Scheduling with local Precedence Constraints

Heinrich-Heine University Düsseldorf

Taha El Amine Kassabi

November 28, 2024

Abstract

Scheduling plays a critical role in optimizing resource utilization across various industries, including manufacturing, logistics, and computing. This thesis investigates a specific variant of the scheduling problem characterized by **local precedence constraints**, where precedence requirements are enforced only within individual machines based on a global job order—a scenario not previously explored in the literature. The primary objective is to **minimize the weighted sum of completion times**, prioritizing higher-weighted jobs to enhance efficiency and cost-effectiveness.

To address this problem, we develop several algorithms, including an exact **dynamic programming** method and various heuristic and approximation techniques designed for practical application in large-scale instances. A comprehensive experimental framework is established to evaluate these algorithms across diverse scenarios, varying in the number of machines and job weight distributions.

The findings highlight the conditions under which specific **algorithms** perform optimally or near-optimally, providing insights into their strengths and limitations. For example, certain algorithms demonstrate exceptional efficiency in handling instances with highly skewed weight distributions, while others excel in scenarios with ordered weights. The research also delves into the trade-offs between computational complexity and solution quality, noting that some algorithms achieve faster runtimes at the expense of optimality, while others deliver highly optimal solutions with increased computational demands.

These contributions advance the understanding of scheduling with local precedence constraints, filling a notable gap in existing research. The results offer a robust foundation for future studies and practical implementations, paving the way for more efficient and effective scheduling strategies in relevant industries.

Declaration

I hereby declare that this thesis is my own work and that I have not used any unauthorized assistance. I acknowledge the use of generative AI tools, in the development and refinement of this thesis. All code and data supporting this research are available at my GitHub repository: `https://github.com/dpkass/Bachelor-Thesis`.

Contents

1 Introduction **3**

2 Problem Definition **6**
2.1 Formal Problem Definition 6
2.2 Dynamic Programming Approach 8

3 Exploratory Analysis **15**
3.1 Experiment Setup 15
3.2 Algorithm Definitions 17
3.3 Observations 21
3.3.1 Relative Performance Ratio 21
3.3.2 Standard Deviation of RPR 24
3.3.3 Relative Improvement 24

4 Conclusion and Future Work **27**

A Omitted Proofs **29**
A.1 Proof of Equation 2.1 29
A.2 Proof of Equation 2.3 29

List of Theorems **31**

List of Algorithms **32**

Bibliography **33**

Chapter 1

Introduction

Scheduling is a cornerstone problem in operations research and computer science, pivotal for optimizing the utilization of resources across various industries such as manufacturing, logistics, and computing. Efficient scheduling algorithms are essential for minimizing operational costs, reducing processing times, and enhancing overall productivity. In manufacturing, for instance, effective scheduling ensures that production lines operate smoothly, minimizing downtime and maximizing output. Similarly, in logistics, optimal scheduling of deliveries can lead to significant cost savings and improved service quality. In the realm of computing, especially in parallel and distributed systems, scheduling tasks efficiently is crucial for maximizing computational throughput and minimizing latency.

This thesis focuses on a specific variant of the scheduling problem characterized by **local precedence constraints**. In this context, precedence constraints must be respected only within each machine, rather than globally across all machines. This means that while each machine has its own set of job orderings that must adhere to a predefined global sequence, there are no dependencies between jobs assigned to different machines. Such constraints are prevalent in real-world scenarios. For example, in manufacturing processes, certain tasks on the same machine must follow a specific order to ensure product quality and operational efficiency. Similarly, in parallel computing, tasks assigned to the same processor may have dependencies that necessitate a particular execution sequence to maintain system stability and performance.

The primary objective in this scheduling problem is to **minimize the weighted sum of completion times**. This metric is a reflection of both efficiency and cost-effectiveness, as it accounts for the importance or cost associated with each job. By minimizing this sum, the schedule ensures that higher-weighted (more critical or costly) jobs are completed earlier, thereby optimizing overall performance and resource utilization.

However, the introduction of local precedence constraints adds a layer of complexity to the scheduling problem. Balancing the load across multiple machines while respecting these orderings of tasks requires sophisticated algorithmic strategies. The challenge lies in navigating the combinatorial explosion of possible job assignments and sequences, especially as the number of jobs and machines increases.

The landscape of scheduling problems is vast, encompassing various models and constraints that cater to different real-world applications. Classical scheduling problems, such as **single-machine scheduling**, have been extensively studied, with numerous algorithms developed to find optimal

or near-optimal solutions [2]. These foundational studies provide a basis for understanding more complex scheduling scenarios.

In **multi-machine scheduling**, the complexity of effectively distributing jobs across machines becomes substantially greater. Greedy algorithms and list scheduling are commonly used heuristics that strive to balance the load while minimizing performance metrics like makespan. Despite their simplicity and efficiency in many scenarios, these heuristics often fall short when precedence constraints are introduced. Global precedence constraints, which mandate that certain jobs must be completed before others across all machines, introduce intricate dependencies that complicate the scheduling process. Its NP-hardness has long been established, and even the best polynomial-time approximation algorithms for this problem face limitations, with approximation ratios no better than $\frac{4}{3}$ [5]. This underscores the significant computational challenges posed by precedence constraints in multi-machine scheduling.

Existing literature on scheduling with precedence constraints encompasses both exact algorithms and approximation methods. Exact algorithms, including dynamic programming and branch-and-bound techniques, strive to find optimal solutions but often suffer from high computational costs, making them impractical for large instances. On the other hand, approximation algorithms offer solutions that are provably close to optimal within certain bounds, providing a balance between solution quality and computational efficiency. Heuristic methods, such as genetic algorithms and simulated annealing, have also been explored for their ability to find good-enough solutions in a reasonable timeframe, especially when dealing with complex or large-scale scheduling problems.

Despite the extensive research, there remains a notable gap in the study of scheduling problems with **local precedence constraints**. Most existing studies focus on global precedence constraints or scenarios without any precedence relations. The specific case where precedence constraints are localized within each machine, based on a global job order, has not been investigated. This gap highlights the need for specialized algorithms and analytical approaches to address the unique challenges posed by local precedence constraints. Additionally, the computational complexity of this problem variant remains **undetermined**, further emphasizing the necessity for in-depth research and exploration in this area.

The online platforms The Scheduling Zoo [1] provides interactive tools for experimenting with various scheduling problem configurations and offer access to related research papers. They also provide a comprehensive overview of complexity analyses and the best known polynomial-time approximation algorithms (or the absence thereof). However, *The Scheduling Zoo* currently does not include configurations for scheduling with local precedence constraints, thereby underscoring the existing gap in research and the need for specialized algorithms to address this particular variant.

This thesis presents significant advancements in the study of scheduling with local precedence constraints, addressing gaps in existing research and contributing to a deeper understanding of this complex problem. Central to this work is the development and evaluation of a diverse suite of algorithms specifically designed for scheduling under local precedence constraints. These include exact methods, such as **dynamic programming**, and **heuristic and approximation techniques** aimed at providing practical solutions for large-scale instances.

To support this algorithmic innovation, we designed a comprehensive **experimental framework** that evaluates algorithm performance across a wide variety of instance types. These instances vary in terms of machine numbers and weight distributions, ensuring a rigorous and thorough assessment of the proposed methods under diverse conditions. This experimental setup enables the systematic analysis of algorithmic efficacy, scalability, and stability, providing valuable insights into their strengths and limitations.

The findings from our extensive experimentation highlight key aspects of algorithm performance. We identified the conditions under which specific algorithms perform optimally or near-optimally, such as scenarios with varying machine numbers or distinct weight distributions. This enables practitioners to select the most suitable algorithm based on the characteristics of the problem instance. Furthermore, the study reveals notable performance trends. For instance, certain algorithms demonstrate exceptional efficiency in handling instances with highly skewed weight distributions, while others are better suited for scenarios with ordered weights. Understanding these trends is crucial for informed decision-making in applying scheduling algorithms to real-world problems.

Additionally, the research provides critical insights into the trade-offs between computational complexity and solution quality. While some algorithms prioritize faster runtimes at the expense of optimality, others achieve highly optimal solutions with increased computational demands. These trade-offs underscore the importance of balancing efficiency and effectiveness when selecting algorithms for practical applications. Together, these contributions advance the field of scheduling with local precedence constraints, offering a robust foundation for future research and development.

The thesis is systematically organized to present these findings in a coherent and logical manner. Following this introductory chapter, Chapter 2 delves into the formal problem definition and the dynamic programming approach developed to address it. Chapter 3 outlines the experimental setup and methodology used for evaluating the algorithms, and presents the experimental results, accompanied by a detailed analysis. Finally, Chapter 4 concludes the thesis, summarizing the key contributions and suggesting avenues for future research.

Chapter 2

Problem Definition

Scheduling is a fundamental problem in operations research and computer science, pivotal for optimizing resource utilization across various industries such as manufacturing, logistics, and computing. Efficient scheduling ensures that tasks are completed in a timely manner while minimizing costs and maximizing productivity.

In this thesis, we focus on a specific variant of the scheduling problem characterized by **local precedence constraints**. Unlike global precedence constraints, which require a strict ordering of tasks across all machines, local precedence constraints mandate that the precedence order is adhered to only within each individual machine based on a predefined global order of jobs. This distinction allows for greater flexibility in scheduling, as jobs can be assigned to different machines without violating the local precedence requirements, provided that the order of jobs on each machine respects the global job ordering.

2.1 Formal Problem Definition

To formally define the scheduling problem under consideration, we introduce the following components and notations:

Definition 1 (Jobs)**.** *Let $J = \{1, 2, \dots, n\}$ denote an ordered set of n jobs. Each job $j \in J$ is associated with a weight $w_j \in \mathbb{R}^+$, representing its importance or cost.*

Definition 2 (Machines)**.** *Let $M = \{1, 2, \dots, m\}$ denote a set of m identical machines available for processing the jobs.*

Let S_k denote the sequence of jobs assigned to machine $k \in M$.

Precedence constraints are defined based on the global order of jobs in J, but are restricted to processing on a specific machine.

Definition 3 (Local Precedence Constraints)**.** *Within each machine $k \in M$, if job i is processed before job j, then it must hold that $i < j$ in the global ordering of J.*

Note, that there are no precedence relations between jobs assigned to different machines. We assume that each job has a unit processing time.

Definition 4 (Completion Time)**.** *The completion time C_j of a job j is defined as the time at which the job finishes processing on its assigned machine m, which is the number of jobs which precede j on m plus* 1*.*

Definition 5 (Objective Function)**.** *The primary objective is to* ***minimize the total weighted sum of completion times****, defined as:*

$$\min \sum_{j \in J} w_j C_j,$$

whilst adhering to the local precedence constraints.

To elucidate the concept of local precedence constraints based on a global job order, consider the following example:

Suppose we have three jobs $J = \{1, 2, 3\}$ with weights $w_1 = 2$, $w_2 = 3$, and $w_3 = 1$, and two machines $M = \{A, B\}$. The jobs are globally ordered as $1 < 2 < 3$. The local precedence constraints require that on any given machine, if a job precedes another, it must do so in the global order.

One feasible schedule could be:

Example 1 (Schedule 1)**.**

- *Machine A: Jobs* $1 \to 3$
- *Machine B: Job* 2

Here, on Machine A, job 1 *precedes job* 3*, which is consistent with the global order (*$1 < 3$*). Job* 2 *on Machine B can be scheduled independently, even if it starts before job* 3 *on Machine A, as there are no precedence relations between machines.*

The completion times would be:

$$C_1 = 1, \quad C_3 = 2, \quad C_2 = 1.$$

The total weighted sum of completion times is:

$$\sum_{j \in J} w_j C_j = 2 \times 1 + 3 \times 1 + 1 \times 2 = 2 + 3 + 2 = 7.$$

Another feasible schedule could be:

Example 2 (Schedule 2)**.**

- *Machine A: Jobs* $1 \to 2$
- *Machine B: Job* 3

The completion times would be:

$$C_1 = 1, \quad C_2 = 2, \quad C_3 = 1.$$

The total weighted sum of completion times is:

$$\sum_{j \in J} w_j C_j = 2 \times 1 + 3 \times 2 + 1 \times 1 = 2 + 6 + 1 = 9.$$

Comparing both schedules, Schedule 1 has a lower total weighted sum of completion times, illustrating how the assignment and sequencing of jobs affect the objective.

Definition 6 (Mathematical Formulation)**.** *The scheduling problem can be mathematically formulated as a surjective function $\phi : J \to M$, while optimizing*

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} w_j C_j \\ \textit{subject to} \quad & C_j = 1 + \sum_{\substack{i \in J \\ i<j, \phi(i)=\phi(j)}} 1 \quad \forall j \in J. \end{aligned}$$

Scheduling with local precedence constraints presents unique challenges compared to traditional scheduling problems. The global order of jobs imposes a structured flexibility, allowing jobs to be assigned to different machines while maintaining local precedence within each machine. This structure can be exploited to design more efficient algorithms. However, balancing the load across multiple machines while adhering to the global order constraints introduces complexity in finding optimal or near-optimal solutions.

Moreover, this problem variant models real-world scenarios where tasks have inherent priorities or dependencies, but these constraints are localized within specific resources or processes. Examples include manufacturing assembly lines, parallel processing tasks in computing, and project management with resource-specific task dependencies. Addressing this problem can lead to more efficient scheduling algorithms that are both computationally feasible and effective in minimizing the total weighted completion time, thereby enhancing operational efficiency in various domains.

2.2 Dynamic Programming Approach

To address the computational challenges posed by the scheduling problem with local precedence constraints, we explore a **Dynamic Programming (DP)** approach aimed at finding the optimal solution. This approach systematically examines all possible job assignments and sequences across machines to identify the schedule that minimizes the total weighted sum of completion times.

Definition 7 (State)**.** *A **state** is defined as a tuple $p = (p_1, p_2, \dots, p_m)$, where p_k denotes the number of jobs assigned to machine k. Each state represents a unique configuration of job assignments across the machines.*

Theorem 1 (Optimality of the DP Algorithm)**.** *The Dynamic Programming algorithm correctly computes the optimal schedule that minimizes the total weighted sum of completion times under local precedence constraints.*

Proof. We establish the correctness of the DP algorithm by invoking the **Principle of Optimality**, which states that an optimal solution to a problem contains within it optimal solutions to its subproblems.

Base Case: The DP algorithm initializes with the base state where no jobs have been assigned to any machine. For this state, the total weighted completion time is zero:

$$DP(0, 0, \dots, 0) = 0.$$

Inductive Step: Assume that for all states representing assignments of the first $j-1$ jobs, the DP table correctly stores the minimum total weighted completion time. We proceed to assign the j-th job by considering all possible machine assignments. Let $p = (p_1, p_2, \dots, p_m)$ represent the current state, where p_k indicates the number of jobs assigned to machine k.

The next state p' is determined by assigning job j to machine k, resulting in:

$$p' = (p_1, p_2, \dots, p_k + 1, \dots, p_m),$$

where $k \in M$. For each successor state p', the new completion time for job j, denoted as $C_{j,k}$, is calculated as $p_k + 1$, indicating the position of job j on machine k.

The DP value for the successor state p' is computed using the recurrence relation:

$$DP(p') = \min_{k \in M} \left\{ DP(p) + w_j \times C_{j,k} \right\},$$

where w_j is the weight of job j. This ensures that for each state, the minimum total weighted completion time is recorded.

It is important to note that different paths may lead to the same successor state. In such cases, the DP table retains only the minimum DP value for that state across all possible paths leading to it. This process guarantees that the optimal solution is built incrementally by considering all possible assignments in a structured manner.

By iterating through all n jobs and considering all machine assignments at each step, the DP algorithm ensures that the optimal schedule is found. The algorithm terminates with the state that has all jobs assigned, representing the minimum total weighted completion time across all possible schedules. □

Definition 8 (State Space)**.** *The* ***state space****, denoted as S, is the set of all possible states $p = (p_1, p_2, \dots, p_m)$ that the DP algorithm may encounter during its execution.*

Theorem 2 (Time and Space Complexity of the DP Algorithm)**.** *The Dynamic Programming algorithm has a time complexity of*

$$\mathcal{O}\left(\frac{n^m}{(m!)^2}\right),$$

and a space complexity of

$$\mathcal{O}\left(p_m(n) - p_m(n-2)\right),$$

where $p_k(n)$ denotes the number of integer partitions of n into at most k non-negative parts.

To analyze the computational complexities of the DP algorithm, we progressively refine our upper bounds through a series of lemmas, ultimately leading to the main theorem regarding time and space complexity.

Lemma 1 (Naive Upper Bound)**.** *The size of the state space $|S|$, and consequently the space and time complexities, are bounded above by m^n.*

Proof. In the naive approach, each of the n jobs can be assigned to any of the m machines independently. Therefore, the total number of possible assignments is m^n. □

This naive upper bound of m^n indicates that the state space grows exponentially with the number of jobs, which becomes computationally infeasible for large n.

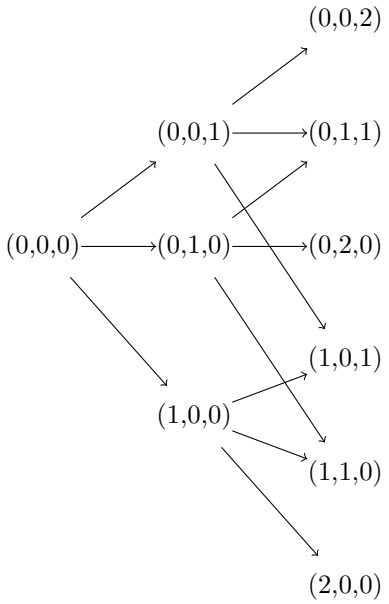

Figure 2.1: State Space for $m = 3$, $n = 2$ with overlapping successors.

Lemma 2 (State Space Upper Bound)**.** *The size of the state space $|S|$ is bounded above by n^m.*

Proof. Each machine can have up to n jobs, and there are m machines. Thus, the total number of possible states is bounded by n^m. □

This upper bound of n^m is significantly smaller than m^n, because the number of machines m is much less than the number of jobs n. This upper bound applies primarily to the space complexity, as each state could (and will) require its own computation.

The next step in finding a smaller upper bound involves determining how many successors overlap, or equivalently, how many predecessors each state has. A state has as many predecessors as there are machines with more than 0 jobs assigned to them, since each $p_k > 0$ can be decremented by 1 to yield a distinct predecessor.

Before deriving the number of overlapping successors, let's revisit our definition of states. A state is solely defined by the number of jobs assigned to each machine. These can overlap, as illustrated in Figure 2.1.

For the following derivation, we augment the state definition by including its predecessor:

Definition 9 (State With Predecessor)**.** *A state is defined as $s = (p, s')$, where s' is the predecessor of s, and p remains $(p_1, p_2, \ldots, p_m)$.*

***Note:** This definition is only valid until the proof of Corollary 1.*

This modification eliminates overlapping successors, as seen in Figure 2.2. Using this definition, we calculate the number of states that share the same value p but have different predecessors s'.

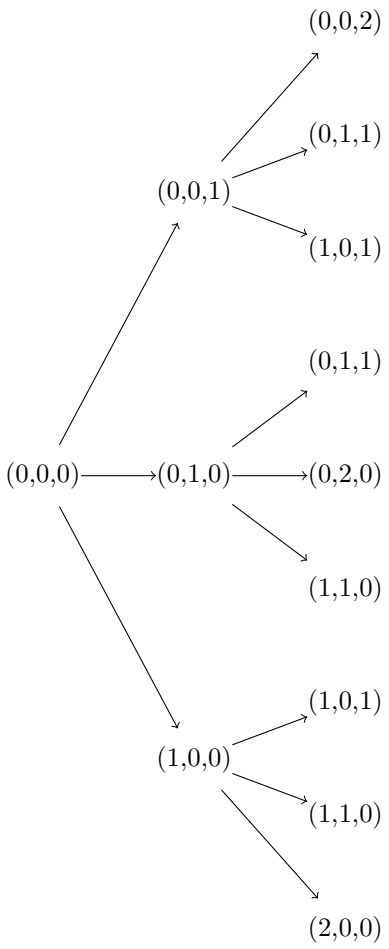

Figure 2.2: State Space for $m = 3$, $n = 2$ without overlapping successors.

Proposition 1 (Number of States with l Predecessors (No Overlap))**.** *The number of states that have l predecessors for given n and m is described by the following function:*

$$SP_{n,m}(l) = \sum_{n_i=0}^{n} \binom{m}{l} \left\{ {n_i \atop l} \right\} l!,$$

where $\left\{ {n_i \atop l} \right\}$ denotes the Stirling number of the second kind.

Proof. There are $\binom{m}{l}$ ways to select l out of the m machines. For each selected subset of l machines, there are $\left\{ {n_i \atop l} \right\}$ ways to partition n_i jobs into l non-empty subsets. Additionally, there are $l!$ ways to assign these subsets to the selected machines. □

Corollary 1 (Exact State Space Size & Time Complexity without Overlapping Successors)**.** *Following Proposition 1, the exact number of states and the corresponding time complexity (using Definition 9) is given by $|S| = \sum_{l=0}^{m} SP_{n,m}(l)$.*

Proof. The state space size is the sum of the number of states with each possible number of predecessors. Since each state has only one predecessor in this configuration, the time complexity is exactly proportional to the state space size. □

Thus, the state space size is the sum of all possible configurations with different numbers of predecessors, calculated without accounting for overlapping states.

Now, reverting back to our original definition of states 7, we consider merging states by taking the minimum of the possible values for that state. To calculate the state space size under this new definition, we introduce a new counting function for the number of states with a given number of predecessors.

Proposition 2 (Number of States with l Predecessors)**.** *The number of states that have l predecessors for given n and m is described by the following function:*

$$SP'_{n,m}(l) = \sum_{n_i=0}^{n} \binom{m}{l} c_l(n_i),$$

where $c_k(n) = \binom{n-1}{k-1}$ is the number of compositions (ordered partitions) of n into exactly k parts.

Proof. Basically, we replaced $\binom{m}{l}\left\{ {n_i \atop l} \right\}$ with $c_l(n_i)$. $l!$ can be discarded, because compositions already account for the ordering. The Stirling number $\left\{ {n_i \atop l} \right\}$ counts the ways to partition n_i jobs into l subsets, while $c_l(n_i) = \binom{n_i-1}{l-1}$ represents the number of ordered partitions of n_i into l parts. By focusing on the cardinality of each subset, which defines the state p, we align with the definition of integer compositions. □

Lemma 3 (State Space Size with Overlapping Successors)**.** *The size of the state space considering overlapping successors is $\frac{n^{\overline{m}}}{m!}$, where $n^{\overline{m}}$ denotes the Pochhammer Symbol (rising factorial).*

Proof. Analogously to Corollary 1, we derive the state space size as $|S| = \sum_{l=0}^{m} SP'_{n,m}(l)$.

Using the following binomial identity (proven in A.1)

$$\sum_{k=0}^{m} \binom{m}{k} \binom{n-1}{k-1} = \binom{n+m-1}{m}, \tag{2.1}$$

$m \setminus n$	**0**	**1**	**2**	**3**	**4**	**5**
1	1	2	3	4	5	6
2	1	3	6	10	15	21
3	1	4	10	20	35	56
4	1	5	15	35	70	126

Table 2.1: $n^{\overline{m}}$ for Selected Indices m and n

and the identity

$$\sum_{k=0}^{m} \binom{n+k}{n} = \binom{n+m+1}{m} \text{ [6]}, \tag{2.2}$$

we obtain:

$$\begin{aligned}
|S| &= \sum_{l=0}^{m} \sum_{n_i=0}^{n} \binom{m}{l} c_l(n_i) \\
&= \sum_{n_i=0}^{n} \sum_{l=0}^{m} \binom{m}{l} \binom{n_i - 1}{l-1} && \text{(composition definition)} \\
&= \sum_{n_i=0}^{n} \binom{n_i + m - 1}{m} && (2.1) \\
&= \binom{n+m}{m} && (2.2) \\
&= \frac{(n+m)!}{m!\, n!} && \text{(binomial coefficient definition)} \\
&= \frac{n^{\overline{m}}}{m!} && \text{(Pochhammer Symbol definition).}
\end{aligned}$$

□

Corollary 2 (Time Complexity with Overlapping Successors)**.** *When considering overlapping successors, the time complexity is reduced to* $\frac{n^{\overline{m}}}{m!} \cdot m \in \mathcal{O}\left(\frac{n^{\overline{m}}}{(m-1)!}\right)$

This is true, because we calculate each state from it's predecessors, each of which can have up to m.

To further optimize the time and space complexities, we employ **symmetry breaking** techniques. Since assigning jobs to machines in different orders but with the same number of jobs per machine results in equivalent states, we can enforce an ordering to eliminate redundant states. Specifically, when multiple machines have the same number of assigned jobs, we always choose the machine with the smallest index.

By transitioning from compositions (ordered partitions) to unordered partitions, we effectively reduces the state space by recognizing symmetric states as identical.

Lemma 4 (State Space Size with Symmetry Breaking)**.** *By enforcing an ordering on machine assignments, the state space size can be further reduced to:*

$$|S| = \sum_{n_i=0}^{n} p_m(n_i) \in \mathcal{O}\left(\frac{n^m}{m \cdot (m!)^2}\right), \tag{2.3}$$

where $p_k(n)$ denotes the number of integer partitions of n into at most k non-negative parts.

Note that the asymptotic growth of $p_m(n_i)$, for fixed m, is

$$p_m(n_i) \sim \mathcal{O}\left(\frac{n^{m-1}}{m!(m-1)!}\right) [3].$$

See the proof for Equation 2.3 in A.2.

Lemma 5 (Space Complexity with Layer-by-Layer Processing)**.** *By storing only the current and preceding layers of states, the space complexity can be further optimized to:*

$$\textit{Space Complexity: } \mathcal{O}\left(p_m(n) - p_m(n-2)\right),$$

where $p_m(n)$ is the number of integer partitions of n into at most m parts.

Proof. Each state depends only on its preceding states. We define a layer of states as all states that represent having processed i jobs, formally $L_i = \{p \mid \sum_k p_k = i\}$.

Each layer L_i depends only on the preceding layer L_{i-1}, since each state $p \in L_i$ is derived from assigning a job to one of the machines in states from L_{i-1}. By maintaining only the current and preceding layers and discarding older layers, we limit the number of stored states at any time to:

$$|L_i| + |L_{i-1}| = p_m(n_i) - p_m(n_{i-2}) \text{ for each step } i.$$

Therefore, the space complexity is bounded by:

$$\text{Space Complexity: } \mathcal{O}\left(p_m(n) - p_m(n-2)\right).$$

□

Proof for Theorem 2. In Lemma 5, the space complexity was already proven.

From Lemma 4, by applying symmetry breaking, the state space size is reduced to

$$|S| \in \mathcal{O}\left(\frac{n^m}{m \cdot (m!)^2}\right).$$

As stated before, states are derived from their respective predecessors, of which each state has up to m. Thus, the time complexity is

$$\text{Time Complexity: } \mathcal{O}\left(|S| \cdot m\right) \subseteq \mathcal{O}\left(\frac{n^m}{(m!)^2}\right)$$

□

Chapter 3

Exploratory Analysis

This chapter presents the exploratory analysis conducted to evaluate the performance of various scheduling algorithms under local precedence constraints. By systematically experimenting with different algorithms and instance generators, we aim to identify the most effective strategies for minimizing the total weighted sum of completion times. The analysis seeks to uncover each algorithm's strengths and weaknesses across diverse job weight distributions and machine configurations. The following sections detail the experimental setup and provide comprehensive definitions of the algorithms under consideration.

3.1 Experiment Setup

In this section, we describe the experimental setup used to evaluate the performance of various scheduling algorithms under local precedence constraints. The setup encompasses the computational environment, the algorithms implemented, the instance generators for creating test cases, the parameters considered, the performance metrics employed, and the data collection methods.

We evaluated six algorithms designed to address the scheduling problem with local precedence constraints in distinct ways. These include:

- **Dynamic Programming (DP) Algorithm**: Serves as the optimal solution and benchmark for comparing other algorithms.
- **Least Loaded**: A simple heuristic that assigns jobs sequentially to the machine that has the lowest sum of weighted completion times.
- **Heavy First**: Sorts jobs by weight in non-increasing order and assigns the heaviest jobs first, following a greedy strategy.
- **k-Lookahead Algorithm**: Incorporates future job information by considering the next k jobs before making a scheduling decision.
- **Sort & Split**: Sorts jobs by weight and splits them into m equally sized arrays, then assigns each array to a machine while maintaining local precedence.

- **Balanced Sequential Insert (BSI)**: Aims to balance the workload across machines by distributing jobs based on their weights.

To assess the algorithms across diverse scenarios, we employed several instance generators that produce different weight distributions. These generators are categorized as follows:

Generator Name	Interval	Description
Increasing Weights	$[1, n]$	Generates jobs with weights increasing sequentially in the range $[1, n]$.
Decreasing Weights	$[1, n]$	Generates jobs with weights decreasing sequentially in the range $[1, n]$.
Small Weights	$[1, 100)$	Generates jobs with uniformly random weights.
Small Span Large Weights	[100,000, 100,100)	Generates jobs with uniformly random weights.
Large Span Large Weights	[10,000, 100,000)	Generates jobs with uniformly random weights.
Large Span Non-Decreasing Weights	[1, 100,000)	Generates jobs with uniformly random weights sorted in non-decreasing order.
Large Span Non-Increasing Weights	[1, 100,000)	Generates jobs with uniformly random weights sorted in non-increasing order.
Low Then High Weights	$[1, 100)$ and $[900, 1{,}000)$	The first half of the jobs have weights in $[1, 100)$, and the second half in $[900, 1{,}000)$.
High Then Low Weights	$[900, 1{,}000)$ and $[1, 100)$	The first half of the jobs have weights in $[900, 1{,}000)$, and the second half in $[1, 100)$.

Table 3.1: Instance Generators for Scheduling Algorithms

These instance generators were chosen to simulate a variety of real-world scenarios and to understand how different weight distributions affect the performance of the scheduling algorithms. By testing the algorithms on both ordered and random weight distributions, we aim to uncover their strengths and weaknesses under varying conditions.

The experiments were conducted using the following parameters:

- **Number of Jobs (n)**: 150.
- **Number of Machines (m)**: 2, 3, and 4.
- **Random Seeds**: 0 to 9

The random seeds ensure reproducibility and allow us to assess performance stability across different random states. To evaluate and compare the algorithms, we employed the following performance metrics:

- **Relative Performance Ratio (Quality)**: The ratio of the algorithm's total weighted completion time to that of the optimal solution.
- **Standard Deviation of RPR**: The standard deviation of the relative performance ratio over different seeds.

- **Relative Improvement**: The improvement in performance relative to increasing the number of machines m.

These metrics are used to highlight differences between algorithms, measure the stability of each algorithm's performance, and assess whether algorithms perform better or asymptotically approach the optimal solution as the number of machines increases.

Results from the experiments were stored using `xarray` DataArrays and Datasets, facilitating easy manipulation and analysis of multi-dimensional data. Each computed value was saved in NetCDF (`.nc`) files for persistence and later retrieval.

3.2 Algorithm Definitions

In this section, we provide detailed descriptions of the algorithms evaluated in our experiments. Each algorithm is presented with pseudocode and an intuitive explanation to clarify its operation. We aim for consistency in the presentation to facilitate understanding and comparison.

The following notation is introduced:

- S_k is the sequence of jobs currently assigned to machine k.
- $t(S_k)$ computes the sum of weighted completion times of jobs assigned to machine k.
- `defaultdict(x)` returns a hashmap. When accessing a key that does not exist, it returns the default value x.

We consider two implementations of the dynamic programming (DP) approach:

Dictionary-Based DP (DP-DICT): Utilizes a dictionary to store state-value pairs, enhancing memory efficiency by only retaining relevant states. This approach may introduce minimal computational overhead due to dictionary operations.

Multi-Dimensional Array DP (DP-MDIM): Employs a multi-dimensional array to represent states, allowing for faster access times. However, this method incurs significantly higher memory consumption, especially as the number of machines increases.

For detailed theoretical underpinnings and the correctness of the DP approach, please refer back to Chapter 2.

Algorithm 1 Dictionary-Based Dynamic Programming (DP-DICT)

Input: Ordered set of jobs J, set of machines M, job weights w
Output: Minimum weighted sum of completion times
Initialize $dp \leftarrow \text{defaultdict}(\infty)$
$dp[(0, 0, \ldots, 0)] \leftarrow 0$
for $j \in J$ **do**
    $next\_dp \leftarrow \text{defaultdict}(\infty)$
    for $p \in dp.\text{keys}()$ **do**
        for $k \in M$ **do**
            if $p[k] < p[k-1]$ or $k = 1$ **then**
                $p' \leftarrow (p[1], \ldots, p[k]+1, \ldots, p[m])$
                $next\_dp[p'] \leftarrow \min(next\_dp[p'], dp[p] + p'[k] \cdot w_j)$
            end if
        end for
    end for
    $dp \leftarrow next\_dp$
end for
return $\min(dp.\text{values}())$

The DP-DICT algorithm iteratively builds a dictionary of states, where each state p represents a possible distribution of jobs across machines. For each job j, it considers assigning it to each machine k under the symmetry-breaking condition ($p[k] < p[k-1]$ or $k = 1$) to avoid redundant computations. The cost of a state is updated based on the cumulative weighted completion time. By using a dictionary, the algorithm efficiently stores only relevant states, optimizing memory usage.

Algorithm 2 Multi-Dimensional Array Dynamic Programming (DP-MDIM)

Input: Ordered set of jobs J, set of machines M, job weights w
Output: Minimum weighted sum of completion times
Initialize dp array of shape $[n+1, \lceil (n+1)/2 \rceil, \ldots, \lceil (n+1)/m \rceil]$ with ∞
$dp[0, 0, \ldots, 0] \leftarrow 0$
Initialize $states \leftarrow \{(0, 0, \ldots, 0)\}$
for $j \in J$ **do**
    $next\_states \leftarrow \emptyset$
    for $p \in states$ **do**
        for $k \in M$ **do**
            if $p[k] < p[k-1]$ or $k = 1$ **then**
                $p' \leftarrow (p[1], \ldots, p[k]+1, \ldots, p[m])$
                $dp[p'] \leftarrow \min(dp[p'], dp[p] + p'[k] \cdot w_j)$
                $next\_states \leftarrow next\_states \cup \{p'\}$
            end if
        end for
    end for
    $states \leftarrow next\_states$
end for
return $\min(dp[p])$ for all $p \in states$

Similar to DP-DICT, the DP-MDIM algorithm explores possible job assignments to machines but uses a multi-dimensional array for state storage. Each dimension corresponds to a machine, and indices represent the number of jobs assigned. The algorithm updates the array by considering all valid states and keeps track of the minimal cost. While access times are faster, this method requires more memory, making it less practical for larger problems.

Algorithm 3 Least Loaded

Input: Ordered set of jobs J, set of machines M, job weights w
Output: Weighted sum of completion times
Initialize priority queue pq with m entries, each $(0, 0)$
for $j \in J$ **do**
    Extract (t_m, c_m) with minimal t_m from pq
    $c_m \leftarrow c_m + 1$
    $t_m \leftarrow t_m + w_j \cdot c$
    Push (t_m, c_m) back into pq
end for
return $\sum_{(t_m, c_m) \in pq} t_m$

The Least Loaded algorithm assigns each job in the order of J to the machine with the current minimal total weighted completion time. By always choosing the least loaded machine, it aims to balance the cost. The sum of completion times t_m and job count c_m are updated for each machine. This greedy approach is particularly fast and simple.

Algorithm 4 Heavy First

Input: Ordered set of jobs J, set of machines M, job weights w
Output: Total weighted sum of completion times
Sort J in non-increasing order of w_j
Initialize $state = [S_1, \ldots, S_m]$
for $j \in J$ **do**
    for $m \in M$ **do**
        Compute *potential_cost* if job j is assigned to machine m
    end for
    Assign job j to m with minimal *potential_cost* and update *state*
end for
return $\sum_{k \in M} t(S_k)$

The Heavy First algorithm assigns jobs in order of decreasing weight, in a greedy fashion. For each job, it evaluates the potential cost of assigning it to each machine and selects the machine that results in the minimal increase in total weighted completion time. This helps in placing heavy jobs where they have the least negative impact. As it only accounts for current state, the negative impact on future placements is not regarded.

Algorithm 5 k-Lookahead

Input: Ordered set of jobs J, set of machines M, job weights w, lookahead parameter k
Output: Total weighted sum of completion times
Initialize $state = [S_1, \dots, S_m]$
for $j \in J$ **do**
    for $m \in M$ **do**
        Simulate assigning job j to machine m
        Compute *potential_cost* for next $k-1$ jobs using DP (1)
    end for
    Assign job j to m with minimal *potential_cost* and update *state*
end for
return $\sum_{k \in M} t(S_k)$

The k-Lookahead algorithm also improves upon the greedy approaches by considering the impact of current decisions on the next k jobs. For each job j, it simulates assigning it to each machine m and computes the potential cost using DP for the subsequent $k-1$ jobs. It then selects the machine that results in the minimal estimated total cost, aiming for better long-term scheduling decisions. To compute *potential_cost*, simply adjust Algorithm (1) by initializing $dp[(|S_1|, \dots, |S_m|)] \leftarrow \sum_{m \in M} t(S_m)$.

Algorithm 6 Sort & Split

Input: Ordered set of jobs J, set of machines M, job weights w
Output: Total weighted sum of completion times
Sort J in non-increasing order of w_j
Split sorted jobs into m job sequences: $L_1, \dots, L_m$
for $k \in M$ **do**
    Sort L_k by precedence
    Assign L_k to machine k
end for
return $\sum_{k \in M} t(S_k)$

The Sort & Split algorithm sorts the jobs in non-increasing order of their weights and divides them into equal subsets, assigning each subset to a different machine. By grouping jobs based on weight, it reduces competition between heavier and lighter jobs for minimal completion times. However, this method does not account for the specific impact of individual job assignments, which may prevent it from achieving an optimal schedule.

Algorithm 7 Balanced Sequential Insert

Input: Ordered set of jobs J, set of machines M, job weights w
Output: Total weighted sum of completion times
Sort J in non-increasing order of w_j
Initialize $state \leftarrow [S_1, \ldots, S_m]$
Initialize $i \leftarrow 1$
while not all jobs are assigned **do**
    Reset *state*
    $S_1 \leftarrow J[1, \ldots, i]$
    $t_1 \leftarrow t(S_1)$
    for $k \in M \setminus \{1\}$ **do**
        while $t(S_k) < t_1$ **do**
            Assign next job to machine k
        end while
    end for
    $i \leftarrow i + 1$
end while
return $\sum_{k \in M} t(S_k)$

Building upon the Sort & Split approach, the Balanced Sequential Insert (BSI) algorithm dynamically assigns jobs to machines to further balance the total weighted completion times. After sorting the jobs by weight, BSI incrementally assigns the heaviest jobs to the first machine and distributes subsequent jobs to other machines in a manner that maintains balanced weighted completion times across all machines. If not all jobs can be assigned under the current distribution, the algorithm increases the number of jobs assigned to the primary machine and redistributes the remaining jobs accordingly. This iterative process ensures a more balanced and efficient distribution of jobs, enhancing the effectiveness of the initial Sort & Split method.

3.3 Observations

In this section, we analyze the performance of the scheduling algorithms across different instances and configurations, focusing on the Relative Performance Ratio (RPR), the standard deviation of RPR, and the Relative Improvement when increasing the number of machines.

3.3.1 Relative Performance Ratio

The Relative Performance Ratio (RPR) is defined as the ratio of an algorithm's solution to the optimal solution, averaged over different seeds. Figure 3.1 presents a heatmap of the RPR for all combinations of generators and algorithms at $m = 2, 4, 6$ machines.

To facilitate analysis, we organized the generators and algorithms in a deliberate order, grouping similar ones together. For the generators, certain pairs exhibit equivalent behavior: *Constant* and *Random Small Span Large*, *+1 Increasing* and *Random Non-Decreasing Large Span*, -1 *Decreasing* and *Random Non-Increasing Large Span*, and *Random Small* and *Random Large Span Large*. Thus, we simplified the set of generators to six distinct types. The **Constant** generator represents trivial cases and is not examined further. The **Non-Increasing** category, originally -1 *Decreasing*

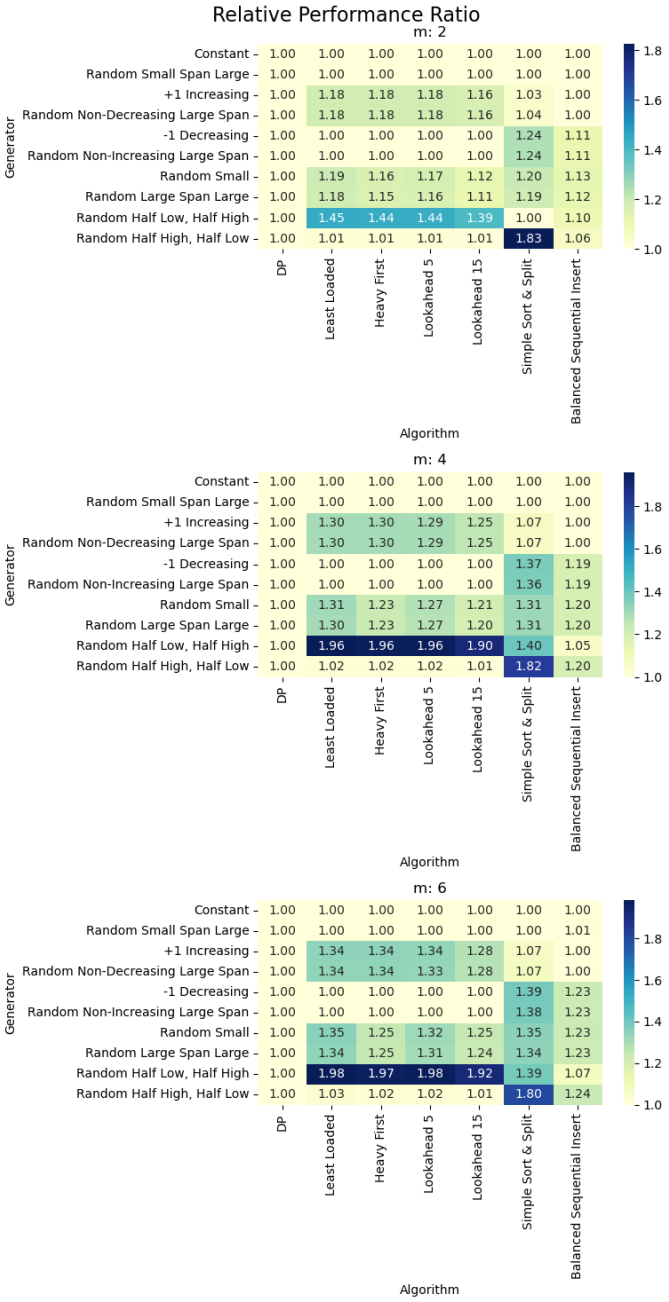

Figure 3.1: Relative Performance Ratio heatmap for all generators and algorithms at $m = 2, 4, 6$

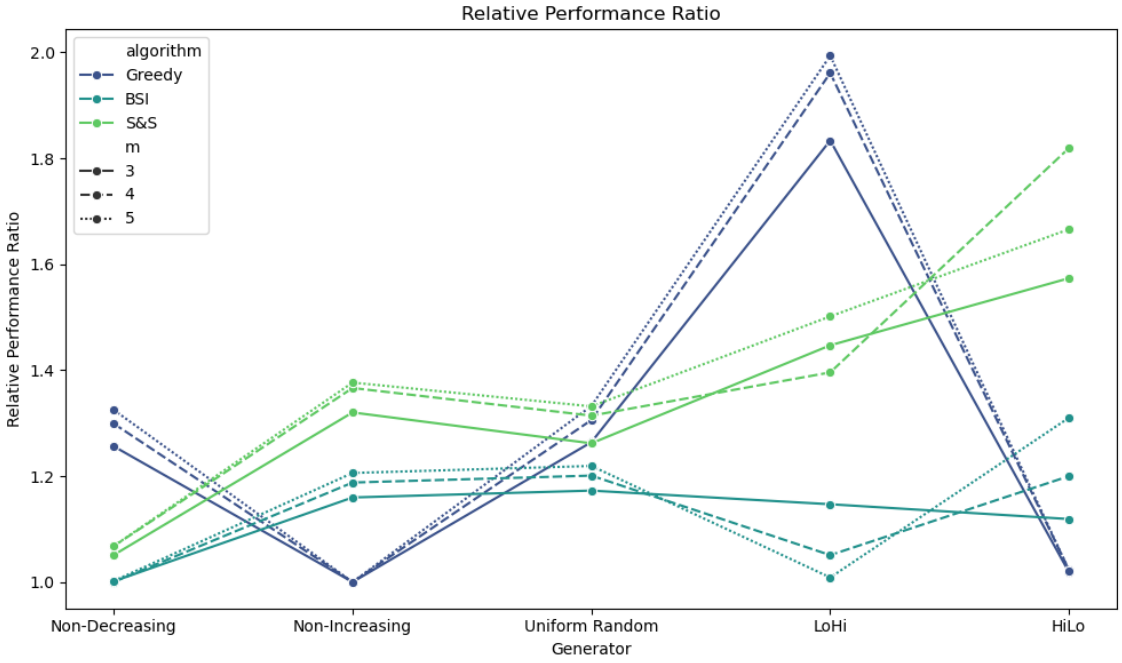

Figure 3.2: Relative Performance Ratio for selected algorithms and generators at $m = 3, 4, 5$

and *Random Non-Increasing Large Span*, henceforth represented by -1 *Decreasing*. Analogously, the **Non-Decreasing** category includes *+1 Increasing* and *Random Non-Decreasing Large Span*, represented by *+1 Increasing*. The **Uniform Random** category, originally *Random Small* and *Random Large Span Large*, is represented by *Random Small*; it is important that the maximum weight is several times greater than the minimum to avoid behavior similar to the *Constant* generator. The **LoHi** and **HiLo** generators are originally named *Half Low, then Half High* and *Half High, then Half Low*, respectively.

For the algorithms, we observed that *Least Loaded*, *Heavy First*, *Lookahead 5*, and *Lookahead 15* perform very similarly, with *Lookahead 15* having a slight edge. The *Sort & Split* algorithm generally underperforms compared to others, while the *Balanced Sequential Insert (BSI)* algorithm maintains consistent performance as the number of machines increases and clearly outperforms *Sort & Split*. Therefore, we simplified the set of algorithms to four main categories: **Dynamic Programming (DP)** as the optimal solution, **Balanced Sequential Insert (BSI)**, **Sort & Split (S&S)**, and **Greedy**, which represents the group of algorithms originally called *Least Loaded*, *Heavy First*, *Lookahead 5*, and *Lookahead 15*.

Focusing on the *Greedy*, *BSI*, and *S&S* algorithms with the simplified set of generators, Figure 3.2 shows the RPR for these algorithms at $m = 3, 4, 5$.

From the figure, we observe that the *BSI* algorithm generally outperforms the others, except for the *Non-Increasing* and *HiLo* generators, where the *Greedy* algorithm performs better. In these instances, the weights are such that they benefit from being uniformly distributed across the

machines, aligning with the behavior of the *Greedy* algorithm. This suggests that combining the strengths of both *BSI* and *Greedy* algorithms could lead to near-optimal performance across different types of instances.

Figure 3.3 provides a heatmap highlighting the strengths and weaknesses of each algorithm for the selected generators across a range of machines from $m = 2$ to 6.

The *Greedy* algorithm exhibits three distinct performance groups. It performs very well on the *Non-Increasing* and *HiLo* generators, with RPR in the range [1.0, 1.02]. On the *Non-Decreasing* and *Uniform Random* generators, its performance is poor, with RPR in the range [1.18, 1.35]. The algorithm performs very poorly on the *LoHi* generator, with RPR ranging from 1.45 to 1.98.

The *BSI* algorithm shows relatively stable performance across generators and machine counts. It performs very well on the *Non-Decreasing* generator, with an RPR around 1.0. On the *LoHi* generator, it performs well, with RPR between 1.01 and 1.15. For the *Non-Increasing* and *Uniform Random* generators, it achieves mediocre performance, with RPR between 1.11 and 1.23. However, its performance fluctuates on the *HiLo* generator, with RPR ranging from 1.06 to 1.31.

The *Sort & Split* algorithm performs poorly to very poorly across the board, showing acceptable performance only with the *Non-Decreasing* generator. These observations reinforce the potential benefit of combining the *Greedy* and *BSI* algorithms to exploit their respective strengths. By selecting the appropriate algorithm based on the instance characteristics, we can achieve near-optimal performance across different generators. Together, they provide very good performance on the *Non-Increasing*, *Non-Decreasing*, and *HiLo* generators, with RPR between 1.0 and 1.02; good performance on the *LoHi* generator, with RPR between 1.01 and 1.15; and mediocre performance on the *Uniform Random* generator, with RPR between 1.11 and 1.23.

3.3.2 Standard Deviation of RPR

We examined the standard deviation of the RPR over different seeds to assess the stability of the algorithms. Figure 3.4 displays a bar chart of the standard deviation for all randomized generators and algorithms (excluding *DP*) at $m = 5$.

Overall, the standard deviations are relatively low, less than 4%, indicating consistent performance across different seeds. The *Uniform Random* generators exhibit the highest variability. The standard deviation is largely indifferent to the number of machines m, showing minimal differences. While this variability is acceptable in the current context, it could become problematic at larger scales, suggesting an area for future work.

3.3.3 Relative Improvement

We analyzed the Relative Improvement in the solution when increasing the number of machines from $m - 1$ to m, defined as the percentage reduction in the solution value. Figure 3.5 illustrates the Relative Improvement for all algorithms (excluding *DP*), averaged over randomized generators and seeds, across $m = 2, 3, 4, 6$.

The results show no major differences among the algorithms, with all following a similar trend. This suggests that the algorithms neither asymptotically approach the optimal solution as the number of machines increases nor significantly diverge. Further investigation into asymptotic behavior over larger numbers of jobs (n) could provide additional insights.

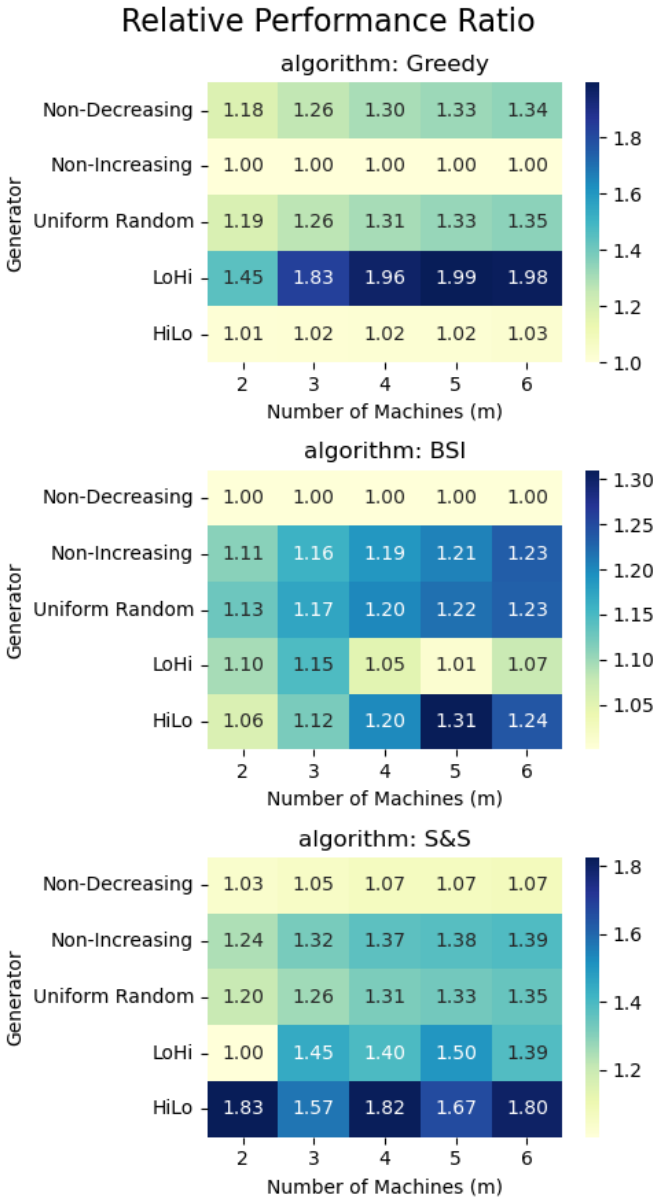

Figure 3.3: Relative Performance Ratio heatmap for selected algorithms and generators across $m = 2$ to 6

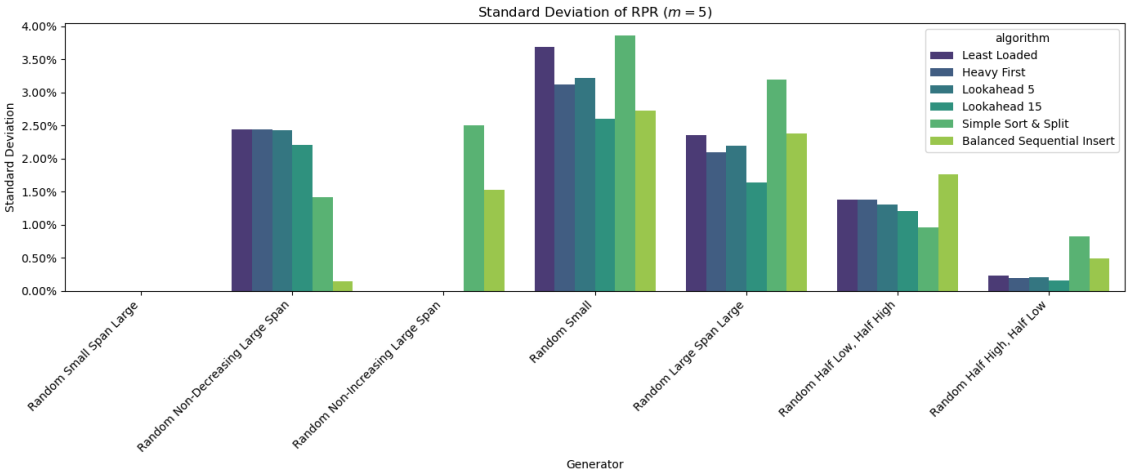

Figure 3.4: Standard deviation of Relative Performance Ratio over seeds for randomized generators at $m = 5$

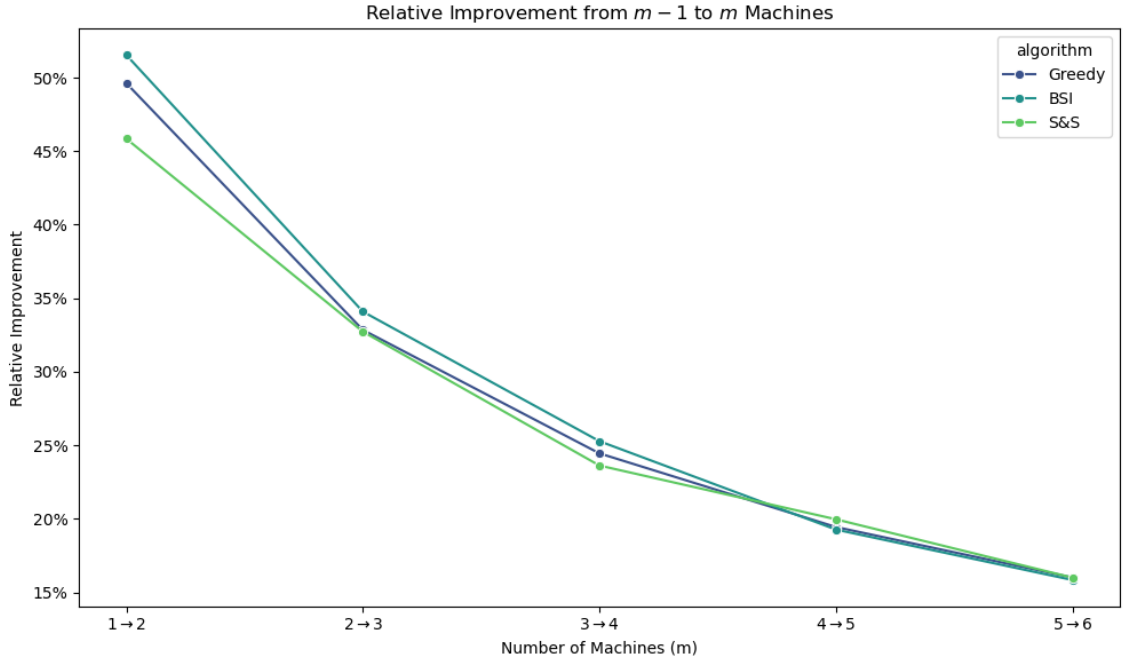

Figure 3.5: Relative Improvement in solution quality when increasing the number of machines

Chapter 4

Conclusion and Future Work

This thesis has delved into the complex domain of scheduling with local precedence constraints, addressing significant gaps in existing research and contributing valuable insights to the field. By developing and evaluating a diverse suite of algorithms tailored specifically for this problem variant, we have enhanced the understanding of how different strategies perform under varying conditions. The comprehensive experimental framework established in this study allowed for a rigorous assessment of algorithmic efficacy, scalability, and stability across a wide array of instance types, including variations in machine numbers and weight distributions.

Our findings highlight the strengths and limitations of each algorithm, providing practitioners with guidance on selecting the most suitable method based on specific problem characteristics. Notably, the *Balanced Sequential Insert (BSI)* algorithm demonstrated robust performance across several scenarios, particularly in handling instances with highly skewed weight distributions. Conversely, greedy algorithms excelled in instances where weights decreased or followed a high-to-low pattern. These insights underscore the importance of algorithm selection in achieving near-optimal solutions and offer a foundation for informed decision-making in practical applications.

Despite these advancements, the research presented in this thesis is not without limitations. The scope of instance types considered, while diverse, does not encompass the full spectrum of real-world scenarios. Computational resource constraints also limited the size and complexity of the instances we could feasibly analyze, potentially affecting the generalizability of the results. Acknowledging these limitations provides a pathway for future research endeavors aimed at building upon and extending the work initiated here.

Looking forward, several avenues offer promising directions for further exploration and development. Enhancing the algorithms through optimization and the incorporation of more advanced heuristics or metaheuristic approaches could yield significant improvements in performance and computational efficiency. For instance, exploring genetic algorithms or tabu search methods may provide more adaptive and robust solutions capable of handling larger and more complex scheduling instances. Specifically, optimizing existing algorithms, such as implementing a ternary search instead of a linear search in the BSI algorithm, could reduce computational costs and improve scalability.

Expanding the problem scope presents another opportunity for gaining deeper insights into scheduling complexities. Investigating scheduling with global precedence constraints would address a broader class of problems where dependencies extend across all machines, introducing new challenges and necessitating more sophisticated algorithmic solutions. Additionally, considering

variable processing times and machine-specific constraints would align the problem more closely with real-world conditions, where such variations are commonplace. This expansion would require the development of more generalized models and could lead to the discovery of novel scheduling strategies.

Applying the algorithms to real-world datasets obtained from industry partners would not only validate their practical applicability but also uncover unique challenges inherent in operational environments. Such empirical studies could reveal factors not accounted for in theoretical models, prompting refinements and adaptations that enhance the algorithms' effectiveness in practice. Assessing practical feasibility and effectiveness through real-world application is crucial for bridging the gap between theoretical research and industry needs.

Addressing scalability concerns is essential for the algorithms to remain relevant as the size and complexity of scheduling problems continue to grow. Developing parallel implementations of the algorithms could significantly improve their ability to handle larger instances by leveraging modern multi-core processors and distributed computing systems. This enhancement would expand the algorithms' applicability to high-demand industrial settings where large-scale scheduling is a critical component of operational efficiency.

Finally, overcoming the limitations identified in this study is vital for advancing the field. Future research should aim to diversify the range of instance types examined, incorporating more varied and complex scenarios to test the algorithms' robustness and adaptability. Employing advanced computational resources or optimizing algorithmic efficiency could mitigate computational constraints, allowing for the analysis of larger and more intricate scheduling problems. By addressing these challenges, subsequent studies can enhance the reliability and applicability of scheduling solutions in diverse and dynamic environments.

In conclusion, this thesis has made meaningful contributions to the understanding and solving of scheduling problems with *local precedence constraints*. The development of specialized algorithms and the comprehensive analysis of their performance provide valuable insights for both academia and industry. The proposed extensions and future research directions offer a roadmap for continued exploration, promising to further advance the field and unlock new possibilities for efficient and effective scheduling across various domains.

Appendix A

Omitted Proofs

A.1 Proof of Equation 2.1

To prove, we use **Vandermonde's convolution**:

$$\binom{n+m}{r} = \sum_{k=0}^{r} \binom{n}{k}\binom{m}{r-k},$$

and the symmetry property of binomial coefficients:

$$\binom{n}{k} = \binom{n}{n-k}.$$

Starting with the left-hand side:

$$\begin{aligned}
\sum_{k=0}^{m} \binom{m}{k}\binom{n-1}{k-1} &= \sum_{k=0}^{m} \binom{n-1}{k-1}\binom{m}{k} && \text{(reordering terms)} \\
&= \sum_{k=0}^{m} \binom{n-1}{k}\binom{m}{m-k} && \text{(applying symmetry)} \\
&= \binom{n+m-1}{m}, && \text{(by Vandermonde's convolution)}
\end{aligned}$$

which proves the proposition.

A.2 Proof of Equation 2.3

To prove, we use the following key equations:

1. The asymptotic formula for the number of partitions of n into at most k parts:

$$p_k(n) \sim \frac{1}{(k-1)!}\frac{n^{k-1}}{k!}[3]$$

2. Faulhaber's formula for $\sum_{i=0}^{n} i^p$:

$$\sum_{i=0}^{n} i^p \sim \frac{n^{p+1}}{p+1} + \mathcal{O}(n^p).$$

Starting with the given summation:

$$\begin{aligned}
\sum_{n_i=0}^{n} p_m(n_i) &\sim \sum_{n_i=0}^{n} \frac{n_i^{m-1}}{(m-1)!m!} && \text{(using } p_m(n) \text{ asymptotics)} \\
&= \frac{1}{(m-1)!m!} \sum_{n_i=0}^{n} n_i^{m-1} && \text{(factoring constants)} \\
&\sim \frac{1}{(m-1)!m!} \left(\frac{n^m}{m} + \mathcal{O}(n^{m-1}) \right) && \text{(applying summation of powers)} \\
&= \frac{n^m}{m \cdot (m!)^2} + \mathcal{O}\left(\frac{n^{m-1}}{(m!)^2} \right). && \text{(simplifying terms)}
\end{aligned}$$

Thus, the summation satisfies:

$$\sum_{n_i=0}^{n} p_m(n_i) \in \mathcal{O}\left(\frac{n^m}{m \cdot (m!)^2} \right).$$

List of Theorems

Definition 1 (Jobs) . . . 6
Definition 2 (Machines) . . . 6
Definition 3 (Local Precedence Constraints) . . . 6
Definition 4 (Completion Time) . . . 7
Definition 5 (Objective Function) . . . 7
Definition 6 (Mathematical Formulation) . . . 8
Definition 7 (State) . . . 8
Theorem 1 (Optimality of the DP Algorithm) . . . 8
Definition 8 (State Space) . . . 9
Theorem 2 (Time and Space Complexity of the DP Algorithm) . . . 9
Lemma 1 (Naive Upper Bound) . . . 9
Lemma 2 (State Space Upper Bound) . . . 10
Definition 9 (State With Predecessor) . . . 10
Proposition 1 (Number of States with l Predecessors (No Overlap)) . . . 10
Corollary 1 (Exact State Space Size & Time Complexity without Overlapping Successors) 12
Proposition 2 (Number of States with l Predecessors) . . . 12
Lemma 3 (State Space Size with Overlapping Successors) . . . 12
Corollary 2 (Time Complexity with Overlapping Successors) . . . 13
Lemma 4 (State Space Size with Symmetry Breaking) . . . 13
Lemma 5 (Space Complexity with Layer-by-Layer Processing) . . . 14

List of Algorithms

1 Dictionary-Based Dynamic Programming (DP-DICT) 18
2 Multi-Dimensional Array Dynamic Programming (DP-MDIM) 18
3 Least Loaded 19
4 Heavy First 19
5 k-Lookahead 20
6 Sort & Split 20
7 Balanced Sequential Insert 21

Bibliography

[1] P. Brucker, S. Jäger, C. Dürr, S. Knust, D. Prot, R. van Stee, and Óscar C. Vásquez. The scheduling zoo. `http://schedulingzoo.lip6.fr/`, 2024. A searchable bibliography in scheduling, following the 3-field notation introduced by Graham et al.

[2] C. Chekuri and R. Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics*, 98(1):29–38, 1999.

[3] *NIST Digital Library of Mathematical Functions*. `https://dlmf.nist.gov/`, Release 1.2.2 of 2024-09-15. F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, B. V. Saunders, H. S. Cohl, and M. A. McClain, eds. Specific reference to Section 26.9, Equation (26.9.E10).

[4] *NIST Digital Library of Mathematical Functions*. `https://dlmf.nist.gov/`, Release 1.2.2 of 2024-09-15. F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, B. V. Saunders, H. S. Cohl, and M. A. McClain, eds. Specific reference to Section 26.11, Equation (26.11.E3).

[5] J. Lenstra and A. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Oper. Res.*, 26(1):22–35, 1978.

[6] M. R. Spiegel. *Mathematical Handbook of Formulas and Tables*. Schaum's Outline Series. McGraw-Hill, New York, 1968.