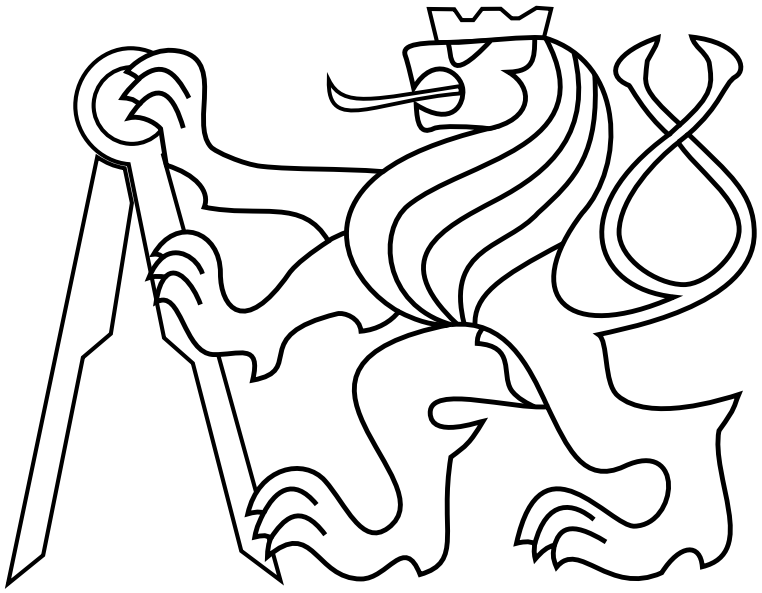
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering
Department of Computer Science

MASTER'S THESIS



Vilém Heinz

Scheduling Algorithms for Non-overlapping Setups Problem

Study program: Open Informatics
Field of study: Artificial Intelligence

Thesis supervisor: **Ing. Antonín Novák**

Prague 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Heinz** Jméno: **Vilém** Osobní číslo: **435324**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Rozvrhovací algoritmy pro problém s nepřekrývajícími se přestavbovými časy

Název diplomové práce anglicky:

Scheduling Algorithms for Non-overlapping Setups Problem

Pokyny pro vypracování:

1. Seznamte se s problémem rozvrhování s nepřekrývajícími se přestavbovými časy. Analyzujte existující a příbuzné práce.
2. Formalizujte problém optimalizace délky rozvrhu s nepřekrývajícími se sekvenčně závislými přestavbovými časy. Rozšiřte model o nededikované úlohy a o překrývání maximálně k přestaveb.
3. Navrhněte a implementujte exaktní a heuristické algoritmy pro řešení daného problému. Zaměřte se i na integraci obou přístupů v rámci hybridního algoritmu.
4. Otestujte algoritmy na dostupných benchmarcích a porovnejte s existujícími příbuznými přístupy.

Seznam doporučené literatury:

[1] Young Hoon; Lee Mi-Yi Kim. MIP models and hybrid algorithm for minimizing the makespan of parallel machines scheduling problem with a single server. Computers & Operations Research, 2017.
[2] Gokalp Yildiz; Alper Hamzadayi. Modeling and solving static m identical parallel machines scheduling problem with a common server and sequence dependent setup times. Computers & Industrial Engineering, 2011.
[3] Xiaoyue Zhang; Simin Huang; Linning Cai. Parallel dedicated machine scheduling problem with sequence-dependent setups and a single server. Computers & Industrial Engineering, 2009.
[4] Vlk, M.; Novák, A.; Hanzálek, Z.; Malapert, A. Non-overlapping Sequence-Dependent Setup Scheduling with Dedicated Tasks In: Operations Research and Enterprise Systems. Cham: Springer, 2020. p. 23-46.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Antonín Novák, katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **08.09.2020** Termín odevzdání diplomové práce: **05.01.2021**

Platnost zadání diplomové práce: **19.02.2022**

Ing. Antonín Novák podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne..... .....

Poděkování

Děkuji svému vedoucímu Ing. Antonínu Novákovi za jeho odbornou pomoc, připomínky a nápady k obsahu práce a za čas, který mi věnoval. Dále děkuji Mgr. Marku Vlkovi za jeho poskytnutou expertízu v tzv. Constrant Programmingu a cenné připomínky k heuristickým přístupům. Děkuji i panu prof. Dr. Ing. Zdeňku Hanzálkovi za jeho předmět Kombinatorické optimalizace, který mě k tématice přivedl. Poděkování patří také mé rodině za jejich vytrvalou podporu během studia.

Abstract

This thesis examines the problem of task scheduling on parallel identical machines with setup operation for every pair of scheduled consecutive tasks. The length of the setup is sequence-dependent, and the setup must be executed in-between the relevant tasks by a machine setter. On any machine at any given time, there can be only one task or setup processed. No machine setter can work multiple setups at the same time. The goal is to minimize the makespan, which is the time when the last machine ends. We propose Constraint programming (CP) models to find the optimal solution and heuristic algorithms that are suitable for large instances of the problem. We propose warm start techniques that greatly increase the CP effectivity. We propose a hybrid algorithm called SMETI, which combines the mentioned methods with sub-problem CP modeling to provide fast, good results while being able to reach optimum. Experimental comparisons to the state-of-the-art solutions of similar problems show the proposed approaches to be more effective, finding the same quality solution faster or providing a better solution at the same time limit.

Keywords: Scheduling; Identical parallel machines; Multiple servers; Sequence-dependent constrained setup times; Constraint programming; Heuristics; Hybrid algorithm

Abstrakt

Tato diplomová práce se zabývá rozvrhováním úloh na paralelní identické stroje, s operací přenastavení stroje pro každou dvojici naplánovaných úloh. Délka této operace je závislá na obou úlohách. Operace musí být vykonána pracovníkem mezi danou dvojící úloh. Na každém stroji může být v jeden okamžik vykonávána právě jedna úloha či operace přenastavení. Žádný pracovník nesmí vykonávat více operací zároveň. Úkolem je minimalizovat tzv. makespan, což je konec poslední úlohy na nejpozději končící mašině.
V práci navrhujeme Constraint programming (CP) modely které jsou schopné najít optimální řešení a heuristické algoritmy, které jsou vhodné pro velké instance kde CP není dostatečně výkonný. Navrhujeme tzv. warm start techniky, které výrazně zvyšují efektivitu CP modelů. Dále navrhujeme hybridní algoritmus zvaný SMETI, který kombinuje výše zmíněné metody s tzv. subproblem CP modelováním, který poskytuje velmi kvalitní výsledky v dobrém čase a zároveň má schopnost dosáhnout optimálního řešení. Experimentální porovnání s tzv. state-of-the-art přístupy ukazuje, že navržené metody jsou efektivnější, dosahují stejné kvality řešení rychleji či lepšího řešení ve stejném čase.

Klíčová slova: Rozvrhování; Identické paralelní stroje; Vícero serverů; Operace závislé na pořadí; Constraint programming; Heuristiky; Hybridní algoritmus

Contents

List of Figures

List of Tables

1 Introduction

With today's trends in manufacturing, with factories handling a large variety of products on multiple production lines, customer's demand for highly proprietary products of small-batch volumes, and companies forced to adapt to market shifts constantly, we are in an environment where the organization of work and effective utilization of production capacities is a crucial, but also very complex, problem to tackle. Creating a production schedule by trial and error is often inconvenient and ineffective even for very small productions. From an effective and efficient solution to this multifaceted problem can benefit almost any type of production, increasing its possible output and/or reducing costs while seamlessly tailoring the production to the customer's needs.

In this thesis, we focus on a typical situation often encountered in production scheduling. We have a given number of parallel identical machines, which can also represent whole identical production lines, tasks that are to be executed on these machines, and setups that are to be performed between those tasks. Any task can be scheduled for any machine. Between any two consecutive tasks on every machine, a setup operation must be performed by a machine setter. The setup's length depends on the particular pair of tasks, referred to as sequence-dependent setups in the scientific literature. The number of available machine setters in production can be arbitrary but is set for the whole problem. The machine setters are considered identical, meaning that any machine setter can perform any setup. This exact problem was solved in one company manufacturing plastic tubes. The company produced many types of these tubes on multiple machines, and machine setters were performing setups necessary between producing two different types. As there were different adjustments needed between every pair of tube types, the setup times were sequence-dependent. This is just one real-world example of many possible existing productions that can benefit from a solution to the considered problem.

It might seem that sequence-dependent setups might be a niche restricted to a very narrow type of productions where frequent mechanical adjustments on machines are needed, but it is actually a common requirement. We show this on an example of a paint shop. Even though changing the color tank is always the same operation, the amount of time needed is also dependent on cleaning up the nozzles spraying the color. This operation always has the same principle, but it takes a different time depending on how clean we need the nozzles. In case we would paint using orange color and then red, we might not have to clean the nozzles at all. However, if we paint red and then white, any residue left in the nozzles would result in an obvious defect, meaning we have to clean the nozzles very thoroughly. But painting these in reverse order is also different since little white in red is way less visible than the other way around. Therefore we do not only care about individual pairs of tasks but also about their ordering. This is also true for foods, printing, chemical, and many other types of productions.

Below is an example schedule of the problem respecting these constraints. In this case, we have 3 machines (M_1 to M_3), 2 machine setters (W_1 and W_2) and 11 tasks (T_1 to T_{11}). Each row represents the schedule of one problem's resource, either a machine or a machine setter. Tasks are assigned and executed on machines and setups between pairs of tasks performed by machine setters. Setups on machines are marked by machine setter performing them. The zero represents the start and C_{max} the end of the schedule.

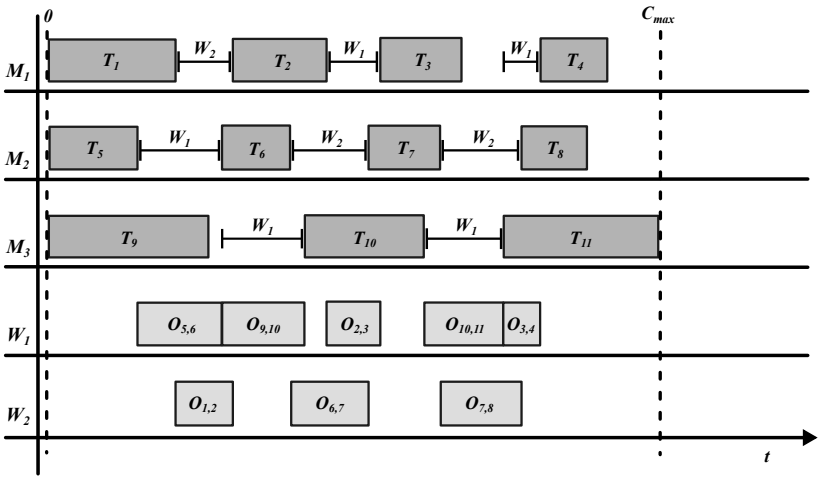


Figure 1: A Gantt chart of identical parallel machine production with non-overlapping sequence-dependent setup times performed by machine setters.

1.1 Problem Statement

In the subsequent text we will use following notation:

- Let $T = \{T_1, \ldots, T_t\}$ be a set of independent tasks where each task can be executed on any machine. We index using i and j.
- Let $M = \{M_1, \ldots, M_m\}$ be a set of machines where every machine can execute at most one task or have one setup performed on itself at a time. Machines are considered to be identical. We index using k.
- Let $W = \{W_1, \ldots, W_w\}$ be a set of available machine setters. Machine setters are considered to be identical. We index using x.
- Let $\boldsymbol{O} \in \mathbb{N}^{t \times t}$ be a matrix of setup times where indexes i and j will denote the specific pair of tasks T_i and T_j to which the setup belongs. So the setup time between tasks T_i and T_j with T_i preceding T_j is denoted $o_{i,j}$.

We also add following notations for other parameters of the problem. Every task is described by its processing time, denoted $p_i \in \mathbb{N}$. Let $s_i \in \mathbb{N}_0$ be start time and $c_i \in \mathbb{N}$ completion time of that task in the schedule. Every setup is described by its processing time between tasks, denoted $o_{i,j}$. Because not every setup between all possible pairs of tasks is used, we define set of used setups as $U = \{U_1, \ldots, U_u\}$. We index using y and z. Let $\overline{s}_y \in \mathbb{N}$ be the start, $\overline{p}_y \in \mathbb{N}$ the execution and $\overline{c}_y \in \mathbb{N}$ the completion time of the setup U_y in the schedule.

The constraints of the problem are:

(A1) Every task must be executed.

(A2) Tasks are non-preemptive, i.e., they cannot be suspended during the execution, so $s_i + p_i = c_i$ must hold.

(A3) If task T_j is next after T_i in the machine sequence, setup $o_{i,j}$ must be performed by machine setter and s_j - $c_i \geq o_{i,j}$ must hold. This also means that no setup must be executed before the first task on each machine. [1]

(A4) Setup must be performed by exactly one machine setter.

(A5) Setups are non-preemptive, $\overline{s}_y + \overline{p}_y = \overline{c}_y$ must hold for any setup from U.

(A6) If U_y is the setup between tasks T_i and T_j then $c_i \leq \overline{s}_y$ and $\overline{c}_y \leq s_j$ must hold.

(A7) For any two setups U_y, U_z performed by same W_x, $\overline{s}_z \geq \overline{c}_y$ or $\overline{s}_y \geq \overline{c}_z$ must hold. [2]

(A8) Machine setter can perform at most one setup at a time.

Since machine setters are considered identical, we do not have to represent which specific machine setter is performing which setup. To simplify the modeling of the problem, rather than considering each particular machine setter, we will think of it as how many machine setters we need at a certain time to perform the setups currently scheduled. Later we will show that this view is very efficient. After solving the problem this way, we can then assign machine setters to setups by taking arbitrary machine setter from the currently available, not performing, ones and assign them to the setups. There always exists a solution to machine setter to setup assignment, which is guaranteed by the found solution. It does not matter which machine setter we assign to which setup as long as that machine setter was available at the time of assignment.

The problem's objective is to minimize the schedule length, also further referred to as minimizing the makespan. In our case, this means finding a schedule that minimizes the completion time of the latest ending machine. This, in turn, also means minimizing the latest task completion time on such a machine. Subsequently, the whole problem looks as follows:

$$\min \max_{T_i \in T} c_i$$

s.t.

$$(A1) - (A8)$$

The formal statement of the considered problem.

[1] We can also solve instances where setup before the first task is required by adding virtual tasks, which is explained later.

[2] This condition can be inferred from other conditions but it is explicitly mentioned for clearer understanding.

1.2 Complexity of the Problem

The problem addressed in this thesis is a more general variant of the $\mathcal{NP}$-hard [19] problem tackled in *Vlk et al.* [29], meaning we can do a polynomial reduction of their problem and solve it using our approaches. The differences are that our problem admits multiple machine setters instead of just one, and our tasks are not dedicated to a particular machine only. When reducing their problem to ours, the availability of just one machine setter is not a problem. However, task dedication to a particular machine is a little trickier. We can enforce it by setting the setup times between all pairs of tasks dedicated to different machines in the original instance to be infinity. This makes the solution where two tasks from originally different machines are placed on the same machine infeasible. This shows that our problem is at least as hard as the problem in *Vlk et al.* [29] thus at least $\mathcal{NP}$-hard. We also know that the problem is in $\mathcal{NP}$ because we were successfully able to encode the problem using Constraint programming, which is $\mathcal{NP}$-complete.

1.3 Approach

To tackle the problem, we first employed Constraint programming (CP) [23]. CP, which will be described in more detail later, provides much like Mixed-integer programming (MIP) [15], a way to represent a problem using equations. We developed two CP models of the problem producing optimal solutions, provided enough time is given.

Unfortunately, most of the real-world problems in scheduling are $\mathcal{NP}$-hard. As shown before, this applies to the problem considered as well. This means that solving it up to the optimality is almost always infeasible. It is often more effective to employ a different approach that does not guarantee optimality but will provide satisfactory results in a far more reasonable time limit. Or, in case of large problem instances, provide at least some feasible solution rather than none. While testing various instances of our problem, we verified that it is simply impossible to use only CP to get a result, which would be satisfactory in every scenario. Therefore, we developed new constructive heuristic approaches, which can handle big instances of the problem. This also leads to the idea of a warm starting. Warm starting means using a solution obtained by another approach as a starting point for the main approach, in this case, CP, hoping that it improves the whole runtime. We developed a Vehicle Routing Problem (VRP) warm start by reducing the relaxed version of our problem and solving it using the *OR-Tools* [8] library. However, this did not entirely live up to our expectations of dramatically decreasing the runtime or improving the solution reached in the same time limit given. On the other hand, solving the instance using our fast proposed constructive heuristic algorithms and using that solution as a warm start brought huge improvements in computational speed and solution quality.

However, even with a warm start, CP and constructive heuristic algorithms are suitable for the opposite extremes of instance sizes. That is why we propose improvements to these heuristic algorithms, improving their results in exchange for time. However, this still does not cover every possible instance. More importantly, it also does not address the selection of a proper approach for a particular instance. We want the approach to adapt to a given instance and always proceed in a way that is the best for it and the time limit provided.

That is why we developed a hybrid approach combining most of the above with new additions to create an approach suitable for any problem instance. In this hybrid approach, constructive heuristic algorithms are used to provide a starting point. After that, CP models for subproblems of the original problem are built to decrease the model size by lowering the number of conditions imposed. When there are no more improvements to be had, we use the CP model of the original problem to further improve the solution up to the optimum. This approach is called *Subproblem Modeling Evolving Through Iteration*, shortly SMETI. SMETI provides fast results while also gradually converging to optimum, providing the ultimate way of solving the problem without considering which approach is the most suitable for which instance.

1.4 Contribution and Thesis Outline

The key contributions of this thesis are:

1. Addressing of new scheduling problem with machine setters[3], arising as a natural generalization of particular cases considered in existing literature, combining multiple machine setters, sequence-dependent setup times, machine-task independence with parallel identical machines, while also retaining the ability to solve the less general problems.
2. Proposition of CP models utilizing cumulative resource function which can solve instances of up to ten machines and tens of tasks to the optimality usually in a matter of minutes. Using the warm start technique, CP models can be used on instances of lower tens of machines and hundreds of tasks.
3. Proposition of domain-specific heuristics which can solve instances of hundreds of machines and tens of thousands of tasks in under a minute of runtime, with the solution often just single percents worse than the optimal one. The use of domain-specific heuristics over metaheuristics like Genetic Algorithms was preferred because metaheuristics are already widely used in scheduling problems like in paper *Lunardi et al.* [16] with their effectiveness studied in survey paper *Pellerin et al.* [20]. We believe that by proposing domain-specific heuristics, we provide a new angle of approach for similar scheduling problems and a deeper understanding of the considered problem's features.
4. Combination of constructive heuristics with CP modeling resulting in hybrid multi-step procedure SMETI. SMETI provides an effective way of solving any instance of the considered problem efficiently and adds a way of better solving instances that were not well suited for neither CP models nor heuristic approaches.
5. Improvement over the state-of-the-art approaches for less general problems. Both CP models and SMETI show improvements in efficiency over their respective state-of-the-art counterparts.

[3]Also referred to as (common) servers in the scientific literature.

The thesis consists of 9 sections. This section described the real-world problematics involved, formally stated the problem, introduced the naming scheme used in this thesis, and described the problem's constraints and connections. Section 2 lists scientific papers on similar thematic and discusses the current state-of-the-art approaches. Section 3 describes two proposed CP models used to solve the problem optimally if enough time is provided. Section 4 describes proposed constructive heuristic approaches best suited for larger instances where CP models are not fast enough. Section 5 explains the idea behind VRP warm starting and warm start use of constructive heuristic algorithms to improve CP models effectivity. Section 6 takes an extra look at the constructed heuristic algorithms' effectiveness and proposes ways of improving solution objective value in a trade-off with time. Section 7 describes the hybrid algorithm SMETI, which combines CP modeling with constructive heuristics. Section 8 discusses the differences between our approaches and compares the results to calculated bounds. The comparison to state-of-the-art approaches is provided in Subsection 8.3. Section 9 contains the conclusion of findings from this thesis.

2 Related Work

As survey paper *Allahverdi et al.* [3] shows, there are papers on sequence-dependent setup scheduling. Papers *Vallada et al.* [27] and *Lee et al.* [14] focus on heuristic approaches handling instances of up to lower hundreds of tasks in a few minutes of computation time. But the research on the problems where the setups require an extra resource is less prominent.

An Integer Linear Programming (ILP) formulation of scheduling problem with a single server/setup operator and two parallel identical machines is proposed in paper *Amir et al.* [1]. It is stated in the paper that the computational time of this approach for instances bigger than 12 jobs (tasks) is too excessive to be used. However, it is shown that if specific conditions are met, there exist very fast approaches that can be useful in such constraint scenarios.

A scheduling and lot sizing problem with a common setup operator is studied in *Tempelmeier et al.* [25]. ILP formulations for a problem described as a dynamic capacitated multi-item multi-machine one-setup-operator lot sizing problem are given. The setups performed by the operator are considered to be scheduled without overlapping. The setups are associated with the task following, so setups do not depend on a pair of tasks, which makes them sequence-independent. The proposed approach can solve instances of tens of tasks, usually under a 1-minute time limit.

A problem involving machines with setups performed by operators of different capabilities has been studied in *Chen et al.* [6]. The problem is modeled using time-indexed formulation and solved by decomposing the problem into smaller subproblems using Lagrangian relaxation. Subproblems are solved using Dynamic Programming (DP) [22]. A feasible solution to the problem is obtained by the composition of the subproblems' solutions using a heuristic algorithm. If that is impossible, the Lagrangian multipliers are updated using the surrogate subgradient method as in *Zhao et al.* [30]. This approach's downside is that the time-indexed formulation yields a pseudo-polynomial size model, which is especially not suitable if large processing and setup times are present. The results show that the proposed approach can handle instances of tens of machines in a matter of minutes.

Papers *Vlk et al.* [28] and *Vlk et al.* [29] study problem with sequence-dependent non-overlapping setups, but tasks are dedicated to machines, and only a single machine setter is assumed. In both papers, CP models, an ILP model, and heuristics utilizing the problem's decomposition are proposed. The resulting subproblems deal with task ordering on machines independently. The paper results show that proposed CP models can find a solution for instances of tens of machines and tens of tasks for each machine in 1 minute. However, the proposed LOFAS algorithm with a heuristic can solve instances of up to 1000 tasks on 5 machines.

The same problem as in *Vlk et al.* [29] is studied in *Huang et al.* [24]. Instead of CP, an ILP model is proposed, and the heuristic approach is realized using a Genetic Algorithm (GA). It is stated in the paper that ILP formulation is unusable in a real-world scenario. The GA approach can solve instances of 10 machines and 100 tasks in under 50 seconds.

The problem in *Hamzadayi et al.* [4] assumes the assignment of tasks to any machine. Compared to the two previous papers, setups are sequence-independent, so there is no direct way to compare this paper's generality to the aforementioned two. Again, only one machine setter is allowed. ILP formulations are provided, and the heuristic approach is again realized using a GA. The results show the MIP model can solve instances of 6 machines and up to 40 tasks optimally or close to the optimum in 3600s. However, our testing suggests that only in a minority of cases with the optimal solution could the model actually prove its optimality. The GA solves the same instances in under 1 minute, with 2 to 5 % worse results than the MIP.

Kim et al. [18] generalize *Vlk et al.* [29], *Huang et al.* [24] and *Hamzadayi et al.* [4], offering a solution to machine-independent and setup sequence-dependent problems. However, the problem addressed in this paper still lacks the support of multiple machine setters, which limits its usage in real-world scenarios. Exact solutions are realized using ILP while heuristic solution using a genetic algorithm. The results show that the MIP approach can find the optimal solution in 18000 seconds for the instance of 3 machines and 9 tasks. The GA solution shows the ability to solve instances of 10 machines and 100 tasks in one minute, providing a very good solution.

The papers discussed in this section are more or less ordered by their increasing problem complexity. It is very noticeable that as complexity increases, the ability to solve bigger instances of a particular problem goes down dramatically. This is especially true for the exact approaches realized using ILP/MIP. However, the results of the papers provided are only very general summarizations. The problems often have parameters involved in testing, and some even have additional conditions in the problem definition. The more detailed analysis of state-of-the-art approaches performance and how they compare to our approaches is in Section 8.

To sum up, the research shows that even though papers addressing similar problems exist, no papers tackle this type of problem with such generality as the proposed solutions in this thesis. Furthermore, the results from *Vlk et al.* [29] show that CP seems a far more suitable approach than MIP even though MIP is so often applied to this type of problem in the literature. This finding is also further demonstrated in Section 8.

3 Constraint Programming Approaches

The first approach we use to solve the problem is Constraint programming (CP). We propose two models differing in the way how they handle the modeling of constraints related to setups. The presented models extend ideas from models of a less general problem assuming only one machine setter and lacking non-dedication of tasks described in *Vlk et al.* [29].

In Subsection 3.1, we introduce basic and used concepts of CP. In Subsection 3.2, we describe the constraints shared across both models. The following two subsections 3.3 and 3.4 then describe the respective constraints of each model.

3.1 Preliminaries and Notation

First, we briefly introduce the main concepts of CP, and then we will move to the core parts and specifics of each model. Further information about CP modeling and *CP Optimizer* which are beyond the scope of the considered problem can be found in *CP Optimizer user's manual* [12]. The main modeling expression of CP is *interval variable*. As the name suggests, it represents some activity with its required time in the schedule. Its length can be set by expression LengthOf. Its start in the schedule is denoted by expression StartOf and its completion time by expression EndOf. We use the interval variables to represent the tasks

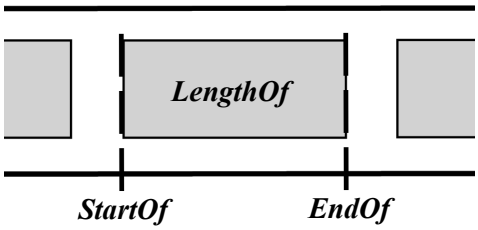


Figure 2: Interval variable expressions in the machine schedule.

and setups in our problem. The start and completion times of tasks and setups in the schedule are determined by the CP solver. Their execution time is given by the instance description. We will denote the set of interval variables representing tasks as I^T. Interval variable of specific task T_i will be denoted I_i^T.

Another important modeling construct of CP used in the proposed models is the NoOverlap constraint. This constraint ensures that for a given set of interval variables, no two interval variables from this set overlap each other in the produced schedule. In other words, in a sequence consisting of interval variables constrained by NoOverlap, only one interval variable is present at any given time in the schedule. We can also provide a so-called *transition matrix* to the NoOverlap constraint, which defines spaces between the end of one interval variable and the start of the next one. We use our setup times matrix $\boldsymbol{O} \in \mathbb{N}^{t \times t}$ as the *transition matrix* of the NoOverlap constraint to insert setup times between tasks in the sequence.

Another important concept is *optional interval variable*. This means that depending on the model's conditions; the optional interval variable can either be *present* or *absent*. The

absent variable does not have to conform to the model's constraints. The CP solver decides if the variable should be present or absent in the solution unless we specifically state otherwise. We can state the presence and absence of variable manually by introducing constraints which imply either or by using the constraint Alternative. The Alternative constraint is assigned a set of variables, and only one of the variables is then chosen to be present in the solution while others are set to be absent.

3.2 Common Modeling Principles

The core models' parts are shared over both proposed CP models. First, we need to ensure that every task is present in the produced schedule. We achieve this by assigning task lengths given by the problem instance to their respective interval variables representing them:

$$\textsc{LengthOf}(I_i^T) = p_i. \tag{CP-C1}$$

Since we do not know which task will be assigned to which machine, we use the constraint Alternative mentioned earlier. For every $I_i^T \in I^T$ we add a new optional interval variable $I_{i,k}^{T^{Opt}}$ to the schedule of every machine $M_k \in M$. Then, the CP solver decides which of these interval variables in every set will be present and absent depending on other constraints and optimization criterion. This way, every task is assigned to exactly one machine. We also need to ensure that the selected optional interval variable for every task T_i will be associated with interval I_i^T representing our task since those are also used in the model's conditions. This can be done by passing I_i^T as an argument of the Alternative constraint representing the so-called master interval variable. To this variable, the selected present optional variable is automatically assigned:

$$\forall I_i^T \in I^T : \textsc{Alternative}\left(I_i^T, \bigcup_{M_k \in M} \left\{I_{i,k}^{T^{Opt}}\right\}\right). \tag{CP-C2}$$

To ensure that tasks do not overlap on any particular machine, we use aforementioned NoOverlap constraint for every machine. We use the optional interval variables here because we do not know yet which task will be assigned to which machine so we let the solver handle it. We also supply the setup matrix $\boldsymbol{O}$ to ensure that at least the length of setup time between EndOf one present task and StartOf following present task in the sequence is reserved:

$$\forall M_k \in M : \textsc{NoOverlap}\left(\bigcup_{I_i^T \in I^T} \{I_{i,k}^{T^{Opt}}\}, \boldsymbol{O}\right). \tag{CP-C3}$$

In both CP models, we minimize schedule length, also called the makespan, which, as derived before, is actually the same as minimizing the end of the last task on the machine with the longest schedule. The interval variable of the task which is to be minimized is:

$$\textsc{Max}\left(\bigcup_{I_i^T \in I^T} \textsc{EndOf}(I_i^T)\right). \tag{CP-OBJ}$$

3.3 CP Pairwise: Pairwise Optional Setups

In this model, the concept of *optional interval variable* is utilized again. We will enforce a setups' presence, depending on all pairs of tasks on all machines. We achieve this by using the expression PresenceOf. If a certain pair of tasks is present on any machine, the optional setup is put into the expression PresenceOf, which will ensure its presence in the solution. In the following text, this model will be referred to as $\texttt{CP}_{Pairwise}$.

The optional interval variable representing specific setup between a pair of tasks T_i and T_j, $i \neq j$ will be denoted as $I_{i,j}^{S^{Opt}}$. The set of all optional interval variables representing setups between pairs of tasks will be denoted as $I^{S^{Opt}}$. There are $t(t-1)$ variables in this set. It is important to note, that while setup spaces in the makespan are already guaranteed thanks to the setup matrix $\boldsymbol{O}$ passed to the NoOverlap constraint, we need variables from $I^{S^{Opt}}$ to connect setups with machine setters. The lengths of these interval variables are set to corresponding setup times provided by the problem instance:

$$\forall I_{i,j}^{S^{Opt}} \in I^{S^{Opt}} : \textsc{LengthOf}(I_{i,j}^{S^{Opt}}) = o_{i,j}. \tag{CPP1}$$

To guarantee that the setups are processed in between the relevant pair of tasks and not anywhere else in the schedule, we use the constraint $\textsc{EndBeforeStart}(I_A, I_B)$. This constraint ensures that interval variable I_A is completed before interval variable I_B can start. If either of the interval variables is absent, the constraint is implicitly satisfied. Thus, the following constraints are added:

$$\forall I_i^T, I_j^T \in I^T, i \neq j : \textsc{EndBeforeStart}(I_i^T, I_{i,j}^{S^{Opt}}) \wedge \textsc{EndBeforeStart}(I_{i,j}^{S^{Opt}}, I_j^T). \tag{CPP2}$$

Finally, we ensure the presence of necessary optional interval variables representing setups between pairs of tasks present in the schedule. We use the expression $\textsc{Next}(I')$, which returns the next interval variable in the sequence. In our case, this is interval variable representing the task following the task given by the I'. This way, we can find out which task follows which in the schedule. According to this, we can enforce the presence of setup in between these two tasks. Notice that each task which is present will be followed by exactly one setup except for the last task on each machine, which has no following task:

$$\forall I_i^T, I_j^T \in I^T, i \neq j, \forall M_k \in M : \textsc{Next}(I_{i,k}^{T^{Opt}}) = I_{j,k}^{T^{Opt}} \implies \textsc{PresenceOf}(I_{i,j}^S). \tag{CPP3}$$

The last set of constraints limits the maximum number of setups running simultaneously to the number of machine setters available. There are many ways of achieving this, but the computationally superior way proposed in *Vlk et al.* [29] uses the expression $\textsc{Pulse}(I', a)$. This expression specifies that a units of resource, in our case machine setters, is used during interval I'. The cumulative function is composed of Pulse terms for each $I_{i,j}^{S^{Opt}}$ representing the usage of exactly one machine setter. The function must remain lower or equal than the number of machine setters w provided at any point in time in the schedule:

$$\sum_{\forall I_{i,j}^{S^{Opt}} \in I^{S^{Opt}}} \textsc{Pulse}(I_{i,j}^{S^{Opt}}, 1) \leq w. \tag{CPP4}$$

The complete model using the defined constraints looks as follows:

MINIMIZE	(CP-OBJ)
s.t.	
	(CP-C1) – (CP-C3)
	(CPP1) – (CPP4)

The $\texttt{CP}_{Pairwise}$ model.

3.4 CP Flexible: Flexible Setup Intervals

The key difference between this model and $\texttt{CP}_{Pairwise}$ is that we do not represent every possible setup by an optional interval variable. We avoid the quadratic number of setup constraints, which is very important for bigger problem instances. The number of interval variables representing setups in this model is only equal to the number of tasks. This is achieved by representing every possible setup following a task by only one interval variable. To make this possible, we do not set the length of the task and setup interval variables equal to their respective values given by instance description, but we use inequalities instead. In the following text, this model will be referred to as $\texttt{CP}_{Flexible}$.

First, we use inequality to relax the assignment of processing times to interval variables in I^T. The length of interval variable I_i^T will now be at least its processing time p_i. This way, the interval variable representing task can prolong itself and wait until a machine setter is available to perform the following setup. This will be useful later. Thus, the task length constraints imposed in (CP-C1) are relaxed because they must only be greater than the corresponding processing times:

$$\forall I_i^T \in I^T : \textsc{LengthOf}(I_i^T) \geq p_i. \tag{CPF1}$$

We define a set I_i^S representing setups following their respective task interval variable I_i^T from I^T. Considering that this way we have an interval variable representing setup after every task, the setup after the last task on each machine becomes a zero-length dummy setup. Due to that, we merely set the length of the setup variables to be at least zero and the actual length of the setup executed will be determined later:

$$\forall I_i^S \in I^S : \textsc{LengthOf}(I_i^S) \geq 0. \tag{CPF2}$$

The cumulative function use is very similar to $\texttt{CP}_{Pairwise}$. The only difference is that we do not have setups for all pairs and orderings of tasks, so we only demand that the setup interval variable following every task interval variable I_i^T will use a unit of resource of cumulative function. Again the sum must be lower or equal to the number of provided machine setters:

$$\sum_{I_i^S \in I^S} \textsc{Pulse}(I_i^S, 1) \leq w. \tag{CPF3}$$

Next, we synchronize the start and completion times between the tasks and setups. We use the constraint EndAtStart(I_A, I_B), which ensures that the interval variable I_A is completed exactly when the interval variable I_B starts. We set the end of the task equal to the start of the following setup:

$$\forall I_i^T \in I^T : \textsc{EndAtStart}(I_i^T, I_i^S). \tag{CPF4}$$

It is important to note that this is only possible because we set the task's length by inequality. As a result of this, the task interval variable ends exactly as the following setup starts. This is very important because the execution length of setup between tasks is already implicitly ensured by the setup times matrix passed to the NoOverlap constraint in the shared part of the model. No additional conditions are needed.

Now we only have to constraint the end of task setup. This can be done by using the constraint StartOfNext(I') which returns the start time of the next interval variable following I' in the sequence. In our case, this is the start time of the interval variable representing following task. As we do not know which task will be assigned to which machine, we have to use our optional interval variables introduced in (CP-C2). Thus, the constraints are as follows:

$$\forall I_i^S \in I^S, \forall M_k \in M : \textsc{EndOf}(I_i^S) \geq \textsc{StartOfNext}(I_{i,k}^{T^{Opt}}). \tag{CPF5}$$

Note that the inequality in constraint (CPF5) is necessary because StartOfNext evaluates 0 for absent $I_{i,k}^{T^{Opt}}$ and also for the last task on each machine. The start time of the setup is forced by condition EndAtStart to be after the end of the interval variable representing the task, which would never be 0, making the model unsolvable. This way, the last setup's completion time is equal to its start time, and the setup's length is 0.

The complete model using the defined constraints looks as follows:

Minimize	(CP-OBJ)
s.t.	
	(CP-C2) – (CP-C3)
	(CPF1) – (CPF5)

The $CP_{Flexible}$ model.

3.5 Experimental Results

The Table 1 demonstrates computational differences between $\mathsf{CP}_{Pairwise}$ and $\mathsf{CP}_{Flexible}$. While small instances can be solved by both models with similar effectivity, as the number of tasks grows, the number of optional setups in model $\mathsf{CP}_{Pairwise}$ becomes a problem. This leads to a sharp increase in makespans differences between models.

Table 1: Comparison of CP exact solutions.

	parameters			**objective value [-]**	
#	m	n	w	$\mathsf{CP}_{Pairwise}$	$\mathsf{CP}_{Flexible}$
1	6	18	5	92	**91**
2	6	36	5	1673	**175**
3	8	32	5	219	**128**
4	8	32	8	184	**126**
5	8	64	5	9996	**261**
6	8	64	8	6321	**257**
7	10	50	5	5901	**173**
8	10	50	8	3746	**164**
9	10	100	5	25146	**306**
10	10	100	8	15693	**299**
11	12	72	5	12706	**162**
12	12	72	8	7928	**164**
13	12	144	5	∞	**336**
14	12	144	8	32542	**338**

Every row represents one generated instance of m machines, n tasks, and w machine setters with the respective model's objective value. The calculation times given were between a couple of seconds to one minute. The detailed information about the implementation and hardware used for testing can be found in Section 8.

4 Constructive Heuristic Approaches

Since the considered problem is $\mathcal{NP}$-hard, it is infeasible to compute an optimal solution for many real-world instances. As shown in paper *Andrade et al.* [5], one of the possible solutions to such problems is to apply a heuristic approach, obtaining the suboptimal but still reasonable solution to the instances where other approaches would fail completely. Later in this section, two heuristic approaches to the problem are proposed:

- *Locally Optimal Selection of Setups* (LOSOS),
- *Resolution of Setup Overlaps Lazily* (ROSOL).

Compared to the CP models from the previous section, LOSOS and ROSOL bring considerable speed improvements. They can easily solve instances of multiple magnitudes larger than CP models. Another advantage over CP models is that they do not require a CP solver like *CP Optimizer* [11] used in this thesis. On the other hand, LOSOS and ROSOL do not guarantee the solution's optimality. However, they usually produce very good results.

Some parts of LOSOS and ROSOL are common to both algorithms. The main difference between them is that while LOSOS handles machine setters while scheduling tasks, ROSOL handles machine setters in a separate phase after task assignments and their orderings are already given.

In the pseudocode of both algorithms, there are three additional functions called, GenerateStartingTasks, SelectNextTask and OptimizeScheduleEnds. These functions are used to encapsulate parts of the algorithm, where improvements to the algorithms are applied. They are described in detail in Section 6. The pseudocodes of their base greedy versions are as follows:

Algorithm 1 Generate starting tasks, one for every machine.

```
1: function GenerateStartingTasks
2:     T^StartingTasks ← ∅
3:
4:     for each M_m ∈ M do
5:         T^StartingTasks ← T^StartingTasks ⋃ T_m
6:     end for
7:
8:     return T^StartingTasks
9: end function
```

Algorithm 2 Select next task to follow after a given task.

```
1: function SelectNextTask(Task_given, T^Remaining)
2:     T_next ← T_1^Remaining                ▷ T^Remaining denotes set of tasks remaining to be scheduled.
3:     MinimalSetupLength ← ∞
4:
5:     for each T_i ∈ T^Remaining do
6:         if o_given,i < MinimalSetupLength then        ▷ If setup time given by matrix is smaller.
7:             T_next ← T_i
8:             MinimalSetupLength ← o_given,i
9:         end if
10:    end for
11:
12:    return T_next, o_given,next
13: end function
```

Algorithm 3 Optimize ends of all machines schedules.

```
1: function OptimizeScheduleEnds
2:     SwapFound ← True
3:
4:     while SwapFound do
5:         SwapFound ← False
6:         M_longest ← M_1
7:         LongestMachineLength ← 0
8:
9:         for each M_m ∈ M do
10:            if LongestMachineLength < T_−1,m.End then      ▷ T_−1,m denotes last task on M_m.
11:                M_longest ← M_m
12:                LongestMachineLength ← T_−1,m.End      ▷ .End denotes end of task in schedule.
13:            end if
14:        end for
15:
16:        for each M_m ∈ M do
17:            M_m ← M_m ⋃ T_−1,longest                   ▷ Schedule T_−1,longest to M_m.
18:            if T_−1,m.End < T_−1,longest.End then
19:                M_longest ← M_longest \ T_−1,longest    ▷ Undo scheduling of T_−1,longest on M_longest.
20:                SwapFound ← True
21:            else
22:                M_m ← M_m \ T_−1,m                     ▷ Undo scheduling of T_−1,longest to M_m if ineffective.
23:            end if
24:        end for
25:    end while
26: end function
```

4.1 Locally Optimal Selection of Setups (LOSOS)

The main idea of LOSOS is to create chains of tasks on machines with the best possible setups between them. This is done by constructing the solution step by step, assigning the setup to the machine when the currently scheduled task execution on the machine ends. In the baseline version of the algorithm, the following setup is chosen greedily as the setup with the shortest execution time after the currently ending task. The pseudocode is given in Algorithm 4.

Algorithm 4 Solve instance using LOSOS algorithm.

```
1:  function LOSOS SOLVE
2:      T^StartingTasks ← GENERATESTARTINGTASKS                    ▷ called function
3:      for each M_m ∈ M do
4:          Schedule M_m ← T_m^StartingTasks
5:      end for
6:      T^Remaining ← T \ T^StartingTasks
7:
8:      WorkersFreeFromTime ← ∅
9:      for each W_w ∈ W do
10:         WorkersFreeFromTime ← WorkersFreeFromTime ⋃ 0
11:     end for
12:
13:     while T^Remaining ≠ ∅ do
14:         ClosestEndingMachine ← M_1
15:         ClosestEndingTask ← T_{-1,1}          ▷ Last currently scheduled task on M_1.
16:         for each M_m ∈ M do
17:             if T_{-1,m}.End < ClosestEndingTask.End then
18:                 ClosestEndingTask ← T_{-1,m}
19:                 ClosestEndingMachine ← M_m
20:             end if
21:         end for
22:         NextTask, SetupLength ← SELECTNEXTTASK(CLOSESTENDINGTASK, T^Remaining)
23:                                                                ▷ called function
24:         NextSetup.Start ← ClosestEndingTask.End
25:         NextTask.Start ← ClosestEndingTask.End + SetupLength
26:         ClosestEndingMachine ← ClosestEndingMachine ⋃ NextSetup
27:         ClosestEndingMachine ← ClosestEndingMachine ⋃ NextTask
28:                          ▷ Schedule NextSetup and NextTask to ClosestEndingMachine.
29:
30:         argmin(WorkersFreeFromTime) ← NextTask.Start
31:         T^Remaining ← T \ ClosestEndingTask
32:     end while
33:
34:     OPTIMIZESCHEDULEENDS                                       ▷ called function
35: end function
```

First, the starting tasks $T^{StartingTasks}$ are chosen for every machine and assigned to them. There are multiple possible criteria for choosing $T^{StartingTasks}$ which improve the solution besides the simple ineffective selection method described in Algorithm 1. These other methods are described in Section 6. After the first tasks are assigned, they are removed from the set of tasks remaining to be assigned, which is denoted by $T^{Remaining}$. ***(see lines 2 - 6)***

Next up, the data structure representing all machine setters is created. It is denoted by $WorkersFreeFromTime$ in the pseudocode. The best way is to use the priority queue with the earliest available machine setter on top so it can be easily accessed. At the start, all machine setters are set to be ready to work, available from time 0. ***(see lines 8 - 11)***

Now, the earliest ending task, denoted by $ClosestEndingTask$, with its respective machine, is found. The more effective way than the way proposed in the pseudocode is to use the priority queue again. Then, we find a suitable task, denoted by $NextTask$, and its respective setup length to follow after $ClosestEndingTask$. The $NextTask$ can be either chosen greedily as in Algorithm 2 or by one of the selection criteria described in Section 6. Using the greedy selection strategy, $NextTask$ is a task with the shortest possible setup from $ClosestEndingTask$ to any other unscheduled tasks. ***(see lines 13 - 22)***

Next up, the earliest ending machine setter is selected to perform the setup between $ClosestEndingTask$ and $NextTask$. The setup's starting time is calculated according to machine setter's and machine's availability and is scheduled to $ClosestEndingMachine$. After that $NextTask$ itself is scheduled. Now $NextTask$ is considered scheduled and removed from $T^{Remaining}$. ***(see lines 24 - 32)***

Finally, the end of the schedule is optimized by calling the Algorithm 3 or its better version proposed in Section 6. This is especially important if the Algorithm 2 selection strategy is used as it tends to leave long setups to the end. ***(see line 34)***

The asymptotic complexity of the baseline version of the algorithm with priority queues used for machine setters and machines is $\mathcal{O}(\log(w)\log(m)n^2 + mn)$.

4.2 Resolution of Setup Overlaps Lazily (ROSOL)

The main idea of ROSOL is to schedule tasks and setups in a locally optimal way and ignore machine setter overlaps, which are resolved later. In other words, we relax machine setter constraints and resolve them later by moving the start times of tasks and setups, if necessary. The pseudocode of ROSOL is in Algorithm 5. H denotes the upper bound of the final makespan length, and $Time_p$ denotes p th time point in the schedule.

The selection and assignment of $T^{StartingTasks}$ are the same as in LOSOS. ***(see lines 2 - 6)***

The selection of $ClosestEndingTask$, $ClosestEndingMachine$ and $NextTask$ are also the same as in LOSOS. The only difference here is that no data structure for machine setters is present. ***(see lines 8 - 17)***

Next, the $NextSetup$ and $NextTask$ are assigned. In this case, we do not consider machine setter availability as we did in LOSOS. ***(see lines 19 - 25)***

Now, the problem is solved but without adhering to the constraints of machine setters. The solution has to be modified, so machine setters are accounted for. So at every time point in the makespan, the number of currently executed setups is counted. ***(see lines 27 - 36)***

Algorithm 5 Solve instance using ROSOL algorithm.

```
1: function ROSOL SOLVE
2:     T^StartingTasks ← GENERATESTARTINGTASKS                    ▷ called function
3:     for each M_m ∈ M do
4:         Schedule M_m ← T_m^StartingTasks
5:     end for
6:     T^Remaining ← T \ T^StartingTasks
7:
8:     while T^Remaining ≠ ∅ do
9:         ClosestEndingMachine ← M_1
10:        ClosestEndingTask ← T_{-1,1}          ▷ Last currently scheduled task on M_1.
11:        for each M_m ∈ M do
12:            if T_{-1,m}.End < ClosestEndingTask.End then
13:                ClosestEndingTask ← T_{-1,m}
14:                ClosestEndingMachine ← M_m
15:            end if
16:        end for
17:        NextTask, SetupLength ← SELECTNEXTTASK(CLOSESTENDINGTASK, T^Remaining)
18:                                                               ▷ called function
19:        NextSetup.Start ← ClosestEndingTask.End
20:        NextTask.Start ← ClosestEndingTask.End + SetupLength
21:        ClosestEndingMachine ← ClosestEndingMachine ⋃ NextSetup
22:        ClosestEndingMachine ← ClosestEndingMachine ⋃ NextTask
23:                          ▷ Schedule NextSetup and NextTask to ClosestEndingMachine.
24:        T^Remaining ← T \ ClosestEndingTask
25:    end while
26:
27:    p ← 0
28:    while p ≠ H do
29:        CurrentSetupsCount ← 0
30:        for each M_m ∈ M do
31:            M^StartingSetups ← ∅          ▷ Set of machines with setup starting at this time point.
32:            if M_m in Time_p executes setup then
33:                CurrentSetupsCount ← CurrentSetupsCount + 1
34:                M^StartingSetups ← M^StartingSetups ⋃ M_m
35:            end if
36:        end for
37:
38:        if NumberOfWorkers < CurrentSetups then
39:            SetupsToMove ← CurrentSetups − NumberOfWorkers
40:            while SetupsToMove ≠ 0 do
41:                ClosestEndingMachine ← M_1^StartingTasks
42:                ClosestEndingTask ← T_{-1,1}
43:                for each M_m ∈ M do
44:                    if T_{-1,m}.End < ClosestEndingTask.End then
45:                        ClosestEndingTask ← T_{-1,m}
46:                        ClosestEndingMachine ← M_m
47:                    end if
48:                end for
49:                Move current ClosestEndingMachine setup one time point forward
50:                M^StartingTasks ← M^StartingTasks \ ClosestEndingMachine
51:                SetupsToMove ← SetupsToMove − 1
52:            end while
53:        end if
54:        p ← p + 1
55:    end while
56:
57:    OPTIMIZESCHEDULEENDS                                       ▷ called function
58: end function
```

If the number of concurrent setups running at a certain time point rises above the number of machine setters provided, it means that one or more setup starting at this time point are over the machine setter capacity. The start of some of these setups must be moved one time step forward, so a maximum of machine setters count of setups are performed. ***(see lines 38 - 39)***

The setup or setups selected to move are the ones on machines with the currently shortest schedule lengths. It is important to note that once the setup is set to be executed and not moved, it is considered immovable in the next time point. Otherwise, the algorithm could enter an infinite loop of setup moves. Machine's schedule length could become currently shortest while already executing some setup, which would interrupt it and move it forward. This could happen in a loop between multiple machines, with every setup being canceled mid execution. After executing the necessary setup moves, we move one time point forward in the schedule. ***(see lines 40 - 55)***

The end of the schedule is optimized in the same way as in LOSOS. ***(see line 57)***

The asymptotic complexity of the baseline version of the algorithm with priority queues used is $\mathcal{O}(\log(m)n^2 + Hm(m - w))$, where H is the length of the final makespan.

4.3 Experimental Results

The difference between LOSOS and ROSOL in Table 2 is not very pronounced. Particularly for the smaller instances, the results tend to be the same. When solving bigger instances, ROSOL slowly becomes better than LOSOS, but at the cost of increased computational cost as the complexity is not the same for both algorithms. This is an important takeaway for the next section, which describes the warm starting of CP models. With barely any result differences on instances where CP models are used and more taxing computation of ROSOL, LOSOS is more fitting to be a warm start algorithm of choice for CP models.

Table 2: Comparison of constructive heuristic algorithms.

	parameters			**objective value** [-]	
#	m	n	w	LOSOS	ROSOL
1	10	100	4	**295**	**295**
2	13	195	4	**447**	**447**
3	16	320	4	**572**	**572**
4	19	475	4	**734**	**734**
5	10	300	4	**823**	**823**
6	13	130	4	**302**	**302**
7	16	240	4	**438**	**438**
8	19	380	4	**553**	**553**
9	10	250	4	**697**	**697**
10	13	390	4	**810**	**810**
11	16	160	4	**326**	**326**
12	19	285	4	**440**	**440**
13	10	200	4	**578**	582
14	13	325	4	**722**	**722**
15	16	480	4	860	**856**
16	19	190	4	308	**307**
17	10	150	4	**460**	**460**
18	13	260	4	578	**577**
19	16	400	4	697	**692**
20	19	570	4	907	**900**

Every row represents one generated instance of m machines, n tasks, and w machine setters, with the respective objective value of the algorithm. The calculation times of algorithms on given instances were up to tens of milliseconds. The detailed information about the implementation and hardware used for testing can be found in Section 8.

5 Warm Starting the CP Solver

Since the solution space of the considered problem is large, it is worth considering the warm starting of the solution. Warm starting is a commonly used technique of obtaining a feasible starting solution in a reasonable time to improve the computational speed of the algorithm. It is often used in scheduling as in paper *Shahrzad et al.* [17] as well in combination with CP/MIP modeling as in paper *Fairbrother et al.* [7]. To warm start, we usually need to reduce our original problem to another more familiar one, to which some good and optimized algorithms already exist. One such suitable problem which we selected is the Vehicle Routing Problem (VRP). It is a very well-known and studied problem, with many existing algorithms and libraries providing solutions with good performances.

5.1 VRP Warm Start

The Vehicle Routing Problem, discussed in detail in the paper *Toth et al.* [26], has a very close resemblance to our considered problem. We build routes for vehicles with sequence-dependent crossings between each pair of points in the problem. The vehicles in VRP can represent our machines, points our tasks, and crossings our sequence-dependent setups. From the existing libraries, we chose Google's *OR-Tools* [8], which contains support for VRP solving and is freely available. Before we run the VRP solver, we reduced our problem to the VRP as follows:

1. The machines in the original problem are represented by vehicles in the reduced VRP. Therefore, the route of the vehicle represents the machine schedule.
2. The tasks in the original problem are represented by points on the routes of vehicles in the reduced VRP.
3. We add a new point which will represent *depot* of the VRP. The *depot* is a special point in VRP, where all vehicles must start and end their routes. We connect it by edges $(depot, point)$ to all points in the problem. The weights of these edges are equal to the processing times of tasks represented by those points.
4. We connect all points to the depot by edges $(point, depot)$. These edges' weights are equal to the sum of maximum setup time and maximum task processing time in the problem. This ensures that we never travel through the depot when going from one point to another. We have to do this because it would yield a solution that would be infeasible after reducing to our original problem.
5. All other points are connected with each other by edges $(point_a, point_b)$. The weight is equal to the sum of the task processing time represented by the $point_b$ and the setup length between them. Because setups are sequence-dependent we have asymmetric VRP, meaning that edge $(point_a, point_b)$ can have different length than $(point_b, point_a)$.
6. After OR-Tools solves the problem, we remove the depot in each vehicle route and map each route as a sequence of tasks on one machine.

It is important to note that this substitute problem does not consider that machine setter could be working multiple setups at once. Machine setters are not represented in the VRP at all. We have to assign the solution extracted from VRP to the CP model and let it complete it to the feasible starting solution. However, this solution still has a much smaller objective value compared to the starting solution of the CP model without a warm start. Using VRP warm start usually leads to better results in the same time limit, even though it uses part of the time limit provided for its execution.

5.2 LOSOS/ROSOL Warm Start Use

Because both LOSOS and ROSOL shown good and fast results, they were also considered as warm start candidates. It has been found that they are far more effective for warm starting than the VRP reduction. Because of that, they are used as warm start algorithms instead of VRP reduction.

The warm start of choice for $\text{CP}_{Pairwise}$ and $\text{CP}_{Flexible}$ is LOSOS because it is faster and there is no big difference between results[4], with ROSOL being usually only slightly better. However, both LOSOS and ROSOL are run as a warm start step in SMETI. There are two reasons why:

- There is a difference between the computational complexity of LOSOS *(polynomial)*, ROSOL *(pseudo-polynomial)*, and CP ($\mathcal{NP}$-hard). For smaller instances, CP models can improve their incumbent solution very fast with the benefit of systematically cutting down the solution space. On the other hand, with bigger instances, the nature of $\mathcal{NP}$-hard problems will make execution times of LOSOS and ROSOL negligible in perspective to the CP models runtimes.
- SMETI uses local improvements. This means that it is likely that most parts of the solution will not change very often. This is why it is so important in this case to have the best possible solution provided at the start.

After LOSOS or ROSOL warm start execution, their solution is then translated into the model of $\text{CP}_{Pairwise}$, $\text{CP}_{Flexible}$ or SMETI. This is done by assigning tasks and their orderings to the machines according to the warm start solution. We also add setup time spaces between those tasks and set their start and end times according to the warm start.

[4] This has been shown in Section 4.3.

5.3 Experimental Results

To distinguish the CP models with and without the warm start, we append *ws* behind the model's name when warm start is used. The Table 3 demonstrates the impact of warm start on both $\mathsf{CP}_{Pairwise}$ and $\mathsf{CP}_{Flexible}$. At first glance, three things are noticeable from the table. First, a warm start helps $\mathsf{CP}_{Pairwise}$ to find a reasonable solution in instances where it was not possible before. Second, a warm start solution is sometimes better than $\mathsf{CP}_{Pairwise}\mathsf{WS}$. Third, the warm start does not seem to affect $\mathsf{CP}_{Flexible}$ for small instances, sometimes even giving a slightly worse solution.

Table 3: Comparison of CP models with CP models using warm starting.

	parameters			**objective value [-]**				
#	m	n	w	$\mathsf{CP}_{Pairwise}$	$\mathsf{CP}_{Pairwise}\mathsf{WS}$	$\mathsf{CP}_{Flexible}$	$\mathsf{CP}_{Flexible}\mathsf{WS}$	LOSOS
1	6	18	5	92	**90**	91	94	123
2	6	36	5	1673	177	175	**168**	184
3	8	32	5	219	125	128	**120**	125
4	8	32	8	184	122	126	**120**	125
5	8	64	5	9996	268	261	**257**	263
6	8	64	8	6321	266	257	**253**	263
7	10	50	5	5901	171	173	**164**	167
8	10	50	8	3746	177	**164**	166	167
9	10	100	5	25146	310	306	**303**	317
10	10	100	8	15693	310	**299**	302	317
11	12	72	5	12706	171	162	**161**	184
12	12	72	8	7928	177	**164**	**164**	184
13	12	144	5	∞	48469	**336**	338	357
14	12	144	8	32542	31661	338	**336**	357

Every row represents one generated instance of m machines, n tasks, and w machine setters, with the respective objective value of CP models and the warm start objective value provided by LOSOS*. The calculation times given were between a couple of seconds and one minute. The detailed information about the implementation and hardware used for testing can be found in Section 8.*

The first observation is no surprise. However, the second observation is more interesting. The reason why the results of LOSOS are sometimes better than the results of $\mathsf{CP}_{Flexible}\mathsf{WS}$ is that warm start is not being translated to the model entirely. Some things, like the pulse function, are not encoded into the starting point of the CP model, so the CP solver has to take the warm start and complete it. In some cases, this takes time, and because $\mathsf{CP}_{Pairwise}$ model is generally not very effective, it can happen that the time provided is not enough to meet the warm start solution.

The third observation is also affected by this, but there are other reasons as well. We do not see any significant difference on smaller instances due to the $\mathsf{CP}_{Flexible}$ solution already being close to the optimum, where improvements are generally slow. Also, due to the nature of the CP solver search, when a solution is built from the warm start, the CP solver uses its makespan as a bound and expands the search around it. When no warm start is provided, the CP solver determines the starting point from which it works the solution space down. The solver may expand different branches of solutions in each scenario, sometimes favoring the no warm start approach. To prove that warm start also makes the difference for the $\mathsf{CP}_{Flexible}$ model, we have to consider bigger problem instances or reduce the time limit. With bigger instances, the warm start bound is more efficient as it cuts bigger parts of the solution space. In the next tests, we decreased the calculation times to a maximum of a couple of seconds and increased the instance sizes to underline the warm start effectivity.

Table 4: Comparison of the warm started and non warm started $\mathsf{CP}_{Flexible}$ model.

	parameters			**objective value [-]**		
#	m	n	w	$\mathsf{CP}_{Flexible}$	$\mathsf{CP}_{Flexible}\mathsf{WS}$	LOSOS
1	14	98	5	2015	**204**	218
2	14	98	8	484	**205**	218
3	14	98	13	844	**205**	218
4	14	196	5	4507	**368**	379
5	14	196	8	3357	**371**	379
6	14	196	13	4074	**371**	379
7	16	128	5	2921	**233**	250
8	16	128	8	476	**231**	250
9	16	128	13	932	**231**	250
10	16	256	5	6203	**451**	500
11	16	256	8	6017	**449**	500
12	16	256	13	4697	**446**	500
13	18	162	5	4052	**272**	291
14	18	162	8	2392	**270**	291
15	18	162	13	1818	**271**	291
16	18	324	5	7985	**523**	567
17	18	324	8	4656	**523**	565
18	18	324	13	5551	**518**	565

We can see that in most cases, $\mathsf{CP}_{Flexible}\mathsf{WS}$ solution is magnitudes better than the $\mathsf{CP}_{Flexible}$. Depending on the instance size, time limit, and chance, there is a moment where the CP solver expands the solution space closer to the optimum, and the makespan radically drops. However, this happened only in a few of the testing instances in Table 4. Hence, a warm start helps us get better results faster. It also ensures that our solution will not be way off even if we tackle big instances or have a small time limit available, which is arguably its most important benefit.

6 Constructive Heuristics Improvements

Algorithms LOSOS and ROSOL are straightforward and efficient in most cases, but they are greedy heuristics. As shown for even less complex $\mathcal{NP}$ Integer Knapsack Problem in paper *Kohli et al.* [13], bounds and results of certain instances for simple greedy heuristics can be very far from the optimal solution to the problem. In this section, we focus on how to improve the effectivity, especially in situations where the greedy nature of the heuristic algorithms incur substantial losses to the solution quality. We propose these improvements:

- Selecting starting tasks in an informed way.
- Optimizing the endings of machine schedules with additional task swapping.
- Replacing the greedy setup selection with coefficient based selection.

LOSOS can be further improved using task selection driven by the reduction of setup overlaps. This is not applicable to ROSOL because machine setters are added into the problem when all the tasks are already assigned to machines.

6.1 Starting Tasks Selection

Picking first m tasks in the problem to be the machines' starting tasks, as it is done in the baseline function GenerateStartingTasks 1 in Section 4, will unlikely lead to a good result. There is no guaranteed way to pick the best possible starting tasks without solving the whole problem, but we can choose tasks, so they probably yield a better result than the naive selection. The proposed approaches are:

1. We set starting tasks at random. This option is only mentioned because it is not the same as the naive selection of m first tasks, but on average, it yields the same results, and it is the worst out of the proposed methods.
2. We find m tasks, such that the shortest possible setup from any previous task to these is one of the m longest setups between all shortest setups of all pairs of tasks. Let us define the set of these tasks as T^m, denote task from this set as T_j and any other task as T_i. We are looking for T_j such that the shortest possible setup between any T_i and T_j is one of the m longest setups between all shortest setups of all possible pairs of tasks. In other words, we are looking for m tasks with their $\max(\min(o_{i,j}))$ being one of the m biggest in the whole problem instance. This way, we will avoid tasks that would require long setups even when placed after their best compatible preceding task. Since only setup times have variable length depending on the sequence of tasks, the testing results show a considerable improvement to the solution length.
3. We find m tasks with minimal processing time, so machine setters can start working as soon as possible. This way, machine setters will have the possibility to be present over a greater percentage of the makespan, hopefully preventing future delays. The testing results also show significant improvement over the random start task selection.

4. We treat the tasks as they would be dedicated to machines. If problem instance has 3 machines and 15 tasks, tasks T_1 to T_5 would be assigned to M_1, tasks T_6 to T_{10} to M_2 and so on. After that, the second or third selection rule is applied to every machine-task subset. This can be useful if we assume that the instance was already pre-processed or solved, and the task assignments were made efficiently. An example of this can be a solved real-world scenario, which we want to improve further.

Both method 2 and method 3 provide very good results, but the final method used in the LOSOS and ROSOL implementations is the method 2. It provides more stable results than the method 3, even though in the best-case scenario, the method 3 often provides better improvements to the solution makespan.

6.2 End of Schedule Optimization

From the solutions examined, it has been observed that the most problematic part of the schedule is its end. The remaining tasks have only a few possible options where and when to be scheduled with their best fit preceding tasks usually already used elsewhere. The best fit preceding task is a task where the setup between the preceding task and the following is the smallest of all possible preceding tasks. Therefore, it is crucial to re-optimize the end of the schedule to minimize this as much as possible. Improving is split into two phases:

1. First, we find the machine with the longest schedule denoted $M_{longest}$. We take the last task and check if the task can be moved to another machine while decreasing the makespan. We do that by checking every machine and calculating its schedule length after the task is appended to its schedule. If the resulting schedule length is smaller than the $M_{longest}$ before the task was moved, we move the task to that machine. If multiple machines meet this condition, one with the shortest resulting schedule length is selected. Now we check if $M_{longest}$ still has the longest schedule. If not, we find the machine currently having the longest schedule and mark it as $M_{longest}$. We repeat this step until no machine meets the condition for task swapping. This is already described in function OptimizeScheduleEnds 3 in Section 4.

2. Second, we switch ending tasks between pairs of machines to achieve further reduction of the makespan. The pairs of machines are chosen in a way that the makespan decreases after the switch. This means that both machines have a shorter schedule than the length of the longer machine's schedule before the swapping of tasks. If no machine pair can switch the ending tasks in a way that the makespan decreases, we conclude the end of schedule optimization.

6.3 Coefficient of Task Selection Priority

In baseline versions of both LOSOS and ROSOL, when choosing the next task to follow the currently ending one, the algorithms make a choice greedily based on the setup length minimization. This generally works well, but it has a fundamental flaw. As we deplete tasks with small setup times, it can happen that tasks with only long setups remaining will stay until the end and cause huge setbacks to the solution makespan. In other words, instead of taking minor setbacks to obtain a globally optimal solution, a locally optimal sequence is chosen, hurting the objective value of the solution. The problem is further amplified because as long setups stay until the end of the schedule, they cause a shortage of available machine setters. If long setups were to be dispersed throughout the solution against the shorter ones, the problem would not occur. Hence, it is often much better to take a locally suboptimal solution to obtain shorter setup times at the end of the schedule.

Example 1: Let us imagine an example problem. We have tasks T_1 and T_2 scheduled on M_1 and M_2 respectively. Tasks T_3 and T_4 remain to be scheduled with setups $o_{1,3} = 1$, $o_{1,4} = 2$, $o_{2,3} = 30$ and $o_{2,4} = 100$ between pairs of tasks indexed by numbers. For simplicity, we do not consider setups between T_3 and T_4 because placing them both on one machine would be suboptimal anyway.

The first task to end is T_1. If we choose greedily, we schedule T_3 after T_1 because $o_{1,3} < o_{1,4}$. However when task T_2 ends, we will have to execute setup $o_{2,4} = 100$, last remaining possibility.

On the other hand, if we were not to pick the task greedily and pick T_4 instead to follow after task T_1, we would execute setup $o_{1,4} = 2$. Then after task T_2 ends, we would schedule task T_3, executing setup $o_{2,3} = 30$. In most scenarios, this would produce a solution with a shorter makespan. So, we are looking for a way of predicting which tasks should be saved for later and which tasks are to be scheduled now to avoid similar situations.

To achieve this, we propose an evaluation function that produces a coefficient, determining with which priority we want to schedule which task at a given moment. We need to consider more than just the best setup time, going from the currently ending task to the task we choose to follow. We weigh in multiple best setup times from unfinished (both scheduled and unscheduled) tasks to the task we consider for scheduling. The coefficient is calculated as a function of those setup times. In other words, when choosing the next task, we consider where else it could be used and how efficiently. Multiple variations of this function were tested, while the polynomial function provided below has shown the best results on average.

$$Coefficient = o_{i,j}^4 + |o_{i,j} - o_{x,j}| \cdot (o_{i,j} - o_{x,j}) + |o_{i,j} - o_{x2,j}| \cdot (o_{i,j} - o_{x2,j}) + |o_{i,j} - o_{x3,j}|$$

T_i is the currently ending scheduled task, while T_j is the task considered to be scheduled next. $o_{x,j}$ denotes the smallest setup time achievable from any T_x to T_j, $o_{x2,j}$ denotes the second smallest achievable setup, and $o_{x3,j}$ the third smallest one.

This way, the largest emphasis is still on the following setup between the current and next task, but other future scheduling possibilities are considered. If we get back to the example

above and calculate the coefficient for it, we get the following results.

$$Coefficient_{T_1,T_3} = 1 + |(1-1)\cdot(1-1)| + |(1-30)\cdot(1-30)| + 0 = -840$$

$$Coefficient_{T_1,T_4} = 16 + |(2-2)\cdot(2-2)| + |(2-100)\cdot(2-100)| + 0 = -9588$$

Note that we have zeros as the last element in both equations because only two tasks are remaining. As before, we do not consider T_3 and T_4 to have a possible setup between them, as it would likely not matter anyway. The smaller the coefficient is, the more priority the task has to be used now, so T_4 would be scheduled after T_1. We can think of the coefficient value as a penalty; the lower it is, the better.

Example 2: Let us give one more example. We have four tasks, T_5 to T_8, with setups $o_{5,7} = 10$, $o_{6,7} = 20$, $o_{5,8} = 15$, and $o_{6,8} = 17$. T_5 is currently ending and T_6 is still being executed on another machine. The calculation of coefficients would look as follows.

$$Coefficient_{T_5,T_7} = 10000 + |(10-10)\cdot(10-10)| + |(10-20)\cdot(10-20)| - 0 = 9900$$

$$Coefficient_{T_5,T_8} = 50625 + |(15-15)\cdot(15-15)| + |(15-17)\cdot(15-17)| - 0 = 50621$$

$Coeft_{T_5,T_7} < Coef_{T_5,T_8}$ so T_7 would be picked. In this case, T_7 is also a locally optimal choice.

Illustration of the coefficient benefit can be seen in figures 3 and 4 on the next page. Both executions were on the same problem instance and without any other improvement techniques used to isolate the coefficient's effect. The first graph shows the solution produced without the use of the coefficient, while the second graph shows the solution while using the coefficient. We can see that while setups in the first half of the first graph might be slightly shorter than in the second one, almost every setup is costly at the end of the schedule. On the other hand, in the second graph where the coefficient was used, setups at the end of the schedule are mostly reduced, and the coefficient solution yields a 9 % decrease in objective value.

However, the effect of the coefficient is not always as profound as it is in the given graphs as it is largely dependent on the distribution of the setup times. If setups are quite balanced, it has little to no effect. However, if huge differences between setups are present, then the coefficient is more useful, i.e., multiple groups of similar products in the real-world scenario. Depending on the problem domain, the coefficient equation best suited for the concrete problem can differ. It is also important that coefficient use comes with a computational cost, which is usually an order of magnitude more expensive than execution without it. On the other hand, the execution time is still negligible when compared to CP models.

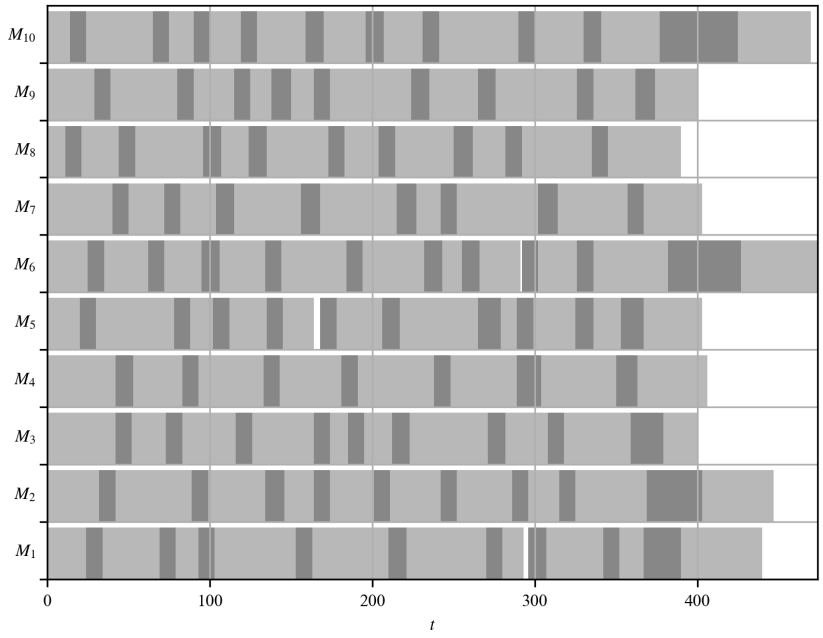


Figure 3: Instance of 10 machines, 100 tasks and 5 machine setters. Execution without coefficient took 3 ms, producing solution with makespan 474.

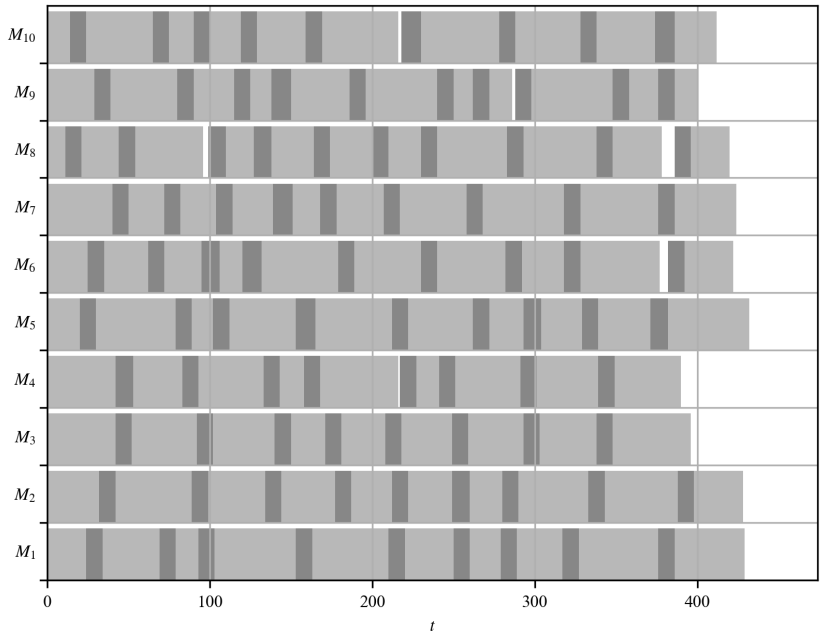


Figure 4: Instance of 10 machines, 100 tasks and 5 machine setters. Execution with coefficient took 508 ms, producing solution with makespan 432.

6.4 LOSOS: Setup Overlap Reduction

We can further upgrade the solution of LOSOS by taking into consideration machine setter availability. Before the explanation, we define the following notation:

- T_i denotes the currently ending task in the schedule.
- T_j denotes the next earliest ending task in the schedule after T_i.
- W_l denotes the only available machine setter at the moment when the setup of the task following T_i is being scheduled. This is only relevant in situations where only one last machine setter is free.
- W_s denotes the earliest ending machine setter in the schedule except for W_l.
- $o_{i,x}$ denotes any possible setup from the task T_i to all the tasks not scheduled yet.
- $o_{j,x}$ denotes any possible setup from the task T_j to all the tasks not scheduled yet.

Now, let us consider two following issues with the current way of scheduling:

1. If we assign too long $o_{i,x}$ to W_l, it might happen that when T_j ends, there will be no machine setter available to start executing $o_{j,x}$. Therefore, the machine would idle instead of executing another setup.
2. If we assign too short $o_{i,x}$ to W_l, after $o_{i,x}$ is finished, there might be no machine requiring machine setter, because all machines would still execute other tasks and setups. Then, W_l would idle instead of possibly executing other longer $o_{i,x}$, which might have to be executed later, at a higher cost to the makespan. If we know that we have a long time available to execute setup, because no other machine requires machine setter, we can execute longer setup, maximizing usage of the free window and possibly saving task with shorter setups for later, getting the task with longer setups out of the way.

To address the issue number 1, we have to consider if T_j ends before W_s is available. If not, meaning W_s will be available before T_j is completed, we do not have to do anything, machine setter will be available for the execution of $o_{j,x}$. However, if T_j ends before W_s is available, the machine where T_j is scheduled would idle. So, we calculate the time difference between ends of T_j and T_i, $c_j - c_i = t_1$. Now, we know that we can assign setup $o_{i,x} \leq t_1$ to W_l after T_i finishes, so W_l will finish the selected $o_{i,x}$ in time to execute $o_{j,x}$ when T_j finishes. Obviously, this might not always be possible. If it is not, we will take the smallest possible $o_{i,x}$ to minimize the time the machine will idle. To illustrate, an example is provided.

We have two tasks T_1 and T_2 with $c_1 = 10$ and $c_2 = 15$ being executed. We have 2 machine setters, one available and one busy until $Time = 17$. When T_1 ends at $Time = 10$, we have 2 possible following setups, $o_{1,3} = 4$ and $o_{1,4} = 10$. If we would use the coefficient described in Subsection6.3, the values would be $Coefficient_{T_1,T_3} = 900$ and $Coefficient_{T_1,T_4} = 400$. According to the coefficients, we would pick setup $o_{1,4}$ to follow after T_1. But we want to consult the new condition to see if no machine will be left idle. $c_2 - c_1 = 5$ so if we used

$o_{1,4}$, when the T_2 ends at $Time = 15$, we would have no machine setter available for the next setup. So we select $o_{1,3}$ instead.

Now, to address the issue number 2, we again calculate the difference $c_j - c_i = t_2$. However, now we will take the size of this difference and recalculate the task coefficient described in Section 6.3 using the value t_2. Instead of using the length of setup between current and following task as a first member of the coefficient equation, we will use $max(0, o_{i,x} - t_2)$. This way, we only emphasize the portion of the setup time, which is over the size of t_2. This makes the impact of other members of the coefficient function more important. To illustrate, an example is provided.

We have three tasks T_1, T_2 and T_3 with $c_1 = 100$, $c_2 = 150$ and $c_3 = 200$ being executed. The possible setups are $o_{1,4} = 20$, $o_{1,5} = 50$, $o_{2,4} = 5$, $o_{2,5} = 60$, $o_{3,4} = 40$ and $o_{3,5} = 70$. In this case, we assume we have enough available machine setters. Now, T_1 ends at $Time = 100$ and the calculated coefficients are as follows.

$$Coefficient_{T_1,T_4} = max(0,(20-50))^4 + |(20-5)| \cdot (20-5) + |(20-40)| \cdot (20-40) + 0 = -175$$

$$Coefficient_{T_1,T_5} = max(0,(50-50))^4 + |(50-60)| \cdot (50-60) + |(50-70)| \cdot (50-70) + 0 = -500$$

We pick T_5 to follow after T_1 according to these new coefficient equations. If we would not alter the equation, T_4 would be calculated as a more suitable candidate instead, which would be less effective as T_4 is more suitable to follow after task T_2. Also the saved difference between $o_{2,4} = 5$ and $o_{2,5} = 60$ is bigger than difference between $o_{1,4} = 20$ and $o_{1,5} = 50$ and because we know we will not need machine setter any time soon, we are guaranteed that this does not creates a shortage of machine setters.

The important thing to note is that the condition 1 is more important than the condition 2. We rather let machine setter wait while machines work because, in the end, we care about the makespan, which is affected by the machine schedule length. We can also see that the improvement of the solution is not guaranteed. If the coefficients were too far apart, and we chose according to the first condition, we might be worse off. It should be possible to derive some trade-off thresholds when it is worthy of using these conditions and when not, but this was not further examined in this thesis.

In conclusion, this improved task selection helps us avoid or reduce idle times of machines and machine setters. However, it does not guarantee that it always improve the solution quality. As it relies on using machine setters in the planning phase, it is not possible to use it with ROSOL.

6.5 Experimental Results

To test the effectiveness of the improvements proposed, LOSOS algorithm was run on multiple instances using different improvements. The results can be seen in Table 5. The starting tasks were always selected according to the method 2 proposed in Subsection 6.1. While optimization of the schedule ends proposed in Subsection 6.2, denoted as LOSOS$_{End}$, can never degrade the original solution, use of the priority coefficient proposed in Subsection 6.3, denoted as LOSOS$_{Coef}$, and setup overlap reduction proposed in Subsection 6.4, denoted as LOSOS$_{Setup}$, can produce worse solution than the one produced without them. We can see from the table that the improvement with the greatest effect is the priority coefficient. Using the priority coefficient, together with setup overlap reduction, can also further improve the solution.

Table 5: Comparison of LOSOS without and with improvements.

	parameters			**objective value [-]**				
#	m	n	w	LOSOS	LOSOS$_{End}$	LOSOS$_{Setup}$	LOSOS$_{Coef}$	LOSOS$_{Setup+Coef}$
1	10	100	2	305	303	**287**	301	301
2	10	100	5	**295**	**295**	**295**	301	301
3	10	122	2	370	**365**	384	366	372
4	10	122	5	378	**358**	378	366	366
5	10	144	2	437	421	408	**390**	**390**
6	10	144	5	411	404	411	**388**	**388**
7	12	100	2	284	284	253	**233**	234
8	12	100	5	287	269	287	**239**	**239**
9	12	122	2	329	315	306	306	**298**
10	12	122	5	315	313	315	**306**	**306**
11	12	144	2	371	371	345	347	**333**
12	12	144	5	356	**341**	356	347	347
$\sum$	-	-	-	4138	4045	4025	3890	**3875**

Every row represents one generated instance of m machines, n tasks, and w machine setters with the respective objective values of LOSOS executions. The detailed information about the implementation and hardware used for testing can be found in Section 8.

From the graph below, it is clearly visible that the priority coefficient is not only the most effective but also the most expensive improvement. We can also see that setup overlap reduction cost can increase rapidly if a small number of machine setters is available. This is because the smaller the number of available machine setters is, the bigger the number of conflicts that must be computed is. In conclusion, the ideal improvement to use can vary depending on the situation. If we are solving a very large instance with a very small time limit, it might be wise not to use the priority coefficient and instead combine the end of the schedule optimization with setup overlap reduction. If we also have a very small number of available machine setters in the problem, it might also be good not to use the setup overlap reduction because its cost can increase rapidly. However, as the most expensive execution among the testing examples took only 1 second, in most cases, the time limit is not an issue.

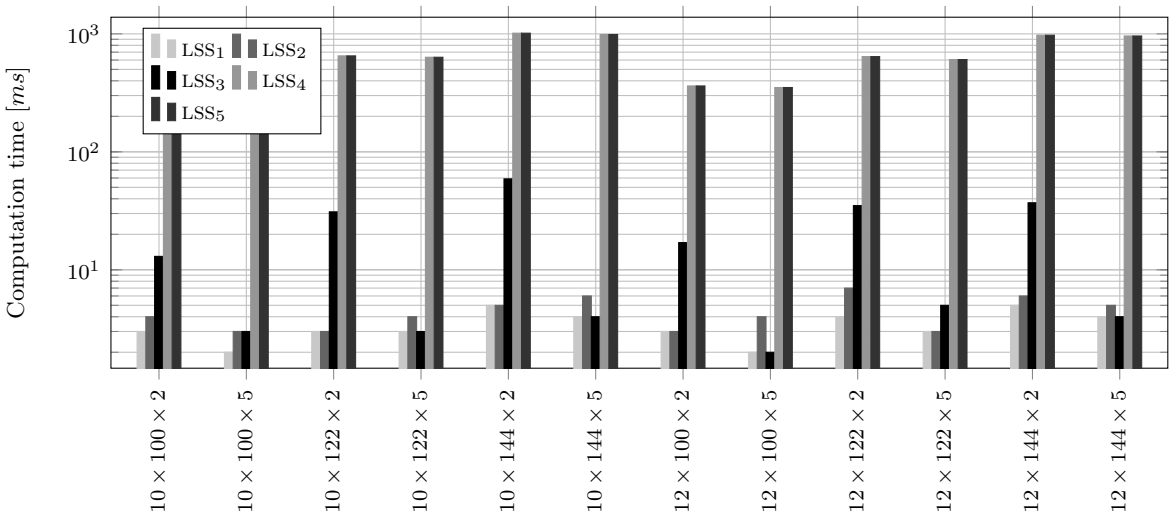


Figure 5: Computation times of LOSOS executions.

Every grouping of bars represents an instance of m machines, n tasks, and w machine setters. The height of the bar represents the length of the execution of one LOSOS execution in the logarithmic scale. The numbering of bars corresponds to the ordering of LOSOS executions in the Table 5.

7 Subproblem Modeling Evolving Through Iteration (SMETI)

In the previous sections, we proposed CP models and greedy heuristic algorithms. CP models showed proficiency when solving small instances, while heuristic algorithms excelled when solving very large ones. However, we would like to have an approach that would be suited for the remainder of instances in between. This is why we propose a hybrid algorithm called *Subproblem Modeling Evolving Through Iteration* (SMETI).

SMETI is a large neighborhood search algorithm [21]. This means we repeatedly transform current solution into a different one, trying to minimize its objective value. Because the obtained solution is a result of the previous solution transformation, we call it being in previous solution's neighborhood. SMETI repeatedly transforms the solution using various steps of conditional swaps and CP solver optimizations. This technique was recently applied to a very similar problem of Clustered Vehicle-Routing Problem by *Hintsch et al.* [10] and proved to be promising.

The first step of SMETI is obtaining a feasible solution to the problem. This can be provided by any means mentioned in the previous sections, but the most effective way is to obtain the starting solution using LOSOS and ROSOL. After running both LOSOS and ROSOL, the solution with a shorter makespan is selected as the starting point. Then SMETI constructs the model according to the starting solution while leaving some parts of the solution unconstrained. This allows the CP solver to re-optimize it. The lower number of constraints necessary to solve the subproblem enables the CP solver to handle much larger problems than exact CP models, locally optimizing parts of them. When one part of the problem is solved into a local optimum, we move to the next one and repeat this until the time limit runs out or all defined options are depleted. If the latter is true and there is still time left, the best solution found so far is passed as a warm start solution to the $\mathsf{CP}_{Flexible}$.

7.1 Algorithm Description

The key question is how to construct the subproblems to bring efficient improvements to the starting solution; in other words, how to define the search neighborhood. The most logical aspect to focus on first is the machine with the longest schedule, denoted by $M_{longest}$, which is the critical path [2] of the problem. The entire problem's makespan is equal to the schedule length of $M_{longest}$, so unless we change its length, other changes will not affect the objective value.

We use this observation when assigning the tasks to the machines according to the starting solution. On all machines but the $M_{longest}$, we fix the task orderings according to the starting solution. However, on the $M_{longest}$ we leave the order of tasks unconstrained. This way, when the CP solver is started, the tasks on $M_{longest}$ can change their order given by the starting solution, thus potentially improving the schedule length of $M_{longest}$. If the schedule length of $M_{longest}$ is improved, the whole solution's makespan is improved as well. If, after re-optimization, another machine becomes the new $M_{longest}$, we repeat the same process for that machine. After this phase can no longer improve the solution, we know that we cannot improve the solution using only one machine re-optimization.

After the first phase finishes, to improve the solution further, we need to start swapping tasks between pairs of machines. There are multiple ways of swapping tasks, and they are described in the step by step explanation below. Every pair of selected machines contains $M_{longest}$ as we need to reduce its schedule length to improve the solution. These swapping methods are cyclically repeated until one whole swapping cycle brings no further improvement to the solution. If there is time left, we use the best-found solution and pass it as a warm start solution to the $\mathsf{CP}_{Flexible}$.

Before we formally describe the steps of SMETI, we need to define the notation used. We define two subsets of machines, one without $M_{longest}$ and one without both $M_{longest}$ and without $M_{shortest}$, which denotes the machine with the shortest schedule length:

$$M^L = M \setminus M_{longest}, \tag{1}$$

$$M^{L,S} = M^L \setminus M_{shortest}. \tag{2}$$

We will denote the set of tasks in the starting solution as T^{WS} and the set of tasks in the CP model as T^{CP}. $T^{WS}_{a,k}$ will represent task of order a on machine k in the starting solution and $T^{CP}_{a,k}$ will represent task of order a on machine k in the CP model. The indexes i and j will denote the task's orders within the whole instance, which is the same as in previous sections.

The steps of SMETI are as follows:

1. Execute LOSOS and ROSOL and set the better solution as the starting solution for SMETI.
2. Set tasks to machines according to the starting solution. We use $I^{T^{Opt}}_{i,k}$ as used for example in constraint (CP-C2) to represent the *optional interval variable* of a task assigned to a certain machine in CP model. Notice that the index a is translated to the task's index i, indexing the task within the whole instance. This is because we need the index to be compatible with the indexing used by existing variables in the model:

$$\forall M_k \in M^L,\ \forall T^{WS}_{a,k} \in T^{WS}_k : \textsc{PresenceOf}\left\{I^{T^{Opt}}_{i,k}\right\}. \tag{3}$$

   We could also set all the other optional interval variables representing tasks on machines to absent, but no pronounced effect on the solution speed was observed.

3. Set the tasks' orderings according to the starting solution on every machine except the $M_{longest}$. The orderings of tasks are enforced by setting the setup between two scheduled following tasks on the machine to be present. This, in turn, gives the ordering of the whole task set for the machine. We denote setup between tasks $T^{WS}_{a,k}$ and $T^{WS}_{a+1,k}$ as $I^{S^{Opt}}_{i,j}$. Notice that the indexes a and $a+1$ are translated to the task's indexes i and j within the whole instance to be compatible with the existing variables in the model:

$$\forall M_k \in M^L,\ \forall T^{WS}_{a,k}, T^{WS}_{a+1,k} \in T^{WS}_k : \textsc{PresenceOf}\left\{I^{S^{Opt}}_{i,j}\right\}. \tag{4}$$

   The machine setters are not fixed according to the starting solution because their assignments can change thanks to the re-optimization of $M_{longest}$. The rest of the model is

the same as in $\mathsf{CP}_{Pairwise}$, but since there is not a quadratic number of optional setups between all possible tasks in the problem and most of the model's conditions are fixed, $\mathsf{CP}_{Pairwise}$ has no problem with effectivity.

4. Execute the CP solver. The model gets optimized and $M_{longest}$, which in our case is the critical path, gets re-optimized according to the constraints. It is important to note that the retrieved solution can be far from the original problem's optimal solution, yet any improvement in this phase brings improvement to the whole makespan.
5. Find the machine with the longest schedule and check if it is the same machine as the $M_{longest}$.
   (a) If $M_{longest}$ is the same as the machine with the currently longest schedule, we proceed to the next step because we know that the $M_{longest}$ is already locally optimal. By re-optimizing it, one would get the same result.
   (b) If the machine with the currently longest schedule is a different machine from $M_{longest}$, it implies that the previous $M_{longest}$ was improved and is no longer the critical path. We set the machine with the currently longest schedule as new $M_{longest}$ and go back to step 2 and rebuild the model. However, before doing so, we update the starting solution's orderings according to the result obtained by the CP solver.
6. Not being able to further improve the solution by changing tasks ordering on one machine, SMETI starts switching tasks between machine pairs. We find $M_{longest}$ and $M_{shortest}$ in the solution. $M_{longest}$ must be changed to improve the solution to the original problem, and $M_{shortest}$ is heuristically the best candidate for absorbing longer tasks and setups from $M_{longest}$. If $M_{shortest}$ would become new $M_{longest}$ after re-optimization, it will be re-optimized again in the next step, so choosing $M_{shortest}$ should not create a problem. This time we only set tasks and orderings to machines in $M^{L,S}$:

$$\forall M_k \in M^{L,S},\ \forall T^{WS}_{a,k} \in T^{WS}_k : \textsc{PresenceOf}\left\{I^{T^{Opt}}_{i,k}\right\}, \tag{5}$$

$$\forall M_k \in M^{L,S},\ \forall T^{WS}_{a,k}, T^{WS}_{a+1,k} \in T^{WS}_k : \textsc{PresenceOf}\left\{I^{S^{Opt}}_{i,j}\right\}. \tag{6}$$

7. Swap tasks between $M_{longest}$ and $M_{shortest}$ according to the current swapping criterion. There are three swapping criteria to have a better chance of avoiding getting stuck in a local minimum. These criteria are cyclically changed in the run time after the third criterion comes the first again. We change to another swapping criterion when we get the same $M_{shortest}$ and $M_{longest}$ twice in a row for the current swapping criterion selected. The criteria are as follows:
   (a) Swap the task with maximum processing time on $M_{longest}$ with the task with minimum processing time on $M_{shortest}$. The goal of this swap is to shorten the critical path as much as possible regardless of the setup lengths:

$$T^{CP}_{shortest,M_{shortest}} = T^{WS}_{longest,M_{longest}}, \tag{7}$$

$$T^{CP}_{longest,M_{longest}} = T^{WS}_{shortest,M_{shortest}}. \tag{8}$$

(b) Swap the task with maximum processing time on $M_{longest}$ with the task on $M_{shortest}$, which will make the difference between $M_{shortest}$ and $M_{longest}$ machine schedule lengths as small as possible. The task is denoted by $T_{balanced}$. When the difference is computed, the $M_{shortest}$'s schedule length and the length of $T_{balanced}$ are considered, but not the setup lengths. This is because we do not know after which task on $M_{longest}$ will $T_{balanced}$ be placed; hence, we do not know the preceding's setup length. The goal of this swap is to balance machines' schedule lengths and also introduce variability into the cycle:

$$T^{CP}_{balanced,M_{shortest}} = T^{WS}_{longest,M_{longest}}, \tag{9}$$

$$T^{CP}_{longest,M_{longest}} = T^{WS}_{balanced,M_{shortest}}. \tag{10}$$

(c) Swap task with the longest preceding setup time on $M_{longest}$ with the task on $M_{shortest}$ with shortest average setup time to tasks on $M_{longest}$. The task selected on $M_{longest}$ is denoted by $T_{incompatible}$ while the selected task on $M_{shortest}$ is denoted by $T_{compatible}$. The goal of this swap is to reduce unnecessary setup times on $M_{longest}$ by exchanging the task with a long preceding setup for a task with more compatible setups to the tasks on $M_{longest}$:

$$T^{CP}_{compatible,M_{shortest}} = T^{WS}_{incompatible,M_{longest}}, \tag{11}$$

$$T^{CP}_{incompatible,M_{longest}} = T^{WS}_{compatible,M_{shortest}}. \tag{12}$$

8. Execute the CP solver and update the solution. We repeat the steps 6 to 8 in a loop while updating the current solution as we did in the first phase between steps 2 and 5. Note that this updated solution becomes a starting point for the next step in the cycle. Machines not involved in swapping have fixed orders. If we cycled through all swapping criteria, and in the whole cycle, we did not improve the solution, we escape the cycle and save the best solution so far. We also break from the cycle if the time for finding the solution ran out.

9. If the time limit was still not reached, we take the best-obtained solution so far and set it as a warm start for the exact $\mathsf{CP}_{Flexible}$ model so it can be further optimized before the time runs out.

Example 3: Let us demonstrate the execution of SMETI on an example. To keep the example succinct, we consider that the re-optimizations affect only the currently selected machine or machines in each step. In actual execution, other machines can be affected as well because machine setters availability changes, thus the start of setups on non-selected machines can change. The machine's schedule length will be denoted L_k in the following text.

Consider three machines with their schedule lengths in the starting solution to be $L_1 = 500$, $L_2 = 540$ and $L_3 = 530$. In the first phase of the algorithm, we execute the loop between steps 2 and 5. In step 2, M_2 is picked as our $M_{longest}$ and after re-optimization its schedule length decreases to $L_2 = 510$. In a real-world scenario, this could happen by removing long suboptimal setup at the end of the schedule caused by the greediness of warm start approach. Now M_3 has the longest schedule so it becomes $M_{longest}$. After re-optimization its schedule length decreases to $L_3 = 520$. Because M_3 remained the $M_{longest}$, we continue to step 6 since no possible improvement is achievable by repeating cycle between steps 2 and 5. After first part of the SMETI algorithm our machine schedule lengths are $L_1 = 500$, $L_2 = 510$ and $L_3 = 520$.

Now we execute the loop between steps 6 and 8. $M_{longest}$ is still M_3 and the $M_{shortest}$ is M_1. By swapping the longest task from $M_{longest}$ for the shortest task from $M_{shortest}$ as described in 7a, the schedule length of $L_3 = 515$ has slightly improved, and the schedule length of $L_1 = 508$ slightly deteriorated. In this case, the swapping also increased the overall sum of all setup times as the deterioration was larger than the improvement. However, this is still okay because the makespan of the whole solution was lowered. Now we would reapply the 7a step even though the $M_{shortest}$ and $M_{longest}$ stayed the same because we only swapped one pair of tasks. We skip this for the sake of simplicity and move to the next swapping method 7b.

The selected task from $M_{longest}$ is still the longest one, but the task from $M_{shortest}$ is selected as follows. We denote the subtraction of the schedule length of $M_{shortest}$ from the schedule length of $M_{longest}$ as r_1. We denote the subtraction of the selected task length from $M_{shortest}$ from the selected task length from $M_{longest}$ as r_2. We are looking for such a task from $M_{shortest}$, which minimizes the difference between r_1 and r_2, so hopefully, after the swap, both machines' schedule lengths are as close as possible. However, this is not entirely guaranteed because setup times before and after the swapped tasks on their new respective machines can play a part. Since they are not known before the CP solver execution, we cannot take them into account. After re-optimization of our example the schedule lengths are $L_1 = 511$, $L_3 = 513$ and $L_2 = 510$. As before, we would reapply this step, but for simplicity, we move directly to 7c.

The $M_{shortest}$ is now M_2, the $M_{longest}$ is still M_3. Now we want to select the "best compatible" task for $M_{longest}$ from $M_{shortest}$. This is achieved by checking all of the tasks on $M_{shortest}$ and finding the one which has the shortest average setup between tasks from $M_{longest}$ and itself. From $M_{longest}$, we select the task with the longest setup preceding it for swapping. After the swap we get solution $L_1 = 511$, $L_2 = 512$ and $L_3 = 507$. Again, we would repeat this step, but we skip it for the sake of simplicity.

After finishing the cycle, we would now repeat it whole from 7a to 7c again. We would keep repeating it as long as in any step of each cycle there was some improvement found. For now, we assume that no further improvements were found, thus $L_1 = 511$, $L_2 = 512$ and $L_3 = 507$ is our final solution. It is important to note that we always remember the best solution found so far, so if the solution somehow deteriorated over the course of execution, we would use the best solution as a result or as a warm start for the $\mathsf{CP}_{Flexible}$ if the time limit was not exhausted.

7.2 Experimental Results

The difference between LOSOS and ROSOL was described in Subsection 4.3, so in the following comparisons, we will refer to them generally as constructive heuristics. The conclusion drawn from the Table 6 is that SMETI noticeably improves the starting solution provided by constructive heuristics. Also, unlike constructive heuristics, if enough time is given, SMETI is guaranteed to achieve the optimal solution using $\text{CP}_{Flexible}$. Even though the improvements are usually only a few percent, they have a great impact because the starting solution is usually close to the optimum. In real-world scenarios, the last percents are generally the most important ones, especially if looking for an optimization of the existing schedule.

In the following comparison, the time limit was set to 60 seconds to limit or completely eliminate the last step of SMETI, which is $\text{CP}_{Flexible}$, to isolate only the improvements acquired through the heuristic subproblem modeling of SMETI. It shows that in a scenario where the CP model alone cannot be utilized because it is not time efficient or the model is too complex to be built in the given time, SMETI can provide an improved solution over the constructive heuristics. In total, 60 tests were run; 20 of them are shown in the table.

Table 6: Comparison of heuristic algorithms.

	parameters			**objective value** [-]	
#	m	n	w	SMETI	MIN(LOSOS, ROSOL)
1	10	100	4	**272**	295
2	13	195	4	**417**	447
3	16	320	4	**555**	572
4	19	475	4	**734**	**734**
5	10	300	4	**808**	823
6	13	130	4	**291**	302
7	16	240	4	**435**	438
8	19	380	4	**537**	553
9	10	250	4	**677**	697
10	13	390	4	**804**	810
11	16	160	4	**294**	326
12	19	285	4	**426**	440
13	10	200	4	**577**	582
14	13	325	4	**722**	**722**
15	16	480	4	**847**	856
16	19	190	4	**277**	307
17	10	150	4	**460**	**460**
18	13	260	4	**572**	577
19	16	400	4	**692**	**692**
20	19	570	4	**900**	**900**

Every row represents instance of m machines, n tasks, and w machine setters with the objective value of SMETI and the result of a better performing heuristic. The detailed information about the implementation and hardware used for testing can be found in Section 8.

In 14 of all 60 tested instances, the better of the constructive heuristics was able to achieve the same result as SMETI. The complete sums of objective values from the testing are:

1. LOSOS: 36338 (+4.4 % compared to SMETI on average case),
2. ROSOL: 36184 (+4 % compared to SMETI on average case),
3. SMETI: 35128.

The most notable differences between results of constructive heuristics compared to SMETI were seen on instances with a small number of machine setters available, where the efficiency of constructive heuristic approaches is lessened, and the use of CP subproblem modeling steps negate that. With the increasing size of the instance, the differences between approaches were smaller and smaller as the time limit was often insufficient to execute all heuristic steps of SMETI. To obtain more favorable results for SMETI, we would have to scale time with the instance's size. On the other hand, the point of this comparison is to show all advantages and shortcomings and the near-real-time (in the context of planning) solution search. According to the 60 tests conducted, instances with a small amount of machine setters computed by constructive heuristics provide 5-15 % worse results than SMETI. When more machine setters are available, using SMETI changes the value usually only in terms of single percents. A correlation between the minimum and maximum setup time and result difference between constructive heuristic algorithms and SMETI was also observed. When the gap between minimal and maximal setup length was increased, the improvement provided by SMETI also increased by an additional 2% on average.

8 Experimental Evaluation

All experimental results in sections 3 to 7 were obtained on a single core of Intel i7-6700 processor running at 3.4GHz with the maximum allowed RAM usage of 12GB. However, the usual RAM usage of the CP solver was under 2GB. Only the largest problem instances surpassed this threshold. Usage of RAM for heuristic approaches was insignificant. The tasks' and setups' processing times for experimental testing in previous chapters were drawn from uniform distributions of (1 to 50) or (25 to 50) time units per task or setup.

Comparisons to the other existing solutions in this section were executed on a single core of Intel Xeon 4110 processor running at 2.1GHz. Regarding the fairness of comparison, if the paper describing the other existing solution used a CPU with a lower clock rate to measure the results and their computed values and bounds were used for the comparisons, we readjusted the time limits accordingly. A graphical card was not used in any of the calculations. Algorithms were implemented using C++. For CP, *ILOG CP Optimizer 12.09* [11] was used. For ILP, *Gurobi 9.0* [9] was used.

8.1 Comparison of CP Models and Heuristic Approaches

The comparison between exact CP models and heuristic algorithms is virtually impossible because the suitable problem instances to be solved by each approach differ fundamentally. Even for the instances where the CP models barely provide any solution in an hour of execution, heuristic algorithms would usually provide a solution in under a second, also of better quality than the CP solution obtained. Beyond these instances, the model is too complex even to be built in a reasonable time, which shows the importance of subproblem modeling in SMETI. On the other hand, given small enough instances, the heuristic algorithm cannot leverage the time limit given to optimize the solution beyond the proposed improvements in Section 6. Meanwhile, CP models use this time to get to the optimal solution. This conclusion underlines the importance of combining various approaches in the hybrid algorithm SMETI to obtain the best approach for any situation.

8.2 Comparison of Heuristic Approaches to the Optimum

As it was discussed in previous sections, the machine setters count has pronounced influence on the relative quality of constructive heuristics solutions to the optimum. When using SMETI, this negative effect is partially mitigated by CP subproblem modeling and, if enough time is provided, completely eliminated by $\mathsf{CP}_{Flexible}$. However, when using only a constructive heuristic algorithm, the partial mitigation achieved by using the improvements in Section 6 is not as effective as in SMETI. While an instance of 15 machines, 150 tasks, and 1 machine setter solved by constructive heuristics would generally have a very suboptimal solution, consider Figure 6 of LOSOS algorithm solving instance with 150 machines, 15000 tasks, and 10 machine setters. The ratio of tasks being executed to the number of machine setters is the same in both examples, but the uneven nature of setup lengths with the probability multiple tasks end simultaneously is smoothed over. In the latter example, we obtain a result with the objective

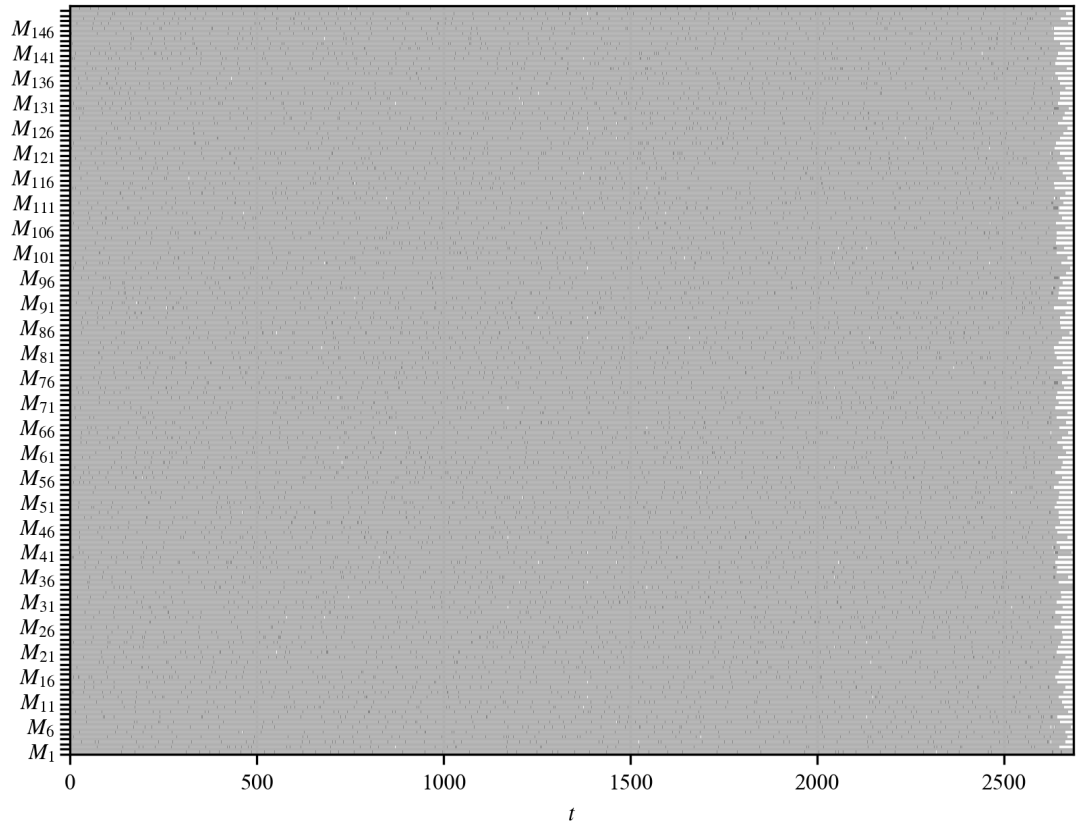


Figure 6: Instance of 150 machines, 15000 tasks and 10 machine setters.
Execution took 35 s, producing solution with makespan 2686.

value of 2686 in 35-second runtime, which, considering the instance's size, is very fast. Also, if we look at the graph, the makespan is very efficiently packed without any noticeable waiting times.

Furthermore, we can show that the result is very likely to be very close to the optimum. In this specific example, tasks and setups are between 1 and 50 time units long, which yields an average of 25.5 per task/setup. If only tasks were considered, they would create a schedule of roughly 2550 units long on every machine. The same would apply for setups, but we can choose setups optimally to achieve much better results. If we would consider a hypothetical scenario where it would be possible to pick the minimal length setup every single time for all scheduled pairs of tasks, we would add an extra 99 time units to the schedule's length of every machine. This would result in the makespan of 2649 time units. We would also have to have zero overlap of machine setters. The difference between this calculated lower bound of 2649 and the actual result of 2686 is only 1.3%. Now, if we consider the size of the SMETI improvements over the results of LOSOS and ROSOL in Section 7, it makes them even more significant as the results of LOSOS and ROSOL and already very close to the optimum.

8.3 Comparison to Existing Solutions

To evaluate whether approaches proposed in this thesis are of state-of-the-art quality, we took existing approaches to similar problems and compared them to ours. We chose 3 papers, each solving a subset of problem that our approaches can solve. The chosen papers are:

1. **Parallel dedicated machine scheduling problem with sequence-dependent setups and a single server** [24] *(abbreviated further by the authors' names as HCZ)*
   - Tasks are dedicated to machines.
   - Only 1 machine setter is available.
   - We encoded the described MIP model using *Gurobi 9.0*. We generated the instances according to the paper's distributions and ran both MIP and CP models. For the CP models, we had to convert the instances so that they would have the same optimal solution as if the tasks were dedicated to machines. That was achieved by setting infinite length setup times between tasks that are not supposed to be on the same machine. In CP models, the infinite setup time was represented by $IloIntervalMax$ value. This way, if the CP solver would schedule two tasks, which were dedicated to two different machines on the same machine, the resulting schedule would be infinitely long, thus infeasible.
   - Paper proposes a way to calculate lower bound (LB) to which their Genetic Algorithm (GA) approach effectivity is compared. Since we could not obtain the testing data from the authors of the paper, we calculated LB on our own instances according to the method described. We compared the percentage difference between LB and our SMETI to their percentage difference between LB and GA, obtaining a fair comparison. Execution times for the instances were used according to the paper.
2. **MIP models and hybrid algorithm for minimizing the makespan of parallel machines scheduling problem with a single server** [4] *(abbreviated further by the authors' names and the version of the proposed compared MIP model as KL2.)*
   - Setups are sequence-independent, and the setup must be executed before any task regardless of what task, if any, were executed beforehand.
   - Only 1 machine setter is available.
   - Sequence independence and the necessity of a setup preceding even the first task on every machine was simulated by changes to the setup matrix, forcing CP solver to search for feasible solutions in the domain of the original problem. MIP model was encoded using *Gurobi 9.0*, and the results were compared to the results from CP solver on the same generated instances.
   - Paper compares their GA solution to the solutions acquired from MIP executed for 3600 seconds. We obtained those results when comparing their MIP model to our CP model, so we used these results to compare GA against SMETI by comparing the results of SMETI to the MIP results.

3. **Modeling and solving static m identical parallel machines scheduling problem with a common server and sequence-dependent setup times** [18] *(abbreviated further by the authors' names as HY)*

    - Only 1 machine setter is available.
    - MIP model was encoded using *Gurobi 9.0* and compared to CP solver on the same generated instances.
    - Paper compares their GA solution mostly to other techniques with unknown testing sets. Unlike the two previous papers, we were not able to find a fair comparison method. We attempted to implement GA according to the paper's description. On average, there is a noticeable difference between results obtained by our and their original implementation regarding makespan values of instances drawn from the same distributions. SMETI was executed on the same instances as their GA re-implementation.

Abbreviations shared across all comparison tables are m for machine count, n/m for task count per one machine, and n for the number of tasks in the whole problem summed over all machines. ∞ in the table means that there was no feasible solution was found for the selected approach. It is worth noting that for the first two compared approaches, even though the instances are adjusted, so the results are the same, our approaches still have to search in bigger solution space. The adjusted instances have the same complexity as if they were instances of our considered problem.

8.3.1 HCZ

There were no results provided for the MIP model in the paper, only a statement that it performs poorly. We encoded the model and tested it against $\mathsf{CP}_{Flexible}$ without the warm start, so only the model's performance is measured. The results show that $\mathsf{CP}_{Flexible}$ performed much better even though it was handicapped by solving more general problem. $\mathsf{CP}_{Flexible}$ was also able to solve instances that the MIP model was not able to. To get a better picture of the difference, results for all tested instances are provided in Table 7. The models were given a 3600-second time limit.

Table 7: HCZ MIP model results compared to $\mathsf{CP}_{Flexible}$ results.

	parameters		**objective value [-]**		**CPU time [s]**	
#	m	n/m	MIP[24]	$\mathsf{CP}_{Flexible}$	MIP[24]	$\mathsf{CP}_{Flexible}$
1	2	4	**329**	**329**	0.194	**0.006**
2	2	4	**351**	**351**	0.132	**0.004**
3	2	4	**394**	**394**	0.152	**0.004**
4	2	6	**403**	**403**	2.847	**0.009**
5	2	6	**423**	**423**	4.31	**0.013**
6	2	6	**508**	**508**	5.818	**0.016**
7	2	8	**469**	**469**	3600.0	**0.01**
8	2	8	**489**	**489**	1094.538	**0.022**
9	2	8	628	**627**	3600.0	**9.465**
10	3	6	**419**	**419**	13.242	**0.011**
11	3	6	**432**	**432**	112.127	**0.019**
12	3	6	540	**539**	3600.0	**11.143**
13	3	9	686	**673**	3600.0	**0.061**
14	3	9	724	**693**	3600.0	**0.033**
15	3	9	929	**837**	3600.0	**21.53**
16	3	12	∞	**743**	3600.0	**143.366**
17	3	12	∞	**767**	3600.0	**150.809**
18	3	12	1477	**1053**	**3600.0**	**3600.0**
19	4	8	701	**552**	3600.0	**5.443**
20	4	8	∞	**582**	3600.0	**9.873**
21	4	8	1162	**831**	**3600.0**	**3600.0**
22	4	12	∞	**768**	3600.0	**123.216**
23	4	12	∞	**813**	3600.0	**294.951**
24	4	12	∞	**1277**	**3600.0**	**3600.0**
25	4	16	∞	**1078**	3600.0	**0.635**
26	4	16	∞	**1097**	3600.0	**1137.193**
27	4	16	∞	**1687**	**3600.0**	**3600.0**

For the heuristic approaches comparison, we calculated LB described in the paper on instances generated according to the paper provided parameters. Then we executed $\mathsf{CP}_{Flexible}$WS on these instances and compared the results with the paper results. The reason why we used $\mathsf{CP}_{Flexible}$ instead of SMETI in this comparison is that SMETI heuristic steps utilize heavily the swapping of tasks between machines, which is prohibited in this scenario. We also had to limit the functionality of LOSOS warm start, which made it less effective. The objective values of LB and $\mathsf{CP}_{Flexible}$, as well as the gaps between LB and respective approaches, can be seen in the Table 8.

Table 8: HCZ GA effectivity compared to CP$_{Flexible}$WS effectivity.

	parameters				objective value [-]		LB2 gap [-]	
#	m	n/m	Setup LB	Setup UB	LB2	CP$_{Flexible}$WS	GA[24]	CP$_{Flexible}$WS
1	2	5	5	25	380	380	**0.00**	**0.0**
2	2	5	5	50	390	390	0.47	**0.0**
3	2	5	25	50	463	463	0.43	**0.0**
4	2	10	5	25	671	671	0.46	**0.0**
5	2	10	5	50	691	691	1.10	**0.0**
6	2	10	25	50	855	855	1.08	**0.0**
7	3	5	5	25	380	380	0.04	**0.0**
8	3	5	5	50	390	390	0.42	**0.0**
9	3	5	25	50	463	463	1.56	**0.0**
10	3	10	5	25	689	689	0.57	**0.0**
11	3	10	5	50	717	717	1.51	**0.0**
12	3	10	25	50	874	925	**1.82**	5.84
13	5	5	5	25	380	380	0.09	**0.0**
14	5	5	5	50	390	390	1.94	**0.0**
15	5	5	25	50	552	599	**2.03**	8.51
16	5	10	5	25	689	689	0.69	**0.0**
17	5	10	5	50	717	736	**2.14**	2.65
18	5	10	25	50	1225	1323	**5.48**	8.0
19	7	5	5	25	380	380	0.48	**0.0**
20	7	5	5	50	390	399	4.38	**2.31**
21	7	5	25	50	769	837	**1.89**	8.84
22	7	10	5	25	689	702	**1.18**	1.89
23	7	10	5	50	717	858	**6.37**	19.67
24	7	10	25	50	1683	1837	**5.94**	9.15
25	10	5	5	25	402	410	2.36	**1.99**
26	10	5	5	50	418	591	**6.99**	41.39
27	10	5	25	50	1112	1195	**2.31**	7.46
28	10	10	5	25	689	815	**3.81**	18.29
29	10	10	5	50	717	1102	**22.98**	53.7
30	10	10	25	50	2391	2628	**6.87**	9.91

In the same time limit, out of 30 cases, CP$_{Flexible}$WS provided a better solution than the GA in 16 and worse in 13 of them. CP$_{Flexible}$WS excelled at solving smaller instances, but solutions of bigger instances were worse. This is because we could not utilize SMETI and the CP$_{Flexible}$WS model solves more general problem than the GA. In bigger instances, it took a long time to build the model and narrow it down, even with the warm start. Also, the altered warm start provided was not as good. The very short time limit is also inconvenient for CP models. The biggest tested instance had only a 30-second time limit. When we use a 50-second time limit instead, we obtain results on par or better than the GA's. This is also an important upside of using the CP$_{Flexible}$WS. If more time is given, we get steadily better results, which is not guaranteed by the GA. In conclusion, the comparison shows that even when tackling a quite different problem with a less suitable proposed approach, it still produces good results.

8.3.2 KL2

The results table of the MIP model was provided in the paper; however, it was not very useful for further comparison as it gave little to no detail about single executions and objective values. Also, as in the previous paper comparison, we did not have access to the original instances used. Therefore we implemented the MIP model using *Gurobi*. We generated instances according to the methodology described in the paper and executed both the MIP model and $\text{CP}_{Flexible}$ on those instances. The Table 9 shows the comparison for three types of instances, instances of 6 machines with either 20, 30, or 40 tasks, with two additional settings, α and p. α represents the amount of variance in task and setup times, 0.1 meaning a maximum of 10% deviation from average on either side. p describes a multiplier of the setup length; the bigger it is, the longer the setups can be. The table's style is adapted from the original paper because of its compactness for this type of testing data. The MIP-2 in the table stands for the original paper's best performing MIP model. The results show that $\text{CP}_{Flexible}$ always returned the same or better solution than the MIP model, essentially superseding it. The models were given a 3600-second time limit.

Table 9: KL2 MIP-2 model results compared to $\text{CP}_{Flexible}$ results.

parameters		**(20, 6)**		**(30, 6)**		**(40, 6)**	
α	p	MIP-2[4]	$\text{CP}_{Flexible}$	MIP-2[4]	$\text{CP}_{Flexible}$	MIP-2[4]	$\text{CP}_{Flexible}$
0.1	0.5	**205**	**205**	283	**279**	392	**370**
0.1	0.7	**210**	**210**	291	**288**	392	**380**
0.1	1.0	**225**	**225**	322	**318**	428	**412**
0.3	0.5	188	**185**	284	**274**	390	**366**
0.3	0.7	195	**192**	289	**282**	396	**375**
0.3	1.0	**212**	**212**	320	**307**	418	**402**
0.5	0.5	190	**186**	288	**279**	394	**373**
0.5	0.7	193	**189**	293	**282**	394	**377**
0.5	1.0	214	**212**	322	**311**	434	**407**

To evaluate the efficiency of their GA, the authors compared the GA results to the results of their MIP models running for 3600 seconds. We obtained these results when comparing the MIP-2 to the $\text{CP}_{Flexible}$, so we used them again to compare them to the SMETI results. If there is any difference between computational power of their and our hardware used, it is irrelevant because the MIP-2 results and the SMETI results were obtained on the same hardware. Thus if our machine would be faster, the obtained MIP-2 results to which we compare our SMETI results would also profit from the same increase in computational power. The results are shown in Table 10.

Table 10: KL2 GA effectivity compared to SMETI effectivity.

	parameters				objective value [-]		difference [%]	
#	m	n	p	α	MIP-2[4]	SMETI	GA[4]/MIP-2[4]	SMETI/MIP-2[4]
1	6	20	0.5	0.1	205	205	1.87	**0.0**
2	6	20	0.7	0.1	210	210	2.34	**0.0**
3	6	20	1.0	0.1	225	225	2.72	**0.0**
4	6	20	0.5	0.3	188	186	1.06	**-1.06**
5	6	20	0.7	0.3	195	194	2.15	**-0.51**
6	6	20	1.0	0.3	212	216	2.72	**1.89**
7	6	20	0.5	0.5	190	188	3.15	**-1.05**
8	6	20	0.7	0.5	193	192	2.02	**-0.52**
9	6	20	1.0	0.5	214	219	4.03	**2.34**
10	6	30	0.5	0.1	283	281	1.24	**-0.71**
11	6	30	0.7	0.1	291	291	2.21	**0.0**
12	6	30	1.0	0.1	322	320	1.74	**-0.62**
13	6	30	0.5	0.3	284	275	1.29	**-3.17**
14	6	30	0.7	0.3	289	283	2.65	**-2.08**
15	6	30	1.0	0.3	320	310	2.42	**-3.12**
16	6	30	0.5	0.5	288	280	2.68	**-2.78**
17	6	30	0.7	0.5	293	286	2.9	**-2.39**
18	6	30	1.0	0.5	322	315	2.7	**-2.17**
19	6	40	0.5	0.1	392	373	2.22	**-4.85**
20	6	40	0.7	0.1	392	383	3.55	**-2.3**
21	6	40	1.0	0.1	428	414	2.05	**-3.27**
22	6	40	0.5	0.3	390	368	1.4	**-5.64**
23	6	40	0.7	0.3	396	377	2.7	**-4.8**
24	6	40	1.0	0.3	418	409	4.66	**-2.15**
25	6	40	0.5	0.5	394	376	0.91	**-4.57**
26	6	40	0.7	0.5	394	380	2.36	**-3.55**
27	6	40	1.0	0.5	434	416	3.86	**-4.15**

In 25 out of 30 instances, SMETI with a time limit of 60 seconds provided a better solution than the MIP model with a time limit of 3600 seconds. Only in 1 case, the solution was worse. SMETI also provided a better solution than the GA in the same time limit in every single instance. SMETI can also continue and find the optimal solution if enough time is given.

8.3.3 HY

We had no access to the original instances, so we generated our own according to the paper's description. We changed the original time limit from 18000 seconds to 3600 seconds to obtain results in a more reasonable time. This is applied for the MIP model as well as for our $\mathsf{CP}_{Flexible}$. It can be clearly seen from the Table 11 that $\mathsf{CP}_{Flexible}$ was always on par or better than the MIP model. In fact, in cases where results were the same, we know that $\mathsf{CP}_{Flexible}$ reached the optimal solution; hence the results could not be different. $\mathsf{CP}_{Flexible}$ was also able to prove optimality in some instances where the MIP model did not even reach the optimal solution, let alone proved its optimality.

Table 11: HY MIP model results compared to $\mathsf{CP}_{Flexible}$ results.

	parameters		**objective value [-]**		**CPU time [s]**	
#	m	n	MIP[18]	$\mathsf{CP}_{Flexible}$	MIP[18]	$\mathsf{CP}_{Flexible}$
1	2	6	**200**	**200**	1.6	**0.1**
2	2	8	**237**	**237**	161.5	**0.4**
3	2	10	337	**334**	3600.0	**1.1**
4	2	14	459	**415**	**3600.0**	**3600.0**
5	2	20	∞	**615**	**3600.0**	**3600.0**
6	3	9	**188**	**188**	291.5	**0.7**
7	3	12	265	**253**	3600.0	**8.9**
8	3	15	394	**346**	**3600.0**	**3600.0**
9	3	21	469	**373**	**3600.0**	**3600.0**
10	3	30	∞	**670**	**3600.0**	**3600.0**
11	4	12	199	**196**	3600.0	**15.3**
12	4	16	259	**199**	**3600.0**	**3600.0**
13	4	20	∞	**309**	**3600.0**	**3600.0**
14	4	28	∞	**446**	**3600.0**	**3600.0**
15	4	40	∞	**653**	**3600.0**	**3600.0**
16	5	15	250	**209**	**3600.0**	**3600.0**
17	5	20	428	**247**	**3600.0**	**3600.0**
18	5	25	∞	**301**	**3600.0**	**3600.0**
19	5	35	∞	**457**	**3600.0**	**3600.0**
20	5	50	∞	**670**	**3600.0**	**3600.0**
21	7	21	∞	**165**	**3600.0**	**3600.0**
22	7	28	∞	**262**	**3600.0**	**3600.0**
23	7	35	∞	**334**	**3600.0**	**3600.0**
24	7	49	∞	**430**	**3600.0**	**3600.0**
25	7	70	∞	**650**	**3600.0**	**3600.0**
26	10	30	∞	**211**	**3600.0**	**3600.0**
27	10	40	∞	**290**	**3600.0**	**3600.0**
28	10	50	∞	**349**	**3600.0**	**3600.0**
29	10	70	∞	**465**	**3600.0**	**3600.0**
30	10	100	∞	**672**	**3600.0**	**3600.0**

The authors evaluated the GA effectivity by comparing it to other proposed approaches. However, most of the instances had no results for the MIP model, so we had no way of measuring the GA performance without re-implementing it too. We did our best, but there were ambiguous places in the description where we were not ultimately sure. We were unsuccessful in reaching the authors for further clarification. While MIP models are unambiguous, the results of this comparison might be skewed thanks to a possible misunderstanding of the description. Every machine-task combination was run 20 times to even out the GA's random nature to obtain a fairer comparison. The results in Table 12 show SMETI outperforming GA in every case except one. The last column shows the ratio between SMETI and GA objective value. It is noticeable that with the growing instance's size, the difference gets bigger. This is probably the result of GA's inability to find improvements in such a huge solution space effectively.

Table 12: HY GA results compared to SMETI results.

	parameters		**objective value [-]**		**ratio [-]**
#	m	n	SMETI	GA reimpl[18]	SMETI/GA reimpl[18]
1	2	6	**201**	203	0.99
2	2	8	**255**	260	0.98
3	2	10	**332**	340	0.98
4	2	14	**440**	**440**	1.00
5	2	20	**641**	643	1.00
6	3	9	**198**	210	0.94
7	3	12	**262**	284	0.92
8	3	15	**325**	342	0.95
9	3	21	**443**	478	0.93
10	3	30	**628**	681	0.92
11	4	12	**198**	225	0.88
12	4	16	**245**	278	0.88
13	4	20	**325**	364	0.89
14	4	28	**447**	509	0.88
15	4	40	**637**	728	0.88
16	5	15	**198**	226	0.88
17	5	20	**264**	309	0.85
18	5	25	**322**	384	0.84
19	5	35	**437**	519	0.84
20	5	50	**652**	782	0.83
21	7	21	**199**	230	0.87
22	7	28	**267**	319	0.84
23	7	35	**322**	388	0.83
24	7	49	**446**	549	0.81
25	7	70	**639**	770	0.83
26	10	30	**207**	278	0.74
27	10	40	**280**	345	0.81
28	10	50	**353**	443	0.80
29	10	70	**501**	581	0.86
30	10	100	**691**	824	0.84

9 Conclusion and Future Work

This thesis addresses non-overlapping scheduling of tasks to non-dedicated machines with sequence-dependent setups executed by a given number of machine setters. This means we have a given number of tasks and machines, and any task can be scheduled for any machine. Before every task except the first one on every machine, machine setter has to execute a setup operation whose length depends on both the task before the setup and the task following it. There is no limitation on the quantity of machine setters in the problem.

Initially, Constraint Programming (CP) models were proposed to formalize and optimally solve the problem. However, these are generally suitable to solve only small instances. To extend the usefulness of CP models, we proposed warm start techniques that greatly increased CP models' ability to solve larger instances. With a warm start, $\mathsf{CP}_{Flexible}$ can successfully solve instances of tens of machines and hundreds of tasks. However, this still might not be enough to accommodate any real-world scenario of the considered problem. For even larger instances, heuristic approaches are proposed. These provide a very fast way to obtain the solution to instances of hundreds of machines and tens of thousands of tasks, effectively solving any conceivable real-world scenario in under a minute. We also proposed improvements to these methods, enhancing the solution's quality in a trade-off with time.

To achieve the best possible result for a larger specter of problem instances, we proposed a multi-step hybrid algorithm called SMETI, which combines heuristic approaches, CP sub-problem modeling, and the CP exact model $\mathsf{CP}_{Flexible}$. SMETI provides a way of solving both smaller instances in an optimal way and huge instances very well, thus accommodating any real-world scenario of the considered problem.

All proposed approaches and their respective parts were measured for efficiency against each other and against the existing state-of-the-art approaches. Since, to the best of our knowledge, there is no paper tackling exactly the same problem, we had to compare our solutions to other less general ones. We achieved this by instance transformations to obtain the problem with the same optimal solution as the original one but solvable by our approaches. However, this means that our algorithms still had to search in larger solution spaces because the transformed instance is of the same size as would be an instance of the same input parameters for our more general problem. Despite this slightly favored the other existing approaches in the comparison, our CP models proved to be better than existing MIP models and our SMETI algorithm to prevail over other heuristic approaches. SMETI also has one additional advantage over the heuristic approaches in other papers because it can solve instances of any size up to the optimum if enough time is given.

In conclusion, we proved that our approaches are effective, relevant, and can be used for more specific problems with better efficiency than other existing approaches. We consider this and the fact that we bring a solution to a completely new, more general problem the key contributions of this thesis and see a great potential of using provided algorithms in real-world production scenarios. Also, extensions and further optimizations were considered and will be looked into and further expanded in the future.

References

[1] Amir H. Abdekhodaee and Andrew Wirth. Scheduling parallel machines with a single server: some solvable cases and heuristics. *Computers & Operations Research*, 29(3):295–315, 2002.

[2] Ihtisham Ali and Susmit Bagchi. Isolating critical flow path and algorithmic partitioning of the and/or mobile workflow graph. *Future Generation Computer Systems*, 103:28 – 43, 2020.

[3] Ali Allahverdi, CT Ng, TC Edwin Cheng, and Mikhail Y Kovalyov. A survey of scheduling problems with setup times or costs. *European journal of operational research*, 187(3):985–1032, 2008.

[4] Gokalp Yildiz Alper Hamzadayi. Modeling and solving static m identical parallel machines scheduling problem with a common server and sequence dependent setup times. *Computers & Industrial Engineering*, 2011.

[5] Carlos E. Andrade, Shabbir Ahmed, George L. Nemhauser, and Yufen Shao. A hybrid primal heuristic for finding feasible solutions to mixed integer programs. *European Journal of Operational Research*, 263(1):62 – 71, 2017.

[6] Dong Chen, Peter B Luh, Lakshman S Thakur, and Jack Moreno Jr. Optimization-based manufacturing scheduling with multiple resources, setup requirements, and transfer lots. *IIE Transactions*, 35(10):973–985, 2003.

[7] Jamie Fairbrother and Konstantinos G. Zografos. Optimal scheduling of slots with season segmentation. *European Journal of Operational Research*, 2020.

[8] Google's OR-Tools. `https://developers.google.com/optimization/`. Accessed September, 2020.

[9] Gurobi. `https://www.gurobi.com//`. Accessed September, 2020.

[10] Timo Hintsch and Stefan Irnich. Large multiple neighborhood search for the clustered vehicle-routing problem. *European Journal of Operational Research*, 270(1):118 – 131, 2018.

[11] CPLEX CP Optimizer. `https://www.ibm.com/cz-en/analytics/cplex-cp-optimizer`. Accessed December, 2020.

[12] IBM ILOG CPLEX Optimization Studio CP Optimizer User's Manual. `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.studio.help/pdf/usrcpoptimizer.pdf`. Accessed December, 2020.

[13] Rajeev Kohli, Ramesh Krishnamurti, and Prakash Mirchandani. Average performance of greedy heuristics for the integer knapsack problem. *European Journal of Operational Research*, 154(1):36 – 45, 2004.

[14] Young Hoon Lee and Michael Pinedo. Scheduling jobs on parallel machines with sequence-dependent setup times. *European Journal of Operational Research*, 100(3):464–474, 1997.

[15] Andrea Lodi. *Mixed Integer Programming Computation*, pages 619–645. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[16] Willian T. Lunardi, Ernesto G. Birgin, Débora P. Ronconi, and Holger Voos. Metaheuristics for the online printing shop scheduling problem. *European Journal of Operational Research*, 2020.

[17] Shahrzad M. Pour, John H. Drake, Lena Secher Ejlertsen, Kourosh Marjani Rasmussen, and Edmund K. Burke. A hybrid constraint programming/mixed integer programming framework for the preventive signaling maintenance crew scheduling problem. *European Journal of Operational Research*, 269(1):341 – 352, 2018.

[18] Young Hoon Lee Mi-Yi Kim. Mip models and hybrid algorithm for minimizing the makespan of parallel machines scheduling problem with a single server. *Computers & Operations Research*, 2017.

[19] NP-hard. `https://xlinux.nist.gov/dads/HTML/nphard.html`. Accessed December, 2020.

[20] Robert Pellerin, Nathalie Perrier, and François Berthaut. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 280(2):395 – 416, 2020.

[21] David Pisinger and Stefan Ropke. *Large Neighborhood Search*, pages 399–419. Springer, 09 2010.

[22] Martin L. Puterman. Dynamic programming. In Robert A. Meyers, editor, *Encyclopedia of Physical Science and Technology (Third Edition)*, pages 673 – 696. Academic Press, New York, third edition edition, 2003.

[23] Francesca Rossi, Peter van Beek, and Toby Walsh. Chapter 1 - introduction. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 3 – 12. Elsevier, 2006.

[24] Xiaoyue Zhang Simin Huang, Linning Cai. Parallel dedicated machine scheduling problem with sequence-dependent setups and a single server. *Computers & Industrial Engineering*, 2009.

[25] Horst Tempelmeier and Lisbeth Buschkuhl. Dynamic multi-machine lotsizing and sequencing with simultaneous scheduling of a common setup resource. *International Journal of Production Economics*, 113(1):401–412, 2008.

[26] Paolo Toth and Daniele Vigo. Models, relaxations and exact approaches for the capacitated vehicle routing problem. *Discrete Applied Mathematics*, 123(1):487 – 512, 2002.

[27] Eva Vallada and Ruben Ruiz. A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. *European Journal of Operational Research*, 211(3):612–622, 2011.

[28] Marek Vlk, Antonin Novak, and Zdenek Hanzalek. Makespan minimization with sequence-dependent non-overlapping setups. In *Proceedings of the 8th International Conference on Operations Research and Enterprise Systems - Volume 1: ICORES,*, pages 91–101. INSTICC, SciTePress, 2019.

[29] Marek Vlk, Antonin Novak, Zdenek Hanzalek, and Arnaud Malapert. Non-overlapping sequence-dependent setup scheduling with dedicated tasks. In *International Conference on Operations Research and Enterprise Systems*, pages 23–46. Springer, 2019.

[30] Xing Zhao, Peter B Luh, and Jihua Wang. Surrogate gradient algorithm for lagrangian relaxation. *Journal of optimization Theory and Applications*, 100(3):699–712, 1999.

Appendix A List of Abbreviations

In Table 13 are listed abbreviations used in this thesis.

Abbreviation	Meaning
CP	Constraint Programming
DP	Dynamic Programming
GA	Genetic Algorithm
ILP	Integer Linear Programming
LB	lower bound
LOSOS	Locally Optimal Selection of Setups
MIP	Mixed Integer Programming
ROSOL	Resolution of Setup Overlaps Lazily
SMETI	Subproblem Modeling Evolving Through Iteration
VRP	Vehicle Routing Problem

Table 13: List of abbreviations

Appendix B CD Contents

In Table 14 are listed names of all directories on CD.

Directory name	**Description**
thesis	Master's thesis in pdf format.
thesis_sources	Latex Overleaf project sources.
project_sources	Root of the project structure.
project_sources\Smeti	C++ project containing code of the solution.
project_sources\Scripts	Folder containing python scripts generating figures.

Table 14: CD Contents

Appendix C Project Organization

The project is written in C++ and for successful execution it requires *ILOG CP optimizer (tested with version 12.09)* for CP models' execution, *Gurobi (tested with version 9.0)* for ILP models' execution, and *Google OR-Tools (tested with version 7.3)* for VRP warm starting. Since VRP warm start was rendered obsolete, the code using it might be commented out. The ILP models in the project are only used for comparison with other papers so that they might be commented out as well. The only requirement to run approaches proposed in this thesis is *ILOG CP Optimizer*. The project is ready to be run in Microsoft Visual Studio, but depending on the environment, reconfiguration of dependencies might be required.

The project is organized into the following Microsoft Visual Studio filters:

1. **CP models:** Contains classes representing CP models. $CP3$ represents model $\text{CP}_{Pairwise}$ and $CP4$ represents model $\text{CP}_{Flexible}$.
2. **Heuristic algos:** Contains everything related to LOSOS, ROSOL, and SMETI. It also contains reimplemented GA described in *Kim et al.* [18]. In code, the old naming scheme of algorithms is still present. LOSOS is named SHS, ROSOL is named $WCEA$ and SMETI is named $CPHeuristic$.
3. **ILP models:** Contains ILP models described in papers which were used in the comparison in Subsection 8.3.
4. **Machine history:** Contains classes for building the solution into objects. It also contains optimization methods and methods for testing and validation.
5. **Main classes:** Contains main class *Program.cpp* which handles the selection and calling of methods and other helper classes used over the whole project like *RunConfig* or *Instance*.
6. **Warmstarts:** Contains classes representing only warm start approaches. Legacy TSP warm start is also present.

There is also a helper folder containing Python scripts that build various graphical representations and figures from the solutions.

1. **exact_comp_tables.py:** Provides script which generates LaTeX or PNG tables comparing ILP models with CP models.
2. **graph_comp.py:** Provides script generating graphs comparing our proposed approaches.
3. **graphical_sol.py:** Provides script generating a graphical representation of a single solution obtained by any proposed method.
4. **heuristic_comp_tables.py:** Provides script which generates LaTeX or PNG tables comparing heuristic approaches from other papers to our SMETI.

Appendix D Testing and Output

As solutions are impossible to validate or draw by hand; there are methods provided to do that automatically. Function *validateSolution()* takes the solution and verifies if it correctly adheres to all constraints. Function *saveOutput()* generates text file representation of single instance solution. The batch text files containing results for multiple instances are generated by their respective functions in *Program.cpp*. The generated result text files can then be used to generate images, tables, and graphs using the provided Python scripts.

D.1 Running the Project

As mentioned, *Program.cpp* is the main project file. To run the project, the *main()* function in *Program.cpp* must be called and configuration of execution must be set. This can be done by passing arguments to the *Program.cpp* when executing it. The first argument is a boolean switch, which represents if we run on Linux, true, or Windows, false. The second argument is an integer variable representing one of the possible run modes. Run modes can represent single, batch, or comparison executions of various approaches and can be found described in *Program.cpp*. The third argument is a boolean switch between non-heuristic, true, or heuristic, false executions for the comparison run modes. The alternative way to set these parameters is to set them directly in the configuration header file *Program.h*. Output files are generated into the folder *results* which is in the same folder as the *Program.cpp*.