Informatics

Optimizing the Cell Assignment Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Anoki Eischer, BSc
Matrikelnummer 11824091

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Professor Dr. Nysret Musliu
Mitwirkung: Dr. Felix Winter

Wien, 2. Dezember 2024

Anoki Eischer   Nysret Musliu

Informatics

Optimizing the Cell Assignment Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Anoki Eischer, BSc
Registration Number 11824091

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Professor Dr. Nysret Musliu
Assistance: Dr. Felix Winter

Vienna, December 2, 2024

Anoki Eischer Nysret Musliu

Erklärung zur Verfassung der Arbeit

Anoki Eischer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 2. Dezember 2024

Anoki Eischer

Acknowledgements

I want to express sincere appreciation to my advisor Associate Professor Dr. Nysret Musliu and co-advisor Dr. Felix Winter for motivating me, especially in the last weeks. I do not know how much time I would have needed to finish this thesis without their everlasting support.

Furthermore, I want to thank to my family for fighting the consequences of the flood in Kritzendorf when I could not afford the time because of this thesis.

I am also deeply grateful to my friends for reminding me of the joy of life in dark days. Special thanks to my friend Jonas, who is working on a similar thesis. Exchanging experiences with him often grounded me in challenging times.

Finally, I want to sincerely thank my girlfriend Sabi for always supporting me and making me laugh whenever I needed it.

The work in this thesis was performed in the Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling. The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

Kurzfassung

Moderne Produktionswerke bestehen oft aus mehreren Maschinen und Zellen, in denen Zwischen- oder Enderzeugnisse gelagert werden. Diese Diplomarbeit beschäftigt sich mit Lösungsansätzen für ein in der Praxis relevantes Problem, welches in solchen Netzwerken auftritt. In dem behandelten Problem wird von jeder Maschine eine vorgegebene Sequenz an Aufträgen ausgeführt, welche Material konsumieren und produzieren. Für jeden Auftrag muss eine Zelle zugewiesen werden, um dessen produziertes Material zu lagern. Außerdem müssen gültige Startzeiten für die Aufträge aller Maschinen gefunden werden. Das Ziel des Problems ist, die kumulierte Verspätung aller Lieferaufträge zu minimieren, wobei ein Lieferauftrag gewisse Mengen von Materialien benötigt.

In der Vergangenheit wurden bereits einige Varianten von Planungsproblemen aus der Produktionsbranche behandelt. Allerdings werden nach unserem Wissensstand die spezifischen Ressourceneinschränkungen des von uns untersuchten Problems von keiner Variante, die in der wissenschaftlichen Literatur untersucht wurde, behandelt. Daher können existierende Methoden nicht verwendet werden, um qualitative Lösungen auf effiziente Art und Weise zu finden. Da in der Praxis auftretende Probleminstanzen sehr groß sein können, ist es ein herausforderndes Problem, hochwertige Lösungen in akzeptablem Zeitraum zu finden. Um dieses Ziel zu erreichen, werden innovative Lösungsmethoden benötigt.

In dieser Diplomarbeit stellen wir eine formale Spezifikation des zu lösenden Problems zur Verfügung und modellieren dieses mittels zwei verschiedener Constraint Optimization Formulierungen, welche für exakte Lösungsverfahren verwendet werden können. Weiters werden eine Konstruktionsheuristik sowie Nachbarschaftsoperatoren für das behandelte Problem entwickelt. Darauf basierend wird Simulated Annealing als metaheuristischer Lösungsansatz auf das Problem angewandt. Die verschiedenen Methoden werden auf einer praktischen Instanz, sowie einer großen Menge diverser, zufällig generierter Instanzen, getestet. Ein Vergleich bezüglich Qualität und Laufzeit wird aufgestellt. Beide Constraintprogrammierungs-Modelle sind erfolgreich darin, optimale Lösungen für kleinere Instanzen zu finden. Für viele der größeren Instanzen kann nur eines der Modelle die optimale Lösung im gegebenen Zeitlimit finden. Der Ansatz mittels Simulated Annealing löst einige Instanzen optimal und findet gültige Lösungen für die meisten Instanzen. Außerdem können die meisten Lösungen, welche von der Konstruktionsheuristik erzeugt werden, durch Simulated Annealing verbessert werden. Die evaluierte praktische Instanz wurde wegen deren Größe von keinem der Constraintprogrammierungs-Modelle gelöst, die Konstruktionsheuristik konnte allerdings eine gültige Lösung erzeugen.

Abstract

In the modern industry, production networks often consist of multiple machines and cells, which serve as storage facilities for final or intermediate products. In this thesis we are investigating solution approaches for a real-life problem appearing in such production networks. In the considered problem, each machine processes a fixed sequence of jobs that consume and produce material. Each job must be assigned a cell, which is used for storing the produced material. Additionally, a feasible schedule of jobs over all machines needs to be found. The objective of the problem is to minimize the total tardiness of a set of delivery orders, which require a certain amount of material.

Many variants of job scheduling problems from industrial manufacturing have been studied in the past. However, to the best of our knowledge, none of the investigated variants captures the specific resource constraints appearing in the considered problem. Thus, existing methods cannot be used to find good solutions efficiently. Due to the large size of real-life instances, finding high-quality solutions in a reasonable amount of time is a challenging problem that requires novel and innovative solution approaches.

This thesis provides a formal specification of the problem at hand and investigates two alternative constraint programming formulations which can be used for exact solving approaches. Furthermore, a construction heuristic is developed and neighborhood operators for the considered problem are proposed. Based on these operators, Simulated Annealing is applied as a metaheuristic technique for solving the problem. Experiments are performed on one real-life instance and a diverse set of instances that we randomly generated. We compare the different methods regarding solution quality and runtime and present an overview of the results. Both constraint models turned out to be successful in finding optimal solutions for small instances. For many of the larger instances, only one model found solutions within the given time limit. Simulated Annealing could solve numerous instances optimally and found feasible solutions for the majority of instances. Furthermore, the Simulated Annealing approach provided improved solutions for most of the instances. None of the constraint models solved the real-life instance due to its large size, whereas the construction heuristic was able to provide a feasible solution.

Contents

CHAPTER 1

Introduction

Decision-making in industrial manufacturing processes can become a complex task, since factories may consist of a large number of machines and storage facilities, and often many jobs need to be planned over a long period of time. The decisions to be made may consist of scheduling jobs, but also assigning them to machines and storage facilities. Handling such planning tasks manually using human resources not only consumes a lot of time but may also result in suboptimal decisions. Therefore, there is a strong need to investigate automated solution methods that can provide optimized solutions to such problems.

In this thesis, we are investigating a real-life problem arising in the agricultural animal feed industry, which we call the Cell Assignment Problem (CAP). In general, the CAP consists of managing the process in production networks composed of multiple machines and cells, which represent storage facilities such as silos. A sequence of jobs, that require and produce material, is fixed for every machine. One part of the problem is to find a feasible processing schedule over all machines. Additionally, cells must be assigned to jobs, which store the produced material. Another important part of the problem are the delivery orders, which consume certain quantities of certain materials from cells as soon as their requirements are satisfied and their due date is reached. The goal of the problem is to minimize the total tardiness of delivery orders, meaning that every delivery order starts as soon as possible after its due date.

Similar problems appearing in the literature are the resource-constrained project scheduling problem (RCPSP) and the job-shop scheduling problem (JSSP). Extensive overviews of research on solution methods for different variants of these problems are provided in [HB22] and [XSRH22], respectively. While the RCPSP shares with the CAP the concept of jobs which require resources, the aspect of the CAP that multiple machines work in parallel relates more to the JSSP. However, to the best of our knowledge, none of the investigated problem variants tackles the specific resource constraints emerging in the CAP.

Real-life instances of the CAP are using a large number of cells, jobs, and delivery orders. Solving the CAP under such extensive requirements is a challenging task, as a large number of complex constraints and objectives have to be regarded. Therefore, efficient modeling techniques and innovative algorithms are needed to ensure an efficient and punctual production process.

1.1 Aims of this Thesis

The main goals of this thesis are:

- Providing a formal definition of the CAP. This includes a specification of the input, decision variables, constraints, and objective.
- Providing an overview of related literature on similar problems.
- Formulating and implementing a constraint programming model as an exact solution approach to the CAP.
- Solving the CAP using a metaheuristic approach which we expect to produce high-quality solutions in a reasonable amount of time.
- Evaluating the various approaches on a diverse set of instances.

1.2 Main Contributions

The main results of this thesis are:

- We formulated a high-level constraint programming model for the CAP, which also serves as a problem specification. Furthermore, a sophisticated model has been developed, which uses an adapted input representation and additional predecessor variables to decrease the number of generated constraints.
- Using Simulated Annealing, a novel metaheuristic approach to the CAP is provided. To accomplish that, we developed a construction heuristic to find initial solutions of reasonable quality as a starting point for Simulated Annealing. Additionally, four types of moves have been designed, which are used to obtain neighborhood solutions.
- We developed a random instance generator based on real-life instances to establish a diverse set of realistic instances. Extensive parameter tuning and experimental evaluation was performed with the resulting instances.
- A summary of the experimental results is provided. This includes a comparison of the solutions found by the different methods regarding solution quality. The experiments show that the exact methods are able to provide optimal solutions for the majority of small instances. Furthermore, it can be seen that Simulated Annealing is used successfully to find feasible solutions also for larger instances and to solve many instances optimally.

1.3 Organization

Chapter 2 starts by formally defining the CAP and identifying related problems in the literature. We continue in Chapter 3 with a description of two different constraint programming models that can be used as an exact approach to the CAP. In Chapter 4, our metaheuristic approach to the CAP using Simulated Annealing is outlined. Chapter 5 explains the setup of the experiments performed and gives an overview of their outcome. We provide concluding remarks and mention future work in Chapter 6.

CHAPTER 2

The Cell Assignment Problem

Production networks in the CAP consist of multiple *machines*[1] and *cells*. The machines and cells are partly connected by directed *paths*. A small example of such a network can be seen in Figure 2.1.

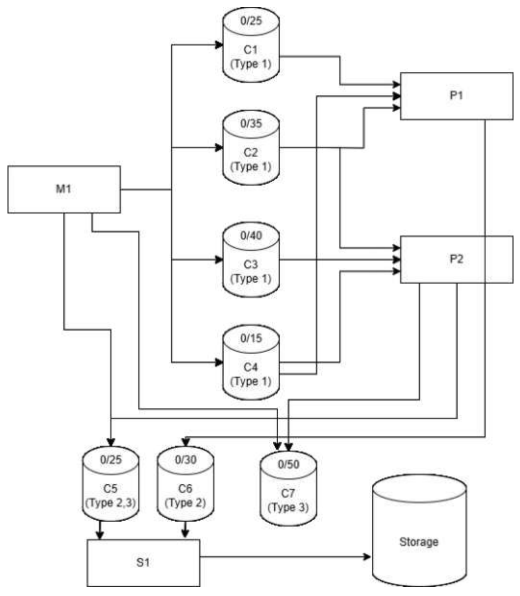


Figure 2.1: Example of a production network

During the production process, the feed has to be mixed by a mixing machine (M_1 in Figure 1) and, depending on the desired product, additionally pressed by a pressing machine (P_1 and P_2). After every production step, the feed is stored in a cell, which has a certain capacity and type.

[1]Machines are also denoted as *resources* in later chapters.

Cells of type 1 (pre-cells) store intermediate products that still need to be pressed. Cells of type 2 (bag-loading/sacking cells) store feed which needs to be packed into bags before delivering it to the customers. This is done by a bag loading machine (S_1 in Figure 1). The resulting bags are stored in the storage and waiting to be delivered. It is also possible to deliver feed directly from the cells if they are of type 3 (loading cells). A cell can only contain one material, representing a specific type of product, at a time. In contrast to cells, the storage may contain an arbitrary number of different materials and is not capacitated.

One production step is represented by a *task*, which requires a certain quantity of a certain material. This quantity must be taken from a cell c connected via a path to the machine processing the task. The cell c is then called the *input cell* of this task. One exception are tasks processed by the mixing machine, which do not require any material. The mixing machine is therefore also called a *source*. Every task t produces a certain material and quantity, which may be different from the consumed one. The produced material must be stored in a cell having a certain type, which is called the *output cell* of t. If t is processed by the bag loading machine, its produced material must be stored in the storage instead.

In practice, many tasks have identical properties, such as the required and produced material and quantity. This is because they are part of a bigger production step, called a *job*. In general, tasks of the same job can be processed by different machines of the same type (e.g. they can be processed by two different pressing machines). Other job-specific parameters are the earliest start time, which is the same for all tasks of this job, and the processing times for tasks belonging to this job.

The processing times specified for one job may be different on different machines. Additionally to the processing time, there is also a machine-specific transfer time for every job, which is the time needed to transport the material produced by one task of the job to its output cell. If tasks of the same job are processed multiple times in a row on the same machine, every consecutive task can be processed while the material from its predecessor is still being transported, i.e. the successor task does not need to wait until the end of its predecessor's transfer time. Such a task is called *parallelizable*. However, this is not possible if the predecessor task is part of a different job. For illustration, assume a machine has the sequence T_1, T_2, T_3, T_4, where T_1, T_2 and T_4 belong to the same job, but T_3 does not. Then, T_2 is parallelizable and can be started after T_1 has finished processing, but before the end of its transfer time. The other tasks are not parallelizable since they all have a predecessor task which is part of a different job.

Material can only be transported between machines and cells if they are connected by a path. A special constraint which needs to be considered is that some paths are in conflict with each other, meaning that they are not allowed to be active (i.e. used to transport material) at the same point of time. For practical reasons, the paths transporting the required material to and from the machine processing a task are considered as active during the transfer *and* processing time of the task. Furthermore, only one task at a time can transport material to a cell and only one task at a time can take material from a cell (but these two tasks are allowed to do this at the same time).

There are two types of delivery orders: Loading deliveries and bag-loading deliveries. Deliveries of the former type take the required material directly from one or multiple cells, which must be of type 3, while bag-loading deliveries consume the material from the storage. A delivery order may require multiple materials in different quantities. Every delivery has a due date which is also an earliest start date, meaning that the delivery cannot be delivered before that point of time. The objective of the problem is to minimize the total tardiness, i.e. the sum of the differences between the actual start time of every delivery and its due date.

In practice, trucks are usually collecting the materials required by delivery orders. Since the arrival time of these trucks cannot be planned exactly, there is a loading time needed, meaning that after the start of a delivery, the cells are assumed to contain the provided material until the loading time is finished. The storage can be freed instantly at the start of a bag-loading delivery since it is not capacitated.

In general, the problem to be solved consists of scheduling every task of a given set of tasks on one of its feasible machines and assigning the cells where the required material is taken from and the produced material is transported to. The scheduling part of the problem is currently solved using dispatching rules, which assign the tasks to machines and fix an order of all tasks assigned to one machine. However, these rules do not consider the assignment of cells. In this thesis, we are formulating and solving this problem: Assuming we already have fixed for every machine a sequence of tasks, for each task the cells storing the material consumed and produced are determined and exact starting times are assigned to every task and delivery order. We are referring to this problem as the Cell Assignment Problem, as a key aspect of a solution is the assignment of cells storing the material produced and consumed by the tasks.

2.1 Example Instance

An example instance of the CAP using the network from Figure 2.1 is specified in Figure 2.2. Path conflicts, earliest start times for jobs, and due dates for deliveries are omitted in the illustration.

On the top left, the task sequence for each machine is depicted. Since a task can be uniquely identified by the machine processing it, its position in the sequence, and the job it belongs to, only the latter is specified for each task in the sequence.

Furthermore, two delivery orders are specified. For each order, the required materials and quantities are denoted. For example, D_1 requires a quantity of 5 of material 4 and a quantity of 50 of material 5. Additionally, it is specified whether the material needs to be taken from the storage (in case of a bag-loading delivery) or cells. Remember that only cells of type 3 (loading cells) can be used to provide material for delivery orders.

For each job j, the material required and produced is denoted. Furthermore, the quantity required and produced by one task of j is stated. For jobs processed by the mixing-machine M_1, these parameters are not required, since they do not consume material. Additionally, the required type of output cell is specified for every job. The material produced by tasks of job J_6, which are processed by the bag-loading machine, must be transported to the storage. Finally, for each job, the processing and transfer time for one task of this job is depicted. Since these times depend on the machine processing the task, they are stated for each of the eligible machines.

$M_1 : J_1, J_3, J_5, J_5$
$P_1 : J_2$
$P_2 : J_2, J_4$
$S_1 : J_6, J_6$

Deliveries	D_1	D_2
req. mat.	$\{Mat_4, Mat_5\}$	$\{Mat_2\}$
req. quan.	$\{5, 50\}$	$\{30\}$
bag-loading	false	true

Jobs	J_1	J_2	J_3	J_4	J_5	J_6
mat. req.	-	Mat_1	-	Mat_3	-	Mat_2
mat. prod.	Mat_1	Mat_2	Mat_3	Mat_4	Mat_5	Mat_2
quan. req.	-	20	-	10	-	15
quan. prod.	20	15	10	5	25	15
req. cell type	Type 1	Type 2	Type 1	Type 3	Type 3	Storage
proc. times	$\{M_1 : 15\}$	$\{P_1 : 30, P_2 : 15\}$	$\{M_1 : 3\}$	$\{P_2 : 10\}$	$\{M_1 : 10\}$	$\{S_1 : 30\}$
trans. times	$\{M_1 : 4\}$	$\{P_1 : 10, P_2 : 6\}$	$\{M_1 : 5\}$	$\{P_2 : 5\}$	$\{M_1 : 5\}$	$\{S_1 : 0\}$

Figure 2.2: Example instance using the network illustrated in Figure 2.1

In the following, a few steps of a solution approach are illustrated. Since at the beginning, all cells are empty, no tasks requiring material can be started. The first task of M_1 belongs to job J_1, which does not require material. Therefore, J_1 is started. Its product is transported to C_1, which is indicated by the colored path in Figure 2.3.

After 15 minutes of processing and 4 minutes of transfer time, the task is finished. C_1 now contains the produced material, i.e. a quantity of 20 of material 1 (indicated by the red color). Now M_1 can resume with the next task. Furthermore, the first task of P_1 can be started using the material contained in C_1. This step is illustrated in Figure 2.4.

After 3 minutes of processing and 5 minutes of transfer time, the second task of M_1 is completed. The produced material can potentially be used for the second task of P_2 as this task belongs to job J_4 which requires a quantity of 10 of material 3. However, this task cannot be started at the

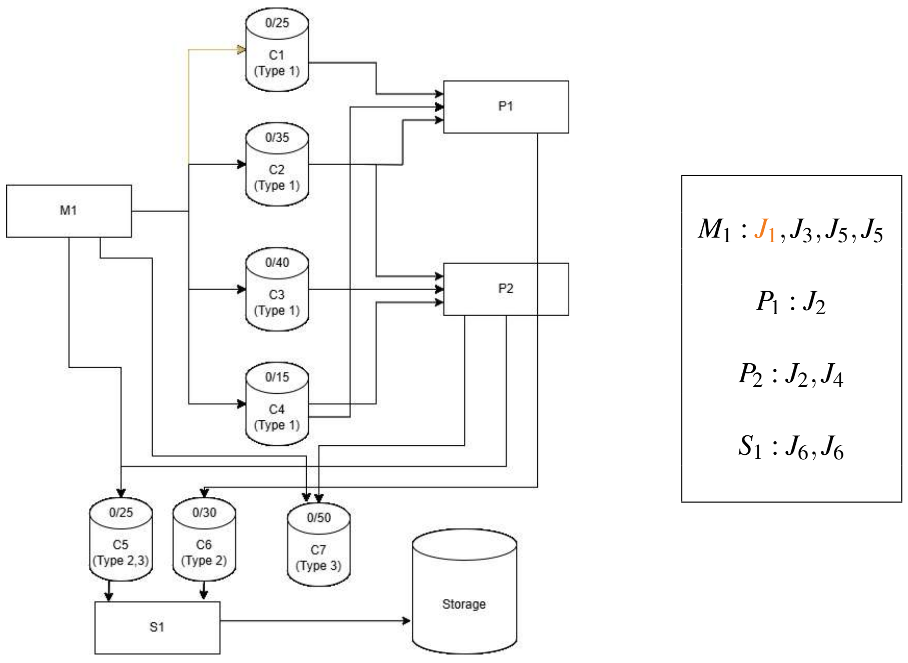


Figure 2.3: First step of the production process.

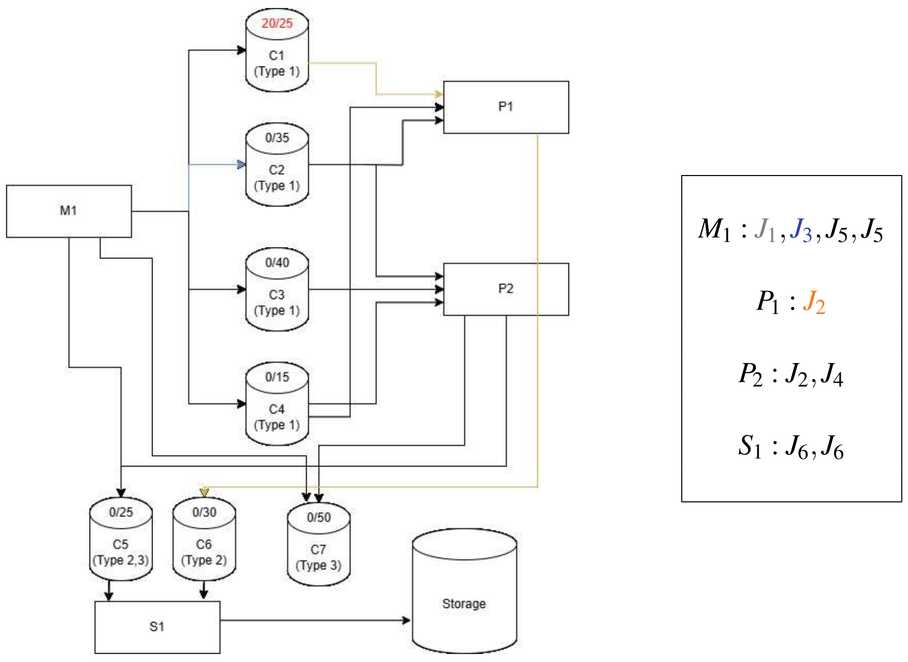


Figure 2.4: Second step (19 minutes after start)

moment, since the first task on P_2 needs to be processed before. M_1 resumes with its next task. This step is depicted in Figure 2.5.

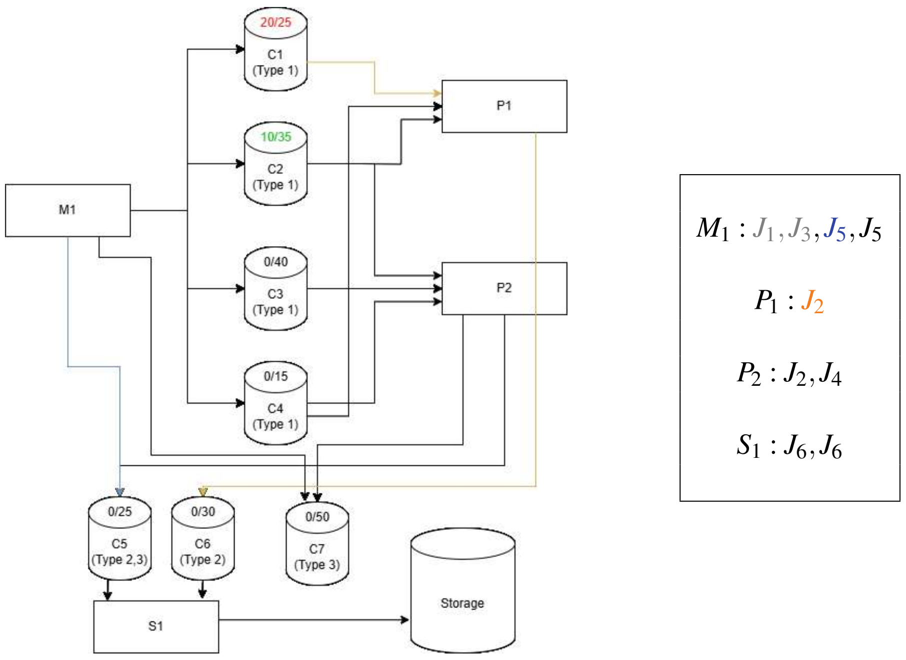
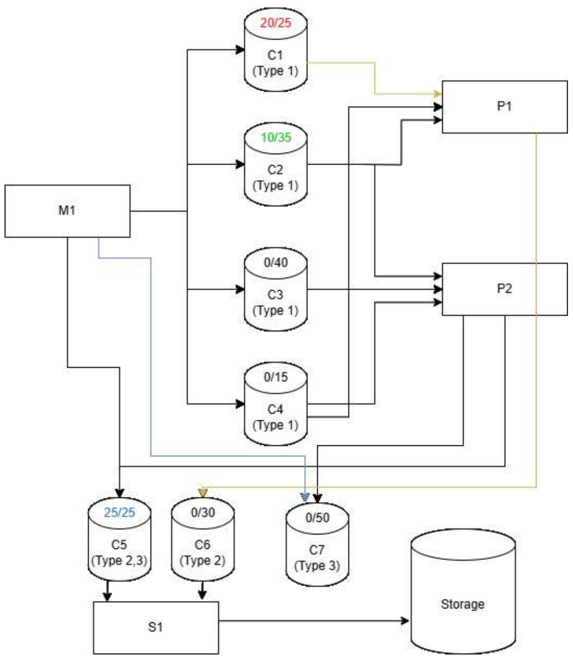


Figure 2.5: Third step of production process (27 minutes after start)

Since on M_1, two tasks of J_5 are processed in a row, the fourth task does not need to wait for the completion of the transfer time of the third one. Therefore, after 10 minutes of processing time, the fourth task of M_1 can be started. Since C_5 will be occupied by the material produced by the third task, another path is chosen. This step is not shown, but Figure 2.6 illustrates the state after the third task is finished (after 5 more minutes of transfer time).

Since the fourth task already started 5 minutes ago, it only requires 5 more minutes of processing and 5 minutes of transfer time to be completed. The resulting state is depicted in Figure 2.7. Now, the instance is not solvable anymore: Delivery order D_1 requires a quantity of 5 of material 4. Since D_1 is not a bag-loading delivery, the required material must be taken from loading cells, which are cells of type 3. However, both loading cells are already occupied with material 5. This material is also required by D_1 and cannot be removed until all requirements of this delivery are satisfied. Since a cell can only store one material at once, none of the loading cells is allowed to store material 4 anymore.

The instance could be solved by choosing cell C_6 for storing the material produced by both tasks belonging to job J_5 since this cell has enough capacity for the amount of material produced, in contrast to cell C_5. Then C_5 could be used to store material 4, which would be sufficient to satisfy the requirements of delivery order D_1.



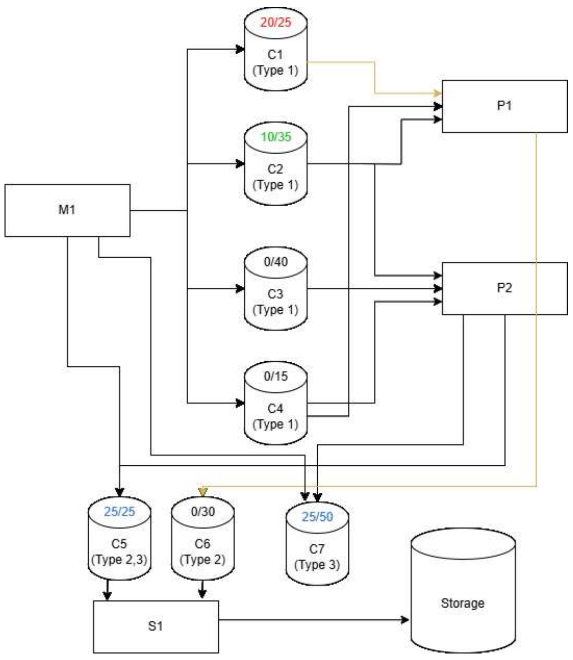
$M_1 : J_1, J_3, J_5, J_5$

$P_1 : J_2$

$P_2 : J_2, J_4$

$S_1 : J_6, J_6$

Figure 2.6: Fourth step of production process (42 minutes after start)



$M_1 : J_1, J_3, J_5, J_5$

$P_1 : J_2$

$P_2 : J_2, J_4$

$S_1 : J_6, J_6$

Figure 2.7: Fifth step of production process (52 minutes after start)

2.2 Related Work

The CAP can be viewed as a special case of the resource-constrained project scheduling problem (RCPSP). In [HB10] fundamental variants and models for this problem are discussed. An updated survey is provided in [HB22], considering new extensions of the RCPSP that emerged in the last decade. In the general RCPSP, activities (which may correspond to tasks in the CAP) of a project need to be scheduled. The order of the activities is constrained by predecessor relations. Furthermore, each activity requires and consumes different quantities of one or multiple resources (which may correspond to material in the CAP), that renew themselves over time.

The objective of the classical RCPSP is to minimize the makespan, i.e. the finishing time of the last activity. However, other variants concerning the minimization of tardiness with respect to certain due dates, similar to the objective of the CAP, exist. One version of the RCPSP where activities have due dates is discussed in [WLZ20]. In contrast to the CAP, they consider weights for each activity and the weighted tardiness as the objective.

In the CAP, resources do not renew themselves but are provided by tasks. This variant of the RCPSP is an extension to so-called cumulative resources. A version of this problem where the availability of resources at a time is upper-bounded, which is similar to cell capacities in the CAP, is considered in [BGZ11]. However, the problem described does not include finding a schedule for activities on multiple machines working in parallel, which plays a key role in the CAP.

Other versions of the RCPSP extending the resource concept are considered by [Fu14] and [ZSN17]. They introduce a new resource category called materials. These materials are consumed by activities and can be re-ordered if necessary. In addition to finding an activity schedule, deciding when which material should be ordered is part of the proposed problems. We may interpret a delivery in the CAP as an activity in such an extension of the RCPSP. A task providing the required material can then be viewed as ordering material for the respective delivery. However, tasks in the CAP are subject to resource requirements as well, which is not the case for material orders in the considered problems.

Another interesting variant of the RCPSP is the preemptive RCPSP, where activities can be interrupted. Although in the CAP, the interruption of tasks is prohibited, we can interpret a job (consisting of multiple tasks) as an activity in the RCPSP, which may be interrupted between the executions of its tasks. Since the CAP allows consecutive tasks to overlap timewise if and only if they are part of the same job, the interruption of a job results in setup times for its successor. Variants of the preemptive RCPSP with setup times for interrupted activities are discussed in [LVDVDC19] and [ANM14]. In the considered problem formulations, the interruption of activities is allowed at discrete time points. The CAP is more restrictive in this regard since jobs can only be interrupted between the execution of tasks and tasks may differ in processing time, i.e. the intervals between potential interruptions are not necessarily equal.

Another important aspect of our problem is that the demand for multiple deliveries needs to be met, where each delivery needs a set of tasks to be finished beforehand to produce the required material. Therefore, a delivery can be interpreted as a project in a multi-project variant of the RCPSP. A version of this problem where each project has a due date which is also its earliest finishing time is discussed in [BY10]. This is similar to the concept of due dates in the CAP. However, in the problem described, there are explicit predecessor constraints given, whereas in the CAP these arise implicitly by material requirements. Furthermore, in contrast to the CAP, the parallel execution of activities is not allowed.

Although the CAP shares characteristics with these extensions of the RCPSP, some aspects are not covered by any proposed variant. For example, the concept of machines does not appear in the survey provided by [HB22]. In this sense, the CAP is more similar to the job-shop scheduling problem (JSSP). An overview of the main aspects and variations of the JSSP is provided in [XSRH22]. In the general JSSP, jobs are split into so-called operations, which must be processed in a predefined order. Each operation is assigned to one of multiple machines which needs a certain processing time to complete the operation. The goal of the JSSP is to find operation sequences for all machines s.t. the makespan is minimized.

By interpreting a task in the CAP as an operation contributing to a bigger job, a similarity to the JSSP emerges in the sense that in both problems, fragments of a job are processed by predefined machines and a feasible sequence of these fragments over all machines needs to be found. However, one difference between these problems is that while in the JSSP, finding a sequence for each machine is part of the problem, the machine-internal job sequences are fixed in the CAP. Furthermore, the classical JSSP is not subject to resource constraints. Respective problem variants appear in the literature, one example is the resource-constrained flexible job-shop scheduling problem (RCFJSP), discussed in [FTSCC06]. Here, an operation needs (renewable) resources to be processed. Since in the CAP, resources are not renewable, it can be seen as an extension to cumulative resources. However, in the flexible version of the Job Shop Scheduling Problem, the operations can be assigned to any machines, whereas in the CAP these assignments are fixed. Additionally, the resource constraints in the RCFJSP allow every job to potentially access every resource available. In contrast, in the CAP resources are consumed from cells and the assignment of cells storing the material consumed and produced by tasks plays a key role when solving the CAP.

A rather practice-oriented problem with resource constraints similar to those in the CAP is the Grain silo location-allocation problem discussed in [MKKT18]. Here, food grain must be transported from procurement centers to demand points and on the way needs to be stored in silos coming in two different levels (similar to the different cell types in the CAP). The quantity produced and consumed at the different locations is known beforehand and the assignment of silos is a key aspect of the problem. Although there is no concept of machines, we can interpret the transportation of food grain to and from silos as tasks that need to be processed. In contrast to the CAP, where multiple materials exist, in the considered problem there is only one type of product. Furthermore, the relocation of silos plays an important role, whereas cell locations are not relevant in the CAP.

To conclude, although there are many problems sharing characteristics with the CAP, to the best of our knowledge no problem includes consideration of multiple machines operating in parallel on a production network, which constrains the availability of resources in the way it happens in the CAP. Thus, innovative solution methods are required to tackle this specific problem.

CHAPTER 3

Exact Methods

In this chapter, we propose a high-level constraint model **M1** for the CAP which can be used as an exact solution approach. This model additionally serves as a formal specification for the CAP. **M1** is described in Section 3.1. Additionally, we investigated a second model by partly adapting the solution representation and reformulating constraints, which resulted in the model **M2** described in Section 3.2. Model **M2** requires some additional assumptions on the instances. First, that there are at most two paths between the same resource and cell. The second assumption concerns conflicting paths and is further described in Section 3.2.1. Both assumptions were satisfied by the instances we consider in this thesis.

3.1 High-Level Constraint Model M1

This model makes use of high-level constraints to provide a rather direct specification of the CAP.

3.1.1 Input

Input	C	M	P	R	J	T	D
Set of	Cells	Materials	Paths	Resources	Jobs	Tasks	Delivery Orders

Table 3.1: Input sets

The input for **M1** consists of a set of cells, materials, paths, resources[1], jobs, tasks, and delivery orders. The respective notations are specified in Table 3.1. Each object is described by the parameters stated in Table 3.2. The 3 cell types are represented by the integers 1-3. Although the *cellReq*-parameter requires a cell type as value, it may also take the dummy type 0, representing that the produced material must be transported to the storage. The encoding of cell types is

[1]The term *resource* is used as a synonym for machine in the following.

specified in Table 3.3. Since in general a cell can have several types, a Boolean parameter $cellOfType_{i,j}$ is needed, stating for every cell i if it is of type j.

The union of resources and cells is considered as the set N of *nodes*. This set represents the domain of the start and end locations of paths. Since paths may also end in the storage, N is extended by 0, which represents the storage.

For each task, the resource processing it is specified, as well as to which job it belongs. Tasks on the same resource are assumed to be given in the order they need to be processed, i.e. if $resOfTask_i = resOfTask_{i+1}$, $i+1$ is the task coming directly after i in the sequence of the respective resource.

3.1.2 Variables

A solution consists for every task t of an assignment of paths used to transport material to and from t, as well as the start time of t. The endtime of t can be directly derived from its start.

To keep track of the contents of cells and the storage, the production process is modeled as a sequence of events sorted by time. Since the contents of cells or the storage change only at the end of a task or a delivery, a *proper* event always corresponds to such an incident. In the following the set of proper events is called E. The variable $taskDelOfEvent_i$ therefore represents the task or delivery which ends at event i. The materials and quantities contained in the cells and the storage is updated after every event by respective helper variables.

Additionally, a solution contains for every delivery d its start time, as well as which material and quantity required by d is provided by which cells. These values will always be 0 for sack deliveries[2] since they only need material from the storage.

The variables and their intended meaning are described below. Note that the domain of some variables is extended by a dummy value, which is indicated by the subscript 0. This has various reasons: *pathToTask* is allowed to be 0 for tasks processed by resources without incoming paths. The set E_0 extends the set of proper events by the special event 0, representing the start of the production process. In case of the *matAfter* and *cellProvMat* variables, the value 0 represents that no material is contained in the respective cell.

The objective is to minimize the total tardiness, i.e. for each delivery the the difference between its due date and its start time:

$$min \sum_{i \in D} dueDate_i - startOfDelivery_i$$

It must be noted that the difference between the due date and the start of a delivery cannot be negative, since a delivery is not allowed to start before its due date.

[2]The term sack delivery is in the following used as a synonym for bag-loading delivery.

	Parameter	Domain	Meaning
Cell parameters $\forall i \in C, j \in \{1,2,3\}$	cap_i	$\mathbb{N}$	maximum capacity of cell i
	$cellOfType_{i,j}$	bool	cell i is of type j
	$initMat_i$	M	material contained by cell i initially
	$initQuan_i$	$\mathbb{N}$	quantity contained by cell i initially
Storage parameters $\forall i \in M$	$initStorQuan_i$	$\mathbb{N}$	quantity of material i contained by storage initially
Path parameters $\forall i, j \in P$	$startL_i$	N	startLocation of path i
	$endL_i$	N	endLocation of path i
	$confi_{i,j}$	bool	paths i and j are conflicting
Delivery params. $\forall i \in D, j \in M$	$delReqQuan_{i,j}$	$\mathbb{N}$	quantity of material j required for delivery order i
	$sackDelivery_i$	bool	Delivery order i requires material from the storage
	$dueDate_i$	$\mathbb{N}$	due date of delivery order i
Job parameters $\forall i \in J, j \in R$	$matProd_i$	M	material produced by tasks of job i
	$quanProd_i$	$\mathbb{N}$	quantity produced by one task of job i
	$matReq_i$	M	material required by tasks of job i
	$quanReq_i$	$\mathbb{N}$	quantity required by one task of job i
	$processTime_{i,j}$	$\mathbb{N}$	processing time for one task of job i on resource j
	$transTime_{i,j}$	$\mathbb{N}$	transfer time for one task of job i on resource j
	$earliestStart_i$	$\mathbb{N}$	earliest start time of any task of job i
	$cellReq_i$	$\{0,1,2,3\}$	cell type required after job i
Task parameters $\forall i \in T$	$resOfTask_i$	R	resource processing task i
	$jobOfTask_i$	J	job of task i

Table 3.2: Input Parameters for **M1**

ID	0	1	2	3
Cell type	Storage	Pre-Press	Sacking	Loading

Table 3.3: Encoding of cell types

	Variable	Domain	Meaning
Task Vars. $\forall i \in T$	$pathToTask_i$	P_0	material is transported to task i over this path
	$pathFromTask_i$	P	material is transported from task i over this path
	$startOfTask_i$	$\mathbb{N}$	start time of task i
	$endOfTask_i$	$\mathbb{N}$	end time of task i including transportation time
Event Vars. $\forall i \in C, j \in E_0, k \in E, m \in M$	$taskDelOfEvent_k$	$T \cup D$	task / delivery ending at the kth event
	$timeOfEvent_j$	$\mathbb{N}$	time when the jth event is happening
	$startOfEvent_j$	$\mathbb{N}$	start time of task/delivery ending at the jth event
	$isDeliveryEvent_k$	bool	true iff event k corresponds to the end of a delivery
	$quanAfter_{i,j}$	$\mathbb{N}$	quantity contained by cell i after event j
	$matAfter_{i,j}$	M_0	material contained by cell i after event j
	$storQuanAfter_{m,j}$	$\mathbb{N}$	quantity of material m contained by storage after event j
Delivery Vars. $\forall i \in C, j \in D$	$startOfDelivery_j$	$\mathbb{N}$	time when delivery j is started
	$cellProvQuan_{i,j}$	$\mathbb{N}$	quantity provided by cell i for delivery j
	$cellProvMat_{i,j}$	M_0	material provided by cell i for delivery j

Table 3.4: Variables used in **M1**

3.1.3 Constraints

M1 can be encoded by the following constraints, which are subdivided into different categories, depending on their function.

Location Constraints

$$pathToTask_i \neq 0 \iff \exists p \in P \mid endL_p = resOfTask_i, \forall i \in T \tag{1.1}$$

$$startL_{pathFromTask_i} = resOfTask_i, \forall i \in T \tag{1.2}$$

$$pathToTask_i \neq 0 \implies endL_{pathToTask_i} = resOfTask_i, \forall i \in T \tag{1.3}$$

$$cellReq_{jobOfTask_i} = 0 \implies endL_{pathFromTask_i} = 0, \forall i \in T \tag{1.4}$$

$$cellReq_{jobOfTask_i} \neq 0 \implies cellOfType_{endL_{pathFromTask_i}, cellReq_{jobOfTask_i}}, \forall i \in T \tag{1.5}$$

Constraints 1.2 and 1.3 ensure that the assigned path to/from a task ends in/starts from the resource processing the task. Since some tasks are processed by a resource having no incoming paths, in this case, $pathToTask$ is set to a dummy path 0 by constraint 1.1. Constraints 1.4 and 1.5 force that the path from a task ends in a cell having the required cell type, or being the storage (type 0), respectively.

Conflict Constraints

$$\begin{aligned} &i \neq j \wedge (startOfTask_i < endOfTask_j \wedge startOfTask_j < endOfTask_i) \implies \\ &\qquad \neg conf_{pathToTask_i, pathToTask_j} \wedge \neg conf_{pathFromTask_i, pathFromTask_j}, \forall i, j \in T \end{aligned} \tag{1.6}$$

$$\begin{aligned} &i \neq j \wedge (startOfTask_i < endOfTask_j \wedge startOfTask_j < endOfTask_i) \wedge \\ &\qquad resOfTask_i \neq resOfTask_j \wedge pathToTask_i \neq 0 \wedge pathToTask_j \neq 0 \\ &\qquad \implies startL_{pathToTask_i} \neq startL_{pathToTask_j}, \forall i, j \in T \end{aligned} \tag{1.7}$$

$$\begin{aligned} &i \neq j \wedge (startOfTask_i < endOfTask_j \wedge startOfTask_j < endOfTask_i) \wedge \\ &\qquad resOfTask_i \neq resOfTask_j \wedge endL_{pathFromTask_i} \neq 0 \\ &\qquad \implies endL_{pathFromTask_i} \neq endL_{pathFromTask_j}, \forall i, j \in T \end{aligned} \tag{1.8}$$

Constraint 1.6 ensures that if the execution of two tasks overlap, the respective paths used to/from the tasks do not conflict. Constraints 1.7 and 1.8 are needed additionally to prohibit overlapping tasks from delivering to/from the same cell with some exceptions: If for at least one of these tasks, $pathToTask = 0$, i.e. the task is processed by a mixing-resource (having no incoming paths), constraint 1.7 is not required and if the task delivers to the storage, constraint 1.8 is not required, since it is allowed to deliver to the storage concurrently. Another exception for both constraints happens if the overlapping tasks are processed by the same resource, which is only possible if the task j executed later is parallelizable, in which case it is allowed to use the same cells.

Timing Constraints

$$\begin{aligned} endOfTask_i = startOfTask_i &+ processTime_{resOfTask_i, jobOfTask_i} \\ &+ transTime_{resOfTask_i, jobOfTask_i}, \forall i \in T \end{aligned} \tag{1.9}$$

$$\begin{aligned} resOfTask_i = resOfTask_j \land i > j \implies startOfTask_i \geq endOfTask_j \\ -[jobOfTask_i = jobOfTask_j] \cdot transTime_{resOfTask_i, jobOfTask_i}, \forall i, j \in T \end{aligned} \tag{1.10}$$

$$startOfTask_i \geq earliestStart_{jobOfTask_i}, \forall i \in T \tag{1.11}$$

$$startOfDelivery_i \geq dueDate_i, \forall i \in D \tag{1.12}$$

Constraint 1.9 links the variables representing the start and end time of tasks. Constraint 1.10 prohibits two tasks that are processed by the same resource from overlapping, with the exception that the successor task can start within the transfer time of its predecessor if they are part of the same job. Finally, constraints 1.11 and 1.12 ensure that a task/delivery does not start before the earliest start of its corresponding job, or its due date, respectively.

Event Constraints

$$timeOfEvent_{i-1} \leq timeOfEvent_i, \forall i \in E \tag{1.13}$$

$$i \neq j \iff taskDelOfEvent_i \neq taskDelOfEvent_j, \forall i, j \in E \tag{1.14}$$

$$taskDelOfEvent_i \in D \iff isDeliveryEvent_i, \forall i \in E \tag{1.15}$$

$$\begin{aligned} timeOfEvent_i = [\neg isDeliveryEvent_i] \cdot endOfTask_t \\ +[isDeliveryEvent_i] \cdot (startOfDelivery_t + [\neg sackDelivery_t] \cdot 60) \\ \text{where } t = taskDelOfEvent_i, \forall i \in E \end{aligned} \tag{1.16}$$

$$\begin{aligned} startOfEvent_i = \neg isDeliveryEvent_i \cdot startOfTask_t \\ +isDeliveryEvent_i \cdot startOfDelivery_t \\ \text{where } t = taskDelOfEvent_i, \forall i \in E \end{aligned} \tag{1.17}$$

Constraint 1.13 forces that events are sorted by their time points in ascending order. Constraint 1.14 ensures that every event represents a different task or delivery and, since the domain of events is exactly the union of the domain of tasks and deliveries, which are disjoint, every task and delivery is represented by a unique event. Constraint 1.15 forces that for all delivery events *isDeliveryEvent* is true, whereas for all task events is evaluates to false. Constraints 1.16 and 1.17 link the start and time of an event with the start/end of its corresponding task/delivery. It must be noted that in Constraint 1.16 a case distinction between sack and loading deliveries is made: While the former do not require any time (i.e. the start equals the end time), loading deliveries need a loading time of 60 minutes.

Requirement Constraints

$$\begin{aligned} \neg isDeliveryEvent_i \wedge pathToTask_t \neq 0 \wedge quanReq_{jobOfTask_t} > 0 \wedge \\ (timeOfEvent_e > startOfTask_t \vee e = max(j \in E_0 \mid timeOfEvent_j \leq startOfEvent_i) \\ \implies matAfter_{startL_{pathToTask_t},e} = matReq_{jobOfTask_t} \wedge \\ quanAfter_{startL_{pathToTask_t},e} \geq quanReq_{jobOfTask_t} \\ \text{where } t = taskDelOfEvent_i, \forall i \in E, e \in 0 \ldots i-1 \end{aligned} \quad (1.18)$$

$$\begin{aligned} isDeliveryEvent_i \wedge \neg sackDelivery_d \wedge cellProvMat_{c,d} \neq 0 \wedge \\ (timeOfEvent_e > startOfDel_d \vee e = max(j \in E_0 \mid timeOfEvent_j \leq startOfEvent_i) \\ \implies matAfter_{c,e} = cellProvMat_{c,t} \\ \text{where } d = taskDelOfEvent_i, \forall i \in E, \forall e \in 0 \ldots i-1, c \in C \end{aligned} \quad (1.19)$$

$$\begin{aligned} \sum_{i \in E} cellProvQuan_{c,d}[isDeliveryEvent_i \wedge \neg sackDelivery_d \wedge cellProvMat_{c,d} = m \\ \wedge (timeOfEvent_e > startOfDel_d \vee e = max(j \in E_0 \mid timeOfEvent_j \leq startOfEvent_i) \\ \text{where } d = taskDelOfEvent_i] \leq quanAfter_{c,e}, \forall e \in 0 \ldots i-1, c \in C, m \in M \end{aligned} \quad (1.20)$$

$$\begin{aligned} \neg sackDelivery_i \implies \sum_{c \in C} (cellProvMat_{c,i} = j) \cdot cellProvQuan_{c,i} \\ = delReqQuan_{i,j}, \forall i \in D, j \in M \end{aligned} \quad (1.21)$$

$$\neg cellOfType_{i,3} \implies cellProvQuan_{i,j} = 0, \forall i \in C, j \in D \quad (1.22)$$

Constraints 1.18 - 1.20 ensure that after every event $e < i$ satisfying certain criteria, the material and quantity requirements of the task/delivery d ending at event i are met. These criteria are the following: Either e happens after the start of d, which together with $e < i$ implies that e happens during the execution of d, or e is the last event happening before the start of d. Note that in constraint 1.20 after event e the quantity contained in c must be at least the sum of quantities provided by all deliveries taking some material m from c and overlapping with the execution of the task/delivery ending at e or happening directly after e. Here summing over all these deliveries is necessary because it is allowed that multiple deliveries take material concurrently from one cell. Since constraints 1.19 and 1.20 refer to the variables deciding how much quantity and which material is provided by which cell for the delivery, constraint 1.21 is needed additionally to ensure that in total the right amount of the right material is provided. Constraint 1.22 prohibits non-loading cells from providing material for deliveries.

Inventory constraints

$$\begin{aligned} quanAfter_{i,j} = quanAfter_{i,j-1} + \neg isDeliveryEvent_j \cdot \\ (-(startL_{pathToTask_t} = i) \cdot quanReq_{jobOfTask_t} \\ +(endL_{pathFromTask_t} = i) \cdot quanProd_{jobOfTask_t}) \\ -isDeliveryEvent_j \cdot \neg sackDelivery_t \cdot cellProvQuan_{i,t} \\ \text{where } t = taskDelOfEvent_j, \forall j \in E, i \in C \end{aligned} \tag{1.23}$$

$$\begin{aligned} storageQuanAfter_{i,j} = storageQuanAfter_{i,j-1} + \neg isDeliveryEvent_j \cdot \\ (endL_{pathFromTask_t} = 0 \wedge matProd_{jobOfTask_t} = i) \cdot quanProd_{jobOfTask_t} \\ -isDeliveryEvent_j \cdot sackDelivery_t \cdot delReqQuan_{t,i} \\ \text{where } t = taskDelOfEvent_j, \forall j \in E, i \in M \end{aligned} \tag{1.24}$$

$$\begin{aligned} quanAfter_{i,j+1} > 0 \wedge matAfter_{i,j} \neq 0 \\ \implies matAfter_{i,j+1} = matAfter_{i,j}, \forall j \in E, i \in C \end{aligned} \tag{1.25}$$

$$\begin{aligned} \neg isDeliveryEvent_i \wedge endL_{pathFromTask_t} \neq 0 \implies matAfter_{endL_{pathFromTask_t},i} \\ = matProd_{jobOfTask_t} \text{ where } t = taskDelOfEvent_j, \forall j \in E \end{aligned} \tag{1.26}$$

$$0 \leq quanAfter_{i,j} \leq cap_i, \forall i \in C, j \in E_0 \tag{1.27}$$

$$storageQuanAfter_{i,j} \geq 0, \forall i \in M, j \in E_0 \tag{1.28}$$

These constraints ensure that the cell/storage content is altered in a feasible way after every event. Constraints 1.23 and 1.24 calculate the cell/storage quantity and constraint 1.25 forces that if a cell is nonempty before and after an event, the material contained stays the same. Constraint 1.26 ensures that the material contained in the output cell of a task is the material produced by the task. Finally, constraints 1.27 and 1.28 restrict the domain of the quantity variables s.t. they are positive and the cell capacity is not exceeded.

Initialization constraints

$$timeOfEvent_0 = 0 \wedge startOfEvent_0 = 0 \tag{1.29}$$

$$quanAfter_{c,0} = initQuan_c \wedge matAfter_{c,0} = initMat_c, \forall c \in C \tag{1.30}$$

$$storageQuanAfter_{i,0} = initStorageQuan_i, \forall i \in M \tag{1.31}$$

The initialization constraints initialize the variables concerning event 0, which represents the start of the manufacturing process, to the initial values.

Symmetry breaking constraints

$$quanAfter_{c,e} = 0 \iff matAfter_{c,e} = 0, \forall c \in C, e \in E_0 \tag{1.32}$$

$$\begin{aligned} sackDelivery_i \implies cellProvQuan_{c,i} = 0 \wedge \\ cellProvMat_{c,i} = 0, \forall i \in D, c \in C \end{aligned} \tag{1.33}$$

$$cellProvQuan_{i,j} = 0, \leftrightarrow cellProvMat_{i,j} = 0, \forall i \in C, j \in D \tag{1.34}$$

In some cases multiple different assignments to a variable lead practically to the same solution, if the other variables are fixed. To give an example, if a cell is empty, this cell could contain an arbitrary material without violating any of the constraints formulated in the previous sections.

The emerge of such symmetric solutions increases the size of the search space and can be avoided by formulating symmetry breaking constraints.

Constraint 1.32 prohibits an empty cell from containing a material different from 0. Constraint 1.33 forces cells to provide quantity and material 0 for sack deliveries, since these deliveries only need material from the storage. Constraint 1.34 forces a cell to provide the dummy material 0 if it does not provide quantity for a delivery and vice versa.

3.2 Alternative Model M2

As an alternative to **M1**, the solution representation and constraints are partly adapted with the motivation of giving a more efficient problem formulation. This is mainly achieved by reformulating the conflict constraints and requirements of deliveries and by introducing predecessor variables to decrease the number of generated constraints. To accomplish that, a different input format is required and additional assumptions on the structure of the input are taken. Although this makes the model unusable for special cases of the CAP, the instances considered in this thesis satisfy these assumptions.

3.2.1 Input

The cell- and storage parameters used in **M2** are the same as in **M1** (see Table 3.2). The first difference compared to **M1** is that while in **M1** the property that a resource is a source is computed by a constraint checking if an incoming path exists, in **M2** this is computed in the preprocessing phase and specified as a resource parameter $isSource_i$. Furthermore, the notion of paths is completely removed. Connections between resources and cells are specified indirectly using the new resource parameters $noOutPathExists$ and $noInPathExists$. Note that they also allow cell 0 as argument, which represents that the respective resource is not connected to the storage. Another difference compared to **M1** is that in the input of **M2** for every resource i the subset of tasks processed by i is specified explicitly by the $tasksOfRes_i$ parameter. Although this is redundant information, it makes it possible to decrease the number of constraints generated. Note that some constraints refer to the set $tasksOfRes_i^+$, which is defined as $tasksOfRes_i$ without the first task processed by i.

Parameter	Domain	Meaning	
$isSource_i$	bool	resource i has no incoming paths	$\forall i \in R$
$noOutPathExists_{i,j}$	bool	no path starting from res. i and ending in cell j exists	$\forall i \in R, j \in C_0$
$noInPathExists_{i,j}$	bool	no path starting from cell j and ending in res. i exists	$\forall i \in R, j \in C_0$
$tasksOfRes_i$	$\subseteq T$	tasks processed by resource i	$\forall i \in R$

Table 3.5: New resource parameters for **M2**

To encode conflicting paths, conflicts between tuples of resource-cell pairs are introduced. In the following, such a pair (r,c) is considered as active if any of the paths connecting r and c is active.

We distinguish between two types of conflicts: The simpler type is a so called *pair-conflict* between two paths p_1 and p_2, where p_1 is the only path connecting resource r_1 with cell c_1 and p_2 is the only path connecting resource r_2 with cell c_2. The set of pair conflicts is denoted by K_2. These conflicts are represented using 4 arrays, where the value of each array at index i represents either one of the resources or one of the cells involved in pair conflict i. Since the direction of the corresponding paths could be ambiguous using this formulation, we additionally made use of the fact that in realistic instances, conflicts only appear between paths starting from a resource and ending in a cell. Therefore, only these paths are considered. For example, if $pairConflictR1_1 = r_1$, $pairConflictR2_1 = r_2$, $pairConflictC1_1 = c_1$ and $pairConflictC2_1 = c_2$, then pair conflict 1 is the conflict between the path connecting r_1 and c_1 and the path connecting r_2 and c_2.

Parameter	Domain	Meaning	
$pairConflictR1_i$	R	first resource of pair conflict i	$\forall i \in K_2$
$pairConflictR2_i$	R	second resource of pair conflict i	$\forall i \in K_2$
$pairConflictC1_i$	C	first cell of pair conflict i	$\forall i \in K_2$
$pairConflictC2_i$	C	second cell of pair conflict i	$\forall i \in K_2$

Table 3.6: New pair-conflict parameters for **M2**

The more complex *triple conflicts*, represented by the set K_3, occur when there are two paths p_1 and p_2 between the same resource r and cell c where p_1 conflicts with different paths than p_2. Then, for each pair of paths q_1, q_2 where p_1 conflicts with q_1 and p_2 conflicts with q_2, the following holds: While one of the paths of p_1 and p_2 is active, not both q_1 and q_2 can be used. Let s_1, s_2 be the resources and d_1, d_2 the cells connected by the paths q_1 and q_2 respectively. Then we can break this down to resource-cell pairs by prohibiting the activation of both (s_1, d_1) and (s_2, d_2) while the combination (r, c) is used. This abstraction however requires some additional assumptions: First, that p_1 and p_2 are the only paths between r and c. Since the number of paths connecting the same resource and cell is at most 2 in realistic instances, they satisfy this assumption. Second, that if there is a path between s_1 and d_1 which is not q_1 (or symmetrically for s_2, d_2 and q_2), it conflicts with p_1 or p_2, since otherwise is could be used as an alternative path which can be activated independently. The realistic instances also satisfied this criterion.

Since most of the deliveries arising in practice only require one material, the *delReqQuan* parameters used in **M1** turned out to be a very sparse matrices, which resulted in the generation of many zero-sum constraints. To avoid this, the set of *subdeliveries* S is added to the input of **M2**, where one subdelivery represents the requirement of one material by one delivery. Therefore, there is only a certain quantity of *one* material required by each subdelivery. To define the requirements of a delivery, for each delivery the corresponding subset of subdeliveries is specified.

Another difference between **M1** and **M2** is that in the latter the set of jobs is completely removed.

Parameter	Domain	Meaning	
$tripleConflictR1_i$	R	first resource of triple conflict i	$\forall i \in K_3$
$tripleConflictR2_i$	R	second resource of triple conflict i	$\forall i \in K_3$
$tripleConflictR3_i$	R	third resource of triple conflict i	$\forall i \in K_3$
$tripleConflictC1_i$	C	first cell of triple conflict i	$\forall i \in K_3$
$tripleConflictC2_i$	C	second cell of triple conflict i	$\forall i \in K_3$
$tripleConflictC3_i$	C	third cell of triple conflict i	$\forall i \in K_3$

Table 3.7: New triple-conflict parameters for **M2**

Parameter	Domain	Meaning	
$quanReqBySubDel_i$	$\mathbb{N}$	quantity required by subDelivery j	$\forall i \in S$
$matReqBySubDel_i$	M	material required by subDelivery j	$\forall i \in S$
$subDelsOfDel_i$	$\subseteq S$	subdeliveries belonging to delivery i	$\forall i \in D$

Table 3.8: New delivery parameters for **M2**

Instead, all job-specific properties are computed for each task in a preprocessing phase. The notion of jobs would still be necessary since tasks can be started in parallel if they are part of the same job and are scheduled consecutively on the same resource. Instead of using this formulation, this property is explicitly defined for each task i as $isParallelizable_i$. Furthermore, for each task t the set of cells that are infeasible choices for storing the material produced by t due to their cell type is computed.

3.2.2 Variables

Since there are no paths in **M2**, the variable sets $pathToTask$ and $pathFromTask$ are replaced by variable sets $inCellTask$ and $outCellTask$. Note that both variable sets are allowed to take the dummy value 0, where $inCellTask = 0$ represents that the resource is a source and $outCellTask = 0$ represents that the task delivers to the storage. Another difference between **M1** and **M2** is that in the latter the $taskDelOfEvent$ variables are replaced by the variables $eventOfTask$ and $eventOfDel$. Furthermore, the $startOfEvent$ variables are removed. Their only use case in **M1** is finding for every task/delivery i the last event before the start of i, since its requirements must be satisfied after this event. Therefore, new variables $predEventOfTask$, $predEventOfDel$, $predTimeOfTask$ and $predTimeOfDel$ are defined. Note that the material required must be available during the whole duration of a task/delivery, which is handled in **M1** by enforcing that the material is available after every event overlapping the duration of the consuming task/delivery. To decrease the number of generated constraints in this way, in **M2** the additional $potentialQuanAfter_{i,j}$ variables are introduced, representing the potential quantity contained in cell i after event j, which is defined by the actual quantity contained minus the

Parameter	Domain	Meaning	
$matProd_i$	M	material produced by task i	$\forall i \in T$
$quanProd_i$	$\mathbb{N}$	quantity produced by task i	$\forall i \in T$
$matReq_i$	M	material required for task i	$\forall i \in T$
$quanReq_i$	$\mathbb{N}$	quantity required for task i	$\forall i \in T$
$processTime_i$	$\mathbb{N}$	processing time for task i	$\forall i \in T$
$transTime_i$	$\mathbb{N}$	transfer time for task i	$\forall i \in T$
$earliestStart_i$	$\mathbb{N}$	earliest start time of task i	$\forall i \in T$
$cellReq_i$	$[0,3]$	cell type required after task i	$\forall i \in T$
$resOfTask_i$	R	resource processing task i	$\forall i \in T$
$isParallelizable_i$	bool	task i executes the same job as its predecessor task on the same resource	$\forall i \in T$
$infTypeForTask_{i,j}$	bool	cell j is not of type $cellReq_i$	$\forall i \in T, j \in C$

Table 3.9: New task Parameters for **M2**

material required by any tasks/deliveries consuming from cell i which have already started or have j as predecessor event. Note that no corresponding variables for the storage are needed, since it is not capacitated.

The objective is the same as in **M1**:

$$min \sum_{i \in D} dueDate_i - startOfDelivery_i$$

3.2.3 Constraints

M2 can be encoded by the following constraints, which are subdivided into different categories, depending on their function.

Location Constraints

$$inCellTask_i = 0 \iff isSource_{resOfTask_i}, \forall i \in T \tag{2.1}$$

$$inCellTask_i \neq j, \forall i \in T, j \in C \text{ where } \neg isSource_{resOfTask_i} \land noInPathExists_{resOfTask_i} j \tag{2.2}$$

$$outCellTask_i \neq j, \forall i \in T, j \in C \text{ where } noOutPathExists_{resOfTask_i} j \tag{2.3}$$

	Variable	Domain	Meaning
Task Vars. $\forall i \in T$	$inCellTask_i$	C_0	cell providing material for task i
	$outCellTask_i$	C_0	cell storing material produced by task i
	$startOfTask_i$	$\mathbb{N}$	start time of task i
	$endOfTask_i$	$\mathbb{N}$	end time of task i including transportation time
	$eventOfTask_i$	E	event representing the end of task i
	$predEventOfTask_i$	E_0	last event happening before task i starts
	$predTimeOfTask_i$	$\mathbb{N}$	time of $predEventOfTask_i$
Event Vars. $i \in C, j \in E$, $k \in M$	$timeOfEvent_j$	$\mathbb{N}$	time when the jth event is happening
	$quanAfter_{i,j}$	$\mathbb{N}$	quantity contained by cell i after event j
	$potentialQuanAfter_{i,j}$	$\mathbb{N}$	potential quantity contained by cell i after event j
	$matAfter_{i,j}$	M_0	material contained by cell i after event j
	$storQuanAfter_{k,j}$	$\mathbb{N}$	quantity of material k contained by storage after event j
Delivery Vars. $\forall i \in C, j \in D$	$startOfDelivery_j$	$\mathbb{N}$	time when delivery j is started
	$cellProvQuan_{i,j}$	$\mathbb{N}$	quantity provided by cell i for delivery j
	$cellProvMat_{i,j}$	M_0	material provided by cell i for delivery j
	$eventOfDel_j$	E	event representing the end of delivery j
	$predEventOfDel_j$	E_0	last event happening before delivery j starts
	$predTimeOfDel_j$	$\mathbb{N}$	time of $predEventOfDel_j$

Table 3.10: Variables used in **M2**

$$outCellTask_i = 0, \forall i \in T \text{ where } cellReq_i = 0 \tag{2.4}$$

$$outCellTask_i \neq j, \forall i \in T, j \in C \text{ where } cellReq_i \neq 0 \land infTypeForTask_{i,j} \tag{2.5}$$

$$inCellTask_i \neq outCellTask_i, \forall i \in T \tag{2.6}$$

Constraints 2.1 - 2.5 are just translations from constraints 1.1 - 1.5 using the new variables. Note that compared to the old formulation, in constraint 2.1 no existential statement is needed and constraints 2.2 - 2.5 do not use variables in indices anymore, which may result in better performance. The additional Constraint 2.6 prohibits a task to use the same cell as in- and out-cell. Although this is theoretically allowed by the other constraints and also by model **M1**, in realistic production instances this is not possible due to the network structure. Therefore, Constraint 2.6 decreases the size of the search space although it is a redundant constraint, assuming a realistic network structure.

Timing Constraints

$$endOfTask_i = startOfTask_i + processTime_i + transTime_i, \forall i \in T \tag{2.7}$$

$$startOfTask_j \geq endOfTask_{j-1}, \forall i \in R, j \in tasksOfRes_i^+ \text{ where } \neg isParallelizable_j \tag{2.8}$$

$$\begin{aligned} startOfTask_j \geq endOfTask_{j-1} - transTime_j, \forall i \in R, j \in tasksOfRes_i \\ \text{where } j-1 \in tasksOfRes_i \wedge isParallelizable_j \end{aligned} \tag{2.9}$$

$$startOfTask_i \geq earliestStart_i, \forall i \in T \tag{2.10}$$

$$startOfDelivery_i = predTimeOfDel_i, \forall i \in D \text{ where } dueDate_i \leq predTimeOfDel_i \tag{2.11}$$

$$startOfDelivery_i = dueDate_i, \forall i \in D \text{ where } dueDate_i > predTimeOfDel_i \tag{2.12}$$

Constraints 2.7 and 2.10 are translations from constraint 1.9 and 1.11 using the new task-specific input parameters. Constraints 2.8 and 2.9 are a more efficient formulation of constraint 1.10 since only linearly many constraints (in the number of tasks) are generated. They make use of the fact that tasks on the same resource are given in the scheduled order and deal separately with the case that a task j and its predecessor on the same resource can be parallelized or not. Constraints 2.11 and 2.12 ensure that a delivery cannot start before its due date, but additionally exploit that in an optimal solution a delivery must start as soon as it is satisfied and its due date is reached. Constraint 2.11 therefore forces that an overdue delivery starts directly after its predecessor event is finished while constraint 2.12 ensures that a delivery satisfied before its due date t starts as soon as t is reached. The fact that it must be satisfied comes from the requirement and inventory constraints, stating that delivery requirements must be met after its predecessor event.

Conflict Constraints

$$\begin{aligned} & outCellTask_i = outCellConflictC1_k \land outCellTask_j = outCellConflictC2_k \implies \\ & \neg(startOfTask_i < endOfTask_j \land startOfTask_j < endOfTask_j), \forall i,j \in T, k \in K_2 \\ & \text{where } resOfTask_i = outCellConflictR1_k \land resOfTask_j = outCellConflictR2_k \end{aligned} \tag{2.13}$$

$$\begin{aligned} outCellTask_r = tripleConflictC1_i \land outCellTask_s = tripleConflictC2_i \land \\ outCellTask_t = tripleConflictC3_i \implies \\ \neg(startOfTask_r < endOfTask_s \land startOfTask_s < endOfTask_r) \lor \\ \neg(startOfTask_t < endOfTask_s \land startOfTask_s < endOfTask_t), \\ \forall r,s,t \in T, i \in K_3 \text{ where } resOfTask_r = tripleConflictR1_i \land \\ resOfTask_s = tripleConflictR2_i \land resOfTask_t = tripleConflictR3_i \end{aligned} \tag{2.14}$$

$$\begin{aligned} inCellTask_i \neq inCellTask_j, \forall i,j \in T \text{ where } j > i \land inCellTask_i \neq 0 \land \\ startOfTask_i < endOfTask_j \land startOfTask_j < endOfTask_i \land \\ (resOfTask_i \neq resOfTask_j \lor \neg isParallelizable_j) \end{aligned} \tag{2.15}$$

$$\begin{aligned} outCellTask_i \neq outCellTask_j, \forall i,j \in T \text{ where } j > i \land outCellTask_i \neq 0 \land \\ startOfTask_i < endOfTask_j \land startOfTask_j < endOfTask_i \land \\ (resOfTask_i \neq resOfTask_j \lor \neg isParallelizable_j) \end{aligned} \tag{2.16}$$

Constraint 2.13 encodes that for every pair of tasks i, j processed by resources which are conflicting when certain out-cells are used (no matter which path to these cells is chosen), if tasks i and j use these cells respectively, then they must not overlap temporally. Constraint 2.14 is a similar constraint dealing with triple conflicts: In this case, there are two paths connecting the resource and out-cell used by task s, where one path conflicts with the path connecting the resource and out-cell used by task r and the other one conflicts with the path connecting the resource and out-cell used by task t. Therefore, task r or task t must not overlap temporally with task s, since otherwise none of the two paths between the resource and out-cell used by s can be activated. Constraints 2.15 and 2.16 ensure that for every pair of tasks which overlap temporally they must use different in and out cells since they are not allowed to access the same cell at the same time. One exception is if i and j are executed on the same resource and j is allowed to run in parallel.

Event Constraints

$$timeOfEvent_{i-1} \leq timeOfEvent_i, \forall i \in E \tag{2.17}$$

$$allDifferent(\{eventOfTask_i \mid i \in T\} \cup \{eventOfDel_i \mid i \in D\}) \tag{2.18}$$

$$eventOfTask_{j-1} < eventOfTask_j, \forall i \in R, j \in tasksOfRes_i^+ \tag{2.19}$$

$$timeOfEvent_{eventOfTask_i} = endOfTask_i, \forall i \in T \tag{2.20}$$

$$timeOfEvent_{eventOfDel_i} = startOfDelivery_i + 60, \forall i \in D \text{ where } \neg sackDelivery_i \tag{2.21}$$

$$timeOfEvent_{eventOfDel_i} = startOfDelivery_i, \forall i \in D \text{ where } sackDelivery_i \tag{2.22}$$

Constraint 2.17 equals constraint 1.13. The global constraint 2.18 equals constraint set 1.14. The redundant constraint 2.19 is using the property that the tasks of a resource are specified in the order they are scheduled to decrease the size of the search space. Constraints 2.20 - 2.22 encode constraint 1.16, dealing with each case separately.

Predecessor Constraints

$$predTimeOfTask_i \leq startOfTask_i, \forall i \in T \tag{2.23}$$

$$\begin{gathered} predTimeOfTask_i \geq timeOfEvent_j, \forall i \in T, j \in E_0 \\ \text{where } j < eventOfTask_i \wedge timeOfEvent_j \leq startOfTask_i \end{gathered} \tag{2.24}$$

$$\begin{aligned} predEventOfTask_i = j \implies predTimeOfTask_i = timeOfEvent_j, \\ \forall i \in T, j \in E_0 \text{ where } j < eventOfTask_i \end{aligned} \tag{2.25}$$

$$predTimeOfDel_i \leq startOfDel_i, \forall i \in D \tag{2.26}$$

$$\begin{gathered} predTimeOfDel_i \geq timeOfEvent_j, \forall i \in D, j \in E_0 \\ \text{where } j < eventOfDel_i \wedge timeOfEvent_j \leq startOfDel_i \end{gathered} \tag{2.27}$$

$$\begin{aligned} predEventOfDel_i = j \implies predTimeOfDel_i = timeOfEvent_j, \\ \forall i \in D, j \in E_0 \text{ where } j < eventOfDel_i \end{aligned} \tag{2.28}$$

As already pointed out, **M2** uses predecessor variables to reformulate requirement constraints. Therefore, new predecessor constraints are needed to define the behaviour of these variables. Constraints 2.23 and 2.24 ensure that $predTimeOfTask_i$ is upper bounded by the start of task i, but at least as big as the time of any event happening before. Constraint 2.25 is needed additionally to force that $predTimeOfTask_i$ is the time of an event. To generate less constraints, only events $j < eventOfTask_i$, i.e. events happening before the end of task i, are considered as potential predecessor events. Constraints 2.26 - 2.28 are the corresponding formulations for the delivery variables.

Requirement Constraints

$$matAfter_{inCellTask_i, predEventOfTask_i} = matReq_i, \forall i \in T \text{ where } quanReq_i > 0 \tag{2.29}$$

$$matAfter_{i, predEventOfDel_j} = cellProvMat_{i,j}, \forall i \in C, j \in D \text{ where } cellProvQuan_{i,j} > 0 \tag{2.30}$$

$$\begin{aligned} \sum_{c \in C} (cellProvMat_{c,i} = matReqBySubDel_j) \cdot cellProvQuan_{c,i} \\ = quanReqBySubDel_j, \forall i \in D, j \in subDelsOfDel_i \end{aligned} \tag{2.31}$$

$$\begin{aligned} cellProvMat_{i,j} \neq m, \forall i \in C, j \in D, m \in M \text{ where} \\ \neg sackDelivery_j \wedge m \notin \{matReqBySubDel_k \mid k \in subDelsOfDel_j\} \end{aligned} \tag{2.32}$$

$$cellProvQuan_{i,j} = 0, \forall i \in C, j \in D \text{ where } \neg cellOfType_{i,3} \tag{2.33}$$

Constraints 2.29 and 2.30 are weaker versions of 1.18 and 1.19, since they only force the material after the predecessor event of a task/delivery to match the material required / provided by the respective cell. The quantity constraints are handled by inventory constraints 2.40-2.42 which are described in further detail below. The advantage of this formulation over the one used in **M1** is that the constraints are not checking for every event if it is the predecessor, but refer directly to the predecessor event using the respective variables. Constraint 2.31 is a translation of constraint 1.21 using subdeliveries. Since a delivery contains no subdeliveries for materials which are not (i.e. in a quantity of 0) required, constraint 2.32 is needed additionally to prohibit cells from providing such materials. Constraint 2.33 equals constraint 1.22, ensuring that material for deliveries is taken from loading cells only.

Inventory constraints

$$\begin{aligned} quanAfter_{i,j} = quanAfter_{i,j-1} + quanProd_t, \forall i \in C, j \in E, t \in T \\ \text{where } i = outCell_t \wedge j = eventOfTask_t \end{aligned} \tag{2.34}$$

$$\begin{aligned} quanAfter_{i,j} = quanAfter_{i,j-1} - quanReq_t, \forall i \in C, j \in E, t \in T \\ \text{where } i = inCell_t \wedge j = eventOfTask_t \end{aligned} \tag{2.35}$$

$$\begin{aligned} quanAfter_{i,j} = quanAfter_{i,j-1}, \forall i \in C, j \in E, t \in T \\ \text{where } i \neq inCell_t \wedge i \neq outCell_t \wedge j = eventOfTask_t \end{aligned} \tag{2.36}$$

$$\begin{aligned} quanAfter_{i,j} = quanAfter_{i,j-1} - cellProvidingQuan_{i,d}, \forall i \in C, j \in E, d \in D \\ \text{where } j = eventOfDel_d \end{aligned} \tag{2.37}$$

$$\begin{aligned} quanAfter_{i,j+1} > 0 \wedge matAfter_{i,j} \neq 0 \\ \implies matAfter_{i,j+1} = matAfter_{i,j}, \forall j \in E, i \in C \end{aligned} \tag{2.38}$$

$$matAfter_{c,i} = matProd_t, \forall i \in C, j \in E, t \in T \text{ where } i = eventOfTask_t \wedge c = outCell_t \tag{2.39}$$

$$\begin{aligned} potentialQuanAfter_{i,j} = {} & potentialQuanAfter_{i,j-1} + quanProd_k \\ & - \sum_{t \in T} quanReq_t[j = predEvent_t \land c = inCell_t] \\ & - \sum_{d \in D} cellProvidingQuan_{i,d}[j = predEvent_d \land \neg sackDel_d], \\ & \forall i \in C, j \in E, k \in T \text{ where } i = outCell_k \land j = eventOfTask_k \end{aligned} \tag{2.40}$$

$$\begin{aligned} potentialQuanAfter_{i,j} = {} & potentialQuanAfter_{i,j-1} \\ & - \sum_{t \in T} quanReq_t[j = predEvent_t \land c = inCell_t] \\ & - \sum_{d \in D} cellProvidingQuan_{i,d}[j = predEvent_d \land \neg sackDel_d], \\ & \forall i \in C, j \in E, k \in T \text{ where } i \neq outCell_k \land j = eventOfTask_k \end{aligned} \tag{2.41}$$

$$\begin{aligned} potentialQuanAfter_{i,j} = {} & potentialQuanAfter_{i,j-1} \\ & - \sum_{t \in T} quanReq_t[j = predEvent_t \land c = inCell_t] \\ & - \sum_{d \in D} cellProvidingQuan_{i,d}[j = predEvent_d \land \neg sackDel_d], \\ & \forall i \in C, j \in E, k \in D \text{ where } j = eventOfDel_k \end{aligned} \tag{2.42}$$

$$\begin{aligned} storageQuanAfter_{i,j} = {} & storageQuanAfter_{i,j-1} - \sum_{d \in D, s \in subDelsOfDel_d} quanReqBySubDel_s \\ & [j = predEvent_d \land sackDel_d \land i = matReqBySubDel_s] + quanProd_k, \\ & \forall i \in M, j \in E, k \in T \text{ where } i = matProd_k \land j = eventOfTask_k \land outCell_k = 0 \end{aligned} \tag{2.43}$$

$$\begin{aligned} storageQuanAfter_{i,j} = {} & storageQuanAfter_{i,j-1} - \sum_{d \in D, s \in subDelsOfDel_d} quanReqBySubDel_s \\ & [j = predEvent_d \land sackDel_d \land i = matReqBySubDel_s], \\ & \forall i \in M, j \in E, k \in T \text{ where } (i \neq matProd_k \lor outCell_k \neq 0) \land j = eventOfTask_k \end{aligned} \tag{2.44}$$

$$\begin{aligned} storageQuanAfter_{i,j} = {} & storageQuanAfter_{i,j-1} - \sum_{d \in D, s \in subDelsOfDel_d} quanReqBySubDel_s \\ & [j = predEvent_d \land sackDel_d \land i = matReqBySubDel_s], \\ & \forall i \in M, j \in E, k \in D \text{ where } j = eventOfDel_k \end{aligned} \tag{2.45}$$

$$0 \leq quanAfter_{i,j} \leq cap_i, \forall i \in C, j \in E_0 \tag{2.46}$$

$$storageQuanAfter_{i,j} \geq 0, \forall i \in M, j \in E_0 \tag{2.47}$$

Constraints 2.34 - 2.37 are implementing constraint 1.23, dealing with each case separately. Similarly, constraints 2.38 and 2.39 are translations from 1.24 and 1.25 using the new variables. Constraints 2.40 - 2.42 ensure that the new variables representing the potential quantity of a cell after an event behave as intended: After an event *i*, the potential quantity is reduced by the quantity consumed by any task or delivery having *i* as predecessor event, meaning that *i* is the last end event happening before their start. It is feasible to consume the quantity already after event *i*, since no material will be added before the start of the respective tasks and deliveries. If *i* represents the end of a task *k*, the material produced by it must be added into its out cell, which is enforced by constraint 2.40. Constraints 2.43 - 2.45 implement the quantity change of the different materials contained in the storage. Here, similar to the potential quantity of cells, the material required by a sack delivery is consumed already after its predecessor event. Since the storage is not capacitated, there is no need to differentiate between potential and actual quantity contained in the storage. Note that in these constraints the *quanReqBySubDel*-variables are used and that each subdelivery of a delivery requires exactly one material, which is specified by the *matReqBySubDel*-variable. Similar to constraint 2.40, constraint 2.43 ensures that the quantity produced by a task *k* delivering to the storage (which is indicated by $outCell_k = 0$) must be added into the storage after the respective event.

Initialization constraints

$$timeOfEvent_0 = 0 \tag{2.48}$$

$$quanAfter_{c,0} = initQuan_c \land matAfter_{c,0} = initMat_c \forall c \in C \tag{2.49}$$

$$\begin{aligned} potentialQuanAfter_{c,0} = initQuan_c - \sum_{t \in T} quanReq_t [predEvent_t = 0 \land c = inCell_t] \\ \land matAfter_{c,0} = initMat_c, \forall c \in C \end{aligned} \tag{2.50}$$

$$\begin{aligned} storageQuanAfter_{i,0} = initStorageQuan_i - \sum_{d \in D, s \in subDelsOfDel_d} quanReqBySubDel_s \\ [predEvent_d = 0 \land sackDel_d \land i = matReqBySubDel_s], \forall i \in M \end{aligned} \tag{2.51}$$

Constraints 2.48 and 2.49 equal constraints 1.29 and 1.30. The new constraint 2.50 ensures the initial potential quantity is calculated as intended. Similarly, constraint 2.51 calculates the initial potential storage quantities.

Symmetry breaking constraints

$$quanAfter_{c,e} = 0 \iff matAfter_{c,e} = 0, \forall c \in C, e \in E_0 \tag{2.52}$$

$$cellProvQuan_{c,i} = 0 \wedge cellProvMat_{c,i} = 0, \forall i \in D, c \in C \text{ where } sackDel_i \tag{2.53}$$

$$cellProvQuan_{i,j} = 0, \leftrightarrow cellProvMat_{i,j} = 0, \forall i \in C, j \in D \tag{2.54}$$

These constraints equal the symmetry-breaking constraints 1.32 - 1.34 from **M1** up to some minor reformulations.

CHAPTER 4

Metaheuristic Approach

In this chapter, we develop an alternative approach to solve the CAP using a Simulated Annealing based local search. A construction heuristic for the problem is proposed to provide initial solutions for Simulated Annealing. We continue by describing the solution representation and the neighborhood operators used. Furthermore, a modified solution cost is presented, which incorporates the number of hard constraint violations and is required to evaluate infeasible solutions. Finally, the full procedure of the metaheuristic approach is outlined.

4.1 Construction Heuristic

The construction heuristic focuses on finding a feasible solution for the given instance. The heuristic works essentially by processing an event queue Q, where an event usually represents the end of a task or delivery. In some cases, which are further described below, an event represents the potential start of a task or delivery.

The initial phase of the construction heuristic is outlined in Algorithm 4.1. It starts by sorting the deliveries by their due date in ascending order and checking for each delivery if it is satisfied, which means that in total enough quantity of the required material is contained in one or multiple loading cells in the case of a loading delivery or in the storage in the case of a sack delivery. Some deliveries may be satisfied although their due date has not been reached yet. In this case, a potential start event for the corresponding delivery is added at the time of its due date into the event queue, which will be processed later. After starting all satisfied overdue deliveries, the procedure $tryToStartPendingTasks()$ checks for every pending task if its requirements are met, in which case the task is started.

A task j is defined as *pending* if the resource r processing it is currently free and j is the next task processed by r. The procedures checking tasks and deliveries for satisfiability are defined in Algorithm 4.2. Note that starting a task or delivery sets the variables representing the start of a task/delivery as well as the variables determining for each cell the material and quantity provided.

Furthermore, the *potential quantity* contained in the respective cells is updated. The potential quantity contained in a cell c is the quantity contained minus the quantity that will be consumed from c by any tasks or deliveries that have already started.

Algorithm 4.1: Initial phase of the construction heuristic

```
Procedure initialPhase():
    order deliveries by due date;
    for $d \in D$ do
        tryToStartDelivery($d$);
    end
    tryToStartPendingTasks();
```

Algorithm 4.2: Procedures checking satisfiability of pending tasks and deliveries

```
Procedure tryToStartDelivery($d$):
    if $d$ is not queued and delivery is satisfied then
        if due date of d reached then
            start delivery;
        else
            mark $d$ as queued;
            insert potential start event for delivery $d$ into $Q$ at due date;
        end
    end
Procedure tryToStartPendingTasks():
    for $r \in R$ do
        if $r$ is currently free then
            tryToStartNextTask($r$);
        end
    end
Procedure tryToStartNextTask($r$):
    $j$ = next task to be processed by $r$;
    if $j$ is not queued and $(r.isSource \vee p := findInPathForNextTask(r) \neq null) \wedge q :=$
      $findOutPathForNextTask(r) \neq null$ then
        if earliest start of $j$ reached then
            start task $j$ on resource $r$ using incoming path $p$ and outgoing path $q$;
        else
            mark $j$ as queued;
            insert potential start event of task $j$ on resource $r$ into $Q$ at earliest start;
        end
    end
```

The requirements of a task j processed by resource r are met if a feasible incoming path for j (except the case that r is a source) and a feasible outgoing path for j can be found. Assuming that j is the next task to be processed by r, the procedures $findInPathForNextTask(r)$ (Algorithm 4.3) and $findOutPathForNextTask(r)$ (Algorithm 4.4) respectively return a feasible incoming and outgoing path for j, if one exists, and *null* otherwise. If the requirements of a pending task are met before its earliest start, a potential start event for this task is added into Q at the time of its earliest start.

Algorithm 4.3: Procedure for finding a feasible incoming path

```
1 Procedure findInPathForNextTask(r):
2     $j$ = next task to be processed by $r$;
3     for $p \in$ paths ending in $r$ do
4         $c = p.startLocation$;
5         if $c$ is currently not delivering another resource $\wedge$ no path conflicting with $p$ is
            currently active $\wedge$ material contained in $c = j.matReq$ $\wedge$ potential quantity
            contained in $c \geq j.quanReq$ then
6             return $p$;
7         end
8     end
9     return $null$;
```

A path ending in r is a feasible incoming path for j if it does not conflict with any currently active path and starts from a cell currently not delivering any other resource. Furthermore, the cell must contain enough potential quantity of the material required by j.

If r is a sacking resource, there is only one path starting from r, which is the path connecting r to the storage. This path is a feasible outgoing path for a task processed by r in any case. If r is not a sacking resource, a path starting from r is a feasible outgoing path for j if it ends in a cell c which is of the type required by j, currently not delivered by any other resource and not conflicting with any currently active paths. Furthermore, c must be connected to a resource processing a *successor task* of j. A successor task of a task j is defined as a task that has not started yet and requires the material produced by j. Additionally, c needs to have enough *potential capacity* for the quantity produced by j. The potential capacity of a cell c is its remaining capacity minus the quantity consumed by tasks/deliveries that have already started. Since the material produced by a task is placed in its output cell not until its end, cells having too little remaining capacity but sufficient potential capacity can be considered as potential output cells.

It remains to calculate at which point of time c will be a *feasible output cell* for j. A cell c is a feasible output cell for task j if it is empty or contains the same material which is produced by j. Additionally, c must have enough remaining capacity for the quantity produced by j. If these requirements are satisfied when $findOutPathForNextTask$ is called, j can be started using path p. Otherwise, j can only be started using path p if c will be a feasible output cell for j at the end of its process and transfer time. Therefore, if one or multiple tasks and deliveries consuming enough capacity are currently in progress, j can be started as soon as its end aligns with the end

Algorithm 4.4: Procedure for finding a feasible outgoing path

```
Procedure findOutPathForNextTask(r):
    $j$ = next task to be processed by $r$;
    if $r$ is sacking-resource then
        return path starting from $r$ and ending in the storage;
    end
    for $p \in$ paths starting from $r$ and ending in a cell of type $cellReq_j$ do
        $c = p.endLocation$;
        if ($j$ has no successor task $\vee$ $c$ has an outgoing path to a resource processing a
          successor task of $j$) $\wedge$ $c$ is currently not delivered by another resource $\wedge$ no path
          conflicting with $p$ is currently active $\wedge$ the potential capacity of $c$ is
          $\geq j.quanProd$ then
            if ($c$ is empty or material contained in $c = j.matProd$) and remaining
              capacity of $c \geq j.quanProd$ then
                return $p$;
            end
            // c is currently not a feasible output cell for j
            $t = calcPotentialStart(c, j)$;
            if $t \leq currentTime$ then
                return $p$;
            else
                insert potential start event for task $j$ into $Q$ at time $t$;
            end
        end
    end
    return $null$;
```

of the task/delivery after which c is a feasible output cell for j. This point of time is calculated by the procedure $calcPotentialStart(c, j)$. Let this point of time be t. If t does not lie in the future, j can be started immediately, since c will be a feasible output cell at the end of j. Otherwise, a potential start event for j is added into Q at time t. At this point of time, $tryToStartNextTask(r)$ will be called again (if j is still the next task to be processed by r).

Whenever starting a task or delivery, an event representing its end is added into the event queue Q at the respective point of time. After the initial step, the construction heuristic starts iterating over Q and handling the events. Start events are handled simply by checking if the respective task/delivery is still satisfied, in which case it is started. When handling a task start event it is additionally necessary to check if the task is still pending, since multiple potential start events for the same task can be added by $findOutPathForNextTask$, where each event marks the potential start of j using a different output cell. At an end event, the content of affected cells and the storage are updated. If the event represents the end of a task j, some deliveries requiring material produced by j may be satisfied now, so each of these deliveries is checked for satisfiability and started if possible. All pending tasks are tried to be started as well, as the end of j may result in less path conflicts or provide material necessary for other tasks. Since deliveries remove material, they may free a cell necessary for the execution of a task, so the pending tasks are checked for satisfiability also at the end of each delivery. Therefore, handling an event may add one or more events into the event queue, if new tasks or deliveries are satisfied. The procedure handling events is described in Algorithm 4.5.

The event queue is processed until either all tasks and deliveries ended, in which case the returned solution can be assumed to be feasible, or if none of the pending tasks and deliveries can be satisfied. In this case, a random pending task is started.[1] Since it is not satisfied, this results in an infeasible solution. Note that the start of a task adds the corresponding end event into Q, which then can be continued processing as before. An overview of the construction heuristic is described in Algorithm 4.6.

Algorithm 4.5: Procedure for handling events

```
Procedure handleEvent(e):
    if e is a potential start event for task j on resource r and j is still pending then
        if time of event e = earliest start of j then
            // earliest start reached
            mark j as not queued;
        end
        tryToStartNextTask(r);
    else if e is potential start event for delivery d then
        mark d as not queued;
        tryToStartDelivery(d);
    else if e is task end event then
        update content of cells/storage;
        for d ∈ deliveries requiring material produced by task do
            tryToStartDelivery(d);
        end
        tryToStartPendingTasks();
    else
        // delivery end event
        update content of cells/storage;
        tryToStartPendingTasks();
    end
```

[1] We can exclude the case that unsatisfied deliveries, but no pending tasks exist, since in a solvable instance the material required by all deliveries must be available after all tasks finished.

Algorithm 4.6: Construction Heuristic

```
1 initialPhase();
2 while $Q \neq \emptyset$ do
3 |   handleEvent(Q.pop());
4 end
5 if pending tasks exist then
  |   // Solution is infeasible, start random unsatisfied task
6 |   $r$ = random resource with pending task $j$;
7 |   start $j$ on $r$ with random incoming and outgoing path;
8 |   go to 2;
9 end
```

4.2 Solution Representation

In contrast to the solution representations used for the exact methods, we do not want to consider every minute as a possible start time of a task or delivery since this would result in an unnecessarily large search space. Therefore, in the solution representation used for Simulated Annealing, we relinquished the time variables and instead introduced the variable set $startEvent$, where $startEvent_i$ indicates which task or delivery starts as ith event (the value of such a variable is either the id of a task or of a delivery, assuming that tasks and deliveries have disjoint domains).

Variable	Domain	Meaning	
$startEvent_i$	$T \cup D$	task/delivery starting at position i	$\forall i \in E$
$pathToTask_i$	P_0	material is transported to task i over this path	$\forall i \in T$
$pathFromTask_i$	P	material is transported from task i over this path	$\forall i \in T$
$cellProvQuan_{i,j}$	$\mathbb{N}$	quantity provided by cell i for delivery j	$\forall i \in C, j \in D$
$cellProvMat_{i,j}$	M	material provided by cell i for delivery j	$\forall i \in C, j \in D$

Table 4.1: Variables used in Simulated Annealing

A solution can now uniquely be identified by the global sequence of start events together with the path assignments specified by the $pathToTask$ and $pathFromTask$ variables and the variable sets $cellProvQuan$ and $cellProvMat$, indicating for each cell the quantity and material provided for each delivery. From these variables, the optimal start times and the respective objective value are calculated by a dispatching algorithm that iterates over the start events in the given order and dispatches the next task/delivery as soon as it is satisfied. The algorithm works similarly to the construction heuristic but has less freedom since all variables except those determining the starting times are fixed and a predefined sequence of start events is given. As in the construction heuristic, after all events have been handled, the next event is started even if it is not satisfied. This allows the dispatching of infeasible solutions.

Dispatching the start times using this method excludes many solutions, but only suboptimal ones: Assuming that the sequence of start events is fixed, in every optimal solution using this sequence a task or delivery is started as soon as it is satisfied. This is because it does not make sense to wait, since no other task or delivery can be started instead due to the predefined sequence.

4.3 Neighborhood operators

We designed and implemented four types of neighborhood operators, also called moves, which operate on the variables specified in Table 4.1. After applying a move, the start times are re-dispatched as described above to calculate the objective value. It is often unnecessary to re-dispatch completely: It might be that all start times happening before a certain point of time t are not affected by the move, but everything happening after t must be re-dispatched. The calculation of this t depends on the applied move and is described in further detail in the following sections.

4.3.1 Task-Path-Change move

This move changes the *out-path* p of a task j, which is the path used to transport the material produced by j. Let j be processed by resource r. Then the new out-path of j is changed to another path q which must be *generally feasible* for j. This means, that q starts from the resource r where j is processed by and ends in a cell c of type required by j. Furthermore, if j has one or multiple successor tasks (requiring the material produced by j) processed by resources $s_1, \dots s_n$, c must be connected to at least one of these resources since the successor task must be able to use the material produced by j. [2]

Depending on the chosen task j, additional changes must be applied. If j has a successor task k processed by resource q, j is called an *intermediate task*. In this case, the path used to transport the material consumed by k must be changed to a path starting from cell c to ensure that the material can be used by k. An example application of such a move is shown in Table 4.1. If j has no successor task, it is called a *final task* and produces material directly required by a delivery d. Then, there are two cases to consider: Either j is processed by a sacking resource r, which implies that there is only one feasible path from r (namely the path to the storage), so the move has no effect. Otherwise, the material produced by j is stored in a loading cell, and the change of the out-path p of j to path q might also change which cell stores the produced material and provides it for delivery d. Therefore, the respective $cellProvMat$ and $cellProvQuan$ variables need to be adapted. A move of this type is illustrated in Table 4.2.

After changing the respective variables, the solution must be re-dispatched since a new path assignment of a task j might enable another task k to start earlier or later (e.g. if the out-path used by k conflicts with the old or new out-path of j, respectively). Although the new path assignment might influence the start times of tasks or deliveries that overlap temporally with j and events

[2] Note that general feasibility for j is a property that does not change during the production process. Therefore, it is weaker than the property of being a feasible outgoing path for j (as defined in Section 4.1) since the latter also depends on the point of time by considering inventory and conflict constraints.

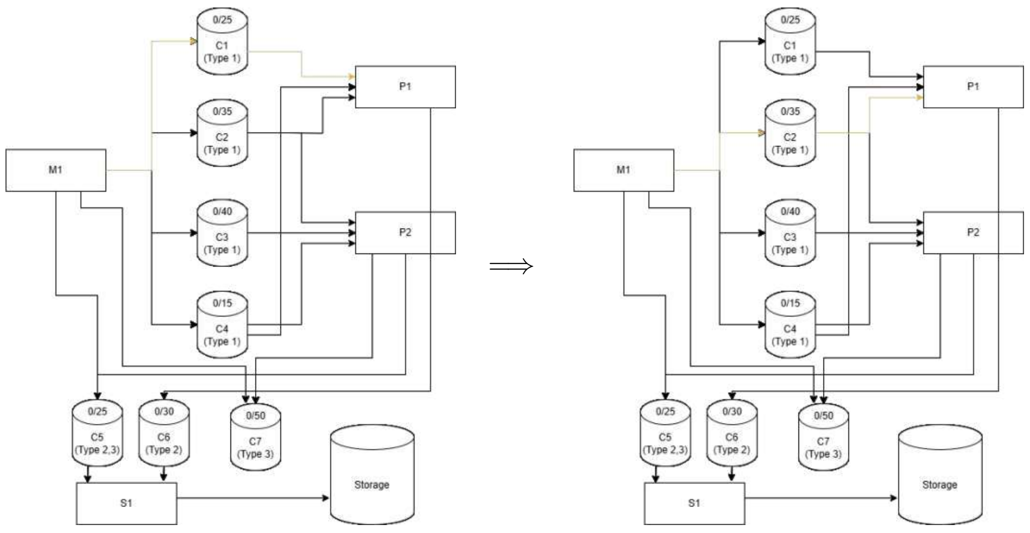


Figure 4.1: Changing the out-path of an intermediate task

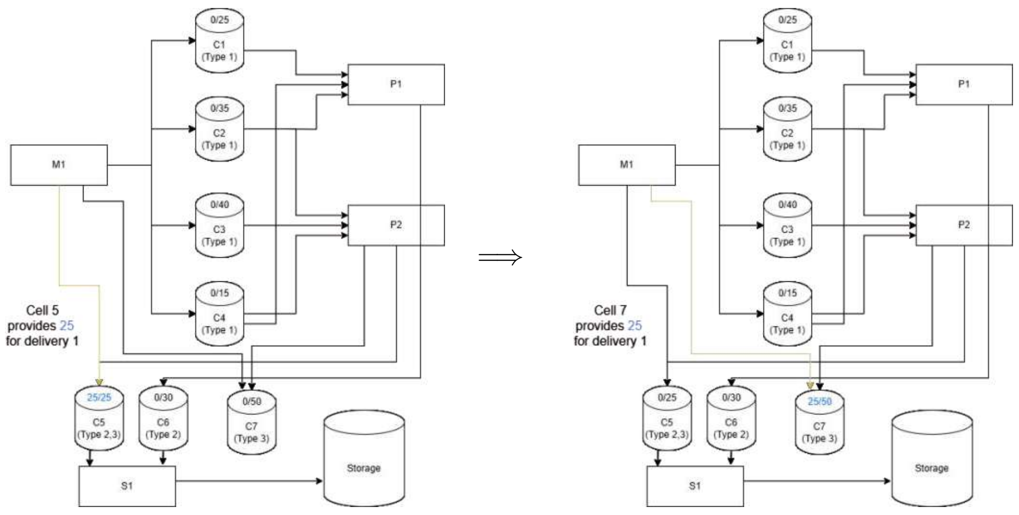


Figure 4.2: Changing the out-path of a final task

happening later, all tasks and deliveries ending before the start of j cannot be influenced. For any delivery d, even a more relaxed condition holds, since its start time can only be influenced indirectly by the change of start times of tasks producing material required by d. Therefore it is sufficient to compute the earliest starting time t of all tasks overlapping with j, and keep all start times happening before t. The remaining solution must be re-dispatched.

4.3.2 Job-Path-Change move

In general, it makes sense to use the same path for all tasks executing the same job since they produce the same material, and occupying multiple cells with one material is often not necessary. Therefore we introduced a compound move changing the out-path of all tasks $j_1 \dots j_n$ executing the same job on the same resource (since otherwise no common path exists) to a path q generally feasible for all these tasks (which is identical to being generally feasible for any of these tasks).

Additional changes concerning the $pathToTask$, $cellProvMat$, and $cellProvQuan$ variables of the affected successor tasks and deliveries are applied, as described in Section 4.3.1, for each move individually. In contrast, re-dispatching the solution is only necessary for calculating the objective value and therefore carried out after all moves have been applied.

Let j_1 be the earliest task of $j_1 \dots j_n$. Then, as argued in Section 4.3.1, the start times of tasks and deliveries ending before the start of j_1 are not affected, but the remaining solution must be re-dispatched.

4.3.3 Reinsert move

Given the sequence of start events specified by the $startEvent$ variables, this move removes an event e and re-inserts it at another point in the sequence. One scenario where this move can be useful is if there are two satisfied tasks on different resources, but only one of them can be started because they would have to consume material from the same cell. In this case the construction heuristic randomly starts one of these jobs, which might result in a suboptimal solution. Such decisions can be reverted by changing the order of start events.

An example-application of a reinsert move is illustrated in Figure 4.3: Assume that the tasks J_1 and J_2 both require the material contained in C_2 and are processed by P_1 and P_2 respectively. In the solution illustrated on the left, J_1 starts before J_2. By reinserting J_2 before J_1, the solution on the right is obtained.

Let e represent the start of the i-th task processed by resource r. Then e must be inserted between the start events of the tasks processed by resource r on position $i-1$ and $i+1$.[3] This is because the resource-internal sequence is fixed and must not be violated. A sequence of start events satisfying the resource-internal sequences is called a *resource-feasible* sequence. Although only allowing such sequences restricts the diversity of single moves, it is still possible to explore all resource-feasible sequences just by applying moves of this type. This can be done by iteratively reinserting the first task processed by r at the start of the sequence and the last one at the end, the second task after the first one, and so on.

Since the dispatching algorithm iterates over the start events, all events happening before the new position i of the reinserted event cannot be affected by a move of this type. It follows that the solution must be re-dispatched from that point on only.

[3]Two exceptions are if $i = 1$ or i is equal to the number of tasks processed by r. In this case, there is no bound in the direction of the beginning or the end of the sequence, respectively.

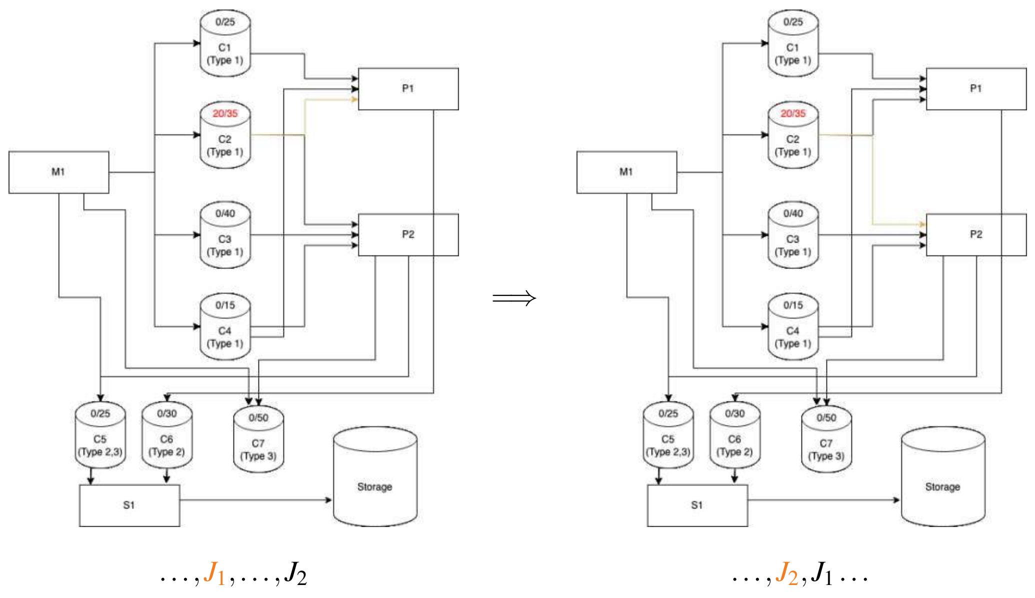


Figure 4.3: Reinserting J_1 after J_2

4.3.4 Swap move

This move simply swaps a start event e_1 with its successor e_2 in the sequence of start events. As explained in Section 4.3.3, violations of resource-internal sequences are not permitted, which means that a swap is only allowed if e_1 and e_2 are not start events of tasks processed by the same resource. The motivation for this move comes from the observation that in some cases optimal start event sequences can be obtained from sequences returned by the construction heuristic by applying a small number of moves of this type.

By applying such a move, the start times happening before e_1 and e_2 cannot be affected. Therefore it is sufficient to re-dispatch the sequence from e_2 on (which happens before e_1 in the modified solution).

4.4 Modified Solution Cost

As already pointed out, the construction heuristic might return an infeasible solution. Since Simulated Annealing must be able to evaluate and compare such solutions, it is not sufficient just to use tardiness as the objective to be minimized as this could result in preferring infeasible solutions with lower tardiness over feasible ones. Therefore, we incorporated the number of violated hard constraints into the solution cost considered by Simulated Annealing.

To ensure that infeasible solutions are evaluated worse than feasible ones independent of their tardiness, the penalty M for hard constraint violations must be chosen large enough. Thus, we considered M to be an instance-dependent upper-bound on the tardiness by assuming that every

Violation	Violated Constraint(s)
Two active paths conflict (each conflicting pair is penalized with M)	Constraint 1.6
Two active paths use the same input-cell	Constraint 1.7
Two active paths use the same output-cell	Constraint 1.8
At the start of a task its input cell does not contain the material required	Constraint 1.18
At the start of a task its input cell contains too little potential quantity	Constraint 1.18
At the end of a task, its output cell contains wrong material	Constraint 1.26
At the end of a task, its output cell does not have enough remaining capacity	Constraints 1.23, 1.27
At the start of a delivery, a providing cell does not contain correct material	Constraint 1.19
At the start of a delivery, a providing cell contains too little potential quantity	Constraint 1.20
At the start of a sack delivery, the storage contains too little quantity of some required material	Constraints 1.24, 1.28

Table 4.2: Potentially violated hard constraints

delivery order starts at the latest possible point of time, which is the scheduling horizon. It follows that all feasible solutions have solution cost $\leq M$.

The violations potentially happening are listed in Table 4.2. Each violation adds a factor of M to the solution cost. The table includes references to formal definitions of the violated constraints based on the constraint programming model **M1**. This is to provide a better understanding and should not be interpreted too accurately.

4.5 Simulated Annealing

Simulated Annealing is a metaheuristic proposed by [KGV83]. It is a variant of local search, inspired by the cooling process of annealing in solids. Starting from an initial solution, e.g. a solution provided by the construction heuristic or a randomly generated one, the current solution is iteratively modified using a set of moves. A random move is applied to the current solution, and if it does not worsen the solution, the move is accepted, i.e. the modified solution becomes the current solution. If the move brings no improvement, it is still accepted with a certain probability, which correlates with the so-called *temperature*. The temperature initially is high (specified by T_{max}) and decreases after every iteration by a cooling rate α. Thus, at the beginning many non-improving moves are accepted, but as the algorithm converges, it becomes more similar to a Hill-Climber, which is a local-search technique for finding local optima by only accepting improving moves. The idea behind Simulated Annealing is to escape local optima by initially exploring a large amount of the search space and doing the fine-grained optimization at the end.

We considered a variant of Simulated Annealing which adapts the cooling rate s.t. a desired minimum temperature T_{min} is reached approximately when a certain time l elapsed, where l

Algorithm 4.7: Simulated Annealing

```
Procedure simulatedAnnealing(initialSolution):
    $currentSolution = initialSolution$;
    $bestSolution = initialSolution$;
    $T_{current} = T_{max}$;
    $i = 0$;
    while time limit not reached do
        $newSolution = randomNeighbour(currentSolution)$;
        if $accept(newSolution)$ then
            $currentSolution = newSolution$;
            if currentSolution is better than bestSolution then
                $bestSolution = currentSolution$;
            end
        end
        $i++$;
        $r = \frac{i \cdot (l - elapsedTime)}{elapsedTime}$;
        $\alpha = \sqrt[r]{T_{min}/T_{current}}$;
        $T_{current} = T_{current} \cdot \alpha$
    end
```

represents the time limit. To achieve this, the number of iterations is counted by a counter i and after every iteration, the number r of remaining iterations is estimated. The cooling rate α can then be calculated s.t. it behaves as intended by the formula $\sqrt[r]{T_{min}/T_{current}}$, where $T_{current}$ represents the current temperature. The pseudo-code for this variant of Simulated Annealing is specified in Algorithm 4.7.

The function $randomNeighbour(s)$ applies one of the moves specified in Section 4.3 on the given solution s and returns the resulting solution s'. As the name indicates, selecting the applied move is a probabilistic procedure, which works as follows: First, the type of move is determined. A probability is given for each of the 4 types considered, s.t. the 4 probabilities sum up to 1. The move type can therefore be selected by a roulette wheel procedure using these probabilities. Thereafter, the move-specific parameters are determined. These depend on the move type and are specified in Table 4.3. Within the feasible parameter choices[4] the move-specific parameters are selected uniformly at random.

The acceptance criterion for moves is represented by the *accept*-function. As already pointed out, if the new solution (i.e. the solution after the move) is equal to or better than the current solution, it is accepted in any case. Otherwise, it is accepted with probability $e^{-\delta/T_{current}}$, where δ is the cost of the new solution minus the cost of the current solution. Thus, δ is positive iff the considered move worsens the current solution. The acceptance probability depends on δ in such a way that more worsening moves are accepted less likely than moves changing the cost

[4]see Section 4.3 for more details

Move type	Parameter	Domain	Meaning
Task-Path-Change	t	T	task which changes its out-path
	p	P	new out-path for t
Job-Path-Change	j	J	job indicating set of affected tasks
	r	R	resource indicating set of affected tasks
	p	P	new out-path for all tasks executing j on r
Reinsert	e	E	reinserted start event
	i	E	new position of event e
Swap	e	E	event which is swapped with its successor event

Table 4.3: Move-specific Parameters

only by a small degree. Furthermore, the probability of accepting a move depends on the current temperature, where a higher temperature results in a higher probability.

CHAPTER 5

Experiments

To evaluate the various approaches, a large set of instances is needed. Therefore, a random instance generator is proposed and the properties of the generated instances are listed. Hereafter, the automated tuning procedure to determine the parameters for Simulated Annealing is outlined and the best-evaluated configurations are presented. Finally, the experimental setup is described and an overview of the results is provided.

5.1 Instances

We were provided one real-life instance by our industrial partner. For a systematic evaluation of the various solving approaches, a larger set of instances was required. To accomplish that, we developed an instance generator based on real-life instances.

5.1.1 Instance Generator

The generator works as follows: From a given real-life instance, a fraction of delivery orders is deleted randomly, which also decreases the number of materials required by such. The respective materials might still be required by tasks, so in a consecutive step, all tasks not contributing to any remaining delivery are removed. A task *contributes* to a delivery d if it produces material directly required by d or if it has a successor task contributing to d. Thus, exactly the tasks producing material that is either directly or indirectly required by some remaining delivery are kept. The resulting instance might now contain resources with an empty task sequence, which can be removed. This makes all cells that are only connected to such resources unusable, so they can also be deleted, together with the respective paths. Finally, all materials that are neither required by any task nor delivery are removed together with all cells initially containing such material.

The instances generated using the described method turned out to have a simple structure: For all instances that were small enough to be solved optimally by one of the exact methods, the construction heuristic found the optimal solution as well.[1] One possible explanation for this is that in the realistic instance provided, every job produces and consumes a unique material and it is never the case that tasks executing the same job are processed by different resources. Thus, for all intermediate tasks producing the same material, there is only one feasible successor resource, which must be reachable from the out-cells chosen by these tasks to yield a feasible solution. Since the construction heuristic only considers cells satisfying this criterion, this might be sufficient to solve the provided instances optimally.

Therefore, we modified the generator to increase the structural complexity of generated instances, while maintaining the reduced size to keep the instances solvable by the exact methods. To accomplish that, the instance generator re-assigns the tasks over all feasible resources and shuffles the resource sequences. This makes the instances harder to solve, in the sense that there is a much broader choice of feasible intermediate cells. However, there is no guarantee that the instances generated in this way are still solvable since it might be that the cells reachable from the new resources executing a task lack the required capacity.

Model **M2** requires additional criteria to be satisfied by instances to be applicable. In Section 3.2.1 we formulated these criteria, which regarded the structural complexity of conflicts and the maximum number of paths between the same resource and cell. We argued that these assumptions are satisfied by real-life instances. It is important to note that the instance generator does only remove parts of the given network. Since the generation is based on a real-life instance, the generated instances satisfy the assumptions as well. This makes **M2** applicable to these instances.

5.1.2 Generated Instances

We generated 160 instances which can be divided into 4 different size categories, containing 40 instances each. The instances of one category share the same number of delivery orders, which is between 1 and 4, depending on the category. It must be noted that the number of resources, tasks, etc. may still have high variance within one category since they depend on which deliveries are remaining and the materials required by them. The size range of the categories can be seen in Table 5.1.

Category	#Deliveries	#Resources	#Cells	#Paths	#Materials	#Jobs	#Tasks
1	1	1-4	22-36	30-116	1-2	1-3	2-20
2	2	1-5	22-37	30-129	1-3	2-5	4-26
3	3	1-5	23-37	32-129	2-4	3-7	6-34
4	4	2-5	31-37	58-133	3-5	5-9	11-39

Table 5.1: Size categories of generated instances

These instances were divided into a set of 60 (15 of each category) *tuning* and 100 (25 of each category) *testing* instances. The tuning instances were used to tune the parameters of Simulated

[1]This was only the case for preliminary instances, not for the instances used for evaluation, which were constructed by the modified generator.

Annealing (see Section 5.2). The final evaluation of the exact and heuristic methods (outlined in Section 5.4) was performed on the testing instances.

5.2 Tuning

Multiple parameters appear in the Simulated Annealing algorithm. They can have a major impact on the solving process and therefore also on the quality of the solution returned. Since the choice of these parameters is not straightforward, we performed systematic parameter tuning using irace, a software package proposed by [LIDLP$^+$16] for automatic algorithm configuration by iterated racing procedures.

Parameter	Possible Values	Default
T_{max}	$[1000, 100000]$	10000
T_{min}	$\{10^{-3}, 10^{-2}, 10^{-1}, 1\}$	10^{-2}
$p_{Task-Path-Change}$	$\{0, 10, 20, 30, 40, 50\}$	30
$p_{Job-Path-Change}$	$\{0, 10, 20, 30, 40, 50\}$	30
$p_{Reinsert}$	$\{0, 10, 20, 30, 40, 50\}$	20
p_{Swap}	$\{0, 10, 20, 30, 40, 50\}$	20

Table 5.2: Tuning parameters

The variant of Simulated Annealing applied by us makes use of the parameters T_{max} and T_{min}, indicating the initial and final temperature. Meaningful ranges for these parameters depend on the solution costs appearing and are hard to determine beforehand. Thus, a broad value range is chosen. Note that for the initial temperature, each integer between 1000 and 100000 is allowed. The other parameters appearing are the probabilities p_t for applying a move of type t (specified for each of the four move types). Since it is not possible to explore the whole parameter space just by applying one type of move, as it might be necessary to change path assignments and the global sequence of events, we only considered probabilities up to 50 percent. An additional constraint concerning the probability values is that they must sum up to 100. The parameters to be tuned, together with their value ranges and default values[2] can be seen in Table 5.2.

We fixed a tuning budget of 60000 runs of Simulated Annealing with a time limit of one minute each. In each run, one of the 60 tuning instances was tried to be solved. The superior configurations according to our results are listed in Table 5.3. It is noticeable that the probability for the swap move equals 0 in the top 3 configurations. A possible explanation for this is that every swap move can be simulated by a reinsert move.

[2]When using irace, it is recommended to specify default values for the parameters to be tuned.

T_{max}	T_{min}	$p_{Task-Path-Change}$	$p_{Job-Path-Change}$	p_{Swap}	$p_{Reinsert}$
64637	**1**	**10**	**40**	**0**	**50**
87573	1	30	40	0	30
72333	1	10	40	0	50
86193	10^{-1}	30	40	10	20

Table 5.3: Best configurations returned by irace, from top to bottom

5.3 Experimental Setup

All experiments (including tuning) were performed on a computing cluster with 13 nodes, each featuring two Intel Xeon E5-2650 v4 CPUs (12 cores @ 2.20GHz).

The models **M1** and **M2** described in Chapter 3 were implemented in the solver-independent constraint modeling language MiniZinc v2.8.5 [NSB+07]. We performed preliminary experiments by evaluating the developed models using different solvers. From the solvers tested by us (Gecode v6.3.0 [Gec19], Chuffed v0.13.1 [GC23], Gurobi v10.0.1 [Gur23], CP-Sat v9.10 [PD24]), both models achieved best results when solved with CP-Sat. Thus, this solver was chosen for further evaluation. Both models were evaluated on each of the 100 testing instances using 8 cores with a time limit of 1 hour per model and instance.

On each testing instance, 10 runs of Simulated Annealing were performed, starting from the solution provided by the construction heuristic. We used the best parameter configuration found by irace in the tuning phase (see Table 5.3). Each run was given a time limit of one minute.

5.4 Results

An overview of the results is provided in tables 5.4 - 5.6. The instance names are encoded as **set**.**number**, where **set** indicates the size category (see Table 5.1) and **number** is a unique id within the set. It must be noted that some instances are not listed. This means that all 4 algorithms found the optimal solution.

The four values listed in the table section *Objective Value* indicate the objective value, i.e. the total tardiness, of the solutions found by constraint programming models **M1** and **M2**, the construction heuristic (CH) and Simulated Annealing (SA), respectively. The best objective values of the feasible solutions provided are formatted in bold. Since the exact methods do not always return a solution in time, the symbol **-** represents that no solution was found. If a constraint programming model proved the unsatisfiability of an instance, this is denoted as UNSAT. In contrast to the constraint programming models, the heuristics sometimes returned an infeasible solution. In this case, the number of hard constraint violations is specified instead of the objective. The minimum objective value (or number of constraint violations, if all solutions were infeasible) found over the 10 runs is denoted for Simulated Annealing.

Furthermore, the average (AvgObj), standard deviation (StdDev) and average deviation (AvgDev) of the objective values of feasible instances found in the 10 runs of Simulated Annealing are specified. Additionally, the number of feasible solutions returned (feas) and the average number of hard constraint violations (AvgVio) are denoted.

	Objective Value				Simulated Annealing				
Instance	M1	M2	CH	SA	AvgObj	StdDev	AvgDev	feas	AvgVio
1.2	**284**	**284**	2 vio	**284**	284.00	0.00	0.00	10	0.0
1.4	**204**	**204**	6 vio	**204**	204.30	0.90	0.54	10	0.0
1.11	**284**	**284**	2 vio	**284**	284.00	0.00	0.00	10	0.0
1.12	**341**	**341**	9 vio	**341**	341.60	1.20	0.96	10	0.0
1.17	**341**	**341**	9 vio	**341**	343.40	3.23	2.88	10	0.0
1.18	**98**	**98**	1 vio	**98**	98.00	0.00	0.00	10	0.0
1.19	**135**	**135**	2 vio	**135**	135.00	0.00	0.00	10	0.0
1.20	**40**	**40**	1 vio	**40**	40.00	0.00	0.00	10	0.0
1.21	**135**	**135**	2 vio	**135**	135.00	0.00	0.00	10	0.0
1.22	**68**	**68**	1 vio	**68**	68.00	0.00	0.00	10	0.0
1.23	**328**	**328**	3 vio	**328**	328.00	0.00	0.00	10	0.0
2.1	**55**	**55**	1 vio	**55**	55.00	0.00	0.00	10	0.0
2.3	**118**	**118**	5 vio	1 vio	-	-	-	0	1.0
2.4	-	-	6 vio	**1962**	2086.14	140.88	120.78	7	0.5
2.5	-	-	5 vio	**2483**	2488.40	5.97	4.20	10	0.0
2.6	-	**616**	4 vio	**616**	677.13	91.36	62.69	8	0.7
2.7	**231**	**231**	1 vio	**231**	231.00	0.00	0.00	10	0.0
2.8	**229**	**229**	5 vio	**229**	244.50	15.95	15.60	10	0.0
2.11	-	-	1 vio	**635**	635.00	0.00	0.00	10	0.0
2.12	**327**	**327**	5 vio	**327**	336.10	25.33	15.18	10	0.0
2.14	-	**164**	**164**	**164**	164.00	0.00	0.00	10	0.0
2.16	-	-	5 vio	**469**	485.50	16.50	16.50	4	0.6
2.17	-	-	4 vio	**764**	767.90	3.30	2.88	10	0.0
2.21	**104**	**104**	1 vio	**104**	104.00	0.00	0.00	10	0.0
2.24	**31**	**31**	2 vio	**31**	31.00	0.00	0.00	10	0.0
2.25	-	**568**	4 vio	**568**	568.60	1.20	0.96	10	0.0

Table 5.4: Results of Instance Sets 1 and 2

In the experiments we performed, all solutions found by the exact methods are optimal, i.e. they either returned the optimal solution together with an optimality proof or no solution at all. This is interesting since in general, the CP-Sat solver might also return suboptimal solutions. One possible explanation for this phenomenon is that optimizing a feasible solution and proving its optimality might be relatively easy compared to finding one.

	Objective Value				Simulated Annealing				
Instance	M1	M2	CH	SA	AvgObj	StdDev	AvgDev	feas	AvgVio
3.1	-	-	9 vio	1 vio	-	-	-	0	1.9
3.2	-	-	**817**	**817**	817	0	0	10	0.0
3.3	-	UNSAT	8 vio	2 vio	-	-	-	0	2.3
3.6	**202**	**202**	1 vio	**202**	207.6	5.5	3.08	10	0.0
3.7	-	UNSAT	16 vio	2 vio	-	-	-	0	3.2
3.8	**699**	**699**	1 vio	**699**	699	0	0	10	0.0
3.9	-	**575**	6 vio	**575**	796.67	89.34	69.48	9	0.1
3.11	-	-	1 vio	**749**	771.1	27.08	26.52	10	0.0
3.12	-	UNSAT	5 vio	2 vio	-	-	-	0	2.2
3.13	-	-	2 vio	**290**	293	9	5.4	10	0.0
3.14	-	-	10 vio	**3092**	3092	0	0	1	2.1
3.15	-	-	**539**	**539**	539	0	0	10	0.0
3.16	-	-	11 vio	1 vio	-	-	-	0	2.4
3.17	**206**	**206**	295	295	295	0	0	10	0.0
3.18	-	-	1 vio	**939**	939	0	0	10	0.0
3.19	-	-	**382**	**382**	382	0	0	10	0.0
3.20	-	-	6 vio	**1493**	1496.3	4.54	3.48	10	0.0
3.21	-	**396**	1 vio	**396**	432	46.76	42	5	0.5
3.22	**140**	**140**	180	168	175.3	5.76	5.64	10	0.0
3.23	-	-	10 vio	1 vio	-	-	-	0	2.2
3.24	-	UNSAT	6 vio	2 vio	-	-	-	0	2.0
3.25	-	**809**	**809**	**809**	809	0	0	10	0.0

Table 5.5: Results of Instance Set 3

As one can see, both constraint programming models found the optimal solution for all instances of instance set 1 (since the instances solved optimally by all approaches are not listed). In contrast, for multiple larger instances evaluated, **M2** found a solution in time, while **M1** did not. **M2** provided unsatisfiability proofs for 8 instances, whereas **M1** could only prove the unsatisfiability of instance 4.10.

It is noticeable that for most of the instances where the optimum is known[3], the construction heuristic either found the optimal solution or an infeasible one. In the two cases where it is known that the construction heuristic returned a feasible and sub-optimal solution (instances 3.17 and 3.22), Simulated Annealing could not find the optimal solution as well.

While the construction heuristic returned feasible (and optimal) solutions for approximately half of the instances from sets 1 and 2 (which are mostly not listed), its success rate became smaller

[3]Since we generated the instances randomly, the optimal solution for an instance is known only if one of the constraint programming models found a solution.

	Objective Value				Simulated Annealing				
Instance	M1	M2	CH	SA	AvgObj	StdDev	AvgDev	feas	AvgVio
4.1	-	-	2 vio	**1295**	1296.50	2.01	1.80	10	0.0
4.2	-	-	**4169**	**4169**	4169.00	0.00	0.00	10	0.0
4.3	-	-	3 vio	**1596**	1702.14	66.72	62.41	7	0.5
4.4	-	**418**	1 vio	1 vio	-	-	-	0	1.0
4.5	-	**435**	3 vio	**435**	449.30	23.46	20.02	10	0.0
4.6	-	**1142**	**1142**	**1142**	1142.00	0.00	0.00	10	0.0
4.7	-	UNSAT	22 vio	8 vio	-	-	-	0	9.4
4.9	-	UNSAT	2 vio	1 vio	-	-	-	0	1.0
4.10	UNSAT	UNSAT	2 vio	2 vio	-	-	-	0	2.0
4.11	-	**1930**	2 vio	1 vio	-	-	-	0	1.2
4.12	-	-	1 vio	**853**	1056.57	496.20	347.27	7	0.3
4.13	-	-	**1291**	**1291**	1291.00	0.00	0.00	10	0.0
4.14	-	-	1 vio	**1494**	1495.50	1.50	1.50	2	0.8
4.15	-	-	**751**	**751**	751.00	0.00	0.00	10	0.0
4.16	-	-	2 vio	**498**	498.00	0.00	0.00	4	0.6
4.18	-	-	8 vio	**2926**	3076.60	176.39	158.04	10	0.0
4.20	-	UNSAT	10 vio	2 vio	-	-	-	0	3.0
4.21	-	-	12 vio	4 vio	-	-	-	0	4.6
4.22	-	-	**1528**	**1528**	1528.00	0.00	0.00	10	0.0
4.24	-	-	**647**	**647**	647.00	0.00	0.00	10	0.0
4.25	-	-	**1996**	**1996**	1996.00	0.00	0.00	10	0.0

Table 5.6: Results of Instance Set 4

when solving larger instances. Simulated Annealing was able to improve suboptimal solutions of the construction heuristic for most of the instances, at least through minimizing the number of constraint violations. Known exceptions are instances 3.17 and 4.4. For some instances that were not improved by Simulated Annealing, we do not know if the solution provided by the construction heuristic was already optimal or not. For instances where the optimum is known, Simulated Annealing was able to find optimal solutions in most of the cases. Exceptions are instances 2.3, 3.17, 3.22, 4.4 and 4.11.

We showed that the improvement performed by Simulated Annealing compared to the construction heuristic is statistically significant, by applying the Wilcoxon signed-rank test, with the costs of initial solutions provided by the construction heuristic and the average costs of solutions found by Simulated Annealing, over all instances. Here we considered the solution cost with incorporated constraint violations since this was the main difference between the solutions returned by the two approaches. We formulated the null hypothesis that the costs of solutions provided by the construction heuristic are not significantly higher. The hypothesis was rejected with a p-value of less than 0.001, which means that the probability of this is very low.

The improvement performed by the only successful run of Simulated Annealing on instance 3.14 is illustrated in Figure 5.1. The x-axis shows the time passed in seconds, while the y-axis depicts the solution cost including penalized constraint violations. A dot represents the solution cost of a new optimal solution found at a specific point of time. The plot on the right shows only the time fragment after a feasible solution has been found.

Comparing model **M2** with Simulated Annealing, it can be seen that Simulated Annealing was able to find more feasible solutions. This holds especially for the larger instances in our experiments. However, no clear dominance of one of these approaches regarding solution quality is evident.

We chose a subset of instances to compare the time required to find optimal solutions by **M2** and Simulated Annealing. We analyzed this on the set of instances solved by **M2** and every run of Simulated Annealing optimally, although the construction heuristic returned an infeasible solution. Furthermore, a few outliers where **M2** needed 5-10 minutes to find optimal solutions were not considered. These instances can be interpreted as the subset of instances where both approaches performed similarly. Two boxplots illustrating the required time in seconds for solving the respective set of instances optimally by **M2** and the individual runs of Simulated Annealing are provided in 5.2. It can be seen that on average, Simulated Annealing is faster in finding optimal solutions. However, it must be noted that it is not always successful in solving instances optimally and that in contrast to the exact methods, no optimality proof is provided by Simulated Annealing.

Additionally, we tested the developed approaches on the real-life instance provided by our industrial partner with increased time limits of multiple hours. Due to the large instance size, none of the exact methods was able to find solutions. In contrast, the construction heuristic successfully solved the real-life instance. No improvement was performed by applying Simulated Annealing on the resulting solution.

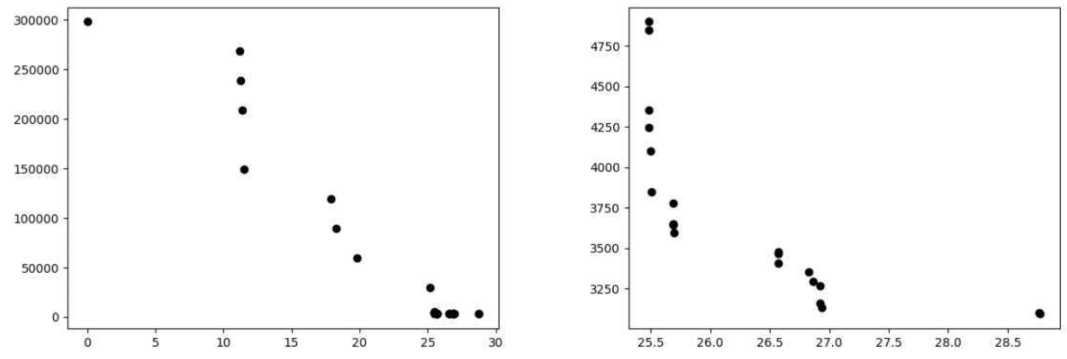


Figure 5.1: Decrease of solution cost over time in seconds during a successful Simulated Annealing run on instance 3.14

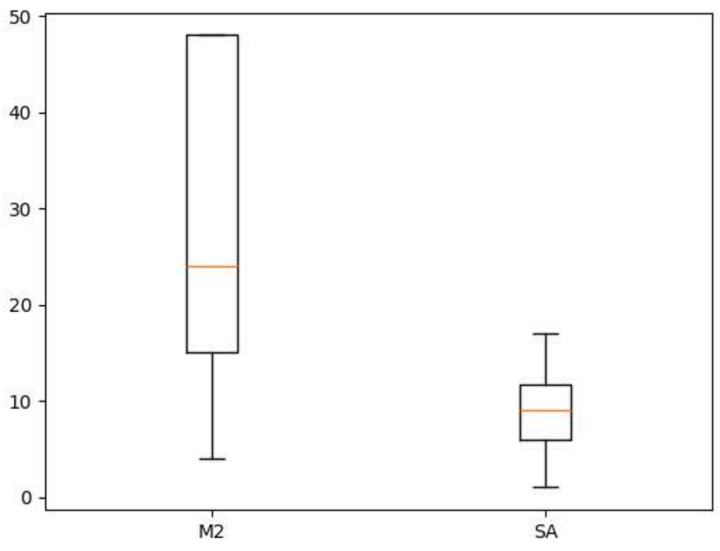


Figure 5.2: Comparison of the time in seconds to reach optimal solutions on a selected subset of instances

CHAPTER 6

Conclusion

This thesis tackled a challenging real-life problem which consists of scheduling jobs in a production network and assigning cells as storage facilities. A formal problem specification has been provided and various solving approaches have been developed. These include two constraint models that can be used as exact approaches with the potential to construct optimal solutions for the considered problem. Furthermore, a construction heuristic and a novel metaheuristic approach have been designed to find high-quality solutions quickly.

The first exact method considered is constraint programming based on the high-level constraint programming model **M1**. By reformulating the representation of path conflicts and deliveries and by introducing additional predecessor variables for events to decrease the number of generated constraints, the model **M2** is provided, which can be used as an alternative constraint programming model.

Additionally, a Simulated Annealing-based local search procedure has been applied to the problem as a metaheuristic solution approach. To accomplish this, first, a construction heuristic for the generation of initial solutions has been developed. Furthermore, a solution representation for local search and four novel types of moves to modify existing solutions have been designed.

Since only one real-life instance was available to us, the need for artificial instances emerged. Therefore, a randomized instance generator was implemented, and 160 instances were created. These instances were used to perform extensive parameter tuning and test the quality of solutions provided by the different approaches.

Experimental results showed that exact solvers based on constraint programming model **M1** or **M2** are able to solve the majority of the evaluated smaller instances optimally. It turned out that the model **M2** performed better than the high-level model **M1** on some of the larger instances evaluated. While the construction heuristic sometimes constructed good solutions even for larger instances, where none of the exact methods could find solutions in time, it often achieved worse results than the exact methods when solving smaller instances. Simulated Annealing could improve infeasible solutions provided by the construction heuristic in most of the cases. The majority of them were repaired and for some other solutions, at least the number of violated hard constraints could be lowered. If an instance was known to be solvable, feasible solutions were found by Simulated Annealing with a few exceptions. The cost of these solutions matched the optimal cost for most of the instances where the optimum is known. Although the real-life instance was too large to be solved by the exact approaches, the construction heuristic could successfully be used to provide a feasible solution.

Many of the constraints formulated in the constraint models refer to job intervals. Therefore, possible future work could consider the development of a model using interval variables, which may allow more efficient formulations of such constraints. Furthermore, it would be interesting to investigate more neighborhood operators, which could, for example, change path assignments and the order of events within one move.

Overview of Generative AI Tools Used

No generative AI tools were used while working on this thesis.

List of Figures

List of Tables

List of Algorithms

Acronyms

CAP Cell Assignment Problem. 1

Bibliography

[ANM14] Behrouz Afshar-Nadjafi and Mahyar Majlesi. Resource constrained project scheduling problem with setup times after preemptive processes. *Computers & Chemical Engineering*, 69:16–25, 2014.

[BGZ11] Jan-Hendrik Bartels, Thorsten Gather, and Jürgen Zimmermann. Dismantling of nuclear power plants at optimal npv. *Annals of Operations Research*, 186: 407–427, 2011.

[BY10] Tyson R. Browning and Ali A. Yassine. Resource-constrained multi-project scheduling: Priority rule performance revisited. *International Journal of Production Economics*, 126(2):212–228, 2010. ISSN 0925-5273. URL `https://www.sciencedirect.com/science/article/pii/S0925527310000915`.

[FTSCC06] T. C. Wong F. T. S. Chan and L. Y. Chan. Flexible job-shop scheduling problem under resource constraints. *International Journal of Production Research*, 44(11): 2071–2089, 2006.

[Fu14] Fang Fu. Integrated scheduling and batch ordering for construction project. *Applied Mathematical Modelling*, 38(2):784–797, 2014.

[GC23] Andreas Schutt Thorsten Ehlers Graeme Gange Kathryn Francis Geoffrey Chu, Peter J. Stuckey. Chuffed, a lazy clause generation solver (v0.13.1), 2023. URL `https://github.com/chuffed/chuffed?tab=readme-ov-file#chuffed-a-lazy-clause-generation-solver`.

[Gec19] Gecode Team. Gecode: Generic constraint development environment (v6.3.0), 2019. URL `http://www.gecode.org`.

[Gur23] Gurobi Optimization, LLC. Gurobi Optimizer (v10.0.1), 2023. URL `https://www.gurobi.com`.

[HB10] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, 2010. ISSN

0377-2217. URL https://www.sciencedirect.com/science/article/pii/S0377221709008558.

[HB22] Sönke Hartmann and Dirk Briskorn. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 297(1):1–14, 2022. ISSN 0377-2217. URL https://www.sciencedirect.com/science/article/pii/S0377221721003982.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. URL https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[LIDLP+16] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016. ISSN 2214-7160. URL https://www.sciencedirect.com/science/article/pii/S2214716015300270.

[LVDVDC19] Pieter Leyman, Niels Van Driessche, Mario Vanhoucke, and Patrick De Causmaecker. The impact of solution representations on heuristic net present value optimization in discrete time/cost trade-off project scheduling with multiple cash flow and payment models. *Computers and Operations Research*, 103:184 – 197, 2019. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85056879339&doi=10.1016%2fj.cor.2018.11.011&partnerID=40&md5=936401c77e1ebf2651d4aed67c53e667. Cited by: 27; All Open Access, Green Open Access.

[MKKT18] D.G. Mogale, Mukesh Kumar, Sri Krishna Kumar, and Manoj Kumar Tiwari. Grain silo location-allocation problem with dwell time for optimization of food grain supply chain network. *Transportation Research Part E: Logistics and Transportation Review*, 111:40–69, 2018. ISSN 1366-5545. URL https://www.sciencedirect.com/science/article/pii/S1366554517306361.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-74970-7.

[PD24] Laurent Perron and Frédéric Didier. Cp-sat (v9.10), 2024. URL https://developers.google.com/optimization/cp/cp_solver/.

[WLZ20] Qiang Wang, Changchun Liu, and Li Zheng. A column-generation-based algorithm for a resource-constrained project scheduling problem with a fractional shared resource. *Engineering Optimization*, 2020.

[XSRH22] Hegen Xiong, Shuangyuan Shi, Danni Ren, and Jinjin Hu. A survey of job shop scheduling problem: The types and models. *Computers Operations Research*, 142:105731, 2022. ISSN 0305-0548. URL `https://www.sciencedirect.com/science/article/pii/S0305054822000338`.

[ZSN17] Nima Zoraghi, Aria Shahsavar, and Seyed Taghi Akhavan Niaki. A hybrid project scheduling and material ordering problem: Modeling and solution algorithms. *Applied soft computing*, 58:700–713, 2017.