Branch-and-Bound Algorithms as Polynomial-time Approximation Schemes

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

Branch-and-bound algorithms (B&B) and polynomial-time approximation schemes (PTAS) are two seemingly distant areas of combinatorial optimization. We intend to (partially) bridge the gap between them while expanding the boundary of theoretical knowledge on the B&B framework. Branch-and-bound algorithms typically guarantee that an optimal solution is eventually found. However, we show that the standard implementation of branch-and-bound for certain knapsack and scheduling problems also exhibits PTAS-like behavior, yielding increasingly better solutions within polynomial time. Our findings are supported by computational experiments and comparisons with benchmark methods.

2012 ACM Subject Classification Theory of computation → Branch-and-bound; Theory of computation → Numeric approximation algorithms; Theory of computation → Scheduling algorithms

Keywords and phrases Branch-and-bound algorithm, Polynomial-time approximation scheme, Parallel machine scheduling problem, Knapsack problem

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Branch-and-bound algorithms (B&B) have been a central part of combinatorial optimization for quite some time. They serve as a go-to method for several optimization problems both in theory and in practice, and many NP-hard optimization problems are still frequently attacked by a variant of the branch-and-bound method or solvers having it at their core. Their great success could partially be attributed to the generality of the framework, which allows it to be applied to fundamentally different problems; along with the flexibility of choosing parameters such as the branching rule or the search strategy, resulting in an extensive list of options for exploiting the same underlying principles of the framework. For an introduction of the idea and a thorough description, we refer to Kohler and Steiglitz [19]; whereas a survey of recent advancements regarding best practices of parameter tuning can be found in [18].

Broadly speaking, the B&B framework consists of iteratively refining a partition of the search space of a given combinatorial optimization problem. The process is commonly depicted with the help of an auxiliary tree, where the leaves of the tree (often called *active nodes*) correspond to the currently considered partition classes. In this context, refining the partition by further dividing one class equates to creating child nodes for the corresponding leaf in the tree. In addition, each node in the tree has an attributed lower or upper bound (depending on the type of problem in consideration) on the objective value of the best solution in that particular partition class; typically they are calculated from a linear relaxation of the integer formulation of the problem. Occasionally, a node can be discarded from the search process if its attributed bound is worse than an already found solution.

It is evident from the above description that the framework has various degrees of freedom. The key components that can be adjusted independently are the following:

- **Branching**: The process of dividing a problem into sub-problems.
- **Bounding**: Calculating lower/upper bounds to limit the search space.
- **Selection/Search**: Choosing the next sub-problem (active node) to explore based on some kind of ranking.

The choice of these parameters could of course be influenced by the underlying problem; When it comes to traversing the branching tree, certain heuristics might be fixed a priori that do not depend on the nature of the problem. Common search heuristics include breadth-first search (BFS), depth-first search (DFS), or best-first search; the latter of which chooses the active subproblem with the best attributed lower/upper bound.

Several attempts have been made to analyze the effect of preferring one selection strategy to another. For instance, Greenberg and Hegerich [14] compare the DFS and the best-first methods applied to the knapsack problem. Moreover, with the increasing interest in applications of artificial intelligence, researchers have even deployed machine learning-based methods that try to *learn* an optimal selection strategy (See, e.g., [7]. See [1] for a recent survey). However, these comparisons either rely on empirical evidence and rank strategies according to their practical performance or argue intuitively about the dominance of one method from a certain aspect: DFS is often considered the memory-efficient alternative, whereas the best-first search is often regarded as the "intuitive" best choice. To the best of our knowledge, little to no effort has been made to justify why one strategy outperforms another, given the problem type.

Meaningful applications of the B&B framework mostly revolve around NP-hard optimization problems, where a standard worst-case analysis does not illuminate the true power of algorithms. Instead, recent results focus on the average-case analysis, and study the performance on randomly sampled instances. Pataki, Tural, and Wong [27] analyze the complexity of B&B for the integer feasibility problem. They show that if the magnitude of the coefficients in the constraint matrix is sufficiently large, then, up to a reformulation technique, almost all instances can be solved directly at the root node. Recently, [10] show that, with any branching rule and best-first as node selection, B&B reaches the optimum with a polynomial number of nodes on randomly sampled instances, for a fixed number of constraints. Some "negative" results have also been proposed. Dash [9] showed an exponential lower bound on Branch and Cut for 0-1 integer programming when just a few families of cuts are enforced. Bell and Frieze [2] show that any B&B method for the Asymmetric Traveling Salesman Problem via the assignment problem relaxation has an exponential number of nodes. More recently, [11] showed that for the Vertex Cover problem, choosing full strong branching as the variable selection rule can either perform exceptionally well or be exponentially worse than any other rule, depending on the class of instances considered.

The efficiency of B&B appears to be strongly dependent on the problem at hand, as well as the choice of lower bound, branching rule, and node selection strategy. This makes it particularly interesting to investigate for which problems, under what conditions, and with which specific choices B&B can be made to run in polynomial time.

In our current work, we endeavor to explain the intuitive advantage of the best-first search strategy by giving a worst-case theoretical analysis from a slightly unusual point of view. Namely, we will show that for the makespan minimization of unrelated parallel machine scheduling problem with a fixed number of machines (denoted by $Rm||C_{max}$ following standard three-field representation; see [5]) and the multiple knapsack problem, the best-first strategy paired with other natural linear programming-based branching and bounding

strategies yields a polynomial-time approximation scheme. Thus, practical observations regarding its superiority are strengthened by the guarantee that it is able to find fast a solution *arbitrarily close to the optimum*.

Our contributions are as follows: first (Section 2), we consider a family of branch-and-bound algorithms with the best-first tree traversal rule for the multiple knapsack problem, and show that they form a polynomial-time approximation scheme, in which their execution time is constrained to be within polynomial bounds for any fixed approximation ratio. The family A_α^{knap} (parametrized by the approximation parameter α) relies on a linear programming-based upper bound and a branching rule that exploits the specific structure of the linear program (see Proposition 1).

▶ **Theorem 1.** *For every fixed $0 < \alpha < 1$, the algorithm A_α^{knap} returns an α-approximate solution to the multiple knapsack problem, after processing $O(n^{c_\alpha+1} \cdot m^{c_\alpha})$-many nodes in the branching tree for some constant c_α that depends on α.*

Based on the same underlying idea, we provide (Section 3) similar results for the makespan minimization of unrelated parallel machine scheduling problem *with a fixed number of machines*, in which we prove that a certain B&B algorithm is an efficient polynomial-time approximation scheme (EPTAS[1]) for $Rm||C_{max}$. The family $A_\epsilon^{\text{unrel}}$ again relies on a linear programming relaxation and its approximation property. The following theorem is proven in Appendix 7.1 due to space limitations, and easily implies a polynomial running time for any fixed error.

▶ **Theorem 2.** *For every fixed $\epsilon > 0$, the algorithm A_ϵ^{unrel} returns a $(1+\epsilon)$-approximate solution to the unrelated machine scheduling problem, after processing at most $m^{\lfloor \frac{m^2}{\epsilon} \rfloor}$-many nodes in the branching tree.*

Exploration and exploitation are two fundamental concepts in search and optimization algorithms. Exploration searches diverse areas, while exploitation refines known good solutions for efficiency. A balance is crucial in algorithms like branch-and-bound to prevent slow convergence. In many branching points of the algorithm, there are decision ambiguities; meaning that (almost) indistinguishable subproblems keep reappearing in the process. In Section 4, we demonstrate that if two nodes represent "similar" situations, it is not necessary to explore all such "similar" cases to obtain an approximate solution. Given a polynomial bound on the running time, we can put on hold the exploration of some nodes while prioritizing others that are not similar to already explored sub-problems, resulting in an improved exploration of the search space. Standard rounding techniques can be used to identify "similar nodes". For the special case of the *uniform machine scheduling* problem with fixed number m of machines (denoted by $Qm||C_{max}$, see [5]), this enhances the diversification of the best-first search, resulting in better exploration of the search space and the achievement of a fully polynomial-time approximation scheme (FPTAS). We will denote the enhanced algorithm by $A_\epsilon^{\text{sim-prof}}$ and prove the following theorem in Appendix 7.2:

▶ **Theorem 3.** *For every fixed $\epsilon > 0$, the algorithm $A_\epsilon^{sim\text{-}prof}$ returns a $(1+\epsilon)$ - approximate solution to the uniform machine scheduling problem, after processing at most $n \cdot \left(\frac{5n}{\epsilon}\right)^m$ nodes in the branching tree.*

[1] A scheme is called EPTAS when, for an arbitrary ϵ and inputs of size n, its running time is $O(n^c \cdot f(1/\epsilon))$ for some constant c that is independent from ϵ.

To prove these results, we provide structural properties of the vertices of the corresponding polyhedra in Lemma 6. We continue with computational experiments supporting our theoretical results in Section 5. Finally, with the hope of fueling future research, we list (Section 6) the fundamental properties of optimization problems that made our approach applicable. Proofs and auxiliary lemmas are presented in Appendix 7.

2 A B&B PTAS for the Multi-Knapsack Problem

In the one-dimensional $0-1$ multiple knapsack problem (referred to as *multiple knapsack problem* or *multi-knapsack problem*), we are given n items characterized by weights $\boldsymbol{w} = (w_1, \ldots, w_n)$ and profits $\boldsymbol{p} = (p_1, \ldots, p_n)$. Additionally, we have m knapsacks with capacities $\boldsymbol{C} = (C_1, \ldots, C_m)$, where m is a fixed constant. The goal is to select a subset of items to maximize the total profit while ensuring that the weight constraints of each knapsack are satisfied. We assume that all weights, profits, and knapsack capacities are nonnegative integers.

When m is fixed, that is the case we are considering in this paper, the problem is weakly NP-hard and admits an FPTAS [17, 21]. For reference on a comprehensive theory of the problem, we mention the Martello-Toth book [24]. A survey on recent improvements can be found in [4].

The use of branch-and-bound algorithms for the knapsack problem is well-established. Notable early contributions include those by Kolesar [20], Greenberg and Hegerich [14], and Horowitz and Sahni [15] for the single-knapsack case. These approaches often leverage various heuristics, such as DFS or BFS search strategies and fractional-item pivot selection rules. However, most of these results focus primarily on computational experiments, with little emphasis on formal theoretical guarantees.

In this work, we demonstrate that a "standard" branch-and-bound implementation for the multi-knapsack problem naturally yields a PTAS. Specifically, at each node, the **bounding** step involves solving the linear programming relaxation known as *surrogate relaxation* (see [24], Chapter 6). We then apply the standard **rounding** technique of Dantzig [8] to obtain an $(m+1)$-approximate feasible solution, and **branch** according to the most profitable fractional item. The **selection** strategy is the best-first rule, where we choose the node to be processed next whose upper bound (the fractional optimum) is the greatest; we will denote this strategy by GUB in the experimental section. We terminate whenever the ratio between the global upper bound and the best integer solution reaches or goes above a fixed constant $\alpha \in (0, 1)$. The full details of our algorithm, A_α^{knap}, are provided in Section 2.1, along with a proof of the following result:

▶ **Theorem 1.** *For every fixed $0 < \alpha < 1$, the algorithm A_α^{knap} returns an α-approximate solution to the multiple knapsack problem, after processing $O(n^{c_\alpha+1} \cdot m^{c_\alpha})$-many nodes in the branching tree for some constant c_α that depends on α.*

2.1 Proof of Theorem 1

In this section, we demonstrate that a "standard" branch-and-bound implementation for the multi-knapsack problem naturally yields a PTAS. We will need the standard integer programming formulation of the problem:

$$MK_m(\boldsymbol{C}, \boldsymbol{w}, \boldsymbol{p}) : \max \sum_{j=1}^{n} \sum_{i=1}^{m} p_j \cdot x_{j,i} \quad \text{s.t.} \tag{1}$$

$$\begin{cases} \sum_{j=1}^{n} w_j \cdot x_{j,i} \leq C_i, & i \in [m], \quad (2a) \\ \sum_{i=1}^{m} x_{j,i} \leq 1, & j \in [n], \quad (2b) \\ x_{j,i} \in \mathbb{N}, & j \in [n],\ i \in [m], \quad (2c) \end{cases}$$

For the **Branching** and **Bounding** components, we will rely on a relaxation of (1)–(2c). Several such relaxations are discussed in [24]; the ones that are relevant to us are the *linear programming* relaxation ((1), (2a), (2b), and non-negativity constraints instead of (2c)) and the *surrogate relaxation*, which in essence merges the m knapsacks into one single knapsack with capacity $\sum_{i=1}^{m} C_i$:

$$S\text{-}MK_m(\boldsymbol{C}, \boldsymbol{w}, \boldsymbol{p}) = MK_1\left(\sum_{i=1}^{m} C_i, \boldsymbol{w}, \boldsymbol{p}\right) \quad (3)$$

Martello and Toth show in [24] that the optimum of the linear relaxation of the surrogate relaxation coincides with the optimum of the linear relaxation of the original problem. Using this relationship and the well-known observation of George Dantzig [8], they describe an algorithm that returns an $(m+1)$-approximate solution. They sort the n items decreasingly by their unit profit $\frac{p_i}{w_i}$, and greedily fill each knapsack in this order until an item no longer fits inside entirely. Then they cut the excessive part and assign it to the next knapsack, and resume the process on this new knapsack with the next item in the queue. Let $\boldsymbol{x}^*$ denote this optimal solution to the linear relaxation, and let $s_1, \ldots, s_m$ denote the (at most) m items that are fractionally assigned in the process. Item s_k is referred to as the *critical item relative to knapsack* k, and is obtained as $s_k = \min\left\{j : \sum_{l=1}^{j} w_l > \sum_{i=1}^{k} C_i\right\}$.

They conclude that the (at most) m individual critical items and the collection of integrally assigned items yield (at most) $m+1$ feasible solutions to the integer program, the best of which has a profit of at least $\frac{1}{m+1}$ times the fractional optimum. Let $\boldsymbol{x'}$ denote the most profitable of these $m+1$ assignments. With these notations, we have that

▶ **Proposition 1** (Martello, Toth; [24])**.**

$$\boldsymbol{p} \cdot \boldsymbol{x}' = \max\left\{p_{s_1}, \ldots, p_{s_m}, \sum_{k=1}^{m} \sum_{j=s_{k-1}+1}^{s_k - 1} p_j\right\} \geq \frac{1}{m+1}(\boldsymbol{p} \cdot \boldsymbol{x}^*).$$

In the analysis of the algorithm A_α^{knap}, we are going to need the following simple observation, which connects the profit of the best critical item j^* (i.e. the critical item with the highest profit) with the gap of $\boldsymbol{x}'$ with respect to the optimal $\boldsymbol{x}^*$:

▶ **Lemma 1.**

$$\frac{p_{j^*}}{\boldsymbol{p} \cdot \boldsymbol{x}^*} \geq \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot \left(1 - \frac{\boldsymbol{p} \cdot \boldsymbol{x}'}{\boldsymbol{p} \cdot \boldsymbol{x}^*}\right)\right\}.$$

For a fixed $0 < \alpha < 1$, the specifications of algorithm A_α^{knap} are as follows: as input, we have a triple $(\boldsymbol{C}, \boldsymbol{w}, \boldsymbol{p})$ defining an instance of the multi-knapsack problem. At each step, we select a node v (the *branching node*) among the leaves (the *active nodes*) of a tree we build

step-by-step; each node corresponds to a subproblem in which we fix some variables that are given by the unique path from the root to the node. The **selection** in our case occurs according to the best-first strategy, where we select the node to be processed next whose attributed upper bound (described later) is the largest of all active nodes. For convenience, let us introduce the "dummy" variables $\bar{x}_j = 1 - \sum_{i=1}^{m} x_{j,i}$ for $j = 1, \ldots, n$. Suppose that in the unique path from the root of the tree to v, we have fixed $x_{j_1,i_1} = x_{j_2,i_2} = \ldots = x_{j_k,i_k} = 1$, and $\bar{x}_{e_1} = \bar{x}_{e_2} = \ldots = \bar{x}_{e_l} = 1$. In other words, items $j_1, \ldots, j_k$ are set to be included in knapsack $i_1, \ldots, i_k$ respectively; whereas items $e_1, \ldots, e_l$ are completely disposed of. Consequently, node v encodes the knapsack sub-problem on the ground set $S = [n] \setminus \left(\bigcup_{z=1}^{k} j_z \cup \bigcup_{z=1}^{l} e_z\right)$ given by $(\boldsymbol{C}_v, \boldsymbol{w}_v, \boldsymbol{p}_v)$ with $\boldsymbol{C}_v = (C'_1, \ldots, C'_m)$ where $C'_i = C_i - \sum_{z:\, i_z = i} w_{j_z}$, $i \in [m]$; $\boldsymbol{w}_v = \boldsymbol{w}|_S$ and $\boldsymbol{p}_v = \boldsymbol{p}|_S$.

In the **bounding** component, a local upper and lower bound $U(v)$ and $L(v)$ is determined in the following manner: we consider the appropriate integer program in (1) with parameters $(\boldsymbol{C}_v, \boldsymbol{w}_v, \boldsymbol{p}_v)$, and its surrogate relaxation. The linear relaxation of (3) is solved to optimality by Dantzig's method giving $\boldsymbol{x}^*$, and the $(m+1)$-approximate integer solution $\boldsymbol{x}'$ is obtained according to Proposition 1. We save the subproblem optimum and feasible solution into the variables $SU(v) = \boldsymbol{p}_v \cdot \boldsymbol{x}^*$ and $SL(v) = \boldsymbol{p}_v \cdot \boldsymbol{x}'$; and from these, we create a feasible solution and a local upper bound *to the original problem* by putting items $j_1, \ldots, j_k$ back in knapsacks $i_1, \ldots, i_k$, respectively. Therefore, we set $U(v) = \boldsymbol{p}_v \cdot \boldsymbol{x}^* + (p_{j_1} + \ldots p_{j_k})$, and $L(v) = \boldsymbol{p}_v \cdot \boldsymbol{x}' + (p_{j_1} + \ldots p_{j_k})$.

Next, we expand the current tree by creating $m+1$ new subproblems that are going to be represented by the children of v; this **branching** rule is determined by setting the best critical item j^* as the pivot element. For $i < m+1$, the i-th new branch is identified by fixing $x_{j^*,i} = 1$, and corresponds to putting the best critical item j^* in knapsack i while reducing its capacity by w_{j^*}. The $(m+1)$-th new branch corresponds to setting $\bar{x}_{j^*} = 1$ (and $x_{j^*,1} = \ldots = x_{j^*,m} = 0$ at the same time), and means that we exclude item j^* from all of the knapsacks. We calculate the pertaining upper and lower bounds of all $m+1$ sub-problems. If any of them are infeasible (including the case when item j^* does not fit into knapsack i for some i), or their upper bounds are lower than the value of an already found feasible integer solution, we prune the corresponding branch. Otherwise, we add them to the set of active nodes while removing v. We update the highest local upper bound (GU) and the best integer solution found so far (GL). Finally, we terminate whenever the multiplicative gap between the global upper bound and the best solution found so far reaches or goes above α; i.e. $\frac{GL}{GU} \geq \alpha$.

Since processing a node v clearly takes polynomial time, a polynomial upper bound on the number of visited nodes in the branching tree means that the above scheme is a PTAS.

Proof of Theorem 1. The proof relies on deriving an upper bound on the number of visited nodes. Let $F = F(\boldsymbol{C}, \boldsymbol{w}, \boldsymbol{p})$ denote the branching tree at termination. On a root-leaf path in F, we call an edge *right-turn* if the child node of the edge is the rightmost child of the parent out of the $m+1$ potential children. In other words, the node encodes a step when we leave an item out from all of the knapsacks. Any other step in the path will be called a *left-turn*. The theorem is a simple consequence of the following lemma:

▶ **Lemma 2.** *There exists a constant c_α (that depends on α) such that in the tree F, every root-leaf path contains at most c_α-many left-turns.*

This lemma indeed implies our theorem, since it implies that the number of different

root-leaf paths in F is at most $\binom{n}{c_\alpha} \cdot m^{c_\alpha}$, so we can have at most $O(n^{c_\alpha+1} \cdot m^{c_\alpha})$ nodes in F. ◀

Proof of Lemma 2. For a given α and n, let $l(\alpha, n)$ be the maximum number of left-turns in F for any input of n items. Formally,

$$l(\alpha, n) = \sup_{\boldsymbol{w},\boldsymbol{p}\in\mathbb{N}^n,\, \boldsymbol{C}\in\mathbb{N}^m} \{\text{max. number of left-turns in any path of } F(\boldsymbol{C}, \boldsymbol{w}, \boldsymbol{p})\}$$

Since the maximal number of left-turns is at most n in every possible path in every possible branching tree with n items, we can replace the supremum with maximum. Therefore, we can consider an infinite sequence $(\boldsymbol{C}_n, \boldsymbol{w}_n, \boldsymbol{p}_n)$ and F_n realizing the maximum for each n. Our goal now is to prove that $l(\alpha, n)$ remains constant as n tends to infinity. Let us fix n, and the corresponding triple $(\boldsymbol{C}_n, \boldsymbol{w}_n, \boldsymbol{p}_n)$ with F_n such that they realize $l(\alpha, n)$. For the sake of simplicity, we will omit n from the notation, and simply consider $(\boldsymbol{C}, \boldsymbol{w}, \boldsymbol{p})$ and F.

Let F' be the inner nodes of F (the tree we get by getting rid of all leaves). Since $|F'| \geq \frac{1}{m+1} \cdot |F|$ (as each group of at most $m+1$ sibling leaves in F has one unique parent in F'), it is enough to prove the upper bound on left-turns in F'. Consider a root-leaf path in F' with $l(\alpha, n)$-many left-turns, and let the corresponding nodes be $v_1, v_2, \ldots, v_{l(\alpha,n)}$, with items $j_1, \ldots, j_{l(\alpha,n)}$ fixed to be in knapsacks $i_1, \ldots, i_{l(\alpha,n)}$ along the path.

For the node v_t with $1 \leq t \leq l(\alpha, n)$, let $\boldsymbol{x}^*_{v_t}$ and $\boldsymbol{x}'_{v_t}$ denote fractional optimum of the sub-problem's surrogate relaxation and its $(m+1)$-approximate integer rounding, with objective function values $SU(v_t)$ and $SL(v_t)$, respectively. Let $\frac{SL(v_t)}{SU(v_t)} = r_t$. Recall that $r_t \geq \frac{1}{m+1}$, and that

$$\frac{L(v_t)}{U(v_t)} = \frac{SL(v_t) + p_{j_1} + \ldots + p_{j_{t-1}}}{SU(v_t) + p_{j_1} + \ldots p_{j_{t-1}}}.$$

The key observations are the following:

- Since v_t is the inner node of F, the algorithm processed it in a previous step and did not halt. Consequently, if GL' and GU' denote the back-then global lower and upper bounds, we have that $\alpha > \frac{GL'}{GU'}$.
- GL' was defined as the best of all integer solutions found previously, so $GL' \geq L(v_t)$.
- By the choice of the tree-traversal strategy being best-first, $U(v_t) = GU'$.

Putting these together, we have that

$$\alpha > \frac{GL'}{GU'} \geq \frac{L(v_t)}{U(v_t)} = \frac{SL(v_t) + p_{j_1} + \ldots + p_{j_{t-1}}}{SU(v_t) + p_{j_1} + \ldots p_{j_{t-1}}} =$$

$$= \frac{r_t \cdot SU(v_t) + p_{j_1} + \ldots + p_{j_{t-1}}}{SU(v_t) + p_{j_1} + \ldots p_{j_{t-1}}} = r_t + (1 - r_t) \cdot \frac{p_{j_1} + \ldots + p_{j_{t-1}}}{SU(v_t) + p_{j_1} + \ldots + p_{j_{t-1}}} =$$

$$= r_t + (1 - r_t) \cdot \frac{\frac{p_{j_1}}{SU(v_t)} + \ldots + \frac{p_{j_{t-1}}}{SU(v_t)}}{1 + \frac{p_{j_1}}{SU(v_t)} + \ldots + \frac{p_{j_{t-1}}}{SU(v_t)}}.$$

As an immediate consequence, we have that $\alpha > r_t$, and so

$$1 - r_t > 1 - \alpha, \quad t = 1, \ldots, l(\alpha, n). \tag{4}$$

With a universal bound (in t) established in (4), we can now focus on the last left-turn in the path, $v_{l(\alpha,n)}$. Let us introduce a shorthand notation for the fraction on the right-hand side, and repeat what we have for $t = l(\alpha, n)$:

$$f_{l(\alpha,n)} := \frac{\frac{p_{j_1}}{SU(v_{l(\alpha,n)})} + \ldots + \frac{p_{j_{l(\alpha,n)-1}}}{SU(v_{l(\alpha,n)})}}{1 + \frac{p_{j_1}}{SU(v_{l(\alpha,n)})} + \ldots + \frac{p_{j_{l(\alpha,n)-1}}}{SU(v_{l(\alpha,n)})}},$$

and

$$\alpha > r_{l(\alpha,n)} + (1 - r_{l(\alpha,n)}) \cdot f_{l(\alpha,n)} = 1 - (1 - r_{l(\alpha,n)})(1 - f_{l(\alpha,n)}). \tag{5}$$

▶ **Lemma 3.**

$$f_{l(\alpha,n)} \geq \frac{(l(\alpha,n)-1) \cdot \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-\alpha)\right\}}{1 + (l(\alpha,n)-1) \cdot \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-\alpha)\right\}}.$$

Proof of Lemma 3. First, note that

$$SU(v_{l(\alpha,n)}) \leq SU(v_{l(\alpha,n)-1}) \leq \ldots \leq SU(v_2) \leq SU(v_1),$$

since any two consecutive nodes in the sequence v_i and v_{i+1} encode problems where the problem in node v_{i+1} is a sub-problem of the one given by node v_i. Second, recall from Lemma 1 that

$$\frac{p_{j_t}}{SU(v_t)} \geq \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1 - r_t)\right\}, \;\; t = 1, \ldots, l(\alpha,n).$$

Last, notice that $f_{l(\alpha,n)}$ is a fraction of the type $\frac{x}{x+1}$, therefore it is monotone increasing in the value of the nominator. Combining these observations with (4), we can write that

$$f_{l(\alpha,n)} = \frac{\frac{p_{j_1}}{SU(v_{l(\alpha,n)})} + \ldots + \frac{p_{j_{l(\alpha,n)-1}}}{SU(v_{l(\alpha,n)})}}{1 + \frac{p_{j_1}}{SU(v_{l(\alpha,n)})} + \ldots + \frac{p_{j_{l(\alpha,n)-1}}}{SU(v_{l(\alpha,n)})}} \geq \frac{\frac{p_{j_1}}{SU(v_1)} + \ldots + \frac{p_{j_{l(\alpha,n)-1}}}{SU(v_{l(\alpha,n)-1})}}{1 + \frac{p_{j_1}}{SU(v_1)} + \ldots + \frac{p_{j_{l(\alpha,n)-1}}}{SU(v_{l(\alpha,n)-1})}} \geq$$

$$\geq \frac{\min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-r_1)\right\} + \ldots + \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-r_{l(\alpha,n)-1})\right\}}{1 + \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-r_1)\right\} + \ldots + \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-r_{l(\alpha,n)-1})\right\}} \geq$$

$$\geq \frac{(l(\alpha,n)-1) \cdot \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-\alpha)\right\}}{1 + (l(\alpha,n)-1) \cdot \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-\alpha)\right\}}.$$

◀

Now suppose for contradiction that $\lim_{n\to\infty} l(\alpha,n) = \infty$. Then (5) implies

$$\alpha \geq 1 - \lim_{n\to\infty} (1 - r_{l(\alpha,n)})(1 - f_{l(\alpha,n)}) = 1,$$

since $1 \geq r_{l(\alpha,n)} > 0$ and

$$\lim_{n\to\infty} f_{l(\alpha,n)} \geq \lim_{n\to\infty} \frac{(l(\alpha,n)-1) \cdot \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-\alpha)\right\}}{1 + (l(\alpha,n)-1) \cdot \min\left\{\frac{1}{m+1}, \frac{1}{m} \cdot (1-\alpha)\right\}} =$$

$$= \lim_{x\to\infty} \frac{x}{1+x} = 1.$$

However, $1 > \alpha > 0$ was assumed. Contradiction. ◀

▶ Remark 1. With a more careful analysis, we can determine an exact bound on $l(\alpha, n)$ that depends on alpha, just like we did in Theorems 2 and 3. In particular, observe that (5) implies $\alpha > f_{l(\alpha,n)}$, and assume that in Lemma 3, the minimum is obtained by $\frac{1}{m} \cdot (1-\alpha)$. It follows that

$$\alpha > \frac{(l(\alpha,n)-1) \cdot \frac{1}{m} \cdot (1-\alpha)}{1+(l(\alpha,n)-1) \cdot \frac{1}{m} \cdot (1-\alpha)} = 1 - \frac{1}{1+(l(\alpha,n)-1) \cdot \frac{1}{m} \cdot (1-\alpha)},$$

and

$$1+(l(\alpha,n)-1) \cdot \frac{1}{m} \cdot (1-\alpha) < \frac{1}{1-\alpha},$$

so

$$l(\alpha,n) < \left(\frac{1}{1-\alpha} - 1\right) \cdot \frac{m}{1-\alpha} + 1 = \frac{m\alpha}{(1-\alpha)^2} + 1.$$

On the other hand, if the minimum in Lemma 3 is achieved by $\frac{1}{m+1}$, a similar analysis shows that

$$l(\alpha,n) < \frac{m+1}{1-\alpha} + 1.$$

3 A B&B EPTAS for the Unrelated Machine Scheduling Problem

In the *unrelated parallel machine scheduling* problem, n jobs are assigned to m machines to minimize the *makespan* $\max\{C_S(i) : i = 1, \ldots, m\}$, where $C_S(i)$ is the completion time of machine i according to the schedule S. Each job j has a machine-dependent processing time $p_{j,i} \in \mathbb{N}$. The problem is strongly NP-complete when m is part of the input [13], ruling out an FPTAS unless P = NP. Furthermore, even a PTAS would imply P = NP [22]. When m is a fixed constant, the problem is denoted by $Rm||C_{\max}$ (following [5]), and an FPTAS is possible [16]. See, e.g. [26] for a survey of more recent advances. Several applications of the B&B framework were developed for machine scheduling problems, with diverse lower bound strategies ranging from surrogate relaxations [29] to lagrangian relaxations [23].

In this section, we study a B&B implementation for $Rm||C_{\max}$. Again, we follow a simple and standard implementation of the B&B framework. At a given node, the corresponding sub-problem is modeled as an integer program. Its LP-based relaxation is solved by a Binary Search (BS) subroutine in the **bounding** component, followed by a common rounding technique to determine an $(m+1)$-approximate schedule (see [30], Chapter 17.3). Then, we **branch** according to the fractional job that maximizes the minimal processing time $\min\{p_{j,i} : i \in [m]\}$. The **selection** strategy is again the best-first rule, where the active node with the lowest fractional optimum (denoted by the acronym LLB in the experiments) is picked for processing. We stop whenever the ratio between the global lower bound and the best integer schedule discovered so far reaches or goes below $(1+\epsilon)$, where $\epsilon > 0$ is a fixed constant. We provide exact details of the algorithm $A_\epsilon^{\text{unrel}}$ in Appendix 7.1, where we show the following:

▶ **Theorem 2.** *For every fixed $\epsilon > 0$, the algorithm A_ϵ^{unrel} returns a $(1+\epsilon)$-approximate solution to the unrelated machine scheduling problem, after processing at most $m^{\lfloor \frac{m^2}{\epsilon} \rfloor}$-many nodes in the branching tree.*

4 A B&B FPTAS for the Uniform Machine Scheduling Problem

In the *identical machine scheduling* setup, every job takes the same amount of time to complete on each of the machines, and so job j is associated with a single processing time

p_j. The *uniform machine scheduling* setup extends this by associating a speed s_i with each machine $i \in [m]$, and thus rendering the processing time of job j on machine i to be $p_{j,i} = \frac{p_j}{s_i}$. In this paper, we assume that the vector of processing times $\boldsymbol{p}$ and the vector of speeds $\boldsymbol{s}$ are such that $p_{j,i} \in \mathbb{N}, \forall j \in [n], \forall i \in [m]$. The problem is frequently denoted by $Qm||C_{max}$ when m is a constant (following [5]); it is weakly NP-hard and, being a special case of $Rm||C_{max}$, it admits an FPTAS.

In this section, we enhance $A_\epsilon^{\text{unrel}}$ for the special case $Qm||C_{max}$ by exploiting a simple observation. During the course of the algorithm, certain repetitive patterns can be identified based on the jobs that are fixed at a given node. Situations may arise where two or more nodes encode sub-problems that could be classified as similar; provided that the fixed job-machine pairings yield schedules that are sufficiently close to each other (the coordinate-wise distance of schedules required to declare them similar is based on the error tolerance factor ϵ). When aiming for an approximate solution, the processing of these similar nodes can be delayed indefinitely. When the modified B&B runs its full course, the returned schedule is either optimal or it is within a range of ϵ of the real optimal solution that we put on hold due to similarity.

As we will see, this modified scheme $A_\epsilon^{\text{sim-prof}}$ (which, apart from the enhanced node selection rule, is equivalent with $A_\epsilon^{\text{unrel}}$) reduces the search space by such a great factor that the running time will be polynomial in $1/\epsilon$ as well. The following theorem is proven in Appendix 7.2.

▶ **Theorem 3.** *For every fixed $\epsilon > 0$, the algorithm $A_\epsilon^{sim\text{-}prof}$ returns a $(1+\epsilon)$ - approximate solution to the uniform machine scheduling problem, after processing at most $n \cdot \left(\frac{5n}{\epsilon}\right)^m$ nodes in the branching tree.*

Apart from its increased efficiency, the scheme offers the additional advantage of adaptability to changing requirements and time constraints. If the algorithm completes before its designated time limit or a higher-quality solution is needed, it can resume processing the delayed nodes, further refining the search space using a more precise similarity measure with a smaller ϵ.

5 Computational Experiments

In this section, we aim to assess the performance of our proposed algorithm on some randomly generated instances. Specifically, we compare our proposed strategies, which gave us theoretical guarantees, with other ones commonly used. The goal is to assess whether our theoretical guarantees are also observable in practice. In Appendix 8, we provide a detailed runtime analysis. For the instances under study, a carefully optimized B&B implementation, such as SCIP [3], outperforms our naive implementation. However, we chose to reimplement everything from scratch, focusing on simplicity rather than efficiency.

Experimental Setting. All experiments were conducted on a Linux computer equipped with Intel Xeon E5-2650 v3 CPUs, each running at 2.3 GHz, and 64 GB of RAM. Our main code was implemented in Python 3.10.14, and all optimization routines were carried out using SCIP [3]. The code will be made available upon acceptance of this paper.

5.1 Multiple knapsacks

To test our algorithm, we generate 30 random instances for each pair $(n, m) \in \{(5,2), (10,2), (10,5), (50,2), (50,5), (50,15), (100,2), (100,5), (100,10), (100,15)\}$. Capacities are uniformly

sampled integers from the range $[c_{\min}, c_{\max}]$, where $c_{\min} = \min_j w_j$ and $c_{max} = \left\lceil \frac{\sum w_j}{n} \right\rceil - c_{\min}$. The lower bound ensures that each item fits inside at least one of the knapsacks, while the upper bound ensures that (on average) half of the items fit in the union of the knapsacks, as discussed in [6].

As baselines for node selection, we test DFS and BFS alongside the proposed GUB rule. For branching rules, we evaluate two approaches in addition to the previously introduced "critical element" (CE) strategy. In one strategy, we branch on the items among the *fractional* ones with the largest profit-to-weight ratio (PPW). In the other strategy, as suggested by [20], we branch on the item among the *unfixed* ones with the largest profit-to-weight ratio (K). We test these strategies for different values of α and collect various metrics, including the number of nodes explored, the gap to the optimum, the maximum depth reached, the number of nodes after finding the optimum, and the number of left turns. Here, we report partial results, while a more extensive set of experiments is available in the interactive notebook, which we will provide upon acceptance of this paper.

Figure 1a shows the number of nodes explored to get an $\alpha = 0.97$ approximation. Since our implementation is a proof of concept and not fully optimized, we encountered memory issues. To address this, we imposed a threshold of 10^4 nodes explored, beyond which we return the best solution found so far. We observe that, in terms of number of nodes explored, the Greatest Upper Bound (GUB) strategy consistently outperforms the others. This is particularly evident in the "hard" instances $(100, 10), (100, 15), (50, 15)$, where all successful methods in at least one instance involve the GUB strategy. Our proposed strategy (yellow box) frequently achieves the best overall performance. Interestingly, in several cases, branching using the PPW rule yields better results compared to branching based on the Critical Element (CE) criterion.

In our analysis, we also record the optimality gap of the returned solution, defined as

$$\frac{|z - z^*|}{\max(z, z^*)}$$

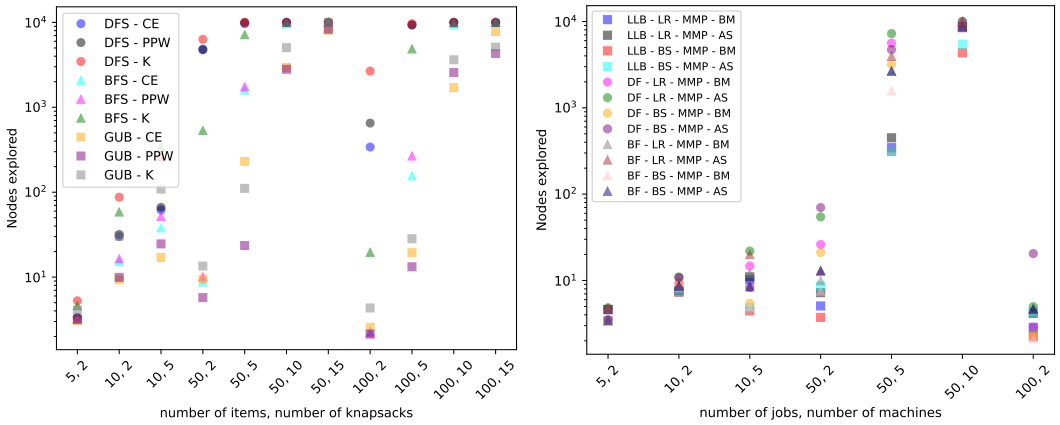
where z is the solution as returned by our algorithms and z^* is the optimal solution we computed using state-of-art Google OR-Tools [28] with SCIP [3] as a linear solver.

Figure 2a presents this information, clearly showing that GUB is often a winning strategy in terms of producing high-quality solutions. In this case, we do not observe any significant difference between CE and PPW.

5.2 Unrelated machine scheduling problem

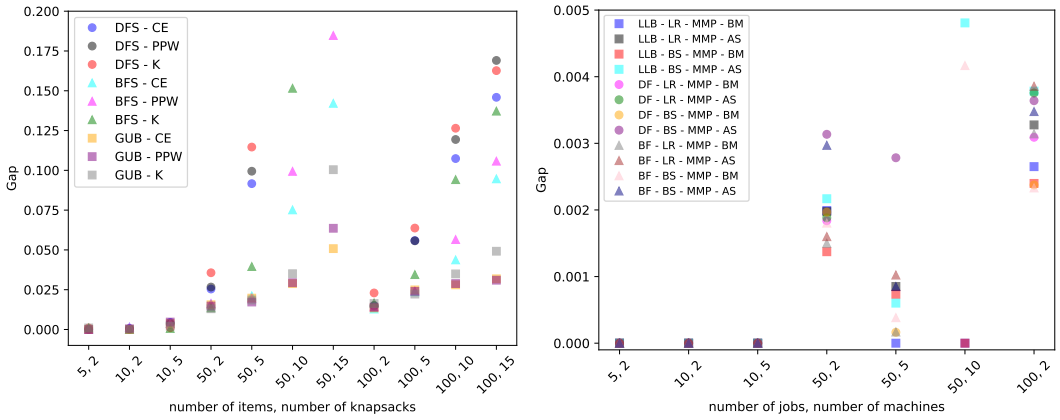
In this case, we generate 30 random instances for each pair $(n, m) \in \{(5, 2), (10, 2), (10, 5), (50, 2), (50, 5), (50, 10), (100, 2)\}$. Job lengths are uniformly sampled integers from the range $[1, 100]$. Note that, in this case, the analysis of the $(50, 5)$ instance could not be completed within our 48-hour time frame. Hence, we report only the average over the instances that were successfully solved. We attempt to understand why this occurred, given that the Multi-Knapsack framework initially seemed more tractable. In this case, the binary search involves repeatedly solving LPs, significantly increasing computational overhead. We have 12 different B&B-like algorithms to evaluate, whereas, in the Multi-Knapsack setting, there were only 9. Lastly, unlike Multi-Knapsack, there is no pruning by infeasibility, making it harder to discard unpromising nodes quickly.

As baselines for node selection, we test DFS and BFS along with the proposed rule LLB. For the lower bound, we chose both the proposed BS scheme and the Linear Relaxation



(a) Multi knapsack, $\alpha = 0.97$ **(b)** Unrelated Machine Scheduling, $\epsilon = 0.01$

Figure 1 Performance of different strategies in the branch-and-bound method for the Multi-Knapsack and Unrelated Machine Scheduling problems. The number of nodes explored before termination (or reaching the stopping condition) is reported on a logarithmic scale.



(a) Multi knapsack, $\alpha = 0.97$ **(b)** Unrelated Machine Scheduling, $\epsilon = 0.01$

Figure 2 Performance of different strategies in the branch-and-bound method for the Multi-Knapsack and Unrelated Machine Scheduling problems. The optimality gap is measured before termination (either upon reaching the stopping condition or exiting). The y-axis has been limited to highlight the most relevant portion of the plot.

(LR) of Integer Linear Programming minimizing the makespan that is commonly used in unrelated parallel machine scheduling. As the branching rule, we test only the one we propose: branching on the variable with the largest minimum processing time across machines (MMP). Both BS and LR return a solution that may contain fractional components, which we need to round to obtain an upper bound on the optimal solution. We compare two different rounding strategies (i) The one we prove leads to a PTAS, which assigns *A*ll fractional jobs to the machine where their processing time is *S*hortest (AS); (ii) An alternative approach based on Best Matching (BM) of the at most m fractional jobs, where we find a matching that minimizes the total makespan. Even in this case, we observe a similar trend: LLB results in fewer nodes explored. However, on average, BM appears to be a better rounding strategy compared to AS. Regarding the optimality gap, interestingly, BFS slightly outperforms LLB in some instances (e.g., $n = 100, m = 5$). This is not surprising, as our theoretical results (Proposition 2) suggest that BFS also guarantees a PTAS. Overall, we observe a significant difference in the order of magnitude of the average gap between the two problems. In the first case, a gap of 0 is rarely achieved, whereas, in the Unrelated Job Scheduling problem, the algorithm reaches the optimal solution for $n \in \{5, 10\}$ and for the instance $(100, 2)$, regardless of the choice of branching, bounding, and selection strategy.

Although it is not the focus of our paper, we provide an analysis of exact running times of our algorithms in Appendix 8.

6 Concluding Remarks

Let us collect general observations about our methods. Most importantly, we note that the best-first rule can be replaced by BFS in $A_\epsilon^{\text{unrel}}$ without losing the theoretical worst-case guarantee (while keeping the other parameters fixed). The key element of our proof was to limit the depth of the final tree of the algorithm (F) at $\lfloor \frac{m^2}{\epsilon} \rfloor$. Let F' be the B&B tree of the alternative with BFS, limited to depth $\lfloor \frac{m^2}{\epsilon} \rfloor$. Since $F \subseteq F'$, the lowest lower bound in F' (denoted by GL') is greater or equal than the lowest lower bound in F (denoted by GL), and the best integer solution found in F' (denoted by GU') is better than the best integer solution we found with $A_\epsilon^{\text{unrel}}$ (denoted by GU). Hence,

$$\frac{GU'}{GL'} \leq \frac{GU}{GL} \leq 1 + \epsilon,$$

and it follows that with BFS as the search strategy (denoted by $A_\epsilon^{\text{BFS-unrel}}$), we terminate no later than processing F'. Since $|F'| \leq m^{\lfloor \frac{m^2}{\epsilon} \rfloor}$, we have that

▶ **Proposition 2.** *For every fixed $\epsilon > 0$, the algorithm $A_\epsilon^{BFS\text{-}unrel}$ returns a $(1+\epsilon)$-approximate solution to the unrelated machine scheduling problem, after processing at most $m^{\lfloor \frac{m^2}{\epsilon} \rfloor}$-many nodes in the branching tree.*

However, according to our experiments, best-first seems empirically better in terms both of optimality gap and number of nodes explored. For A_α^{knap}, we do not have similar guarantees as there we only have bound on the number of left-turns in the branching tree and the depth of F can potentially be as large as n. For the algorithm $A_\epsilon^{\text{sim-prof}}$, the bound on the number of visited nodes was independent of the tree traversal strategy, since our proof only relies on the limited number of different nodes at each level. Therefore, any alternative strategy can be used to replace the best-first one with the same worst-case guarantee.

Next, we note that for the machine scheduling problem, the results and algorithms can be modified to work with the "standard" LP-formulation lower bound instead of the binary

search one, but the lower bound itself is trivially worse. Last, we mention that for the special case of identical parallel machines, $A_\epsilon^{\text{unrel}}$ can be improved with a slight change in the rounding method. This version visits $m^{\lfloor \frac{m}{\epsilon} \rfloor}$ nodes in the worst case, by keeping the exact same argument. Furthermore, there are fast and intuitive heuristics for finding vertices of the polyhedra, thus the time spent in individual nodes can also be reduced.

We conclude by collecting the most essential properties of the knapsack and machine scheduling problems that were exploited during the investigation, intending to set the ground for generalizing the results to a larger class of problems.

Perhaps the most paramount property the two problems have in common is the notion of *self-similarity*. To repeatedly apply the same argument for each node of a path in the branching tree, we needed the sub-problems encoded by these nodes to fall into the same category as the original problem. In other words, fixing one variable to 1 or 0 should result in a problem that is in the same class as the original one. This property was by default true for the knapsack problem: setting $x_{j,i} = 1$ in $MK_m(\boldsymbol{C}, \boldsymbol{w}, \boldsymbol{p})$ yields the sub-problem $MK_m(\boldsymbol{C'}, \boldsymbol{w}|_{[n]-j}, \boldsymbol{p}|_{[n]-j})$ with $\boldsymbol{C'} = (C_1, \ldots, C_{i-1}, C_i - w_j, C_{i+1}, \ldots, C_m)$, while the rightmost branch corresponds to $MK_m(\boldsymbol{C}, \boldsymbol{w}|_{[n]-j}, \boldsymbol{p}|_{[n]-j})$. For the machine scheduling problem, on the other hand, the default description was not sufficient. If we fix a binary variable to 1 in the standard linear programming formulation, the resulting LP will not correspond to a machine scheduling problem of the same type. However, introducing overheads ensures the desired property, since now setting a variable to 1 corresponds to increasing the appropriate machine's overhead by the processing time of the fixed-job.

Strongly related to this property, we relied on the monotonicity of subproblems: for maximization problems, the local upper bound of a node is greater than the local upper bound of any of its children (for minimization problems, a similar property holds). However, this is a direct consequence of using the same objective function on subsequently smaller sets.

We also exploited that there was a quantifiable relationship between a node's lower/upper bounds and the job/item that was fixed at the node. For the knapsack problem, this relationship is guaranteed by Lemma 1, whereas for the machine scheduling problem, the inequality

$$\frac{p'_j}{L(v)} \leq \frac{m}{k}$$

provided the connection.

Finally, the least demanding requirement we need to pose is that of approximability. When rounding a fractional solution of a sub-problem at a node, an $(m+1)$-approximation algorithm was used for both the knapsack problem and the machine scheduling problem. However, for the machine scheduling problem, the proof did not rely upon this local rounding guarantee, and hence the necessity of a constant-factor approximation rounding is unclear.

It is important to acknowledge the limitations and drawbacks of our approach. During the last couple of decades, several approximation schemes have been described for both the knapsack and the machine scheduling problem. In fact, both problems are known to admit a fully polynomial-time approximation scheme (FPTAS), which is far superior to the PTAS framework in which the desired proximity ratio (ϵ or α) appears in the exponent of the running time. Furthermore, as we see in Appendix 7.2, our arguments are not directly repeatable or extendable for some of the cases. Nevertheless, we believe that the connection between B&B and approximation algorithms explored in the paper adds a surprising flavor to the theory of branch-and-bound algorithms, and sheds some light on their good behavior observed in practice.

For future research directions, we mention the possibility of a B&B yielding an FPTAS for the unrelated machine scheduling problem (or even more complex scheduling paradigms such as the job shop problem), with a possibly different choice of parameters and additional rounding tricks.

References

1. Karen Aardal, Andrea Lodi, Neil Yorke-Smith, and Lara Scavuzzo. Machine learning augmented branch and bound for mixed integer linear programming. *Mathematical Programming*, 2024.
2. Tolson Bell and Alan Frieze. Solving a Random Asymmetric TSP Exactly in Quasi-Polynomial Time w.h.p. 2023. Publisher: arXiv Version Number: 12. URL: `https://arxiv.org/abs/2308.02946`, `doi:10.48550/ARXIV.2308.02946`.
3. Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Liding Xu. The SCIP Optimization Suite 9.0, 2024. URL: `https://arxiv.org/abs/2402.17702`, `arXiv:2402.17702`.
4. Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. Knapsack problems - an overview of recent advances. part ii: Multiple, multidimensional and quadratic knapsack problems. *Computers & Operations Research*, 143(C):1–13, 2022.
5. Bo Chen, Chris N. Potts, and Gerhard J. Woeginger. *A Review of Machine Scheduling: Complexity, Algorithms and Approximability*, pages 1493–1641. Springer US, Boston, MA, 1998. `doi:10.1007/978-1-4613-0303-9_25`.
6. Vasek Chvátal. Hard knapsack problems. *Operations Research*, 28(6):402–411, 1980.
7. Grégoire Danoy and Gwen Maudet. Search strategy generation for branch and bound using genetic programming, 2024. URL: `https://arxiv.org/abs/2412.09444`, `arXiv:2412.09444`.
8. George B. Dantzig. Discrete variable extremum problems. *Operations Research*, 5(2):266–277, 1957.
9. Sanjeeb Dash. Exponential Lower Bounds on the Lengths of Some Classes of Branch-and-Cut Proofs. *Mathematics of Operations Research*, 30(3):678–700, 2005. Publisher: INFORMS. URL: `https://www.jstor.org/stable/25151677`.
10. Santanu S. Dey, Yatharth Dubey, and Marco Molinaro. Branch-and-bound solves random binary IPs in poly(n)-time. *Mathematical Programming*, 200(1):569–587, June 2023. `doi:10.1007/s10107-022-01895-4`.
11. Santanu S. Dey, Yatharth Dubey, Marco Molinaro, and Prachi Shah. A theoretical and computational analysis of full strong-branching. *Mathematical Programming*, 205:303–336, 2023.
12. Aleksei V. Fishkin, Klaus Jansen, and Monaldo Mastrolilli. Grouping techniques for scheduling problems: simpler and faster. *Algorithmica*, 51:183–189, 2008.
13. Michael R. Garey and David S. Johnson. Complexity result for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.
14. Harold Greenberg and Robert L. Hegerich. A branch search algorithm for the knapsack problem. *Management Science*, 16(5):327–332, 1970.
15. Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.
16. Ellis Horowitz and Sartaj Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM*, 23(2):317–327, 1976.
17. Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

18 Sheldon H. Jacobson, David R. Morrison, Jason J. Sauppe, and Edward C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching and pruning. *Discrete Optimization*, 19:79–102, 2016.

19 Walter H. Kohler and Kenneth Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of the ACM*, 21(1):140–156, 1974.

20 Peter J. Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13(9):723–735, 1967.

21 Eugene L. Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4(4):339–356, 1979.

22 Jan K. Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. In *28th Annual Symposium on Foundations of Computer Science*, pages 217–224, 1987.

23 Silvano Martello, Francois Soumis, and Paolo Toth. Exact and approximation algorithms for makespan minimization on unrelated parallel machines. *Discrete Applied Mathematics*, 75(2):169–188, 1997.

24 Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

25 Monaldo Mastrolilli. Efficient approximation schemes for scheduling problems with release dates and delivery times. *Journal of Scheduling*, 6:523–531, 2003.

26 Ethel Mokotoff. Parallel machine scheduling problems: A survey. *Asia-Pacific Journal of Operations Research*, 18(2):193–242, 2001.

27 Gábor Pataki, Mustafa Tural, and Erick B. Wong. Basis Reduction and the Complexity of Branch-and-Bound. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1254–1261. Society for Industrial and Applied Mathematics, January 2010. URL: `https://epubs.siam.org/doi/10.1137/1.9781611973075.100`, `doi:10.1137/1.9781611973075.100`.

28 Laurent Perron and Vincent Furnon. Or-tools. URL: `https://developers.google.com/optimization/`.

29 Steef van de Velde. Duality-based algorithms for scheduling unrelated parallel machines. *ORSA Journal on Computing*, 5(2):192–205, 1993.

30 Vijay V. Vazirani. *Approximation Algorithms*, pages 1493–1641. Springer-Verlag, Berlin, Heidelberg, 2001.

7 Appendix

7.1 A B&B EPTAS for the unrelated machine scheduling problem

In this section, we consider the problem $Rm||C_{max}$. Our analysis will rely on a *self-similarity* property of the problem; that is, each node of the branching tree must correspond to the same class of LP-formulations. For the sake of exploiting this property, we need that fixing a job to any of the machines should yield another machine scheduling problem. This is not true for the default description, so we introduce the concept of *overheads* denoting the earliest time machines can start completing jobs. Fixing a job j to machine i now corresponds to increasing the overhead of machine i with $p_{j,i}$. Machine i with an overhead $t_i \in \mathbb{N}$ has a completion time $C'_S(i) = t_i + C_S(i)$.

Again, the **bounding** and **branching** components will heavily rely on an integer programming formulation of the problem and its linear relaxation. The most straightforward formulation, however, gives rise to some concerns. Instead, we opt to follow in the footsteps of Vazirani [30] and Lenstra, Shmoys, and Tardos [22]. Their proofs rely on a technique called *parametric pruning*, which consists of a binary search for a "guess" on the optimal integer

makespan while disregarding job-machine pairings that immediately exceed the current guess. For this purpose, they define the following modification of the "standard" program: for a given $\boldsymbol{P} \in \mathbb{N}^{n\times m}$ and "guess" $T \in \mathbb{N}$, let S_T be the set of (job, machine) pairings which do not immediately violate the time limit T.

$$S_T := \{(j,i) : p_{j,i} \leq T\}.$$

▶ **Definition 2.** *For $\boldsymbol{P} \in \mathbb{N}^{n\times m}$, $T \in \mathbb{N}$ and $\boldsymbol{t} \in \mathbb{N}^m$, let $PARTIAL\text{-}LP\text{-}MS_m(\boldsymbol{P}, \boldsymbol{t}, T)$ be the polyhedron determined by the following set of inequalities:*

$$\begin{cases} \sum\limits_{i:(j,i)\in S_T} x_{j,i} = 1, & j \in [n], \\ \sum\limits_{j:(j,i)\in S_T} p_{j,i} \cdot x_{j,i} \leq T - t_i, & i \in [m], \\ x_{j,i} \geq 0, & (j,i) \in S_T. \end{cases} \tag{6}$$

The key properties described in [22] and [30] are easily transcribed to our version with overheads with little to no modification, since they are only dependent on the constraint matrix describing (6) and not on the right-hand side of the inequalities. Let us recall that for a feasible solution $\boldsymbol{x}$, job j is called *fractional* if there exists i such that $x_{j,i}$ does not equal 0 or 1 (and therefore has a fractional value); otherwise job j is called *integral*.

▶ **Lemma 4** (Lenstra, Shmoys, Tardos; [22]). *If the linear program described in (6) is feasible, then each vertex $\boldsymbol{x}^*$ has at most m fractional jobs. Furthermore, there exists an injection from fractional jobs to the m machines such that each fractional job j is matched to a machine i where $x_{j,i} \neq 0$. Moreover, the schedule we get by keeping integral jobs in $\boldsymbol{x}^*$ and reassigning fractional jobs to machines according to the injection has a makespan of at most $2T$.*

Lenstra et al. designed a binary search procedure (starting from an arbitrary integer schedule) for the smallest integer value of T for which the program in (6) is feasible. They prove that their procedure runs in polynomial time.

▶ **Proposition 3** (Lenstra, Shmoys, Tardos; [22]). *Let T' be the result of the binary search; i.e. the smallest integer T for which (6) is feasible. Furthermore, let T_{opt} be the fractional optimum. Then the rounding procedure from Lemma 4 applied to a schedule with makespan T' yields an integer schedule with makespan at most $2T_{opt}$.*

For a fixed $\epsilon > 0$, the specifications of algorithm $A_\epsilon^{\text{unrel}}$ are as follows: as input, we have a matrix $\boldsymbol{P} \in \mathbb{N}^{n\times m}$ defining an instance of the unrelated machine scheduling problem. The overhead at the beginning is $\boldsymbol{t} \equiv 0$. At each step, we select a node v (the *branching node*) among the leaves (the *active nodes*) of a tree we build step-by-step; each node corresponds to a subproblem in which we fix some job-machine pairings identified by the unique path from the root to the node. The **selection** in our case occurs according to the best-first selection rule, where we select the node to be processed next whose attributed lower bound (described later) is the smallest of all active nodes. Suppose that in the unique path of length k from the root of the tree to v, we have fixed $x_{j_1,i_1} = x_{j_2,i_2} = \ldots = x_{j_k,i_k} = 1$. In other words, job j_1 is fixed to machine i_1, job j_2 is fixed to machine i_2, and so on. Consequently, node v encodes the sub-problem with jobs $S = [n] \setminus \bigcup_{z=1}^{k} j_z$ given by processing times $\boldsymbol{P}_v = \boldsymbol{P}|_{S\times[m]}$ and overhead vector determined by the already fixed job-machine pairings: $\boldsymbol{t}_v = (t_1, \ldots, t_m)$ with $t_i = \sum\limits_{z:i_z=i} p_{j_z,i_z}$, $i \in [m]$. With these, the **bounding** takes place: a local lower and

upper bound $L(v)$ and $U(v)$ is determined by applying the binary search of Lenstra et al. to find the smallest integer T (denoted by T') for which (6) is feasible, and by rounding a vertex of the corresponding polyhedron $PARTIAL\text{-}LP\text{-}MS_m(\boldsymbol{P}_v, \boldsymbol{t}_v, T')$ to an integer assignment with makespan at most $(m+1) \cdot T'$. The rounding consists of assigning each fractional job to the machine where its processing time is minimal. We then **branch** according to the fractional job j whose minimal processing time ($\min\{p_{j,i} : i \in [m]\}$) is maximal. Branch i out of the m new branches fixes job j to machine i and increases its overhead by $p_{j,i}$.

We calculate the local lower and upper bounds of all m new subproblems. If their lower bounds are greater than the makespan of an already found integer solution, we **prune** them. Otherwise, we add them to the set of active nodes while removing v. We update the global lower and upper bounds GL and GU: at a given step, they are defined as the minimal local lower bound of the active nodes and the best makespan of an integer solution found so far, respectively. Finally, we terminate whenever the multiplicative gap between the global lower bound and the current champion makespan, $\frac{GU}{GL}$, reaches or goes below $1+\epsilon$.

Proof of Theorem 2. Let F be the resulting branching tree, and let v be an arbitrary node different from the one at which the algorithm terminates. Let GU' and GL' denote the global upper and lower bounds at the time of processing v. By definition, we have $GU' \leq U(v)$; and by the best-first selection strategy, $GL' = L(v)$. The algorithm did not stop after processing v; hence

$$1 + \epsilon < \frac{GU'}{GL'} \leq \frac{U(v)}{L(v)}.$$

Let j^* be the fixed job at v; i.e. the job among the at most m fractional jobs of the vertex whose shortest processing time is maximal. For each job j', let $p'_{j'} = \min\{p_{j',1}, \dots p_{j',m}\}$ denote the shortest one out of all the m processing times. Suppose that job j^* is the k-th according to the decreasing order of $p'_{j'}$-s, and the first k jobs in this order are $j_1, j_2, \dots, j_{k-1}, j^*$. It is evident that rounding up the fractional jobs cannot increase the makespan by more than $m \cdot p'_{j^*}$, so $U(v) \leq L(v) + m \cdot p'_{j^*}$.

Observe that $L(v) \geq \frac{p'_1 + \dots p'_n}{m}$, since the latter is a lower bound on the global fractional optimum, whereas the first is the value of some feasible (fractional) solution to a subproblem. Moreover,

$$p'_1 + \dots p'_n \geq p'_{j_1} + \dots p'_{j_{k-1}} + p'_{j^*} \geq k \cdot p'_{j^*}.$$

From these, we have that

$$1 + \epsilon < \frac{GU'}{GL'} \leq \frac{U(v)}{L(v)} \leq 1 + \frac{m \cdot p'_{j^*}}{L(v)} \leq 1 + \frac{m^2 \cdot p'_{j^*}}{k \cdot p'_{j^*}} = 1 + \frac{m^2}{k}, \tag{7}$$

and $k \leq \lfloor \frac{m^2}{\epsilon} \rfloor$. On the path from the root of F to v, one job cannot be fixed more than once; so we have that when the depth of v exceeds $\lfloor \frac{m^2}{\epsilon} \rfloor$, at least one fixed job j' will violate $j' \leq \lfloor \frac{m^2}{\epsilon} \rfloor$. Therefore, the depth of F is at most $\lfloor \frac{m^2}{\epsilon} \rfloor$ (disregarding the terminating node), and the number of nodes in F is at most $m^{\lfloor \frac{m^2}{\epsilon} \rfloor}$. Since processing a node takes time that is polynomial in n, the overall running time is again polynomial. ◀

7.2 A B&B FPTAS for the uniform machine scheduling problem

In this section, we consider the problem $Qm||C_{\max}$, and we will enhance the previous algorithm $A_\epsilon^{\text{unrel}}$ by exploiting common input-modifying techniques that are frequently used for obtaining fully polynomial-time approximation schemes. In general, these techniques

consist of applying a series of transformations on the input instance, while keeping the objective value sufficiently close to the optimum. Most often, the modifications are a mixture of rounding down processing times to the nearest value of some finite sequence, and grouping small jobs together to reduce the number of jobs in the input. The rounding of processing times allows for greater control on feasible solutions and gives way to create *profiles* that collect equivalent schedules. On the other hand, grouping small jobs together results in a smaller instance for which even a complete enumeration of schedules would be feasible. If the parameters of the modification are chosen carefully, the combination of these two steps guarantees an algorithm that runs in polynomial time in both n and $\frac{1}{\epsilon}$. For a detailed background, we refer to [12] and [25].

Let us fix $\epsilon > 0$, and consider an input $\boldsymbol{P} \in \mathbb{N}^{n \times m}$ to the uniform machine scheduling problem with m fixed machines and n jobs. For the sake of a simpler analysis, let us divide each processing time with the global fractional optimum of the "standard" LP-relaxation. This step does not affect the optimal integer assignment, and its new makespan T_{opt} satisfies that

$$1 \leq T_{\text{opt}} \leq 2.$$

In our current investigation, we will solely rely on the first type of modification: rounding down processing times to the nearest value of some sequence. But, instead of directly modifying the input, we will design a scheme that allows us to obtain the same effect without touching the input first, thus guaranteeing a more "natural" approach. Our method builds on the concept of *profiles*: for a (partial) assignment S of some jobs, the profile of S is the m-tuple $(C_S(1), \ldots, C_S(m))$ of completion times. We call two profiles $\Pi(S_1)$ and $\Pi(S_2)$ similar if $|\Pi(S_1)_i - \Pi(S_2)_i| \leq \frac{\epsilon}{n}, \ \forall i \in [m]$. The key observation is the following: let $\epsilon < 1$, and note that a $(1+\epsilon)$-approximate solution has a makespan of at most $2(1+\epsilon) = 2 + 2\epsilon$. Consider the m-dimensional cube $[0, 2+2\epsilon]^m$, and consider its partition given by the set of points $[0, \frac{\epsilon}{n}, \frac{2\epsilon}{n}, \ldots, \frac{n(2+2\epsilon)}{\epsilon} \cdot \frac{\epsilon}{n}]^m$. If we have two profiles falling into the same partition class, then they are similar. Conversely, any set of profiles with makespan at most $2+2\epsilon$ that does not have 2 similar profiles has at most $\left(1 + \frac{n(2+2\epsilon)}{\epsilon}\right)^m \leq \left(\frac{5n}{\epsilon}\right)^m$ elements.

For a node v in the branching tree, its profile $\Pi(v)$ is defined as the profile of the partial schedule made up of the jobs fixed at v. In other words, the profile is simply the overhead vector associated with the integer programming formulation corresponding to the sub-problem at v: $\Pi(v) = \boldsymbol{t}_v$. The concept of similar profiles allows us to consider nodes of the branching tree "equivalent" if they have similar profiles, and they have the same set of jobs fixed so far. Note that we need *both* the same profiles and the same fixed jobs in order to declare two nodes equivalent, as shown by the following identical instance with 2 machines given by processing time $(n, n, 1, \ldots, 1)$ with n-many 1-jobs. We can have two partial assignments with the same profile (n, n), but one of them is made up of one n-job and n 1-jobs while the other is made up of two n-jobs. It is not justified to deem them equivalent as the best extension of the first profile has a makespan of $2n$, while the latter can be extended to a schedule with makespan $\frac{3}{2}n$.

However, the following Lemma gives a natural way to ensure that all nodes at a given level have the same fixed jobs in the uniform setup.

▶ **Lemma 5.** *Let $(\boldsymbol{P}, \boldsymbol{t}) \in (\mathbb{N}^{(n \times m)}, \mathbb{N}^m)$ be an instance of the uniform machine scheduling problem with n jobs where $\boldsymbol{P}$ is given by processing times $\boldsymbol{p} \in \mathbb{N}^n$ and machine speeds $\boldsymbol{s} \in \mathbb{N}^m$, and let n be the job whose processing time is maximal. Let T' denote the smallest integer T for which $PARTIAL - LP - MS_m(\boldsymbol{P}, \boldsymbol{t}, T)$ is feasible. If there exists a schedule $\boldsymbol{x}^*$ with at least one fractional job such that $\boldsymbol{x}^*$ is a vertex of $PARTIAL - LP - MS_m(\boldsymbol{P}, \boldsymbol{t}, T')$, then*

there exists a schedule $\hat{\boldsymbol{x}}$ in which job n is fractional and $\hat{\boldsymbol{x}}$ is a vertex of the same polyhedron.

In the proof of Lemma 5, we will exploit useful properties of vertices of the polytope described in (6). Namely, in the uniform machine scheduling model, we can extend the result of Lemma 4 and characterize vertices of the polyhedra. The basic idea of the lemma is the following: for a feasible solution $\boldsymbol{x}$, they construct a bipartite auxiliary graph $G(\boldsymbol{x})$ with the 2 classes corresponding to the m machines and the at most m fractional jobs, and they add an edge between $i \in [m]$ and $j \in [n]$ if $x_{j,i} > 0$ and is fractional. They conclude that $G(\boldsymbol{x})$ must be a *pseudo-forest*, and use this fact to construct a matching between machines and fractional jobs.

We can strengthen their observation in the uniform model, and use it to our advantage for characterizing vertices of the corresponding polyhedra. Let $(\boldsymbol{p}, \boldsymbol{s}) \in (\mathbb{N}^n, \mathbb{N}^m)$ denote an input to the uniform machine scheduling problem with $\boldsymbol{p}$ being the vector of processing times, and $\boldsymbol{s}$ being the vector of machine speeds. The corresponding input matrix is $\boldsymbol{P} = (p_{j,i})_{i,j=1,1}^{m,n}$ with $p_{j,i} = \frac{p_j}{s_i}$. Recall that $\boldsymbol{P} \in \mathbb{N}^{n \times m}$ is assumed, although it is not explicitly used in the proof. Let us recall that a machine's completion time according to some schedule S is denoted by $C'_S(i)$ when taking into account overhead t_i as well. With a little abuse of notation, a fractional solution $\boldsymbol{x}$ of the linear program can be interpreted as a fractional schedule, where the completion time at machine i is denoted by $C'_{\boldsymbol{x}}(i)$.

▶ **Lemma 6.** *Let $(\boldsymbol{P}, \boldsymbol{t})$ be an input to the uniform machine scheduling problem, and let T' denote the smallest integer T for which* (6) *is feasible; let $\boldsymbol{x}$ be a feasible solution of $PARTIAL - LP - MS_m(\boldsymbol{P}, \boldsymbol{t}, T')$. Then $\boldsymbol{x}$ is a vertex if and only if these two conditions hold: (i) $G(\boldsymbol{x})$ is a forest, and (ii) each connected component of $G(\boldsymbol{x})$ contains at most one machine-node i for which $C'_{\boldsymbol{x}}(i) < T'$.*

Proof of Lemma 6. First, we will prove that if $G(\boldsymbol{x})$ is a forest, then (i) and (ii) hold true. By contradiction, assume that $G(\boldsymbol{x})$ is a proper pseudo forest; that is, there exists a connected component that contains a cycle. By relabeling jobs and machines, we can assume that the edges of the cycle are given by the non-zero fractional variables $x_{11}, x_{21}, x_{22}, x_{32}, \ldots, x_{k,k}, x_{1,k}$. We will show that $\boldsymbol{x}$ is not a vertex by proving that both $(\boldsymbol{x} + \boldsymbol{\epsilon})$ and $(\boldsymbol{x} - \boldsymbol{\epsilon})$ are feasible for a vector $\boldsymbol{\epsilon}$ of length $n \cdot m$. Clearly, $\boldsymbol{\epsilon}_{ij} = 0$ must hold every time $x_{ij} = 0$ or $x_{ij} = 1$, so it is enough to consider variables corresponding to edges of $G(\boldsymbol{x})$. We claim that we can find an appropriate $\boldsymbol{\epsilon}$ that is non-zero only on these edges. For this to hold, we clearly need that $\epsilon_{11} = -\epsilon_{1,k}, \ldots, \epsilon_{k,k-1} = -\epsilon_{k,k}$, so we can write the desired vector in the form $\epsilon_{11} = \epsilon_1, \epsilon_{1,k} = -\epsilon_1, \ldots, \epsilon_{k,k-1} = \epsilon_k, \epsilon_{k,k} = -\epsilon_k$.

In order for $\boldsymbol{x} + \boldsymbol{\epsilon}$ to have the same (or smaller) makespan as $\boldsymbol{x}$, we need the following conditions to be satisfied for each $i = 1, \ldots, k$ (for $i = k$, $i + 1$ is to be understood as 1):

$$(x_{i,i} + \epsilon_i) \cdot p_{i,i} + (x_{i+1,i} - \epsilon_{i+1}) \cdot p_{i+1,1} \leq x_{i,i} \cdot p_{i,i} + x_{i+1,i} \cdot p_{i+1,i},$$

or equivalently,

$$\epsilon_i \cdot p_{i,i} - \epsilon_{i+1} \cdot p_{i+1,1} \leq 0,$$

and

$$\frac{\epsilon_i}{\epsilon_{i+1}} \leq \frac{p_{i+1,i}}{p_{i,i}}.$$

Since we are in the uniform case, we know that $p_{j,k} = \frac{p_j}{s_k}$ with a profit p_j and machine speed s_k, and so we need that

$$\frac{\epsilon_i}{\epsilon_{i+1}} \leq \frac{p_{i+1}/s_i}{p_i/s_i} = \frac{p_{i+1}}{p_i}, \quad i = 1, \ldots, k. \tag{8}$$

Let us choose $\epsilon_1 > 0$ arbitrarily, then we recursively define

$$\epsilon_{i+1} := \epsilon_i \cdot \frac{p_i}{p_{i+1}}.$$

These trivially satisfy with equality each inequality from (8) apart from the $i = k$ case; but since the product of all the left-hand sides, as well as the product of all right-hand sides, is equal to 1, the remaining inequality must be satisfied (with equality) as well.

Apart from (8), the values of the parameters ϵ_i must adhere to the constraints

$$\begin{cases} x_{i,i} + \epsilon_i \leq 1, & i = 1, \dots k, \\ x_{i+1,i} - \epsilon_i \geq 0, & i = 1, \dots, k. \end{cases} \tag{9}$$

But these are satisfied if the values for ϵ_i are chosen to be small enough since $0 < x_{i,i}, x_{i+1,i} < 1$. Notice that multiplying each ϵ_i with the same constant does not change the fractions in (8); so by choosing an appropriately small constant, we can guarantee that both (8) and (9) are satisfied.

Repeating a similar reasoning, we gather that the same $\boldsymbol{\epsilon}$ satisfies the corresponding versions of (8) for $\boldsymbol{x} - \boldsymbol{\epsilon}$ as well, and if $\mathbf{0} \leq \boldsymbol{x} - \boldsymbol{\epsilon} \leq \mathbf{1}$ is not satisfied, we can again multiply $\boldsymbol{\epsilon}$ with a small enough constant to guarantee these inequalities while maintaining the feasibility of the makespan constraints. In conclusion, if $G(\boldsymbol{x})$ has a cycle, $\boldsymbol{x}$ cannot be a vertex.

Now, suppose that $\boldsymbol{x}$ is a vertex and assume for contradiction that there are at least two machines in the same component of $G(\boldsymbol{x})$, i_1, and i_2, whose completion times are strictly smaller than T'. Consider a path connecting these two nodes, and observe that we can construct a vector $\boldsymbol{\epsilon} \in \mathbb{R}^{n \times m}_{\geq 0}$ which is zero apart from the coordinates of the path, and for which $(\boldsymbol{x} \pm \boldsymbol{\epsilon})$ are feasible. To do so, notice that we can repeat the process we described for the case of having a cycle, with the sole difference that in (8) the first and last inequalities lack one of the variables, and (9) has two less constraints. Therefore, the solution we derived for cycles is feasible for paths as well. This construction in fact can be seen as a special case of having a cycle, by splitting a "dummy job" between machines i_1 and i_2.

Conversely, suppose that $G(\boldsymbol{x})$ does not contain a cycle, and each connected component has at most one machine with completion time strictly smaller than T'. Suppose for contradiction that there is a vector $\boldsymbol{\epsilon}$ such that $(\boldsymbol{x} + \boldsymbol{\epsilon})$ and $(\boldsymbol{x} - \boldsymbol{\epsilon})$ are both feasible. The coordinates of $\boldsymbol{\epsilon}$ which are different from 0 and 1 must correspond to edges in $G(\boldsymbol{x})$; and as it does not contain a cycle, the subgraph spanned by these edges must be a forest. Consider an arbitrary connected component of this forest having at least two nodes; this must have at least two nodes of degree 1. If any of these nodes correspond to a job j, the constraints

$$\sum_{i=1}^{n} x_{j,i} = 1$$

would be violated by both $\boldsymbol{x} \pm \boldsymbol{\epsilon}$. Therefore, we have two machines, i_1 and i_2, that have degree one and belong to the same connected component in the subgraph spanned by the fractional coordinates of $\boldsymbol{\epsilon}$. Furthermore, since the connected components of this subgraph must be part of some connected components of $G(\boldsymbol{x})$, by assumption we have that at least one of $C'_{\boldsymbol{x}}(i_1) = T'$ or $C'_{\boldsymbol{x}}(i_2) = T'$ holds; let us assume that it is the first one. But then $C'_{\boldsymbol{x} \pm \boldsymbol{\epsilon}}(i_1) > T'$ would hold for exactly one of $\boldsymbol{x} \pm \boldsymbol{\epsilon}$, and so this vector would not be feasible. ◀

Proof of Lemma 5. If job n is fractionally assigned in $\boldsymbol{x}^*$, we can choose $\hat{\boldsymbol{x}} = \boldsymbol{x}^*$. Otherwise, assume that job n is integrally assigned to a machine; by relabeling machines, we can further assume that it is integrally assigned to machine 1 and hence $x^*_{n,1} = 1$. Since $\boldsymbol{x^*}$ is fractional,

there must exist a job j which is assigned fractionally. We distinguish between two cases. If there is a job that has a fractional part assigned to machine 1, then let j be this job. Apart from $x^*_{j,1}$, there must be another nonzero coordinate for job j; we can assume that it is $x^*_{j,2}$. If there is no job having a fractional coordinate at machine 1, let j be an arbitrary fractional job and assume that $x^*_{j,2} \neq 0$.

Let ϵ_1 and ϵ_2 be parameters whose value we fix later, and consider the following vector $\hat{\boldsymbol{x}}$:

$$\hat{x}_{k,i} = \begin{cases} x^*_{n,1} - \epsilon_1 = 1 - \epsilon_1, & \text{if } (k,i) = (n,1), \\ x^*_{n,2} + \epsilon_1 = \epsilon_1, & \text{if } (k,i) = (n,2), \\ x^*_{j,1} + \epsilon_2, & \text{if } (k,i) = (j,1), \\ x^*_{j,2} - \epsilon_2, & \text{if } (k,i) = (j,2), \\ x^*_{k,i}, & \text{else.} \end{cases}$$

Let us choose the values of ϵ_1 and ϵ_2 such that the completion times of machines 1 and 2 are the same in $\boldsymbol{x}^*$ and $\hat{\boldsymbol{x}}$. For this to hold, we need that

$$-\epsilon_1 \cdot \frac{p_n}{s_1} + \epsilon_2 \cdot \frac{p_j}{s_1} = 0,$$

and

$$+\epsilon_1 \cdot \frac{p_n}{s_2} - \epsilon_2 \cdot \frac{p_j}{s_2} = 0.$$

These equalities are satisfied by any ϵ_1, ϵ_2 for which

$$\frac{\epsilon_1}{\epsilon_2} = \frac{p_j}{p_n}$$

holds. Let us choose $\epsilon_2 = \min\{1 - x^*_{j,1}, x^*_{j,2}\} = x^*_{j,2} < 1$, and $\epsilon_1 = \epsilon_2 \cdot \frac{p_j}{p_n}$. With this choice, it also holds that $\hat{x}_{j,2} = 0$, $\hat{x}_{j,1} \leq 1$; and $\epsilon_1 = \epsilon_2 \cdot \frac{p_j}{p_n} \leq \epsilon_2 < 1$ implies that $\hat{x}_{n,1} > 0$, $\hat{x}_{n,2} < 1$ is also true.

In other words, the modification splits job n fractionally between machines 1 and 2, while relocating the fractional part of job j from machine 2 to machine 1. The resulting $\hat{\boldsymbol{x}}$ is feasible for the corresponding polyhedra; and by Lemma 6, it is enough to guarantee that $G(\hat{\boldsymbol{x}})$ is a forest for concluding that $\hat{\boldsymbol{x}}$ is a vertex, since the completion times of machines are left unchanged.

Let u_1, u_2, w_n, w_j be the nodes of $G(\hat{\boldsymbol{x}})$ corresponding to machine 1, machine 2, job n and job j, respectively. We have that $w_n u_1, w_n u_2, w_j u_1 \in E(G(\hat{\boldsymbol{x}}))$ and $w_j u_2 \notin E(G(\hat{\boldsymbol{x}}))$. We also know that $\deg_{G(\hat{\boldsymbol{x}})}(w_n) = 2$ and $w_n \notin V(G(\boldsymbol{x}^*))$.

Consider the first case, when job j has a fractional part on machine 1 in $\boldsymbol{x}^*$. Suppose for contradiction that there is a cycle C in $G(\hat{\boldsymbol{x}})$. If $w_n \notin V(C)$, then $C \subseteq G(\boldsymbol{x}^*)$ would hold. If $w_n \in V(C)$, then $P := C - \{w_n u_1, w_n u_2\}$ would be a $u_1 - u_2$-path such that $P \subseteq G(\boldsymbol{x}^*)$ and $P \triangle \{w_j u_1, w_j u_2\}$ would contain a cycle in $G(\boldsymbol{x}^*)$, where $\triangle$ denotes the symmetric difference of two sets.

In the second case, $\deg_{G(\hat{\boldsymbol{x}})}(u_1) \leq 2$, $\deg_{G(\boldsymbol{x}^*)}(u_1) = 0$, $w_j u_1 \notin E(G(\boldsymbol{x}^*))$. Assume for contradiction that there is a cycle C in $G(\hat{\boldsymbol{x}})$. If C does not contain any of the edges $w_j u_1, u_1 w_n, w_n u_2$, then $C \subseteq G(\boldsymbol{x}^*)$ would hold. Otherwise, since $\deg_{G(\hat{\boldsymbol{x}})}(u_1) \leq 2$ and $\deg_{G(\hat{\boldsymbol{x}})}(w_n) = 2$, C must contain all three edges. But then $P := C - \{w_j u_1, u_1 w_n, w_n u_2\}$ is a $u_2 - w_j$ path in $G(\boldsymbol{x}^*)$, which together with $u_2 w_j$ would form a cycle in $G(\boldsymbol{x}^*)$. ◀

With this, we are ready to define our final enhanced algorithm $A^{\text{sim-prof}}_\epsilon$. It takes as input an instance of the uniform machine scheduling problem $(\boldsymbol{P}, \boldsymbol{0})$ where $\boldsymbol{P}$ is given by

$(\boldsymbol{p}, \boldsymbol{s}) \in \mathbb{N}^{n+m}$. It rearranges the jobs such that $p_1 \geq \ldots \geq p_n$, then creates an equivalent instance $\boldsymbol{P'}$ by dividing $\boldsymbol{P}$ with the global fractional optimum. Then it proceeds as a branch-and-bound algorithm with the following specifications: when processing a node v, it first finds a vertex of the corresponding relaxation of the sub-problem (6) with the smallest T for which the program is feasible. If the vertex is integer, the algorithm stops, as it has found a globally optimal schedule (due to the best-first selection criterion). If the vertex is fractional and the longest unfixed job is not fractional, then it follows Lemma 5 to arrive at another optimal vertex in which the longest unfixed job is fractional. Then, it rounds up this vertex to find an integer solution, according to an arbitrary matching between machines and fractional jobs. The pivot element at node v will be the longest fractional job, which by now coincides with the longest unfixed job. m new branches are created, labelled by the machine on which the longest unfixed job is fixed at the next level. For each new node u, the algorithm checks whether it has already found a schedule with a makespan better than $L(u)$, in which case u is discarded. Next, $\Pi(u) = \boldsymbol{t}_u$ is compared with all previous profiles at the same depth. If $\max\{\Pi(u)_i : i \in [m]\} > 2 + 2\epsilon$, or there already exists a node at the same depth whose profile is similar to $\Pi(u)$, u is discarded. The remaining of the m new nodes are added to the list of active nodes, the list of profiles is appended with the new ones, and the next node to process is selected according to the best-first tree traversal rule.

The process terminates when the ratio between the makespan of the best discovered schedule and the lowest lower bound satisfies $\frac{GU}{GL} \leq 1 + \epsilon$, at which point it returns the best schedule found so far.

▶ **Lemma 7.** *Let F be the final branching tree traversed by algorithm $A_\epsilon^{sim\text{-}prof}$. The number of nodes in F is at most $n \cdot \left(\frac{5n}{\epsilon}\right)^m$.*

Proof of Lemma 7. By the property that each vertex is either integral (in which case the algorithm stops, having found the optimal integer schedule), or the longest unfixed job is fractional in the vertex, the "longest fractional job" pivot selection rule guarantees that at depth j, job j will be the pivot element. Furthermore, nodes at a given level in F must have profiles such that no two of them are similar, and we have seen in the beginning of the section that the cardinality of such a set is at most $\left(\frac{5n}{\epsilon}\right)^m$ for $\epsilon < 1$. Since the depth of F is at most n, the total number of nodes is at most $n \cdot \left(\frac{5n}{\epsilon}\right)^m$. If $\epsilon \geq 1$, the root node will provide a $(1 + \epsilon \geq)2$-approximative solution. ◀

Proof of Theorem 3. What remains to be shown is the approximation property of the algorithm. Let $\epsilon < 1$, and let $F' \supseteq F$ be the tree which we obtain by modifying $A_\epsilon^{\text{sim-prof}}$ to terminate only when it processed every node. It suffices to show that there is a leaf of F' at level n which corresponds to a schedule with a makespan at most $(1 + \epsilon)$ times the optimal one. Let S denote a (globally) optimal schedule, and suppose we got rid of it at level k by discarding a node v whose profile was similar to an already found profile of another node u. Let S' be the schedule which we obtain from S by rearranging the first k jobs according to the partial schedule corresponding to u. This modification incurs a change of at most $\frac{\epsilon}{n}$ on each machine, since the profiles of v and u were similar, and jobs $k+1, \ldots, n$ remained at their original machine. Therefore, the makespan of S' is greater than the makespan of S by at most $\frac{\epsilon}{n}$. Repeat this argument, if necessary, each time the champion schedule gets discarded because of similarity. Note that the discarding can never happen because of a too large makespan, since by induction each champion schedule has completion times at most $T_{\text{opt}} + n \cdot \frac{\epsilon}{n} = T_{\text{opt}} + \epsilon \leq 2 + \epsilon < 2 + 2\epsilon$. Each time, we can find an alternative schedule serving as the new champion, while losing only an additive factor of at most $\frac{\epsilon}{n}$. Altogether,

the final champion at level n will have a makespan at most $\frac{\epsilon}{n} \cdot n = \epsilon$ greater than the optimal one, concluding our proof. ◀

In what follows we show how the algorithm $A_\epsilon^{\text{sim-prof}}$ generalizes a dynamic programming approach for the machine scheduling problem. The latter consists of constructing a matrix $\boldsymbol{M}$, where the n rows are labeled by the n jobs in some fixed order. Column i pertains to a representative profile $\Pi(i)$ of the partition classes of the cube $[0, 2+2\epsilon]^m$ given by points $[0, \frac{\epsilon}{n}, \ldots, \frac{n(2+2\epsilon)}{\epsilon} \cdot \frac{\epsilon}{n}]^m$. The entry at column i and row j is 1 if there exists a partial assignment with the first j jobs whose profile is similar to $\Pi(i)$; otherwise, the entry is 0. The algorithm fills in the entries of the matrix by the best-first principle, then checks all 1-entries of the last row and determines the best makespan of the corresponding profiles. By our above reasoning, the returned schedule will be a $(1+\epsilon)$-approximate solution.

The embedding of the dynamic programming in $A_\epsilon^{\text{sim-prof}}$ can be described as follows: each level of the branching tree F has the same job fixed at every node, therefore level j contains nodes where the fixed jobs are $1, \ldots, j$. Moreover, we only prune a node when either its profile is similar to another one already found (implying that at most one profile is considered from each partition class), or its lower bound is worse than an already found integer solution. Therefore, nodes at depth j in F have a one-to-one correspondence with cells of the j-th row of M whose entry is 1, except for some profiles that were discarded for having a too-high lower bound. In other words, $A_\epsilon^{\text{sim-prof}}$ can be seen as the dynamic programming algorithm embedded in the branch-and-bound framework, where $\boldsymbol{M}$ is traversed according to the best-first logic, and some entries are disregarded when even the best possible extension of their profile is worse than an already found feasible solution. In the worst case, each cell of $\boldsymbol{M}$ is visited before a $(1+\epsilon)$-approximate solution is found; but it happens no later than the processing of the last entry, according to Theorem 3.

The embedding becomes even more evident if we replace the best-first node selection rule with BFS. Then, traversing level j of the tree is nothing else but processing the j-th row of $\boldsymbol{M}$ except for some entries that are stepped over because of their lower bound.

To conclude our work, we point out the infeasibility of repeating our results for the unrelated machine scheduling problem and the multiple knapsack problem. The notion of profiles and the $\frac{\epsilon}{n}$-partition of their space can be extended without changing anything. The difficulty lies in guaranteeing the highly structured property of the branching tree, in which the same job is fixed at all nodes of a given level. Of course, one can simply hard-code this into the algorithm, but giving a pivot rule that achieves this naturally seems infeasible. In particular, the following example shows that the "maximal shortest processing time" selection rule does not have this guarantee: let $m = 3$ and consider the following input with $n = 2k+2$ jobs: $p_{1,1} = p_{1,2} = p_{1,3} = 3k+2$, $p_{j,2} = p_{j,3} = 3$, $j = 2, \ldots, n-1$ and $p_{n,2} = p_{n,3} = 2$. The rest of the processing times are chosen such that $p_{j,1} \leq 3k+1$, $j = 2, \ldots, n$. It is easy to check that in the second iteration, there is no vertex in the polyhedron where the job with the maximal shortest processing time is fractional. This instance also serves as a counterexample for a bunch of other pivot selection strategies, such as "maximal average completion time" or "maximal longest processing time".

A similar phenomenon takes place in the case of the (multiple) knapsack problem, with the exception that we *know* the infeasibility of having a structure where each node in the same level has the same job fixed. In particular, for the single knapsack problem, the two children of a given node have different pivot elements (provided that they are both feasible).

▶ **Lemma 3.** *Let $(C, \boldsymbol{w}, \boldsymbol{p})$ denote an input to the single knapsack problem. Assume that the items are such that $\frac{p_1}{w_1} > \ldots > \frac{p_n}{w_n}$, and the pivot element is j^*. Let $(C - w_{j^*}, \boldsymbol{w'}, \boldsymbol{p'})$*

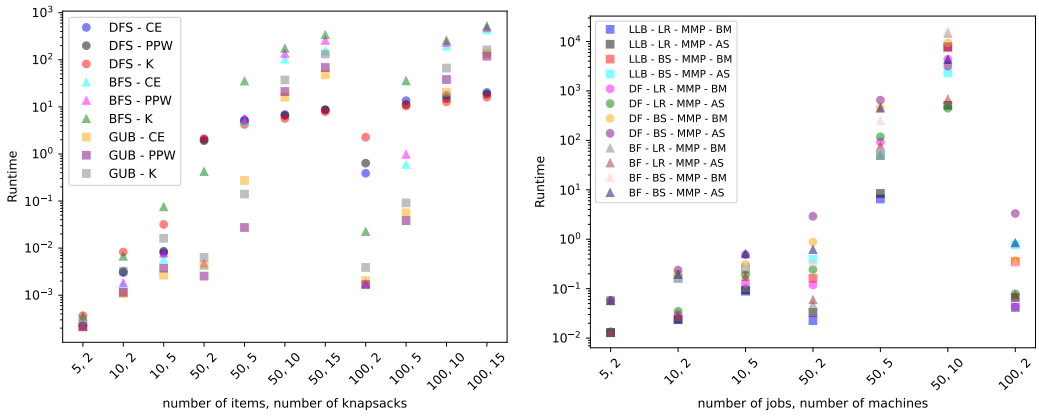


Figure 3 Performance of different strategies in the branch-and-bound method for the Multi-Knapsack and Unrelated Machine Scheduling problems. The runtime (in seconds) is reported on a logarithmic scale.

and $(C, \boldsymbol{w}', \boldsymbol{p}')$ *denote the two subproblems corresponding to including and excluding item* j^* *from the knapsack, with* $\boldsymbol{w}' = \boldsymbol{w}|_{[n]-j^*}$ *and* $\boldsymbol{p}' = \boldsymbol{p}|_{[n]-j^*}$. *Assume that both subproblems are feasible, and the corresponding pivot elements are* j_1 *and* j_2. *Then* $j_1 < j^* < j_2$.

Proof. Items $1, \ldots, j^* - 1$ do not fit inside the knapsack with reduced capacity $C - w_{j^*}$ (because items $1, \ldots, j^* - 1, j^*$ did not fit inside the knapsack with original capacity C), but they do fit inside the original capacity C. Hence, $j_1 < j^* < j_2$. ◀

8 Analysis of the runtime of the proposed algorithm

In Figure 3, we report the runtime of our proposed algorithm for the Multi-Knapsack problem (left) and the Unrelated Machine Job Scheduling problem (right).

8.1 Multiple Knapsacks

First, we observe that the runtime is quite long, even for relatively small instances in both cases. As previously mentioned, our implementation is not highly optimized. To put this into perspective, solving the benchmark instance $(100, 15)$ in the Multi-Knapsack framework typically requires 7.82 ± 4.40 seconds with the B&B implemented in SCIP[2], which is significantly shorter than our results.

However, an interesting pattern emerges from the analysis of our strategies. For the Multi-Knapsack problem, we observe that when the number of knapsacks is ≥ 10, DFS outperforms GUB in terms of speed. We suspect this is because, in the Multi-Knapsack setting, the DFS strategy consistently reaches the node exploration limit, set to 10^4. As a result, it likely prunes many nodes quickly due to infeasibility or bounding, avoiding complex computations on the explored nodes. Hence, we arrive at the node limit faster.

[2] In this case, for fairness, we disable presolve, cutting plane, heuristics, restarts, and propagation, and set a branching nodes limit equal to 10^4

8.2 Unrelated machine scheduling problem

As in the previous section, we compare our approach with SCIP for the case $(50, 10)$, where we observe a runtime of 10.051 ± 1.63, which is again shorter than the observed runtime of our algorithm.

In this case, we find that the LLB strategy results in faster solutions. This outcome is expected: since all nodes are explored and pruning by infeasibility is not possible (as every subproblem remains feasible), the time required per node remains roughly the same.

Since the LLB strategy achieves better solutions with fewer node explorations, we can conclude that our theoretical expectations align with the experimental results.

9 Pseudo-code of A_α^{knap} and A_ϵ^{unrel}

Pseudocode for A_α^{knap}

We provide a pseudo-code for our algorithm A_α^{knap} in this section. Pseudocodes for the other algorithms can be derived in an almost identical fashion.

Algorithm 1 Subroutine

Input: Problem instance, a threshold that we wish to guarantee to our solution quality
Output: High-quality solution
Do any necessary preprocessing
Initialize global lower bound (GLB) and global upper bound (GUB) (best feasible solution)
Compute initial lower bound using a relaxation method
if is integer **then return**
end if
Initialize priority queue (heap) with root node
while queue is not empty **do**
    Extract the most promising node from the queue
    if node is integral **then**
        Update GUB* if a better solution is found
        continue
    end if
    Select a branching item/job
    for each possible branch (child node) **do**
        Apply feasibility check
        Compute new upper bound and lower bound
        if new lower bound < GUB* **then**
            Add new node to the queue
        end if
    end for
    Update GLB* as max remaining lower bound in queue
    if A relation between GUB, GLB and the threshold is satisfied **then**
        return
    end if
end while

First, we define the subroutine that returns the fractional optimum of a given knapsack instance, along with the set of critical elements and rounded-up integer solution:

We describe the algorithm in detail below. Each node v will be identified by the unique sets (I, E) with $I = \{i_1, \dots i_k\}$ being the included items and $E = \{j_1, \dots, j_l\}$ being the excluded items. Let A denote the subroutine that on input $(C, \mathbf{w}, \mathbf{p})$, returns the triple (i^*, x^*, x').

$\mathbf{A}_\alpha(C, \mathbf{w}, \mathbf{p})$:

1. $n \leftarrow \text{len}(\mathbf{w}) = \text{len}(\mathbf{p})$.
2. $r \leftarrow (\emptyset, \emptyset)$, $(i^*, x^*, x') \leftarrow A(C, \mathbf{w}, \mathbf{p})$.
3. $U(r) \leftarrow \mathbf{p} \cdot x^*$, $L(r) = \mathbf{p} \cdot x'$.
4. $GU = U(r)$, $GL = L(r)$.
5. $S \leftarrow \{r\}$.
6. while $\frac{GL}{GU} \leq \alpha$:
   a. $v \leftarrow \arg\max\{U(s) : s \in S\}$, $v = (I, E)$.
   b. $S \leftarrow S \setminus \{v\}$.
   c. $C' \leftarrow C - \sum_{i \in I} w_i$, $\mathbf{w}' = \mathbf{w}|_{[n]\setminus(I \cup E)}$, $\mathbf{p} = \mathbf{p}|_{[n]\setminus(I \cup E)}$
   d. $(i^*, x^*, x') \leftarrow A(C', \mathbf{w}', \mathbf{p}')$.
   e. $u \leftarrow (I \cup \{i^*\}, E)$, $w \leftarrow (I, E \cup \{i^*\})$.
   f. $C_w \leftarrow C'$, $C_u \leftarrow C' - w_{i^*}$.
   g. $\mathbf{w}_u = \mathbf{w}_w = \mathbf{w}|_{[n]\setminus(I \cup E \cup \{i^*\})}$, $\mathbf{p}_u = \mathbf{p}_w = \mathbf{p}|_{[n]\setminus(I \cup E \cup \{i^*\})}$.
   h. $(i^*_u, x^*_u, x'_u) \leftarrow A(C_u, \mathbf{w}_u, \mathbf{p}_u)$, $(i^*_w, x^*_w, x'_w) \leftarrow A(C_w, \mathbf{w}_w, \mathbf{p}_w)$.
   i. $SU(u) \leftarrow \mathbf{p}_u \cdot x^*_u$, $SL(u) \leftarrow \mathbf{p}_u \cdot x'_u$.
   j. $SU(w) \leftarrow \mathbf{p}_w \cdot x^*_w$, $SL(w) \leftarrow \mathbf{p}_w \cdot x'_w$.
   k. $U(u) \leftarrow SU(u) + \sum_{i \in I \cup \{i^*\}} p_i$, $L(u) \leftarrow SL(u) + \sum_{i \in I \cup \{i^*\}} p_i$.
   l. $U(w) \leftarrow SU(w) + \sum_{i \in I} p_i$, $L(w) \leftarrow SL(w) + \sum_{i \in I} p_i$.
   m. if $U(u) > GL$: $S \leftarrow S \cup \{u\}$.
   n. if $U(w) > GL$: $S \leftarrow S \cup \{w\}$.
   o. $GU = \max\{U(s) : s \in S\}$.
   p. $GL = \max\{L(s) : s \in S\}$.
7. $v \leftarrow \arg\max\{U(s) : s \in S\}$, $v = (I, E)$.
8. $C' \leftarrow C - \sum_{i \in I} w_i$, $\mathbf{w}' = \mathbf{w}|_{[n]\setminus(I \cup E)}$, $\mathbf{p}' = \mathbf{p}|_{[n]\setminus(I \cup E)}$.
9. $(i^*, x^*, x') \leftarrow A(C', \mathbf{w}', \mathbf{p}')$.
10. return $I_{\text{approx}} = I \cup \{i : x'_i > 0\}$.

9.1 Relation between a general B&B code and our algorithms

As already discussed in Section 1, any B&B methods run some basic functions that can be highly customized. Algorithm 2 details a general B&B framework for a minimization problem, but a similar argument can occur with a maximization one.

In the remaining of this section, we describe how we specialize each line of Algorithm 2 for the algorithms under study.

Algorithm 2 Branch and Bound Algorithm. The steps denoted with $*$ must be changed when switching from minimization to maximization.

```
Input: Problem instance, a threshold that we wish to guarantee to our solution quality
Output: High-quality solution
Do any necessary preprocessing
Initialize global lower bound (GLB) and global upper bound (GUB) (best feasible solution)
Compute initial lower bound using a relaxation method
if is integer then return
end if
Initialize priority queue (heap) with root node
while queue is not empty do
    Extract the most promising node from the queue
    if node is integral then
        Update GUB* if a better solution is found
        continue
    end if
    Select a branching item/job
    for each possible branch (child node) do
        Apply feasibility check
        Compute new upper bound and lower bound
        if new lower bound < GUB* then
            Add new node to the queue
        end if
    end for
    Update GLB* as max remaining lower bound in queue
    if A relation between GUB, GLB and the threshold is satisfied then
        return
    end if
end while
```

9.1.1 Multiple Knapsacks

- (Line 1): The problem instance is given by the vector of the capacities $\boldsymbol{C}$ and the vectors of weights and profits $\boldsymbol{w}, \boldsymbol{p}$. The threshold is $0 < \alpha < 1$, as described in Section 2.
- (Line 3): As preprocessing, we discard all the items that cannot be assigned integrally to any knapsack, that is, we assign them to a fictional knapsack $m + 1$
- (Line 4): GLB is initialized to $-\infty$, while GUB to ∞.
- (Line 5) We compute a standard linear relaxation by solving (1) under (2a) – (2c), we hence obtain an upper bound (UB) and a fractional solution $\boldsymbol{x}$.
- (Line 6). If the solution is integral, then we found the global optimum, that for sure satisfies the quality condition enforced at Line 2. Hence, the algorithm stops with the optimal solution.
- (Line 8). We initialize a priority queue: according to the strategy, we pop the node having the maximum lowerbound (GUB), the maximal depth (DF), or the minimal depth (BF). We add to the priority queue *nodes*, that are objects having and UB the point $\boldsymbol{x}$ attaining such UB; from $\boldsymbol{x}$, we derive an integer solution $\boldsymbol{y}$ associated with a value LB, that is, the *lower bound* of the optimal solution. We also carry on $\boldsymbol{C}, \boldsymbol{p}, \boldsymbol{w}$ as obtained after the preprocessing.
- (Line 9 – 10). While the queue is not empty, we pop the best node.
- (Line 11 – 14) If the solution x is integral, then we cannot branch on any variable. If provides a better lower bound, we update it (*prune by integrality*).
- (Line 15) On another hand, we select a variable to branch on. This can be done with several strategies. In this framework, a variable x_{ij} represents the assignment of item i to knapsack j. Let this be i^*.
- (Line 16) We fix x_{i^*j}, $j \in \{1, \dots, m+1\}$ as assigning item i^* to knapsack j. We then reduce each capacity C_j by the quantity w_{i^*}. We discard all the items that cannot fit integrally anymore, similarly to what we did in Line 3.
- (Line 17) We apply a feasibility check, and discard all the nodes that are not feasible (*prune by infeasibility*).
- (Line 18 – 20) We then compute a new upper bound by solving (1) under (2a) – (2c) using the new capacities, weights and profits. If the upper bound we get is smaller than GLB, it means that that branch cannot give an optimal solution. Hence, we *prune by bound*. Otherwise, we add the node to the queue.
- (Line 23) We update GUB as the greatest upper bound among the active nodes
- (Line 24) Lastly, we do the quality check. Note that if $\frac{\text{GLB}}{\text{GUB}} \geq \alpha$, we return with a solution of the desired quality.

9.1.2 Unrelated machine scheduling

- (Line 1): The problem instance is given by the vector of the processing times. The threshold is $0 < \epsilon$, as described in Section 3.
- (Line 3): Here, we do not do any particular preprocessing.
- (Line 4): GLB is initialized to $-\infty$, while GUB to ∞.
- (Line 5) We compute a lower bound LB by solving via binary search (6), obtaining also a fractional solution $\boldsymbol{x}$.
- (Line 6). If the solution is integral, then we found the global optimum, that for sure satisfy the quality condition enforced at Line 2. Hence, we stop the algorithm, returning the values at the root node.

- (Line 8). We initialize a priority queue: acording to the strategy, we pop the node having the lowest lowerbound (LLB), the maximal depth (DF), or the minimal depth (BF). We add to the priority queue *nodes*, that are objects having and LB the point $\boldsymbol{x}$ attaining such LB; from $\boldsymbol{x}$, we derive an integer solution $\boldsymbol{y}$ associated with an upper bound UB.
- (Line 9 – 10). While the queue is not empty, we pop the best node
- (Line 11 – 14) If the solution x is integral, then we cannot branch on any variable. If provides a better upper bound, we update it (*prune by integrality*).
- (Line 15) On another hand, we select a variable to branch on. This can be done with several strategies. In this framework, a variable x_{ij} represents the assignment of job i to machine j. Let this be i^*
- (Line 16) We fix x_{i^*j}, $j \in \{1, \ldots, m+1\}$. In the binary search strategy, we reduce the quantity T on machine j by p_{i^*}. .
- (Line 17) Here, all the nodes are feasible.
- (Line 18 – 20) We then compute a new upper bound by solving (6) using the reduces values of T in the binary search. If the lower bound we get is greater than GUB, it means that that branch cannot give an optimal solution. Hence, we *prune by bound*. Otherwise, we add the node to the queue.
- (Line 23) We update GLB as the smallest lower bound among the active nodes
- (Line 24) Lastly, we do the quality check. Note that if $\frac{\text{GUB}}{\text{GLB}} \leq 1 + \epsilon$, we return with a solution of the desired quality.