MASTERARBEIT / MASTER’S THESIS

Titel der Masterarbeit / Title of the Master‘s Thesis

„A block relocation algorithm for the steel slab stacking problem using heavy-duty vehicles“

verfasst von / submitted by

Iryna Pidsadochna, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066915
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Betriebswirtschaft UG2002
Betreut von / Supervisor:	Univ.-Prof. Mag. Dr. Karl Franz Dörner

Acknowledgement

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

Abstract

Optimization of warehouse operations is a key aspect of improving the cost-effectiveness and manageability of a storage facility. An example of a real-life situation where an efficient and automated way of stacking and retrieving objects is sought is a steel slab storage. Given the data from an industrial partner, we defined the Steel Slab Stacking Problem (SSSP) as a generalization of the Block Relocation Problem (BRP). The particularities of SSSP, as opposed to BRP, are batch moves, heterogenous items, movement restrictions, incompatibility restrictions and the retrieval procedure. We analyzed numerous BRP solution methods from the related literature and concluded that SSSP requires a special approach which takes into account its particularities, while also applying some established rules for solving BRP. Thus, a new algorithm for SSSP named Combined Multi-Target Algorithm (CMTA) was developed. This method is a heuristic which analyzes the state of the bay and determines the most beneficial actions step by step. CMTA uses a look-ahead principle in order to exploit the possibility of merging moves and retrieving as many slabs as possible at a time, as long as it contributes to decreasing the total number of moves. CMTA was tested on BRP benchmark instances, as well as real-world instances provided by the industrial partner. For testing on the benchmark instances, all relevant parameters were adjusted in order for CMTA to approach the problem as a standard BRP (with homogenous items, with height restrictions instead of movement restrictions and without batch moves). Compared to the state-of-the-art BRP solution methods from the literature, CMTA showed inferior yet acceptable performance. Furthermore, we also tested the performance of CMTA on the benchmark instances with batch moves allowed in order to analyze the improvements of the objective value. Drastic improvements were observed as a result of increasing the lifting capacity. We came to a conclusion that not only the lifting capacity of 4 slabs is most realistic, but also that the relative improvement from further increasing it was not as substantial. Having chosen the lifting capacity of 4 slabs, we also studied the changes in the objective value while increasing the maximal stack height. Predictably, the objective value improved along with these increases with decreasing marginal improvement. Thus, at a certain point further augmentation of the maximal stack height did not boost the objective value noticeably. The higher the stacks in the initial bay are, however, the longer does the increase in the maximal stack height have a significant impact on the objective value. The results of testing CMTA on real-world instances also met all expectations, although the performance of CMTA on these instance sets could not be compared to that of other methods, as there are no methods designed for this particular type of BRP.

Kurzfassung

Optimierung der Lagertätigkeiten ist von großer Bedeutung für schnellere und kostengünstigere Verwaltung des Lagers. Als Beispiel einer Situation, wo eine effiziente und automatisierte Lösung für Stapelung und Entnahme gelagerter Objekte gesucht wird, ist ein Stahlbrammenlager. Unter Berücksichtigung der von einem Industrie-Partner (Voestalpine Stahl GmbH) zur Verfügung gestellten Daten wurde das Stahlbrammenstapelungsproblem (engl. SSSP, Steel Slab Stacking Problem) als Verallgemeinerung des bereits im Wissenschaftsfeld Operations Research bekannten Stapelungsproblems (engl. BRP, Block Relocation Problem) formuliert. Die Besonderheiten des SSSP im Vergleich zum Standard-BRP sind: die Möglichkeit der Versetzung mehrerer Objekte gleichzeitig, heterogene Objekte, Bewegungs- und Inkompatibilitätsrestriktionen sowie abweichende Entnahmebedingungen. Im Rahmen dieser Masterarbeit wurden zahlreiche Lösungsmethoden für BRP untersucht und der Schluss gezogen, dass die Lösung des SSSP eine besondere Vorgehensweise erfordert, die zwar gewisse Ähnlichkeiten zu den bereits vorhandenen BRP-Lösungsmethoden aufweisen sollte, jedoch an alle Besonderheiten des SSSP genau angepasst werden muss. Somit wurde ein neuer Algorithmus zur Lösung des SSSP entwickelt (engl. CMTA, Combined Multi-Target Algorithm). Diese Methode ist eine Heuristik, die den Zustand des Lagers fortgehend analysiert und schrittweise die vorteilhaften Bewegungen bestimmt, beruhend auf einer vorausblickenden Handlungsstrategie. CMTA geht weitsichtig vor, um die Möglichkeit auszunutzen, Versetzungen zu kombinieren und möglichst viele Brammen gleichzeitig zu entnehmen. Dabei wird ständig die Sinnhaftigkeit und die Realisierbarkeit der geplanten Bewegungen überprüft. CMTA wurde sowohl auf Benchmark-Instanzen für BRP als auch auf den von der Voestalpine Stahl GmbH bereitgestellten Echtwelt-Instanzen getestet. Im Rahmen der Testung auf den Benchmark-Instanzen wurden alle relevanten Parameter angepasst, sodass das Problem als Standard-BRP gelöst wird (mit homogenen Objekten, ohne besondere Bewegungseinschränkungen und mit einer Hebekapazität von nur einem Objekt). So konnte die Qualität der von CMTA gelieferten Lösung zu den in der Forschungsliteratur festgehaltenen Ergebnissen anerkannter Lösungsmethoden für Standard-BRP verglichen werden. Im Vergleich zu den besten Methoden aus der Literatur lieferte CMTA schlechtere, jedoch akzeptable Ergebnisse. Des Weiteren wurden die Benchmark-Instanzen mit CMTA mit größeren Hebekapazitäten als 1 gelöst, um die damit verbundenen Verbesserungen der Zielfunktion zu analysieren. Im Zusammenhang mit erhöhter Hebekapazität konnten erhebliche Verbesserungen erreicht werden. Erwartungsgemäß sank mit steigender Hebekapazität auch der Grenznutzen der Kapazitätssteigerung (d. h., die relative Verbesserung der Zielfunktion). Angesichts dieser Beobachtung sowie auch der Tatsache, dass eine Hebekapazität von mehr als 4 Objekten in der Praxis selten vorkommt, sei es bezogen auf Stapelung von Stahlbrammen oder auch anderer Objekte, wurde für weitere Tests die Hebekapazität von 4 ausgewählt. Mit dieser Hebekapazität wurden alle Benchmark-Instanzen auch unter der Annahme jeweils unterschiedlicher maximaler Stapelhöhe gelöst. Wie auch erwartet, die Steigerung dieses Parameters hatte eine positive Auswirkung auf die Zielfunktion. Jedoch war diese Verbesserung nur bis zu einer gewissen Stapelhöhe wesentlich. Je größer ist die ursprüngliche Stapelhöhe im Lager, desto bedeutender ist die Differenz zwischen der ursprünglichen und der höchstzulässigen Stapelhöhe für die Zielfunktion. Die von CMTA gelieferten Ergebnisse für die Echtwelt-Instanzen wurden ebenso den Erwartungen gerecht.

Table of Contents

Acknowledgement 1
Abstract 2
Kurzfassung 3
List of figures 5
List of tables 6
1. Motivation and problem setting 7
   1.1. Some facts and figures on the steel industry 9
   1.2. Introduction to the steel slab stacking problem (SSSP) 14
2. Literature Review 15
   2.1. Review of literature on BRP solution approaches: exact methods 18
   2.2. Heuristic methods 22
3. The steel slab stacking problem (SSSP) 39
4. The Combined Multi-Target Algorithm (CMTA) 42
5. Results 60
   5.1. Testing on benchmark instances 60
   5.2. Testing on real-world instances 71
6. Conclusions 73
References 76

List of figures

Figure 1. Global steel demand (finished steel). Source: (4) 11
Figure 2. Global steel demand (finished steel) by regions. Source: (4).................... 11
Figure 3. Development of steel demand in China, in million tonnes. Source: (4).................... 12
Figure 4. Straddle carrier. Source: Die Logistik Service GmbH 14
Figure 5. A small instance of BRP solved 15
Figure 6. A small BRP instance without batch moves.................... 32
Figure 7. The instance from Fig. 3 solved with batch moves allowed 32
Figure 8. Alternative removal sequence.................... 33
Figure 9. Second possible optimal removal sequence.................... 33
Figure 10. Third possible optimal move sequence 34
Figure 11. Straddle carriers moving over rows of slab stacks. (source: voestalpine Stahl GmbH).................... 40
Figure 12. CMTA performed on a small BRP instance.................... 48
Figure 13. Another example of CMTA performed on a small BRP instance..................... 50
Figure 14. CMTA in a flowchart.................... 52

List of tables

Table 1. Comparison to advanced methods from Tricoire et al. (19) for Caserta instances, H_{max} unlimited 62
Table 2. Comparison to advanced methods from Tricoire et al. (19) for Caserta instances, $H_{max} = \mathrm{H} + 2$ 63
Table 3. Comparison to fast methods from Tricoire et al. (19), H_{max} unlimited 65
Table 4. Relative improvement while changing H_{max} from unlimited to $\mathrm{H} + 2$ 66
Table 5. Comparison to results from Kopfer, Zhang & Liu (8) for Caserta instances 67
Table 6. Results for different vehicle capacities, CMTA with H_{max} unlimited 69
Table 7. CMTA Results for Caserta instances with cap = 4 and varying H_{max} 70
Table 8. Results on real-world instances 71
Table 9. Average results of testing on real-world instances 73

1. Motivation and problem setting

The motivation for this thesis is a real-world optimization problem that occurs in steel production. In recent academic literature in the field of operations research and computational logistics, problems of this kind are known under the term *block relocation problem (BRP)*. However, the abovementioned real-world optimization problem has a few significant differences to the standard block relocation problem, which require modified or new solution approaches, as opposed to those that already exist for the standard BRP and its most common modifications. This specific case of the BRP will be referred to as the steel slab stacking problem (SSSP), from now on. The specific SSSP involved in this research originates from *voestalpine Stahl GmbH*, one of the largest European steel producers, which brings a broad spectrum of high-quality specialized products onto the market. The purpose of this thesis is to develop and implement a suitable method that is able to tackle the steel slab stacking problem and that could be applied in a real-time manner in warehouse logistics. As the SSSP is a specific case of the BRP, a solution method designed for it should be tested on BRP benchmark instances. Furthermore, this thesis presents a comprehensive overview of the current state of research on BRP solution methods, some of which also inspired our algorithm developed for SSSP.

With regard to this problem setting, following questions arise.

- Research question 1: What are the specifics of the SSSP as a generalization of the standard BRP, and what solution methods can be applied to solve the SSSP with a good solution quality and low computational effort?
- Research question 2: How well does the SSSP algorithm developed in this thesis perform on established BRP benchmark instances and on real-world data sets provided by an industrial partner?
- Research question 3: How do different input parameters impact the performance of the proposed SSSP algorithm?
- Research question 4: What methodological differences are there between the proposed SSSP algorithm and solution methods tailored to solve the standard BRP?

Significant motivational factors for choosing this line of research were the importance of steel industry and the ways in which it can benefit from the implementation of decision support systems for operational routines. The steel industry remains at the core of global development. As a permanent material which can be repeatedly recycled, steel is fundamental to a successful circular economy and preserving resources. Like many other industries, steel industry relies on automation to a great extent. The aims of automation are not only improved process control, stable process optimization systems and quality-assured production, but also a safer working environment (according to World Steel Association, (1)). Digital solutions for operative decision support contribute significantly to processing times reduction and better performance significantly. Moreover, they allow to diminish the amount of monotonous, time-consuming, repetitive or error-prone tasks which have to be carried out by employees. This is why automation of operational routines within steel production and logistics is a meaningful research topic.

Digitalization and automation commonly involve decision support systems (DSS). (2) These can be defined as information technology solutions that can be used to support complex decision-making and problem solving. According to D. J. Power (3), DSS can be classified into five major types:

- communication-driven DSS, the purpose of which is enabling and facilitating cooperation
- data-driven DSS, which facilitate access to and manipulation of large amounts of structured data
- document-driven DSS, which aid to manage, find, and manipulate unstructured information in a variety of electronic formats
- knowledge-driven DSS, which provide problem-solving expertise based on given rules, procedures and statistical data
- model-driven DSS assist in decision-making based on a quantitative model – in particular, a statistical, financial, optimization, or simulation model

In this thesis, we aim to develop a model-driven DSS. The procedure for which an optimization technique is sought here is related to efficient retrieval of semi-finished steel products, so-called steel slabs, from a large storage. Such a technique can be developed either as a mathematical formulation that can provide an optimal solution,

or as a heuristic, which is more practicable if a mathematical formulation is too complex and requires too much computational time or effort.

1.1. Some facts and figures on the steel industry

In the years 2008 to 2019, the use of steel in the world has grown by 31.9 % altogether (in Austria, the corresponding increase was about 11.9 %) (1). However, the growth of the steel industry has been slower than expected in 2019, due to the continuing manufacturing recession in developed countries. The decline in global steel demand in 2020 has been quite severe due to the implications of the CoV-19 pandemic. Nevertheless, it is believed that the impact on the steel demand in the medium-term may turn out to be less severe than what was experienced during the global financial crisis in 2008/9 (1).
World crude steel production increased by 3.1% in 2019, but there were significant differences across regions. All regions, with the exception of Asia and the Middle East, saw a decline in steel production in 2019. A strong rate of growth continued in the Middle East (20.5 %), while Asian growth remained robust (5.3 %). In contrast, other regions experienced significant steel production declines: Non-European Union Countries (-8.5 %), South America (-8.4 %), Africa (-6.9 %), and the European Union (EU) (-5 %). Declines were more moderate in North America (-0.8 %) and the Commonwealth of Independent States (CIS) (-0.6 %) (4).
In 2019, Austria was on the 22nd place in the world ranking of steel-producing countries, measured by crude steel production, with a tonnage of 7.4 million. In this year, 7 million tonnes of semi-finished and finished products were exported, and 5.2 million tonnes of steel imported, which resulted in net exports of 2.8 million tonnes. Note that the majority of imports (5 million tonnes) have been constituted by iron ore (1).
Austria's largest steel producer is *voestalpine Stahl GmbH*, the parent company of the Steel Division of *voestalpine AG* and the largest company in the *voestalpine* Group. With a net turnover of € 13 560.7 million, *voestalpine AG* accounted for 42.6 % of Austria's total turnover in the metal industry in 2018 (5). The problem setting, for which a solution approach is developed within this thesis, originates from the steel slab storage of *voestalpine Stahl GmbH* in Linz, Austria. *voestalpine Stahl GmbH* operates a fully integrated metallurgical plant that combines all process steps of steel production at a

single location: coking plant, blast furnace, steelmaking plant, hot-rolling and cold-rolling mill and galvanizing line.

There have been dramatic repercussions of the COVID-19 crisis regarding the concern's business activity (voestalpine Group report, April 2020 (5)). In Europe, the state-mandated lockdowns had a devastating effect on the economy within the first business quarter. The situation in April was quite close to an economic collapse, yet in May there was an onset of a slow recovery. Particularly private consumption rebounded sizeably once the lockdowns were eased. However, in the industrial sector, the investment activity remained on a very low level even after the lockdowns. The reason for this was uncertainty in view of persisting economic implications of the lockdowns, long-term damage to the economy and high probability of further restraining measures being introduced by the governments in the last quarter of 2020 due to the second wave of the pandemic. Although sharp recovery of economic activities could be observed and the initial momentum of recovery was stronger than expected with release, second waves were very predictable and local outbreaks continued. Beginning with August, there was no more acceleration in the steel sector's recovering trend. However, there has been a noticeable upward trend of export demand from outside the EU. The European automotive industry completely stopped production in April and restarted it again gradually in May. By the end of the second quarter of 2020, automotive production still fell far short of the level that obtained prior to the outbreak of the pandemic. The manufacturing sector, however, still demonstrates an upwards trend in recovery. The construction sector's recovery, on the other hand, is being supported by infrastructure investment.

As expected in 202, the recovery pace continued to be slow due to continued social distancing, unemployment and corporate bankruptcies, and continuous uncertainty. Structural changes are expected to accompany the recovery process (6).

Following graphics from the most recent published session of the OECD Steel Committee (24-29 September 2020) demonstrate developments in steel demand in last year and this year, as well as next year (predicted). The absolute demand for finished steel products as well as the relative change in percent, compared to the previous year, are depicted (global and by regions). Noticeably, Chinese economic growth, as well as steel industry in particular, shows a distinct upward trend, unlike any other economy.

This is why additional insights on the recovery of Chinese steel industry have been provided (4).

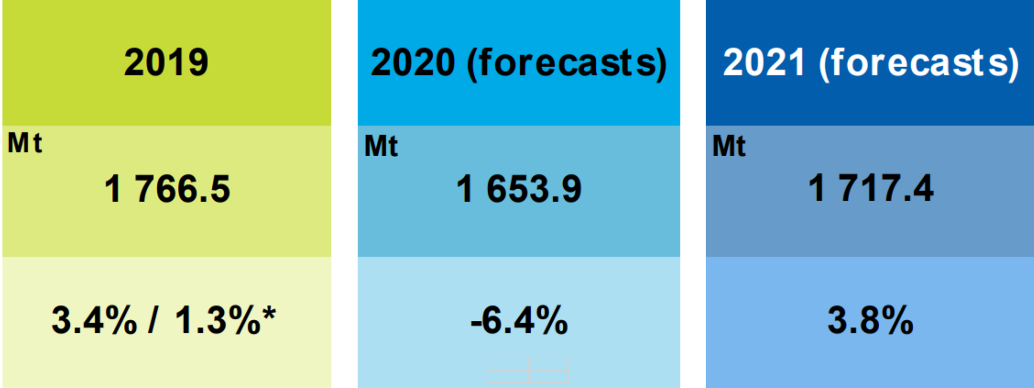


Figure 1. Global steel demand (finished steel). Source: (4)
** World growth rates based on adjusted Chinese growth rates*

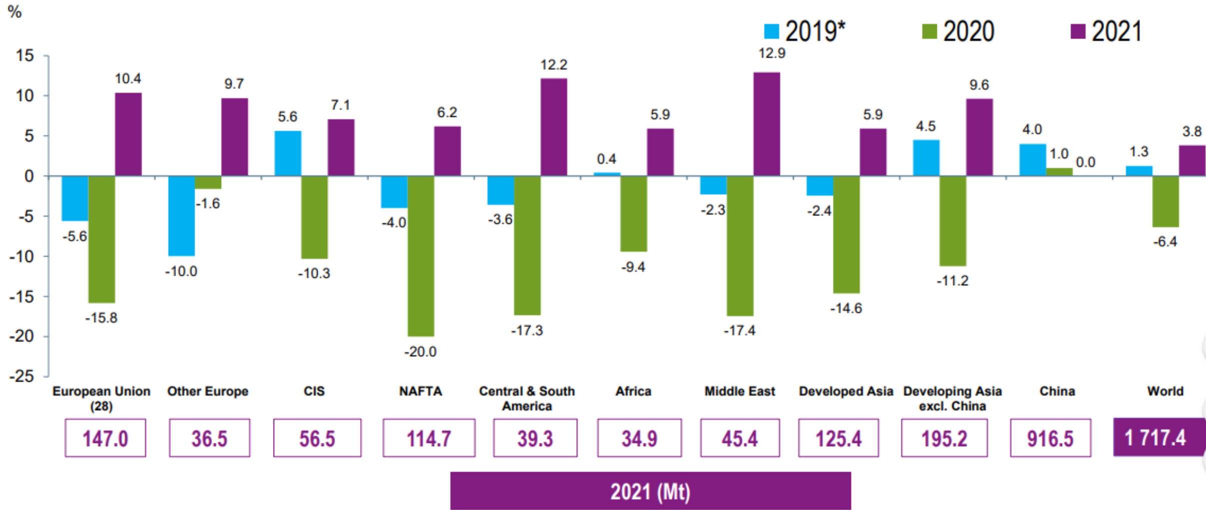


Figure 2. Global steel demand (finished steel) by regions. Source: (4)
** World growth rates based on adjusted Chinese growth rates*

China in a fast recovery mode

	Mt			%			
	2019	2020	2021	19/18	20/19	21/20	2021 as % of 2007
World	1,766.5	1,653.9	1,717.4	3.4/ 1.3*	-6.4	3.8	140.2
China	907.5	916.5	916.5	8.5/ 4.0*	1.0	0.0	219.0

* Growth rates based on Chinese steel using sectors dynamics

- China's GDP in Q2 recorded 3.2% growth after contracting by 6.8% in Q1 20202
- Normal productivity in most sectors was achieved by end April
- Recovery accelerating on government stimulus and restoration of confidence
- Global recession is preventing a stronger manufacturing recovery, but better than expected exports recovery
- Government stimulus on infrastructure and real estate will drive steel use in 2020, and in 2021

Recovery of steel using sectors

Growth y-o-y	Jan-Feb	Jan-Mar	Jan-Apr	Jan-May	Jan-Aug
Industrial production	-13.5%	8.4%	-4.9%	-2.8%	+0.4%
Total FAI	-24.5%	-16.1%	-10.3%	-6.3%	-0.3%
Real estate FAI	-16.3%	-7.7%	-3.3%	-0.3%	+4.6%
Infrastructure FAI	-30.3%	-19.7%	-11.8%	-6.3%	-0.3%
Machinery output	-28.2%	-17.2%	-9.0%	-4.9%	+1.2%
Auto output	-45.8%	-44.6%	-32.3%	-23.6%	-9.0%
Ship delivery	-22.9%	-27.3%	-15.4%	-15.1%	-10.4%
App. steel use	**5.5%**	**3.4%**	**2.5%**	**4.0%**	**7.5%**

Figure 3. *Development of steel demand in China, in million tonnes. Source:* (4)

The *voestalpine* Steel Division's revenue dropped by 29.4 %, from EUR 1,182.1 million in the first quarter of the business year 2019/20 to EUR 834.9 million in the first quarter of the business year 2020/21 (April to June). This development reflects the difficult market environment and has been caused mostly by the large decrease in deliveries, but also by the drop in market prices. The division's cost structures were adjusted to the changed order levels. The division's EBIT (earnings before interest and taxes) for the reporting period has been EUR -13.5 million (margin of -1.6 %), which is slightly negative compared to the EBIT of EUR 60.8 million (margin of 5.1%) in the same quarter of the previous business year. As of June 30, 2020, the number of employees (full-time equivalents, FTE) in the Steel Division fell year over year by 5.1 % to 10,181, due to the adjustments made in view of the substantial drop in production. (5)
However, the production intensity at *voestalpine Stahl GmbH* has been rising since September 2020, and, in particular, the utilization of the production and storage facilities which are the source of the problem discussed in this thesis has been slowly returning to more or less normal levels. Furthermore, if we behold the quarter-on-quarter comparison (QoQ) of *voestalpine Steel Division* between the business year's first and second quarters (7), the economic decline at the beginning of the business year is compensated by a positive trend in the second quarter. Numbers show that the *Steel Division's* revenue rose by 19.2% in the second quarter, compared to the first quarter. The Steel Division succeeded in substantially boosting deliveries of flat steel products (7). In any case, the demand for digitalization and automation of production and logistics is very high in this branch of industry, as well as many others. It should be remarked that not only efficiency, but also the sustainability and social impact of the steel industry can be positively influenced by automation and digital decision support systems. Resources and time can be saved due to computer-supported decision making, work safety and gratification can be improved by delegating dangerous, error-prone, monotonous or cumbersome tasks to machines. Digitalization also brings along interoperability, decentralization and virtualization, which can facilitate process management and allow to reduce the amount of personnel, time, premises and resources required. As to the year 2020, the pandemic prevention measures in all workspaces became a new driver for automation and optimization of business processes, which would allow to

reduce the presence and the concentration of employees directly in the production or transportation areas and allow more people to work remotely.

1.2. Introduction to the steel slab stacking problem (SSSP)

The optimization problem discussed in this thesis originates from an outdoor storage facility where steel slabs are stacked on top of one another. Whenever some of those items have to be picked up from the storage, the retrieval operation has to be carried out in a given sequence. An item (or a batch of items) can only be retrieved if it is (they are) on the top of a stack. As this is often not the case, numerous items have to be unblocked before the retrieval. The stacking operations are carried out by specialized heavy-duty vehicles, so-called straddle carriers (Figure 4), that can grab and move multiple steel slabs at once. A small example of a simplified version of SSSP is demonstrated below in Figure 5. The example represents a storage entity, hereinafter referred to as a bay, with five items illustrated by blocks. The bay is composed out of 3 stacks (see horizontal axis) and each stack contains 5 tiers (see vertical axis). All items have to be retrieved in the order corresponding to their indices (increasing) and placed on top of stack $L + 1$ that is located outside of the bay. A potential solution to this instance

Figure 4. Straddle carrier. Source: Die Logistik Service GmbH

of the BRP would be to first retrieve item 1, relocate item 4 from stack (2) to stack (1), and consequently retrieve items 2, 3 and 4 in that order.

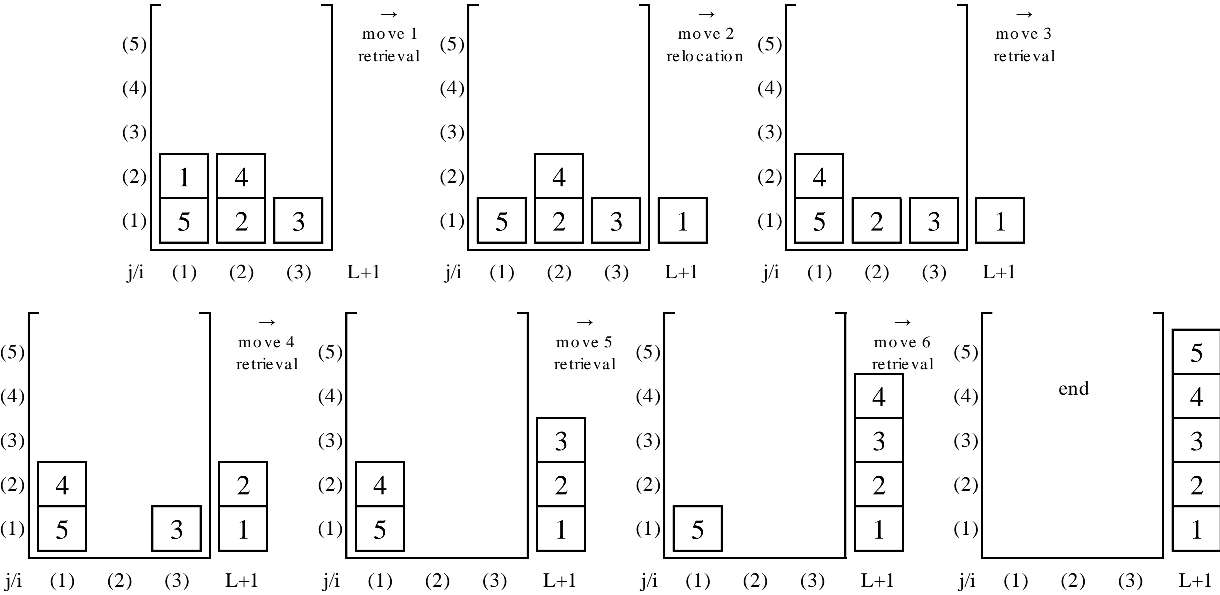


Figure 5. A small instance of BRP solved

The motivation of this thesis is to develop an algorithm by means of a heuristic which could determine an efficient sequence of moves for performing every retrieval operation in a reasonable amount of computational time. Implementing such an algorithm would allow the driver of the vehicle, who is carrying out the retrieval operation, save time and calculation efforts while executing the necessary moves.

2. Literature Review

The SSSP is a variant of the BRP. First, let us define the BRP as it can be found in related literature. We are given a bulk of items (casting products, containers, boxes, elements, etc.) of different types which are collectively stored inside of a bay. Usually unsorted in the initial state of the bay, the items eventually have to be retrieved at some points in time in a specific order. Every item is assigned a unique index, which gives it an unambiguous order number in the required retrieval sequence. It is usually stated that item 1 has to be retrieved first, then item 2, and so on until the bay is fully emptied.

As all items are stored in stacks directly on top of one another out of consideration for space limitations and each stack contains items of different priorities in arbitrary order, it is evident that there are items which are blocked by items of a lower priority on top of them. Such items can only be freed by relocating the blocking items to other stacks, because there are no buffer spaces between stacks, with an exception of stacks which have become empty because all items have been relocated or retrieved from them. The objective of the BRP is to find a sequence of moves that allows us to efficiently retrieve all items of a bay such that the total number of moves is minimized. The most suitable relocation destinations are stacks which do not contain any items of a lower index than the item being relocated. If such a relocation is possible, the relocated items do not have to be moved again until their retrieval. Otherwise, a relocation destination should be chosen so that the items which become blocked by the relocated items have as low of a priority as possible. The larger the number of items and stacks, i.e., the larger the bay, the larger the number of possible moves, and the higher the complexity of the problem. Caserta et al. (8) provide a complexity study of the BRP. They prove that the BRP is NP-hard for any finite maximal stack height and number of stacks smaller than the number of items. In many real-world logistical problem settings (container terminals and other similar storages) the number of stacks and items is large enough to deem a mathematical model (such as a mixed integer problem) not practically applicable as a solution method. This is the reason why numerous heuristic methods have been developed in this field, which still remains quite open for new approaches and different problem modifications.

The device used to move items is problem dependent. Container yards, for example, are commonly equipped with a gantry crane, while indoor facilities operate bridge cranes. In both cases, the cranes most often have a capacity limited to one item (9). In the existing literature, most BRP solution methods focus on minimizing the total number of relocations required to clear the bay. Note that only relocations of items to other stacks are considered as moves in this case, and not retrievals. Obviously, the number and sequence of retrievals is unambiguous in the case when only one item can be retrieved at a time, so only relocations are subject to optimization. However, this is not the case if the retrieving device has a lifting capacity of more than one item. Being able to relocate or retrieve more than one item at a time is usually referred to as batch moves in the literature (as in (10)). The possibility of batch moves means that retrievals

become subject to optimization as well. In case of the SSSP discussed in this thesis, the straddle carrier functions as a mobile crane, due to an integrated crane between its wheels. As opposed to many crane devices assumed in the BRP, the crane of a straddle carrier can lift multiple steel slabs at once up to a certain capacity. Aside from minimizing the number of moves, there may be yet other objectives, such as minimizing the total distance travelled or minimizing the total time. These are relevant rather for BRP settings with vehicles and similar retrieving machines than for cranes, and especially for the BRP with batch moves. Minimizing the distance travelled takes different distances between stacks into consideration, whereas minimizing the total time additionally considers the duration of different moves. The latter can be meaningful if relocations and retrievals have different durations, or if batch moves are possible and the quantity of items lifted influences the duration of relocations and retrievals. It should be noted that the possibility of batch moves has very significant implications for the solution space and will be separately treated in the next chapters, as the main research topic of this thesis is a modification of the BRP which, among other particularities, also involves batch moves. However, as the minimization of the total distance travelled or the total time travelled makes the solution process even more complex, it has not been tackled in this research.

In BRP-related literature it is usually distinguished between restricted and unrestricted BRPs. The restricted BRP considers only relocations of items that block the next item to be retrieved, whereas the unrestricted version considers any other moves, so-called cleaning moves, as well. Solving the BRP in the restricted fashion implies reducing the complexity but losing optimality (as demonstrated by Caserta et al. (11)). Indeed, it is obvious that unblocking only the current target item and not considering the impact on the other items (which also will sooner or later have to be retrieved) is myopic.

Another important criterion of the BRP is whether each item has a unique index (priority) or if there are groups of items sharing the same index. The most common approach is to assign a unique index to every item, even if it is not often the case in reality, in order to facilitate established solution technics. The case where multiple items share the same index is commonly defined as the BRP with duplicated priorities. When there are groups of items sharing the same priority, it does not matter in which order the items with the same priority are retrieved, which is a very likely scenario for many real-life situations. Solving the BRP in this way is more likely to result in a more efficient

relocation sequence, by avoiding extra moves. Still, as duplicated priorities tend to make the solution process much more difficult, they are often avoided by giving every item a unique index (priority).

2.1. Review of literature on BRP solution approaches: exact methods

There have been different mixed integer problem (MIP) formulations for the BRP so far: BRP-I (for unrestricted BRP) and BRP-II (for restricted BRP) by Caserta et al (11) as well as its corrections by Exposito-Izquierdo et al. (12), who also presented a faster branch-and-bound algorithm for the restricted BRP, and an alternative model by Zehendner et al. (13) with less variables, which allowed for a significant reduction of the CPU time. Worth mentioning is the BRP-III model by Petering and Hussein (14) for the unrestricted BRP, which includes significantly less variables than the BRP-I formulation from Caserta et al. (11) and shows better average performance. Still, the authors pointed out that MIP formulations of the BRP require a significant computational effort even for very limited test instances, compared to those occurring in real-world applications (14). Further notable branch-and-bounding methods are a tree search with a 3-step procedure by Bortfeldt and Forster (15), a branch and bound technique by Tanaka and Mizuno (16) which uses three types of dominance properties to eliminate unnecessary nodes in the search tree and, finally, an improvement of an exact branch-and-bound algorithm by Expósito-Izquierdo et al. (12), developed by Tricoire et al. (17), in which the authors managed to improve previously best known benchmark results by using a look-ahead mechanism for the evaluation of successor nodes.
The paper "**A tree search procedure for the container pre-marshalling problem**" by **Bortfeldt and Forster** (15) is a very noticeable contribution because it introduces an efficient solution technique for the unrestricted BRP with duplicated priorities, which is able to solve instances with up to 100 items (10 stacks with 10 items each) in reasonable amounts of time. The authors introduce a useful classification of possible relocations. A relocation can be considered as

- GG (good-good) when it relocates an item which does not block any item of higher priority so that it remains well-placed,
- BG (bad-good) when it relocates a badly located item to a position where it does not block any items of higher priority anymore,

- GB (good-bad) when it moves a well-located item to a position where it blocks at least one item of a higher priority, and
- BB (bad-bad) if a badly located item is moved to a stack where it is badly located as well.

Obviously, badly located means blocking (at least one item of higher priority) and well-located means non-blocking. The first step of the tree search procedure by Bortfeldt and Forster is to construct an initial solution using a greedy heuristic which follows the steps listed below.

1) retrieve all retrievable items when possible
2) if no retrievable items are available, perform following steps:
    a) search for BG-moves to a non-empty stack and perform the one which minimizes the difference in priorities of the relocated item and the top item of the destination stack
    b) if no such moves are possible, look for BG-moves to an empty stack and perform the one where the moved item has minimal priority
    c) if no BG-moves are possible at all, among all BB-relocations that help to free the next item to be retrieved (=any one of those with the highest priority), select the one that maximizes the difference between the relocated item and the item with the highest priority in the destination stack.

These steps are repeated until all containers are retrieved. The authors mention that this greedy approach can be considered a generalisation of the min-max-heuristic described by Caserta et al. (11) that is not limited to BG moves from a stack that contains an item with the lowest index. This min-max-rule is very important in the context of BRPs and can be summarized as follows: when good relocations to multiple stacks are possible, select the relocation to a destination stack which has the minimal lowest-index item of the possible stacks (due to the fact that the higher the lowest index in a stack is, the more items can be relocated to it without blocking anything, i. e. well-relocated). If, however, only bad relocations are possible, choose the destination stack which has the maximal lowest index among all possible destination stacks, in order to delay the time when it will have to be relocated again (which will evidently be as soon as the lowest index item in the destination stack has to be retrieved).

The initial solution obtained by this three-step greedy heuristic delivers the first upper bound. Next, compound moves for the initial layout are generated and performed. To

determine compound moves, however, the computation of a set of productive moves is necessary. This set consists of: 1) all possible retrievals from the current layout; 2) if none, all possible BG-relocations; 3) if none, all possible BB-relocations that free an item with the currently highest priority are sorted in ascending order by the difference between the priority of the item moved and the highest priority in the destination stack and then the list is reduced to a particular size (defined by a parameter) by discarding the corresponding number of moves at the end. After that, all possible GG-relocations are sorted in ascending order, beginning with the relocation with the maximal decrease (minimal increase) of the difference between the priority of the relocated item and the item directly below it before and after the relocation, after which the corresponding number of moves at the end are discarded in order to reduce the list to the size defined by a parameter. Finally, the union of those two sets forms the set of productive moves. This procedure is embedded in the compound move generation procedure., which results in a list of compound moves. Finally, the main procedure *perform_compound_moves* works as follows: 1) checks if an optimal solution has been reached, the time limit has been exceeded or a leaf node[1] was found (updates the current best solution in the first or third case if applicable; 2) calls the procedure *determine_compound_moves* whose output is the list of compound moves and performs a particular number of compound moves (from the beginning of the sorted list) alternatively, calling the procedure *perform_compound_moves* for each of them again – until any of the conditions in 1) arrive and the procedure is terminated or the current best solution is updated.

The authors tested their method on the benchmark instances introduced by Caserta et al. (11). Over all relevant test cases (calculated as the arithmetic mean of the average improvement of each test case), the tree search heuristic found solutions with 47.3%, 9.6%, 5.4% and 0.1% less relocations than the algorithms proposed by Kim and Hong (18) (heuristic rule for relocating blocks), Caserta et al. (11) (corridor method and look-ahead-heuristic) and Caserta and Voß (8) (fine tuning for corridor method), respectively. For most of the test cases, the solutions are computed in less than 1 s. Only the very large test cases (beginning from 6x10) need more computational time (15).

[1] A node on which no further branching is possible.

A recent successful approach is the exact algorithm for the unrestricted BRP with distinct priorities proposed by **Tanaka and Mizuno** (16), which enforces a branch and bound technique with three types of dominance properties to eliminate unnecessary nodes in the search tree. The underlying algorithm is roughly the same as the ones proposed by Tanaka and Takii (19) or Zhu et al. (20). First, a lower bound and an upper bound are calculated and a solution with an objective value which is not larger than the current upper bound is searched for using the branch-and bound method. Afterwards, the upper bound is updated if a smaller one was found. The procedure is then either terminated, if the found upper bound equals the lower bound, or, if not, the lower bound is increased by one and the procedure is repeated. The authors succeeded in improving the efficiency of the traditional branch-and-bound routine by introducing dominance properties to suppress the generation of unnecessary nodes in the search tree. Furthermore, the lower bound introduced by Forster and Bortfeldt (15) was improved, which allowed for solving a larger number of benchmark instances to optimality than the existing exact algorithms in the literature.

F. Tricoire et al. (21) further improve the exact branch-and-bound algorithm by Expósito-Izquierdo et al. (12) by applying a look-ahead mechanism for evaluating successor nodes. They also compared the effectiveness of different bounding procedures and emphasized that the quality of initial upper bounds has a strong impact on the solution. Very importantly, they have introduced some new effective heuristic procedures for the unrestricted BRP. These include two metaheuristics – rake search and pilot method – as well as some very useful subroutines which focus on new concepts introduced by the authors. The concept of safe moves comprises safe-1- and 2-relocates. The former are moves which contribute to unblocking an item, ergo decrease the number of outstanding necessary relocations, without blocking any other one. The latter are moves that enable a safe-1-relocation without blocking any items. Another new notion in the BRP context is that of **decreasing sequences.** These are sequences of at least two blocks stacked in a way that each one has a lower priority than its direct neighbour underneath. Based on those notions which are defining for the decisions on cleaning moves, the authors introduce the rules SM1, SM2, SmSEQ-1 and SmSEQ-2. S(afe)M(oves)1 iteratively performs the best safe 1-relocate until there are none such. S(afe)M(oves)2 iteratively tries to perform the best safe 1-relocate and if none are possible, the best safe 2-relocate). S(afe)m(oves)SEQ(ence)-1 performs the best

decreasing sequence relocate and if none are possible, performs as in SM-1 and S(afe)m(oves)SEQ(ence)-2 performs analogically, but if no decreasing sequence relocate possible, it performs as in SM-2. Using these fast heuristics results in the generation of good initial solutions and, as mentioned above, better starting solutions and bounds are essential for the optimality of the branch-and-bound procedure.

2.2. Heuristic methods

In spite of the existence and constant improvement and development of exact BRP approaches, oftentimes they are not applicable to many real-world problem settings that constitute a large bay size (beginning with approximately 60 blocks). The higher the maximum stack height is, the more costly becomes the solution procedure, even with less stacks – e.g., 6 stacks with height of up to 10 each are more complex to solve than 10 stacks with height up to 6 each, even if the overall number of blocks in both instances is the same. On the contrary, there are quite a few heuristics which offer acceptable solution quality for instances as large as 100 stacks x 100 tiers in reasonable amounts of time. This is why one can assume that heuristic approaches are much more promising than exact methods for application in large-scale logistical settings.
One of the first widely acknowledged heuristic approaches to solve the BRP is the method developed by Kim and Hong (18). First, the authors introduce the notion ENAR: expected number of additional relocations, which is calculated for all stacks lower than the maximum height. Furthermore, actions are defined as either relocating items or retrieving a target item. Cleaning or pre-marshalling moves are not considered in this study. The heuristic rule proposed for determining the storage locations of relocated items consists in choosing the action a with the minimum contribution $R(a)$ to the total number of relocations from all possible actions in the current state. $R(a)$ is evaluated as the difference of total ENAR of the new state S' and the old state S plus the number of relocations realized or confirmed (=deemed necessary) by action a. In case of multiple items sharing the same priority, the next target item is chosen by picking among them the item with minimal $R(a)$.
Caserta et al. (8) applied the so-called corridor method to the BRP. It is based on the general concept of the corridor method, a hybrid metaheuristic for BRP which has been presented by Sniedovich and Voß (22). The basic idea of this method is to use an exact

method over limited portions of the solution space of a given problem. The presuppositions met by the authors include approaching this problem in a restricted way by only allowing forced moves (i. e., while retrieving a target block, only blocks which are contained in the same stack as the target block may be relocated). The authors introduce a recursive formulation and a dynamic programming algorithm for the BRP. The dynamic programming model consists of state variables, decision variables and a state transition function. In particular, s (where $s = (k, i, t, C)$ denotes the state variable, $k \in \{1, \ldots, n\}$ is the block to be retrieved, $i \in \{1, \ldots, m\}$ is the stack in which the target block is found, t denotes the list of blocks located above the target block, and C describes the configuration of the remaining blocks). There are decision variables x for every state s, with $x \in D(s)$, where $D(s)$ denotes the set of all stacks to which the block k (in state s) can be relocated, meaning that x can take on any value from $D(s)$). Lastly, the state transition function T describes the state obtained by applying decision $x \in D(s)$ to the current state s indicated *by* $s' = (k', i', t', C')$ and $s' = T(s, x)$. Of course, having a state $s*$ where no containers are blocking the target container k, $t = \emptyset$, which automatically sets $k' = k + 1$, i' becomes the stack in which block $k + 1$ is currently located, t' becomes a new list of blocks currently above the target block $k + 1$ and, finally, $C' = C$ describes the configuration of the remaining blocks.

The recursive formulation is based on the following functional equation:

$f(s) = 1 + min_{x \in D(s)} \{f(T(s, x))\}, k = n - 1, \ldots, 1,$ where $s = (k, i, t, C)$ indicates the current state, and $T(s, x)$ is the state resulting from the application of decision x to the current state s. The function f calculates the costs of a state s, whereas $f(s) = 1$ for $s = (n, i, \emptyset, C)$, represents the cost of retrieving a block from the bay and moving it to its final destination.

The main disadvantage of the dynamic programming approach itself is the exponentially growing number of reachable states. To constrain this, a two-dimensional corridor is defined around each configuration that limits the number of states that are generated from this configuration. One dimension (also called horizontal) of limitations would be the reduction of the number of stacks to which blocks can be relocated, and another dimension (also called vertical) would be the establishment of a maximum height of a stack. This can be projected as forming a corridor around the current bay configuration, with the “width” of the corridor defined by a parameter λ and its height with a parameter δ. These should be carefully adjusted so that feasibility is granted at each step. The

parameters are supposed to be justified by real conditions, with δ being the maximum stack height and λ defining the stacks onto which a particular block can be relocated so that its length is compatible with that of the relocation stack.

Now, the "constrained" neighbourhood induced by the application of the corridor method upon a given configuration is defined in the following fashion: $s = (k, i, t, C)$ is the current bay configuration, where C = $\{c_1,..., c_m\}\backslash\{c_i\}$ indicates all the stacks of the bay excluding stack i. Let us indicate with $|c_l|$ the number of blocks currently on stack l. Given two parameters δ and λ, the set of restricted admissible decisions is defined as

$$D(s, \delta, \lambda) = \{x \in \{1, \dots, m\}\backslash\{i\} : i - \delta \leq x \leq i + \delta, |c_x| < \lambda\}$$

and the restricted neighbourhood of the current configuration s, meaning the set of "feasible" bay configurations that can be created from s is given by

$$N(s) = \{s' : s' = T(s, x), for\ all\ x \in D(s, \delta, \lambda)\}$$

Consequently, the size of the neighbourhood can be made arbitrarily small by changing the values of δ and λ. These exogenous constraints on the solution space can drastically reduce the number of states that have to be considered, which allows to apply this scheme to quite large problem instances. Evidently, reaching optimality is not granted, but testing the corridor method over instances ranging from 3x3 to 5x10 (dimensions given in tiers pro stack and stacks) showed sizeably better results compared to Kim and Hong heuristic (18) with reasonable computational time. This heuristic also outperforms some newer methods like LA-N (14) on some small and medium instances.

Jovanovic and Voss (23) introduce a new chain heuristic and review different relocation rules. In particular, they highlight the efficiency of the Min-Max rule by Caserta et al. (8), according to which the destination stack for an item being relocated is determined as the stack which has the highest maximum priority, in order to "save" the stacks with lower maximum priorities for later because they can potentially "host" more items. Yet, if all possible relocations result in a deadlock (a situation where an item is put on top of a stack which contains at least one item of a higher priority, meaning that it will have to be relocated again), the stack with the lowest maximal priority should be chosen as destination, according to (23). A simple improvement to this heuristic, as proposed by the authors, is to avoid moving an item to a stack that will reach the maximal number of tiers after that move. This would prevent losing it as a potential relocation destination

and decreases the probability of another situation where creating a deadlock cannot be avoided. In order to avoid stacks from being filled, the authors suggested calculating the priority of stack i as just the highest priority among all its tiers if relocations without deadlocks are possible, yet as the opposite of this in the case where creating a deadlock is unavoidable. Thus, the stack with a *minimal* difference of its priority and that of the relocated item should be chosen as destination, if a relocation without creating a deadlock is possible – and in the opposite case, the stack with a *maximal* difference of its priority (calculated in the alternative way as mentioned above) and that of the relocated item. This means exactly that stacks are never filled up to their maximal height unless in order to avoid creating a deadlock.

Furthermore, the authors introduce a new approach called chain heuristic which involves considering the future impacts of a current move. This includes, in particular, preventing items that are relocated earlier from using stacks that are more suitable for items that will be relocated later. The chain heuristic is based on the fact that at each step *n* when the decision about the relocation of the block r_n is to be made, it is also known which block r_{n+1} will be relocated at the step $n+1$. Either r_{n+1} is an item blocking the same target, which would mean that it is in the same stack just below r_n, or, if r_n was the last item blocking the target item, r_{n+1} will be blocking the next target item. Now, instead of sequentially deciding where to relocate r_n and r_{n+1}, the authors use an extended heuristic function which first determines whether it is better to move the item r_n and then move r_{n+1}, or vice versa, only additionally respecting the constraint that after moving r_n, the decided move of r_{n+1} will still be possible. The comparison is based on the best value of the objective function, which is the same as in the simple Min-Max-heuristic. Furthermore, there are some additional conditions which allow to choose the reverse order, based on which of the moves considered would (or would not) create deadlocks. The authors note that if Min–Max is used as the basic heuristic, the chain heuristic does not need to be calculated at every step. Because of the way the Min–Max heuristic has been defined, it is namely only possible for item r_n to use a stack that is more suitable for r_{n+1} if it has higher priority than the latter.

The authors compare the performance of the chain heuristic to that of following methods: a reshuffle index heuristic by Wu & Ting (24), the Min-Max-heuristic application as given by Caserta et al. (11) implemented by the authors and beam search by Wu and Ting (24). The reshuffle index heuristic makes the decision about the destination

stack of the blocking item which is to be relocated by choosing the stack which has the lowest number of items that have a higher priority than the item being moved. Wu and Ting (24) improved this heuristic by indicating how to resolve the ties which can appear if more than one stack has the lowest number of items of higher priority than the relocated item. Such ties are broken by selecting the stack that has the lowest maximal priority. Jovanovic and Voss (23) emphasize that by using this quite simple rule for resolving ties, a significant improvement has been achieved. As regards beam search by Wu and Ting (24), it has been used as a lower bound by the authors (23). The authors emphasize that although the majority of the beam search solutions are known to be optimal and all of them are best so far, the computational effort/time is drastically higher for this method. Beam search is a heuristic adaptation of the branch-and-bound method. It is based on the breadth-first tree search, with nodes being generated and visited level by level. It does not visit every node in the unbounded feasible solution like branch-and-bound; instead, it maintains only β (a parameter which is called beam width) promising nodes (=beam nodes) at each level. Such promising nodes are usually identified by a lower or upper bound obtained from a cost evaluation function. Only the beam node can create descendant nodes at each level and the other nodes are discarded permanently, which reduces a search space very significantly but, of course, does not guarantee that the optimal solution is not discarded during the search process. Back to the comparison of the chain heuristic to the abovementioned methods, Jovanovic and Voss implemented it using small and medium (from 3x3 to 8x8 tiers x stacks) instances and established that that the use of the chain heuristic approach improved the basic heuristic's results in almost all cases. Furthermore, using the combination of both improvements as described by the authors in their paper (chain approach + modification of the Min-Max heuristic) improved the results even more, on average. Still, the authors acknowledge that, for some instances, the quality of the solution worsened due to the application of the improvements, yet they consider the positive effect overall to be greater.

An approach which is also often referred to in literature is the **three-phase heuristic by Lee and Lee** (25). Its initial phase consists in developing a feasible movement sequence for retrieving all the items in the yard/bay without encountering conflicts. As the items should be retrieved in a particular order, the target items are retrieved immediately if available and otherwise, the containers that block the target container are moved to

nearby stacks by choosing the nearest available stack. The second phase is called movement reduction phase. It consists in attempting to reduce the number of movements in the initial feasible sequence (the one resulting from the first phase) while preserving its feasibility. This procedure is conducted by formulating and solving a mixed-integer program. The third phase is performed in order to reduce altogether working time, with regard to the fact that each movement may have a different timespan depending on the distance. In this phase, a mixed-integer program is formulated that takes the integer program used in the movement reduction phase and adds decision variables as well as constraints to introduce a time dimension into it.

The authors compare the performance of their heuristics to those of Kim and Hong (18) on instances containing 70 or 90 containers with maximal stack height of 6 or 8 tiers and conclude that in all cases except for one their heuristic shows more than 30% improvement.

A well-known heuristic which is explicitly based on “cleaning moves” (=moves aimed not directly at unblocking the target item but at improving the accessibility of other items in the long run) is the **LA-N algorithm by Petering and Hussein** (14). It is based on the following steps:

0) select the value for N (the look ahead)
1) retrieve all retrievable containers
2) if none available, let $N' = min\,\{N$, number of items still residing in the bay}
3) if the set of stacks containing N' highest-prioritized items is equivalent to the set of all stacks OR if all stacks not containing any of the N' highest-prioritized items are full, decrease N' by one and repeat step 3)
4) if the rth item of those located on the tops of the stacks containing the N' highest-prioritized items is located on the top of the target stack (the one containing the target item), go to step 6)
5) a good cleaning move involving this item (n) is sought, but if there are no stacks lower than the maximum stack height which contain no items with lower priority than n OR if n is already the lowest-numbered element in its stack, r is increased by 1 and step 4 is repeated; otherwise, go to step 6)
6) relocate the item n according to the Min-Max-rule and go to step 1.

Worth citing is also the paper of **Bo Jin et al.** (26) which presents a new greedy look-ahead heuristic (GLAH). As the authors describe, the whole method has three depth levels. On the top level of the GLAH, in every stage one relocation is executed to the current layout until the layout becomes empty. The relocation is chosen by the middle level – the look-ahead procedure – which applies a limited tree search over all promising nodes. Whether a node is promising or not is determined by the configuration achievable by applying certain relocations to the current layout). Similarly as in tree search method by Forster and Bortfeldt (15), the authors classify all possible relocations into four main categories (bad-good, bad-bad, good-good, good-bad) and two of those categories (BG and BB, because only those moves can be either target-freeing or not, while GG and GB moves are non-target-freeing per definition) are also divided into the subcategories "target-freeing (FT)" and "not-target-freeing (NF)". Thus, there are six possible kinds of relocations (FT- and NF- BG, FT and NF-BB, as well as GG and GB. Out of each category, a list is generated and sorted according to different criteria. For each BG-relocation, the difference of the priority of the item relocated and the highest-priority item of the destination stack is calculated and the list of BG-relocations is sorted in descending order with regard to this difference. Analogical sorting applies to BB, only the abovementioned difference is calculated vice versa. The GG relocations are sorted in descending order with regard to the *difference* of the priority of the item directly below the relocated item and the highest-priority item on the top of the destination stack, whereas the GB moves are sorted in descending order by the *sum* of those values. After sorting, a particular number of entries at the end of each list are discarded, so that there are no more moves in each category than defined by a respective parameter. All remaining possible moves compose the candidate list and child nodes are generated until the evaluated cost exceeds the current best cost or until a leaf node is reached. The branch nodes are evaluated by determining the best-cost-node among its children. To evaluate leaf nodes, a specific evaluation heuristic performs relocations (according to certain rules) until the layout is cleaned. The total number of relocations corresponds to the cost of the node, and the most promising relocation leading to the best leaf node is transferred to the top level as the next relocation to execute for the greedy mechanism.

The evaluation subroutine has proven to be very effective as an independent construction heuristic, as pointed out and tested by Tricoire et al. (21). It originates from Zhu et

al. (20) and has been improved by Jin et al. (26) by adding a new gap utilization process. This process is to be carried out before every BG or GG relocation in order to fully exploit the gap between the minimum index (i. e. the index of the item with the highest priority) of the destination stack and the relocated item. As already mentioned, the lower index an item has, the higher is its priority. The subroutine functions as follows: first, an “urgent target” item is determined as the one of the highest-priority items which is blocked by the least other items. Obviously, as soon as the urgent target is chosen, the items lying on it have to be relocated. The destination stacks for them are determined as follows:

- case 1: if a BG-relocation is possible, it is performed to the stack with the highest priority of its highest priority item. If no BG move is possible, go to case 2.
- case 2: check if any stack can be “freed” for well-relocating the urgent target by well-relocating its current top item to another stack. If multiple such stacks are available, chose the one with the lowest priority of its top item (which has to be relocated to another stack, and if there is more than one such stack, choose the one with the smallest minimal priority). If not possible, go to case 3.
- case 3: FT-BB relocations are considered. The stack s' with the lowest maximal priority (=largest minimal index) is checked for the following condition: whether its height is equal to $maxHeight - 1$ and the item currently being relocated is not the one with the smallest index of those blocking the target. If so, the destination stack is reselected as the one with **second** minimal index and the relocation of the considered item is carried out. Otherwise, go to case 4.
- case 4: the stack with the largest minimal group index is determined as the destination stack for the container being relocated.

Especially applied to larger instances (beginning from 7x5 stacks x tiers, instances from Caserta et al., (8) and Zhu et al (20), GLAH outperforms the iterative deepening algorithm by Zhu et al (20) and the tree search procedure by Forster and Bortfeldt (15) while also reducing the computational effort very significantly.

Lin and Lee (25) introduce another noticeable heuristic and establish a comparison of it to that of Lee and Lee, showing up to 34% improvement of the objective function while decreasing CPU time by roughly five orders of magnitude over all instances. It is

also remarkable that this heuristic works for BRP with duplicate priorities. It takes the current group g (of items with the same priority) as target destined for retrieval. If all of the containers in g are retrieved, the group with the subsequent priority takes its place. Otherwise, among all of the containers in group g, a target container c – the one in the group g with the least number of overlaying containers – is selected. Ties are broken arbitrarily. Let $d = top(s(c))$ be the identification number of the container on the top of the stack in which c resides, and $g(d)$ is its group index. Now, if there exists at least one stack s, where $h(s) < H$ and $min_g(s) > g(d)$; then, container d can be moved to this stack without generating additional relocations, regardless of the arrangement of the existing containers in that stack. If more than one such stack exists, the one with the smallest $min_g(s)$ is selected. Alternatively, if there is no such stack, container d must be re-relocated at least one more time, no matter where it is relocated to at this moment. In this case, we select another stack u so that $min_g(u) + P_r(row(s))$ (this expression is also called stack selection index, SSI(s)) is maximized. The next important, foresighted feature of this heuristic is that if the selected stack t has available space for more than one container, a check is performed whether moving another container k into stack t before moving container d into it would be optimal. Such container k must satisfy the following four conditions: 1) Container k blocks the way of another container; 2) priority (=group index) of k is lower than that of d; 3) container k is located at the top of a stack and 4) $min_g(t)$– 5 < $g(k)$ < $min_g(t)$). If there is more than one such container, select the container with the largest identification number.

One of the most recent and promising metaheuristic frameworks for BRP is **rake search by Tricoire et al.** (17). Rake search is a method which tries to complete a partial solution with a quick construction heuristic. This algorithm starts as a breadth-first tree search method, which stops as soon as a level with more than a certain (pre-defined as a parameter) number of nodes is reached and continues each node with the application of various construction heuristics. The ones used by the authors were namely SM-1/2 and SmSEQ-1/2, developed by the authors as well (21) and described above (at the end of the section "Exact solution approaches"). Along with that, the authors also introduced a constructive metaheuristic called pilot method. Applied to BRP, it works as follows: a basic construction heuristic is applied to the initial state where every subsequent possible construction step is evaluated using rake search. The authors compare

these methods to GLAH (greedy look-ahead heuristic by Jin et al. (26), which was also described above) as this was the best-performing method to their knowledge and obtain very good solutions in very short CPU time. All three methods deliver identical results for very small instances (up to 4x4) while large instances, pilot method outperforms rake search but provides slightly higher values of the objective function (i. e. average number of relocations over 40 instances of each given size) than GLAH. Noticeable is that even for 10x10 instances, rake search needs less than a second on average, while pilot method needs about 43 seconds and GLAH about 48,5 seconds. Certainly, all these advanced methods are very efficient, but, as highlighted by authors, only up to a particular instance size. This is why in logistical settings with much larger numbers of items than a hundred methods fast methods like LA-N, SM-1/2, SmSEQ-1/2 are more appropriate. Tested on very large instances, SmSEQ-1/2 perform excellently.

Last but not least, it is indispensable to mention here a research on **the block relocation problem with batch moves by R. Zhang, S. Liu, and H. Kopfer** (10), as this BRP modification is particularly relevant to the main research question of this thesis. As already described in the introduction, the possibility of batch moves means that more than one item can be retrieved or relocated at a time. The possibility to lift more than one item creates immensely more possible moves at each state of the problem and therefore, brings much more complexity into the solution process. The existing BRP heuristics can hardly be implemented for a more or less efficient solution of this problem because they do not consider retrieving sequences of items at all. To elaborate, let us consider a small example of retrieving 5 items from a bay, first with only one item liftable at a time and then with a larger lifting capacity.

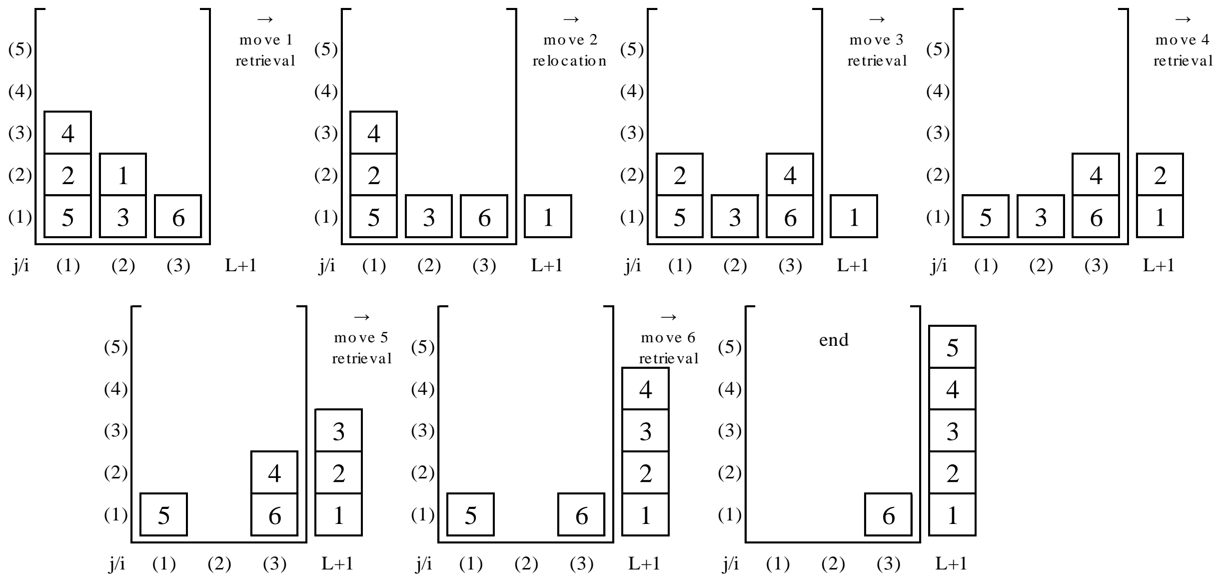


Figure 6. A small BRP instance without batch moves

Now, let us consider the same bay, the same lot and the same task – but this time the lifting and retrieving mechanism is able to lift up to 5 slabs of given size at a time. In this example, it results in the same number of moves as before (however, the travelled distance/time would certainly be smaller here because less movements are made to the assembly stack).

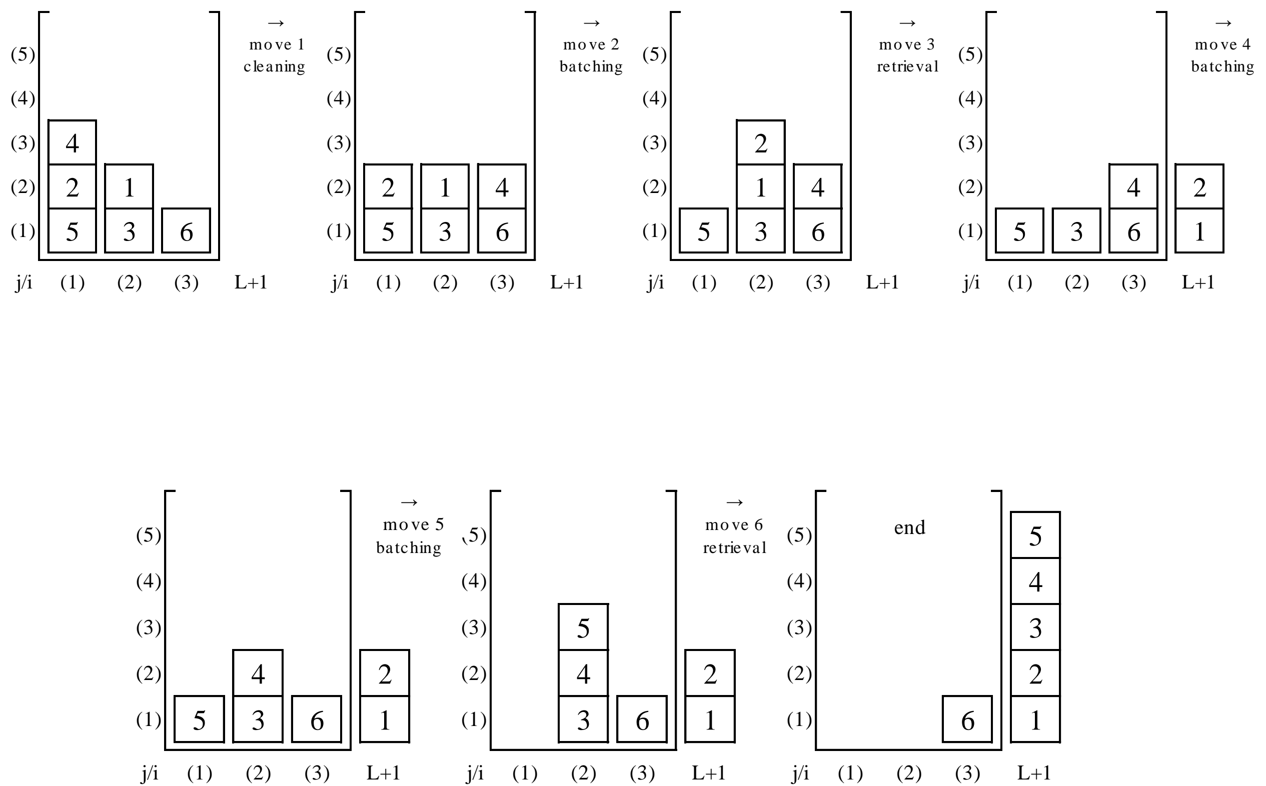


Figure 7. The instance from Fig. 3 solved with batch moves allowed

Another possible move sequence which results in only six moves is depicted below:

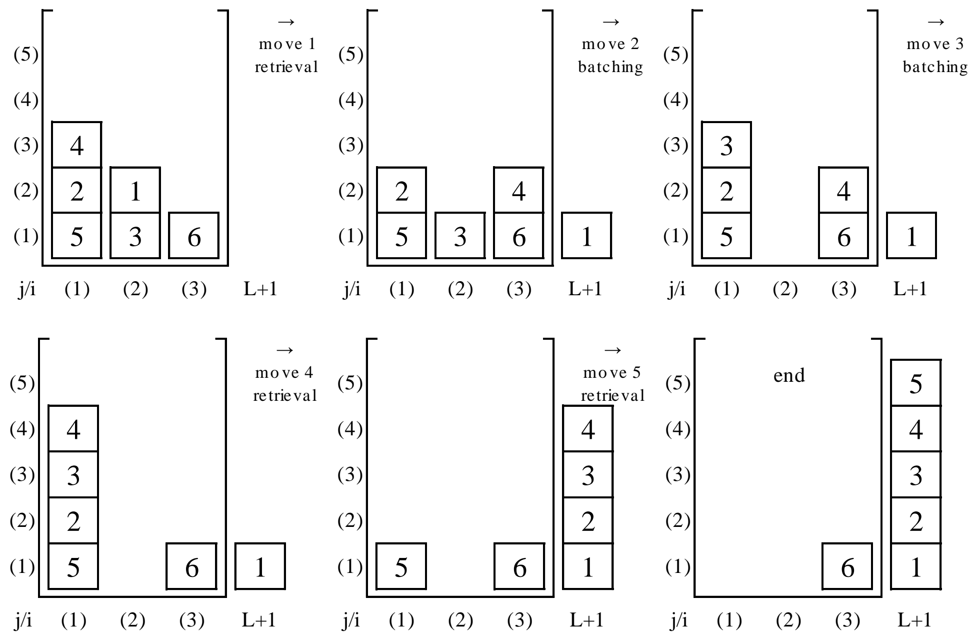


Figure 8. Alternative removal sequence

Further move sequences which also are optimal are illustrated below:

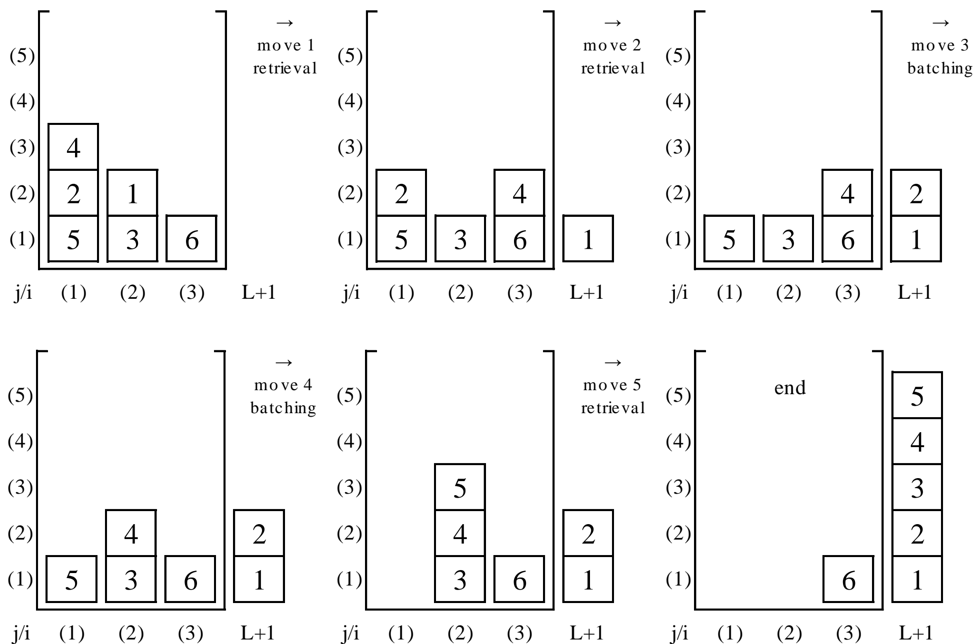


Figure 9. Second possible optimal removal sequence

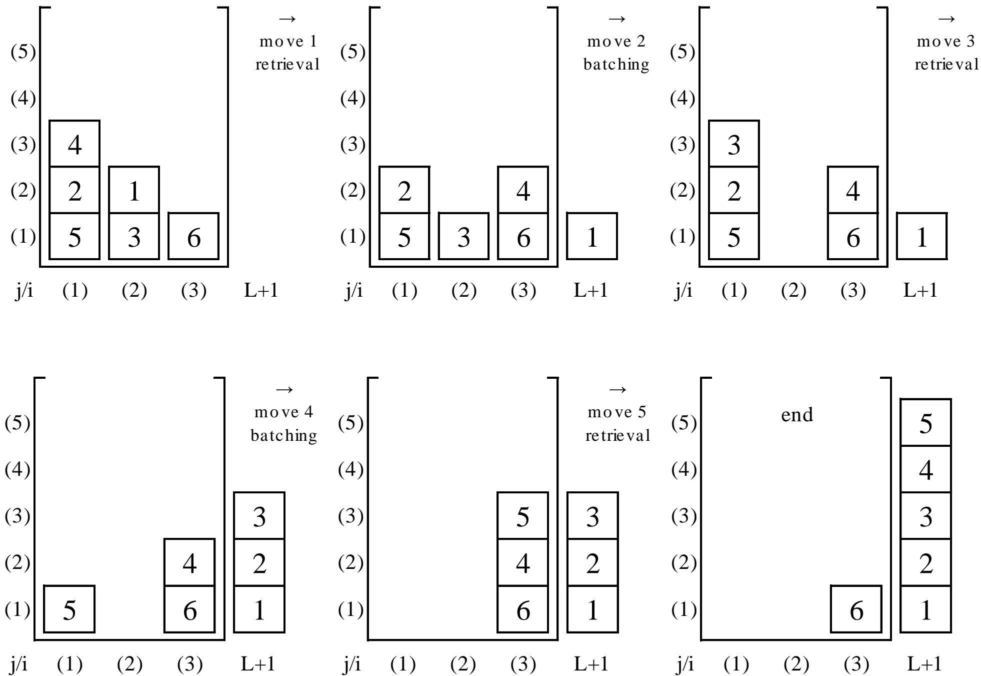


Figure 10. Third possible optimal move sequence

Thus, we can see the difference implicated by a vehicle which can lift multiple slabs: altogether, less moves are likely to be required. As already mentioned, in classical BRP, retrievals are usually not regarded as moves because the number of retrievals is fixed – equal to the number of slabs – and cannot be reduced because each retrieval can involve one slab only, which is why retrievals are not an object of the optimization, which, of course, does not mean that they require less energy or time than relocations. Here, on the contrary, the retrievals can be made more efficient by retrieving *increasing sequences*. This concept is similar to the concept of decreasing sequences by Tricoire et al. (21), but designated for retrievals rather than relocations. As demonstrated in the example, it is possible to retrieve multiple items simultaneously only in the exact retrieval order, counting from bottom to top. For instance, if the required retrieval order is (1, 2, 3), if these items are retrieved one by one, without implementing the use of batch moves, it would result in them being placed on the destination in an increasing sequence, counting from bottom to top. However, if those items were already placed on top of one another in that sequence and batch moves were possible, it would be optimal to retrieve them all at a time, which would result in one move instead of three. Therefore, presence as well as possibility of forming sequences of items in the exact retrieval order is worth being detected, evaluated and exploited.

In their 2015 paper “Tree search procedures for block relocation problem with batch moves” (10), the authors first introduce a specific way of calculation of the lower bounds on the number of relocations and retrievals for this special case of BRP. The sum of these equals the lower bound on the total number of moves, as demonstrated by the authors. First, a greedy algorithm which chooses the best move at each state depending on the priorities of all possible moves is presented. The priorities of relocations are based on following notions (the definitions of well-/ badly-placed items and BG-/ GG- moves are analogical to that of Forster and Bortfeldt (15)):

- if all the relocated items were badly-placed before the relocation and are well-placed after the relocation, it is classified as a Bad-Good (BG) relocation. A BG relocation is considered as a BG-Combinable (BGC) relocation if the top item of the target stack is the direct successor of the lowest relocated item in terms of retrieval (i.e. its index equals the index of the lowest relocated item plus one). Otherwise, a BG relocation is BG-Non-Combinable (BGNC)

-a relocation of items which currently block the smallest item in the whole layout (i. e. reside on top of it) is called a Free-Smallest-Item (FSI) relocation. Again, a FSI relocation is defined as a FSI-Combinable (FSIC) relocation if the index of top item of the target stack equals the index of the lowest relocated item plus one, or a FSI-Non-Combinable (FSINC) relocation otherwise.

- a relocation is defined as a Good–Good (GG) relocation if all the relocated items were well-placed before the relocation and remain so after it.

Different types of moves are prioritized as follows:

I. Retrievals
II. BGC
III. BGNC
IV. FSIC
V. FSINC and GG

In case of several possible moves with same priority, the one with the largest number of items should be chosen.

The greedy algorithm consists of the following main steps:

1. Let L be the initial layout and x_{greedy} be an empty sequence of moves. Set the current move priority to P = 1.

2. If there is a move with a priority P applicable to layout L, let m be this move (if several such moves exist, set m as one with the maximum number of items among them) and go to Step 4. Otherwise, go to Step 3.
3. $P \leftarrow P + 1$; go to Step 2.
4. $x_{greedy} = x_{greedy} + m$. Apply m to L. If L is not empty, let $P = 1$ and return to Step 2. Otherwise, $x_{greedy}$is the solution of the BRP-BM.

The next, more complex algorithm developed by the authors is a depth-first incomplete tree search algorithm based on compound moves (named ICC-TREE). At each node, a pre-defined number of possible compound moves is evaluated according to their length (i.e., the number of moves in the respective compound move) plus the lower bound (of the number of moves) of the new layout which would result from this move. The set of compound moves for a current layout is generated according to the following procedure (referred to by the authors as the inner search of ICC-Tree): First, a set of productive moves are generated for any given layout. These are the moves with the highest priority among all moves applicable to the layout. If the highest priority is 5, the maximum numbers of FSINC relocations and GG relocations, respectively, are considered separately. For each productive move, beginning with ones with the highest priority and proceeding with regard to decreasing priorities, compound moves are generated. The compound moves at each node are limited by a pre-defined number and sorted before further branching. If a given time limit is reached or an optimum solution is provided, the algorithm terminates.

The procedure COMPOUND, which is the inner search of ICC-TREE, is a recursive function with initial arguments x^{tree}, x^{part} (both of these delivered by the main ICC-TREE function), current layout L, an initially empty move sequence $m_{compound}$, $n^{cm} = 1$ and empty set Θ, destined for the compound moves for the current layout. Its main steps are defined as follows:

1. If the given time limit is reached, return.
2. $P = 1$.
3. If $P < 5$, go to Step 4; otherwise, let K be the set of FSINC relocations and GG relocations applicable to L, where the maximum number of FSINC relocations and that of GG relocations have a given threshold, i.e., $n_{fsinc-max}$ and n_{gg-max}, respectively. Go to Step 5.

4. If there is no move with a priority P applicable to L, let $P = P + 1$ and go to Step 3; otherwise, let K be the set of moves with priority P applicable to L.
5. Let m be the first move in K.
6. If layout L after the application of move m is empty, append $m_{compound} + m$ to Θ, which is the set of compound moves applicable to L, and go to Step 9.
7. If the sum of length of the current incomplete solution and length of $m_{compound} + m$ plus the lower bound of layout $L \cup m$ is not smaller than the length of the optimum-so-far solution, ignore m and go to Step 9.
8. If the number of compound moves generated so far (for the current layout) has already reached given limit, append $m_{compound} + m$ to Θ and go to Step 9; otherwise, call COMPOUND with arguments x^{tree}, x^{part}, $L \cup m$, $m_{compound} + m$, n^{cm} multiplied by the total number of moves in K, and Θ.
9. If m is not the last move in K, go to Step 6.

Additionally, the authors present two other tree search algorithms which, unlike ICC-TREE, do not branch on compound moves. These are called IC-TREE (which stands for *incomplete tree search*) and C-TREE (*complete tree search*), respectively. It is stated that they are therefore similar to a degenerated version of ICC-TREE, which is ICC-TREE without branching on compound moves.

For branching, IC-TREE considers the lower bound of the new layout (and not priorities of moves at each node, as ICC-TREE does), choosing only the moves which have minimum lower bound of the corresponding new layout. As the moves which have the same minimum lower bound are not very numerous, there is no further limit on number of branches at each node. IC-TREE requires the same arguments as ICC-TREE and has three main steps:

1. If length of the optimum-so-far solution x_{tree} is not larger than the lower bound on the total number of moves or the given time limit is reached, return optimum-so-far solution.
2. Let Θ be the set of moves applicable to the current layout L with the minimum lower bound.
3. For each move m in Θ, if $L \cap m$ is empty and length of $x_{part} + m$ is shorter than the optimum so-far solution x_{tree}, set x_{tree} to $x_{tree} + m$. If move m does not empty the current layout, set L to $L \cap m$ and x_{part} to $x_{part} + m$ and go to step 1.

Finally, the C-TREE-algorithm has following particularities compared to IC-TREE: it implements no given limit of running time and all moves which can potentially provide an optimum solution are considered at each node. Therefore, C-TREE can provide an optimum solution of the BRP-BM if it finishes. C-TREE has the same arguments as IC-TREE and has the following main steps:

1. If length of the optimum-so-far solution x_{tree} is not larger than the lower bound on the total number of moves or the given time limit is reached, return optimum-so-far solution.
2. Let Θ be the set of moves applicable to the current layout L.
3. Sort the moves $m \in \Theta$ according to the lower bound of $L \cap m$ in ascending order. Ties are resolved by preferring the move with a source stack that has the minimal lowest-index item. If the moves in question have the same source stack, the one is preferred of which the target stack is not the one containing the lowest-index item in the layout.
4. For each move $m \in \Theta$, if $L \cap m$ is empty and length of $x_{part} + m$ is shorter than the optimum so-far solution x_{tree}, set x_{tree} to $x_{tree} + m$. If move m does not empty the current layout, set L to $L \cap m$ and x_{part} to $x_{part} + m$ and go to step 1.

The authors test their methods on their own randomly generated instances, as well as the 840 Caserta instances. The first 15 cases by Zhang, Liu and Kopfer are medium-sized, with number of stacks ranging from 5 to 10 and maximum stack height from 4 to 10, which corresponds to the usual size of problems in many real-life scenarios. As also noted by the authors, the occupancy rate of stacks of 0.5 and lifting capacity of 3 are also common in real-world instances, such as hot rolling slab yards of some steel plants. All these 75 instances are solved using the greedy and C-TREE algorithms. Both the calculation of the lower bound and run of the greedy algorithm for each instance take less than one second.

C-TREE could provide optimum solutions of small-sized instances and even some medium sized instances in a short running time (63 out from 75 optimum solutions). However, for larger instances the running times can extend to up to 1 hour. Expectedly, for larger-sized instances, the greedy algorithm and C-TREE are not as effective as for smaller-sized instances (C-TREE can still find optimal solutions for instances sized up to 6x10 within less than an hour). However, IC-TREE provides a solution faster also only for instances up to 6x10. ICC-TREE, on the other hand, delivers considerably faster

high-quality solutions for larger-sized instances, optimality achieved within 63 cases out of 75. As regards the results of testing on the 840 Caserta instances, the authors indicate that ICC-TREE provides good objective values within the range of 10-13% above the lower bound for small instances (3x3 to 4x7), 21-23% for medium and 27-39% for large instances (5x4 to 5x10 and 6x6 to 10x10, respectively). While considering different values of lifting capacity (M), the authors determined that the average relative lower bound over the 840 instances increased to 1.06 and then to 1.42 with M set to 2 and then to 1. The average relative objective value provided by ICC-TREE increased to 1.07 and then to 1.34, respectively. Furthermore, the comparison of the performance of ICC-TREE M set to 1 on the Caserta instances to that of other existing standard BRP methods resulted in ICC-TREE providing the best average solution for 5 instance sizes (4x5, 4x6, 4x7, 5x4 and 5x7), whereas for other medium and small instances it still provided an objective value only slightly larger than the best result among the other heuristics. When it comes to large instances, the best performing heuristic by far is LA-N by Petering and Hussein (14). Of course, ICC-TREE was designed for BRP with batch moves in the first place, so its good performance for standard BRP is remarkable.

3. The steel slab stacking problem (SSSP)

As already mentioned in the introduction, the BRP scenario where the lifting mechanism is able to retrieve or relocate more than one item at a time requires a different solution approach than the case of only single-item retrievals and relocations. Nevertheless, such problem settings can occur quite often in real-world settings outside of steel slab yards. For example, more advanced quay cranes in container terminals often have multiple spreaders and double trolleys for picking multiple containers at the same time. A leading example of this thesis, however, will be the BRP that occurs in steel slab yards.

Steel slabs are semi-finished casting products of steel producing plants. Each slab is characterized by its width, length and height, i.e., thickness, as well as its weight. Due to their varying metallurgical composition, slabs may have a different weight even if they have identical dimensions. Thus, all slabs are considered to be heterogenous. After their initial formation they are stored in an open-air area (“field”) in stacks on top

of one another in what can be described an arbitrary order. Due to production specifics it is not possible to put the slabs in the optimal order from the beginning, because the order in which they are produced usually differs from the order in which they are further processed and hence picked up for delivery. The pickup is carried out by heavy-duty vehicles (also referred to as straddle carriers) which navigate over rows of slabs and retrieve them from above. Straddle carriers can lift multiple slabs up to a certain total weight cap_{weight} and total height cap_{height}. An empty vehicle can navigate over a particular maximum stack height, whereas each slab held by the vehicle reduces this particular maximum height by its own height. This circumstance will be referred to as the *movement restriction* hereafter. This restriction is the reason for limiting the stack height to H_{max} (1980 mm in our case) such that the vehicle holding one average-sized slab (referring to a height of 220 mm) could still pass over. Thus, if a particular stack height together with the total height of the slabs held by the straddle carrier exceeds 2200 mm, the vehicle is not able to drive over this stack. In such situations, the stacks which are currently preventing the vehicle from passing over them on its way to its destination are known as *blocking stacks*. We shall use this notion hereafter.

Figure 11. Straddle carriers moving over rows of slab stacks. (source: voestalpine Stahl GmbH)

As the vehicles can only navigate forwards or backwards within a row, each row is considered as a separate instance (two-dimensional, with a particular number of stacks and a different number of slabs in each stack). Every row containing items which have to be retrieved has an ordered retrievals list R associated with it, so that retrieving all the slabs listed in R in the required sequence is the solution of the problem for the

respective row. From the assembly stack, the slabs are continuously picked up by another straddle carrier and transported to their destination. When all the required slabs have been retrieved from a row, the same procedure is done for the next row on the schedule, and so on. Given that in this problem every slab which is to be retrieved has a particular place in the retrieval sequence, we will, from now on, use the notions “next slab” or “immediately adjacent successor of slab s” to refer to the slab which is directly after the slab s on the retrieval list. Similarly, the notions “previous slab” or “immediately adjacent predecessor of slab s” will mean the slab which is directly after the slab s on the retrieval list

With regard to the retrieval order and of the numbering of slabs in a bay, it is important to mention that the numbering in a bay is not continuous. Each slab has a number which, generally speaking, correlates with its priority in the same way as it is common for the BRP: the smaller the index, the higher the priority. Yet, the intervals between those numbers can be arbitrary. It is quite likely that slabs with adjacent numbers have also been produced or delivered in the same run and/or have some structural similarities. However, slab numbers usually vary strongly within a bay and within a retrieval order. For example, a retrieval order may contain slab numbers 61, 62, 105, 110, 204, 659.

Another particularity of our SSSP is the existence of slabs with no known priority value, which means that they do not have to be retrieved anytime soon in the future. Such slabs are always numbered with ***-1*** and are always assumed to have the lowest priority. Yet, they sometimes may be part of a retrieval list R, in which case they will be the last ones to be retrieved.

If a row has L stacks, then the stack L+1 is the assembly stack. The slabs have to be assembled on this stack strictly in the order defined by the required retrieval order, counting from the bottom to the top. The size of the batch is determined by the vehicle capacity: the total weight and height of all slabs lifted at a time must not exceed cap_{weight} and cap_{height}, respectively, which are 104000 kg and 880 mm in our particular case of the slab yard.

As briefly mentioned above, there are *incompatibility restrictions* between certain slabs, which means that they cannot be put on top of one another because of their difference in dimension. A slab may not be placed above any other slab which is more than 150 mm narrower or 3000 mm shorter, be it within a stack or a batch. This

restriction holds true for all stacks in the initial state of the bay and in any transitionary state resulting from a relocation, as well as for batches and any slabs that are lifted together. For example, if there is a 900 mm wide slab in a stack, no slab wider than 1050 mm can be placed on top of this stack. Similarly, if multiple slabs are lifted, no slab may be more than 150 mm wider or 3000 mm longer than any of the slabs below it. The reason for these restrictions is the risk that structures of stacks where considerably longer and/or wider slabs reside on top of shorter and/or narrower ones may be easily destabilized and may collapse as a result.

Summing up the particularities of the SSSP compared to the standard BRP:

- Possibility of lifting multiple slabs up to a certain total weight cap_{weight} and height cap_{height}
- Heterogenous slabs with different dimensions; therefore, the height H_i of a stack i is measured in mm and not in units (as opposed to the standard BRP, where all items are homogenous and have the same dimensions, so the number of items in a stacks determines its height unambiguously)
- *Incompatibility restrictions* and *movement restrictions*
- Possibility of making better batching decisions by detecting or creating increasing sequences
- Not all slabs from a bay have to be retrieved

The problem specifics of the SSSP, compared to the standard BRP, introduce alternative paths in the solution process, thus increasing the solution space and requiring a comprehensive and cumbersome algorithm. Not considering all those possibilities can certainly lead to suboptimal solutions, i. e., making unnecessary retrieval moves and relocations. Therefore, a combined algorithm comprising target identification, unblocking, cleaning and retrieving moves has been developed and implemented in C++ programming language. The pseudocode of a suggested algorithm is presented in the following chapter and the C++ code, along with the instances tested upon, is available as an appendix to this thesis.

4. The Combined Multi-Target Algorithm (CMTA)

We propose a heuristic solution approach titled the Combined Multi-Target Algorithm (CMTA). It iteratively performs following actions until the retrieval list R is cleared, where in each iteration n a subset of retrieval list R is retrieved:

- Definition of the current targets (the first n slabs of the retrieval list R that can be retrieved at a time, hereafter noted as current batch β). This means that the slabs from R are added to β one by one until adding the next slab from R would make the size of β exceed the vehicle capacity (cap_{weight} and height cap_{height}) or violate *incompatibility restrictions*. The first target slab s^* has to be retrieved during the current iteration in any case. It is defined as *initial set* I in the algorithm.
- Detection of *increasing sequences* which are part of the current batch β. If such are detected, they are considered as a set from this point on, not as individual slabs (in terms of unblocking and retrieval). For example, if the direct successor(s) of s^* are situated directly on top of s^*, it (they) is (are) added to initial set I as well.
- Performance of efficient moves to unblock the initial target set I, if necessary.
- Unblocking of further members of current batch β, by performing *good relocations* only, if possible.
- Ensuring that the initial target is retrievable with regards to *movement restrictions* by reducing the heights H_i of *blocking stacks*, if there are any.
- Ensuring the possibility of retrieval of further targets from batch β with regards to *movement restrictions* by reducing the heights H_i of *blocking stacks*, if there are any, using only *good relocations*. If this is not possible, the remaining slabs of the current batch are retrieved in the next iteration $n + 1$, i. e., after the vehicle returns to retrieve the slabs from the retrieval list R that have not been retrieved yet.

The outline of CMTA is as follows:

0. Initialize current batch $\beta \leftarrow \emptyset$.
1. Let s^* be the first slab of the retrieval list R. Check whether adding $s^*$to β does not make the total height and weight of batch β exceed the vehicle's height and weight capacity (cap_{weight} and cap_{height}) and whether it does not violate *incompatibility*

restrictions. If both conditions are satisfied, add s^* to β and proceed to 1.1. Otherwise, proceed to 1.2.

1.1. Let s' be the immediately adjacent successor of s^* on the retrieval list R. Check if adding s' to β does not make the total height and weight of batch β exceed the vehicle's height and weight capacity (cap_{weight} and cap_{height}) and if it does not violate *incompatibility restrictions*. If both conditions are satisfied, add s' to β, let $s^* \leftarrow s'$ and repeat this step. Else, proceed to step 1.2.

1.2. Initialize the initial target set I. Identify the first slab s^* of the current batch β and add it to I.

1.3. Let s' be the immediately adjacent successor of s^* on the retrieval list R. If s' is located directly on top of s^*, let $s^* \leftarrow s'$ and add s^* to I (because it is preferred to retrieve them together) and repeat step 1.3. Otherwise, go to step 1.4.

1.4. Should the initial target set I not be on the top of its stack (i. e., if there are any slabs on top of it, preventing it from retrieval), remove those slabs by relocating them efficiently (using function $unblockTarget1$)

1.5. Initialize the list **of current_targets** $\leftarrow \{\boldsymbol{I}\}$, where **current_targets** is a list of sets of slabs. A set within **current_targets** can contain either one slab or an increasing sequence (resulting from step 1.3., if applicable).

2. If all slabs of the current batch β have either been retrieved or included into **current_targets** already, or if all slabs in **current_targets** already use up the whole cap_{weight} and/or cap_{height} , go to step 3. Should both conditions be false, locate the next slab to be retrieved from current batch β which is not in **current_targets** yet. Let us denote this slab with k. If *movement restrictions* allow to retrieve k together with all the slabs that already are in **current_targets**, initialize **additional_set**, consisting of k only (**additional_set** $\leftarrow \{k\}$) and proceed to step 2.1. Else, go to step 3.

2.1. Let the slab residing directly on k be k'. Check if k' does not happen to be the next slab which is to be retrieved (i.e., the immediately adjacent successor of slab k on the retrieval list R). If no, go to 2.3. If yes, check for following conditions:

2.1.1. whether the vehicle capacities cap_{weight} and cap_{height} are not exceeded by the total height and weight of the all the slabs in **current_targets** ∪ **additional_set** ∪ $\{k'\}$

2.1.2. whether slabs in **current_targets** ∪ **additional_set** ∪ $\{k'\}$ are retrievable with regard to *movement restrictions*

2.2. If the conditions 2.1.1. and 2.1.2. apply, add k' to the **additional_set** and let $k \leftarrow k'$. Go to 2.1. If those conditions do not apply, add the current **additional_set** to the list of **current_targets** and go to step 3.

2.3. Add the current **additional_set** to the list of **current_targets.** Go to 2.

3. Unblock additional sets (if any) from the list of **current_targets** one by one if needed, as long as it is possible using *good relocations* only. Begin with the second set of the list **current_targets**, as the first one is the initial target set I which has already been unblocked, if needed, in step 1.4. If **current_targets** only contains one set, go to step 4. Otherwise, let us denote the second set of **current_targets** with λ. For each set from **current_targets,** starting with λ and moving towards the end of the list, proceed through following steps:

3.1. Check if set λ is on the top of its stack. If yes, update λ to be the next set on the list **current_targets** and repeat step 3.1. If no, go to 3.1.1. If the end of **current_targets** is reached, go to 4.

3.2. Check if it possible to relocate all the slabs on top of λ by relocating them by means of *good relocations* (function $unblockTarget2$) only.

3.2.1. If the condition in 3.2. is true, employ $unblockTarget2$ let λ be the next set on the list of **current_targets** and go to 3.1.

3.2.2. If the condition in 3.1.1. is false, remove λ and all the sets that follow λ in **current_targets** from the list **current_targets** and go to step 4.

4. Now the list **current_targets** contains only the slabs of the current batch β which are currently free to be retrieved. Again, the list **current_targets** consists of at least one set of slabs. A set of slabs can contain either a single slab or an increasing sequence of slabs. However, it still has to be checked if, after the eventual unblocking procedures of step 3.1.1., all the targets can be retrieved together without violating the *movement restrictions*. Proceed to 4.1.

4.1. Let the first set on the **current_targets** list (the initial target set I) be σ. Check if there are any stacks between the stack where σ resides and the assembly stack $L+1$ which are too high for carrying σ above them with regard to *movement restrictions* (i.e. *blocking stacks*). If so, relocate the required number of slabs from these stacks (function $reduceHeight1$, tbd). Initialize the **final_targets** list

with **final_targets** ← $\{ I \}$, where **final_targets** is a list of sets of slabs. Proceed to 4.2.

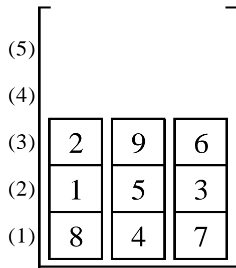
4.2. If the end of the **current_targets** list has already been reached, go to step 5. Else, update σ to be the next set of slabs on the **current_targets** list and check if σ can be retrieved with regard to the *movement restrictions*. If so, add σ to the **final_targets** list. Else, check if it is possible to reduce the height of all the *blocking stacks* using only *good relocations* by calling the boolean function $reduceHeight2$. If this is feasible, the function returns *true* and performs all the necessary relocations in the best possible manner. If not, the function returns *false*.

4.2.1. If the function $reduceHeight2$ returns *true*, add σ to the **final_targets** list. Go to 4.2.

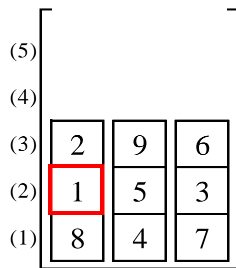
4.2.2. If the function $reduceHeight2$ returns *false*, or if all targets from the **current_targets** are already added to the **final_targets** list, go to step 5.

5. Pick the sets of slabs from the **final_targets** list one by one, beginning with the last one. After the whole sequence has been picked, move it to the assembly stack $L + 1$. Delete the retrieved slabs from the current batch β and from the retrieval list R.
6. If R is not empty, go to step 1. Otherwise, terminate.

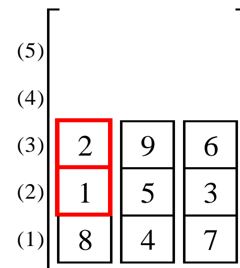
We exemplify the performance of the on a small BRP instance below, where we consider homogenous slabs and neglect incompatibility restrictions, for simplification purposes. Note that the maximal stack height H_{max} of 5 means that the vehicle can pass over a stack of this height while carrying one slab. Should it carry more slabs, the number of slabs carried should be subtracted from H_{max} to calculate the maximal stack height over which the vehicle can move. This allows us to simplify the movement restrictions. Given is R= {1,2,3,4,5} and H_{max} =5.



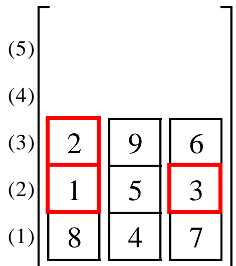
0. $\beta \leftarrow \emptyset$
1. β = {1}
1.1. repeated 4 times: adding slabs 2, 3, 4, 5 to β since vehicle capacity allows it. Now β = {1,2,3,4,5}



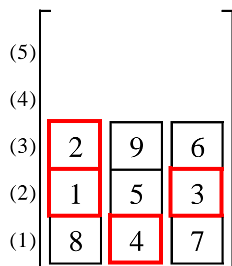
1.2. s^*=1; I = {1}



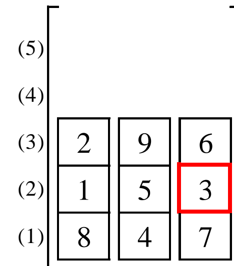
1.3. s'=2; I ={2,1}
1.4. n/a
1.5. **current_targets** ={ {2,1} }



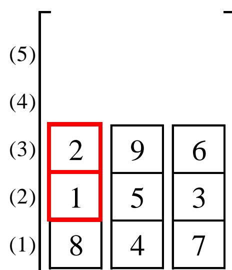
2. k = 3. **additional_set**={3}
2.1. k' = 6 (and not 4), therefore
2.2. is n/a
2.3. **current_targets** = { {2,1}, {3} }
Repeat step 2 (since vehicle capacity still allows to add a further slab



2. k = 4. However, this slab cannot be retrieved together with all the slabs that already are included into **current_targets** due to *movement restrictions*, therefore, go to step 3.

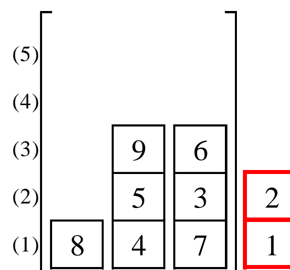


3. λ={3}.
3.1. λ is not on top of its stack
3.2. As λ cannot be unblocked using *good relocations only*, the 3.2.1. is not applicable. We remove λ from **current_targets** and go to step 4

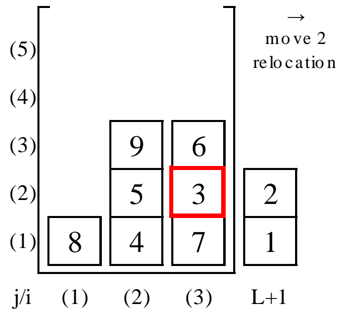


→
move 1
retrieval

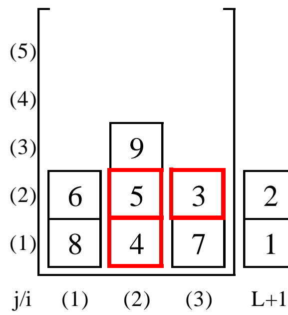
4. Checking the possibility of retrieval of **current_targets** = { {2,1} } with regard to *movement restrictions*.
4.1. **final_targets** = {2,1}. Retrieval is possible.
4.2. As there are no further sets in **current_targets**, proceed to step 5.



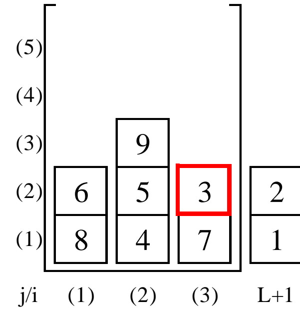
5. Retrieve {2,1} and remove slabs 1,2 fromβ and R. $\beta = \{3,4,5\}, R = \{3,4,5\}$
6. Since $R \neq \emptyset$ yet, go to step *1.*



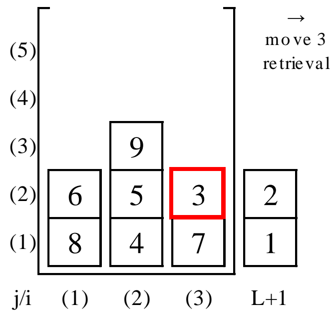
1. β = {3,4,5}.
1.1. n/a
1.2. I={3}
1.3. n/a
1.4. As there is a slab (6) on top of I, relocate it. *Good relocation* is possible to stack 1.
1.5. **current_targets**={ {3} }



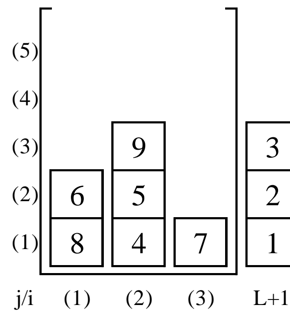
2. k = *4*. **additional_set** ={4}
2.1. k' *=5* (which is the direct successor of k = 4, therefore, check conditions 2.1.1.-2.1.2.)
As they do apply, let k = *5* and **additional_set** ={5,4}. Go to 2.1.
2.1. k' *=6* (which is not the direct successor of k = *5*, therefore, go to 2.3.)
2.3. **current_targets**={ {3}, {4,5} }



3. λ = {5,4}.
3.1. λ is not on top of its stack
3.1.1. not possible to unblock λ using *good relocations* only;
3.1.2. is therefore not applicable.
3.1.3.: **current_targets**={ {3} }.
Go to step 4



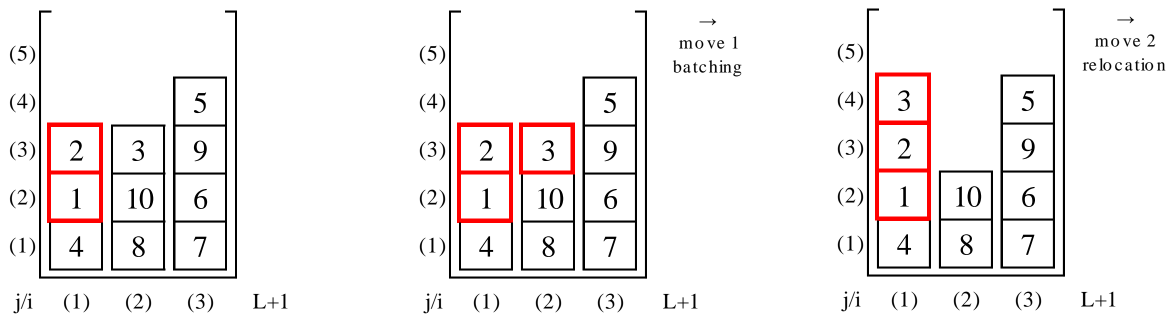
4. Check if **current_targets** are retrievable with regard to *movement restrictions*.
4.1. Check affirmative. **final_targets** = { {3} }.
4.2. As there are no further sets in **current_targets**, go to step 5.



5. Retrieve {3}, delete slab 3 from β and R.
6. Since $R \neq \emptyset$ yet, go to step 1.

Figure 12. CMTA performed on a small BRP instance

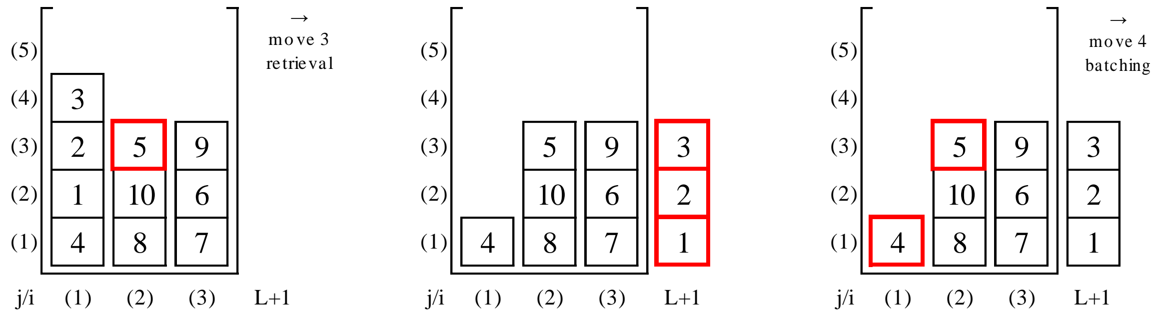
Another small example of a situation where more slabs can be retrieved at a time and some moves are performed to enable the retrieval with regard to movement restrictions is presented below. Same as before, R= {1,2,3,4,5} and H_{max} = 5.



0. $\beta \leftarrow \emptyset$
1. β = {1}
1.1. repeated 4 times: adding slabs 2, 3, 4, 5 to β since vehicle capacity allows it. Now β = {1,2,3,4,5}
1.2. s^* = 1, I ={1}
1.3. s'=2 is located directly on s^* → I ={2,1}
1.4. I is free to retrieve.
1.5. **current_targets** = { {2,1} }

2. k=3. **additional_set** ={3}
2.1. There is no slab residing on k → go to 2.3.
2.3. **current_targets**={ {2,1}, {3} }. Go to 2.
2. k = 4. Since k would not be retrievable together with all the slabs which already are in **current_targets,** go to step 3.
3. λ = {3}.
3.1. λ is on top of its stack and it is the last set in **current_targets**→ go to step 4.
4. Check if all **current_targets** can be retrieved together without violating the *movement restrictions*.
4.1. σ = {2,1} is retrievable → **final_targets** = {2,1}

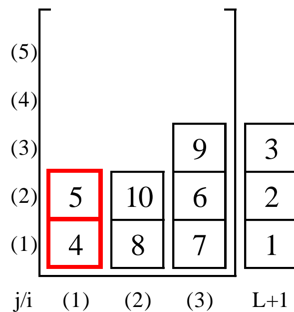
4.2. σ ={3} would be retrievable together with **final_targets** = {2,1} if the height of stack 3 was reduced by one slab.
4.2.1. As the height of stack 3 can be reduced by moving slab 5 to stack 2 after batching slab 3 with {2,1} and it would be a *good relocation*, $reduceHeight2$ returns true as it performs batching moves and relocations.
final_targets = { {3}, {2,1} }.



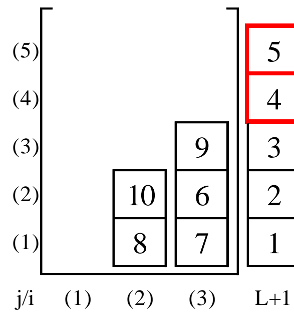
4.2.2. As there are no further sets in **current_targets**, go to step 5.

5. Retrieve {3,2,1}, delete slabs 1, 2 and 3 from β and R.
6. Since $R \neq \emptyset$ yet, go to step 1.

1. β = {4,5}.
1.1. n/a
1.2. $s^* = 4$, I={4}
1.3. $s' = 5$, which is immediately adjacent successor of s^*. Therefore, I={5,4}
1.4. n/a
1.5. **current_targets**={ {5,4} }



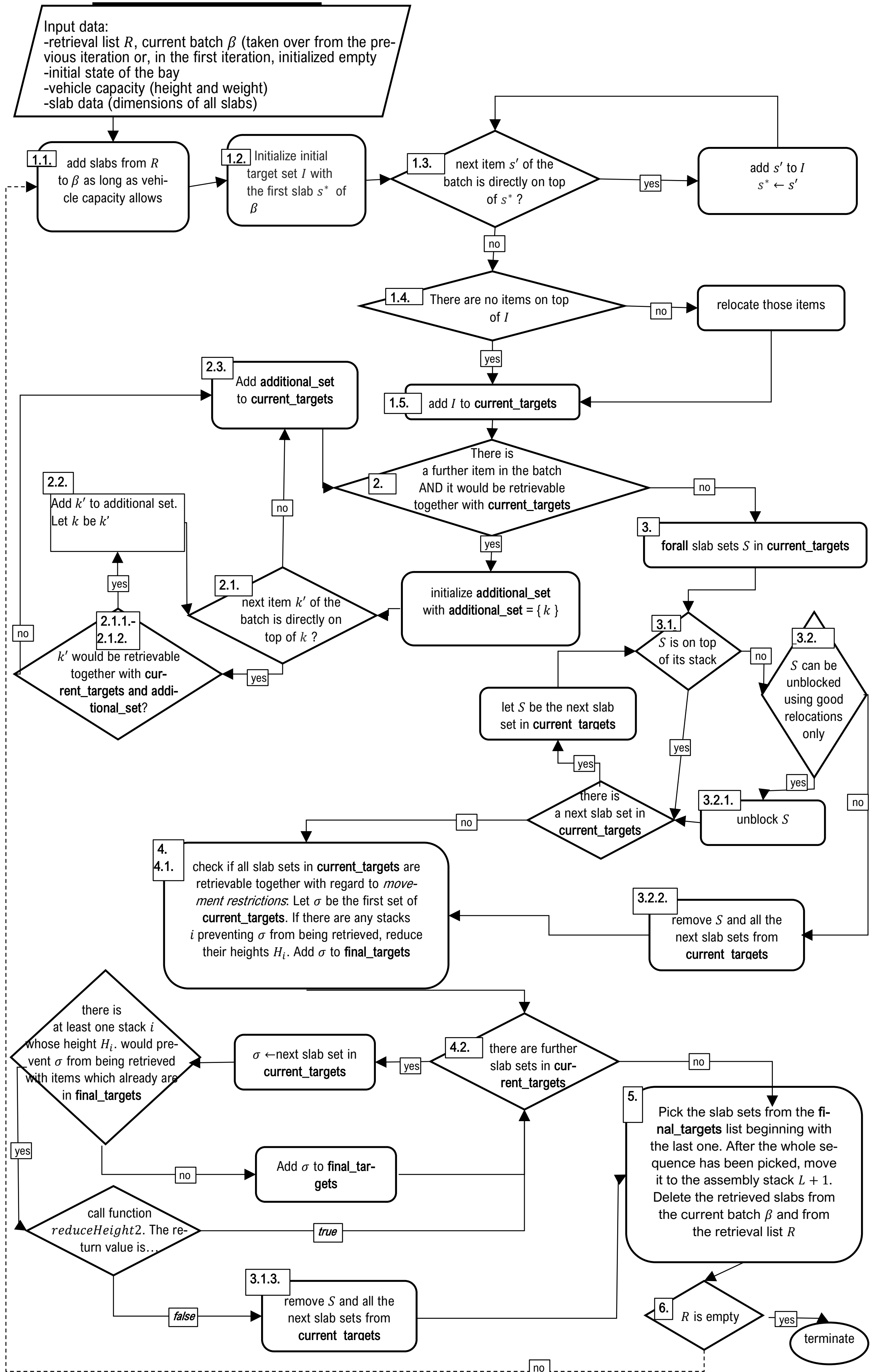
2. As there are no further targets in β, go to step 3.
3. λ = {5,4}.
3.1. λ is on top of its stack, no unblocking needed. Go to step 4.
4. σ = {5,4}.
σ is retrievable with regard to *movement restrictions*. **final_targets** ={5,4}.
4.2. As the end of the **current_targets** list has already been reached, go to step 5.



5. Retrieve {5,4}, delete slabs 4 and 5 from β and R.
6. Since R is empty, terminate.

Figure 13. Another example of CMTA performed on a small BRP instance.

To visually illustrate all steps of the CMTA, we include a flowchart in Figure 13 below.



Input data:
-retrieval list R, current batch β (taken over from the previous iteration or, in the first iteration, initialized empty
-initial state of the bay
-vehicle capacity (height and weight)
-slab data (dimensions of all slabs)
1.1. add slabs from R to β as long as vehicle capacity allows
1.2. Initialize initial target set I with the first slab s^* of β
1.3. next item s' of the batch is directly on top of s^* ?
yes
add s' to I $s^* \leftarrow s'$
no
1.4. There are no items on top of I
no
relocate those items
yes
1.5. add I to **current_targets**
2.3. Add **additional_set** to **current_targets**
2. There is a further item in the batch AND it would be retrievable together with **current_targets**
no
3. **forall** slab sets S in **current_targets**
yes
initialize **additional_set** with **additional_set** = { k }
2.1. next item k' of the batch is directly on top of k ?
no
yes
2.2. Add k' to additional set. Let k be k'
yes
2.1.1.-2.1.2. k' would be retrievable together with **current_targets and additional_set**?
no
3.1. S is on top of its stack
no
3.2. S can be unblocked using good relocations only
yes
3.2.1. unblock S
no
let S be the next slab set in **current_targets**
yes
there is a next slab set in **current_targets**
yes
no
4. 4.1. check if all slab sets in **current_targets** are retrievable together with regard to *movement restrictions*: Let σ be the first set of **current_targets**. If there are any stacks i preventing σ from being retrieved, reduce their heights H_i. Add σ to **final_targets**
3.2.2. remove S and all the next slab sets from **current_targets**
4.2. there are further slab sets in **current_targets**
yes
σ ←next slab set in **current_targets**
there is at least one stack i whose height H_i. would prevent σ from being retrieved with items which already are in **final_targets**
no
Add σ to **final_targets**
yes
call function $reduceHeight2$. The return value is…
true
false
3.1.3. remove S and all the next slab sets from **current_targets**
no
5. Pick the slab sets from the **final_targets** list beginning with the last one. After the whole sequence has been picked, move it to the assembly stack $L+1$. Delete the retrieved slabs from the current batch β and from the retrieval list R
6. R is empty
yes
terminate
no

Figure 14. CMTA in a flowchart

Before we elaborate the most important functions from CMTA, let us present the notation of variables and functions used in the pseudocode:

- $upper(slab\ x)$ or $upper(set\ S)$: the slab which lies directly on the slab x or slab set S
- $top(set\ S)\ or\ top(stack\ t)$: the top slab of a set S or stack t
- $bottom(set\ S)\ or\ bottom(stack\ t)$: the bottom slab of a set S or stack t
- $stack(slab\ x)$: the stack in which the slab x is situated
- $slab(p)$: the slab with the index p
- $currentLotSize$: the number of slabs in the current lot
- $number\ (slab\ s)$: the position of a slab s in its stack, counting from the bottom of the stack
- $number(stack\ t)$: the number of stack t in the bay
- $targetStack$: the number of the stack where the target with currently lo is next to be retrieved in its stack, counting from the bottom of the stack
- $targetSlabPosition$: the position of the slab which is next to be retrieved in its stack, counting from the bottom of the stack
- $commonSlabs(set\ S, stack\ t)$: Slabs which are still on top of their stacks but also a part of the current target set/the set of which the feasibility of retrieval is considered. While considering the height of the stacks on the way as to if they may block the retrieval, the height of any slab which are on the top of stack being considered but also in the current target set (addition) should be deducted from its overall height.
- $lastSlabs$: a set to record the last slab of each *blocking stack* which should be left on it in order to reduce its height to the required one
- k: initial counter for the last of the slabs which are to be retrieved
- p: an auxiliary counter for the slabs within the current batch which are candidates for being considered additional targets
- $wayClear$: a boolean variable which is a result of determining whether the relocation to a particular stack is possible with regard to the stacks which are in the way (i. e. if they are not too high to carry the relocation set over them) $index(slab\ s)$: index of the slab s

- $size(set\ S)$: the total height of all the slabs in set S

goodReloPoss: a boolean function which determines if a good relocation of a given slab set is possible, returns $true$ if so and puts together a set of all stacks – $availableStacks$ – to which a good relocation of the given slab set is possible (otherwise, it just returns $false$). A good relocation to a stack t is possible when following conditions are satisfied:

- the minimal index in the stack t is greater than the maximal index of $blocking\ set$, so that the relocation does not block any slabs in the stack t ($stackMinIndex(t)) > maxIndex$), where $stackMinIndex$ obtains the minimal index in stack s. $blockingSet$ is the set of slabs which need to be relocated
- the relocation does not block any slabs from the same lot, even those with a higher index (it could still be a good relocation even if it does, though, but should be avoided if there are other good-relocation options. If there are not, this kind of relocation is the first to be chosen by the function $bestRelocateBad$ anyway, if no better relocations are possible ($blocksSameLot = false)$
- relocation is possible with regard to height restrictions (i. e. there are no stacks in the way from current stack to stack t over which the blocking set cannot be carried because they are too high), in which case the variable $wayClear$ takes on the value $true$ and otherwise $false$.

Input: current state of bay ($stacks$), set to be relocated ($blockingSet), availableStacks$ (initially empty).

Output: $true$ or $false$; $availableStacks$ (set of the stacks available for good relocation of the set which is to be relocated)

Function **goodReloPoss** (*stacks, blockingSet, availableStacks*)

1: *goodReloPoss*=true

2: *availableStacks* ←∅

3: *maxIndex* = 0

4: ∀ slabs *s* ∈ *blockingSet* **if** index(*s*) > *maxIndex* **then** *maxIndex* = index(*s*)

5: ∀ stacks t ∈ *stacks* **if** (*stackMinIndex(t)* > *maxIndex* ∪ *index(bottom(blockingSet))*=*index(top(t))* +1)∩ size(*t*) + size(*blockingSet*) < H_{max} ∩ *wayClear* = true **then** *availableStacks* ← *availableStacks* ∪ {*t*}

6: **if** availableStacks = ∅ **then**

7: goodReloPoss=false

8: **end if**

9: **return** goodReloPoss

---

isRetrievable: a boolean function which determines if a given set of slabs can be retrieved with regard to the vehicle capacity and the heights of the stacks on the way. Input: set of slabs ($currentSet$), current state of bay (*stacks*), number of the stack where the given slabs reside ($currentStackNumber$).

Output: $true$ or $false$

Function **isRetrievable**(*currentSet, stacks, currentStackNumber*)

1: *canPass* = true
2: ∀ stacks *i* | *currentStackNumber* ≤ *i* ≤ size(*stacks*)
3: **if** size(*stacks*(*i*)) + size(*currentSet*) − size(*commonSlabs*) > H_{max} **then** ⇒ *commonSlabs includes slabs which are on top(s) of a stack(s) blocking the way but are to be retrieved anyway, i. e. are part of currentSet*
4: *canPass*=false
5: **end if**
6: **return** *canPass*

---

bestRelocateGood: function which performs the best good relocation (i.e. records it as a move, adds this move to the move sequence and changes the state of the bay accordingly)

Input: set to be relocated ($currentSet$), current state of bay ($stacks$), stack where the current set resides ($currentStack$), set $availableStacks$ which results from the function $goodReloPoss$ and consists of all the stacks to which a good relocation of the given set of slabs ($currentSet$) is possible, $moveSequence$ (all moves performed so far)

Output: $stacks$ (the new state of the bay), $moveSequence$ extended by new move.

Function **bestRelocateGood**(*currentSet, stacks, currentStack, availableStacks, moveSequence*)

1: *minIndex* = 999;
2: ∀ stacks *s* ∈ *availableStacks* **do**
3: **if** *minIndex* > stackMinIndex(*s*) **then**
4: *minIndex* = stackMinIndex(*s*) ⇒*Best good relocation is to a stack with minimal lowest index*
5: *bestStackNumber* = number(*s*) ⇒*the number of stack s in the bay*
6: **end if**
7: **end do**
8: Relocate*currentSet* to stack number *bestStackNumber* and append this move to the move sequence

---

bestRelocateBad: function which performs the best bad relocation (i.e. records it as a move, adds this move to the move sequence and changes the state of the bay accordingly). Obviously, only if no *good relocations* are possible.

Input: set to be relocated ($currentSet$), current state of bay ($stacks$), stack where the current set resides ($currentStack$), set $availableStacks$ (initially empty), $moveSequence$ (all moves performed so far).

Output: $stacks$ (the new state of the bay), $moveSequence$ extended by new move.

Function **bestRelocateBad**(*currentSet, stacks, currentStack, availableStacks, moveSequence)*

1: maxIndex = 0;
2: ∀ stacks $s \in stacks$ **do**
3: **if** size(s) + size($currentSet$) < H_{max} ∩ wayClear = *true* **then**
4:     $availableStacks \leftarrow availableStacks \cup \{s\}$
5: **end if**
6: ∀ stacks $s \in availableStacks$ **do**
7:     **if** $maxIndex$ > stackMinIndex(s) **then**
8:         $maxIndex$ = stackMinIndex(s)   ⇒ *Best bad relocation is to a stack with maximal lowest index*
9:         $bestStackNumber$ = number(s)   ⇒*the number of stack s in the bay*
10:     **end if**
11: **end do**
12: Relocate *currentSet* to stack number *bestStackNumber* and append this move to the move sequence

---

unblockTarget1: a function which defines the most efficient move sequence in order to free a given slab set (i. e. by relocating all slabs which are on top of it).

Input: position of the last slab of the given slab set in its stack ($targetSlabPosition$), the stack where the given slab set resides ($targetStack$), current state of bay ($stacks$), all moves performed so far ($moveSequence$), capacity of the vehicle($vehicleCapacity$).

Output: $stacks$ (the new state of the bay), $moveSequence$ extended by new moves.

function **unblockTarget1**(targetSlabPosition, targetStack, stacks, moveSequence, *vehicleCapacity*)

```
1: blockingSet ← ∅
2: i ← number(top(targetStack))
3:while i ≠ targetSlabPosition do   ⇒Add all slabs that block the target slab to the blockingSet
4:          blockingSet ← blockingSet ∪ {targetStack[i]}
5:          i ← i - 1
6: end while
7: availableStacks ← ∅                  ⇒Initialize the list of stacks available for good relocations
8: reloSet ← ∅
9: if goodReloPoss(reloSet,stacks,availableStacks) then
10:         while goodReloPoss(reloSet,stacks,availableStacks) ∩ blockingSet ≠ ∅ do
11:                 reloSet1 ← reloSet
12:                 reloSet1 ← reloSet1 ∪ {top(blockingSet)}
13:                 if goodReloPoss(reloSet1,stacks,availableStacks) ∩ size(reloSet1) <= vehicleCapacity then
14:                         blockingSet ← blockingSet\{top(blockingSet)}
15:                         reloSet ← reloSet1   ⇒ this check should be carried on until it is not possible to add another
    item to the relocation set without making a good relocation of this set impossible, or until the relocation set equals the blocking set
16:                 end if
17:         end while
18:         bestRelocateGood(reloSet,availableStacks,stacks,targetStack,moveSequence)
19:  else            ⇒ If no good relocation of the first slab of blocking set (which is in reloSet) is possible
20:         if blockingSet ≠ ∅ then          ⇒ If there initially was more than one slab in the blocking set and the first
    one can definitely be only bad-relocated, it is to decide what to do with the next ones.
21:                 nextSlab← {top(blockingSet)}      ⇒ Add the second top slab of the blocking set to reloSet
22:                 while goodReloPoss (nextSlab,stacks,availableStacks)= false ∩ size(re-
    loSet∪nextSlab) <= vehicleCapacity do ⇒ If it can be well-relocated, bad-relocate the top one and, beginning
    from the next one, perform the same check as at the beginning of the function
23:                         reloSet ← reloSet ∪ nextSlab     ⇒Relocate as many slabs from the blocking set at a time as
    possible, if they cannot be well-relocated anyway
24:                         blockingSet ← blockingSet\{top(blockingSet)}
25:                 end while
26:         end if
27:         bestRelocateBad(reloSet,availableStacks,stacks,targetStack,moveSequence)
28: end if
```

unblockTarget2: a boolean function which defines the most efficient move sequence in order to free a given slab set by performing only *good relocations*. If no such relocations possible, it returns $false$ (otherwise $true$).

Input: position of the last slab of the given slab set in its stack ($targetSlabPosition$), the stack where the given slab set resides ($targetStack$), current state of bay ($stacks$), all moves performed so far ($moveSequence$), capacity of the vehicle ($vehicleCapacity$).

Output: $true$ or $false$, $stacks$ (the new state of the bay), extended $moveSequence$ (only if a relocation was performed).

function **unblockTarget2** (*targetSlabPosition, targetStack, stacks, moveSequence, vehicleCapacity*)

1: *unblock* = *true*
2: *blockingSet* ←∅
3: *i* ← *number(top(targetStack))*
4: **while** *i* ≠ *targetSlabPosition* **do**
5: *blockingSet* ← *blockingSet* ∪ {*targetStack[i]*}
6: *i* ← *i* - 1
7: **end while**
8: *availableStacks* ←∅ ⇒ *Initialize the list of stacks available for good relocations*
9: *reloSet* ←∅ ⇒ *Initialize the first set of slabs to be relocated*
10: **if** *goodReloPoss*(*reloSet,stacks,availableStacks*) **then**
11: **while** *goodReloPoss*(*reloSet,stacks,availableStacks*) ∩ *blockingSet* ≠∅ **do**
12: *reloSet*1 ← *reloSet*
13: *reloSet*1 ← *reloSet*1 ∪{*top*(*blockingSet*)}
14: **if** *goodReloPoss*(*reloSet*1,*stacks,availableStacks*) ∩ *size*(*reloSet*1) ≤ *vehicleCapacity* **then**
15: *blockingSet* ← *blockingSet*\{*top*(*blockingSet*)}
16: *reloSet* = *reloSet*1 ⇒ *this check should be carried on until it is not possible to add another item to the relocation set without making a good relocation of this set impossible, or until the relocation set equals the blocking set*
17: **else**
18: *unblock* = *false*
19: **end if**
20: **end while**
21: **if** *unblock* = *true* **then**
22: *bestRelocateGood*(*reloSet, availableStacks,stacks,targetStack,moveSequence*)
23: **end if**
24: **else**
25: *unblock* = *false*
26: **end if**
27: **return** *unblock*

---

reduceHeight1: a function which defines the most efficient move sequence in order to reduce the height of a stack to a required degree by relocating the corresponding number of slabs from its top. Works exactly in the same fashion as **unblockTarget1** but requires an additional input parameter, namely, the required height of the stack. This parameter is determined before calling this function as soon as the check in 4.1. deems a stack too high to retrieve the initial target set I. The required height would be the maximal stack height over which the vehicle carrying the initial target set I can pass.

Input: the stack whose height needs to be reduced ($targetStack$), current state of bay ($stacks$), all moves performed so far ($moveSequence$), $vehicleCapacity$, the required height of the stack ($requiredHeight$)

Output: $stacks$ (the new state of the bay), $moveSequence$ extended by new moves.

function **reduceHeight1**(*targetStack, stacks, moveSequence, vehicleCapacity, requiredHeight*)

```
1: blockingSet ← ∅          ⇒Initialize the set of slabs which have to be relocated to reduce the height of targetStack (=current blocking stack)
2: i ← number(top(targetStack))
3: while size(targetStack) – size(blockingSet) > requiredHeight do
4:          blockingSet ← blockingSet ∪ {targetStack[i]}
5:          i ← i - 1
6: end while
7: availableStacks ← ∅          ⇒Initialize the list of stacks available for good relocations
8: reloSet ← ∅
9: if goodReloPoss(reloSet,stacks,availableStacks) then
10:         while goodReloPoss(reloSet,stacks,availableStacks) ∩ blockingSet ≠ ∅ do
11:                 reloSet1 ← reloSet
12:                 reloSet1 ← reloSet1 ∪ {top(blockingSet)}
13:                 if goodReloPoss(reloSet1,stacks,availableStacks) ∩ size(reloSet1) <= vehicleCapacity then
14:                         blockingSet ← blockingSet\{top(blockingSet)}
15:                         reloSet ← reloSet1     ⇒ this check should be carried on until it is not possible to add another
      item to the relocation set without making a good relocation of this set impossible, or until the relocation set equals the blocking set
16:                 end if
17:         end while
18:         bestRelocateGood(reloSet,availableStacks,stacks,targetStack,moveSequence)
19: else          ⇒ If no good relocation of the first slab of blocking set (which is in reloSet) is possible
20:         if blockingSet ≠ ∅ then          ⇒ If there initially was more than one slab in the blocking set and the first
      one can definitely be only bad-relocated, it is to decide what to do with the next ones.
21:                 nextSlab← {top(blockingSet)}          ⇒ Add the second top slab of the blocking set to reloSet
22:                 while goodReloPoss (nextSlab,stacks,availableStacks)= false ∩ size(re-
      loSet∪nextSlab) <= vehicleCapacity do          ⇒ If it can be well-relocated, bad-relocate the top one and,
      beginning from the next one, perform the same check as at the beginning of the function
23:                         reloSet ← reloSet ∪ nextSlab          ⇒Relocate as many slabs from the blocking set at a time as
      possible, if they cannot be well-relocated anyway
24:                         blockingSet ← blockingSet\{top(blockingSet)}
25:                 end while
26:         end if
27:         bestRelocateBad(reloSet,availableStacks,stacks,targetStack,moveSequence)
28: end if
```

reduceHeight2: a boolean function which defines the most efficient move sequence in order to reduce the height of a stack to a required degree by relocating the corresponding number of slabs from its top by performing only *good relocations*. If no such relocations are possible, it returns $false$ (otherwise $true$). Works exactly in the same fashion as **unblockTarget1** but requires an additional input parameter, namely, the required height of the stack. This parameter is determined before this function is called, as soon as the check in step 4.2. deems a stack too high to retrieve **current_targets.** The required height would be the maximal stack height over which the vehicle carrying the initial target set I can pass.

Input: position of the last slab of the given slab set in its stack ($targetSlabPosition$), the stack where the given slab set resides ($targetStack$), *stacks, moveSequence, vehicleCapacity*

Output: $true$ or $false$, $stacks$ (the new state of the bay), extended $moveSequence$ (only if a relocation was performed)

---

function **reduceHeight2** (*targetStack, stacks, moveSequence, vehicleCapacity, requiredHeight*)

1: *unblock* ← *true*
2: *blockingSet* ←∅
3: *i* ← *number(top(targetStack))*
4: **while** *size(targetStack) – size(blockingSet) > requiredHeight* **do**
5: *blockingSet* ← *blockingSet* ∪ *targetStack*[*i*]
6: *i* ← *i* - 1
7: **end while**
8: *availableStacks* ←∅ ⇒ *Initialize the list of stacks available for good relocations*
9: *reloSet* ←∅ ⇒ *Initialize the first set of slabs to be relocated*
10: **if** *goodReloPoss*(*reloSet,stacks,availableStacks)* **then**
11: **while** *goodReloPoss*(*reloSet,stacks,availableStacks*) ∩ *blockingSet* ≠∅ **do**
12: *reloSet*1 ← *reloSet*
13: *reloSet*1 ← *reloSet*1 ∪{*top*(*blockingSet*)}
14: **if** *goodReloPoss*(*reloSet*1,*stacks,availableStacks*) ∩ *size*(*reloSet*1) ≤ *vehicleCapacity* **then**
15: *blockingSet* ← *blockingSet*\{*top*(*blockingSet*)}
16: *reloSet* = *reloSet*1 ⇒ *this check should be carried on until it is not possible to add another item to the relocation set without making a good relocation of this set impossible, or until the relocation set equals the blocking set*
17: **else**
18: *unblock* ← *false*
19: **end if**
20: **end while**
21: **if** *unblock* = *true* **then**
22: *bestRelocateGood*(*reloSet, availableStacks,stacks,targetStack,moveSequence*)
23: **end if**
24: **else**
25: *unblock* = *false*
26: **end if**
27: **return** *unblock*

---

5. Results

CMTA has been tested on existing instances from the literature, as well as real-world instances based on data provided by *Voestalpine*.

5.1. Testing on benchmark instances

CMTA has been tested on the benchmark Caserta (11) instances. The instances are defined by their dimensions as in H x S, where H indicating the height of all stacks, which is uniform and measured in the number of slabs in a stack, and where S is the number of stacks. For example, a 4x5 instance would consist of 5 stacks with 4 slabs in each of them. A total of 840 instances are provided. These constitute 40 different instance categories of different dimensions ranging from 3x3 to 10x10. These instances are designed for the standard BRP, so they consist of homogenous slabs, all of which have to be retrieved from the bay in the order corresponding to their indices. The input data for each instance includes therefore only one file depicting the initial state of the bay, without any restriction on the maximum stack height. The height of stacks is defined only by the number of slabs, as all slabs are assumed to have identical heights. We solve these instances using different input parameters $H_{max}$and cap, the former ranging from $H + 1$ to unlimited, the latter from 1 to 7. Note that there is only one kind of vehicle capacity cap in this case, measured in the number of slabs.
Table 1 compares the performance of CMTA to the state-of-the-art methods from literature. Note that the bold values indicate the best result among all methods. CMTA delivers results which are inferior to the advanced methods in this table (the average, best and worst gap between CMTA and the best result is 12.52%, 2.11% and 35.41%, respectively). Table 2 provides the same insight where merely the parameter H_{max} is changed. Again, CMTA is outperformed by the advanced methods (the average, best and worst gap between CMTA and the best result is 14.16%, 2.11% and 35.41%, respectively).
Testing CMTA with cap = 1 has been merely for the purpose of comparison of its performance to that of other BRP methods from the literature. As described in chapter 2, almost all these methods are designed for standard BRP without batch moves. As in

standard BRP the number of retrievals is fixed and equal to the number of slabs to be retrieved, the objective value always corresponds to the number of relocations.

The results of this comparison can be seen in the tables below (the values have been copied from the paper by Tricoire et al. (21) and extended by the CMTA results). The objective values and CPU time (seconds) shown in the table, correspond to the average values computed over all instances of each category. For the Caserta instances in Table 3, all compared methods from the literature terminate within one second, and CPU times of CMTA have already been reported in Table 1, so we do not report CPU values in this table. Note that Tricoire et al. (21) implemented the greedy look-ahead heuristic by Jin et al. in full accordance with the algorithm description in Jin et al. (26) to compare their methods to GLAH, as the code for the latter was not available online. The differences between the values obtained by Tricoire et al. (21) from this implementation slightly differ from the ones presented in Jin et al. (26) due to the possibility of arbitrary choices at a particular stage of GLAH (if there are several possible relocations with the same look-ahead value, any of them can be chosen).

Table 1. Comparison to advanced methods from Tricoire et al. (21) for Caserta instances, H_{max} unlimited

		Pilot method		Rake search		GLAH, D=3 run by the authors		GLAH, D=4 run by the authors		GLAH, objective values from Jin et al.		CMTA		%-difference betw. CMTA and the best result
H	***S***	Obj.	CPU	Obj.	CPU	Obj.	CPU	Obj.	CPU	D=3	D=4	Obj.	CPU	
3	3	**4.950**	0.02	**4.950**	0.00	**4.950**	0.00	**4.950**	0.00	**4.950**	**4.950**	5.070	0.17	2.42
3	4	**6.025**	0.06	**6.025**	0.00	**6.025**	0.00	**6.025**	0.00	6.030	6.030	6.300	0.17	4.56
3	5	**6.850**	0.08	**6.850**	0.00	**6.850**	0.00	**6.850**	0.00	**6.850**	**6.850**	7.075	0.16	3.28
3	6	**8.275**	0.43	**8.275**	0.01	**8.275**	0.00	**8.275**	0.00	8.280	8.280	8.450	0.16	2.11
3	7	**9.100**	0.71	**9.100**	0.02	9.125	0.00	9.125	0.00	**9.100**	**9.100**	9.385	0.16	3.13
3	8	**10.300**	0.15	**10.300**	0.07	10.400	0.00	10.400	0.01	**10.300**	**10.300**	10.700	0.25	3.88
4	4	**9.675**	0.10	**9.675**	0.00	9.700	0.00	9.725	0.00	9.700	9.700	10.925	0.16	12.92
4	5	**12.250**	0.11	12.275	0.01	12.275	0.00	12.275	0.01	**12.250**	**12.250**	13.575	0.29	10.82
4	6	**13.325**	0.42	13.350	0.05	13.375	0.00	13.350	0.02	13.330	13.300	14.475	0.33	8.63
4	7	15.400	0.79	**15.375**	0.10	15.475	0.01	15.400	0.02	15.400	15.380	16.700	0.35	8.62
5	4	14.550	0.15	14.650	0.01	14.550	0.01	**14.475**	0.05	14.480	14.480	16.450	0.16	13.64
5	5	**17.500**	0.15	17.625	0.05	17.625	0.02	17.600	0.11	17.580	17.530	20.375	0.4	16.43
5	6	20.950	0.52	20.925	0.13	21.000	0.04	20.950	0.24	20.930	**20.900**	23.650	0.38	13.02
5	7	22.575	1.00	22.650	0.23	22.800	0.05	22.675	0.29	22.530	**22.500**	25.675	0.35	13.96
5	8	**25.775**	0.54	25.800	0.44	25.975	0.07	25.975	0.48	25.850	25.850	28.795	0.53	11.72
5	9	28.550	0.41	**28.475**	0.69	28.675	0.08	28.650	0.55	28.550	28.500	31.950	0.72	12.20
5	10	31.075	0.19	**31.000**	1.11	31.350	0.13	31.325	0.88	31.130	31.130	34.400	0.74	10.97

6	6	28.400	0.59	28.450	0.29	28.400	0.11	28.150	0.79	28.130	**28.000**	34.150	0.51	21.40
6	10	42.025	0.26	41.725	2.85	42.150	0.36	42.025	2.94	41.700	**41.630**	48.800	1.02	17.03
10	6	70.175	1.09	68.575	3.14	68.750	1.44	69.150	11.00	66.030	**64.500**	90.300	1.20	36.76
10	10	99.875	0.64	96.625	42.74	97.000	4.29	97.100	48.55	93.700	**93.350**	126.875	2.81	35.41
										Average gap in % between CMTA and best result				*12.52*

Table 2. Comparison to advanced methods from Tricoire et al. (21) for Caserta instances, $H_{max} = H + 2$

H	***S***	Pilot method		Rake search		GLAH, D=3 run by the authors		GLAH, D=4 run by the authors		GLAH, results from Jin et al.		CMTA		%-difference betw. CMTA and the best result
		Obj.	CPU	Obj.	CPU	Obj.	CPU	Obj.	CPU	D=3	D=4	Obj.	CPU	
3	3	**4.975**	0.02	**4.975**	0.00	**4.975**	0.00	**4.975**	0.00	4.980	4.980	5.080	0.19	2.11
3	4	**6.025**	0.05	**6.025**	0.00	**6.025**	0.00	**6.025**	0.00	6.030	6.030	6.300	0.64	4.56
3	5	**6.850**	0.07	**6.850**	0.00	**6.850**	0.00	**6.850**	0.00	**6.850**	**6.850**	7.075	0.79	3.28
3	6	**8.275**	0.39	**8.275**	0.01	**8.275**	0.00	8.275	0.00	8.280	8.280	8.450	0.36	2.11
3	7	9.125	0.67	9.125	0.02	9.125	0.00	9.125	0.00	**9.100**	**9.100**	9.410	0.2	3.41
3	8	10.300	0.15	10.325	0.06	10.400	0.00	10.400	0.01	**10.300**	**10.300**	10.725	0.36	4.13
4	4	**9.750**	0.07	9.775	0.00	9.775	0.00	9.800	0.00	9.800	9.780	11.000	0.17	12.82
4	5	12.325	0.11	12.350	0.01	12.300	0.00	12.275	0.01	**12.280**	**12.280**	13.700	0.26	11.61

4	6	13.350	0.39	13.375	0.04	13.350	0.00	13.325	0.02	13.330	**13.300**	14.750	0.54	10.69
4	7	15.400	0.74	15.400	0.10	15.500	0.00	15.400	0.02	15.430	**15.380**	17.025	0.29	10.55
5	4	14.925	0.10	14.975	0.01	14.925	0.01	14.875	0.05	14.900	**14.800**	16.800	0.25	12.94
5	5	**17.700**	0.14	17.825	0.05	17.875	0.02	17.725	0.10	17.780	**17.700**	21.275	0.41	20.20
5	6	21.125	0.49	21.150	0.12	21.175	0.04	21.100	0.24	**21.000**	20.980	24.275	0.40	15.60
5	7	22.775	0.92	22.900	0.22	23.000	0.05	22.825	0.30	22.730	22.650	**26.475**	0.51	16.48
5	8	26.050	0.51	**26.000**	0.44	26.250	0.07	26.175	0.47	26.100	26.030	29.795	0.95	14.60
5	9	28.750	0.41	**28.575**	0.70	28.950	0.09	28.825	0.58	28.850	28.700	32.770	0.79	14.68
5	10	31.275	0.20	**31.175**	1.16	31.500	0.12	31.425	0.85	31.300	31.250	35.750	1.17	14.68
6	6	28.850	0.58	28.925	0.26	29.275	0.12	28.875	0.82	28.980	**28.450**	35.750	0.77	23.92
6	10	42.950	0.28	42.550	2.94	43.000	0.39	42.800	3.17	42.450	**42.350**	50.075	1.33	17.96
10	6	75.200	1.14	73.700	2.74	75.700	1.39	75.050	10.20	71.730	**70.780**	101.800	1.67	41.92
10	10	105.625	0.72	102.950	36.09	105.975	5.31	105.625	51.36	100.830	**100.700**	140.300	3.57	39.15
										Average gap in % between CMTA and best result				*14.16*

Table 3. Comparison to fast methods from Tricoire et al. (21), H_{max} unlimited

H	***S***	LA-S-1	SM-1	SM-2	SmSEQ-1	SmSEQ-2	JZW	CMTA	%-difference in obj. value betw. CMTA and the best result
		Obj.	Obj.	Obj.	Obj.	Obj.	Obj.	Obj.	
3	3	5.025	5.025	5.025	5.100	5.125	**5.000**	5.070	1.40
3	4	6.225	6.225	6.175	6.200	**6.150**	**6.150**	6.300	2.44
3	5	6.975	6.900	6.950	**6.900**	**6.900**	6.925	7.075	2.54
3	6	8.450	8.350	**8.300**	8.325	**8.300**	8.350	8.450	1.81
3	7	9.225	9.200	9.225	**9.150**	9.200	**9.150**	9.385	2.57
3	8	10.700	10.600	10.575	10.550	**10.525**	**10.525**	10.700	1.66
4	4	10.350	10.125	10.150	10.425	10.375	**10.025**	10.925	8.98
4	5	12.925	12.800	12.675	12.825	12.850	**12.650**	13.575	7.31
4	6	13.925	13.900	13.750	13.825	13.825	**13.650**	14.475	6.04
4	7	16.225	16.150	15.925	15.900	15.950	**15.800**	16.700	5.70
5	4	15.850	15.875	15.65	15.925	15.875	**15.375**	16.450	6.99
5	5	19.625	19.375	19.225	19.500	19.575	**19.175**	20.375	6.26
5	6	22.550	22.125	**21.85**	22.025	22.000	22.000	23.65	8.24
5	7	24.725	24.075	**23.425**	23.975	23.650	23.825	25.675	9.61
5	8	27.675	27.400	**26.975**	27.000	27.000	27.250	28.795	6.75
5	9	30.425	30.025	**29.600**	30.000	29.900	29.675	31.950	7.94
5	10	33.150	32.525	32.025	32.350	**32.075**	32.300	34.400	7.42
6	6	32.500	31.775	31.075	31.375	31.300	**30.975**	34.150	10.25
6	10	46.475	45.175	43.950	45.500	44.700	**44.225**	48.800	11.04
10	6	84.375	84.775	81.475	83.100	82.450	**79.350**	90.300	13.80
10	10	118.975	113.25	108.600	113.65	111.975	**108.125**	126.875	17.34
							Average gap in % between CMTA and best result		***6.96***

As it can be seen in the Table 1-Table 3, CMTA does not show particularly good results compared to the acknowledged methods from the literature for standard BRP, which is by all means expectable, as this algorithm has been designed specifically for the BRP

with batch moves, and even more so for the specific kind of steel slab stacking problem dealt with in the real-world setting of *Voestalpine* slab yards.

In particular, it should be noted that the results of every algorithm tend to improve with increasing H_{max} (the smaller the instance, the less noticeable the improvement). Table 4 compares Table 1 and 2 by showing the relative improvements of the objective values between H_{max} unlimited and $H_{max} = H + 2$ for the case with $cap = 1$ for different advanced methods.

Table 4. Relative improvement while changing H_{max} from unlimited to $H + 2$

Gap between results with $\boldsymbol{H_{max}}$ unlimited and $\boldsymbol{H_{max} = H + 2}$		Pilot method	Rake search	GLAH, D=3	GLAH, D=4	GLAH, Jin et al.	CMTA
H	S	Obj.	Obj.	Obj.	Obj.	Obj.	Obj.
3	3	0.005	0.005	0.005	0.005	0.006	0.002
3	4	0.000	0.000	0.000	0.000	0.000	0.000
3	5	0.000	0.000	0.000	0.000	0.000	0.000
3	6	0.000	0.000	0.000	0.000	0.000	0.000
3	7	0.003	0.003	0.000	0.000	0.000	0.003
3	8	0.000	0.002	0.000	0.000	0.000	0.002
4	4	0.008	0.010	0.008	0.008	0.010	0.007
4	5	0.006	0.006	0.002	0.000	0.002	0.009
4	6	0.002	0.002	-0.002	-0.002	0.000	0.019
4	7	0.000	0.002	0.002	0.000	0.002	0.019
5	4	0.026	0.022	0.026	0.028	0.029	0.021
5	5	0.011	0.011	0.014	0.007	0.011	0.044
5	6	0.008	0.011	0.008	0.007	0.003	0.026
5	7	0.009	0.011	0.009	0.007	0.009	0.031
5	8	0.011	0.008	0.011	0.008	0.010	0.035
5	9	0.007	0.004	0.010	0.006	0.011	0.026
5	10	0.006	0.006	0.005	0.003	0.005	0.039
6	6	0.016	0.017	0.031	0.026	0.030	0.047
6	10	0.022	0.020	0.020	0.018	0.018	0.026
10	6	0.072	0.075	0.101	0.085	0.086	0.127
10	10	0.058	0.065	0.093	0.088	0.076	0.106

In Table 5, we compare our results to the ICC-TREE method provided by Kopfer, Zhang and Liu (2015). The authors compare their method to the Corridor method and LA-N.

Hence, we take all three methods into account. Again, best values are highlighted in bold font.

Table 5. Comparison to results from Kopfer, Zhang & Liu (10) for Caserta instances

S	H	H_{max}	Corridor Method Obj.	LA-N (best result of parameter N) Obj.	ICC-TREE Obj.	ICC-TREE CPU	CMTA Obj.	CMTA CPU	%-difference in obj. value between CMTA and the best result
3	3	5	5.4	5.1	5.2	<1	5.1	<1	-0.39
4	3	5	6.5	**6.3**	6.5	<1	6.3	<1	0.00
5	3	5	7.3	**7.0**	7.1	<1	7.1	<1	1.07
6	3	5	**7.9**	8.4	8.6	<1	8.5	<1	6.96
7	3	5	**8.6**	9.2	9.5	<1	9.4	<1	9.42
8	3	5	**10.5**	10.6	10.7	<1	10.7	<1	2.14
4	4	7	9.9	**10.4**	10.4	<1	10.9	<1	10.35
5	4	7	16.5	13.0	**12.9**	<1	13.7	<1	6.01
6	4	7	19.8	**14.0**	**14.0**	<1	14.5	<1	3.75
7	4	7	21.5	16.4	**16.3**	2	16.8	<1	3.07
4	5	9	16.6	15.8	**15.7**	1	16.5	1.177	5.10
5	5	9	**18.8**	19.7	19.2	3	20.5	<1	8.78
6	5	9	**22.1**	22.6	22.5	10	22.9	<1	3.73
7	5	9	25.8	24.8	**24.4**	22	25.7	<1	5.33
8	5	9	30.1	**27.8**	27.9	36	28.9	<1	3.86
9	5	9	33.1	**30.7**	31.7	52	32.1	<1	4.65
10	5	9	36.4	**33.5**	34.8	54	34.8	<1	3.88
6	6	11	**32.4**	32.6	35.7	52	34.3	<1	5.79
10	6	11	49.5	**46.8**	52.9	60	49.1	1.13	4.86
6	10	19	102.0	**85.0**	105.7	60	91.0	1.41	7.00
10	10	19	128.3	**119.5**	154.5	60	127.5	1.96	6.67
Average gap in % between CMTA and best result									***4.86***

This table demonstrates quite an acceptable performance of CMTA, compared to other methods, in very small amounts of time. Especially for the largest instances, CMTA demonstrates the second best result each time.

Unfortunately, this was the only comparison that could be drawn to the results achieved by Zhang, Liu and Kopfer (10), as their paper does not contain detailed results for the objective values achieved for BRP with actual batch moves (i. e., lifting capacity greater than 1) while testing their methods on Caserta (11) instances. For their own instances, the objective values for lifting capacity $cap = 3$, where cap specifies the number of items, have been provided, but unfortunately, we were not able to receive those instances from the authors and therefore we cannot compare our CMTA method to their ICC-Tree method, which is a major lapse in the comprehensive validation of the algorithm. Yet, some conclusions can certainly be drawn from the extent of improvement of the objective function for the same Caserta instances when the lifting capacity is increased.

Table 6 shows that the most significant improvements have been achieved by increasing the lifting capacity up to 4, whereas further lifting capacity increases from this point on have not improved the objective value dramatically. As also calculated in Table 6, relative improvement from increasing the lifting capacity from 1 to 4 is drastically bigger than from doing so for the capacity 4 to 7. Besides, lifting capacity of more items than 4 would be less common in many real-world problem settings where straddle carriers are used. Based on these observations, we compared the objective values for lifting capacity = 4 and different values of H_{max}. The results, as shown in Table 7, show the improvement of the objective values as the maximal allowed stack height increases. As we can see, the improvements from increasing H_{max} are considerable up to the $H_{max} = H + 4$ (i.e. the maximal height of a stack can be at most 4 tiers larger than the initial height of all the stacks in the bay), whereas the objective value does not improve significantly along with further increase of H_{max}. Note that H_{max} has to be at least $H + 1$ for the problem to be feasible. Given certain parameters, if no feasible solution exists, then no result is provided.

Table 6. Results for different vehicle capacities, CMTA with H_{max} unlimited

H	S	$cap = 1$	$cap = 2$	$cap = 3$	$cap = 4$	$cap = 5$	$cap = 6$	$cap = 7$	%-difference betw. $cap = 1$ and $cap = 4$	%-difference betw. $cap = 4$ and $cap = 7$
3	3	5.070	2.949	1.923	1.461	1.590	1.590	1.590	-71.18	8.83
3	4	6.300	4.250	3.500	2.975	2.575	2.525	2.525	-52.78	-15.13
3	5	7.075	4.975	4.000	3.925	3.350	3.250	3.250	-44.52	-17.20
3	6	8.450	6.589	4.950	4.425	4.250	3.974	3.850	-47.63	-12.99
3	7	9.385	8.725	6.585	6.350	5.700	5.500	5.500	-32.34	-13.39
3	8	10.700	8.300	6.900	5.900	5.925	5.600	5.625	-44.86	-4.66
4	4	10.925	7.575	6.350	5.350	4.872	4.950	4.872	-51.03	-8.93
4	5	13.575	9.725	7.575	7.350	6.700	6.500	6.500	-45.86	-11.56
4	6	14.475	10.536	8.825	8.650	8.175	7.650	7.850	-40.24	-9.25
4	7	16.700	12.744	10.821	9.718	9.487	9.282	8.795	-41.81	-9.50
5	4	16.450	11.650	10.350	9.125	8.825	8.750	8.775	-44.53	-3.84
5	5	20.375	15.375	12.550	11.900	11.050	10.675	10.500	-41.60	-11.76
5	6	23.650	18.150	15.175	13.575	12.975	12.800	12.575	-42.60	-7.37
5	7	25.675	20.525	16.950	15.800	15.200	14.900	14.750	-38.46	-6.65
5	8	28.795	23.282	20.128	18.333	17.923	17.718	17.410	-36.33	-5.03
5	9	31.950	24.487	21.330	19.949	19.128	18.821	18.539	-37.56	-7.07
5	10	34.400	27.750	24.150	22.800	22.250	21.475	21.425	-33.72	-6.03
6	6	34.150	25.575	22.375	20.700	19.275	18.850	18.550	-39.39	-10.39
6	10	48.800	37.950	33.150	30.800	30.100	29.175	28.625	-36.89	-7.06
10	6	90.300	56.675	52.275	50.325	49.950	48.350	47.525	-44.27	-5.56
10	10	126.875	84.050	78.425	75.975	76.000	73.550	71.800	-40.12	-5.50

Table 7. CMTA Results for Caserta instances with cap = 4 and varying H_{max}

H	S	$H_{max} = 4$	$H_{max} = 5$	$H_{max} = 6$	$H_{max} = 7$	$H_{max} = 8$	$H_{max} = 9$	$H_{max} = 10$	$H_{max} = 11$	$H_{max} = 12$	$H_{max} = 13$	$H_{max} = 14$	$H_{max} = \infty$
3	3	1.949	1.718	1.718	1.461	1.461	1.461	1.461	1.461	1.461	1.461	1.461	1.461
3	4	3.450	3.150	2.975	2.975	2.975	2.975	2.975	2.975	2.975	2.975	2.975	2.975
3	5	4.975	4.400	3.975	3.975	3.925	3.925	3.925	3.925	3.925	3.925	3.925	3.925
3	6	6.416	5.600	4.575	4.875	4.475	4.425	4.425	4.425	4.425	4.425	4.425	4.425
3	7	7.615	6.786	6.077	5.872	5.641	5.615	5.615	5.615	5.615	5.615	5.615	5.615
3	8	9.158	7.444	6.500	5.947	5.790	5.737	5.737	5.737	5.737	5.737	5.737	5.737
4	4	-	6.900	6.050	5.525	5.375	5.350	5.350	5.350	5.350	5.350	5.350	5.350
4	5	-	9.950	8.675	7.625	7.425	7.275	7.350	7.350	7.350	7.350	7.350	7.350
4	6	-	11.300	9.825	9.325	8.825	8.775	8.650	8.650	8.650	8.650	8.650	8.650
4	7	-	14.575	11.900	10.500	10.05	9.925	9.850	9.825	9.800	9.775	9.750	9.718
5	4	-	-	12.820	10.400	9.350	9.250	9.275	9.150	9.125	9.125	9.125	9.125
5	5	-	-	16.250	13.800	12.625	12.650	11.750	11.900	11.900	11.900	11.900	11.900
5	6	-	-	19.750	17.275	15.450	14.550	13.875	13.725	43.650	13.575	13.575	13.575
5	7	-	-	22.100	19.400	17.075	16.750	16.550	16.175	16.050	15.875	15.860	15.800
5	8	-	-	26.539	22.718	20.667	19.718	18.615	18.744	18.410	18.385	18.41	18.333
5	9	-	-	29.564	24.692	22.436	21.000	20.744	20.282	20.154	20.154	19.949	19.949
5	10	-	-	32.425	27.975	25.635	23.675	22.975	23.250	22.900	22.950	22.875	22.800
6	6	-	-	-	28.550	26.100	23.400	22.475	21.225	20.950	20.700	20.775	20.700
6	10	-	-	-	43.525	37.775	34.700	33.125	32.125	31.725	31.475	31.200	30.800
10	6	-	-	-	-	-	-	-	88.700	78.900	72.300	68.675	50.325
10	10	-	-	-	-	-	-	-	128.600	115.150	107.825	105.150	75.975

5.2. Testing on real-world instances

For real-world instances, the input consists of 3 files: the first one contains the initial state of the bay, the second one contains slab data (dimensions and weight of each slab), and the third one contains the retrieval order (which slabs have to be retrieved and in which order).

The CMTA is tested on all available data, which contains a total of 224 bay instances with different numbers of stacks and slabs, as well as different numbers of slabs to be retrieved, ranging from 1 to 12. The detailed results for all those instances (excluding those with only 1 or 2 slabs to retrieve, because they are not of much interest for our research) are presented in the Table 8. Hence, only 66 instances are reported in this thesis. The columns "№ of slabs to be retrieved" and "№ of stacks" are trivial. "№ of slabs" specifies the total number of slabs contained in the initial bay, whereas the next three columns represent the number of moves (relocations, retrievals and total) which were made to retrieve the required slabs. We categorize the instances according to "№ of slabs to be retrieved". The average results over every category are presented in Table 9

Table 8. Results on real-world instances

Bay title	№ of slabs to be retrieved	№ of stacks	№ of slabs	№ of relocations	№ of retrievals	Total number of moves
0_17	3	9	26	3	2	5
0_27	3	9	31	2	3	5
0_35	3	7	5	1	3	4
0_44	3	6	22	3	1	4
1_7	3	8	18	3	2	5
1_10	3	8	15	5	1	6
1_29	3	7	20	1	3	4
2_41	3	7	4	2	2	4
3_9	3	8	63	17	3	20
3_17	3	9	28	4	1	5
3_19	3	9	14	0	3	3
3_20	3	9	15	2	2	4
3_24	3	7	21	4	2	6
3_28	3	8	31	2	3	5
3_48	3	5	11	1	1	2
4_6	3	8	18	2	2	4
4_22	3	9	50	5	3	8
4_24	3	9	37	5	3	8
4_34	3	5	3	0	3	3

4_36	3	7	39	2	3	5
4_42	3	7	6	0	3	3
0_6	4	8	53	9	4	13
0_7	4	8	66	11	4	15
0_13	4	8	50	3	3	6
0_16	4	8	36	2	3	5
1_2	4	7	21	3	2	5
1_26	4	7	10	0	4	4
1_33	4	6	18	5	4	9
1_35	4	6	15	4	2	6
2_3	4	7	12	1	4	5
2_19	4	8	20	2	2	4
2_20	4	8	17	1	2	3
2_40	4	7	9	0	1	1
2_44	4	6	13	0	4	4
3_3	4	7	6	1	4	5
3_13	4	8	51	11	3	14
3_15	4	8	23	3	2	5
3_16	4	9	18	3	3	6
3_25	4	6	22	3	3	6
3_27	4	9	31	5	2	7
4_1	4	7	26	2	4	6
4_5	4	7	21	3	3	6
4_19	4	9	24	1	3	4
4_41	4	4	4	1	4	5
0_4	5	8	19	5	4	9
0_14	5	8	56	8	5	13
0_21	5	9	42	1	4	5
3_29	5	9	34	6	4	10
4_11	5	8	47	4	3	7
4_12	5	7	38	14	5	19
0_9	6	7	56	19	6	25
0_34	6	4	6	0	2	2
1_34	6	4	7	7	3	10
2_36	6	2	6	0	6	6
2_37	6	5	15	0	6	6
2_39	6	7	33	4	5	9
4_40	6	4	17	0	6	6
0_32	7	4	7	2	4	6
1_1	7	7	11	3	6	9
2_45	7	5	16	9	5	14
3_10	7	8	56	13	6	19
3_41	7	6	31	5	9	14
2_38	8	6	37	6	5	11
3_8	8	8	62	33	9	42
4_45	10	6	26	8	7	15
3_42	12	4	28	19	4	23

Table 9. Average results of testing on real-world instances

№ of instances	№ of slabs to be retrieved	№ of stacks	№ of slabs	№ of relocations	№ of retrievals	Total number of moves
21	3	7.67	22.71	3.05	2.33	5.38
23	4	7.30	24.61	3.22	3.04	6.26
6	5	8.17	39.33	6.33	4.17	10.50
7	6	4.71	20.00	4.29	4.86	9.14
5	7	6.00	24.20	6.40	6.00	12.40
2	8	7.00	49.50	19.50	7.00	26.50
1	10	6.00	26	8	7	15
1	12	4	28	19	4	23

6. Conclusions

The aim of this thesis was to develop a fast and efficient algorithm specifically tailored for a particular type of real-world problem, namely the SSSP, as well as to provide a comprehensive review of existing literature on the block relocation problem. Corresponding to research question 1, the specifics of the SSSP were defined, the existing BRP methods were analyzed for their applicability to the SSSP and a new approach was designed specifically for tackling the SSSP. With reference to research question 2 (the performance of the new algorithm on benchmark instance sets), we can say that the presented algorithm has proven itself as a fast method, performing well for BRP with batch moves, both with and without additional constraints. In comparison to the best methods for standard BRP from the literature, such as LA-N and methods from the paper of Tricoire et al. (21), it has shown inferior but decent performance for the standard BRP, which is justifiable, as it has not been created for that purpose. However, it could still outperform some other methods. Unfortunately, it was not possible to compare the performance of CMTA for BRP-BM to that of the ICC-TREE and greedy algorithm by Zhang, Liu and Kopfer (10) because the most conclusive instances on which the authors tested were not available, as well as the comprehensive results of their testing on Caserta (11) instances. However, it has been highlighted by the authors that the performance of the greedy algorithm on larger instances was not of high quality, compared to ICC-TREE. On

the other hand, ICC-TREE had extremely long running times for large instances, starting from 6x10 (tiers x stacks).

The perk of CMTA is its applicability to BRP not only with batch moves, but also with movement and compatibility restrictions. It can also be applied to BRP with duplicate priorities, but only if the items retrieved are still assigned unique indices. To elaborate, let us assume that there are multiple items with same indices in the bay. In this case, CMTA does not distinguish between the items sharing the same index that remain in the bay. However, it has to assign additional indices to the items on the retrieval list sharing the same priority in order to distinguish them.

As CMTA and its implementation in this research were tailored for tackling a particular real-life problem, the items' dimensions and special requirements for stack height or item placement were taken into account. The source code of the algorithm implementation in C++ can be parameterized to adjust the algorithm to different variants of the problem, such as homogenous items, different stack heights and movement / incompatibility restrictions (as well as absence of those). We can confirm that CMTA performs well on real-world instances, requiring little CPU time and creating efficient move sequences.

As to research question 3, we have studied the fashion in which vehicle capacity impacts the solution of BRP-BM without additional constraints and parameters inherent to the SSSP, such that a bay contains homogenous items only. These tests have been conducted for different vehicle capacity values on the benchmark instances by Caserta (11). We have come to the conclusion that the best trade-off between increasing the capacity of the vehicle and improving the objective value is attained while expanding the capacity up to 4 slabs (see Table 6). Assuming a maximal lifting capacity of 4 items at a time, we have furthermore studied the improvement of objective values while altering the maximal stack height (H_{max}, the maximal height that a stack can reach). The results, as shown in Table 7, show significant improvement of objective values up to a certain height, while further increases in height, from that point on, do not generate significant improvements upon objective values.

Finally, the research question 4 has been addressed by discussing the usefulness of considering new kinds of moves and strategies while approaching the BRP-BM. As discussed in chapter 2, BRP-BM allows merged retrievals (i.e. assembling items

in the correct retrieval order so that they can be retrieved simultaneously, also known as batching moves). By doing so, it evokes further considerations, such as the appropriateness of unblocking more than one item at a time for the sake of simultaneous retrieval. To analyze this, the impact of unblocking additional items while not increasing the number of badly located items has to be evaluated. This approach is similar to assessing voluntary moves in look-ahead heuristics. A major difference is, however, that voluntary moves seek to generally improve the placement of the items in the bay and facilitate the next retrieval, whereas in the BRP-BM it is about unblocking items which can be retrieved together with the first target item. Not to forget is also the importance of detecting sequences of items which already are in a desirable order (known as decreasing sequences).

To summarize this master thesis, we can state that it successfully approached a specific modification of the BRP and highlighted its particularities and requirements for new or customized solution approaches. A review on existing BRP-related knowledge in the literature was presented in order to investigate the existing approaches to this problem, to compare them and to learn from them. The development of CMTA and the investigation of the way in which different values of vehicle capacity and maximal allowed height of the stack affect objective values are novel contributions to the existing literature. Therefore, the aims of the research have been fulfilled.

References

1. World Steel Association . [Online] 2020. [Cited: 15 10 2020.] https://www.worldsteel.org/.

2. Teniwuta W A, Hasyima C L. Decision support system in supply chain: A systematic literature review. *Uncertain Supply Chain Management.* 2020, Vol. 8, 1, pp. 131–148.

3. Power, D J. *Decision support systems: concepts and resources for managers.* Westport : Quorum Books., 2002.

4. Steel Commitee, Organisation for Economic Co-operation and Development. *Steel Market developments: Q2 2020.* Directorate for Science, Technology and Innovation. June 2020. DSTI/SC(2020)1/FINAL.

5. Voestalpine Group. [Online] 2020. [Cited: 15 10 2020.] https://www.voestalpine.com/stahl/en/Companies/voestalpine-Stahl-GmbH.

6. Nae Hee Han. Director, Economic Studies, World Steel Association. *88th Session, virtual meeting, 22-24 September 2020.* OECD Steel Commitee.

7. Voestalpine Steel Division. *Management Report.* First Half 2020/21.

8. Caserta M, Schwarze S, Voss S. Applying the corridor method to a blocks relocation problem. *OR Spectr.* 2011, Vol. 33(4), pp. 915-929.

9. Kemme, Nils. Container-Terminal Logistics. *Design and Operation of Automated Container Storage Systems.* 2013.

10. Zhang R, Liu S, Kopfer H. Tree search procedures for the blocks relocation. [ed.] Springer Science+Business Media New York 2015. *Flex Serv Manuf J (2016) 28:397–424.* 2015.

11. Caserta M, Schwarze, Voss S. A mathematical formulation and complexity considerations for the blocks relocation problem. *European Journal of Operational Research.* 2012, Vol. 219, 10.2016, pp. 96-104.

12. Expósito-Izquierdo C, Melián-Batista B, Moreno-Vega J. Marcos. An exact approach for the Blocks Relocation Problem. *Expert Systems with Applications.* 2015, Vol. 42, 17-18, pp. 6408-6422.

13. Zehendner E, Caserta M, Feillet D, Schwarze S, Voß S. An improved mathematical formulation for the blocks relocation problem. *European Journal of Operational Research.* 2015, Vol. 245, 2, pp. 415-422.

14. Petering Matthew E.H., Hussein M. I. A new mixed integer program and extended look-ahead heuristic algorithm for the block relocation problem. *European Journal of Operational Research.* 2013, Vol. 231, 1, pp. 120-130.

15. Bortfeldt A, Forster F. A tree search procedure for the container pre-marshalling problem. *European Journal of Operational Research.* ISSN 0377-2217, 2011, Vol. Volume 217, Issue 3, 531-540.

16. Tanaka S, Mizuno F. An exact algorithm for the unrestricted block relocation problem. *Computers & Operations Research.* 2018, Vol. 95, pp. 12-31.

17. Feillet D, Parragh S N, Tricoire F. A local-search based heuristic for the unrestricted block relocation problem. *Computers and Operations Research.* 2019, 108, pp. 44–56.

18. Kim Kap-Hwan, Hong Gyu-Pyo. A heuristic rule for relocating blocks. *Computers & Operations Research.* 33, 2006, pp. 940-954.

19. Tanaka S, Takii K. A Faster Branch-and-Bound Algorithm for the Block Relocation Problem. *IEEE Transactions on Automation Science and Engineering.* 2015, pp. 1-10.

20. Zhu W, Qin H, Lim A, Zhang H. Iterative Deepening A* Algorithms for the Container Relocation Problem. *IEEE Transactions on Automation Science and Engineering.* 2012, Vol. 9, 4, pp. 710-722.

21. Tricoire F, Scagnetti J, Beham A. New insights on the block relocation problem. *Computers & Operations Research,.* 2018, Vol. 89, pp. 127-139.

22. Sniedovich M, Voß S. The corridor method: a dynamic programming inspired metaheuristic. *Control and Cybernetics.* 2006, pp. 551-578.

23. Jovanovic R, Voss S. A chain heuristic for the blocks relocation problem. *Computers & Industrial Engineering.* 2014, Vol. 75, pp. 79-86.

24. Ting Ching-Jung, Wu Kun-Chih. Optimizing container relocation operations at container yards with beam search. *Transportation Research Part E: Logistics and Transportation Review.* 2017, Vol. 103, pp. 17-31.

25. Lin Dung-Ying, Lee Yen-Ju , Lee Yusin. The container retrieval problem with respect to relocation. *Transportation Research Part C: Emerging Technologies.* 2015, Vol. 52, ISSN 0968-090X, pp. 132-143.

26. Jin B, Zhu W, Lim A. Solving the container relocation problem by an improved greedy look-ahead heuristic. *European Journal of Operational Research.* 2015, Vol. 240, 3, pp. 837-847.