejor.2020.01.039
0377-2217/

In this paper the focus is put on three performance-critical components of the algorithm: the lower bound (LB), the branching rule and the use of parallel tree exploration. Although they can be separated on a conceptual level, the main objective of this article is to reveal interactions between these components and to investigate their efficiency according to the size (number of jobs and machines) of problem instances.

Branching rules are rarely studied in the context of B&B algorithms for permutation problems, although for the PFSP a variety of branching rules can be found in the literature. Classical approaches are *forward branching* and *backward branching*, which consist in assigning unscheduled jobs to the first (resp. last) free position in partial schedules. These two branching rules can be combined in a predefined, *static* manner or the algorithm can decide *dynamically* which type of branching is applied to a subproblem. Despite their strong impact on the size of the explored search tree, only very few works propose experimental comparisons of different branching rules (Potts, 1980; Ritt, 2016). In particular, we are not aware of any work that considers the impact of branching rules on the efficiency of lower bounding functions (and vice-versa). This paper aims at filling this lack of experimental evaluation by comparing the efficiency of five different branching rules *in combination* with two different lower bounds. The reported experimental results show that dynamic branching is superior to static rules and that the choice of the branching rule has a strong impact on the relative efficiency of lower bounds.

As part of a general bounding scheme for the PFSP, Lageweg, Lenstra, and Kan (1978) have proposed the so-called two-machine lower bound (LB2), which is the strongest known (polynomial) lower bound for the minimization of makespan. Since its discovery, LB2 has replaced the weaker, but computationally less expensive one-machine lower bound (LB1) in the majority of B&B algorithms for the PFSP. A detailed description of both bounds is provided in Section 3.4 and a complete review of lower bounds can be found in Ladhari and Haouari (2005).

While the strongest lower bound has the best worst-case performance, the usefulness of a LB for the average case depends on the trade-off between its sharpness and its computational requirements. Indeed, a stronger bound eliminates larger parts of the search tree, but if its computation becomes too expensive it may be advantageous to explore a larger number of nodes using a weaker bound with lower computational requirements. The results of an experimental evaluation of this trade-off have been reported in Lageweg et al. (1978), showing the superiority of LB2. However, this computational experience considers only forward branching (as it was published before the discovery of other branching rules) and relatively small problem instances (because of hardware constraints).

In this article, we re-evaluate the efficiency of different branching rule/lower bound combinations for PFSP instances with a total of 600 benchmark instances defined by up to $n = 800$ jobs. The strongest variant of LB2 is obtained by solving two-machine problems for all $\frac{m(m-1)}{2}$ machine-pairs. We propose an online learning approach which aims at evaluating LB2 only for a subset of m machine-pairs, which is predicted to maximize the value of LB2. Experimental results show that this adaptive variant of LB2 provides a better trade-off between sharpness and computational complexity than existing variants.

The design and implementation of efficient parallel B&B algorithms is challenging, mainly because of the highly irregular and unpredictable structure of the search tree. Therefore, research on parallel B&B for the PFSP has mainly focused on algorithmic challenges, such as load balancing strategies to handle this irregularity, the design of efficient data structures to manage large search trees and the efficient exploitation of all levels of parallelism in heterogeneous supercomputers.

Aiming mainly at increasing processing speed, experiments with parallel tree exploration algorithms are usually initialized with known optimal solutions. This ensures that the algorithm explores exactly the set $\{x|LB(x) < C^{\star}_{\max}\}$ of subproblems x having a LB smaller than the optimal makespan $C^{\star}_{\max}$, called *critical nodes*. Consequently, there is very little information available concerning the performance of parallel B&B in the case where the algorithm also explores non-critical nodes. Also, one LB may be more suitable for parallel processing than another. Therefore, without putting this aspect in the center of our investigation, a parallel multi-core implementation of the proposed algorithm is considered in addition to the sequential one. The results show that parallel tree exploration is a key ingredient for the resolution of some large instances as superlinear speedups can be observed, even when using a moderate number of cores, available in commodity multi-core processors.

Extensive computational experiments are performed, using the well-known benchmark proposed by Taillard (1993) and the more recent VRF benchmark proposed by Vallada, Ruiz, and Framinan (2015). Reported results reveal interactions between different B&B components and instance-dependent behaviors, leading to recommendations regarding the efficient design of exact algorithms for the PFSP.

Taillard's benchmark is well-established and widely used for the experimental evaluation of PFSP solvers, including exact B&B methods – optimal solutions for most of the 120 instances are known. We use these instances to compare the performance of the proposed algorithm with approaches from the literature. Experimental results demonstrate that the proposed algorithm is faster and solves more instances to optimality, including two previously unsolved instances (`Ta112` and `Ta116`) defined by $n = 500$ jobs and $m = 20$ machines. Moreover, the proposed approach solves to optimality one very hard instance defined by 50 jobs and 20 machines (`Ta56`), requiring 33 hours of computation on a shared-memory system with 16 CPU cores – instead of 25 days on a cluster of 328 CPUs, as in the first resolution of this instance by Mezmaz, Melab, and Talbi (2007).

The VRF benchmark consists of 240 small instances and 240 large instances with up to 800 jobs and 60 machines and is gaining popularity among researchers over the last years (Dubois-Lacoste, Pagnozzi, & Stützle, 2017; Fernandez-Viagas et al., 2017; Liu, Jin, & Price, 2017; Rossi, Nagano, & Neto, 2016). To the best of our knowledge, the present work is the first to use this benchmark for the evaluation of an exact method. The best-known upper bounds for these instances, reported in Vallada et al. (2015), have been found by state-of-the-art approximate methods – the proofs of optimality and exact optimal solutions we provide for a subset of the VRF instances can help to assess the quality of approximate methods more accurately. In summary, for the 240 `VRF_small` instances, 162 certificates of optimality are produced and 28 improved upper bounds are reported. For the 240 `VRF_large` instances, a total of 61 best-known solutions are improved and 46 of them are proven optimal, showing an optimality gap of at most 0.72% for the previous best-known solutions.

The remainder of this paper is organized as follows. Section 2 provides a review of literature on B&B algorithms for the PFSP with makespan objective. In Section 3, we present our node decomposition scheme integrating different branching rules and lower bounds. Finally, in Section 4 we report results of computational experiments and conclusions are drawn in Section 5.

2. B&B algorithms for PFSP

This section provides an overview of the most important works on B&B algorithms for the PFSP, following a roughly chronological

Table 1
Notations.

σ_1	initial (starting) partial sequence of n_1 jobs
σ_2	final (ending) partial sequence of n_2 jobs
(σ_1, σ_2)	subproblem with $n_1 + n_2$ scheduled jobs
n	number of jobs
J	set of $s = n - n_1 - n_2$ unscheduled jobs
$C^k_{\sigma_1}$	time between start of σ_1 on M_1 and completion of σ_1 on M_k
$C^k_{\sigma_2}$	time between start of σ_2 on M_k and completion of σ_2 on M_m
m	number of machines
k	machine-index
j	job-index
p_{jk}	processing time of job j on machine M_k

order. In Section 2.1 we review early works (1965–1975), establishing some general principles and basic concepts used in subsequent approaches. The focus in Section 2.2 is put on the development of stronger lower bounds and in Section 2.3 we provide a review of branching rules for the PFSP. Section 2.4 gives a brief overview of parallel approaches and related challenges. Finally, Section 2.5 focuses on three different B&B algorithms which are the most efficient ones currently available – according to experimental results with Taillard's benchmark. The notations used throughout this article are summarized in Table 1.

2.1. Foundations

The first B&B algorithms for the exact minimization of makespan in 3-machine PFSP were proposed by Ignall and Schrage (1965) and Lomnicki (1965). Both are based on a *forward branching* rule: starting from the initial problem, represented by an empty partial schedule σ_1 and a set of unscheduled jobs $J = \{1, \dots, n\}$, subproblems are recursively decomposed into smaller, disjoint subproblems $\{(\sigma_1 j), j \in J\}$ by appending unscheduled jobs to the partial schedule. For each generated subproblem the algorithm computes a lower bound on its optimal cost, and subproblems that cannot lead to an improvement of the best solution found so far are eliminated from the search. For $m = 3$, both Ignall and Schrage (1965) and Lomnicki (1965) propose a machine-based lower bound (LB1, described in Section 3.4). The main difference between both algorithms is that the former uses best-first search, while the latter performs depth-first search (DFS), like practically all subsequently developed B&B algorithms. The work of Brown and Lomnicki (1966) adapts the approach to m-machine problems and to an "electronic computer". The following year, McMahon and Burton (1967) proposed a job-based lower bound to be used in combination with LB1. As noted in the comparative study of flow-shop algorithms by Baker (1975), the principal difference among B&B algorithms in the literature lies in the calculation of lower bounds. During this first decade a second promising line of research was the use of dominance rules (Szwarc, 1973), which provide conditions under which certain potential solutions can be ignored. However, these rules require some conditions to hold on all machines, so their application is limited, especially if m is large. A survey of works during the first decade of research on the PFSP can be found in Potts and Strusevich (2009).

2.2. Two-machine bounds

Nabeshima (1967) developed the first lower bound which takes into account idle time by applying Johnson's rule (Johnson, 1954) to unscheduled jobs on adjacent machines M_k and M_{k+1}. This approach was generalized by Lageweg et al. (1978) in the bounding framework for PFSP makespan minimization, which introduces the two-machine bound (LB2, described in detail in Section 3.4) in its most general form. LB2 relies on the exact resolution of two-machine problems obtained by relaxing capacity constraints on all machines, with the exception of a pair of machines $(u, v)_{1 \leq u < v \leq m}$, called *bottleneck machines*. Taking the maximum over the complete set of $\frac{m(m-1)}{2}$ machine-pairs yields a very sharp bound which dominates all other known LBs (of polynomial complexity). In order to alleviate the computational burden of LB2 (quadratic in m), Lageweg et al. (1978) propose to use only a subset of m machine-pairs and suggest that "it may be worth investigating in more detail for which set of machine pairs this bound should be calculated". The computational experiments conducted in Lageweg et al. (1978) compare B&B algorithms using forward branching with different combinations of lower bounds and elimination rules. The results show that LB2 (with a reduced set of machine-pairs) provides the best trade-off between sharpness and computational requirements. Since the discovery of LB2, most B&B algorithms for makespan minimization in the literature use this bound in some form, with different choices for machine-pairs or in combination with LB1 (Carlier & Rebaï, 1996; Cheng, Kise, & Matsumoto, 1997; Cheng, Kise, Steiner, & Stephenson, 2003; Companys & Mateo, 2007; Ladhari & Haouari, 2005; Lemesre, Dhaenens, & Talbi, 2007; Potts, 1980; Ritt, 2016).

2.3. Branching rules

The B&B algorithm proposed by Potts (1980) was the first to use a branching rule based on a combination of forward and backward branching. This idea aims at exploiting the property that the inverse problem (defined by processing times $\tilde{p}_{j,k}$, where $\tilde{p}_{j,k} = p_{j,m-k+1}$, $k = 1, \dots, m$) is sometimes easier to solve and admits the same optimal makespan as the original problem (Brown & Lomnicki, 1966; McMahon & Burton, 1967). Moreover, solving the inverse problem with a forward branching rule is equivalent to a method that solves the initial problem with backward branching. In the algorithm developed by Potts (1980), internal nodes are represented in the form of *initial* and *final* partial sequences (σ_1, σ_2), and each node is either obtained by *forward* branching, appending unscheduled jobs $j \in J$ to the head $(\sigma_1 j, \sigma_2)$, or *backward* branching, appending jobs to the tail $(\sigma_1, j\sigma_2)$. At the algorithm's first visit of a level, both sets $\{(\sigma_1 j, \sigma_2), j \in J\}$ and $\{(\sigma_1, j\sigma_2), j \in J\}$ are evaluated and the set where the minimum lower bound is realized less often is retained. The algorithm remembers the choice of the branching type and applies it to all subsequent decompositions of nodes at this level. Using LB2 (with a subset of m machine-pairs) – recognized as being "the most efficient" (Potts, 1980) – the computational experience shows that this so-called *adaptive branching rule* allows to solve some of the larger instances (a few tens of thousand nodes) more than twice as fast.

Researchers have generally recognized the usefulness of constructing solutions from both ends, as most subsequently published B&B algorithms represent subproblems in the form (σ_1, σ_2) (Carlier & Rebaï, 1996; Chakroun, Melab, Mezmaz, & Tuyttens, 2013; Drozdowski, Marciniak, Pawlak, & Plaza, 2011; Ladhari & Haouari, 2005; Lemesre et al., 2007; Ritt, 2016). We refer to the two sequences σ_1 and σ_2 as "initial/final" subsequences, following the original article (Potts, 1980), but the terms "prefix/suffix" (Ritt, 2016), "starting/finishing" (Drozdowski et al., 2011) or "top/bottom" (Ladhari & Haouari, 2005) are used synonymously. Potts' adaptive branching rule is rarely used in its original form: different variants of this branching scheme can be found in the literature, and there is a lack of experimental evaluation regarding the efficiency of the different approaches.

For instance, Carlier and Rebaï (1996) proposed two B&B algorithms: the first one uses a *dynamic* branching strategy where both sets of descendant nodes are generated and the one with lower cardinality (after eliminating unpromising nodes) is retained; the second one uses a *static* scheme, where the two branching types are applied *alternatively*, starting with a type 1 (forward)

branching. As both algorithms also use different lower bounds, the results do not allow to compare the efficiency of both branching rules.

Ladhari and Haouari (2005) also use the alternate branching scheme, in combination with a composite lower bound. In Lemesre et al. (2007), in the context of a B&B developed for the bi-objective PFSP, "when a node is developed, two sets of solutions are constructed [...]" and the authors "choose to explore the set with the smallest number of promising partial solutions". In other words, the *branching factor* (average number of children per node) is locally minimized.

The term "dynamic branching" seems to be first used in Ritt (2016) to distinguish branching rules which generate two alternative children sets at each node decomposition from static approaches, where the branching type for each level is predefined. The latter work compares alternate and dynamic branching strategies in combination with a composite LB1/LB2 bound and indicates the superiority of the dynamic approach, especially for instances defined by a large number of jobs. However, we are not aware of any work that investigates the interaction of branching rules with different lower bounds, nor the use of different criteria for dynamic branching decisions.

2.4. Parallelism

The PFSP has been frequently used as a test-case for parallel B&B algorithms, as the huge amount of generated nodes and the highly irregular structure of the search tree raise multiple challenges in terms of design and implementation on increasingly complex parallel architectures, e. g. grid computing (Bendjoudi, Melab, & Talbi, 2012; Drozdowski et al., 2011; Mezmaz et al., 2007), multi-core CPUs (Gmys, Leroy, Mezmaz, Melab, & Tuyttens, 2016a; Mezmaz, Leroy, Melab, & Tuyttens, 2014a), GPUs and many-core devices (Chakroun et al., 2013; Gmys, Mezmaz, Melab, & Tuyttens, 2016b; Melab, Gmys, Mezmaz, & Tuyttens, 2018), clusters of GPUs (Vu & Derbel, 2016) or FPGAs (Daouri, Escobar, Xin Chang, & Valderrama, 2015).

As this line of research is primarily focused on algorithmic challenges like the design of efficient data structures and load balancing mechanisms, details on the branching strategy or even on the bounding function are often spared out. However, the default choice seems to be LB2 with a complete evaluation of all machine-pairs, in combination with dynamic branching (Chakroun et al., 2013; Drozdowski et al., 2011; Gmys et al., 2016b). There are several sources of parallelism that can be exploited in a nested way: the parallel exploration of the search tree, the parallel evaluation of generated children nodes and the parallelization/vectorization of the LB computation. It is well-known that the parallel exploration of the search tree may lead to deceleration, detrimental or super-linear speedup anomalies (de Bruin, Kindervater, & Trienekens, 1995; Li & Wah, 1986), as nodes are visited in an order that differs from the sequential search. In order to analyze the scalability of these approaches in the absence such anomalies, the experimental protocol usually ensures that sequential and parallel versions perform an equal amount of work (e. g. by initializing the search at the optimal solution).

2.5. Focus on three efficient B&B algorithms

The most recent and seemingly most successful sequential algorithms have been proposed by Ladhari and Haouari (2005), Companys and Mateo (2007) and Ritt (2016). All three have been experimentally evaluated using Taillard's benchmark instances and the reported results will be used as references for comparison in Section 4.5.

Ladhari and Haouari (2005) have proposed an innovative lower bounding strategy based on the combined use of LBs with different computational requirements and sharpness. The LB computation depends on the depth $d(N)$ of the evaluated node N: close to the root ($0 \leq d(N) \leq \lfloor n/3 \rfloor$), LB1 is used; in the middle ($\lfloor n/3 \rfloor < d(N) \leq \lfloor 2n/3 \rfloor$) the full two-machine bound LB2 is computed only if N is not eliminated by LB1 and close to the leaves ($d(N) > \lfloor 2n/3 \rfloor$) an even stronger (but non-polynomial) bound is computed if N is not eliminated by LB2. The originality of this approach lies in the layered combination of LBs, computing a stronger LB only if a weaker one does not allow to prune a node. A static, alternating branching rule is used and the initial solution is obtained by the Nawaz-Enscore-Ham (NEH) heuristic (Nawaz, Enscore, & Ham, 1983), followed by a truncated B&B exploration. Within a time-limit of 3 hours, on a Pentium IV CPU (1.8 gigahertz) and written in C, the algorithm solves 62 out of the Taillard's 120 benchmark instances (62/70 of the instances defined by $m \leq 10$ machines).

The LOMPEN (LOMnicki PENdular) (Companys & Mateo, 2007) algorithm runs two B&B algorithms that simultaneously solve the original and the inverse instance, exchanging information in order to apply dominance rules and strengthen lower bounds. A forward branching scheme and the full two-machine bound is used by both algorithms. Using a Pentium IV CPU (1.8 gigahertz) the algorithm solves all 70 Taillard instances defined by $m \leq 10$ machines, but requires $> 3h$ processing time for 16 out of these instances. LOMPEN was implemented in C++. The initial solution is obtained by a NEH+ heuristic, which improves the NEH solution by applying a local search procedure.

Ritt (2016) compares the performance of a two B&B algorithms based on dynamic branching with the approaches of Ladhari and Haouari (2005) and Companys and Mateo (2007). For the lower bound, Ritt (2016) follows Ladhari and Haouari (2005) and applies the (full) two-machine bound LB2 only if the current node could not be fathomed by LB1, and has depth at least equal to $\lfloor n/3 \rfloor$. The dynamic branching approach selects the direction which generates the least number of subproblems. The two proposed variants use different search strategies: depth-first (DFS) and cyclic-best-first search (CBFS). The experimental results show that the DFS and CBFS algorithms are capable of solving 69 out of the 70 Taillard instances defined by $m \leq 10$ machines, as well as 3 instances defined by $n = m = 20$ jobs and machines (within a time-limit of 1 hour, using an AMD FX-8150 CPU running at a clock rate of 3.6 gigahertz and C++ as programming language).

3. The proposed Branch-and-Bound algorithm

This section describes our B&B algorithm for the PFSP. First, in Section 3.1 we introduce the (sequential and parallel) search strategy and the associated data structure used for the storage and management of the search space. Section 3.2 describes the proposed node decomposition scheme, based on dynamic branching and a bound refinement approach. Section 3.3 describes the different dynamic branching strategies that can be used in this scheme and Section 3.4 provides details on the different implemented LBs, including the proposed online learning-inspired two-machine LB.

3.1. Search strategy

The search tree is explored in Depth-First Least-Lower-Bound order, i.e. newly branched nodes are sorted in non-decreasing order according to the values of lower bounds. If there are nodes of the same depth with equal lower bounds, the one which produces the least sum of idle time on all machines is chosen first (the incremental idle time added by a job can be easily computed

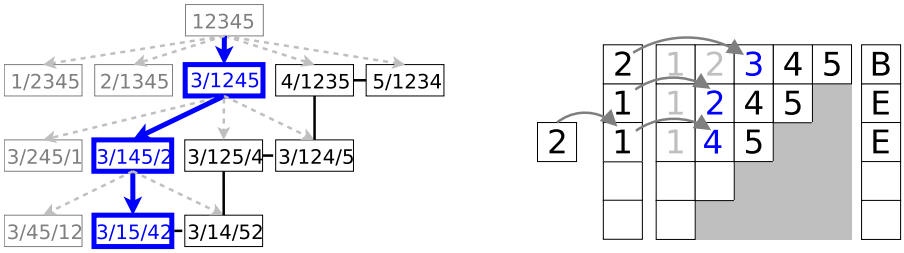


Fig. 1. Tree and IVM-based representation of the search state, solving a permutation problem of size 5.

in the LB evaluation). This search strategy corresponds to the recommendation in Potts (1980). Initial experiments have shown the tie-breaking mechanism to be effective, as sibling nodes with equal lower bounds occur frequently. The choice of depth-first search (DFS) is natural, given the ability of DFS to quickly discover new solutions and the memory requirements induced by the huge number of subproblems generated during the search[1].

For the storage and management of the pool of subproblems we use the Integer-Vector-Matrix (IVM) data structure, proposed by Mezmaz, Leroy, Melab, and Tuyttens (2014b). The IVM data structure is dedicated to permutation problems and provides a memory-efficient alternative to stacks or priority queues, which are commonly used for DFS. The working of the IVM data structure is best introduced with an example. Fig. 1 illustrates a pool of subproblems that could be obtained when solving a permutation problem of size $n = 5$ with a DFS-B&B algorithm that uses forward and backward branching. On the left-hand side, Fig. 1 shows a tree-based representation of this pool. The parent-child relationship between subproblems is designated by dashed gray arrows. The jobs before the first "/" symbol form the initial sequence σ_1, the ones behind the second "/" symbol form the final sequence σ_2 and jobs between the two "/" symbols represent the set of unscheduled jobs J in arbitrary order. The path to the current subproblem (3/15/42) is highlighted by thick blue arrows. Black nodes are generated but not yet explored – a stack-based algorithm stores them in the order indicated by solid black lines.

On the right-hand side, IVM indicates the next subproblem to be solved. The integer I of IVM gives the level of this subproblem – using 0-based counting, i.e. at level 0 a single job is scheduled. In this example, the level of the next subproblem is 2. The vector V contains, for each level up to I, the position of the selected subproblem among its sibling nodes in the tree. In the example, jobs 3, 2 and 4 have been scheduled at levels 0, 1 and 2, respectively. The matrix M contains the jobs to be scheduled at each level: all the n jobs (for a problem with n jobs) for the first row, the $n-1$ remaining jobs for the second row, and so on. The data structure is completed with a binary array of length n that indicates the branching type for each, i.e. whether a selected job is scheduled at the beginning or at the end. In the example, job 3 is scheduled at the beginning, jobs 2 and 4 are scheduled at the end. Thus, the IVM structure indicates that 3/15/42 is the next subproblem to be decomposed. To use the IVM data structure in a DFS-B&B algorithm, the operators acting on the data structure are revisited as follows:

- To branch a selected subproblem, the remaining unscheduled jobs are copied to the next row and the current level I is incremented. If a dynamic branching approach is used, the direction-vector is set according to the choice between the two children sets.
- To prune a subproblem, the corresponding entry in the matrix should be ignored by the selection operator. To flag a subproblem as "pruned" the corresponding entry in the matrix is multiplied by -1 (in the example, to eliminate 3/15/42, $M[2, 1]$ is set to -4). With this convention, branching actually consists in copying absolute values of entries to the next row, i.e. the flags of remaining jobs are removed as they are copied to the next row.
- To select the next subproblem, the values of I and V are modified such that they point to the deepest leftmost non-negative entry in M: the vector V is incremented at position I until a non-pruned entry is found or the end of the row is reached. If the end of the row is reached (i. e. $V[I] = n - I$), then the algorithm "backtracks" to the previous level by decrementing I and again incrementing V.

Note that the Depth-First Least-Lower-Bound search order is obtained by sorting the jobs in each row according to the corresponding lower bounds/idle times – for the sake of simplicity this is omitted in the example.

3.1.1. Parallel tree exploration

Throughout the depth-first exploration, the vector V behaves like a counter. In the example of Fig. 1, V successively takes the values 00000, 00010, 00100, ..., 43200, 43210 (skipping some values due to pruning). These 120 values correspond to the numbering of the 5! solutions using the mixed-radix factorial number system (Knuth, 1997), in which the weight of the kth position is equal to $k!$ and the digits allowed for the kth position are $0, 1, \ldots, k$. Each integer $0 \leq a < n!$ can be uniquely written as a n-digit factorial number – this allows to interpret the DFS tree search as an exploration, from left to right, of the integer interval $[0, n![$. Moreover, by comparing the value of the position-vector V_A to an end-vector V_B, the exploration can be restricted to arbitrary intervals $[A, B[\subset [0, n![$.

Based on this observation, the algorithm extends quite naturally to a parallel tree exploration algorithm. The interval $[0, n![$ can be cut into T pieces and have T threads explore distinct subintervals $[A_i, B_i[\subset [0, n![$, $i = 1, \ldots, T$. In order to deal with the highly irregular and unpredictable nature of the search tree (which translates to an irregular distribution of work in the subintervals) a work stealing approach is used. When a thread i finishes the exploration of its interval $[A_i, B_i[$ (i. e. when $A_i = B_i$) it chooses a "victim" thread j (uniformly at random) and emits a work requests. Between node decompositions, active B&B-threads poll for incoming requests. When the victim thread j detects the work-request, it sends the right half of its interval $[\frac{A_j+B_j}{2}, B_j[$ to the "thief" and continues the exploration of $[A_j, \frac{A_j+B_j}{2}[$. If the victim's interval $[A_j, B_j[$ is empty or smaller than a predefined threshold, thread j answers by sending a "fail" message and the thief chooses a new victim thread. The synchronization of thief and victim-threads is achieved

[1] For example, during the resolution of Ta56, a problem instance defined by 50 jobs and 20 machines, 332×10^9 subproblems are decomposed, so there exists at least one level with more than 6.6×10^9 open subproblems. Exploring this tree in breadth-first order and storing each subproblem as a sequence of 50 32-bit integers, the storage of these subproblems would require $6.6 \times 10^9 \times 50 \times 4$ bytes $= 1.3$ terabytes of memory.

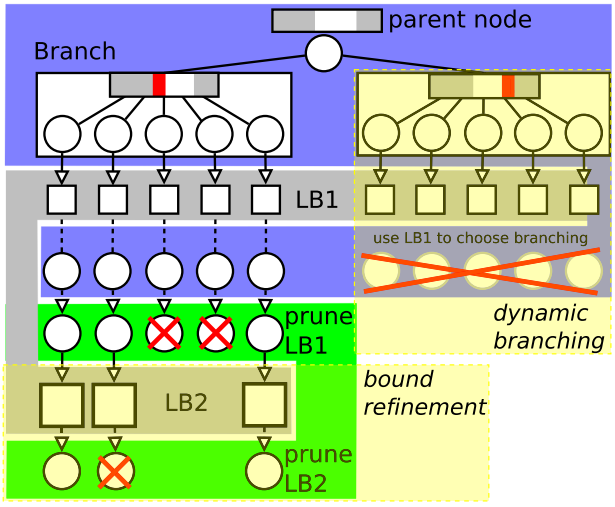


Fig. 2. Illustration of the proposed node decomposition scheme.

by using the POSIX implementation of mutexes and semaphores. For a more detailed description of the shared-memory parallelization we refer the reader to Mezmaz et al. (2014b).

3.2. Proposed node decomposition scheme

At each iteration the selection operator returns the next subproblem (σ_1, σ_2) to be decomposed, unless the exploration is finished or a solution is reached. A solution is reached if and only if $n_1 + n_2 = n$, where $n_1 = |\sigma_1|$ and $n_2 = |\sigma_2|$ designate the number of jobs fixed in the initial and final partial schedules. We call *node decomposition* a procedure which takes a parent node as input and returns a set of promising children nodes as output. The decomposition of a node is basically independent both of the search strategy and of the data structure used to store the pool of active nodes. It consists of branching, bounding and pruning operations, which are strongly interdependent.

Fig. 2 schematically represents the proposed node decomposition scheme. As one can see, the branching, bounding and pruning operations are carried out in two interleaved phases. Receiving a parent node (σ_1, σ_2), the first phase of the branching operation consists in generating both candidate offspring sets $O_{\text{Forward}} = \{(\sigma_1 j, \sigma_2),\ j \in J\}$ and $O_{\text{Backward}} = \{(\sigma_1, j\sigma_2)\ j \in J\}$. Then, using LB1, a lower bound is computed for each node in the offspring sets O_{Forward} and O_{Backward}. Note that it is not necessary to explicitly create the children nodes in memory, as LB1 can be computed more efficiently from the parent node, a job-index and a branching direction (implementation details are provided in Section 3.4). Based on the computed lower bounds exactly one of the two offspring sets is retained, which completes the branching operation. Next, the values of LB1 of the subproblems in the retained offspring set are compared to the current upper bound UB and subproblems that cannot improve the best-found solution so far are eliminated. Then, an attempt is made to eliminate more nodes by computing a stronger lower bound LB2 for the remaining subproblems. The computation of LB2 may reuse some of the computations performed for LB1, so one can view this step as a refinement phase. After the second bounding phase, the pruning operator is called again and the generated nodes are sorted in increasing order according to their lower bounds (not shown in Fig. 2, for the sake of simplicity). Note that the dynamic branching and the bound refinement components of this scheme are optional. We have developed the algorithm in a way that allows to enable/disable these components in a simple configuration file. When enabled, different choices for both options are available – they are detailed in Sections 3.3 and 3.4).

The goal of this node decomposition scheme is to reduce the computational burden of the stronger lower bound LB2 while conserving the ability of the latter to efficiently reduce the size of the search tree. The proposed scheme aims at achieving this by combining the following two ideas:

1. Subproblems are first evaluated by the less expensive lower bound LB1 and the stronger LB2 is only evaluated if the value of LB1 is smaller than UB, the current upper bound, preventing the algorithm from unnecessary computations of LB2. This presumes that it is of no use to strengthen the lower bounds of a subproblem that can be discarded. This conditional refinement of lower bounds is similar to the approach used in the B&B of Ladhari and Haouari (2005), which uses a static branching scheme.
2. If a dynamic branching rule is used, the branching decision can be made on the basis of the weaker bound LB1. However, the underlying assumption is that the LB1-based branching decisions are nearly as effective as a scheme based on LB2. Under this assumption, the proposed node decomposition scheme significantly reduces the cost of redundant bounding operations induced by dynamic branching.

Finally, the computational cost of the LB2 itself can be reduced. Indeed, as soon as a machine-pair provides a lower bound greater than the current UB, the evaluation of other machine-pairs is not necessary. Again, this early exit strategy presumes that it is useless to strengthen the lower bound of a subproblem beyond the point where it can be eliminated. As proposed by Lageweg et al. (1978), instead of evaluating all possible machine-pairs the bound can be obtained by using only a subset of m machine-pairs. In Section 3.4, we propose a new adaptive approach for choosing this subset.

3.3. Branching rules

Static branching. Using the IVM data structure described in Section 3.1, setting all digits of the direction-vector to 0 corresponds to the *forward branching* scheme that constructs solutions from left to right. The Alternate branching scheme is obtained by setting the direction-vector to $(1, 0, 1, 0, \ldots)$.

Dynamic branching. Using a dynamic branching scheme, the algorithm must choose between the sets $\{(\sigma_1 j,\ \sigma_2), j \in J\}$ and $\{(\sigma_1,\ j\sigma_2), j \in J\}$ obtained by branching the parent node (σ_1, σ_2). Assuming that the lower bounds corresponding to these subproblems have been computed, there are several possibilities to make this choice. The dynamic branching strategies considered in this paper are:

- MinBranch. This strategy locally minimizes the branching factor (the average number of children per node) by keeping the set where more nodes can be pruned, i.e. the set which contains a greater number of offspring nodes with their lower bounds greater than the current upper bound. In the case of equality, ties are broken by choosing the set where the sum of lower bounds of the remaining nodes is greater.
- MinMin. In this strategy, the set where the minimum lower bound is realized less often is retained. In the case of equality, the set where more nodes can be pruned is selected. This strategy is based on the assumption that the subtree below internal nodes with smaller lower bounds are much larger. The MinMin rule generalizes Potts' adaptive branching rule which uses this criterion on the first pass of the search to fix branching types at each level.
- MaxSum. This strategy keeps the set where the sum of lower bounds (or equivalently, the average lower bound) is greater, assuming that more subproblems in this set can be eliminated.

A potential advantage of the MaxSum strategy is that branching decisions do not depend on the current upper bound. Indeed, if branching decisions in parallel B&B algorithms depend on the value of the current upper bound, the branching rule is no longer deterministic, i.e. it is not possible to deduce the direction-vector from the position-vector of the IVM structure. In some scenarios this can be problematic. For instance, a checkpoint-restart fault tolerance mechanism that uses an interval-based encoding of subtrees, as in Mezmaz et al. (2007), relies on the ability to re-initialize the search only from a set of position-vectors.

All these three dynamic branching strategies require $\mathcal{O}(n)$ operations per branching decision which is negligible compared to the bounding effort. The largest overhead incurred by dynamic branching rules is the evaluation of twice as many subproblems. In order to be efficient, a dynamic branching approach should therefore result in half as many node expansions as a static approach.

3.4. Lower bounds

In this subsection the two lower bounds used in our algorithm are presented. A summary of the used notations is shown in Table 1.

Complete and partial makespan evaluation. Appending a job j to an initial sequence σ_1, the time between the start of processing $\sigma_1 j$ on machine M_1 and the completion of $\sigma_1 j$ on machine M_k is given recursively by:

$$C^k_{\sigma_1 j} = \max\left(C^k_{\sigma_1}, C^{k-1}_{\sigma_1 j}\right) + p_{jk} \tag{1}$$

where $C^0_\sigma = C^k_\varnothing = 0$ by convention. When σ_1 is a complete permutation, the makespan of σ_1 is equal to $C^m_{\sigma_1}$, which can be computed by starting from an empty schedule to which all jobs in σ_1 are successively appended to the right, using Eq. (1). Consequently, one makespan evaluation requires mn max-add operations.

Appending a job j to a final partial sequence σ_2, the minimum time between the start of processing jobs in $j\sigma_2$ on machine M_k and the completion of all jobs in $j\sigma_2$ on machine M_m is given recursively by:

$$C^k_{j\sigma_2} = \max\left(C^k_{\sigma_2}, C^{k+1}_{j\sigma_2}\right) + p_{jk} \tag{2}$$

where $C^{m+1}_\sigma = C^k_\varnothing = 0$ by convention. Again, when σ_2 is a complete permutation its makespan is obtained by computing $C^1_{\sigma_2}$.

One-machine bound. Given an initial sequence σ_1, the processing of unsequenced jobs $j \in J$ on machine M_k cannot start earlier than

$$r(\sigma_1, k) = \begin{cases} \min\limits_{j\in J} \sum\limits_{l=1}^{k-1} p_{jl} & \text{if } \sigma_1 = \varnothing \\ C^k_{\sigma_1} & \text{if } \sigma_1 \neq \varnothing \end{cases}$$

and the minimum time between the completion of unsequenced jobs on machine M_k and the termination of the last job in σ_2 on the last machine M_m is given by:

$$q(\sigma_2, k) = \begin{cases} \min\limits_{j\in J} \sum\limits_{l=k+1}^{m} p_{jl} & \text{if } \sigma_2 = \varnothing \\ C^k_{\sigma_2} & \text{if } \sigma_2 \neq \varnothing \end{cases}$$

The remaining total work on machine M_k, $p(k) = \sum_{j\in J} p_{jk}$ is clearly an underestimate of the time required for processing all unscheduled jobs $j \in J$ on machine M_k, so a simple lower bound on the optimal cost of a subproblem (σ_1, σ_2) is obtained by:

$$\text{LB1}(\sigma_1, \sigma_2) = \max_{1\leq k\leq m} \{r(\sigma_1, k) + p(k) + q(\sigma_2, k)\}.$$

In practice, the term $\min_{j\in J} \sum_{l=1}^{k-1} p_{jl}$ in $r(\sigma_1, k)$ (respectively $\min_{j\in J} \sum_{l=k+1}^{m} p_{jl}$ in $q(\sigma_2, k)$) can be pre-computed at the root node, taking $J = \{1, \ldots, n\}$. The resulting lower bound is only slightly weaker in the rare case where σ_1 or σ_2 is empty.

It is clear that the evaluation of LB1 for a subproblem (σ_1, σ_2) requires $\mathcal{O}(mn)$ max/add operations. However, in practice one can evaluate the $s = n - n_1 - n_2$ descendant nodes with the same computational complexity, as noted by Lageweg et al. (1978) and shown in Algorithm 1. Indeed, for a parent node (σ_1, σ_2) the quantities $r(\sigma_1, k)$, $q(\sigma_2, k)$ and $p(k)$ can be computed in $\mathcal{O}(mn)$ steps. Then, using these terms and Eq. (1) (resp. Eq. (2)) lower bounds for $(\sigma_1 j, \sigma_2)$ (resp. $(\sigma_1, j\sigma_2)$) can be obtained incrementally in $\mathcal{O}(m)$ steps each. Consequently, the calculation of LB1 for all s child nodes (resp. $2s$ with dynamic branching) requires $\mathcal{O}(mn)$ steps.

It is possible to strengthen LB1 by taking into account the fact that, after the completion of σ_1, machines are unavailable for a minimum amount of time, depending on the set of remaining jobs. However, as noted by Lageweg et al. (1978), the additional computational cost outweighs the benefits of a slight reduction of the tree size. Indeed, if a node (σ_1, σ_2) can be eliminated by adding the minimum idle time occurred by any unscheduled jobs, then all children nodes $(\sigma_1 i, \sigma_2)$ could be eliminated even without adding this term. Thus, we choose to discard this option.

Algorithm 1 LB1, one-machine lower bound

```
1: procedure LB1
Input: (σ1, σ2), J, Forward                          //~resp. (σ1, σ2), J, Backward
Output: {LB1(σ1 j, σ2), j ∈ J}                       //~resp. {LB1(σ1, jσ2), j ∈ J}
2:    for k : 1 → m do
3:       compute r(σ1, k), q(σ2, k) and p(k) = Σ_{j∈J} p_jk   //~head, tail and remaining work for parent subproblem
4:    end for
5:    for j ∈ J do
6:       //compute LB1(σ1 j, σ2),...                 //~...resp. LB1(σ1, jσ2)
7:       t0 ← r(σ1, 1) + p_j1                        //~t0 ← q(σ2, m) + p_jm
8:       lb ← r(σ1, 1) + p(1) + q(σ2, 1)             //~lb ← r(σ1, m) + p(m) + q(σ2, m)
9:       for k : 2 → m do                            //~k : m − 1 → 1
10:         t1 ← max(r(σ1, k), t0)                   //~t1 ← max(r(σ2, k), t0)
11:         t0 ← t1 + p_jk                           //~t0 ← t1 + p_jk
12:         lb ← max(lb, t1 + p(k) + q(σ2, k))       //~lb ← max(lb, t1 + p(k) + r(σ1, k))
13:      end for
14:      LB1_begin(j) ← lb                           //~LB1_end(j) ← lb
15:   end for
16: end procedure
```

Table 2
Notations: Sets of machine-pairs for LB2.

	machine-pairs
W_0	$\{(M_k, M_l), \quad 1 \leq k < l \leq m\}$
W_1	$\{(M_k, M_{k+1}), \quad 1 \leq k \leq m-1\}$
W_2	$\{(M_k, M_m), \quad 1 \leq k \leq m-1\}$
$W_\star$	dynamic set of machine-pairs, according to Algorithm 2

Two-machine bound. A stronger lower bound can be obtained by selecting two different machines M_k and M_l $(1 \leq k < l \leq m)$ – a pair of so-called bottleneck machines – and relaxing the capacity constraints of all other machines. In other words, the constraint that at most one job can be processed at a time is relaxed on machines $M_1, \ldots, M_{k-1}, M_{k+1}, \ldots, M_{l-1}, M_{l+1}, \ldots, M_m$. Introducing time-lags for each job – corresponding to aggregated processing times on machines between M_k and M_l – the resulting two-machine FSP can be exactly solved by applying Johnson's algorithm (Lageweg et al., 1978). For each machine-pair, the resolution of this two-machine FSP provides a lower bound. Denoting $C^{k,l}_{max}(J)$ the optimal makespan for the 2-machine problem defined on the set of unscheduled jobs J and bottleneck machines M_k and M_l a lower bound is

$$\text{LB2}(\sigma_1, \sigma_2) = \max_{1 \leq k < l \leq m} \{r(\sigma_1, k) + C^{k,l}_{max}(J) + q(\sigma_2, l)\}.$$

Pre-sequencing jobs at the root node according to Johnson's rule for all machine-pairs reduces the cost of evaluating $C^{k,l}_{max}(J)$ to $\mathcal{O}(s)$ steps, where $s = n - n_1 - n_2$ denotes the number of unscheduled jobs. As for LB1, $r(\sigma_1, k)$ and $q(\sigma_2, k)$ can be computed only for the parent node. However, the estimation of the unscheduled work content must be repeated for each of the s child nodes. Therefore, evaluating all descendant nodes requires $\mathcal{O}(m^2 s^2)$ steps.

In order to reduce the computational complexity of LB2, it is possible to use only a subset of all machine-pairs, which we denote W_0. Nabeshima's bound (Nabeshima, 1967) corresponds to the set W_1, containing all couples of adjacent machines, as shown in Table 2. The set W_2, proposed by Lageweg et al. (1978), fixes M_m as the second bottleneck machine and is more frequently used. In order to distinguish the resulting lower bounds we denote LB2(W) the two-machine bound corresponding to the set of bottleneck machines $W \subseteq W_0$. Using one of the latter sets of machine-pairs, W_1 or W_2, leads to a computational complexity that grows linearly with the number of machines.

However, the evaluation of all children nodes obtained from one node decomposition still requires $\mathcal{O}(ms^2)$ steps. The computational cost of a node decomposition therefore grows quadratically with n, instead of linearly (for LB1). In addition to the matrix of processing times, the computation of LB2, with pre-computed Johnson schedules and time lags, requires storing one permutation of size n for each machine-pair and time-lags for all jobs and all machine-pairs. These data structures require storing in total $2 \times \frac{m(m-1)n}{2} + mn = m^2 n$ integers, i.e. 800 kilobytes of memory for an instance of size 500×20 and 32 kilobytes for 20×20 instances. For large instances, these data structures – not needed for LB1 – are too large for L1 caches. As they are accessed very frequently and with an irregular access pattern, this is likely to increase cache miss rates, compared to the computation of LB1. Although the memory requirements of both bounds may seem derisory, they should not be neglected, as the cost of data movements in current hardware exceeds the cost of simple arithmetic operations (Giles & Reguly, 2014).

3.5. Machine-pair selection for two-machine bound

Instead of deciding statically which subset of machine-pairs to use for the computation of LB2, we propose to learn a subset of m "successful" machine-pairs at runtime. In the present context we call a pair of machines (M_k, M_l) successful if it yields a sharper lower bound than other machine-pairs, i.e. if (M_k, M_l) "maximizes" $C^{k,l}_{max}(J)$. The mechanism should exploit the acquired knowledge on the most successful machine-pairs and simultaneously try other machine-pairs in order to improve the quality of the learned subset. This task constitutes a classical exploration-exploitation trade-off, similar to the one faced in surrogate-assisted optimization (Jin, 2011) or reinforcement learning (Sutton, Barto et al., 1998).

We denote LB2($W_\star$) the proposed bound with a dynamically maintained subset $W_\star$ of m machine-pairs. It may be viewed as a surrogate model for the "exact" bounding function LB2(W_0), for which it is necessary to find a balance between exact and surrogate evaluations, in order to control and improve the accuracy of the model. It is also possible to view the machine-pairs as arms of a multi-armed bandit with unknown probabilities to maximize the value of the lower bound. As the evaluation of LB2(W_0) can be performed in a fraction of a millisecond (about $\sim 25\,\mu s$ for a 20×20 instance) an important constraint in the present case is that the learning mechanism must work with minimal computational overhead. Thus, despite the existence of several solutions that efficiently deal with the exploration-exploitation dilemma, we decided to use an *ad-hoc* approach for this particular learning task.

Before a discussion on design choices is started, let us first describe the proposed approach, for which a pseudo-code is shown in Algorithm 2. First, we must define what we mean by "successful", i.e. define a reward function. At each computation of LB2($W_\star$) a reward is given to the first machine-pair which provides a lower bound greater than UB (Algorithm 2, line 21) or, if no machine-pair is successful, a reward is given to the first machine-pair that realizes the maximal LB value (Algorithm 2, line 29). Before each computation of LB2, the set of machine-pairs is sorted in a non-increasing order of received rewards and lower bounds for machine-pairs are evaluated in that order (Algorithm 2, line 5). In order to keep low the computational overhead of the learning mechanism, it is very important to choose an appropriate sorting algorithm (Algorithm 2, line 5), for instance an in-place adaptive algorithm like insertion sort. Indeed, from one bounding operation to the next at most one machine-pair changes its position in the reward-sorted set of machine-pairs `machinePairIndex`, which allows insertion sort to run in $\mathcal{O}(r)$ steps (where r is the size of the array). The evaluation is stopped as soon as the node can be eliminated (early exit). The proposed bounding procedure alternates $2n$ complete evaluations using the full set of machine-pairs W_0 and $\alpha \times 2n$ evaluations using $W_\star$, which contains only the m most successful machine-pairs. We decided to fix the length of one learning cycle at $2n$ in order to perform a complete evaluation of all nodes on level 1 at the beginning of the search. The parameter α controls for how many node evaluations the reduced set of machine-pairs $W_\star$ is used. According to preliminary experiments we decided to set this parameter to $\alpha = 100$.

Let us now elaborate some motivations for particular design choices. First of all, the choice of rewarding only one machine-pair per bounding operation may seem unnatural. Indeed, a reward could instead be given to all machine-pairs which realize the maximum in the two-machine bound or to all machine-pairs which allow to eliminate the subproblem at hand. However, during initial experiments we observed that, for a given subproblem the maximum value is usually attained by several pairs, that appear to be strongly correlated. These machine-pairs should not compete and the learning mechanism should extract only one machine-pair that represents the group. To illustrate these observations, Fig. 3 shows a heat map, representing the LB values attained by different machine-pairs during the first 5000 bounding operations when solving a 20-machine instance (`Ta60`). Moreover, preliminary experiments indicate that there is no single machine-pair that

Algorithm 2 LB2($W_\star$), two-machine lower bound with learned machine-pairs

machinePairIndex$\leftarrow [1, 2, \ldots, \frac{m(m-1)}{2}]$ //~initial machine-pair order
rewards[:]$\leftarrow [0, \ldots, 0]$ //~initialize rewards
counter$\leftarrow 0$
procedure LB2
Input: $(\sigma_1 j, \sigma_2)$, J, $r(\sigma_1 j, :)$, $q(\sigma_2, :)$ //~resp. $(\sigma_1, j\sigma_2)$, J, $r(\sigma_1, :)$, $q(j\sigma_2, :)$
Output: LB1$(\sigma_1 j, \sigma_2)$ //~resp. LB1$(\sigma_1, j\sigma_2)$
    SortByKey(machinePairIndex,rewards) //~sort machine-pairs in order of non-increasing rewards
    if (counter $> \alpha \times 2n$) **then** //~reset after $\alpha \times 2n$ steps, $\alpha = 100$
        counter$\leftarrow 0$
        rewards[:]$\leftarrow [0, \ldots, 0]$
    end if
    if (counter$< 2n$) **then** //~for $2n$ steps ...
        numPairs $\leftarrow \frac{m(m-1)}{2}$ //~...use full LB2(W_0) bound;
    else //~for $\alpha \times 2n$ steps ...
        numPairs $\leftarrow m$ //~...use LB2($W_\star$) bound
    end if
    counter++
    LB$\leftarrow 0$
    for (i $\leftarrow$ 1 **to** numPairs) **do**
        [k,l]$\leftarrow$map(machinePairIndex[i]) //~get machine-pair $[k, l]$ from index i
        $lb_{k,l} \leftarrow$evalJohnson([k,l],$r(\sigma_1 j, :), q(\sigma_2, :), J)$ //~lower bound for machine-pair $[k, l]$
        if $lb_{k,l} >$ UB **then** //~(σ_1, σ_2) can be pruned...
            rewards[machinePairIndex[i]]++ //~reward machine-pair [k,l]
            return $lb_{k,l}$ //~early exit
        end if
        if $lb_{k,l} >$ LB **then** //~$lb_{k,l}$ improves LB...
            LB$\leftarrow lb_{k,l}$ //~update LB
            bestpairIndex$\leftarrow$machinePairIndex[i] //~[k,l] candidate for reward
        end if
    end for
    rewards[bestpairIndex]++ //~reward first machine-pair that reached $lb_{k,l}$ = LB
    return LB
end procedure

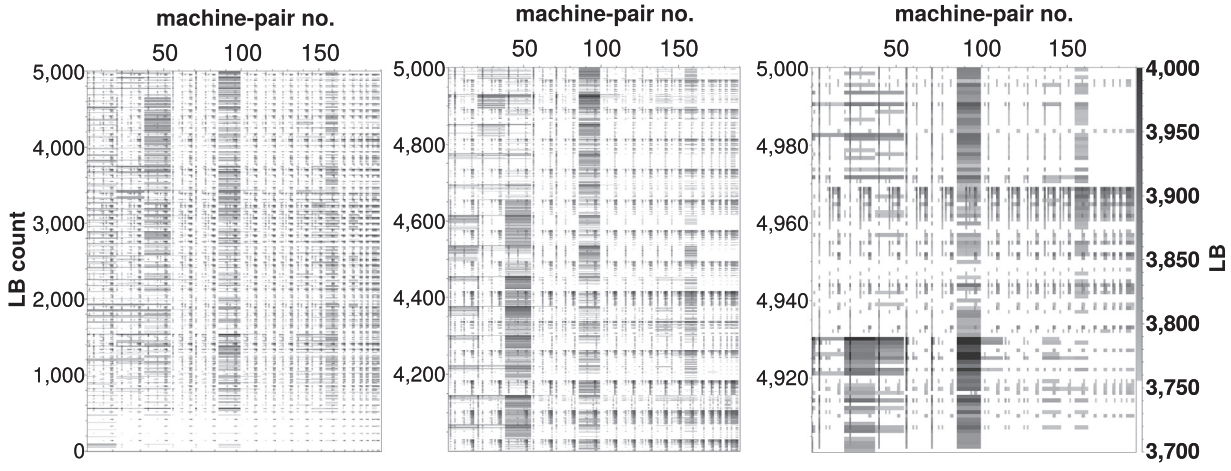


Fig. 3. Lower bound values realized by all 190 machine-pairs during the first 5000 bounding operations (for instance Ta60, defined by $n = 50$ jobs and $m = 20$ machines). **Left**: bounding operations 1-5000, **middle**: zoom on 4000-5000, **right**: zoom on 4900-5000. White indicates LB values smaller than the best known solution (< 3756), gray indicates values that allow to eliminate the subproblem, i.e. greater or equal to the best known solution (≥ 3756). Darker grayscale indicates higher values.

provides the strongest lower bound for all subproblems and that it is not realistic to search for a single most-promising machine-pair.

The choice of alternating exploration and exploitation phases for a fixed number of iterations is mainly due to the fact that the overhead incurred by the learning mechanism must be very low. For instance, we tried an alternative approach inspired by ϵ-greedy learning (Sutton et al., 1998): using the same reward function, replace β% of the machine-pairs in $W_\star$ by less successful machine-pairs picked uniformly at random. However, while this strategy revealed quite effective in learning a good subset of machine-pairs, the computational cost of generating several random numbers at each bounding operation significantly increases the execution time of the algorithm. We observed the same for an approach that consists in choosing the top-m machine-pairs after adding exponential noise to the rewards – similar to the Follow the Perturbed Leader (FPL) algorithm (Sutton et al., 1998). This FPL-inspired approach

also showed very promising results regarding the size of explored trees, but the generation of noise and especially the increased cost for sorting make this approach prohibitively costly.

4. Computational experiments

4.1. Experimental protocol

Notations. In the context of the proposed node decomposition scheme (Fig. 2), we denote

$$(\mathrm{P})\mathrm{BB}-\mathrm{LB}/\mathrm{BR}$$

the (parallel) B&B algorithm using the branching rule BR and the lower bounding scheme LB, where BR designates one of the five branching rules introduced in Section 3.3

$$\mathrm{BR}\in\{\mathrm{Forward},\mathrm{Alternate},\mathrm{MinBranch},\mathrm{MinMin},\mathrm{MaxSum}\}$$

and LB designates one of the following:

- LB1 – only the one-machine bound is used for dynamic branching (if enabled) and pruning. Bound refinement is disabled.
- LB2(W) – only the two-machine bound with the set of machine-pairs W is used for dynamic branching (if enabled) and pruning. W designates one of the four sets of machine-pairs defined in Table 2.
- LB12(W) – LB1 is computed first and used for dynamic branching (if enabled), in a second phase LB2(W) is used as a refinement.

Considering 5 branching rules and 4 variants of LB2(W), there are $5\times(1+4+4)=45$ variants of parallel and sequential algorithms.

Objectives and organization of experiments. The objective of this experimental study is to evaluate the efficiency of different combinations of branching rules, lower bounds and search parallelization, as well as the overall performance of the resulting B&B algorithm. To this end, a series of 6 experiments is performed: the first 4 aim at fixing the best choices for the different components of the algorithm and the last 2 consist in attempts to solve benchmark instances to optimality. Experiments are organized as follows:

1. In Section 4.2 the focus is put on the evaluation of *branching rules*. Two experiments are carried out with the sequential version:
   (a) The first experiment compares the efficiency of the five branching rules, in combination with the weakest and the strongest lower bound LB1 and LB2(W_0).
   (b) The focus of the second experiment is put on the three dynamic branching rules and the question how the the quality of lower bounds affects dynamic branching decisions.
2. In Section 4.3 the proposed *adaptive two-machine bound* (Algorithm 2) is compared to four other lower bounds, investigating the trade-off between tree size and computational effort and considering a wide range of instances from 10 to 800 jobs and 5 to 20 machines.
3. Considering the two most successful lower bounding schemes from the previous experiment, Section 4.4 provides an evaluation of the *parallel tree search* using up to 32 threads for the parallel version.
4. Section 4.5 provides an *overall evaluation* of the sequential and parallel algorithms, using the 120 instances from Taillard's benchmark. The results are compared with approaches in the literature.
5. Section 4.6 reports experimental results obtained with the *VRF benchmark* instances.

Table 3 provides an overview of the experiments and the corresponding algorithm settings.

Initial upper bound. Unless specified otherwise (in Section 4.2), the initial upper bound is obtained by the well-known Nawaz-Enscore-Ham (NEH) heuristic (Nawaz et al., 1983). Although better initial upper bounds can be obtained (using variants of the NEH heuristic or state-of-the-art PFSP solvers) we choose the basic NEH solution in order to facilitate the comparison with other B&B approaches in the literature, which use equal or better initial upper bounds.

Computing platform. All experiments are performed on a dual-socket system equipped with two 8-core Xeon E5-2630v3 CPUs, clocked at 2.4 gigahertz. Hyperthreading capabilities are enabled, so the system has a total of $2\times2\times8=32$ logical cores and the operating system is a CentOS 7 Linux distribution. The B&B code is written in C++ using pthreads for parallel execution and is compiled with GCC 4.8.5.

4.2. Evaluation of branching rules

The goal of this first experiment is to compare the performance of the branching strategies introduced in Section 3.3 and to investigate how branching and bounding operators interact.

Branching decisions impact the size of the explored tree in two ways: (1) they impact the size of the critical tree (composed of nodes with LBs smaller than the optimal makespan, called critical nodes) and (2) they may lead to a faster discovery of better solutions, which reduces the number of explored non-critical nodes. According to initial experiments, none of the considered branching rules is significantly better in finding high-quality solutions, although for particular instances strong differences can be observed. Moreover, the number of non-critical nodes strongly depends on other factors (search strategy, initial upper bound and parallelization). Therefore, we consider the variation of explored non-critical nodes due to the branching rule as noise, which is removed by limiting the search to the critical tree, i.e. the initial upper bound is always set to the optimal solution.

Experiment 1: static vs. dynamic branching rules

Table 4 reports the size of the critical tree for five branching rules (two static rules Forward, Alternate, and three dynamic strategies MaxSum, MinMin and MinBranch) and two lower bounds LB1 and LB2(W_0), where W_0 designates the set of all machine-pairs.

The size of a tree is measured by counting the number of decomposed nodes. For each lower bound the best value (smallest

Table 3
Overview of experiments. ✔indicates that the component is compared to alternative settings, and X indicates a fixed component.

Sect.	BR					LB			W				#thd	UB-init	Inst.
	Fwd	Alt	MinMin	MinBr	MaxSum	LB1	LB12(W)	LB2(W)	W_0	W_1	W_2	$W_\star$			
4.2	✔	✔	✔	✔	✔	✔		✔	X				1	opt	Ta
4.2			✔	✔	✔		✔	✔	X				1	opt	Ta
4.3				X		✔	✔		✔	✔	✔	✔	1	NEH	VRF
4.4				X		✔	✔					X	**1 → 32**	NEH	Ta
4.5				X		X							1,32	NEH	Ta
4.6				X		X							32	NEH	VRF

Table 4
Size of critical tree (in number of decomposed nodes) for different branching rules and lower bounds (for instances Ta01–Ta20). The last two rows give the average tree size (NN) and elapsed time for the sequential algorithm (t).

Ta	LB1					LB2(W_0)				
	Fwd.	Alt.	MaxSum	MinMin	MinBranch	Fwd.	Alt.	MaxSum	MinMin	MinBranch
1	1	1	1	1	1	1	1	1	1	1
2	74	891,014	**38**	**38**	**38**	37	25,543	**13**	**13**	**13**
3	2.6×6	332	60	**42**	**42**	80,107	35	29	29	**27**
4	1.2×6	152	52	39	**32**	33,358	30	49	**24**	26
5	$[7.4 \times 9]$[a]	2.7×6	23,323	**8960**	11359	$[596 \times 6]$[a]	89605	5537	**1468**	1918
6	$[7.4 \times 9]$[a]	868	22	**14**	**14**	117×6	97	27	**14**	**14**
7	1	1	1	1	1	1	1	1	1	1
8	113×6	770	20	**18**	25	13.9×6	51	19	**17**	18
9	1.7×6	2660	57	62	**39**	58,873	491	40	42	**34**
10	83.2×6	140	21	**14**	**14**	8.1×6	22	**9**	12	12
11	6.7×10^6	2.3×10^6	177,297	157,028	**150,416**	438,593	21,2171	88,779	74,190	**71,869**
12	3.9×10^6	1.5×10^6	95,851	95,473	**80,865**	666,965	124,390	56,352	46,460	**43,982**
13	4.1×10^6	1.4×10^6	171,353	154,315	**149,306**	1.3×10^6	196144	126,966	106,453	**106,090**
14	2.6×10^6	243,624	29,793	17,485	**15,662**	144,669	45,161	23,303	11,775	**11,574**
15	9.5×10^6	338,923	38,318	32,084	**31,594**	457,251	42,127	27,550	21,606	**21,270**
16	10.7×10^6	40,750	**1564**	1816	1704	2.6×10^6	4532	1292	1424	**1291**
17	577×10^6	340×10^6	43.6×10^6	40.5×10^6	**35.2×10^6**	17.5×10^6	10.7×10^6	6.7×10^6	**4.3×10^6**	4.5×10^6
18	750×10^6	1.3×10^6	149,250	88,013	**86208**	79.7×10^6	147,113	107,944	62,554	**62143**
19	253	12788	191	176	**134**	140	2300	149	111	**103**
20	859×6	4.2×6	348,475	285,670	**257,606**	4.8×6	333,109	157,251	121,968	**116,813**
NN	862×6	17.8×6	2.2×6	2.1×6	**1.8×10^6**	42.1×6	595,500	365,453	**237,973**	247,003
t	467 seconds	15.5 seconds	2.8 seconds	2.9 seconds	2.3 seconds	323 seconds	13.0 seconds	9.5 seconds	7.0 seconds	7.4 seconds

[a]unsolved after 1 hour.

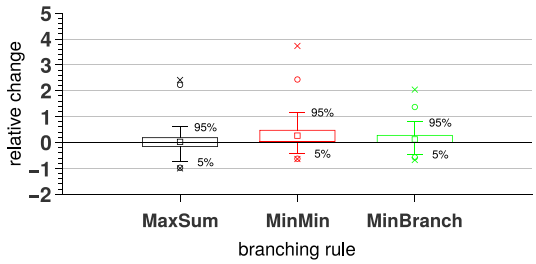
tree size) is highlighted in bold. In this experiment, only Taillard's instances Ta01–Ta20 defined by 20 jobs and 5–10 machines are considered, as the computation time for some of the branching rules becomes excessive for larger instances.

Several observations can be made:

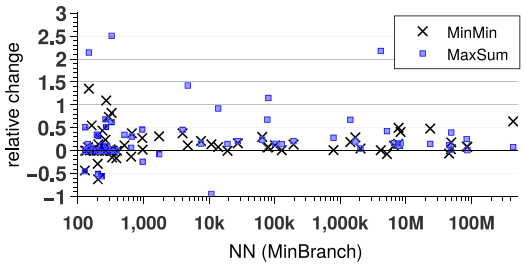
1. Comparing the two static branching schemes Forward and Alternate the latter rule results in much smaller critical trees for almost all instances, reducing the tree size by several orders of magnitude in some cases (e. g. Ta08, Ta10 and Ta16). Exceptions to this observation are instances Ta02 and Ta19. Using Forward branching and either LB1 or LB2(W_0), the algorithm is unable to prove the optimality of the best-known solutions for Ta05 within a time-limit of 1 h.
2. For all tested instances and regardless of the used lower bound, all three dynamic branching rules result in smaller critical trees than the two static branching strategies (with the exception of Ta19). Using a dynamic approach, the 20×5 instances are easily solved by a sequential B&B within a few milliseconds and most of the 20×10 instances are solved within a few seconds. For LB1, the tree explored with dynamic branching rules always contains less than half as many nodes as the tree explored with the Alternate rule (about 6 times smaller on average). Therefore, the tree size reduction compensates for the redundant node evaluation in dynamic branching rules. In most cases this is also valid for LB2(W_0), but the difference is less clear and there are a few exceptions (e. g. Ta13 and Ta15).
3. Among the dynamic branching strategies, the MinBranch rule results in most cases in the smallest tree size. For the considered set of instances the MinMin rule provides similar results and MaxSum seems to be the least efficient of the three dynamic branching rules.
4. For the two static branching rules, using LB2(W_0) allows to reduce the average execution time, compared to LB1. In contrast, *when using dynamic branching,* LB1 *is more efficient than* LB2*(W_0)* – the tree size reduction provided by LB2(W_0) fails to outweigh the additional computational effort. Indeed, the results in Table 4 indicate that the combination of LB2 with dynamic branching produces an antagonistic effect, in the sense that better branching decisions attenuate the efficiency of LB2 (relatively to LB1). Consider, for example, instance Ta11: when using LB1/Forward 6.7×10^6 nodes are decomposed (resp. 2.3×10^6 with Alternate); using the stronger LB2(W_0), the critical tree is 15 times smaller, containing 0.44×10^6 nodes (resp. 10 times smaller, containing 0.21×10^6 nodes). For the same instance and branching dynamically, using LB1 results in critical trees composed of $150–177 \times 10^3$ nodes; using LB2(W_0) the critical tree is only 2 times smaller, containing $72–89 \times 10^3$ nodes. Very similar observations can be made for the other instances considered in Table 4.
5. *To sum up,* if one chooses to use the weaker bound LB1, it is strongly recommended to pair it with a dynamic branching scheme. When using LB2(W_0), the advantage of dynamic schemes is less clear, as Alternate branching provides comparable results while being easier to implement. Simply put, when using dynamic branching, it seems less important to use a strong LB – and conversely, when a strong LB is used, the effect of dynamic branching is less visible. In terms of average execution time and for the (small) set of instances considered here, BB-LB1/dynamic *clearly outperforms the other approaches.*

It should be noted that, to the best of our knowledge, the combination of LB1 with dynamic branching rules has not been considered in the literature up to the time of writing. Indeed, as LB2 has been shown to be the most efficient lower bound in Lageweg et al. (1978), the adaptive branching rule in the algorithm of Potts (1980) has only been evaluated in combination with LB2. In retrospective, if both techniques had been discovered in the opposite order, the power of dynamic branching rules would have appeared more clearly and the LB2 bound maybe would have had less success.

The impact of the branching rule on the efficiency of LB functions is not intuitively clear and we can currently only provide the following tentative explanation for this phenomenon: As the three dynamic branching rules choose the set of descendant nodes which is "easier" to prune, the success-rate (percentage of node



(a) Relative change in (critical) tree size when using LB12(W_0) instead of LB2(W_0).



(b) Comparison of different branching rules in BB-LB12(W_0)/BR, baseline: BR=MinBranch.

Fig. 4. Evaluation of LB1-based dynamic branching approaches using instances `Ta1-50`, `Ta61-80`, `Ta81-100` and `Ta111-120`.

evaluations that lead to an elimination) of the weaker LB1 increases by a greater margin than the success-rate of LB2. To make this clearer, consider the following hypothetical example: suppose that the current upper bound is equal to 12, and for the two sets of descendant nodes O_{Forward} and O_{Backward} the evaluation of LB1 yields lower bounds:

$$LB1_{\text{Forward}} = \{10, 11, 12\} \qquad LB1_{\text{Backward}} = \{12, 13, 14\},$$

while the corresponding evaluation of LB2 yields the stronger lower bounds:

$$LB2_{\text{Forward}} = \{12, 13, 14\} \qquad LB2_{\text{Backward}} = \{14, 15, 16\}.$$

LB2 allows to eliminate all three children nodes, regardless of the selected set of offspring nodes. In the case of LB1, selecting O_{Forward} allows to prune only 1 node, while all three nodes can be pruned if set O_{Backward} is selected, meaning that an algorithm using LB1 will benefit more from dynamic branching.

Experiment 2: comparison of dynamic branching rules. The results in Table 4 do not allow to understand if and how the efficiency of dynamic branching decisions depends on the quality of the lower bounds or on the problem size. Therefore, Fig. 4 shows experimental results obtained with 90 instances of Taillard's benchmark (excluding instances with $m = 20$ machines and $n = 50, 100, 200$ jobs).

On the left, Fig. 4a compares the critical tree sizes for two B&B algorithms: A_{12} =BB-LB12(W_0)/BR and A_2 =BB-LB2(W_0)/BR. Both algorithms make identical pruning decisions for identical subproblems, so the difference between resulting tree sizes can be uniquely attributed to different branching decisions, based on LB1 in A_{12} and based on LB2(W_0) in A_2. The goal is to see whether LB1 can be used as a sufficiently accurate approximation of LB2(W_0) in branching decisions. For the three dynamic branching strategies, Fig. 4a shows a boxplot of the relative change in tree size, using A_2 as the baseline (i. e. $\frac{NN_{A_{12}} - NN_{A_2}}{NN_{A_2}}$ where NN_{X} designates the size of the critical tree obtained with algorithm X. Thus, a relative change equal to 1 indicates that the tree obtained with LB12 is twice as large as the tree obtained with LB2. The results show that no dramatic increase of the tree size occurs when branching decisions are based on the weaker bound LB1, instead of LB2. Indeed, the highest observed change is a four-fold increase in tree size and for more than 90% of the instances relative change is smaller than 1.

On the right, Fig. 4b compares the three dynamic branching strategies BR $\in$ {MinBranch, MinMin, MaxSum} in terms of explored critical nodes, when used in algorithm BB-LB12(W_0)/BR. The MinBranch strategy is used as a baseline and the relative change in critical tree size obtained with the MinMin and MaxSum strategies is shown on the vertical axis. While none of the three strategies outperforms the two other ones in all cases, one can see that the MinBranch strategy results in most cases in the smallest critical tree among the three approaches, confirming the result of Table 4 on a larger set of instances.

4.3. Evaluation of bounding rules

The experiments in the previous section validate the dynamic branching approach using the MinBranch rule based on LB1. In this experiment, we compare the performance of five algorithms, BB-LB12(W)/MinBranch, corresponding to different machine-pair subsets W (as defined in Table 2). Therefore, all B&B variants in this subsection use the same branching scheme and differ only in the bound refinement phase (Section 3.2, Fig. 2). The objective is to determine whether it is efficient to compute stronger (two-machine) lower bounds for unpruned nodes after the first bounding phase, and if so, for which machine-pairs this bound should be computed. The goal is also to understand how the size of an instance (number of jobs and machines) affects this choice. In this subsection, the B&B algorithm is always initialized with the NEH heuristic solution.

A total of 100 instances from the `VRF_small` benchmark are used in this experiment. Instances with 5 machines (resp. with 20 machines and $n \geq 30$ jobs) are spared out as they are too easy (resp. too hard) to solve. The algorithm BB-LB12(W_0) is used as the baseline, as it always results in the smallest tree size. The left column of Fig. 5 shows the relative tree size increase which is observed when W_0 is replaced by machine-pair sets $W_\varnothing$ (corresponding to BB-LB1), W_1, W_2 and $W_\star$. In the left column, smaller values are better and a relative increase of 1 means that the explored tree is twice as large as the baseline. The right column shows the relative speedup with respect to the reference algorithm, i.e. the ratio of execution times $\frac{T(W_0)}{T(W_i)}$ – higher values are better and a relative speedup of 1 indicates equal execution times. The two first rows of Fig. 5 show the results for instances `VRF10_x` and `VRF20_x` defined by $n = 10$, resp. $n = 20$ jobs, and the third row shows the results for instances defined by $n = 30$–60 jobs and $m = 10$ machines.

The size of explored trees varies strongly from one instance to another, even within the same class. Thus, in order to represent values of different order of magnitude in the same figure, axis breaks were introduced in the vertical axis (for `VRF10_x` and `VRF20_x`). For instances which remained unsolved by all 5 algorithms after a time-limit of 15 minutes the relative change and speedup are set to -1. If only some, but not all algorithms solve an instance within the time-limit, the instance is re-solved without time-limit by all algorithms. The following observations can be made:

- For instances `VRF10_x` one can see that the learned machine-pair set $W_\star$ results in tree sizes which are closest to the full machine-pair set W_0. In the worst case the explored tree is

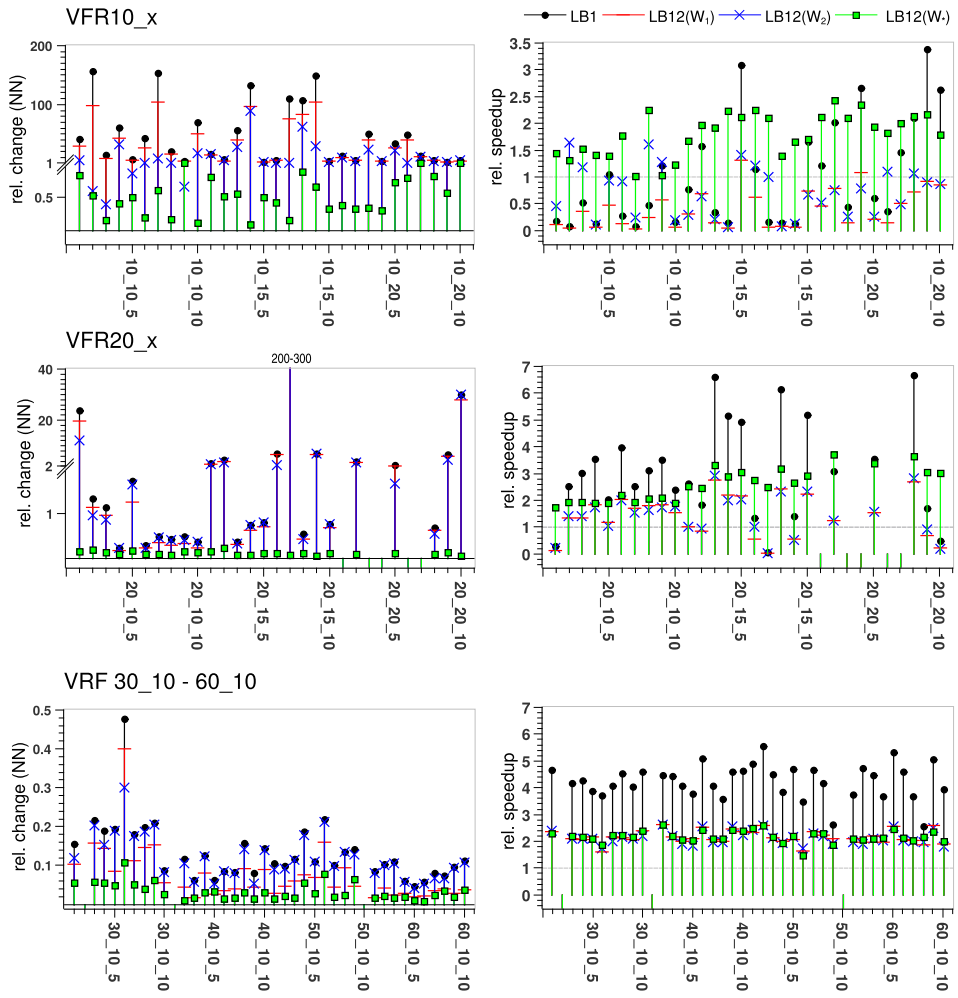


Fig. 5. Evaluation of different sets of machine-pairs for LB12. Baseline: BB-LB12(W_0).

136% larger. Using the fixed machine-pair set W_1 (resp. W_2) the algorithm explores trees which are at least 10× times larger than with W_0 in 13 (resp. 18) out of 40 cases. Naturally, using only LB1 (no refinement) results in the largest trees, up to 150× larger than the baseline. Considering the execution time, neither W_1 nor W_2 allows to complete the search consistently faster than with the full set W_0. The learned machine-set $W_\star$ is the only one which outperforms the full two-machine bound (W_0) in all cases, completing the search up to 2.5× faster.

- For instances VRF20_x the differences in tree size are less extreme (note the change in scale on the y-axis) – with the exception of instance VRF20_15_7. For the latter, BB-LB1, BB-LB12(W_1) and BB-LB12(W_2) explore trees which are 200–300× larger than the baseline, while for BB-LB12($W_\star$) a mere 6% increase is observed. As a consequence, the algorithm using $W_\star$ is 2.5× faster than the baseline, while the others experience a slowdown of 20–50×. Overall, in terms of tree size, the learned set $W_\star$ gives the best result for all solved instances, as the explored trees are at most 36% larger. One can see that the static sets W_1 and W_2 lead to the exploration of trees closer to the worst-case $W_\varnothing$ than to the best-case W_0. In terms of execution time, only the bound LB2($W_\star$) is more efficient than LB2(W_0) for all VRF20_x instances. However, the most efficient choice for 17 out of 25 solved instances is to use only the one-machine bound LB1. This clearly demonstrates that the LB2($W_\star$) refinement bound acts as an efficient safety net – used in addition to LB1, it may prevent an explosion of the search tree, as for VRF20_15_7, but this comes at the cost of spending more time on instances that are efficiently solved by BB-LB1.
- For the VRF instances defined by $m = 10$ machines and $n = 30$–60 jobs all results can be represented in the same figure, without axis breaks. Indeed, even for $W_\varnothing$ (only LB1) the explored tree is at most 48% larger than the baseline. As for the smaller instances, the learning-based bound LB2($W_\star$) mimics the behavior of LB2(W_0) better than LB2(W_1) and LB2(W_2). However, as all the different lower bounds lead to a very similar number of nodes to be explored, the total exploration time depends almost exclusively on the computational cost of a node evaluation. Using any of the machine-pair sets of cardinality m (or $m-1$) allows to complete the exploration about 2× faster than the baseline algorithm. It should also be noted that the execution time for $W_\star$ is almost the same as for W_1 and W_2, which shows that the learning mechanism works with a very small computational overhead. However, without the node refinement phase, i.e. only using LB1, the algorithm is on average about 4× faster than the baseline.

In addition to Fig. 5, which only shows exploration statistics relative to the baseline algorithm, Table 5 provides the average

Table 5
Average execution time and tree size (number of decomposed nodes) for BB-LB12(W_i) ($W_i \in \{W_0, W_1, W_2, W_\star, W_\varnothing\}$) and the three groups of instances shown in Fig. 5.

VRF	W_0		W_1		W_2		$W_\star$		$W_\varnothing$	
	NN	t	NN	t	NN	t	NN	t	NN	t
10_x	1940	0.03	14,370	0.06	35,163	0.11	2651	**0.01**	49,657	0.05
20_x	20×6	476	235×6	1076	273×6	1283	22×6	**158**	319×6	522
30-60	27×6	351	32×6	160	31×6	163	29×6	155	33×6	**77**

Table 6
Parallel performance of PBB-LB1 and PBB-LB12($W_\star$) solving Taillard's instances Ta21-30, 41-50, 91-100, 111-120 to optimality. Time-limit = 10 minutes, using $p = 1, 2, 4, 8, 16, 32$ threads on $2 \times$ E5-2630v3 CPUs

p	#solved		t (seconds)		NN(10^6)		ARPD		#solved		t (seconds)		NN(10^6)		ARPD	
	LB1	LB12	LB1	LB12	LB1	LB12	LB1	LB12	LB1	LB12	LB1	LB12	LB1	LB12	LB1	LB12
	Ta21-30								Ta41-50							
1	5	6	391	409	206	60	0.34	0.31	10	10	41	85	15.6	14.8	0.0	0.0
2	6	6	349	345	351	101	0.22	0.26	10	10	23	43	15.1	14.2	0.0	0.0
4	7	9	260	246	466	129	0.19	0.03	10	10	8.7	17	10.4	9.9	0.0	0.0
8	9	9	156	146	487	136	0.05	0.0	10	10	5.1	10.2	10.4	10.2	0.0	0.0
16	9	10	110	93	634	166	0.05	0.0	10	10	2.8	5.7	9.5	9.5	0.0	0.0
32	9	10	100	80	715	181	0.04	0.0	10	10	2.2	4.1	9.7	9.7	0.0	0.0
	Ta91-100								Ta111-120							
1	9	9	64	66	4.9	2.5	0.03	0.02	0	0	600	600	7.4	4.3	2.21	2.21
2	10	9	56	63	6.4	4.4	0.0	0.02	0	0	600	600	17.8	9.1	1.96	1.93
4	10	10	1.2	2.7	0.24	0.36	0.0	0.0	1	1	560	581	27.6	15.3	1.19	1.17
8	10	10	0.8	2.1	0.29	0.49	0.0	0.0	4	4	403	482	31.7	22.0	0.44	0.44
16	10	10	0.6	1.5	0.32	0.65	0.0	0.0	6	6	287	400	37.9	35.7	0.13	0.16
32	10	10	0.6	1.8	0.92	0.45	0.0	0.0	7	7	265	445	38.2	48.1	0.04	0.1

execution time (in seconds) and tree size for the 5 lower bounds and the 3 groups of instances. As one can see, the VRF10_x instances are solved within a few milliseconds as only a few thousand node decompositions are required to find an optimal solution and prove its optimality. Although a naive and easy-to-implement algorithm (i. e. BB-LB1) will usually be the best choice for these instances, the dynamic set of machine-pairs $W_\star$ provides the best results, solving these instances on average in 14 ms (instead of 27ms with the second-best choice W_0). For the VRF20_x instances, the learning-based bounding approach ($W_\star$) outperforms the second-best approach (W_0) on average by a factor of $3\times$. However, as shown in Fig. 5, this is essentially due to the "pathological" case of VRF20_15_7, and the averaged results underline the "safety" function of LB2($W_\star$). For the group of 10-machine instances defined by $n \geq 30$ jobs (VRF30_10 – VRF60_10) the relative efficiency of the different bounding functions changes radically. Indeed, on average all five algorithms decompose roughly the same amount of nodes ($30 \times 10^6 \pm 10\%$). In this situation, it is clear that the best choice is to use the LB with the smallest computational cost (LB1), which outperforms the second-best LB2($W_\star$) by a factor $2\times$ in terms of execution time.

The observation that the five different LBs tend to be of similar strength as the number of jobs n increases can be explained by the conjecture that, for a fixed m and randomly generated instances,

$$\lim_{n\to\infty} Pr[C^\star_{\max} = LB1] = 1, \tag{3}$$

which was formulated by Taillard (1993) and strongly supported by empirical evidence in Kalczynski and Kamburowski (2009). If Eq. (3) holds for the weakest lower bound, it is obvious that the gap between weaker and stronger lower bounds also goes to zero with probability 1 as n grows to infinity.

Moreover, an immediate corollary of Eq. (3) is that $\lim_{n\to\infty} Pr[LB1 < C^\star_{\max}] = 0$. The set of nodes $\{x | LB1(x) < C^\star_{\max}\}$ is called *critical tree* and corresponds to the portion of the tree which needs to be visited to prove the optimality of $C^\star_{\max}$. The critical trees of most instances in the (VRF30_10 – VRF60_10) group are indeed very small (the algorithm terminates quickly, when initialized with the optimal solution), meaning that the algorithm spends most of its time exploring non-critical nodes. This observation adds further empirical evidence to support this conjecture.

4.4. Evaluation of parallel tree search

When performing parallel tree search with a fixed (e. g. optimal) incumbent solution, the algorithm explores the same tree as in a sequential search. In this case, carefully designed parallel B&B algorithms can achieve linear speedup, provided the total workload is large enough. However, in the general case speedup anomalies may occur, as parallelization approaches usually modify the search trajectory in a non-deterministic manner. Using p parallel B&B processes that explore disjoint portions of the search space and share the same global upper bound, the search may explore less or more (even $> p$ times more) nodes than the sequential search. The work performed by additional processes is of speculative nature. As it corresponds to an anticipated exploration of portions of the search space (with respect to the sequential search) it may: (a) accelerate the discovery of new solutions, (b) correspond to work that needs to be done anyway (critical tree) or (c) be redundant, in the sense that the work could be avoided by delaying it to a later point, when a better upper bound is available. The resulting superlinear, decelerating or detrimental speedup anomalies are a well-known phenomenon in parallel tree search algorithms. Moreover, if dynamic branching decisions depend on the cost of the incumbent solution (e. g. MinBranch), then the faster discovery of good solutions may either improve or degrade the overall performance.

It is beyond the scope of this paper to investigate such behaviors in detail or to formally analyze the conditions under which detrimental speedups can occur. Instead, the goal of this section is to show empirically that the use of parallelism, even exploiting a moderate number of cores, is generally beneficial and a key ingredient for solving larger problem instances. Increasing the number

of threads from $p = 1$ to $p = 32$ and for four groups of Taillard's instances, Table 6 reports the number of instances solved (#solved), the average execution time (in seconds, including instances that reached the time-limit of 10 minutes), the number of decomposed nodes (NN, in 10^6 nodes) and the Average Relative Percent Deviation (ARPD) with respect to the best-known makespan (as reported in Taillard, 2015). The four groups used in this experiment are: Ta21–30 (20×20), Ta41–50 (50×10), Ta91–100 (200×10) and Ta111–120 (500×20). Results are reported for the parallel versions of the two algorithms BB-LB1/MinBranch and BB-LB12($W_\star$)/MinBranch, which were shown to perform best in the previous subsections.

Globally, one can notice that, as the number of threads increases, more instances are solved within the time-limit of 10 minutes and the average resolution time decreases, as well as the ARPD.

For the Ta21–30 group LB12($W_\star$) provides better results than LB1, in terms of solved instances, ARPD and execution time. In contrast, for the remaining groups LB1 requires less computation time, achieving similar results in terms of solution quality. This has already been observed for the VRF instances in the previous section: for the three groups Ta41–50, Ta91–100 and Ta111–120 the number of jobs is "much" larger than the number of machines and empirical evidence suggests that two different lower bounds frequently yield identical values, close or equal to the optimal solution with a high probability.

For the Ta41–50 group, all ten instances are solved by the sequential algorithm in 41, respectively 85 seconds on average. One can see that, doubling the number of threads from 2 to 4 the average execution time is divided by $\sim 2.5\times$, as the average tree size decreases from $\sim 15 \times 10^6$ to $\sim 10 \times 10^6$. Superlinear speedup is also observed for the 200×10 instances *Ta91-100*: while the sequential algorithm leaves one instance (Ta100) unsolved after 600 seconds, the parallel version exploiting all 16 CPU cores solves all ten instances in 0.6 seconds on average.

None of the 500×20 instances Ta111–120 is solved by the sequential B&B while the parallel versions solve 7/10 (with both bounding schemes the same instances are solved). For this class an interesting observation can be made: using 32 threads the LB1-based algorithm explores on average *fewer* subproblems than the algorithm using the stronger LB12 (despite the fact that the averages include unsolved instances, for which the faster LB1 necessarily explores larger trees). The most likely explanation for this is the following. As the LB12 bounding scheme strengthens the lower bounds only if necessary, the evaluation of a promising subproblem takes more time than the evaluation of a poor partial solution. Consequently, worker threads that are on a path leading to high-quality solutions are slowed down with respect to other workers. In contrast, as LB1 is more regular (computationally), this bound allows all workers to progress at approximately the same pace. Obviously, this effect is unpredictable and does not occur systematically, but it clearly illustrates that the LB which offers the most efficient trade-off in a sequential context is not necessarily the best choice in a parallel B&B algorithm, as different computational aspects (e. g. irregularity) must be taken into consideration.

4.5. Comparison with B&B approaches in the literature

For all experimental runs in this section the initial upper bound obtained by the NEH heuristic and only the algorithm (P)BB−LB1/MinBranch is used. Table 7 provides a summary of exploration statistics for the complete set of Taillard's instances using 32 threads. The 12 groups of instances are sorted according to the number of machines, which is a better indicator of hardness than the number of jobs. In order to facilitate a comparison with results reported in the literature, results are also provided for the sequential algorithm. In the literature, results for Taillard's benchmark instances have been reported for the algorithms DFS/dyn and CBFS (Ritt, 2016), BB-LH (Ladhari & Haouari, 2005) and LOMPEN (Companys & Mateo, 2007) – an overview of these algorithms is provided in Section 2. We limit our comparison to the results reported for DFS/dyn and CBFS in Ritt (2016), as latter shows that both DFS/dyn and CBFS outperform the BB-LH and LOMPEN algorithms – solving more instances within a time-limit of 1 h, requiring less time and processing significantly less nodes. We apply the same time-limit as Ritt (2016), using an equivalent CPU and initial upper bounds of lower quality (provided by the NEH heuristic instead of the iterated greedy algorithm (Ruiz & Stützle, 2007)).

Within the time-limit our sequential algorithm solves 78 of the 120 instances, while the parallel algorithm solves 90 instances, including the groups Ta21–30 and Ta111–120 defined by $m = 20$ machines and $n = 20$, respectively $n = 500$ jobs. For all instances defined by $m = 20$ machines and $n = 50$, 100 and 200 jobs the time-limit was reached by both the parallel and sequential version. While the sequential BB-LB1 algorithm solves 9 of the 20×20 instances – decomposing on average 462×10^3 nodes/second, the CBFS algorithm only solves 3 out of 10 instances – at a decomposition rate of 1.8×10^3 nodes/second, i.e. for this class of instances BB-LB1 runs at a $267\times$ higher node processing rate. This ratio increases with an increasing number of jobs: for the 200×10 instances, the results reported in Ritt (2016) indicate a node decomposition rate of 38 nodes/second, i.e. over $2100\times$ less than BB-LB1 (81×10^3 nodes/second). As CBFS uses a better initial upper bound and an equivalent or better search strategy, this gap can only be attributed to the fact that the CBFS algorithm uses the costly LB2(W_0) bound with the complete set of machine-pairs.

There is a second notable difference between the LB1-based approach and the results reported for the DFS/dyn and CBFS algorithms. Like CBFS and DFS/dyn, our sequential algorithm solves 69 of the 70 instances defined by $m \leq 10$ machines. However, the remaining unsolved instance is not the same: while CBFS and DFS/dyn are unable to solve one of the 50×10 instances within 1 h, the BB-LB1 algorithm leaves the 200×10 instance Ta100 unsolved. It seems that the CBFS and DFS/dyn algorithms, as well as BB-LH are successfully solving Ta100 thanks to the use of a strong initial upper bound obtained by the state-of-the-art Iterated Greedy (IG) algorithm (Ruiz & Stützle, 2007), resp. a local search procedure in Ladhari and Haouari (2005). Indeed, for Ta100, if initialized with the optimal solution, the sequential algorithm requires only 9 node decompositions to prove the optimality of the initial upper bound, while 247×10^6 node decompositions are insufficient to find an optimal solution. In other words, using a better optimal initial upper bound – or parallel tree exploration, as in this work – has a potentially huge impact on the overall computing time required to solve this instance. Besides Ta100, there are few other 10-machine instances which dominate the averaged statistics provided in Table 7. For these outliers, more detailed exploration statistics are provided in Table 8.

In the literature, instances Ta101–120, defined by 200 and 500 jobs and 20 machines are excluded from computational experiments "since they seem currently be out of the reach of exact methods" (Ritt, 2016). As one can see in Table 7 the PBB-LB1/MinBranch, using 32 threads, is capable of solving all 10 instances of this group within less than 10 minutes on average. This includes the resolution of instances Ta112 and Ta116, for which optimal solutions remained unknown up to this date. The permutation schedules of optimal solutions are available at Gmys, Mezmaz, Melab, and Tuyttens (2019) and in Table A.13. Interestingly, for both instances the optimal makespan is equal to the previously best-known *lower* bounds (according to Taillard, 2015). It should be emphasized that all three ingredients discussed in this paper are essential to achieve this: (1) a dynamic branching rule that

Table 7
Average number of solved instances (#solved), avg. elapsed walltime (in seconds) and avg. number of decomposed nodes (NN) for all 12 groups of Taillard's benchmark instances. The initial upper bound obtained by NEH heuristic. Time-limit = 1 hour.

Ta	m	n	Parallel ($p = 32$)			Sequential ($p = 1$)			CBFS(Ritt, 2016)		
			#solved	t (seconds)	NN	#solved	t (seconds)	NN	#solved	t (seconds)	NN
01–10	5	20	10	~ 0.0	2476	10	~ 0.0	4533	10	2.7	193
31–40	5	50	10	~ 0.0	3464	10	~ 0.0	1198	10	7.2	5
61–70	5	100	10	0.04	23×3	10	0.08	11×3	10	12	10
11–20	10	20	10	0.60	4.8×6	10	4.8	4.3×6	10	48	470×3
41–50	10	50	10	2.3	9.8×6	10	39.6	15.5×6	9	508	2.2×6
71–80	10	100	10	0.12	95×3	10	1.3	181×3	10	33	2×3
91–100	10	200	10	0.61	363×3	9	64	5.2×6	10	95	3.5×3
21–30	20	20	10	130	888×6	9	958	462×6	3	2981	5.5×6
51–60	20	50	0	3600	9.7×9	0	3600	746×6	0	3600	2.5×6
81–90	20	100	0	3600	5.1×9	0	3600	362×6	0	3600	0.7×6
101–110	20	200	0	3600	2.9×9	0	3600	215×6	–	–	–
111–120	20	500	10[a)]	498	80×6	0	3600	43.1×6	–	–	–
TOT	–	–	**90**	–	–	**78**	–	–	**72**	–	–

[a]includes resolution of previously unsolved instances `Ta112` and `Ta116`.

Table 8
Exploration statistics for some hard 10-machine Taillard instances (outliers). T_p: resolution time (in seconds) with p threads; NN_p number of decomposed nodes with p threads; NN_{crit}: size of critical tree

Inst.	m	n	T_1	T_{32}	$\frac{T_1}{T_{32}}$	NN_1	NN_{32}	$\frac{NN_1}{NN_{32}}$	NN_{crit}
Ta17	10	20	43	5.3	8	40×6	44×6	0.9	35×6
Ta42	10	50	203	9.2	24	77×6	39×6	1.97	4.2×6
Ta43	10	50	98	2.1	48	41×6	9.4×6	4.36	0.8×6
Ta50	10	50	57	9.3	6	23×6	41×6	0.56	2.1×6
Ta100	10	200	3600	0.7	> 3600	247×6	457×3	540	9

efficiently reduces the size of the explored tree, (2) a computationally efficient replacement for the two-machine bound LB2 whose quadratic cost complexity, per node decomposition, is prohibitive, (3) parallel search for the faster discovery of high-quality solutions and increased processing speed. Although all 500×20 instances could be solved within less than 10 minutes on average, none of the 200×20 was solved even after 1 hour of processing time. However, for some of the 200×20 instances, PBB-LB1 succeeds at least in finding better solutions than the initial NEH schedule – for the 50×20 instances only 2 incumbent schedules have "moved" after 1 hour of processing and for all 100×20 instances the best-found solution after 1 hour of processing is equal to the initial NEH schedule, indicating that those instances are extremely hard to solve.

Nevertheless, removing the time limit, we succeeded in solving the 50×20 instance `Ta56` to optimality. The optimal makespan for `Ta56`, equal to 3679 is known since 2007 (Mezmaz et al., 2007). Using on average 328 CPUs of a computational grid the optimal solution was found and proven optimal after 25 days, cumulating to 22 years of computation time. In the experimental setup of Mezmaz et al. (2007) the algorithm – using dynamic branching (MaxSum) and LB2(W_0) – is initialized with the upper bound 3680, i.e. at the optimal cost plus one unit. Using the same initialization, PBB-LB1 using 32 threads solves `Ta56` to optimality within only 33 hours. Exploration statistics corresponding to the resolution of `Ta56` are provided in Table 9: 332.6×10^9 nodes were decomposed and 14.63×10^{12} lower bounds (LB1) evaluated. From these numbers, we can deduce that an average branched node has 22 children and that $\frac{(14.63\times10^{12}/2)-332.6\times10^9}{(14.63\times10^{12}/2)} \times 100\% = 95.45\%$ of all evaluated subproblems are eliminated during the search. In order to see whether bound refinement is useful for this instance, the same resolution is repeated with PBB-LB12($W_\star$). As shown in Table 9, the resolution time is more than doubled, as only 5.76% of the nodes not pruned by LB1 can be eliminated by the stronger bound,

Table 9
Exploration statistics for the resolution of `Ta56` using PBB-LB1 and PBB-LB12($W_\star$) and 32 threads ($2 \times$ Intel Xeon E5-2630v3@2.40 gigahertz).

	LB1	LB2($W_\star$)
Initial UB	3680	3680
Optimal solution	3679	3679
Time	33 hours 0 minutes 1 second	72 hours 7 minutes 48 seconds
Nodes decomposed	332.6×10^9	313.6×10^9
Avg nodes/second	2.80×10^6	1.21×10^6
LB1 evaluations	14.63×10^{12}	13.86×10^{12}
LB2 evaluations	–	332.8×10^9
Avg depth	28.00	27.91
Avg LB1 pruning rate (%)	95.45	95.47
Avg LB2 pruning rate (%)	–	5.76
Avg LB12 pruning rate (%)	–	95.74

4.6. Performance evaluation using VRF benchmark

VRF_small. In this section we evaluate the performance of our B&B algorithm using the VRF benchmark proposed by Vallada et al. (2015). As mentioned in the introduction, this benchmark is gaining momentum among researchers in the field of optimization methods for scheduling problems and to the best of our knowledge, it has never been used in the context of exact methods. The best-known upper bounds are provided in the original paper introducing the benchmark (Vallada et al., 2015). They were obtained by running two state-of-the-art algorithms, HGA (Ruiz, Maroto, & Alcaraz, 2006) and IG (Ruiz & Stützle, 2007) three times each during $\frac{mn}{2} \times 120$ milliseconds for the `VRF_small` instances (respectively $\frac{mn}{2} \times 90$ milliseconds for the `VRF_large` instances).

Table 10 reports the results obtained by the PBB-LB1 algorithm (32 threads) for the 240 `VRF_small` instances, running also with a time-limit of $\frac{mn}{2} \times 120$ milliseconds and using the NEH schedule as initial upper bound. The first column shows the group of

Table 10
VRF_small. Time-limit: $\frac{nm}{2} \times 120$ milliseconds.

VRF	[opt]	[< UB]	$ARPD_{NEH}$	$ARPD_{UB}$	t(seconds)	VRF	[opt]	[< UB]	$ARPD_{NEH}$	$ARPD_{UB}$	t(seconds)
10_5	10	0	−2.13	0.0	0.03	10_10	10	0	−1.55	0.0	0.04
20_5	10	0	−1.34	0.0	0.006	20_10	9	0	−4.53	0.11	1.9
30_5	10	0	−1.02	0.0	0.004	30_10	8	2	−4.75	0.08	6.0
40_5	10	0	−0.91	0.0	0.004	40_10	8	7	−4.60	−0.02	7.1
50_5	10	0	−0.47	0.0	0.008	50_10	8	8	−4.53	−0.21	7.2
60_5	10	0	−0.85	0.0	0.008	60_10	10	9	−4.42	−0.26	2.1
10_15	10	0	−1.59	0.0	0.05	10_20	10	0	−1.61	0.0	0.03
20_15	6	0	−3.45	0.71	10.6	20_20	1	0	−2.16	1.57	22.9
30_15	0	0	−1.40	4.05	27	30_20	0	0	−0.73	4.95	36
40_15	0	0	−0.72	4.83	36	40_20	0	0	−0.05	5.60	48
50_15	0	0	−1.09	5.54	45	50_20	0	0	−0.01	5.79	60
60_15	0	0	−0.66	5.46	54	60_20	0	0	0	6.43	72

instances in the format n_m, where each group is composed of 10 instances. The next columns show the number of instances solved to optimality ([opt]), the number of instances for which the best-known upper bound (Vallada et al., 2015) is improved ([< UB]), the Average Relative Percentage Deviations (ARPD) with respect to the initial upper bound and with respect to the best-known solution (UB). The table also shows the average elapsed time (in seconds), where the elapsed time for unsolved instances is set to the time-limit of $\frac{mn}{2} \times 120$ milliseconds. Further results with larger time-limits are provided in Appendix A.

For all 60 instances defined by $m = 5$ machines, optimal solutions are found and proven optimal within a few milliseconds and all upper bounds provided in Vallada et al. (2015) are optimal. The same observation can be made for all instances defined by $n = 10$ jobs. Indeed, for such small instances even a complete enumeration of $10! = 3.6 \times 10^6$ solutions requires at most a few seconds of computation.

For instances defined by $m = 10$ machines PBB-LB1 produces excellent results. Within the fixed time-limit budget, 53 of these 60 instances are solved to optimality and improved upper bounds are found for 26 of them, using on average less than half of the allowed computation time.

For the VRF_60_10 group, all ten instances were solved on average within 2.1 seconds, producing an $ARPD_{UB}$ of −0.26% with respect to the best-known upper bound.

For the VRF20_x group the number of solved instances decreases as the number of machines increases and only one VRF20_20 instance can be solved within the allowed time-limit. However, allowing more computation time all VRF20_x instances were solved and for this group all upper bounds provided in Vallada et al. (2015) are proven optimal. None of the instances defined by $m \geq 15$ machines and $n \geq 30$ jobs is solved to optimality and that the algorithm barely improves the initial upper bound.

VRF_large. Using the same experimental setup as for VRF_small, Table 11 provides a summary of the results for the 240 VRF_large instances (initializing the algorithm with the NEH schedule and allowing $\frac{mn}{2} \times 90$ milliseconds of computation time).

For the 80 instances defined by $m = 20$ machines, the best-known upper bound is improved in 42 cases and 38 certificates of optimality are produced. Moreover, it is clear that the number of solved instances increases as the number of jobs grows. Indeed, while for none of the 100_20 instances the algorithm is able to improve the initial upper bound, all ten 700_20 instances are solved to optimality within 303 seconds on average (i. e. less than half of the allowed 630 seconds) and the ARPD with respect to the previously best-known solutions is −0.28%. This result does not only demonstrate the excellent performance of exact methods for this class of instances, it also shows that the HGA and IG methods

Table 11
VRF_large. Time-limit: $\frac{nm}{2} \times 90$ milliseconds.

VRF	[opt]	[< UB]	$ARPD_{NEH}$	$ARPD_{UB}$	t(seconds)	TL
100_20	0	0	0	5.68	90	90
200_20	0	0	−0.12	4.23	180	180
300_20	2	2	−1.35	1.62	249	270
400_20	5	6	−2.48	−0.14	293	360
500_20	6	7	−2.00	−0.15	314	450
600_20	7	8	−1.84	−0.21	281	540
700_20	10	10	−1.72	−0.28	303	630
800_20	8	9	−1.30	−0.10	415	720
x_20	**38**	**42**	**−1.35**	**1.33**	**265**	–
x_40	0	0	0	NEH	t/o	–
x_60	0	0	0	NEH	t/o	–

produce near-optimal solutions with an average optimality gap of 0.28%. For the 20-machine instances, more detailed results and improved upper bounds obtained by runs with more computing time are given in Appendix A and permutation schedules for all VRF instances solved to optimality for the first time are made available at Gmys et al. (2019).

For the 160 instances defined by $m = 40$ and 60 machines we did not break down the results into classes, because the outcome of all runs is the same: the algorithm runs out of time and is unable to improve the initial upper bound. The quality of (all available) lower bounds degrades as the number of machines m increases. As a consequence, the algorithm is unable to eliminate subproblems early enough and performs (almost) complete enumerations of large portions of the search space. The same is observed when the algorithm is initialized with the best-known upper bound, i.e. *none* of the 160 best-known solutions is improved or proven optimal.

5. Conclusions and future works

In this paper, we have presented a branch-and-bound (B&B) algorithm for the permutation flowshop scheduling problem (PFSP) with makespan objective. We have compared a total of 45 combinations of branching rules and lower bounds – including a new variant of the well-known two-machine lower bound (LB2), which reduces its computational requirements by predicting promising subsets of machine-pairs based on previous evaluations.

Through a series of computational experiments we have determined that the best overall performance is achieved by a B&B algorithm that combines dynamic branching with a simple one-machine lower bound (LB1). The latter is the least accurate of the considered bounds, but it also has the lowest computational requirements. This combination of dynamic branching with LB1

(introduced over 50 years ago) has not been considered in the literature up to this date.

Experimental results show that this algorithm outperforms current state-of-the-art B&B methods, solving more benchmark instances to optimality (within the same time-limit) and consuming significantly less computation time. In particular, we report two new optimal solutions for previously unsolved instances from Taillard's benchmark (`Ta112` and `Ta116`), defined by 500 jobs and 20 machines. For the first time since its introduction, the VRF (Vallada et al., 2015) benchmark is considered in the context of an exact optimization method. For the 480 VRF instances, a total of 134 of the best-known upper bound are proven optimal and an additional 89 are improved, including proof of optimality for 74 of those.

We have reviewed, revisited and experimentally investigated three key components of the algorithm: the branching rule, lower bounds (LB) and parallelization of the search. Moreover, using a large number of benchmark instances defined by up to 800 jobs has revealed the instance-dependent behavior of the algorithm. In general, empirical evidence indicates that the number of jobs (n) and the number of machines (m) influence the hardness (for B&B) of a PFSP instance in three ways: (1) obviously, n determines the size of the complete search space, which is manageable if n is small, say $n = 10 - 15$, (2) the quality of all known lower bounds degrades quickly as m becomes large ($m > 20$), causing the algorithm to perform almost complete enumerations of large parts of the search space and (3) for fixed m, as n becomes very large ($n \gtrapprox 300$), lower bounds tend to be equal to the optimal makespan with probability 1.

The excellent performance of B&B for certain groups of instances, and its complete failure to produce any good solution for others, suggests that the combination/integration of exact methods with approximate approaches is a promising research direction. This work is a first step in this direction, as it allows to better understand for what types of (sub)problems a (carefully designed) B&B algorithm can be efficient. Experiments have shown that the best choices for different components of the algorithm (e. g. lower bound, branching rule, criteria for prioritizing sibling nodes) are strongly instance-dependent. Therefore, we plan to investigate the auto-tuning of the algorithm's parameters based on partial exploration statistics (e. g. pruning rates, tree size estimation).

Appendix A. Improved upper bounds

Table A12
Improved upper bounds for VRF benchmark (Vallada et al., 2015). A "⋆" indicates that the makespan is proven optimal. Permutation sequences for instances solved to optimality are available at Gmys et al. (2019).

VRFn_m_k	UB [VRF]	Best [B&B]	gap (%)	VRFn_m_k	UB [VRF]	Best [B&B]	gap (%)	VRFn_m_k	UB [VRF]	Best [B&B]	gap (%)
VRF30_10_1	1944	1943⋆	−0.05	VRF40_10_1	2480	2477⋆	−0.12	VRF50_10_1	2926	2912⋆	-0.48
VRF30_10_2	2098	2098⋆	0	VRF40_10_2	2444	2431⋆	−0.53	VRF50_10_2	3035	3027⋆	−0.26
VRF30_10_3	2077	2077⋆	0	VRF40_10_3	2412	2400⋆	−0.50	VRF50_10_3	3019	3002⋆	−0.56
VRF30_10_4	1945	1945⋆	0	VRF40_10_4	2472	2468⋆	−0.16	VRF50_10_4	3003	2997⋆	−0.20
VRF30_10_5	2023	2023⋆	0	VRF40_10_5	2425	2420⋆	−0.21	VRF50_10_5	3252	3250⋆	−0.06
VRF30_10_6	2043	2043⋆	0	VRF40_10_6	2547	2541⋆	−0.24	VRF50_10_6	3149	3149⋆	0
VRF30_10_7	1967	1967⋆	0	VRF40_10_7	2501	2498⋆	−0.12	VRF50_10_7	2842	2835⋆	−0.25
VRF30_10_8	1896	1896⋆	0	VRF40_10_8	2491	2488⋆	−0.12	VRF50_10_8	3072	3055⋆	−0.55
VRF30_10_9	1908	1908⋆	0	VRF40_10_9	2411	2411⋆	0	VRF50_10_9	3022	2999⋆	−0.76
VRF30_10_10	1915	1912⋆	−0.16	VRF40_10_10	2478	2478⋆	0	VRF50_10_10	3056	3050⋆	−0.20
VRF60_10_1	3435	3415⋆	−0.58	VRF60_10_5	3505	3496⋆	−0.26	VRF60_10_9	3685	3677⋆	−0.22
VRF60_10_2	3655	3640⋆	−0.41	VRF60_10_6	3594	3586⋆	−0.22	VRF60_10_10	3492	3486⋆	−0.17
VRF60_10_3	3423	3415⋆	−0.23	VRF60_10_7	3654	3644⋆	−0.27	VRF30_15_8	2366	2366⋆	0
VRF60_10_4	3455	3448⋆	−0.20	VRF60_10_8	3552	3522⋆	0	VRF30_15_10	2385	2385⋆	0
VRF200_20_2	11265	11264	−0.009	VRF200_20_5	11208	11207	−0.009	VRF200_20_8	11141	11137	−0.04
VRF300_20_1	16149	16089	−0.37	VRF400_20_1	21120	21042	−0.37	VRF500_20_1	26411	26253⋆	−0.60
VRF300_20_2	16512			VRF400_20_2	21457	21346⋆	−0.52	VRF500_20_2	26681	26555⋆	−0.47
VRF300_20_3	16173	16167	−0.04	VRF400_20_3	21441	21380	−0.28	VRF500_20_3	26409	26268⋆	−0.53
VRF300_20_4	16181	16172	−0.06	VRF400_20_4	21247	21200	−0.22	VRF500_20_4	26124	25994⋆	−0.50
VRF300_20_5	16342	16283	−0.36	VRF400_20_5	21553	21399⋆	−0.71	VRF500_20_5	26781	26703⋆	−0.29
VRF300_20_6	16137	16021⋆	−0.72	VRF400_20_6	21214	21134	−0.38	VRF500_20_6	26443	26325⋆	−0.45
VRF300_20_7	16266	16260	−0.04	VRF400_20_7	21625	21507⋆	−0.55	VRF500_20_7	26433	26313⋆	−0.45
VRF300_20_8	16416	16409	−0.04	VRF400_20_8	21277	21198⋆	−0.37	VRF500_20_8	26318	26217⋆	−0.38
VRF300_20_9	16376			VRF400_20_9	21346	21236⋆	−0.52	VRF500_20_9	26442	26345⋆	−0.37
VRF300_20_10	16899	16780⋆	−0.70	VRF400_20_10	21538	21456⋆	−0.38	VRF500_20_9	26442	26345⋆	−0.37
VRF600_20_1	31433	31303⋆	−0.41	VRF700_20_1	36394	36285⋆	−0.30	VRF800_20_1	41558	41413⋆	−0.35
VRF600_20_2	31418	31281⋆	−0.44	VRF700_20_2	36337	36220⋆	−0.32	VRF800_20_2	41407	41282⋆	−0.30
VRF600_20_3	31429	31374⋆	−0.17	VRF700_20_3	36568	36419⋆	−0.41	VRF800_20_3	41425	41319⋆	−0.26
VRF600_20_4	31547	31417⋆	−0.41	VRF700_20_4	36452	36361⋆	−0.25	VRF800_20_4	41426	41375⋆	−0.12
VRF600_20_5	31448	31354	−0.30	VRF700_20_5	36584	36496⋆	−0.24	VRF800_20_5	41710	41626⋆	−0.20
VRF600_20_6	31717	31613⋆	−0.33	VRF700_20_6	36671	36556⋆	−0.31	VRF800_20_6	42010	41919⋆	−0.21
VRF600_20_7	31527	31461⋆	−0.21	VRF700_20_7	36624	36540⋆	−0.23	VRF800_20_7	41425	41419	−0.01
VRF600_20_8	31564	31414⋆	−0.48	VRF700_20_8	36522	36418⋆	−0.28	VRF800_20_8	41492	41390⋆	−0.25
VRF600_20_9	31577	31473⋆	−0.33	VRF700_20_9	36329	36212⋆	−0.32	VRF800_20_9	41796	41697⋆	−0.24
VRF600_20_10	31130	31021⋆	−0.35	VRF700_20_10	36417	36362⋆	−0.15	VRF800_20_10	41574	41489⋆	−0.20

Table A13
Optimal makespan (C_{max}) and optimal permutation schedules for Ta112 and Ta116.

Ta	C_{max}	Permutation
112	26500	57 498 446 343 351 253 48 423 230 196 442 471 308 282 403 145 244 47 258 76 490 45 488 101 222 219 364 134 169 140 62 50 318 40 412 457 322 348 8 103 358 311 410 106 224 350 74 335 440 331 472 32 171 324 202 160 135 139 213 193 97 353 422 177 150 49 355 466 204 276 143 125 251 313 19 367 189 207 483 203 180 289 256 390 65 267 80 179 178 211 232 415 186 453 454 269 448 70 81 333 363 431 395 181 273 360 377 388 449 12 109 133 92 279 126 15 500 164 428 66 120 42 357 399 141 117 365 155 39 475 340 90 421 492 278 404 216 334 146 315 297 41 187 370 24 346 118 309 79 274 361 277 407 151 165 242 470 487 185 349 88 426 288 249 22 389 481 4 345 344 303 329 463 300 227 292 158 214 100 469 424 104 6 128 94 310 380 337 59 374 119 200 317 379 275 460 325 23 455 182 383 188 217 497 115 37 154 192 152 226 366 312 459 137 326 494 54 491 476 239 183 167 166 96 156 352 439 208 394 417 247 33 13 209 2 243 123 342 489 266 195 384 356 283 427 240 16 477 107 14 72 372 25 265 304 286 46 210 259 124 52 264 480 10 11 468 85 58 112 236 499 245 400 228 458 250 319 241 419 321 172 495 295 479 113 385 443 131 129 328 235 445 98 425 381 486 30 450 201 89 301 339 68 234 467 402 257 9 320 105 397 462 255 438 314 199 3 341 223 387 170 220 17 272 31 436 231 142 302 136 111 306 153 451 263 87 418 359 369 478 159 281 396 437 157 67 298 221 1 91 405 7 198 354 229 413 95 398 336 132 392 408 246 299 434 55 18 190 296 347 130 260 71 60 194 174 414 441 69 330 237 5 474 270 148 114 102 386 268 285 86 456 376 294 206 496 391 323 485 61 191 122 375 411 465 368 218 176 212 248 149 409 327 138 287 284 401 78 254 406 444 77 173 233 127 51 473 43 116 464 110 73 168 393 493 373 378 147 108 21 29 64 83 20 416 271 293 382 121 262 429 435 371 44 56 84 99 175 38 484 162 447 225 290 280 161 215 144 163 291 307 205 238 35 27 28 26 461 36 433 452 53 305 184 430 197 252 34 332 482 75 338 82 362 432 261 420 93 316 63
116	26469	80 495 259 83 89 377 464 24 218 268 393 13 14 193 285 291 273 357 175 277 138 489 274 207 337 26 131 39 147 84 230 91 471 412 321 168 332 312 17 348 20 431 369 203 320 490 424 479 296 450 326 62 141 313 154 133 206 152 400 58 441 197 468 340 174 216 417 430 19 421 234 23 38 456 258 475 183 37 148 368 330 325 178 156 77 25 43 72 244 224 447 134 161 198 150 423 118 283 185 438 395 196 391 280 263 143 140 116 57 4 236 427 467 86 102 451 128 302 176 300 404 465 61 225 278 119 2 10 12 180 170 205 276 231 264 487 94 440 362 220 472 297 28 392 383 33 69 415 469 54 107 394 439 237 100 112 30 388 65 125 47 305 358 402 162 486 353 7 109 306 454 64 50 255 361 462 22 366 101 106 322 425 49 444 314 372 350 370 42 386 385 378 157 88 105 164 68 275 407 498 166 411 349 139 124 387 293 239 173 127 21 44 189 250 466 453 373 334 481 81 265 201 352 303 15 292 93 270 409 345 181 474 408 494 436 74 311 222 136 290 190 379 376 226 48 59 221 449 266 228 493 90 187 318 418 113 242 435 267 27 87 96 211 146 126 8 227 271 308 132 354 121 272 442 375 145 165 36 339 478 195 66 347 249 55 289 397 79 374 335 382 364 499 122 389 199 200 243 261 233 500 34 344 406 360 184 76 223 414 213 99 319 212 142 460 144 51 483 3 208 159 338 257 95 32 35 110 97 309 473 137 11 288 194 171 341 359 104 75 426 192 179 432 248 115 204 371 461 363 215 232 158 29 365 71 480 299 31 476 410 235 342 129 256 323 459 446 202 384 103 182 343 163 191 433 85 420 307 488 186 497 333 381 135 245 484 251 331 214 217 458 485 1 336 452 401 82 428 98 219 169 390 328 281 210 482 56 172 399 247 315 279 287 5 130 151 422 153 269 240 491 238 434 443 160 70 241 40 111 284 46 298 155 294 437 496 41 16 114 301 470 316 6 477 463 367 108 286 246 416 253 380 355 351 52 329 282 45 304 63 188 229 445 317 403 78 18 73 324 262 398 120 92 492 396 448 252 356 60 53 177 209 405 167 9 413 327 149 457 260 67 455 117 310 429 295 419 346 123 254

References

Baker, K. R. (1975). A comparative study of flow-shop algorithms. *Operations Research, 23*(1), 62–73. doi:10.1287/opre.23.1.62.

Bendjoudi, A., Melab, N., & Talbi, E.-G. (2012). Hierarchical branch and bound algorithm for computational grids. *Future Generation Computer Systems, 28*(8), 1168–1176. doi:10.1016/j.future.2012.03.001.

Brown, A. P. G., & Lomnicki, Z. A. (1966). Some applications of the "branch-and-bound" algorithm to the machine scheduling problem. *Journal of the Operational Research Society, 17*(2), 173–186. doi:10.1057/jors.1966.25.

de Bruin, A., Kindervater, G. A. P., & Trienekens, H. W. J. M. (1995). Asynchronous parallel branch and bound and anomalies. In A. Ferreira, & J. Rolim (Eds.), *Parallel algorithms for irregularly structured problems* (pp. 363–377). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/3-540-60321-2_29.

Carlier, J., & Rebaï, I. (1996). Two branch and bound algorithms for the permutation flow shop problem. *European Journal of Operational Research, 90*(2), 238–251. doi:10.1016/0377-2217(95)00352-5.

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*(12), 1563–1577. doi:10.1016/j.jpdc.2013.07.023.

Cheng, J., Kise, H., & Matsumoto, H. (1997). A branch-and-bound algorithm with fuzzy inference for a permutation flowshop scheduling problem. *European Journal of Operational Research, 96*(3), 578–590. doi:10.1016/S0377-2217(96)00083-5.

Cheng, J., Kise, H., Steiner, G., & Stephenson, P. (2003). *Branch-and-bound algorithms using fuzzy heuristics for solving large-scale flow-shop scheduling problems*. In J.-L. Verdegay (Ed.) (pp. 21–35)). Berlin, Heidelberg: Springer Berlin Heidelberg.

Companys, R., & Mateo, M. (2007). Different behaviour of a double branch-and-bound algorithm on FM|PRMU|CMAX and FM|block|CMAX problems. *Computers and Operations Research, 34*(4), 938–953. doi:10.1016/j.cor.2005.05.018.

Daouri, M., Escobar, F. A., Xin Chang, & Valderrama, C. (2015). A hardware architecture for the branch and bound flow-shop scheduling algorithm. In *Proceedings of the 2015 Norchip international symposium on system-on-chip (SOC) nordic circuits and systems conference (NORCAS):* (pp. 1–4). doi:10.1109/NORCHIP.2015.7364362.

Drozdowski, M., Marciniak, P., Pawlak, G., & Plaza, M. (2011). Grid branch-and-bound for permutation flowshop. In R. Wyrzykowski, J. J. Dongarra, K. Karczewski, & J. Wasniewski (Eds.), *Proceedings of the 9th international conference parallel processing and applied mathematics, PPAM 2011, Torun, Poland, September 11-14, 2011. revised selected papers, part II*. In *Lecture Notes in Computer Science: 7204* (pp. 21–30). Springer. doi:10.1007/978-3-642-31500-8_3.

Dubois-Lacoste, J., Pagnozzi, F., & Stützle, T. (2017). An iterated greedy algorithm with optimization of partial solutions for the makespan permutation flowshop problem. *Computers and Operations Research, 81*, 160–166. doi:10.1016/j.cor.2016.12.021.

Fernandez-Viagas, V., Ruiz, R., & Framinan, J. M. (2017). A new vision of approximate methods for the permutation flowshop to minimise makespan: State-of-the-art and computational evaluation. *European Journal of Operational Research, 257*(3), 707–721. doi:10.1016/j.ejor.2016.09.055.

Framinan, J. M., Gupta, J. N. D., & Leisten, R. (2004). A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *Journal of the Operational Research Society, 55*(12), 1243–1255. doi:10.1057/palgrave.jors.2601784.

Garey, M. R., Johnson, D. S., & Sethi, R. (1976). The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research, 1*(2), pp.117–129. www.jstor.org/stable/3689278.

Giles, M., & Reguly, I. (2014). Trends in high-performance computing for engineering calculations. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 372*(2022), 20130319. doi:10.1098/rsta.2013.0319.

Gmys, J., Leroy, R., Mezmaz, M., Melab, N., & Tuyttens, D. (2016a). Work stealing with private integer vector matrix data structure for multi-core branch-and-bound algorithms. *Concurrency and Computation: Practice and Experience, 28*(18), 4463–4484. doi:10.1002/cpe.3771.

Gmys, J., Mezmaz, M., Melab, N., & Tuyttens, D. (2016b). A gpu-based branch-and-bound algorithm using integer-vector-matrix data structure. *Parallel Computing, 59*, 119–139. doi:10.1016/j.parco.2016.01.008.

Gmys, J., Mezmaz, M., Melab, N., & Tuyttens, D. (2019). Improved upper bounds for permutation flowshop scheduling benchmarks (Taillard and VRF). https://doi.org/10.5281/zenodo.3550553.

Hejazi, S. R., & Saghafian, S. (2005). Flowshop-scheduling problems with makespan criterion: a review. *International Journal of Production Research, 43*(14), 2895–2929. doi:10.1080/00207540500056417.

Ignall, E., & Schrage, L. (1965). Application of the branch and bound technique to some flow-shop scheduling problems. *Operations Research, 13*(3), 400–412. doi:10.1287/opre.13.3.400.

Jin, Y. (2011). Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation, 1*(2), 61–70.

Johnson, S. M. (1954). Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly, 1*(1), 61–68. doi:10.1002/nav.3800010110.

Kalczynski, P. J., & Kamburowski, J. (2009). An empirical analysis of the optimality rate of flow shop heuristics. *European Journal of Operational Research, 198*(1), 93–101. doi:10.1016/j.ejor.2008.08.021.

Knuth, D. (1997). The art of computer programming, Volume 2. *Reading, MA*, 192. ISBN=9780201896848.

Ladhari, T., & Haouari, M. (2005). A computational study of the permutation flow shop problem based on a tight lower bound. *Computers and Operations Research, 32*(7), 1831–1847. doi:10.1016/j.cor.2003.12.001.

Lageweg, B. J., Lenstra, J. K., & Kan, A. H. G. R. (1978). A general bounding scheme for the permutation flow-shop problem. *Operations Research, 26*(1), 53–67. doi:10.1287/opre.26.1.53.

Lemesre, J., Dhaenens, C., & Talbi, E. (2007). An exact parallel method for a bi-objective permutation flowshop problem. *European Journal of Operational Research, 177*(3), 1641–1655. doi:10.1016/j.ejor.2005.10.011.

Li, G., & Wah, B. W. (1986). Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Transactions on Computers, C-35*(6), 568–573. doi:10.1109/TC.1986.5009434.

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. doi:10.1016/j.ijpe.2017.06.026.

Lomnicki, Z. A. (1965). A "branch-and-bound" algorithm for the exact solution of the three-machine scheduling problem. *Journal of the Operational Research Society, 16*(1), 89–100. doi:10.1057/jors.1965.7.

McMahon, G. B., & Burton, P. G. (1967). Flow-shop scheduling with the branch-and-bound method. *Operations Research, 15*(3), 473–481. http://www.jstor.org/stable/168456.

Melab, N., Gmys, J., Mezmaz, M., & Tuyttens, D. (2018). Multi-core versus many-core computing for many-task branch-and-bound applied to big optimization problems. *Future Generation Computer Systems, 82*, 472–481. doi:10.1016/j.future.2016.12.039.

Mezmaz, M., Leroy, R., Melab, N., & Tuyttens, D. (2014a). A multi-core parallel branch-and-bound algorithm using factorial number system. In *Proceedings of the 2014 IEEE 28th international parallel and distributed processing symposium* (pp. 1203–1212). doi:10.1109/IPDPS.2014.124.

Mezmaz, M., Leroy, R., Melab, N., & Tuyttens, D. (2014b). A multi-core parallel branch-and-bound algorithm using factorial number system. In *Proceedings of the 2014 IEEE 28th international parallel and distributed processing symposium* (pp. 1203–1212). doi:10.1109/IPDPS.2014.124.

Mezmaz, M., Melab, N., & Talbi, E. G. (2007). A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. In *Proceedings of the 2007 IEEE international parallel and distributed processing symposium, Long Beach, CA* (pp. 1–9).

Nabeshima, I. (1967). On bound of makespans and its application in m machine scheduling problem. *Journal of the Operations Research Society of Japan, 9*(3-4), 98–+.

Nawaz, M., Enscore, E. E., & Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega, 11*(1), 91–95. doi:10.1016/0305-0483(83)90088-9.

Potts, C. (1980). An adaptive branching rule for the permutation flow-shop problem. *European Journal of Operational Research, 5*(1), 19–25. doi:10.1016/0377-2217(80)90069-7.

Potts, C. N., & Strusevich, V. A. (2009). Fifty years of scheduling: a survey of milestones. *Journal of the Operational Research Society, 60*(sup1), S41–S68. doi:10.1057/jors.2009.2.

Ritt, M. (2016). A branch-and-bound algorithm with cyclic best-first search for the permutation flow shop scheduling problem. In *Proceedings of the IEEE international conference on automation science and engineering, CASE 2016, Fort Worth, TX, USA, August 21–25, 2016* (pp. 872–877). IEEE. doi:10.1109/COASE.2016.7743493.

Rossi, F. L., Nagano, M. S., & Neto, R. F. T. (2016). Evaluation of high performance constructive heuristics for the flow shop with makespan minimization. *The International Journal of Advanced Manufacturing Technology, 87*(1), 125–136. doi:10.1007/s00170-016-8484-9.

Ruiz, R., Maroto, C., & Alcaraz, J. (2006). Two new robust genetic algorithms for the flowshop scheduling problem. *Omega, 34*(5), 461–476. doi:10.1016/j.omega.2004.12.006.

Ruiz, R., & Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research, 177*(3), 2033–2049. doi:10.1016/j.ejor.2005.12.009.

Sutton, R. S., Barto, A. G., et al. (1998). *Introduction to reinforcement learning*: 135. Cambridge: MIT Press.

Szwarc, W. (1973). Optimal elimination methods in the m × n flow-shop scheduling problem. *Operations Research, 21*(6), 1250–1259. doi:10.1287/opre.21.6.1250.

Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research, 64*(2), 278–285. doi:10.1016/0377-2217(93)90182-M.

Taillard, E. (2015). Flow shop sequencing: Summary of best known lower and upper bounds of Taillard's instances. http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/flowshop.dir/best_lb_up.txt.

Vallada, E., Ruiz, R., & Framinan, J. M. (2015). New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research, 240*(3), 666–677. doi:10.1016/j.ejor.2014.07.033.

Vu, T.-T., & Derbel, B. (2016). Parallel branch-and-bound in multi-core multi-cpu multi-GPU heterogeneous environments. *Future Generation Computer Systems, 56*, 95–109. doi:10.1016/j.future.2015.10.009.