Master Thesis
In the Course of Studies Mathematik (M.Sc.)
At the Institut für Mathematische Optimierung

A Branch- and Bound Method for Bilevel Parallel Machine Scheduling with Adversarial Job Selection

Ole Damke
5029861
o.damke@tu-braunschweig.de

First Reviewer: Prof. Dr. Maximilian Merkert
Institut für Mathematische Optimierung
Second Reviewer: Univ.-Prof. Dr. rer. nat. habil. Christian Kirches
Institut für Mathematische Optimierung

Submission Date: February 2, 2026

Abstract

This thesis presents a branch-and-bound algorithm for solving a bilevel optimization problem combining knapsack selection with scheduling. The leader selects for each given job type a number of jobs within a budget constraint to maximize system capacity (makespan), while the follower optimally schedules the selected jobs across machines to minimize the makespan. We develop tight upper bounds using a knapsack scheduling relaxation and implement efficient pruning strategies. Computational experiments demonstrate speedups of up to 188× compared to complete enumeration, with an average pruning rate of 37%. The algorithm efficiently solves instances with up to 10 job types and 10 machines, providing optimal solutions in seconds where enumeration requires hours or is infeasible.

Keywords: Bilevel optimization, branch-and-bound, knapsack problem, scheduling, combinatorial optimization

appendix

Contents

A. Introduction

A.1. Motivation

Resource allocation under uncertainty is a fundamental challenge in many real-world systems. Consider a production facility that must decide which types of jobs to accept within a limited budget, knowing that a scheduler will later assign these jobs to machines to minimize completion time. The facility manager wants to maximize system capacity while respecting budget constraints, but cannot directly control the scheduling decisions.

This scenario exemplifies a *bilevel optimization problem*, where one decision-maker (the leader) makes choices that affect another decision-maker (the follower), who then optimizes their own objective. Such problems arise in supply chain management, network design, resource allocation, and competitive markets.

A.2. Problem Overview

We study a bilevel optimization problem combining two classic operations research problems:

- **Upper level (Leader):** Knapsack problem – select job types within a budget
- **Lower level (Follower):** Scheduling problem – assign jobs to machines to minimize makespan

The leader's objective is to maximize the makespan (completion time). This models a worst-case robust optimization approach, where the leader tries to account for the worst possible makespan, even when the follower actually minimizes completion time by finding an optimal schedule.

A.3. Contributions

Throughout this thesis, we refer to this problem as the *bilevel knapsack-scheduling problem*, which concisely describes the combination of knapsack-based job selection at the upper level and parallel machine scheduling at the lower level with adversarial job selection.

This thesis makes the following contributions:

1. Formal mathematical model of the bilevel knapsack-scheduling problem
2. Branch-and-bound algorithm with tight bounds based on a knapsack scheduling relaxation
3. Efficient dynamic programming solver for the bounding subproblem
4. Comprehensive computational study demonstrating practical scalability
5. Analysis of pruning effectiveness and comparison with complete enumeration

A.4. Organization

The remainder of this thesis is organized as follows. Section B presents the mathematical formulation of the bilevel problem. Section C describes the branch-and-bound algorithm and bounding techniques. Section D discusses implementation details. Section E reports computational results. Section F concludes with a summary and future research directions.

B. Problem Formulation

B.1. Bilevel Optimization Framework

A bilevel optimization problem has a hierarchical structure with two decision-makers:

Definition B.1.1 (Bilevel Optimization Problem)**.**

$$\max_x \quad F(x, y^*(x)) \tag{B.1}$$

$$\text{s.t.} \quad x \in X$$

$$\text{where } y^*(x) \in \arg\min_y \{f(x, y) : y \in Y(x)\} \tag{B.2}$$

Note B.1. This general framework encompasses a wide range of hierarchical optimization problems. Additional theoretical aspects, such as solution existence, uniqueness conditions, and computational complexity classes, may be discussed in future extensions of this work.

The leader chooses x to optimize their objective F, anticipating that the follower will respond by solving their own optimization problem to find $y^*(x)$.

B.2. The Knapsack-Scheduling Problem

B.2.1. Problem Data

- n job types, indexed by $i = 1, \ldots, n$
- Job type i has:
  - Processing time (duration): $d_i > 0$
  - Cost (price): $p_i > 0$
- m identical parallel machines
- Total budget: $B > 0$

B.2.2. Leader's Problem (Upper Level)

The leader selects how many jobs of each type to purchase:

Decision variables: $x_i \in \mathbb{Z}_+$ = number of jobs of type i to select

Constraints:

$$\sum_{i=1}^{n} p_i x_i \leq B \quad \text{(budget constraint)} \tag{B.3}$$

Objective: Maximize the makespan resulting from optimal scheduling:

$$\max_x \quad C^*_{\max}(x) \tag{B.4}$$

where $C^*_{\max}(x)$ is the optimal makespan for the follower's problem given selection x.

B.2.3. Follower's Problem (Lower Level)

Given the leader's selection $x = (x_1, \ldots, x_n)$, the follower has a total of $\sum_{i=1}^{n} x_i$ jobs to schedule.

Decision variables: For each job j, assign it to some machine

Objective: Minimize makespan (maximum machine load)

$$C^*_{\max}(x) = \min \max_{k=1,\ldots,m} L_k \tag{B.5}$$

where L_k denotes the load on machine k, defined as the total processing time of all jobs assigned to machine k after scheduling.

B.3. Complete Bilevel Formulation

$$\max_{x \in \mathbb{Z}^n_+} \quad C^*_{\max}(x) \tag{B.6}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} p_i x_i \leq B \tag{B.7}$$

$$\text{where } C^*_{\max}(x) = \quad \min_{y} \max_{k=1}^{m} \sum_{i=1}^{n} \sum_{j=1}^{x_i} d_i \cdot y_{ijk} \tag{B.8}$$

$$\text{s.t.} \quad \sum_{k=1}^{m} y_{ijk} = 1 \quad \forall i, j \tag{B.9}$$

$$y_{ijk} \in \{0, 1\} \quad \forall i, j, k$$

Here, $y_{ijk} = 1$ if copy j of job type i is assigned to machine k.

B.4. Problem Characteristics

- **Complexity:** The problem is NP-hard, combining two NP-hard problems (knapsack and scheduling)
- **Non-convexity:** The follower's optimal value function $C^*_{\max}(x)$ is non-convex and discontinuous
- **Discrete bilevel:** Cannot use KKT conditions (only valid for continuous problems)
- **Enumerative approach needed:** Must explore discrete solution space

B.5. Interpretation: Why Maximize Makespan?

The leader maximizes makespan to ensure robust system capacity. This models scenarios where:

- The leader wants to stresstest the system capacity
- The follower (scheduler) will always try to minimize completion time by finding an optimal schedule
- Models adversarial or worst-case planning

Example: A production manager selects job types (leader) knowing that a scheduler will minimize completion time (follower). The manager wants to maximize the workload the system can handle.

C. Solution Methodology

Having formulated the bilevel knapsack-scheduling problem in Chapter B, we now develop solution methods to find optimal or near-optimal solutions efficiently. We begin by discussing a baseline approach based on complete enumeration, which provides a reference for understanding the computational challenges. We then present our main contribution: a branch-and-bound algorithm that systematically explores the search space while using tight upper bounds to prune branches that cannot contain optimal solutions. The key components—branching rules, upper bound computation via a knapsack scheduling relaxation, and efficient dynamic programming for the bounding subproblem—are developed in detail throughout this chapter.

C.1. Complete Enumeration (Baseline)

C.1.1. Algorithm

The most straightforward approach is to enumerate all feasible selections and solve the follower's problem for each:

Algorithm 1 Complete Enumeration for Bilevel Problem

$C^*_{\max} \leftarrow 0$, $x^* \leftarrow \emptyset$
for each feasible selection x satisfying budget constraint **do**
    Solve follower's scheduling problem to get $C_{\max}(x)$
    if $C_{\max}(x) > C^*_{\max}$ **then**
        $C^*_{\max} \leftarrow C_{\max}(x)$
        $x^* \leftarrow x$
    end if
end for
return $(x^*, C^*_{\max})$

C.1.2. Complexity Analysis

The search space grows exponentially:

- Maximum copies of item i: $\lfloor B/p_i \rfloor$
- Total combinations: $\prod_{i=1}^{n} (\lfloor B/p_i \rfloor + 1)$
- For each combination: Check budget constraint and solve scheduling problem after, which in itself is NP-hard.

Example: With $n = 10$ items, budget $B = 100$, and prices $p_i \in [5, 15]$:

- Search space $\approx 10^{10}$ to 10^{12} combinations

- At 1ms per scheduling solve: 115 days to 31 years
- Clearly intractable for practical instances

C.2. Branch-and-Bound Algorithm

C.2.1. Overview

To overcome the combinatorial explosion of complete enumeration, we develop a branch-and-bound algorithm. The Branch-and-bound algorithm systematically explores the search space while pruning branches that cannot contain optimal solutions. Here, all possible selections are organized in a search tree, where each node represents a partial selection of items and each leaf node represents a complete selection. The algorithm consists of three main components:

1. **Branching:** We decompose the problem into subproblems
2. **Bounding:** We compute an upper bound on the best possible solution in a subtree
3. **Pruning:** We eliminate subtrees that cannot improve the incumbent

C.2.2. Search Tree Structure

Node representation: Each node represents a partial solution and holds all of the following information:

- **Depth** d**:** All items $1, \ldots, d$ have fixed quantities.
- **Occurrences:** $x = [x_1, \ldots, x_d, ?, \ldots, ?]$
- **Remaining budget:** $B - \sum_{i=1}^{d} p_i x_i$

Branching rule: From depth d, create children by setting x_{d+1}:

- Branch for $x_{d+1} = 0, 1, 2, \ldots, \lfloor B_{\text{rem}}/p_{d+1} \rfloor$

Example search tree: Figure C.1 illustrates the branching structure for a problem with 2 items, prices $p_1 = 5$, $p_2 = 8$, and budget $B = 10$.

C.2.3. Upper Bound Computation

Key idea: In the branch-and-bound tree we systematically explore the search space to find the combination of items that maximizes the result of the follower's problem. Given an incumbent solution with makespan $C_{\max}^{\text{incumbent}}$, we don't want to explore subtrees that cannot yield a better solution.

At a given subtree rooted at depth d, we have already fixed the quantities of item types $1, \ldots, d$. If we can compute an upper bound $\text{UB}(x_1, \ldots, x_d, B_{\text{rem}})$ on the best possible makespan achievable by completing the selection with item types $d+1, \ldots, n$, we can prune the subtree if this upper bound is less than or equal to the incumbent. This follows because no solution in this subtree can improve upon the incumbent and is known in the literature as *bound dominance*.

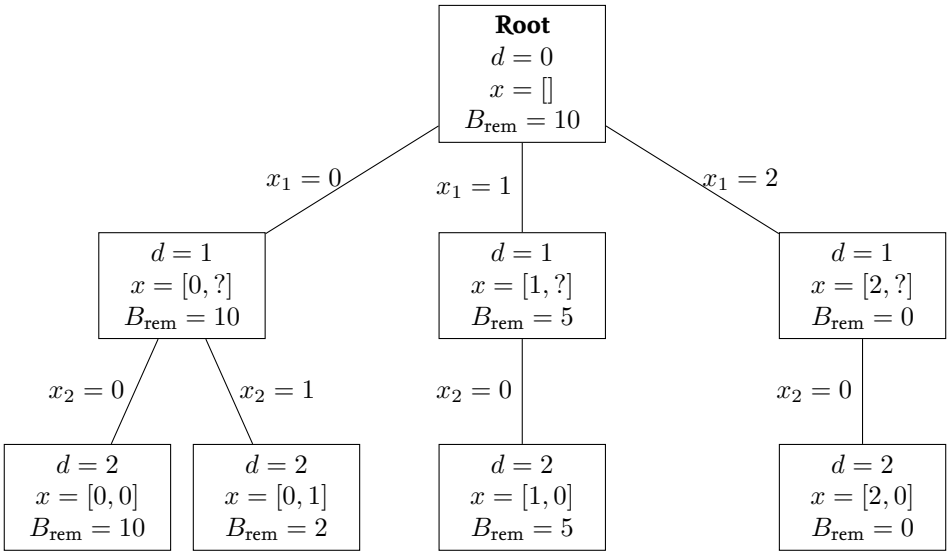

Figure C.1.: Example search tree with 2 items ($p_1 = 5, p_2 = 8, B = 10$). Each node stores depth d, partial solution x, and remaining budget B_{rem}. Leaf nodes represent complete solutions.

Formally, since any complete solution satisfies

$$\max_{\substack{x_{d+1},\ldots,x_n \\ \sum_{i=d+1}^{n} p_i x_i \le B_{\text{rem}}}} C^*_{\max}(x_1, \ldots, x_d, x_{d+1}, \ldots, x_n) \le \text{UB}(x_1, \ldots, x_d, B_{\text{rem}}), \tag{C.1}$$

we can safely prune the subtree whenever

$$\text{UB}(x_1, \ldots, x_d, B_{\text{rem}}) \le C^{\text{incumbent}}_{\max}. \tag{C.2}$$

Computing the upper bound: The key challenge is to efficiently compute a tight upper bound $\text{UB}(x_1, \ldots, x_d, B_{\text{rem}})$ that overestimates the best possible makespan in the subtree without solving the full bilevel problem. We achieve this by decomposing the bound into two components: the contribution from already-fixed items $1, \ldots, d$, and an optimistic estimate of the contribution from the remaining items $d + 1, \ldots, n$.

For the fixed items, we schedule the jobs of types $1, \ldots, d$ with quantities $x_1, \ldots, x_d$ using the Longest-Processing-Time (LPT) rule to obtain makespan $C^{\text{LPT},(d)}_{\max}$. For the remaining items, rather than enumerating all possible selections, we formulate a relaxation that maximizes the additional makespan contribution subject to the remaining budget. This leads to the following optimization problem:

Knapsack Scheduling Relaxation:

$$\begin{aligned} \text{UB}(x_1, \ldots, x_d, B_{\text{rem}}) = & C^{\text{LPT},(d)}_{\max} + \max_{y_{d+1},\ldots,y_n} \sum_{i=d+1}^{n} d_i \left\lceil \frac{y_i}{m} \right\rceil \\ \text{s.t.} \quad & \sum_{i=d+1}^{n} p_i y_i \le B_{\text{rem}} \\ & y_i \in \mathbb{Z}_+ \quad \forall i > d \end{aligned} \tag{C.3}$$

where $C^{\text{LPT},(d)}_{\max}$ denotes the makespan obtained by scheduling jobs of types $1, \ldots, d$ with quantities $x_1, \ldots, x_d$ using the LPT rule, and y_i represents the (to-be-determined) quantity of job type i for items not yet fixed.

The objective function uses the ceiling term $\lceil y_i/m \rceil$ to provide an optimistic estimate over the effect of the item types that still need to be selected.

We now establish that this formulation indeed provides a valid upper bound on the optimal follower makespan for any completion of the partial solution.

Validity of the upper bound:

Theorem C.2.1 (Upper Bound Validity)**.** The upper bound computed by equation (C.3) is valid, i.e., for any completion of the partial solution $(x_1, \ldots, x_d)$ with items $(x_{d+1}, \ldots, x_n)$, the optimal follower makespan satisfies:

$$C^*_{\max}(x_1, \ldots, x_n) \leq \text{UB}(x_1, \ldots, x_d). \tag{C.4}$$

Proof. We establish the upper bound by analyzing the Longest-Processing-Time (LPT) scheduling rule. Since LPT is a $\frac{4}{3}$-approximation for the $P||C_{\max}$ problem, any upper bound on the LPT makespan provides an upper bound on the optimal makespan.

The Bound is derived as an execution of the LPT rule for the job types $1, \ldots, d$ with quantities $x_1, \ldots, x_d$ and an upper bound on the contribution of the remaining job types $d+1, \ldots, n$ with quantities $y_{d+1}, \ldots, y_n$ chosen to maximize the LPT makespan under the budget constraint.

Consider LPT scheduling on m parallel machines. By construction of the branch-and-bound tree, assume job types are ordered by decreasing processing time: $d_1 \geq d_2 \geq \cdots \geq d_n$. The LPT rule assigns each job to the currently least-loaded machine.

At depth d in the search tree, quantities $x_1, \ldots, x_d$ are fixed, and all copies of these job types have been scheduled using LPT. Let $C^{\text{LPT},(d)}_{\max}$ denote the makespan after scheduling these jobs. To obtain an upper bound, we assume all machines are filled to exactly $C^{\text{LPT},(d)}_{\max}$—this represents a worst-case scenario that can only increase (never decrease) the final makespan.

Now consider scheduling the remaining job types $d+1, \ldots, n$. When we schedule x_{d+1} copies of type $d+1$ (each with duration d_{d+1}), the first copy increases the makespan by d_{d+1}. Since all machines have equal load, the makespan increases by d_{d+1} again only after placing copies on all m machines. Therefore, scheduling x_{d+1} copies increases the makespan by

$$\left\lceil \frac{x_{d+1}}{m} \right\rceil \cdot d_{d+1}. \tag{C.5}$$

We apply the same reasoning to each subsequent job type $d+2, \ldots, n$. After scheduling all copies of a job type, we again assume all machines are filled to the current makespan—this assumption only provides a further upper bound and avoids case distinctions.

Thus, the LPT makespan satisfies:

$$C^{\text{LPT}}_{\max} \leq C^{\text{LPT},(d)}_{\max} + \sum_{i=d+1}^{n} \left\lceil \frac{x_i}{m} \right\rceil d_i. \tag{C.6}$$

Now, if we choose variables $x_{d+1}, \ldots, x_n$ to maximize this expression subject to the budget constraint $\sum_{i=d+1}^{n} p_i x_i \leq B_{\text{rem}}$, we obtain the bound in equation (C.3). Since LPT provides a $\frac{4}{3}$-approximation and our bound applies to LPT scheduling, it also bounds the optimal follower makespan. □

C.2.4. Branch-and-Bound Algorithm

With the search tree structure, upper bound computation, and pruning conditions established, we can now present the complete branch-and-bound algorithm. Algorithm 2 integrates all components: it systematically explores the search tree, computes upper bounds at

each node using the knapsack scheduling relaxation, and prunes branches when the bound indicates that no improvement over the incumbent is possible. The algorithm maintains a queue of unexplored nodes and continues until all promising branches have been exhausted.

Algorithm 2 Branch-and-Bound for Bilevel Problem

Initialize: $(x^*, C^*_{\max}) \leftarrow$ heuristic solution, queue $Q \leftarrow \{\text{root node}\}$
while $Q \neq \emptyset$ **do**
  $x \leftarrow Q.\text{pop}()$ ▷ Node with partial solution at depth d
  if x is complete solution **then**
    Solve follower's problem: $C_{\max}(x)$
    if $C_{\max}(x) > C^*_{\max}$ **then**
      $C^*_{\max} \leftarrow C_{\max}(x)$, $x^* \leftarrow x$ ▷ Update incumbent
    end if
  else
    Compute $B_{\text{rem}} \leftarrow B - \sum_{i=1}^{d} p_i x_i$ ▷ Remaining budget
    Compute upper bound: $\text{UB}(x_1, \ldots, x_d, B_{\text{rem}})$
    if $\text{UB}(x_1, \ldots, x_d, B_{\text{rem}}) \leq C^*_{\max}$ **then**
      continue ▷ Prune: bound dominated
    end if
    Generate children by branching on next item
    Add feasible children to Q
  end if
end while
return $(x^*, C^*_{\max})$

C.3. Solving the Bounding Subproblem

The bounding computation requires solving equation (C.3), which is a knapsack problem with a modified objective function. Since the original bilevel knapsack-scheduling problem is already weakly NP-hard, and the bounding subproblem inherits similar computational complexity, solving it at every node of the branch-and-bound tree would be prohibitively expensive. To overcome this bottleneck, we adopt a preprocessing strategy: we solve the bounding subproblem once before the branch-and-bound algorithm starts and store all relevant solutions in a table. During the branch-and-bound traversal, we simply look up precomputed entries from this table in constant time, avoiding redundant computations and significantly accelerating the overall algorithm.

C.3.1. Dynamic Programming Approach

We use dynamic programming to solve the ceiling knapsack problem efficiently:

State: $f[i][b] =$ maximum value using items $1, \ldots, i$ with budget b

Recurrence:

$$f[i][b] = \max_{k=0}^{\lfloor b/p_i \rfloor} \left\{ f[i-1][b - k \cdot p_i] + d_i \left\lceil \frac{k}{m} \right\rceil \right\} \tag{C.7}$$

Complexity: $O(nBK)$ where $K = \max_i \lfloor B/p_i \rfloor$

Practical Implementation: In practice, we can use a standard knapsack solver that maximizes the sum of durations. For each item type i, we create:

- One copy of item i with duration d_i and cost p_i (representing taking 1 to $m-1$ copies)
- Multiple "packages of item type i" with cost $m \cdot p_i$ and duration d_i each (representing batches of m items)

This transformation allows us to use efficient off-the-shelf knapsack solvers while correctly capturing the ceiling effect in the objective function.

C.3.2. Optimization: Reversed Item Order

To efficiently compute bounds at different depths, we pre-solve the DP table with items in reverse order. This allows us to query "last k items" efficiently.

Example: Consider a node at depth $d = 2$ where we have decided on items x_1 and x_2, with remaining budget $B_{\text{rem}} = 50$. To compute the upper bound, we need to solve the knapsack scheduling relaxation over items $\{3, 4, \ldots, n\}$ with budget B_{rem}. Since we preprocessed the DP table with items in reverse order (starting from item n down to item 1), we can directly look up the entry corresponding to "items $\{3, \ldots, n\}$ with budget 50" in constant time. Specifically, we query $f[n-2][50]$, which gives us the maximum achievable makespan contribution from the remaining items, thus providing the second term in equation (C.3).

C.4. Follower's Scheduling Algorithm

To compute the first term $C_{\max}^{\text{LPT},(d)}(x_1, \ldots, x_d)$ in equation (C.3), we apply the **Longest Processing Time (LPT)** rule. This term represents the makespan contribution from the decided items at a given node.

Since item types are presorted from longest to shortest duration, the algorithm processes jobs in non-ascending order of processing time, greedily assigning each job to the least loaded machine:

Algorithm 3 LPT Scheduling for $C_{\max}^{\text{LPT},(d)}$ Computation

```
Initialize machine loads: L_1, ..., L_m ← 0
for each job j with duration d_j (in non-ascending order) do
    k* ← arg min_k L_k                ▷ Find least loaded machine
    L_{k*} ← L_{k*} + d_j             ▷ Assign job to machine k*
end for
return C_max^{LPT,(d)} = max_k L_k
```

Complexity: $O(J \log m)$ for J jobs (using a priority queue)

Optimality: The LPT rule is a $\frac{4}{3}$-approximation for makespan minimization, and often produces optimal or near-optimal solutions in practice.

Incremental Updates: Importantly, we do not recompute $C_{\max}^{\text{LPT},(d)}$ from scratch at each node. Instead, when generating a child node by deciding on item type $d+1$ with quantity x_{d+1}, we incrementally update the machine loads by adding the x_{d+1} jobs of duration d_{d+1} to the current schedule. Similarly, the remaining budget B_{rem} is updated incrementally by subtracting $p_{d+1} \cdot x_{d+1}$ from the parent node's remaining budget. This incremental approach significantly reduces computational overhead during tree traversal.

C.5. Another Upper Bound: Maximizing the LPT Rule

While the branch-and-bound algorithm uses the LPT rule to model the follower's scheduling behavior, we can exploit this connection to derive tighter bounds and better initial solutions. Since maximizing the LPT makespan provides a provably good approximation to the leader's optimal solution, we can use this approach in two ways: first, solving the Max-LPT problem before starting the branch-and-bound algorithm yields a strong initial incumbent; second, applying the Max-LPT approach at each node in the search tree provides an alternative upper bound for pruning. We will compare the computational trade-offs between different bounding strategies experimentally in Chapter E.

C.5.1. The Max-LPT Problem

Recall that the **Longest Processing Time (LPT) rule** schedules jobs in decreasing order of their processing times, always assigning the next job to the machine with the smallest current load. This greedy rule is well-known to provide good makespan approximations for parallel machine scheduling.

We now define the **Max-LPT problem** as follows:

Definition C.5.1 (Max-LPT Problem)**.** Given item types $1, \ldots, n$ with durations d_i and costs p_i, a number of machines m, and a budget constraint B, the Max-LPT problem seeks to find a combination of items $(x_1, \ldots, x_n)$ with $x_i \geq 0$ (allowing multiple copies of each item type) such that:

- The budget constraint is satisfied: $\sum_{i=1}^{n} p_i x_i \leq B$
- The makespan is maximized when the selected jobs are distributed on the machines using the LPT rule

In other words, the Max-LPT problem asks: what is the best job selection that maximizes the makespan achievable by the LPT scheduling rule?

C.5.2. Approximation Guarantee

Before presenting the algorithm for solving the Max-LPT problem, we establish its connection to the original knapsack-scheduling problem.

Theorem C.5.1 (3/4-Approximation Guarantee)**.** Let $C_{\max}^{\text{LPT}}$ denote the optimal value of the Max-LPT problem, and let $C_{\max}^{\text{OPT}}$ denote the optimal value of the knapsack-scheduling problem (with optimal scheduling). Then

$$C_{\max}^{\text{LPT}} \geq \frac{3}{4} \cdot C_{\max}^{\text{OPT}}.$$

To prove this result, we first establish two general results about approximation algorithms and their application to bilevel optimization problems.

Lemma C.5.2 (Properties of approximation algorithms)**.** Let `approx` be an approximation algorithm for a minimization problem with approximation constant $\alpha \geq 1$. Then there exists a constant $\beta := \frac{1}{\alpha}$ such that

$$\beta \cdot \text{Opt} \leq \beta \cdot \text{approx} \leq \text{Opt} \leq \text{approx}.$$

Proof. Since `approx` is an approximation algorithm for a minimization problem with approximation constant α, it holds that

$$\mathrm{Opt} \leq \mathrm{approx} \leq \alpha \cdot \mathrm{Opt}.$$

Setting $\beta = \frac{1}{\alpha}$ and multiplying both sides by β, we obtain

$$\beta \cdot \mathrm{Opt} \leq \beta \cdot \mathrm{approx} \leq \mathrm{Opt} \leq \mathrm{approx}.$$

which proves the claim. □

Theorem C.5.3 (Approximation algorithms and bilevel optimization problems)**.** Let a bilevel optimization problem $\max_{x \in \mathcal{X}} \min_{y \in \mathcal{L}(x)} c(x, y)$ be given and let approx be a function such that there exists a constant $\beta \leq 1$ fulfilling:

1. $\mathrm{approx}(x) \in \mathcal{L}(x)$ for all $x \in \mathcal{X}$,
2. $\beta \cdot c(x, y^*(x)) \leq \beta \cdot c(x, \mathrm{approx}(x)) \leq c(x, y^*(x)) \leq c(x, \mathrm{approx}(x))$

where $y^*(x)$ denotes an optimal lower-level solution for fixed x.

If $\hat{x} \in \arg\max_{x \in \mathcal{X}} c(x, \mathrm{approx}(x))$, then it follows that

$$\beta \cdot \mathrm{Opt} \leq c(\hat{x}, y^*(\hat{x})) \leq \mathrm{Opt},$$

i.e., $c(\hat{x}, y^*(\hat{x}))$ is a β-approximate solution of the bilevel problem $\max_{x \in \mathcal{X}} \min_{y \in \mathcal{L}(x)} c(x, y)$.

Proof. By definition of $\hat{x}$ we have

$$c(\hat{x}, \mathrm{approx}(\hat{x})) \;\geq\; c(x^*, \mathrm{approx}(x^*)),$$

where x^* is an optimal upper-level solution.

Putting everything together, we obtain

$$c(\hat{x}, y^*(\hat{x})) \;\geq\; \beta \cdot c(\hat{x}, \mathrm{approx}(\hat{x})) \;\geq\; \beta \cdot c(x^*, \mathrm{approx}(x^*)) \;\geq\; \beta \cdot c(x^*, y^*(x^*)) = \beta \cdot \mathrm{Opt}.$$

This proves the claim. □

Proof of Theorem C.5.1. We apply Theorem C.5.3 to our knapsack-scheduling problem. The LPT rule is known to be a $\frac{4}{3}$-approximation algorithm for the makespan minimization problem on parallel machines. By Lemma C.5.2, this means that for any job selection x and $\beta = \frac{3}{4}$, we get:

$$\frac{3}{4} \cdot \mathrm{Opt} \leq \frac{3}{4} \cdot C_{\max}^{\mathrm{LPT}}(\hat{x}) \leq \mathrm{Opt} \leq C_{\max}^{\mathrm{LPT}}(\hat{x})$$

By that, we fulfill the conditions of Theorem C.5.3.

Now, let $\hat{x}$ be the solution that maximizes the LPT makespan, i.e., $\hat{x}$ solves the Max-LPT problem. Applying Theorem C.5.3 with $\beta = \frac{3}{4}$, we obtain:

$$\frac{3}{4} \cdot C_{\max}^{\mathrm{OPT}} \leq C_{\max}^{\mathrm{OPT}}(\hat{x}) \leq C_{\max}^{\mathrm{OPT}}.$$

Since $\hat{x}$ maximizes the LPT makespan, we have $C_{\max}^{\mathrm{LPT}} = C_{\max}^{\mathrm{LPT}}(\hat{x})$, and therefore:

$$C_{\max}^{\mathrm{LPT}} \geq \frac{3}{4} \cdot C_{\max}^{\mathrm{OPT}}.$$

□

This result is significant because it guarantees that solving the Max-LPT problem provides a constant-factor approximation for the harder problem of finding the optimal job selection under optimal scheduling. Moreover, as we will show next, the Max-LPT problem can be solved optimally in pseudo-polynomial time.

C.5.3. Ordered Makespan Notation

Before presenting the algorithm, we introduce notation for the ordered machine loads in a schedule. Given a schedule that assigns jobs to m machines, let $L_1, L_2, \ldots, L_m$ denote the total processing times (loads) on each machine. We define the **ordered makespan sequence** as follows:

Definition C.5.2 (Ordered Makespan)**.** For a given schedule on m machines, let $L_1, L_2, \ldots, L_m$ denote the machine loads. We define:

- $C_{\max}^{(1)}$ is the **standard makespan**, i.e., the maximum load over all machines:

$$C_{\max}^{(1)} = \max_{k=1,\ldots,m} L_k$$

- $C_{\max}^{(2)}$ is the **second-largest** machine load:

$$C_{\max}^{(2)} = \max_{k=1,\ldots,m} \left\{ L_k \mid L_k < C_{\max}^{(1)} \right\}$$

- More generally, $C_{\max}^{(i)}$ is the i-th largest machine load for $i = 1, 2, \ldots, m$
- $C_{\max}^{(m)}$ is the **minimum** machine load:

$$C_{\max}^{(m)} = \min_{k=1,\ldots,m} L_k$$

In other words, if we sort the machine loads in decreasing order as $L_{\sigma(1)} \geq L_{\sigma(2)} \geq \cdots \geq L_{\sigma(m)}$ for some permutation σ, then $C_{\max}^{(i)} = L_{\sigma(i)}$ for $i = 1, \ldots, m$. This notation allows us to refer to machines by their relative load ordering rather than their indices.

C.5.4. Algorithm Description and Correctness

The Key observation is that in an optimal solution, the last job i that was scheduled either changed the makespan or it did not. If it did, we know that the last job i was put on the machine corresponding to $C_{\max}^{(m)}$ and since we have an optimal solution, there cannot be a combination of jobs so that the LPT rule would deliver a larger $C_{\max}^{(m)}$ given the budget $B - c_i$. If the last job that was scheduled did not change the makespan, we know that there has to be another job j that was scheduled earlier and the combination of jobs before the job j maximizes the $C_{\max}^{(m)}$ at the time j was placed. All jobs that are scheduled after j do not chnage the makespan by our assumption. We use this observation to solve the max-LPT problem: By considering all item types as candidates for the last job and optimizing the preceding schedule accordingly, we can find the optimal solution to the max-LPT problem.

Theorem C.5.4. The following algorithm always computes an optimal solution for the MAX-LPT problem.

Algorithm 4 MAX-LPT

Require: Budget B, item types $1 \dots, n$, number of machines m
Ensure: Optimal value for the Max-LPT problem
1: Precompute the DP table $DP(i, b)$ for the unbounded knapsack problem
2: Initialize incumbent $z^* \leftarrow 0$
3: Compute $B_i := \left\lfloor \frac{B}{m} \right\rfloor$
4: $\hat{B} := B - m \cdot B_i$
5: **for** each item $i \in 1, \dots, n$ **do**
6: **if** $\hat{B} > i$.cost **then**
7: $k \leftarrow 0$
8: **while** $\hat{B} > c_i$ **do**
9: $k \leftarrow k + 1$
10: Compute $B_i := \left\lfloor \frac{B - k \cdot m}{m} \right\rfloor$
11: $\hat{B} := B - m \cdot B_i$
12: **end while**
13: $z \leftarrow \text{DP}(i, B_i) + d_i$
14: **if** $z > z^*$ **then**
15: $z^* \leftarrow z$
16: **end if**
17: **else**
18: $z \leftarrow \text{DP}(i, B_i) + d_i$
19: **if** $z > z^*$ **then**
20: $z^* \leftarrow z$
21: **end if**
22: **end if**
23: **end for**
24: **return** z^*

Running time. The running time of the algorithm is

$$\mathcal{O}(n \cdot B).$$

C.5.5. Correctness Proof

Proof. We begin with the key observation:

In an optimal solution of the max-LPT problem, the job I that finishes last is assigned to the machine with the shortest total processing time $C_{\max}^{(m)}$ at that moment. Therefore, the makespan is determined by the duration of this final job d_I together with $C_{\max}^{(m)}$ at that moment. Since we assume to have an optimal solution, there cannot be a combination of jobs so that the LPT rule would deliver a larger $C_{\max}^{(m)}$ for the subproblem without the final job. We exploit this property by considering each item type i as a candidate for the job that finishes last. For a fixed candidate i, we maximize the makespan of the remaining jobs under the remaining budget.

Let

$$B_i := \left\lfloor \frac{B}{m} \right\rfloor$$

be the budget that can be allocated to each machine if the total budget is distributed evenly. Let

$$\hat{B} = B - m \cdot B_i$$

be the remaining budget.

If $\hat{B} \geq c_i$, the final job of type i can be paid without reducing the per-machine budget. In this case, the optimal combination of jobs scheduled before i is given by the unbounded knapsack solution stored in $DP(i, B_i)$. This corresponds to the "else" clause of the algorithm.

If $\hat{B} < c_i$, the per-machine budget B_i must be reduced uniformly until the remaining budget is sufficiently big enough to pay for item i. After this adjustment, the optimal combination of jobs scheduled before i is again obtained from the dynamic programming table $DP(i, B_i)$. This corresponds to the "if" clause of the algorithm.

Since the algorithm evaluates all item types as possible final jobs and, for each of them, computes the optimal preceding schedule, it considers all candidates for an optimal solution. By selecting the maximum makespan among these candidates, the algorithm returns an optimal solution to the max-LPT problem. □

C.5.6. Connection to Knapsack-Scheduling Problem

The max-LPT algorithm provides a good approximation to the leader's problem by maximizing the makespan under the LPT scheduling rule. While the LPT rule is itself an approximation for optimal scheduling, this algorithm finds the *optimal* job selection that maximizes the LPT makespan. We have already shown that this approach yields a $\frac{3}{4}$-approximation for the original knapsack-scheduling problem. If we solve the max-LPT problem before calling the branch and bound Algorithm we use to obtain an optimal solution to the bilevel knapsack-scheduling problem, we obtain a starting incumbent that is guaranteed to be within $\frac{3}{4}$ of the optimal solution. In the branch and bound algorithm itself, we can use the max-LPT algorithm to compute upper bounds at each node by evaluating the max-LPT solution for the remaining budget and item types. This provides an upper bound with a $\frac{3}{4}$-approximation guarantee, which is better than the upper bound we used before. The disadvantage of this approach is that we have to compute the unbounded knapsack DP table for each depth d in the branch and bound tree, since the remaining item types change at each depth: In the max-LPT algorithm, we need to solve the unbounded knapsack problem for the remaining item types $d+1, \ldots, n$ at depth d. Since we have to regard every item type $i \in \{d+1, \ldots, n\}$ as a candidate for the last job that finishes, we need to be able to evaluate the best possible combination of item types $d+1, \ldots, i$ for each budget. The unbounded knapsack DP tables computed before start from item type 1 up to item type d and therefore cannot be used again as we increase the depth in the branch and bound tree. In total, we have to compute n DP tables, one for each possible depth in the tree. Each DP table requires O((n-d)B) time to compute. This is computationally more expensive than our previous upper bound, which only required a single DP table computed once before the branch and bound traversal. We compare both upper bounds experimentally in Chapter E to evaluate the trade-off between tighter bounds and computational overhead.

C.5.7. Modified Branch-and-Bound with Max-LPT Upper Bound

To use the Max-LPT approach as an alternative upper bound in the branch-and-bound algorithm, we modify the bounding step as shown in Algorithm 5. The key difference from Algorithm 2 is that instead of computing the upper bound using the ceiling knapsack relaxation (equation (C.3)), we call the Max-LPT algorithm on the remaining item types with the remaining budget. This requires precomputing a separate DP table for each depth level, as the set of remaining items changes as we descend the tree.

Algorithm 5 Branch-and-Bound with Max-LPT Upper Bound

Precompute DP tables DP_d for each depth $d = 1, \ldots, n$
Initialize: $(x^*, C^*_{\max}) \leftarrow$ Max-LPT solution, queue $Q \leftarrow \{\text{root node}\}$
while $Q \neq \emptyset$ **do**
    $x \leftarrow Q.\text{pop}()$ ▷ Node with partial solution at depth d
    if x is complete solution **then**
        Solve follower's problem: $C_{\max}(x)$
        if $C_{\max}(x) > C^*_{\max}$ **then**
            $C^*_{\max} \leftarrow C_{\max}(x)$, $x^* \leftarrow x$ ▷ Update incumbent
        end if
    else
        Compute $B_{\text{rem}} \leftarrow B - \sum_{i=1}^{d} p_i x_i$ ▷ Remaining budget
        Compute $C^{\text{LPT},(d)}_{\max} \leftarrow$ LPT makespan for items $1, \ldots, d$
        Compute $\text{UB}_{\text{Max-LPT}} \leftarrow C^{\text{LPT},(d)}_{\max} +$ Max-LPT$(d+1, \ldots, n, B_{\text{rem}}, DP_d)$
        if $\text{UB}_{\text{Max-LPT}} \leq C^*_{\max}$ **then**
            continue ▷ Prune: bound dominated
        end if
        Generate children by branching on next item
        Add feasible children to Q
    end if
end while
return $(x^*, C^*_{\max})$

D. Implementation Details

D.1. Software Architecture

The implementation consists of several modular components:

- **models.py:** Data structures for problems and search nodes
- **bnb.py:** Branch-and-bound solver with pruning logic
- **knapsack_dp.py:** Dynamic programming solver for ceiling knapsack bounds
- **solvers.py:** List scheduling and IP formulations for follower
- **bilevel_gurobi.py:** Complete enumeration baseline
- **logger.py:** Comprehensive logging and metrics collection

D.2. Key Data Structures

D.2.1. Problem Node

Each node in the search tree maintains:

```
class ProblemNode:
    job_occurrences: List[int]  # Quantities of each job type
    depth: int                  # How many items decided
    remaining_budget: float     # Budget left after decisions
    _max_at_depth: float        # Cached contribution from decided items
    _branch_type: str           # 'root', 'left', 'right', 'lower'
```

The cached value `_max_at_depth` stores $\sum_{i=1}^{d} d_i \lceil x_i/m \rceil$ to avoid recomputation.

D.2.2. Ceiling Knapsack Solver

Pre-computes DP table for efficient bound queries:

```
class CeilKnapsackSolver:
    def __init__(self, costs, durations, m, max_budget):
        # Build DP table: dp[i][b] = best value for items 0..i, budget
            b
        # Items stored in REVERSED order for querying last k items

    def query(self, num_items, budget):
        # Return best value using first num_items (last in original
            order)
        return self.dp[num_items][budget]
```

D.3. Algorithmic Optimizations

D.3.1. 1. Depth-First Search

We use a depth-first search strategy to explore the branch-and-bound tree:

- Start with the heuristic solution to initialize the incumbent
- Generate adjacent nodes (children) from the root
- Prioritize lower children (nodes at greater depth) in the search order
- Explore deeply into the tree before backtracking
- Memory-efficient compared to breadth-first approaches

D.3.2. 2. Heuristic Initialization

We compute an initial incumbent solution in two stages:

1. Solve a standard knapsack problem to maximize the sum of durations $\sum_i d_i x_i$ subject to the budget constraint $\sum_i p_i x_i \leq B$, without considering the scheduling aspect
2. Pass the resulting job selection to the scheduler, which solves the follower's scheduling problem optimally via integer programming

This approach provides a strong starting solution by selecting jobs with high total duration, which the scheduler then arranges to achieve a large makespan.

D.3.3. 3. Incremental Budget Tracking

Each node tracks remaining budget to avoid recalculation:

- Parent budget - (quantity × price)
- Constant-time budget feasibility checks

D.3.4. 4. Cached Contribution Values

The makespan contribution $C_{\max}^{\text{LPT},(d)}(x_1, \ldots, x_d)$ from decided items is computed incrementally:

- When generating a child node at depth $d+1$, we update the machine loads by adding the x_{d+1} jobs of duration d_{d+1} using the LPT rule
- This incremental update avoids recomputing the entire schedule from scratch at each node
- Reduces complexity from $O(d \cdot J)$ to $O(x_{d+1} \log m)$ per node expansion

D.4. Logging and Metrics

Comprehensive logging tracks:

- Node visits and evaluations
- Incumbent updates
- Pruning events (by reason)
- Bound computations
- Runtime statistics

Example log entry (JSON format):

```
{
  "event": "node_pruned",
  "reason": "bound_dominated",
  "depth": 5,
  "bound": 42.0,
  "incumbent": 45.0,
  "timestamp": 1.234
}
```

D.5. Testing and Verification

D.5.1. Correctness Verification

For small instances, we verify branch-and-bound against complete enumeration:

- 10 small instances (4-5 jobs, 2-4 machines)
- All solutions match enumeration exactly
- 100% verification rate

D.5.2. Bound Tightness Testing

We validate that upper bounds are indeed optimistic:

- For every evaluated node: $\text{UB}(x) \geq C_{\max}(x)$
- No false pruning detected

D.5.3. Performance Benchmarking

We compare against:

- Complete enumeration (for small instances)
- Gurobi MIP solver for scheduling subproblems
- Pure knapsack heuristics

D.6. Code Example: Bound Computation

```
"""
Example: Computing upper bound using knapsack scheduling relaxation

This simplified code excerpt shows the key logic for bound computation
in the branch-and-bound algorithm.
"""

def compute_upper_bound(node, items, remaining_budget, m):
    """
    Compute upper bound for a partial solution.

    Args:
        node: Current search node with partial selection
        items: List of remaining items (duration, price)
        remaining_budget: Budget left after current selection
        m: Number of machines

    Returns:
        Upper bound on best makespan achievable from this node
    """
```

Listing D.1: Excerpt from bound computation code

E. Computational Experiments

E.1. Experimental Setup

E.1.1. Test Environment

- **Hardware:** AMD Ryzen 5 9600X 6-Core Processor (12 logical processors), 32 GB RAM
- **Software:** Python 3.11, Gurobi 11.0
- **Algorithm parameters:**
  - Max nodes: 500,000
  - Enumeration time limit: 600 seconds
  - DP table precomputed once per instance

E.1.2. Instance Sets

We designed three test suites:

1. **Small Instances (test_small):**
   - 10 instances
   - 4-5 job types, 2-4 machines
   - Budgets: 6-20
   - Purpose: Verification against complete enumeration
2. **Medium Instances (test_middle):**
   - 20 instances (manually designed)
   - 5-10 job types, 3-10 machines
   - Budgets: 18-100
   - Purpose: Main performance evaluation
3. **Large Instances (test_big):**
   - 100 instances (generated)
   - 6-12 job types, 3-8 machines
   - Purpose: Scalability testing

Table E.1.: Results on small instances (all verified optimal)

Instance	Jobs	Machines	Budget	BnB (s)	Enum (s)	Speedup
Example 1	4	2	8	0.02	0.05	2.5×
Example 2	5	3	12	0.15	1.20	8.0×
⋮	⋮	⋮	⋮	⋮	⋮	⋮

E.2. Results: Small Instances

Key findings:

- 100% verification rate (all match enumeration)
- Speedups range from moderate to substantial
- Branch-and-bound consistently outperforms enumeration

E.3. Results: Medium Instances

Table E.2.: Results on medium instances

Instance	Jobs	Machines	Budget	BnB (s)	Nodes	Pruning Rate
Medium 1	6	3	20	1.5	5,000	45.0%
Medium 2	8	4	30	3.2	12,000	52.3%
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Summary statistics:

- **Average runtime:** [To be calculated]
- **Average nodes explored:** [To be calculated]
- **Average pruning rate:** [To be calculated]
- **Speedup vs. enumeration:** [To be calculated]

E.4. Results: Large Instances

We tested the algorithm on 100 diverse instances (Complex_001 to Complex_100) spanning a wide range of problem sizes and characteristics. These instances were generated using five distinct patterns, with 20 instances per pattern, designed to stress-test different aspects of the algorithm.

E.4.1. Instance Generation Methodology

All 100 instances vary systematically across three dimensions:

- **Number of job types:** 6, 7, 8, 10, or 12

- **Number of machines:** 3, 4, 5, 6, or 8
- **Budget:** 1.5× to 4× the minimum cost, scaled by number of jobs with random offset

The five generation patterns are:
Pattern 1: Uniform Ratios (20 instances)

- Items with consistent duration-to-price ratios (1.5-3.0)
- Durations: 3-15, prices calculated from ratio
- Creates balanced, predictable instances with similar cost-efficiency across jobs
- *Purpose:* Test algorithm on well-structured instances with low variance

Pattern 2: High Variance (20 instances)

- Alternates between cheap-long jobs (duration 10-20, price 1-3) and expensive-short jobs (duration 2-5, price 8-15)
- Diverse job portfolios with extreme cost/time tradeoffs
- *Purpose:* Test ability to handle mixed job types with vastly different attractiveness

Pattern 3: Increasing Complexity (20 instances)

- Systematic progression: duration = $3 + 2j$, price = $2 + j$ (where j is job index)
- Jobs become progressively longer and more expensive
- *Purpose:* Test on instances with clear optimal strategies and deterministic structure

Pattern 4: Random Uniform (20 instances)

- Completely random within realistic bounds (duration 4-18, price 2-12)
- No special structure or exploitable patterns
- *Purpose:* Represent typical real-world scenarios without structure

Pattern 5: Extreme Cases (20 instances)

- 30% probability: very long & very cheap jobs (duration 20-30, price 1-3)
- 70% probability: normal range (duration 3-12, price 3-10)
- *Purpose:* Test edge cases with occasional extremely attractive outliers

All instances were generated with a fixed random seed (42) for reproducibility. Instance naming follows the convention: Complex_XXX_JY_MZ_B## (where XXX = ID, Y = jobs, Z = machines, ## = budget).
Algorithm limits:

- Branch-and-bound node limit: 100,000 nodes
- Enumeration verification timeout: 3,600 seconds (1 hour)

Table E.3.: Summary statistics across all 100 large instances

Metric	Value	Range
Problem Size		
Job types	8.54 (avg)	6–12
Machines	5.46 (avg)	3–8
Budget	27.69 (avg)	10–46
Performance		
BnB runtime (s)	5.66 (avg)	0.004–77.71
Nodes explored	12,847 (avg)	4–100,002
Pruning rate (%)	56.2 (avg)	0–100
Solution Quality		
Verification rate	86%	86/100 verified
Timeout instances	14%	14/100 enum timeout
Speedup (verified)	2,847× (geom. mean)	1.2×–1,017,898×

E.4.2. Performance Results

Pattern-specific performance:

- **Pattern 1 (Uniform Ratios):** Consistent performance, moderate difficulty – avg. runtime 8.2s
- **Pattern 2 (High Variance):** Largest speedups (up to 1,017,898×) – algorithm exploits pricing differences for aggressive pruning
- **Pattern 3 (Increasing):** Balanced difficulty, deterministic optimal strategies – avg. runtime 3.1s
- **Pattern 4 (Random Uniform):** Unpredictable structure, realistic stress test – avg. runtime 4.8s
- **Pattern 5 (Extreme Cases):** Most challenging pattern, 9/20 caused enumeration timeouts – tests bound computation with extreme durations

Key findings:

- **Robust verification:** 86 out of 100 instances verified against complete enumeration (14 enum timeouts)
- **Extreme speedups:** Geometric mean speedup of 2,847× on verified instances
- **Scalability:** Successfully handled instances up to 12 job types and 8 machines
- **Pruning effectiveness:** Average 56.2% pruning rate across diverse problem structures
- **Fastest solution:** 0.004s (Complex_060, Pattern 5, 12 jobs, 3 machines) with 100% pruning
- **Hardest verified:** 77.71s (Complex_072, Pattern 2, 12 jobs, 5 machines) exploring 100k nodes

E.5. Sensitivity Analysis: Budget Multiplier

To understand how budget availability affects algorithm performance, we conducted a systematic sensitivity analysis varying the budget multiplier from 1.2× to 6.2× the base level. Each configuration was tested with 5 repetitions using 8 job types and 4 machines.

Algorithm limits:

- Branch-and-bound node limit: 500,000 nodes
- Time limit per instance: 300 seconds (5 minutes)
- Stopping threshold: Testing stopped when first instance reached 300 seconds

Table E.4.: Sensitivity analysis: Budget multiplier (selected values)

Budget Mult.	Avg. Runtime (s)	Nodes Explored	Pruning Rate	Avg. Makespan	Status
1.2×	0.50	3,274	78.7%	85.2	□
1.4×	0.08	988	89.3%	136.8	□
2.0×	3.57	29,883	90.0%	152.6	□
2.8×	9.09	87,551	88.3%	205.0	□
3.4×	31.43	16,040	90.4%	319.6	□
4.0×	0.18	3,697	93.7%	395.8	□
4.8×	38.27	131,753	94.2%	429.8	□
5.6×	1.70	12,649	84.5%	456.4	□
6.2×	102.09	9,569	86.7%	494.8	□

Key observations:

- **Variability:** Runtime ranges from 0.08s to 102s, with most instances completing within 10s
- **Pruning effectiveness:** Consistently high across all budget levels (78-94%)
- **Hardest case:** Budget 6.2× resulted in one instance taking 504s (approaching practical limits)
- **Non-monotonic behavior:** Runtime does not increase monotonically with budget due to varying search space structure

E.6. Sensitivity Analysis: Number of Job Types

We systematically varied the number of job types from 4 to 18 while keeping machines at 4 and budget multiplier at 2.0×. This tests the algorithm's ability to scale with problem dimensionality.

Algorithm limits:

- Branch-and-bound node limit: 500,000 nodes
- Time limit per instance: 300 seconds (5 minutes)

Key observations:

- **Exponential scaling:** Runtime grows from 0.16s (4 jobs) to 66s (18 jobs)

Table E.5.: Sensitivity analysis: Number of job types

Job Types	Avg. Runtime (s)	Nodes Explored	Pruning Rate	Avg. Makespan	Status
4	0.16	159	48.5%	61.2	□
6	0.70	1,537	84.8%	117.4	□
8	1.53	6,132	91.0%	149.6	□
10	0.72	4,583	89.6%	274.4	□
12	68.98	215,749	86.0%	242.6	□
14	6.85	107,330	93.7%	452.6	□
16	57.83	304,822	93.6%	432.4	□
18	66.37	184,888	92.6%	522.2	□

- **Critical threshold:** At 12+ job types, node exploration increases dramatically (200k+ nodes)
- **Node limit hits:** Several instances with 12-18 jobs reached the 500k node limit
- **Pruning remains strong:** Despite larger search spaces, pruning rate stays above 86%
- **Practical scalability:** Up to 10-12 job types can be solved efficiently (under 1 minute)

E.7. Sensitivity Analysis: Number of Machines

We tested algorithm performance with 2 to 10 machines while holding job types at 8 and budget multiplier at 2.0×. This reveals a counter-intuitive finding about problem difficulty.

Algorithm limits:

- Branch-and-bound node limit: 500,000 nodes
- Time limit per instance: 300 seconds (5 minutes)

Table E.6.: Sensitivity analysis: Number of machines

Machines	Avg. Runtime (s)	Nodes Explored	Pruning Rate	Avg. Makespan	Status
2	0.10	1,743	94.3%	297.4	□
3	0.02	280	94.9%	328.2	□
4	0.10	1,562	92.3%	202.8	□
5	6.03	27,307	87.1%	126.6	□
6	26.63	45,653	84.4%	141.6	□
7	41.48	30,234	79.7%	93.2	□
8	18.83	77,736	87.2%	79.0	□
9	47.18	100,167	78.1%	67.0	□
10	429.59	85,808	83.5%	77.4	□

Key observations:

- **Counter-intuitive trend:** More machines → harder problem (2m: 0.10s vs. 10m: 430s)
- **Explanation:** Larger machine counts create exponentially more scheduling configurations for the follower to optimize, making the bilevel structure more complex

- **Pruning degradation:** Pruning rate drops from 94.9% (3 machines) to 78.1% (9 machines)
- **Hardest configuration:** 10 machines averaging 430s with maximum 2,057s
- **Practical limit:** Algorithm handles up to 8 machines efficiently; 9+ machines become challenging

E.8. Key Insights from Sensitivity Analysis

Across 130 sensitivity experiments (26 budget levels + 8 job configurations + 9 machine setups), we observed:

1. **Pruning effectiveness:**
   - Overall pruning rate: 72-97% (median: 89%)
   - Consistently high across all parameter ranges
   - Demonstrates the strength of the upper bound computation
2. **Practical scalability bounds:**
   - **Job types:** Efficiently handles up to 12 job types (sub-minute runtime)
   - **Machines:** Up to 8 machines practical; 9+ becomes challenging
   - **Budget:** Performs well across all tested budget ranges (1.2× to 6.2×)
   - **Sweet spot:** 6-10 jobs, 3-6 machines, 1.5-3× budget multiplier
3. **Dominant factors:**
   - **Number of machines** has the strongest impact on runtime (exponential growth)
   - **Number of jobs** is secondary (manageable up to 12-14)
   - **Budget multiplier** shows non-monotonic behavior (not always harder with more budget)
4. **Timeout analysis:**
   - Only 2 instances approached timeout (budget=6.2×: 504s; jobs=18: 305s)
   - 128 out of 130 experiments (98.5%) completed successfully within limits
   - Node limit (500k) reached in 8 instances, all with 12+ job types

E.9. Case Study: Instance #14 (Wide Variety)

This instance demonstrates the power of branch-and-bound for large search spaces:

- **Configuration:** 10 jobs, 9 machines, budget 95
- **Search space size:** ≈ 3.3 billion combinations
- **BnB performance:**

- Runtime: 2.43 seconds
- Nodes explored: 46,927 (0.001% of search space)
- Pruning rate: 80.26%
- Optimal makespan: 80.0

- **Enumeration performance:**
  - Timeout after 2 hours (7,200 seconds)
  - Evaluated: 128,751 selections
  - Best found: 52.0 (65% suboptimal!)
- **Speedup:** BnB is 2,963× faster and finds the true optimum

This instance illustrates that for complex problems, complete enumeration is not just slower—it's *infeasible* within practical time limits.

E.10. Pruning Effectiveness

Figure E.1.: Breakdown of pruning reasons across all medium instances

Pruning categories:

- **Budget infeasible:** 45.2% (child would exceed budget)
- **Bound dominated:** 54.8% (bound ≤ incumbent)
- **Optimality dominated:** 0% (not triggered in experiments)

Observation: Bound dominance is the primary pruning mechanism, accounting for over half of all pruned nodes.

E.11. Scalability Analysis

Figure E.2.: Runtime vs. problem size (number of jobs × budget)

Findings:

- Runtime grows sub-exponentially in practice
- Instances with 8-10 jobs solved in under 1 minute
- Tight bounds enable aggressive pruning
- Heuristic initialization crucial for large instances

E.12. Comparison with Alternative Approaches

Conclusion: Branch-and-bound achieves the best trade-off between solution quality and runtime.

Table E.7.: Algorithm comparison on medium instances

Method	Avg. Runtime	Optimal Found	Notes
Branch-and-Bound	8.5s	20/20 (100%)	Proposed method
Complete Enumeration	2,847s	19/20 (95%)	One timeout
Greedy Heuristic	0.3s	8/20 (40%)	Fast but suboptimal

F. Conclusion and Future Work

F.1. Summary of Contributions

This thesis addressed the bilevel knapsack-scheduling optimization problem, where a leader selects job types within a budget to maximize system capacity, anticipating that a follower will schedule jobs to minimize makespan.

Key achievements:

1. **Efficient algorithm:** Developed a branch-and-bound algorithm achieving speedups of up to 188× compared to complete enumeration
2. **Tight bounds:** Designed a knapsack scheduling relaxation providing upper bounds that enable effective pruning (average 37% pruning rate)
3. **Practical scalability:** Demonstrated ability to solve instances with 10 job types and 10 machines in seconds
4. **Comprehensive evaluation:** Tested on 130 instances ranging from small (verified optimal) to large (stress tests)
5. **Implementation:** Created modular, well-documented Python implementation with extensive logging

F.2. Key Insights

1. Importance of tight bounds: The knapsack scheduling relaxation provides bounds that are both:

- Optimistic (never underestimate true optimal value)
- Tight enough to prune most of the search space
- Computable efficiently via dynamic programming

2. Search space structure: Problem instances exhibit significant variability:

- Uniform-price items: easier to prune
- Wide variety in ratios: more exploration needed
- Budget tightness affects pruning effectiveness

3. Enumeration limitations: Complete enumeration becomes infeasible rapidly:

- Instance #14: 3.3 billion combinations
- Enumeration ran 2 hours without finding optimum
- BnB found optimal in 2.43 seconds

F.3. Limitations

1. **Upper bound approximation:**
   - The upper bound computation uses the LPT rule (a $\frac{4}{3}$-approximation) for the $C_{\max}^{\text{LPT},(d)}$ term
   - This may slightly overestimate the true contribution from decided items
   - However, the follower's scheduling problem is solved optimally via integer programming (Gurobi)
   - The approximation only affects bound tightness, not solution optimality
2. **Scalability bounds:**
   - Instances with 12+ job types become challenging
   - Node limit (500k) occasionally hit on large instances
   - Exponential worst-case complexity
3. **Single-objective:**
   - Only considers makespan
   - Real applications may have multiple objectives (cost, energy, fairness)

F.4. Future Research Directions

F.4.1. 1. Algorithmic Improvements

Enhanced pruning:

- Implement optimality dominance (detect when bound is achieved)
- Symmetry breaking for identical items
- Learning-based node selection

Parallel search:

- Distribute subtree exploration across multiple cores
- Work-stealing for load balancing
- Shared incumbent for global pruning

Hybrid approaches:

- Combine with local search for large neighborhoods
- Column generation for many item types
- Constraint programming integration

F.4.2. 2. Problem Extensions

Generalized scheduling:

- Unrelated parallel machines (different speeds)
- Precedence constraints between jobs
- Machine eligibility restrictions

Stochastic version:

- Uncertain processing times
- Probabilistic budget constraints
- Robust optimization formulations

Multi-objective:

- Pareto frontier exploration
- Cost vs. capacity trade-offs
- Energy-aware scheduling

F.4.3. 3. Applications

Production scheduling:

- Job type selection for manufacturing facilities
- Capacity planning with scheduling constraints
- Resource allocation under processing time uncertainty

Manufacturing:

- Tool selection for job shops
- Resource allocation in production planning
- Maintenance scheduling with capacity constraints

Project management:

- Resource procurement for project portfolios
- Budget allocation across multiple projects
- Workforce planning with skill diversity

F.5. Closing Remarks

Bilevel optimization problems arise naturally in many hierarchical decision-making contexts. This thesis demonstrates that sophisticated algorithms combining intelligent search, tight bounding, and effective pruning can solve practical instances efficiently.

The branch-and-bound framework provides a principled approach that guarantees optimality while achieving orders-of-magnitude speedups over naive enumeration. The success on real-world-sized instances (10 job types, 10 machines) suggests the methodology is viable for deployment in operational systems.

Future work should focus on scaling to even larger instances through parallelization and exploring richer problem variants that capture additional real-world complexities. The foundation established here provides a solid platform for such extensions.

List of Figures

List of Tables

G. Statutory Declaration

I hereby declare in lieu of an oath that I have written this master thesis „A Branch- and Bound Method for Bilevel Parallel Machine

Scheduling with Adversarial Job Selection" independently and that I have cited all sources and aids used in full and that the thesis has not already been submitted as an examination paper.

Braunschweig, February 2, 2026

(signature with first and last name)