Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.

http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.

http://www.ub.tuwien.ac.at/eng

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Heuristic Solution Approaches for the Two Dimensional Pre-Marshalling Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Alan Tus
Matrikelnummer 1126941

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Dr. Andrea Rendl

Wien, June 5, 2014

(Unterschrift Alan Tus)

(Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Heuristic Solution Approaches for the Two Dimensional Pre-Marshalling Problem

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Alan Tus
Registration Number 1126941

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Dr. Andrea Rendl

Vienna, June 5, 2014

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Alan Tus
Sechshauserstraße 31/672, 1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Alan Tus)

Acknowledgements

This work is part of the project TRIUMPH, partially funded by the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) within the strategic programme I2VSplus under grant 831736. The author thankfully acknowledges the TRIUMPH project partners Logistikum Steyr (FH OÖ Forschungs & Entwicklungs GmbH), Ennshafen OÖ GmbH, and via donau – Osterreichische Wasserstrassen-GmbH.

Abstract

Today's shipping industry heavily relies on intermodal containers for quick and easy cargo manipulation. Container terminals are used as temporary storage for containers between two forms of transportation where containers are stacked into *container stacks* and those stacks are then ordered next to each other to form a *container bay*. To achieve high efficiency, all containers in a container bay have to be preordered in such a way that at pickup time all containers belonging to the current shipment are directly accessible by cranes.

A problem of reshuffling a container bay to achieve a layout where each container is accessible by a gantry crane at pickup time, using a minimal number of container relocations, is called the *Pre-Marshalling Problem* (PMP). A gantry crane can reach any container that is located on top of a stack. This work considers a new extension of the classical PMP in which we assume that the containers will be picked up by a reach stacker. A container is accessible by a so-called reach stacker if it is located on top of the left- or right-most stack of a container bay. This implies additional constraints that are discussed in this master thesis. We call our extension the *Two Dimensional Pre-Marshalling Problem* (2D-PMP).

We solve the 2D-PMP with two main approaches. First, we adapt a PMP strategic heuristic, the *Least Priority First Heuristic* (LPFH). Since the pickup order is predefined, we assign *priority values* to all containers (the ones with lowest priority value are picked up first). LPFH selects the container with the greatest *priority value* and positions it so that it can remain in its new position in the final container bay layout. For each container move, an origin and destination stack are chosen according to predefined rules. After that, intermediate moves are made to free up the selected container and destination slot allowing the selected container to be moved. Second, we use a metaheuristic, the *Max-Min Ant System* (MMAS). MMAS requires the problem to be represented as a path construction problem. Our initial container bay layout corresponds to the initial node and each move is an edge and each subsequent layout is a new node. The path length reflects the number of moves. Furthermore, we investigate simpler greedy and randomized construction heuristics, a PILOT method, and a local search procedure.

All algorithms are experimentally evaluated on benchmark instances. MMAS and 2D-LPFH are able to solve almost all instances. Among instances that both algorithms solved, results show that MMAS clearly outperforms 2D-LPFH as MMAS's solutions usually require less movements.

Keywords pre-marshalling problem, reach stacker, PILOT method, ant colony optimization, max-min ant system, least priority first heuristic, container terminal

Kurzfassung

Ein wesentliches Element der modernen Frachtindustrie ist der 20-Fuß-Container. Beim Wechsel des Transportmittels, beispielsweise von Bahn auf Schiff, werden diese in Container Terminals in Stapeln zwischengelagert. Solche Stapel werden nebeneinander angeordnet und bilden eine Container Bay. Um höchste Effizienz zu erreichen, müssen alle Container einer Bay so angeordnet sein, dass der Kran jeweils alle Container, die zu einer Lieferung gehören, zum Verladezeitpunkt direkt erreichen kann. Das Problem der Neuanordnung einer Container Bay sodass jeder Container für einen Brück-enkran zugänglich wird, wobei die Minimalanzahl an Containerbewegungen vorgenommen werden soll, ist als Pre-Marshalling Problem (PMP) bekannt. Ein Brückenkran kann einen Container erreichen, wenn dieser oben auf seinem Stapel liegt. In der vorliegenden Arbeit wird eine neue Erweiterung des PMP vorgestellt, für das die Benutzung eines Greifstaplers angenommen wird. Ein Greifstapler erreicht nur Container, die oben auf dem Stapel am äußersten rechten oder linken Rand der Container Bay liegen. Daraus ergeben sich zusätzliche Einschränkungen, die in dieser Arbeit erörtert werden. Wir nennen unsere Erweiterung das Zweidimensionale Pre-Marshalling Problem (2D-PMP).

Zwei Ansätze werden zur Lösung des 2D-PMP herangezogen. Erstens wird eine strategische PMP-Heuristik angepasst, die Least Priority First Heuristic (LPFH). Da die Abholreihenfolge vordefiniert ist, können allen Containern Prioritätswerte zugewiesen werden (diejenigen mit der geringsten Priorität werden zuerst abgeholt). LPFH wählt den Container mit dem höchsten Prioritätswert und positioniert ihn so, dass er an dieser Position bleiben kann. Für jede Containerbewegung wird ein Herkunfts- und ein Zielstapel nach vorher festgelegten Regeln ausgewählt. Danach werden die nötigen Containerbewegungen durchgeführt, die den Container und dessen Zielposition erreichbar machen, sodass der Container schließlich bewegt werden kann. Zweitens verwenden wir eine Metaheuristik, das Max-Min Ant System (MMAS). Dieses erfordert, dass das Problem als Pfadkonstruktionsproblem repräsentiert wird. Hier entspricht die ursprüngliche Konfiguration der Container Bay dem Startknoten, jede Containerbewegung einer Kante und jede weitere Konfiguration der Container Bay einem weiteren Knoten. Die Pfadlänge entspricht der Anzahl der Containerbewegungen. Zusätzlich untersuchen wir einfachere greedy und randomisierte Konstruktionsheuristiken, eine PILOT-Methode und eine Prozedur zur lokalen Suche.

Um sinnvolle Kontrollparameter zu finden werden alle Algorithmen experimentell an einer vordefinierten Menge von Testfällen evaluiert. MMAS und 2D-LPFH lösen fast alle Benchmarkfälle. An den Fällen, die von beiden Algorithmen gelöst werden können, zeigt sich, dass MMAS deutlich besser ist als 2D-LPFH da MMAS meistens Lösungen mit weniger Container-Verschiebungen findet.

Schlüsselworter: PMP, PILOT method, ACO, MMAS, LPFH

Contents

1 Introduction **1**
1.1 Objectives 3

2 The Two Dimensional Pre-Marshalling Problem **5**
2.1 Pre-Marshalling Problem 6
Problem parameters 6
2.2 Two Dimensional Pre-Marshalling Problem 7
Problem parameters extension 8
2.3 Assumptions 9
2.4 Problem representation 10
Problem input 10
Solution 10
2.5 Note on complexity 11

3 Related work **13**

4 Heuristic algorithms **17**
4.1 Evaluation functions 17
Blocking count evaluation function 17
Penalty evaluation function 18
4.2 Random and Greedy construction algorithms 19
4.3 Two Dimensional Lowest Priority First Heuristic 21
Original LPFH 21
Main changes 22
2D-LPFH 25

5 Metaheuristic algorithms **27**
5.1 PILOT method 27
Using a strategic sub-heuristic and compound moves 28
5.2 Ant Colony Optimization 30
Double bridge experiment 30
From colony to algorithm 32
5.3 Max-Min Ant System 33

Pheromone models 34
State based pheromone model 35
Move based pheromone model 35
Initializing, evaporating and updating pheromone values 35
Ant construction algorithm 37
Using a strategic heuristic and compound moves 39
MMAS algorithm 39

6 Local search algorithm 41
6.1 Shortcut heuristic 41

7 Experimental evaluation 43
7.1 Instances 43
7.2 Testing environment 45
7.3 Experiments 45
Result analysis 46
7.4 Random and greedy construction heuristics experiment 46
7.5 2D-LPFH experiments 47
Lambda values 48
Extended run time 50
7.6 PILOT experiments 56
Random and Greedy sub-heuristics 56
2D-LPFH sub-heuristic 57
7.7 MMAS experiments 59
Ant count 59
Alpha and beta values 60
Heuristic algorithms 61
7.8 Comparative analysis of best configurations 65

8 Conclusion 69
8.1 Critical reflection 69
8.2 Future work 70
8.3 Solution Network Analysis Procedure 71

A Instance samples 73

Bibliography 77

CHAPTER 1

Introduction

Containers are a core component of today's shipping industry. Before the 1950's there were no containers and ship cargo was scattered above and below deck. Loading and unloading was done manually and assisted by cranes. Working this way was inefficient because each piece of cargo had to be handled separately and cargo was often damaged, lost or stolen.

During the 1950's a concept of using containers to hold ship cargo was developed by Malcom McLean and engineer Keith Tantlinger. The first ship carrying its cargo in containers sailed in 1956. It was the SS Ideal-X, loaded with 58 35-foot (11 m) containers and it sailed from New Jersey to Houston, Texas. Using unified container sizes for packing cargo meant that all cargo could be handled almost the same way, thereby reducing the price of shipping, reducing loading times, increasing safety and significantly less cargo was damaged, lost or stolen than previously. This created a continuous flow of goods from production sources to customers which played a key role to enable consumers to enjoy goods from all around the world at affordable prices.

Since then, a series of ISO standards has been published by the ISO standards development technical committee for freight containers (ISO/TC 104)[1] to standardize the container and make it as widely used as it is today. Most of the deployed containers comply with ISO 668. The two most commonly used sizes are the 20-foot (6 m) and 40-foot (12 m) long containers. The 20-foot container, referred to as a *Twenty-foot Equivalent Unit* (TEU) became the industry standard reference so now cargo volume and vessel capacity are commonly measured in TEU. The 40-foot container (2 TEU) became known as the *Forty-foot Equivalent Unit* (FEU) and is the most frequently used container today.

In today's market, according to the *Container Census 2013*[2], the world's container fleet has reached 32.9 million TEU (1.3 billion m^3) and it is expected that another 1.6 million TEU have been added during 2013. According to (Chang et al., 2008), the most important factors for a shipping company when choosing a port of operations are: cargo volume, terminal handling

---

[1] `http://www.iso.org/`

[2] `http://www.drewry.co.uk`

Table 1.1: World's largest container terminals

Rank	Port, Country	Volume 2012 (mil. TEUs)	Volume 2011 (mil. TEUs)
1	Shanghai, China	32.53	31.74
2	Singapore, Singapore	31.65	29.94
3	Hong Kong, China	23.10	24.38
4	Shenzhen, China	22.94	22.57
5	Busan, South Korea	17.04	16.18

charge, land connection, service reliability and port location. Table 1.1 shows the world's five largest container terminals[3]. Not surprisingly, three of the top five terminals are located in China.

Table 1.2 shows the world's biggest container ships[4].

Table 1.2: World's largest container ships

Built	Name	Length (m)	Width (m)	Max TEU
2013	Maersk Mc-Kinney Moller	398	58	18 270
2013	Majestic Maersk	398	58	18 270
2013	Mary Maersk	398	58	18 270
2013	Marie Maersk	398	58	18 270
2012	CMA CGM Marco Polo	396	54	16 020

If we take into consideration the volumes mentioned in Table 1.1 and 1.2 one can surely understand why the shipping industry has to rely on computers to efficiently manage containers and container handling machinery.

Container terminals mostly have two container handling interfaces. On the sea-side they handle container vessels that can carry up to 18 000 TEU and on the land-side they need to keep up with the pickup and delivery schedule of trains and trucks. The container yard is used as temporary storage and it is the most critical point in this process.

Containers are stacked in *container stacks*, usually four or five high. A row of container stacks is called a *container bay*. Parallel container bays make a *container block*. Each block has a predefined number of bays and stacks. Two blocks are separated by transport lanes that are used by container handling vehicles like automated guided vehicles, straddle carriers and reach stackers. Stacking and retrieval operations are mostly carried out by gantry cranes (rubber-tired and rail-mounted gantry cranes). Ship to shore cranes are used for loading and unloading vessels.

Reach stackers (Figure 1.1) are extremely useful vehicles because they are very quick and agile. In less than half a minute they can pick up a container and drive away with it. They are usually the only vehicle found in smaller container terminals. Reach stackers are used for

---

[3] `http://www.worldshipping.org/about-the-industry/global-trade/top-50-world-container-ports`

[4] `http://en.wikipedia.org/wiki/List_of_largest_container_ships`

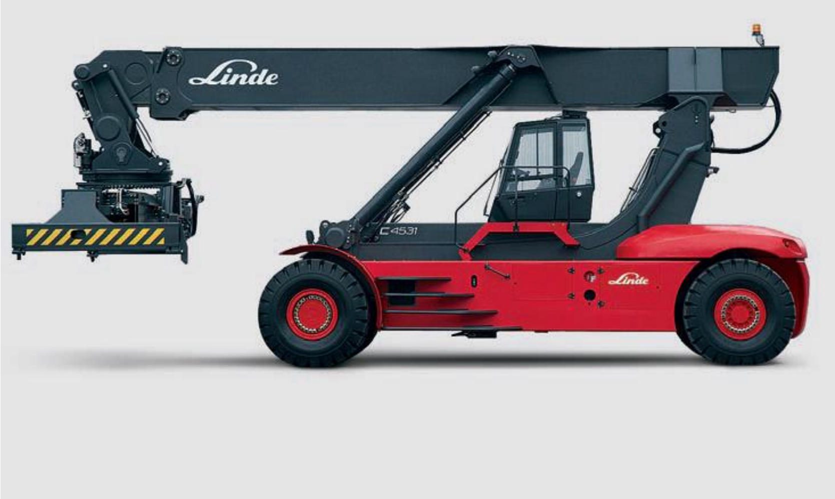


Figure 1.1: Reach stacker

loading and unloading trains and trucks as well as relocating containers within different areas of the container terminal.

Usually, each container in the container yard is already scheduled for pickup. This helps plan the layout of a container bay because we can assign a *priority* to each container. Containers with a *lower priority value* have a closer pickup time than the ones with a *higher priority value*. Containers in a stack are accessible by a LIFO (last-in-first-out) principle so if a container with a lower priority value is placed beneath a container with a higher priority value it would not be directly accessible at pickup time. This requires the crane to do additional moves to retrieve the needed container from within the container bay, thereby delaying the delivery process.

The problem of reshuffling a container bay to achieve a layout where each container is accessible at pickup time, using a minimal number of container relocations, is called the *Pre-Marshalling Problem* (PMP). The same process can be applied to all container bays in a container yard. This is done between delivery and pickup time.

1.1 Objectives

The problem we consider in this thesis is a new extension of the PMP. We extend the classical PMP to a new problem version where the containers can only be removed by a reach stacker. This means that at pick up time the containers can only be retrieved if they are on top of the outermost tiers of the container bay. We call it the **Two Dimensional Pre-Marshalling Problem (2D-PMP)**. It is formally defined in section 2.

In this thesis we:

- describe and define the 2D-PMP,
- show that it is possible to adapt a PMP heuristic to the 2D-PMP, we adapt the Least Priority First Heuristic (Expósito-Izquierdo et al., 2012)

- use a naive approach for solving the 2D-PMP
- use Ant Colony Optimization and PILOT method metaheuristics for solving the 2D-PMP
- implement a local search procedure
- run experiments to set a benchmark value for the 2D-PMP.

In chapter 2 we define all terms needed for understanding this thesis and formally define the 2D-PMP. Afterwards, in chapter 3 we give an overview of related work for the PMP and all algorithms used in this thesis.

Further, in chapter 4 we explain the Least Priority First Heuristic (LPFH) and present our adapted version, the Two Dimensional Least Priority First Heuristic (2D-LPFH). Our adapted version uses the same steps as the original LPFH, but we add additional constraints where needed, to satisfy the 2D-PMP.

Chapter 5 discusses the application of the $\mathcal{MAX}-\mathcal{MIN}$ Ant System (MMAS) (Dorigo and Stützle, 2004) version of the Ant Colony Optimization (ACO) (Dorigo and Stützle, 2004) in solving the 2D-PMP. To achieve this we represent the 2D-PMP as a path construction problem and use a custom designed pheromone model to represent the collective knowledge of the ants.

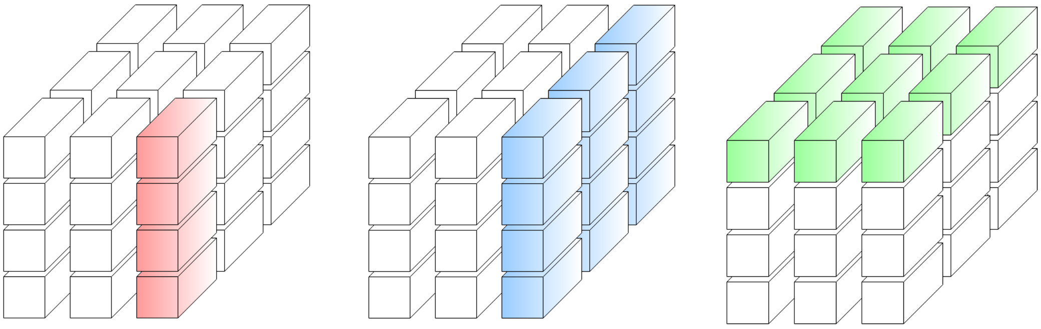
Next, in chapter 6 we present a local search algorithm that is based on our representation of the problem as a path. The algorithms looks for "shortcuts" in the constructed path (solution) and by using them, finds a shorter, more optimal solution.

Finally, we run a series of experiments with the presented algorithms on a predefined set of instances. For performance comparison we use a greedy and a random algorithm, along with the PILOT method (Duin and Voß, 1999; Voß et al., 2005). The evaluation results are discussed in chapter 7 and we give our conclusions in chapter 8.

CHAPTER 2

The Two Dimensional Pre-Marshalling Problem

Pre-marshalling problems arise in container terminals that have a temporary storage for containers. This storage is used as a "buffer" between two forms of transportation. Containers are stored by stacking them in *container stacks* (Figure 2.1a). When stacks are placed in a row, we call this row of stacks a *container bay* (Figure 2.1b). A row of containers within a bay (or across several bays) at the same height is referred to as a *tier* (Figure 2.1c). Each container within a container bay is assigned a *priority value* that defines the order in which the containers will be picked up. The *state* of a container bay is the current position of all containers within that container bay. We always start from a *initial state* and apply moves. By applying a move we change the state. Moves and priorities will be defined in more detail in the following section.



(a) Container stack (b) Container bay (c) Container tier

Figure 2.1: Container terminal

To minimize the amount of work needed at pickup time we want to have all containers in a container bay prearranged in such a way that each container is directly accessible by the crane and only needs to be moved to the vessel. This optimization is called *Intrabay optimization* and it is concerned with optimizing the state of a single container bay, i.e. moving containers within the bay such that emptying the bay of containers can proceed without further moves. The objective is to find a minimal sequence of moves that produces such a state.

There are two variants of Intrabay Optimization: the *Container Relocation Problem* (CRP) and the *Pre-Marshalling Problem* (PMP). In the CRP, the containers are shifted as well as removed from the bay, resulting in an empty bay after the final move. In the PMP, the containers are only shifted but not removed from the bay, i.e. all containers are still in the bay after the final move.

In this chapter, we have already defined some of the general terminology needed for understanding the problems discussed in this thesis. In the following sections we define the remaining terminology, present the PMP problem parameters, introduce the 2D-PMP and extend the problem parameters. Finally, we discuss the assumptions made in this thesis and provide a problem representation.

2.1 Pre-Marshalling Problem

The classical PMP deals with finding a minimal sequence of container moves for a gantry crane that results in a container bay state where all containers can be removed in a predefined order without additional moves.

Problem parameters

PRIORITY VALUE

In a container bay, we are given a set of containers $\mathcal{C}$ where each container $c \in \mathcal{C}$ is assigned a priority value $p_c \in \mathbb{N}$ that indicates the order in which the containers will be picked up from the bay: the container with the lowest p_c is always the next one to be picked up. Therefore, we define $\mathcal{C} = \{1, \ldots, C\}$ as an ordered multiset of all container priorities where $|\mathcal{C}|$ corresponds to the number of containers. To simplify the notation we write c instead of p_c. The number of containers with priority c (multiplicity of c) is given by $m(c)$.

SLOT

The container bay consists of w stacks $s \in \mathcal{S}$ that have a maximum height of h. We represent a stack s by an h-tuple $s = (u_1, \ldots, u_h)$ where each *slot* $u_i \in \mathcal{C} \cup \{0\}$ is either occupied by a container or is empty (in which case $u_i = 0$).

VERTICAL BLOCKING

Given two containers, container A with priority c_A and container B with priority c_B, positioned in the same stack, $s_A = s_B$, where container A is in a slot below container B, $u_A < u_B$, container A will be vertically blocked if the priority value of B is higher than of A, $c_B > c_A$. Container A is called the *blocked* container and container B is called the *blocking* container. Figure 2.2 shows a sample container bay with *blocked* and *blocking* containers.

VALID STACK

A stack is *valid* iff:

$$m_s(c) \leq m(c), \qquad \forall c \in \mathcal{C}, \tag{2.1}$$

$$u_i = 0 \rightarrow u_{i+1} = 0, \qquad \text{for } 1 \leq i < h - 1 \tag{2.2}$$

Meaning that a stack is valid if it does not contain more containers of priority i than are available (Equation 2.1) and if a slot at position i is empty, all subsequent slots must also be empty (Equation 2.2).

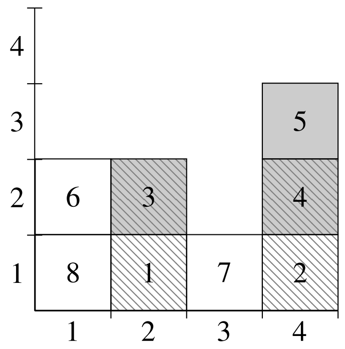


Figure 2.2: An example of a container bay with 4 slots of maximal height 4. Blocked containers are hatched, blocking containers are grey.

PERFECT STACK

A stack is called *perfect*, if it is valid and

$$u_i \geq u_{i+1}, \qquad \text{for } 1 \leq i < h-1 \tag{2.3}$$

holds, i.e. no container in the stack lies underneath a container with a higher priority value (there are no blocked containers).

VALID BAY

Furthermore, we define a container bay b as a set of stacks $\mathcal{B} = \{s_1, \ldots, s_w\}$ where $w = |\mathcal{S}|$. Container bay b is valid, iff:

$$s \text{ is valid}, \qquad \forall s \in \mathcal{B} \tag{2.4}$$

$$\sum_{s \in \mathcal{B}} m_s(c) = m(c), \qquad \forall c \in \mathcal{C} \tag{2.5}$$

PERFECT BAY

This means that if each stack in bay b is valid (Equation 2.4) and if for every priority c, the bay contains exactly $m(c)$ containers (Equation 2.5) then bay b is valid. A bay is called *perfect* if it is valid and all stacks are perfect.

STATE

Throughout this thesis, the word *state* refers to the layout of a container bay b. This is the combination of current positions of container priorities c with position $b_{i,j}$ (slot) in the bay, where $1 \leq i \leq w$ represents the stack and $1 \leq j \leq h$ the tier. States can be seen as two dimensional matrices of priorities where rows are *tiers* and columns are *stacks*. An example can be seen in Figure 2.2, empty slots are not shown.

MOVE

We define a container *move* $r = (i, j)$ as the movement of the topmost container on stack s_i to the top of stack s_j. A move is called *valid* iff $u^i_1 \neq 0$ and $u^j_h = 0$, i.e., at least one container is stored in stack i and the number of containers stored in stack j is less than the maximum height.

SOLUTION

A *solution* $\sigma = (r_1, \ldots, r_m)$ to the PMP is an m-tuple of valid moves that renders the initial bay perfect. In an optimal solution $\sigma^* = (r_1, \ldots, r_{m^*})$, m^* is minimal, thus $m^* \leq m$ holds for all m in other possible solutions σ.

2.2 Two Dimensional Pre-Marshalling Problem

In smaller container terminals, reach stackers (Figure 1.1) are used for delivering containers from the bay to trains and trucks. Most conventional reach stackers can only reach the outermost container stack. If we want to minimize the delivery time, the container with the lowest priority value always has to be positioned on top of one of the outermost non-empty stacks so that the reach stacker can retrieve it without the help of a gantry crane. This is not necessarily satisfied in a solution of the classical PMP. Figure 2.3 shows the bay from Figure 2.2 after it has been pre-marshalled (the classical way) and highlights those containers which are not accessible by the reach stacker when they are due for removal.

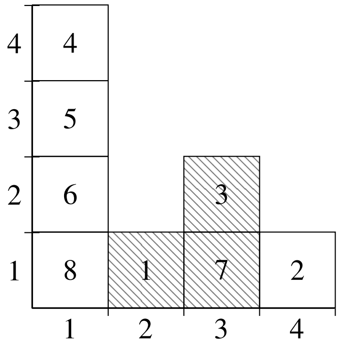


Figure 2.3: Perfect PMP solution. Hatched containers are not reachable by reach stackers at pickup time.

Therefore, we consider an extension of the basic PMP, that we call the *Two Dimensional Pre-Marshalling Problem* (2D-PMP), where we want to find a sequence of container relocations such that in the final bay layout, all containers are positioned on top of either the leftmost or rightmost non-empty stack when they are due for removal.

Problem parameters extension

HORIZONTAL BLOCKING

In addition to the previously defined vertical blocking, container A can also be *horizontally blocked* if it has to be removed by a reach stacker. A container c, positioned in stack i is horizontally blocked if at least one container on each side of stack i has a higher priority value than c. More specifically, container c at $\mathcal{B}_{i,j}$ is *horizontally blocked*, if:

- at least one container on c's left-hand side c'_l positioned at $\mathcal{B}_{l,k}$ with $1 \leq l < i$ and $1 \leq k \leq H$ has a higher priority value than c, thus $\exists c'_l : p(c'_l) > p(c)$ and
- at least one container on c's right-hand side c'_r positioned at $\mathcal{B}_{r,k}$ with $i < r \leq S$ and $1 \leq k \leq H$ has a higher priority value than c, thus $\exists c'_r : p(c'_r) > p(c)$.

Figure 2.4 shows an example of horizontal blocking. This kind of blocking typically occurs if we try to pre-marshal a bay the classical way. To unblock horizontally blocked containers there are two possibilities. First, we can move the blocked container or second, we can move the blocking containers from one side of the blocked container to make it accessible at the right moment from that side.

We can specify the property of the final bay in the 2D-PMP by extending the notion of a *perfect stack* and defining an *r-perfectly placed container* (where '*r*' stands for reach stacker). A

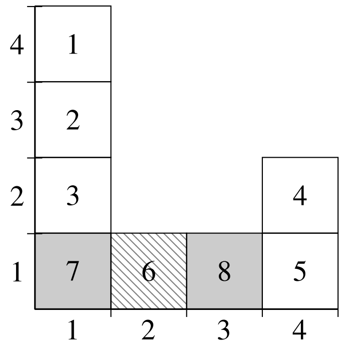


Figure 2.4: An example of horizontal blocking. The horizontally blocked container is hatched and the horizontally blocking containers are grey.

container c is an *r-perfectly placed container* if it is neither vertically nor horizontally blocked. A stack s is an *r-perfect stack* if it contains only *r-perfectly placed containers*. We call a container bay *r-perfect*, if it contains only *r-perfect stacks*.

R-PERFECT

2.3 Assumptions

To avoid confusion and reduce the set of problem parameters we make the following assumptions in this thesis:

- **All containers have the same size.** In container terminals, container bays usually contain only one container size for security reasons and to enable easier and quicker operation. Additionally, containers of different sizes usually cannot be stacked on top of each other.
- **The container weight does not matter.** Again, in most container terminals, containers are stacked to a maximum height of four or five containers high which is allowed by the ISO 668 standard even when they are fully loaded.
- **All containers can be handled by the reach stacker.** There might exist some special types of containers which are too heavy or the reach stacker is unable to grip. Since we assume that all containers within a bay are of the same type, we can thereby assume that these bays are not considered in this thesis.
- **The crane only performs crane movements within one bay when reshuffling a bay.** This is the usual way cranes operate in small container terminals because the inter-bay movements are too slow with respect to the intra-bay movements. To move between bays the whole crane has to reposition itself over a new bay.
- **All container priorities are unique**, i.e. $m(c_i) = 1, \forall i \in C$. This assumption can be made without loss of generality since problems with non-unique container priorities can be mapped to a unique-priority problem. This implies a higher complexity since a non-existent ordering between same-priority containers is introduced and has to be respected

in the solution, but solving a higher complexity problem means that the lower complexity problem can be solved at least equally well or better.

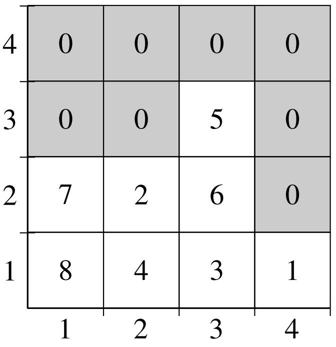
- **We consider all perfect states equally good.** Some perfect states are better than others, but to allow a ranking of perfect states we would have to introduce additional "soft" constraints that would enable a comparison of different solutions. An example where a perfect state $\mathcal{B}_1$ could be better than another perfect state $\mathcal{B}_2$ is if the terminal owner preferred bays where the number of containers was evened out across stacks. Another example is if the terminal operator wants to group containers belonging to one shipment on one side of the bay for quicker access. This is not covered by this thesis.

2.4 Problem representation

In the following sections we provide the problem representation of the 2D-PMP.

Problem input

We are given an initial state B that consists of w stacks of maximal height h. If a container c is located at stack $1 \leq s \leq S$ in slot (tier) $1 \leq u \leq H$ then $\mathcal{B}_{s,u} = p(c)$. In case the slot is empty, then $\mathcal{B}_{s,u} = 0$.



4	0	0	0	0
3	0	0	5	0
2	7	2	6	0
1	8	4	3	1
	1	2	3	4

Figure 2.5: Sample initial state with 4 stacks of maximal height 4 (the empty slots denote those slots that are set to 0)

Solution

A solution $\sigma = \{r_1, \ldots, r_m\}$ to the 2D-PMP is a list of m moves that render the initial state into an r-perfect bay and thus in which reach stackers can empty the bay without further relocations of containers. We specify that a move $r = (i, j)$ denotes moving the topmost container from stack i to the top of stack j. In an optimal solution $\sigma^* = (r_1, \ldots, r_{m^*})$, m^* is minimal, thus $m^* \leq m$ holds for all m in other possible solutions σ. A sample solution σ for the initial state from Figure 2.5 would be:

$$\sigma = \{(1,2),(1,2),(3,4),(3,4),(3,1),(2,3),(2,3),(2,1),(4,3),(4,3)\} \tag{2.6}$$

where each move (i, j) denotes moving the topmost container from stack i to stack j. The final *r-perfect* state is shown in Figure 2.6.

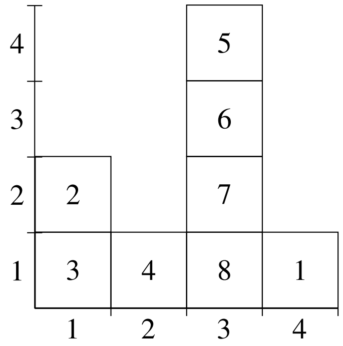


Figure 2.6: *r-perfect* state after applying σ from Equation 2.6 to initial state from Figure 2.5

2.5 Note on complexity

The Re-Marshalling Problem (RMP) is NP-hard (Caserta et al., 2011a) and it is closely related to the classical PMP. In the RMP, the same optimization process from the classical PMP is applied to multiple container bays at once, allowing containers to be moved within bays, as well as between bays using one or more cranes. Using multiple cranes introduces a new problem of crane interference that also needs to be taken into account.

In their paper, Caserta et al. (2011a) also discuss the differences between RMP and the classical PMP. They state that the classical PMP can be seen as a special case of the RMP with the following assumptions: there is only one source bay and only one destination bay, the source and destination bays are the same and only one gantry crane is used so crane interference does not have to be taken into account. Thereby, since RMP is NP-hard we conjecture PMP to be NP-hard.

If we extend the list of assumptions by declaring that containers can only be retrieved using a reach stacker (instead of a gantry crane) the 2D-PMP can be seen as a special case of the RMP. Thereby, since RMP is NP-hard we conjecture 2D-PMP to be NP-hard.

CHAPTER 3

Related work

Upon researching the available literature, a similar problem to 2D-PMP was not found. Therefore we present related work for the classical PMP.

Caserta and Voß (2009) present an approach to solving the classical PMP using the paradigm of the corridor method (Sniedovich and Viß, 2006). They limit the search space by defining constraints on the moves to be further examined. This is called *corridor definition/selection* and is done by first, selecting the target container and then creating a corridor of regulated width to be explored. The width of the corridor is limited by the size of the subset of *admissible stacks* to be explored. Further, they define a neighbourhood search, move evaluation and local search improvement.

Caserta et al. (2009) offers a binary encoding of the Blocks Relocation Problem (BRP), also known as the container relocation problem (CRP). The BRP is a similar problem to the PMP. Blocks are stored in bays similarly to containers and each block has a unique priority, thereby creating a total ordering. Blocks are retrieved according to priority and can only be retrieved from the top of a stack. Whenever the highest priority (lowest priority value) block is accessible, it is retrieved. This process is repeated until there are no more blocks. The authors offer a new *binary encoding* where they encode the current state into a binary $(N+W) \times (N+W)$-matrix, where N is the number of blocks and W is the number of stacks. Furthermore, this binary encoding is used to create a simple heuristic approach. In their heuristic approach they only move blocks located directly above the next block to be retrieved and apply heuristic rules for choosing the new stack. Their approach is implemented in C++. Tests results are compared to a heuristic based on expected value of future relocations (Kim and Hong, 2006) and a metaheuristic corridor method (Caserta et al., 2011b). It performs slightly better with respect to the number of moves, but demonstrates a significant improvement with respect to time, compared to the corridor method. The reason for this is the binary encoding method that offers quick access to information.

Lee and Chao (2009) introduce a neighbourhood search routine consisting of: a neighbourhood search process, an integer programming model and three minor subroutines. The neighbourhood search routine reduces the number of blocked containers with a neighbourhood search technique and the binary integer programming model reduces the number of moves produced in the first step. Interestingly, the authors state that blocking containers might be acceptable if reaching a perfect layout causes significantly more moves to be added to the solution.

Bortfeldt and Forster (2012) offer a heuristic tree search procedure for the classical PMP. In their paper, the authors classify all moves as *Bad-Good*, *Bad-Bad*, *Good-Good* or *Good-Bad* moves. A Bad-Good (BG) move is a type of move where the moved item was badly placed before the move and is well-placed after the move. Other move types are defined analogously. Further, the authors define a lower bound for the number of moves and offer a proof for the calculation steps. The lower bound is calculated based on the initial number of non-well-placed containers and newly defined types of containers. Finally, the authors present their heuristic tree search procedure where the initial node corresponds to the initial layout and all leaf nodes correspond to a final layout. Each successor L' of L is reached by a *compound move*, a permitted sequence of moves with respect to the predecessor L. The search algorithm works by calling a recursive procedure that generates a limited number of possible compound moves that are applied to the current state to reach the successor. This procedure is repeated until a solution has been found or the time limit exceeded. Compound moves are generated by selecting a *dirty* stack s containing non-well-located containers and attempting to perform the maximum number of BG moves from stack s followed by a minimum number of non-BG moves. Each BG move reduces the number of non-well-located containers and reduces the distance to a solution. Generated compound moves are filtered and sorted according to predefined rules to reduce the search space and direct the search. Their tree search procedure is implemented in C and compared to three different approaches: a corridor method based algorithm (Caserta and Voß, 2009), a neighbourhood search heuristic (Lee and Chao, 2009) and a mathematical model based on a multi-commodity flow network representation (Lee and Hsu, 2007). The tree search procedure outperforms all three approaches with respect to the average number of moves, as well as run time.

Forster and Bortfeldt (2012) offer the same heuristic tree search procedure described in Bortfeldt and Forster (2012), adapted for the CRP. The heuristic tree search procedure is implemented in Java and compared to four different approaches: a branch-and-bound algorithm (Kim and Hong, 2006), a corridor-based dynamic programming algorithm (Caserta et al., 2011b), the previously described heuristic approach using a binary representation (Caserta et al., 2009) and a refined corridor-based method (Caserta and Voß, 2009). Based on the results, it outperforms all four approaches with respect to the average number of moves and run time.

Expósito-Izquierdo et al. (2012) presents the Lowest Priority First Heuristic (LPFH) that will be discussed in detail in Section 4.3 and adapted for the 2D-PMP. The main idea of LPFH is to iteratively push the lower priority (higher priority value) containers toward the bottom tiers, while allowing the higher priority containers to reach the top tiers where they are available for pickup. This is done in reverse order of priorities, i.e. starting with the least priority container. Further, the authors define a PMP instance generator based on two concepts: *priority groups* and *levels in the bay*. Priority groups are sets of container priorities, while levels in the bay are sets of consecutive tiers of a bay that can be paired with a priority group. The lower the level occupied by high-priority containers, the harder the instance. The described instance generator is used in this thesis without using the two mentioned concepts, since they are not suitable for the 2D-PMP (discussed in section 7.1). Finally, the authors evaluate their work on a predefined set of instances from Caserta and Voß (2009) and compare their evaluation results with the same paper.

Huang and Lin (2012) present labelling algorithms that require containers to be sorted into predefined cells or regions. They are guided by the idea that perfect stacks need to be organized in such a way to provide space needed for re-marshalling non-perfect stacks.

Prandtstetter (2013) proposes a novel dynamic programming (DP) approach for the classical PMP embedded in a branch-and-bound (B&B) framework. To reduce the search space, the author introduces equivalence of DP states, stating that "whenever a state is reached for a second time, further search can be terminated in that branch of the DP tree". Further, B&B is a divide and conquer technique based on the idea that for each node in the search tree, an upper and lower bound can be defined. These bounds are used to make a decision if the node should be explored further.

CHAPTER 4

Heuristic algorithms

In this chapter we present constructive heuristics for 2D-PMP and evaluation functions used for evaluating container bay states. First, we describe the calculation steps of two evaluation functions and then we present our extension of the LPFH, and a greedy and random construction heuristic. All of the mentioned evaluation functions and heuristics are used in this thesis.

4.1 Evaluation functions

We use evaluation functions to approximately evaluate the *badness* of container bay *states* with respect to blocking containers. The lower the evaluation value, the better the state. Thus, we have a minimization problem where a solution with an evaluation value zero represents a final solution.

Blocking count evaluation function

The *blocking count evaluation function*, e_{bc} counts all *vertically and horizontally blocked* containers in a *state* $\mathcal{B}$. In case the container is both vertically and horizontally blocked, it is counted only once. This is achieved using a logical *or* operator. The evaluation function is shown below:

$$e_{vb}(\mathcal{B}) = \sum_{c \in \mathcal{C}} \max\left(f_{vb}(c), f_{hb}(c)\right), \forall c \in \mathcal{B} \tag{4.1}$$

$$f_{vb}(c) = \begin{cases} 1 & \text{if } c \text{ is a vertically blocked container} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

$$f_{hb}(c) = \begin{cases} 1 & \text{if } c \text{ is a horizontally blocked container} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

For all containers c in the given state $\mathcal{B}$, we select the maximum between the functions $f_{vb}(c)$ and $f_{hb}(c)$. We want to include only one penalty for one container in the final sum. $f_{vb}(c)$

returns 1 in case c is a vertically blocked container, 0 otherwise. $f_{hb}(c)$ returns 1 in case c is a horizontally blocked container, 0 otherwise. Finally, if the container is vertically or horizontally blocked we increase the sum by 1. The output value of $e_{bc}(\mathcal{B})$ is the number of (vertically and horizontally) blocked containers.

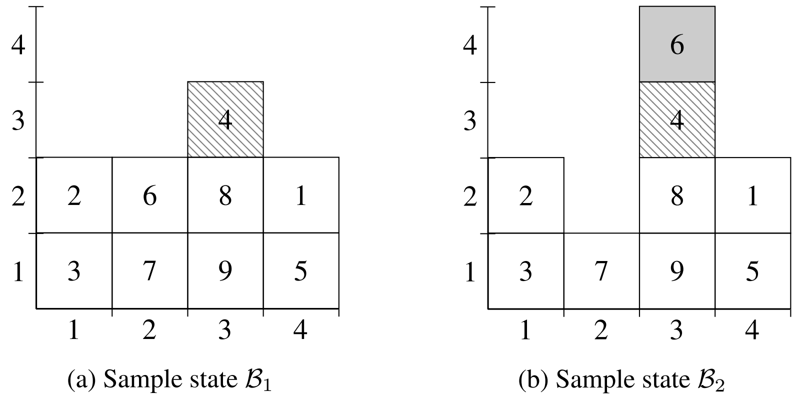
Penalty evaluation function

The *penalty evaluation function* extends the *blocking count evaluation function*. The main difference is that the penalty evaluation function recognizes different types of "problematic" containers and assigns each type its own penalty factor:

p_{vb} *vertically blocked* containers

$p_{vb''}$ *interfering containers*, all containers positioned above the vertically blocked container that are not blocked containers themselves. This includes blocking and non-blocking containers.

p_{hb} *horizontally blocked* containers



(a) Sample state $\mathcal{B}_1$ (b) Sample state $\mathcal{B}_2$

Figure 4.1: Sample states. Blocked containers are hatched, blocking containers are grey. State $\mathcal{B}_2$ requires more moves to resolve the blocking than state $\mathcal{B}_1$.

The calculation steps for the penalty evaluation function e_{pe} are shown below:

$$e_{pe}(\mathcal{B}) = \sum_{c \in \mathcal{C}} \max\left(p_{vb} * f_{vb}(c), p_{vb''} * f_{vb''}(c), p_{hb} * f_{hb}(c)\right), \forall c \in \mathcal{B} \tag{4.4}$$

$$f_{vb''}(c) = \begin{cases} 1 & \text{if } c \text{ is positioned above a blocked container} \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

The function e_{pe} evaluates all containers in a given state $\mathcal{B}$ and assigns each of them a penalty value depending on the type of container. We use the two functions f_{vb} and f_{hb} from Equations 4.2 and 4.3, and introduces a third function $f_{vb''}$. The function $f_{vb''}(c)$ returns 1 in case c is an interfering container, 0 otherwise. As in function e_{bc} (Equation 4.1), we do not want to count a container more than once so we apply the maximum penalty value we calculated for the current container.

This function provides a more accurate evaluation of states. The *blocking count evaluation function* would return the same value for both states shown in Figure 4.1. However, using the *penalty evaluation function* the state $\mathcal{B}_2$ shown in Figure 4.1b will have a slightly higher evaluation value than the state $\mathcal{B}_1$ shown in Figure 4.1a because there is a blocking container on top of the blocked container. Any of the penalty factors can be set to 0 and thereby exclude that type of container from the calculation.

4.2 Random and Greedy construction algorithms

We implemented a random and greedy construction algorithm to compare the other approaches with two completely naive approaches.

The random algorithm chooses a random move from the set of available moves and applies it. These steps are repeated until a solution is reached or there are no more possible moves. To avoid cycles, the same state cannot be visited more than once.

The random construction algorithm is given in Algorithm 4.1. It takes an initial state $\mathcal{B}$ and an evaluation function f as input. As long as the solution is not reached and there are possible moves, we chooses a random move m from the set of possible moves $\mathcal{M}$ in the current state $\mathcal{B}$ and apply it. Finally, we return a list of moves.

Algorithm 4.1 Random algorithm

```
Input: $\mathcal{B}$ : initial state; $f$ : evaluation function
$\sigma \Leftarrow$ empty list
while $f(\mathcal{B}) > 0$ do
    $\mathcal{M} \Leftarrow$ set of all possible moves in state $\mathcal{B}$
    if $\mathcal{M} == \emptyset$ then
        return $\sigma$
    end if
    $m \Leftarrow$ random move from $\mathcal{M}$
    append $m$ to $\sigma$
    apply move $m$ to $\mathcal{B}$ by yielding new state $\mathcal{B}$
end while
return $\sigma$
Output: list of moves
```

The average runtime of one iteration of the Random construction heuristic is $\mathcal{O}(1)$ since we only choose a random move from a set of moves and apply it.

The greedy construction algorithm uses an evaluation function to evaluate all reachable states and always chooses the move that leads to the state with the best (lowest) evaluation value. This process is repeated until a solution is reached or there are no more possible moves. To avoid infinite cycles, in a given state a move can only be chosen once. This does not avoid cycles, but allows an algorithm to perform more exploration without infinitely looping. Cycles are removed in the local search phase. We also implemented a version of this algorithm where the same state could not be visited more than once. Since the greedy algorithm is deterministic this results in many blocked partial solutions. Using the relaxation, more solutions are constructed.

Our greedy algorithm takes an initial state $\mathcal{B}$ and an evaluation function f as input. The steps are shown in Algorithm 4.2. We repeat the following steps as long as the solution is not reached, or there are no more possible moves: we choose the best move from $\mathcal{M}$ and apply it. Finally, we return a list of moves.

Algorithm 4.2 Greedy algorithm

```
Input: $\mathcal{B}$ : initial state; $f$ : evaluation function
$\sigma \Leftarrow$ empty list
while $f(\mathcal{B}) > 0$ do
    $\mathcal{M} \Leftarrow$ set of all possible moves in state $\mathcal{B}$
    if $\mathcal{M} == \emptyset$ then
        return $\sigma$
    end if
    $\mathcal{B}'' \Leftarrow \emptyset$
    $m'' \Leftarrow$ empty move
    for all $m' \in \mathcal{M}$ do
        apply move $m'$ to $\mathcal{B}$ by yielding new state $\mathcal{B}'$
        if $f(\mathcal{B}') < f(\mathcal{B}'') \vee \mathcal{B}'' == \emptyset$ then
            $\mathcal{B}'' \Leftarrow \mathcal{B}'$
            $m'' \Leftarrow m'$
        end if
    end for
    append $m''$ to $\sigma$
    apply move $m''$ to $\mathcal{B}$ by yielding new state $\mathcal{B}$
end while
return $\sigma$
Output: list of moves
```

The average runtime of one iteration of the Greedy construction heuristic is $\mathcal{O}(|\mathcal{M}|)$. The number of possible moves is directly influenced by the number of stacks w. The maximum number of possible moves is equal to:

$$|\mathcal{M}| = w \cdot (w - 1). \tag{4.6}$$

Equation 4.6 is valid in case all stacks have at least one free slot. Thereby we can move the top container of each stack to any other stack. In case there are stacks with no free slots on top, the number of possible moves is reduced and the can be calculated as follows:

$$\begin{aligned} |M| &= (w - \mathring{w}) \cdot (w - 1 - \mathring{w}) + \mathring{w} \cdot (w - \mathring{w}) && (4.7) \\ &= (w - \mathring{w}) \cdot (w - 1) && (4.8) \end{aligned}$$

In Equation 4.8 we divide stacks in two sets: the set of full stacks (no free tiers) and the set of non-full stacks where $\mathring{w}$ and w denote the cardinality of these sets, respectively. We start by calculating the number of possible moves for non-full stacks and then add the number of possible moves for full stacks. This equation is then reduced to a shorter form by factorization. The given formula is valid for all states.

Using Equation 4.6 we can put the time complexity in relation to the number of stacks. One iteration of the Greedy construction heuristic has an expected run time $\mathcal{O}(w^2)$. The overall number of moves has no limit. The algorithm will halt if it starts to loop or if some other condition is satisfied (reached move or runtime limit).

4.3 Two Dimensional Lowest Priority First Heuristic

The Two Dimensional Lowest Priority First Heuristic (2D-LPFH) is the 2D-PMP version of the Lowest Priority First Heuristic (LPFH) for the PMP introduced by Expósito-Izquierdo et al. (2012). In this section we first present the original LPFH, then discuss the main changes that are applied to it and finally present the 2D-LPFH. Our goal is to show that a classical PMP heuristic can be easily adapted to the 2D-PMP.

Original LPFH

The LPFH introduces the notion of *well-located* and *non-located* containers. Non-located containers are equivalent to our *blocking* containers, they all have to be repositioned to obtain a *perfect state*. Well-located containers are non-blocking containers and can remain in their current position in the final solution. The main idea of the LPFH is to choose a non-located container with the lowest priority (highest priority *value*, the one that will be picked up last) and apply one or more moves so that it becomes a well-located container. This process is repeated until there are no more non-located containers.

We present the basic steps of LPFH, taken from Expósito-Izquierdo et al. (2012) and shown in Algorithm 4.3. In this description we aim to familiarize the reader with the LPFH and explain the main ideas behind it so that we can discuss the changes that are needed and present our version, the 2D-LPFH. The details of each step are presented in the last part of this section in the 2D-LPFH description.

The original LPFH given in Algorithm 4.3 takes an initial state $\mathcal{B}$ as input and starts with an empty list of moves σ and a set of non-located containers $\mathcal{N}$. In each iteration we find a non-empty set of containers with the highest priority value $\mathcal{N}_{p_{max}}$ (line 4) and repeat the following steps until the set $\mathcal{N}_{p_{max}}$ is empty: (1) we select a target container c from $\mathcal{N}_{p_{max}}$ (line 6), (2) we select a destination stack s' where the container c will be placed (line 7), (3) we calculate and apply all necessary moves needed to well-position the container c in stack s' and append the moves to σ (line 8), and (4) we remove the container c from $\mathcal{N}_{p_{max}}$ (line 9). After each iteration we refresh the set of non-located containers (line 11) and start a new iteration if the set of non-located containers $\mathcal{N}$ is not empty (line 3). Finally, the algorithm returns a list of moves that renders a perfect state.

The target container c is a random container among the top λ_1 containers from the set λ_1

Algorithm 4.3 Lowest Priority First Heuristic – Basic steps

```
Input: $\mathcal{B}$ : initial state;
$\sigma \Leftarrow$ empty list
$\mathcal{N} \Leftarrow$ set of non-located containers
while $\mathcal{N} \neq \emptyset$ do
    $\mathcal{N}_{p_{max}} \Leftarrow$ set of non-located containers with highest priority value $p_{max}$
    while $\mathcal{N}_{p_{max}} \neq \emptyset$ do
        select a container $c \in \mathcal{N}_{p_{max}}$
        select a destination stack $s'$
        move container $c$ to $s'$ and append moves to $\sigma$
        remove $c$ from $\mathcal{N}_{p_{max}}$
    end while
    $\mathcal{N} \Leftarrow$ set of non-located containers
end while
return $\sigma$
Output: list of moves
```

of non-located containers with the highest priority value $\mathcal{N}_{p_{max}}$ sorted according to the lowest number of containers placed above it. If we use unique priorities, this parameter is not used since there is always only one container with the highest priority value p_{max}.

λ_2

INTERFERING CONT.

TEMPORARY STACKS

λ_3

The destination stack s' for the target container c is a randomly chosen stack among the top λ_2 stacks sorted according to the smallest number of containers that need to be removed so that container c can be well-located in that stack. To enable the move of container c to stack s' we need to remove the containers positioned on top of container c in its origin stack s and the container positioned in and above the destination slot in c's destination stack s'. We call these containers *interfering containers* and they are moved to *temporary stacks*. Temporary stacks are all stacks that have at least one free slot and are not the origin or destination stack of c. When removing containers from the destination stack and placing them on a temporary stack, the temporary stack is chosen among the λ_3 stacks with the lowest priority non-located container in that stack.

Main changes

We introduce three main changes: (1) the λ_1 parameter is not used, (2) a new way of choosing potential destination stacks and (3) the stack filling procedure is not used.

Since we assume unique priorities in the 2D-PMP, the parameter λ_1 has no more use as it could be at most one. In case we were to assume non-unique priorities the behaviour would not be in any way different than in the original LPFH previously described.

In the classical PMP it is enough for the LPFH to push the containers with higher priority values towards the bottom of the bay because we only have to deal with vertical blocking. In the 2D-PMP we have to take horizontal blocking into consideration and therefore a new strategy is developed. This is the key change that is made in comparison to the original LPFH.

In LPFH it is enough to choose a destination stack s' and find a slot where our target container c is well-located. This implicitly resolves vertical blocking. For 2D-LPFH we need a strategy that resolves horizontal blocking as well. We apply the same principle of pushing containers with higher priority values down, but now we need to push them down and to a side, or the middle. This is needed to resolve horizontal blocking and make the container well-located. We establish a concept of *adequate stacks*. Adequate stacks are a set of stacks $\mathcal{R}_c$ that are assigned to a container c based on its priority p_c. We present four different *models of adequate stack assignment* with increasing complexity and flexibility.

The *first model* defines adequate stacks by assigning an equal number of containers to each stack. We calculate the average number of containers per stack q (Equation 4.9) and divide the container's priority value p_c with q to obtain the *adequate stack* s' (Equation 4.10). Decimal values are rounded to the higher value.

$$q = \frac{|\mathcal{C}|}{|\mathcal{S}|} \tag{4.9}$$

$$s' = ceil\left(\frac{p_c}{q}\right) \tag{4.10}$$

The use of the first model forces the algorithm to always render the same solution which yields unnecessary moves and thereby produces worse solutions. Requiring a container to be positioned in an exact stack and well-positioned implicitly preallocates one slot for each priority. An example of a container bay solution acquired using this method is given in Figure 4.2a.

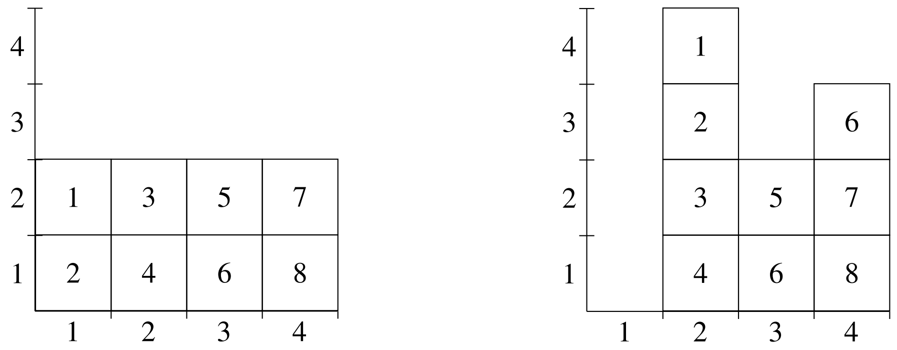
We extend this approach into a *second model* by adding one more stack into $\mathcal{R}_c$ for all containers except the ones located in the stack with the highest priority values (the last stack). The alternate stack s'' is the first stack after s' that holds containers with higher priority values. The formula given in Equation 4.11 defines the second model as the union of s' from the first model (Equation 4.10) and an additional stack which is the next stack unless s' was the last stack.

$$\mathcal{R}_c \Leftarrow \{s'\} \cup \{s'+1\}, \forall s' \in \mathcal{S} : s' < |\mathcal{S}| \tag{4.11}$$

In container bay configurations where the number of containers is bigger than half of the maximum capacity of the container bay, we encounter a situation where the number of containers assigned to a stack could be bigger than the capacity (height) of the stack. Since $\mathcal{R}_c$ is computed in each iteration of the LPFH we introduce a rule that if a stack $\hat{s}$ does not have at least one free slot, stack $\hat{s}$ cannot be added to $\mathcal{R}_c$.

The alternate stack allows more flexibility when constructing a solution yielding less unnecessary moves and rendering a set of solutions instead of only one solution. In Figure 4.2b we give an example using the same container bay configuration as previously, but acquired using the second model.

When retrieving containers, reach stackers can access the container bay from two sides. Both presented models yield solutions where the containers can be retrieved from only one side of the container bay. Let us call this type of solution *one-sided solution*. Accessing the container bay from only one side is a preferable scenario for some container terminals (e.g., where the reach stacker does not have the possibility to quickly reach the opposite side of the container bay). An



(a) Each priority value is assigned only one adequate stack using the *first model*.

(b) Each priority value is assigned at most two adequate stacks using the *second model*.

Figure 4.2: Examples of solutions with 4 stacks, a maximum height of 4 tiers and 8 containers. Each solution was acquired using a different stack assignment model.

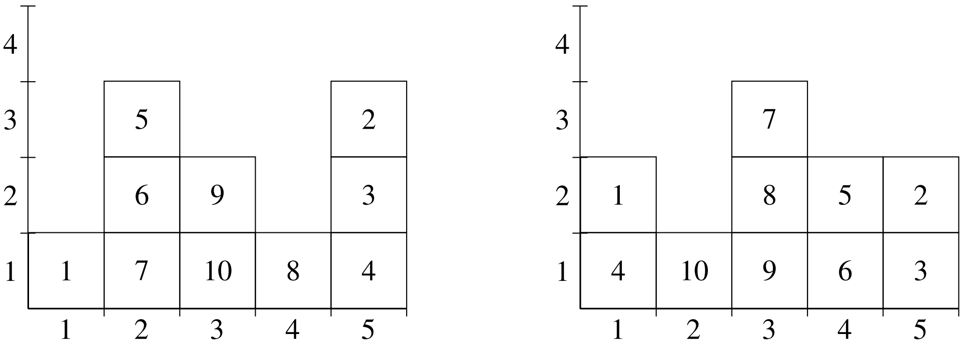
alternate scenario allows *two-sided solutions* where containers are retrieved from both sides of the container bay. The next model yields such solutions.

The *third model* pushes the high priority value containers towards the middle of the container bay. Let us start by arranging the containers from the first model in such a way that the higher priority value containers are positioned in the middle and as we move towards the sides, we encounter containers with lower priority values. In case we have a container bay configuration with an odd number of stacks, we start from the middle stack by positioning q containers in it. In the next step we have two options since there are two neighbouring stacks and we want to move in equal steps towards both ends of the container bay. Instead of choosing the left or right stack, we will add both stacks to $\mathcal{R}_c$ for twice as many containers than previously, $2 \times q$. The second step is also the first step for container bay configurations with an even number of stacks and we repeat it until we reach the outermost stacks. Figure 4.2b shows an example of a solution acquired using the third model. The two stacks in $\mathcal{R}_c$ can be viewed as *mirrored stacks* since they are equidistantly positioned with respect to the middle of the container bay. The exception is the middle stack in container bay configurations with an odd number of stacks.

In the *fourth model* we extend the third model by adding additional adequate stacks to $\mathcal{R}_c$. First, we assign adequate stacks as described in the third model. Second, we add the farther mirrored pair neighbouring with the pair currently in $\mathcal{R}_c$, except for the outermost stacks. In case there is an odd number of stacks, we add the two neighbouring stacks to $\mathcal{R}_c$. Another version of this model is acquired by adding the closer mirrored pair, in which case the middle stack(s) do not get any additional adequate stacks. Figure 4.3b shows a sample solution rendered using the fourth model.

As we can see, choosing the destination stack is a crucial step in the LPFH and changing its implementation directly effects the heuristic's behaviour, performance and usage scenario.

We introduce one more change. The stack filling sub procedure is not used in 2D-LPFH because we were not able to find an equivalent procedure that would improve the solution. In its original form it added unnecessary moves so it was removed. The heuristics performs well



(a) Each priority value is assigned at most two adequate stacks using the *third model*.

(b) Each priority value is assigned at most four adequate stacks using the *fourth model*.

Figure 4.3: Examples of solutions with 4 stacks, a maximum height of 4 tiers and 8 containers. Each solution was acquired using a different stack assignment model.

without this sub procedure so we decided to leave it out in this iteration.

2D-LPFH

We now provide a full description of the 2D-LPFH with all steps given in Algorithm 4.4. This version includes all changes previously discussed.

The 2D-LPFH takes three parameters, an initial state $\mathcal{B}$, λ_2 and λ_3. We first get the set of non-located containers $\mathcal{N}$ and an empty list of moves σ. As long as the set of non-located containers $\mathcal{N}$ is not empty, we perform the following steps: (1) We locate the container c with highest priority value in $\mathcal{N}$ (line 4). (2) Calculate a set of adequate stacks $\mathcal{R}_c$ for c (line 5), sort it ascending by the lowest number of interfering containers in each stack and select a random stack s' as the destination stack from the top λ_2 elements in $\mathcal{R}_c$. (3) We add all interfering containers from the origin s and destination s' stacks into $\mathcal{G}$ (line 8) where we sort them ascending according to their priority values. (4) We move each interfering container g in order from $\mathcal{G}$ to temporary stacks (line 10 - 16): First, we find the set of all available temporary stacks for g, $\mathcal{V}_g$ (all non-full stacks excluding s and s') (line 11). Second, we sort $\mathcal{V}_g$ ascending according to the highest priority valued non-located container in each stack. Third, we choose a random stack s'' from the top λ_3 stacks. Fourth, we move container g from stack s_g to stack s'_g in $\mathcal{B}$ (line 14) and append the move (s_g, s'_g) to the list of moves σ. (5) After all interfering containers have been moved to temporary stacks, we move container c form stack s to stack s' in $\mathcal{B}$ (line 17) and append the move (s, s') to the list of moves σ. (6) We update the set of non-located containers $\mathcal{N}$ and reiterate (line 19). Finally, when $\mathcal{N}$ is empty, we return the list of moves σ that render the initial state a valid final state.

We can find the container with highest priority value in $\mathcal{N}$ (line 4) in $\mathcal{O}(w)$ where w is the number of stacks, since we have to iterate over all stacks. We assume finding the maximum container priority value within a stack can be done in constant time since the maximum stack size is limited to a constant value of four containers. The set of adequate stacks can be calculated in $\big(1\big)$ using Equations 4.9, 4.9 and 4.9 and models explained in the previous section. All sorting

Algorithm 4.4 Two Dimensional Lowest Priority First Heuristic

```
Input: $\mathcal{B}$ : initial state; $\lambda_2$, $\lambda_3$: heuristic parameters
$\sigma \Leftarrow$ empty list
$\mathcal{N} \Leftarrow$ non-located containers in $\mathcal{B}$
while $\mathcal{N} \neq \emptyset$ do
    $c \Leftarrow$ container with highest priority value in $\mathcal{N}$ located at stack $s$
    $\mathcal{R}_c \Leftarrow$ set of adequate stacks for $c$
    sort $\mathcal{R}_c$ ascending by the lowest number of interfering containers in each stack
    $s' \Leftarrow$ select random stack from $\mathcal{R}_c$ among the top $\lambda_2$ stacks
    $\mathcal{G} \Leftarrow$ set of all interfering containers in $s$ and $s'$
    sort $\mathcal{G}$ ascending according to priority values
    for each $g \in \mathcal{G}$ that is positioned at stack $s_g$ do
        $\mathcal{V}_g \Leftarrow$ set of available temporary stacks for $g$
        sort $\mathcal{V}_g$ ascending by the highest priority valued non-located container in each stack
        $s'_g \Leftarrow$ select random stack from $\mathcal{V}_g$ among the top $\lambda_3$ stacks
        apply move $(s_g, s'_g)$ to $\mathcal{B}$ by yielding new state $\mathcal{B}$
        append move $(s_g, s'_g)$ to $\sigma$
    end for
    apply move $(s, s')$ to $\mathcal{B}$ by yielding new state $\mathcal{B}$
    append move $(s, s')$ to $\sigma$
    $\mathcal{N} \Leftarrow$ non-located containers in $\mathcal{B}$
end while
return $\sigma$
Output: sequence of moves
```

procedures can be carried out in $\mathcal{O}(w \log w)$ using e.g. Quicksort. The maximum number of interfering containers has a maximum value of seven containers in case the origin container is placed on the bottom of a full stack and its destination is at the bottom of another full stack. Thereby, the operation of repositioning interfering containers can be carried out in $\mathcal{O}(w \log w)$ due to the sorting of $\mathcal{V}_g$. Applying moves m to a state $\mathcal{B}$ is executed in $\mathcal{O}(1)$. The most time consuming operation within the algorithm are all sort procedures. Thereby, the time complexity of one iteration of 2D-LPFH is $\mathcal{O}(w \log w)$ where w is the number of stacks. The total number of compound moves is smaller or equal to the number of containers $\mathcal{N}$ since we well-position a container with each compound move. However the length of individual compound moves can not be predicted. Therefore, considering compound moves, the overall complexity of 2D-LPFH is $\mathcal{O}(|\mathcal{N}| w \log w)$.

CHAPTER 5

Metaheuristic algorithms

In this chapter we present two metaheuristic algorithms used for solving the 2D-PMP. First, we present the PILOT method, a heuristic repetition algorithm. Second, we outline the basic idea of Ant Colony Optimisation (ACO). Further, we present the chosen version of ACO and how we adapted it to the 2D-PMP. This is followed by two sections where we discuss different pheromone models and provide a detailed description of the ant construction algorithm. Finally, we present our implementation of ACO with a full algorithm description.

5.1 PILOT method

The PILOT method is a metaheuristic algorithm that uses a sub-heuristic to construct a partial solution in all possible directions. Finally, the direction of the best partial solution is chosen and the process is repeated until a complete solution is constructed. It was introduced by Duin and Voß (1999).

The authors of PILOT method refer to it as *"a tempered-greedy method, based on the repetition of another heuristic, the so-called pilot heuristic (or sub-heuristic)"*.

The sub-heuristic can be any algorithm that is used for solving the 2D-PMP. Figure 5.1 shows how the PILOT heuristic starts from a state $\mathcal{B}$ and expands it in all directions until it reaches the move limit of r moves (i.e. 2 moves) or creates a solution. As we can see, a solution was found in branch 4 after 2 moves, in all other branches no solution was found after r moves or there were no more possible moves. In case no solution was found in any of the branches, the branch that had the best evaluation value after 3 or less moves would be chosen.

The PILOT method is given in Algorithm 5.1. The PILOT method takes in four arguments: the initial state $\mathcal{B}$, a sub-heuristic h, the number of moves to look ahead r and an evaluation function f. We start out with an empty list of moves σ and run the algorithm until the evaluation function evaluates the current state $\mathcal{B}$ with 0 (line 2) or there are no possible moves (line 4). In each iteration we find the set of possible moves $\mathcal{M}$ in state $\mathcal{B}$ (line 3) and iterate over these moves (lines 9 to 16). For each move m' we apply it to $\mathcal{B}$ by yielding a new state $\mathcal{B}'$ and apply

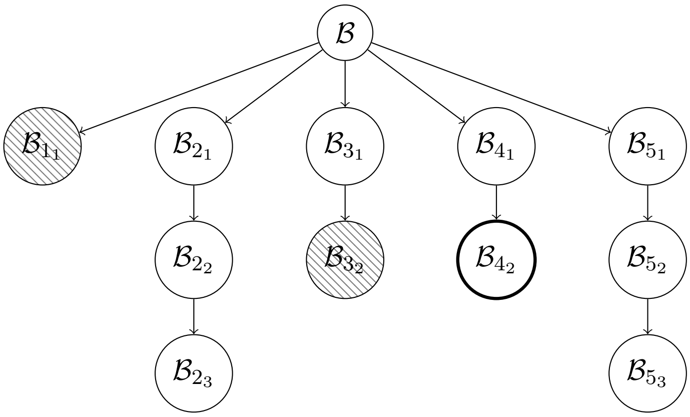


Figure 5.1: Pilot heuristic expansion of state $\mathcal{B}$. Each branch represent an available move in state $\mathcal{B}$ that is then expanded for at most two moves using a sub-heuristic. Blocked states (with no possible moves) are hatched, the solution has a bold outline.

the sub-heuristic h for r moves to $\mathcal{B}'$ (line 11). After the sub-heuristic has finished we check if $\mathcal{B}'$ is the best partial solution so far (line 12) and if true, we store the best move as m'' and the best partial solution as $\mathcal{B}''$. After we iterated through all possible moves $\mathcal{M}$ we append the best move m'' to the list of moves σ and apply it to the current state $\mathcal{B}$ (line 10).

The PILOT method behaves like the Greedy construction heuristic with an additional heuristic calculation at each step. Therefore, the average runtime of the PILOT method is $\mathcal{O}(w^2 \cdot r \cdot o_s)$ where w is the number of stacks, r is the number of steps for the sub-heuristic and o_s is the average run time of the sub-heuristic. There is no upper limit for the number of moves so we can not predict the overall runtime of the PILOT method.

Using a strategic sub-heuristic and compound moves

During our preliminary tests for PILOT method using 2D-LPFH as a sub-heuristic, we notice significantly poorer performance compared to 2D-LPFH without PILOT method. The reason for this is that 2D-LPFH is a *strategic* heuristic that chooses a container c to be moved, creates a sequence of moves $\overline{m}$ to enable the container c to be moved and finally moves it to its final position. PILOT method does not respect the described strategy, instead it applies the first move $\overline{m}(1)$ from the sequence and proceeds to the next iteration. Once $\overline{m}(1)$ is applied, there is no guarantee that 2D-LPFH will pursue the same strategy in the next iteration for two reasons: First, because of its stochastic nature and second, 2D-LPFH no longer decides which strategy to pursue, instead PILOT method makes a greedy decision based on heuristic values returned by the sub-heuristic. Thereby the sub-heuristic is reduced to a function that merely evaluates possible directions and PILOT method greedily chooses and applies moves, one at a time. This behaviour of PILOT method creates unnecessary moves and yields worse results.

To enable usage of strategic sub-heuristics, we add a deeper integration of the sub-heuristic into PILOT method by introducing two main changes: First, we introduce *compound moves* as described by Bortfeldt and Forster (2012). A compound move $\overline{m}$ is a sequence of moves, with

Algorithm 5.1 PILOT method

```
Input: $\mathcal{B}$ : initial state; $h$ : sub-heuristic; $r$ : number of moves to look ahead; $f$ : evaluation function
$\sigma \Leftarrow$ empty list
while $f(\mathcal{B}) > 0$ do
  $\mathcal{M} \Leftarrow$ set of all possible moves in state $\mathcal{B}$
  if $\mathcal{M} == \emptyset$ then
    return $\sigma$
  end if
  $\mathcal{B}'' \Leftarrow \emptyset$
  $m'' \Leftarrow$ empty move
  for all $m' \in \mathcal{M}$ do
    apply move $m'$ to $\mathcal{B}$ by yielding new state $\mathcal{B}'$
    apply sub-heuristic $h$ to $\mathcal{B}'$ for $r - 1$ moves
    if $f(\mathcal{B}') < f(\mathcal{B}'') \vee \mathcal{B}'' == \emptyset$ then
      $\mathcal{B}'' \Leftarrow \mathcal{B}'$
      $m'' \Leftarrow m'$
    end if
  end for
  append $m''$ to $\sigma$
  apply move $m''$ to $\mathcal{B}$ by yielding new state $\mathcal{B}$
end while
return $\sigma$
Output: list of moves
```

at least one move:

$$\overline{m} = (m_1, m_2, \dots, m_p)\,, (p \geq 1)\,. \tag{5.1}$$

Second, the set of possible moves $\mathcal{M}$ is a set of compound moves $\overline{\mathcal{M}}$ returned by the algorithm with respect to the given layout $\mathcal{B}$.

A move m is now a compound move $\overline{m}$ and the set of moves $\mathcal{M}$ is a set of compound moves $\overline{\mathcal{M}}$. For the random and greedy construction heuristic, the set of all moves is equivalent to the set of compound moves returned by the algorithm, where all compound moves have a length of one. Since the introduced changes do not change the behaviour of the previously described Algorithm 5.1, but only impact the way it is implemented, we do not reflect these changes in the pseudocode.

This fix has shown to improve performance, but it confines the search space to the moves that can be performed by the sub-heuristic, thereby disabling exploration. This eliminates the possibility of finding a better solution that the sub-heuristic is not able to construct itself, but this is not a problem for PILOT method as it is intended to serve as a "guide" for the sub-heuristic.

5.2 Ant Colony Optimization

A metaheuristic is a computational method that iteratively tries to improve an initial solution with respect to an evaluation function. Metaheuristics can cover a wide search space, but they do not guarantee to yield an optimal solution. They make little or no assumptions about the problem being solved so they can be applied to a given optimization problem with little or no modifications. Metaheuristics can be seen as a universal heuristic that guides a problem-specific heuristic towards areas of the search space containing better results. Examples of metaheuristics are tabu search (Glover, 1989, 1990), simulated annealing (Kirkpatrick et al., 1983) and ACO (Dorigo and Stützle, 2004).

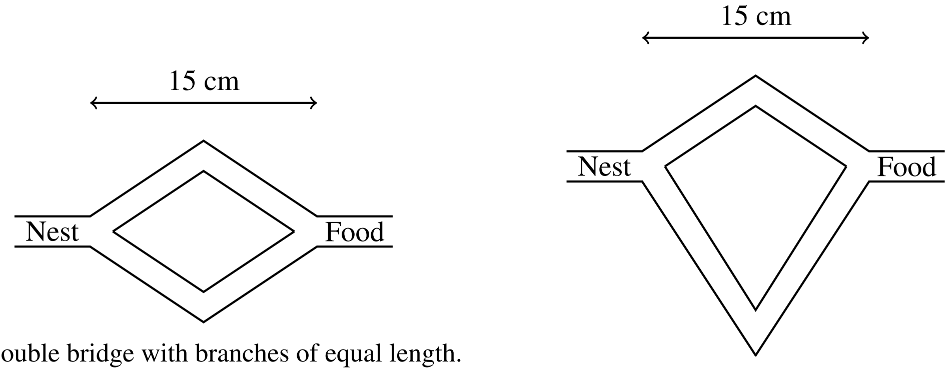
ACO is a population-based metaheuristic inspired by nature that tries to reproduce features of collective intelligence shown by real ants. In a population-based algorithm, instances locally create changes and thereby influence the creation of global features that can be used to improve an initial solution. This allows us to solve complex distributed problems using simple local interactions without the need to create a global centralized system. The basic idea of self-organizing principles that allow highly coordinated behaviour of real ants can serve as inspiration for creating new algorithms whose parts cooperate in solving a problem.

ACO was inspired by the way ants cooperate within a colony. Most ant species have a poorly developed visual perceptive or are completely blind. Their communication is based on leaving ant produced chemical trails on the ground. These chemicals are called *pheromones*. Each ant leaves a trail of pheromones behind him and he can sense pheromone trails left by other ants. Pheromones fade over time by evaporating. If an ant uses a path that another ant used before him, the pheromones trail on this path will be reinforced with new pheromones each time it is used. With time, the most used path will have a dominant trail of pheromones on it and this will cause (almost) all ants to follow the dominant trail.

Double bridge experiment

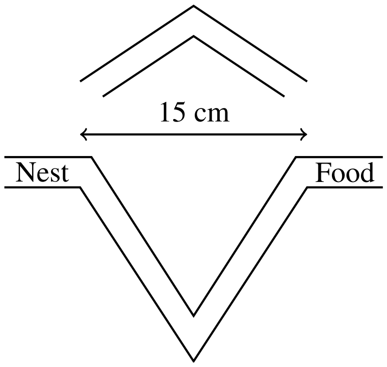
An example of the described behaviour is demonstrated in a double bridge experiment performed by Deneubourg et al. (1990) and Goss et al. (1989) using a colony of the Argentine ant (*I. humilis*). In this experiment, a double bridge is placed between a nest of ants and a food source, as shown in Figure 5.2. Ants are allowed to freely move between the nest and food source and the number of ants that chose one of the branches is observed over time. Before starting the experiment there are no pheromones on the paths and as ants walk they deposit small amounts of pheromones behind them. The experiment is repeated with different lengths of the two branches of the double bridge, where l_l is the length of the longer and l_s is the length of the shorter branch.

In the first case (Figure 5.2a) the bridge has two branches of equal lengths, $l_l/l_s = 1$. Since there are no pheromones, the ants do not have a preferred path so they randomly choose a branch. With time, the majority of ants choose only one of the two branches even though both branches are of equal length. This can be explained if we observe the pheromone trails. Due to random fluctuations, one branch is chosen by a few more ants than the other branch which causes more pheromones to be deposited on that branch. A stronger pheromone trail will attract more ants and result in an even stronger pheromone trail which will in the end cause all ants to converge to one branch. This process is an example of self-organizing behaviour we discussed earlier.



(a) Double bridge with branches of equal length.

(b) Double bridge with branches of different lengths.



(c) Double bridge with branches of different lengths where only the longer branch is accessible.

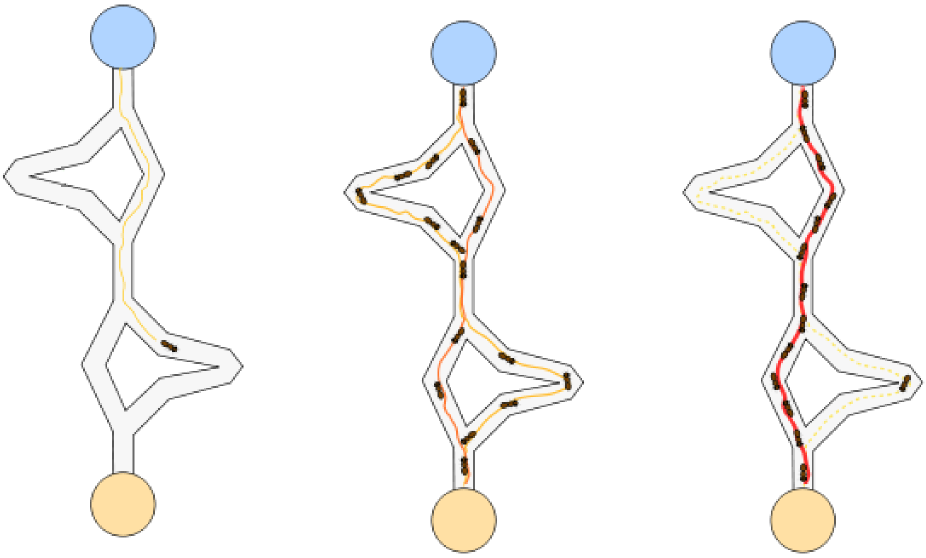
Figure 5.2: Double bridges experiment with Argentine ant colony. The figures show a double bridge placed between a nest of ants and a food source.

In the second case, the bridge has two branches where the longer one is twice the length of the shorter branch, $l_l/l_s = 2$ (Figure 5.2b). When the experiment is started the ants have no pheromone trail to follow so both branches appear equally "good" to them and both branches are chosen equally often as in the first case. The ants that choose the shorter branch are quicker to get to the food source and start their return to the nest. On their way back they have to make another decision and the higher pheromone values on the shorter branch will bias them towards the shorter branch. With time, more pheromones accumulate on the shorter branch and most of the ants choose it over the longer branch. Some ants still choose the longer branch and this can be seen as a form of *path exploration*.

In the third case, the ants are only presented with the longer branch (Figure 5.2c). After 30 minutes the shorter branch is offered but the ants were trapped in using the longer branch. The shorter branch is only sporadically chosen by very few ants. This can be explained by observing the pheromone trails. Over the first 30 minutes of the experiment the ants deposit a large amount of pheromones on the longer branch. After the shorter branch is offered, all ants will choose the dominant pheromone trail and the shorter branch is chosen only sporadically. Even though with time, more ants will start to choose the shorter branch, the majority of ants will still use the longer and keep reinforcing the pheromone trails. This autocatalytic behaviour cannot be avoided because the pheromone trails do not evaporate fast enough.

From colony to algorithm

Figure 5.3 shows two double bridges connected in a sequence between the nest and food source through three stages of ant behaviour. Each bridge has two branches where the longer one is twice the length of the shorter branch, $l_l/l_s = 2$ (Figure 5.2b). In the first stage (Figure 5.3a), there are no pheromone trails yet and the ants randomly select branches until they construct an initial solution. In the second stage (Figure 5.3b) the pheromone trails are existent but there is still no dominant trail that the majority of ants will follow so a lot of exploration can be seen. In the third stage (Figure 5.3c) there is a dominant pheromone trail that is chosen by the majority of ants and follows both shorter branches.



(a) Stage 1: initial path construction

(b) Stage 2: exploration

(c) Stage 3: converging

Figure 5.3: Three stages of the double bridge experiment with unequal lengths of brides. The ants move from their nest (blue) to the food source (yellow) and leave a pheromone trail behind them that is depicted with an orange or red line. The stronger the pheromone trail, the thicker the line.

ACO uses the ant’s behaviour to model a self-organizing distributed system for finding the shortest path. Real ants are replaced with *ant construction algorithms* and pheromone trails are replaced with a *pheromone model* that stores numeric values instead of ant produced chemicals. The ant construction algorithm uses *pheromone values* form the pheromone model and *heuristic values* from a heuristic function to make an informed decision in each step.

The three previously described stages can also be observed in ACO. Before the metaheuristic is started, an initial solution has to be constructed and the pheromone values have to be initialized. When the algorithm is started we repeat three basic steps: construct ant solutions, update pheromones and perform daemon actions (optional). After some time we might observe stagnation in the algorithm because it will converge towards a dominant solution that is strongly reinforced by pheromone values. These three steps are repeated (in parallel or sequence) and the specifics of the implementation are left to its author.

In Algorithm 5.2 we provide these steps in a pseudo-code for ACO exactly as it was described in Dorigo and Stützle (2004) in Figure 2.1, page 38.

Algorithm 5.2 ACO metahueristic – Basic steps

```
while schedule activities do
    construct ant solutions
    update pheromones
    daemon actions % optional
end while
```

Daemon actions are problem specific actions that are applied after the ant construction phase and in some cases after the pheromone update. They include centralized actions that cannot be performed by single ants, such as local search improvement or gathering of global information for future decisions.

5.3 Max-Min Ant System

ACO has multiple available versions and we need to choose a version that will be suitable for 2D-PMP and have good performance. We decided to use the $\mathcal{MAX}-\mathcal{MIN}$ Ant System (MMAS) (Stützle and Hoos, 2000) version of the ACO because it is the most researched and best performing version of ACO. We also consider it to have the best trade-off between simplicity and performance. Some guidelines and best practices are given in Dorigo and Stützle (2004) and we mostly use them when designing our MMAS algorithm for the 2D-PMP.

MMAS is a well-known extension of ACO that strongly favours the best constructed solutions. It is characterized by four features: First, only the best ant may update pheromone trails. Second, pheromone values have strict upper and lower bounds $\tau_{\min}$ and $\tau_{\max}$, respectively. Third, all pheromone values are initiated with $\tau_{\max}$ and the evaporation rate ρ is kept low. Fourth, the algorithm performs restarts after finding no improvement for a certain number of iterations. The first feature of favouring the best solution could cause stagnation so the remaining three features were introduced. Keeping pheromone values within limits does not allow the best path to be "too good" and the less visited paths to be "too bad". The initial maximum value of all pheromones encourages exploration in the beginning and the occasional resetting of pheromone values prevents stagnation.

For solving the 2D-PMP with MMAS, we consider the problem a path-construction problem, where *nodes* represent container bay states, and *edges* represent container movements. Thereby, we search for a (shortest) path from the initial node to a node that is a valid final state.

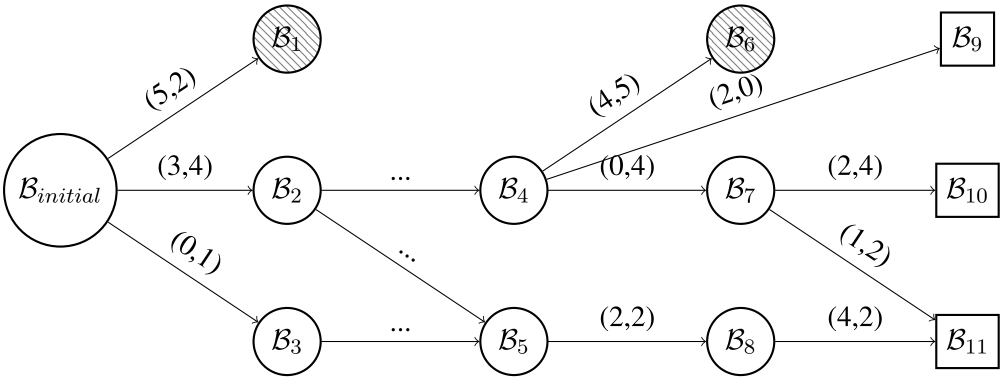


Figure 5.4: An example of an ACO solution network. Nodes are states and edges are moves. Squared nodes are valid final states and hatched nodes are blocked states (no more possible moves). Moves are denoted on edges as (s, s').

Since there are multiple solutions, there are equally many final nodes. This could prove to be a problem for ACO because the pheromone values for reaching the final solution are dispersed over multiple solutions and thus weakened. We did not notice an impact on performance but nevertheless, we partially solve this potential problem by automatically directing the ants toward a final solution if it is reachable in one move from the current state. Other possibilities for solving this problem are representing all final nodes the same way or adding a virtual ultimate final node that is reachable from all other final nodes.

The described problem representation can be seen in Figure 5.4. Nodes are numbered states $\mathcal{B}_i$ and edges are assigned moves (s, s'). Squared nodes are valid solutions, hatched nodes are blocked states with no possible moves and all other nodes are intermediate states in the solution construction.

In the following sections we describe individual algorithm components before we finally present our implementation of MMAS in Section 5.3.

Pheromone models

Pheromone values can be tracked using different features of the problem being solved. Since we represent the 2D-PMP as a path construction problem we want to use the states (nodes) and moves (edges) of our solution. Since we only evaluate states, we can say that each move is as good as the state it leads to. To add more information to the pheromone model we can also track a combination of our current state and the move we make from this state. In this sections we build up these ideas into pheromone models and explain the initialization, evaporation and update procedures.

We start with the simplest pheromone model that only observes the current state and extend it to a pheromone model that includes pheromone values for all possible moves out of each visited state.

State based pheromone model

Each pheromone value in the *state based* pheromone model reflects a move that leads to a particular state. This means that if some state $\mathcal{B}_j$ is reachable from more states $\mathcal{B}_i$, the pheromone value for $\mathcal{B}_j$ will be the same in all $\mathcal{B}_i$ states. The advantage of this pheromone model is that it aggregates pheromone values (in comparison to the move based p.m.) instead of dispersing them. Thus, we get less but more meaningful pheromone values. The obvious disadvantage of this model is that it fails to store valuable information about the current state. One could say that this is a greedy pheromone model because it only stores the quality of the next state.

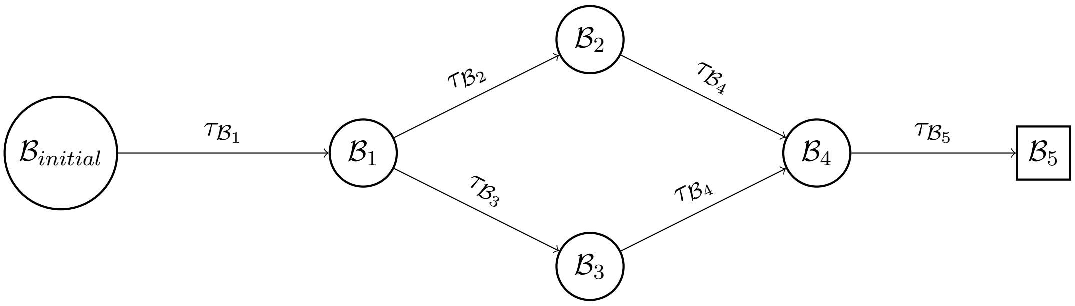


Figure 5.5: State based pheromone model example. States are nodes, squared nodes are valid final states and pheromone values are denoted on edges. Pheromone values depend only on the next state.

An example of how the state based pheromone model works can be seen in Figure 5.5. This model is specific because in any given state $\mathcal{B}_i$, if state $\mathcal{B}_j$ is reachable via a move $m_{\mathcal{B}_i,(s,s')}$, this move m will always have the same pheromone value $\tau_{\mathcal{B}_j}$. The model can be stored as a mapping between states and pheromone values.

Move based pheromone model

The *move based* pheromone model extends the *state based* model by pairing the information about the current state with each move. In a way, this model stores a state based model for each state thus making it a *two dimensional pheromone model*.

Each state $\mathcal{B}$ has a set of possible moves $\mathcal{M}$ and each of these moves m is assigned a pheromone value $\tau_{\mathcal{B},m}$. Each state will have its own unique set of pheromone values. A state $\mathcal{B}$ can be reached via multiple different moves and for each of these moves, $\mathcal{B}$ has a different pheromone value.

An example for this model is shown in Figure 5.6. As we can see, all pheromone values depend on the current state $\mathcal{B}_i$ and the move $m_{s,s'} \in \mathcal{M}$ that is being made. The pheromone value can be denoted as $\tau_{\mathcal{B}_i,m_{s,s'}}$.

Initializing, evaporating and updating pheromone values

Before we run the MMAS algorithm, pheromone values need to be *initialized*. Then, when it is running, pheromone values need to be *evaporated* as time passes and *updated* each time

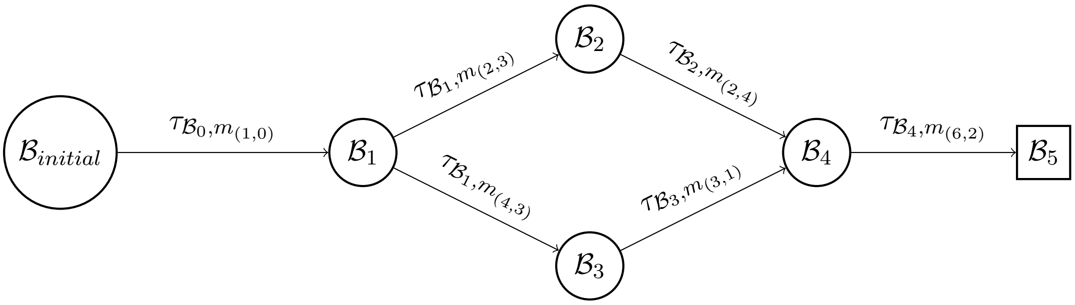

$\mathcal{B}_{initial}$
$\tau_{\mathcal{B}_0,m_{(1,0)}}$
$\mathcal{B}_1$
$\tau_{\mathcal{B}_1,m_{(2,3)}}$
$\tau_{\mathcal{B}_1,m_{(4,3)}}$
$\mathcal{B}_2$
$\mathcal{B}_3$
$\tau_{\mathcal{B}_2,m_{(2,4)}}$
$\tau_{\mathcal{B}_3,m_{(3,1)}}$
$\mathcal{B}_4$
$\tau_{\mathcal{B}_4,m_{(6,2)}}$
$\mathcal{B}_5$

Figure 5.6: Move based pheromone model example. States are nodes, squared nodes are valid final states and pheromone values are denoted on edges. Pheromone values depend on the current state and selected move.

an ant uses a path, just like in the real world example of ants shown in section 5.2. In the following text, we describe how pheromone values are initialized, evaporated and updated in our implementation of MMAS.

MMAS introduces a notion of minimal $\tau_{\min}$ and maximal $\tau_{\max}$ pheromone values to limit the influence of the best solution and encourage exploration of less visited paths. Exploration is also encouraged in the first iterations by setting the initial pheromone values to $\tau_{\max}$ and gradually evaporating them, allowing the algorithm to explore other solutions before a dominant solution appears. Pheromone limits are calculated using formulae given in Dorigo and Stützle (2004), Box 3.3 "Parameter Settings for ACO Algorithms with Local Search" (page 96) and shown below.

$$\tau_{\max} = \frac{1}{\rho * C^{bs}} \tag{5.2}$$

$$\tau_{\min} = \frac{1}{2 \cdot n} \tag{5.3}$$

In Equation 5.2, $\tau_{\max}$ is equal to the reciprocal value of the multiplication of evaporation rate factor ρ and global best solution value C^{bs}. We use the number of moves $|\sigma^*|$ as the values of the best solution. $\tau_{\max}$ is initialized using a heuristic algorithm to construct the first global best solution and recalculated each time a new global best solution is found. In Equation 5.3, $\tau_{\min}$ is equal to the reciprocal value of twice the number of ants n.

When starting the MMAS algorithm, all pheromone values need to be set to an initial value, which is in our case, the maximum value $\tau_{\max}$. The same process is repeated if the algorithms detects stagnation. Stagnation is detected by counting the number of consecutive iterations i without improvement of the global best solution σ^* as shown in Algorithm 5.4. After each iteration, we check if the number of consecutive runs without improvement i has reached the limit $i_{\max}$ (line 10) and if true, we reset all pheromone values to their initial state, $\tau_{\max}$. Otherwise, we proceed with evaporating pheromone values. The global best solution is always kept.

To simulate pheromone evaporation, we decrease all pheromone values for a given factor ρ to ensure that unvisited paths have a lower influence on future solutions. Evaporation is performed using the following formula on all pheromone values:

$$\tau'_{\mathcal{B},m} = \max\left(\tau_{\min}, \tau_{\mathcal{B},m} \cdot (1-\rho)\right) \tag{5.4}$$

The formula given in Equation 5.4 decreases the pheromone value by multiplying it with $(1-\rho)$, but also ensures that the pheromone value $\tau_{\mathcal{B},m}$ cannot be set to a lower value than $\tau_{\min}$.

Pheromone values are updated after each iteration to simulate depositing pheromone trails on a path. This is done by increasing all pheromone values that correspond to a state or combination of state and move (depending on the used pheromone model) found in the given solution σ. As previously mentioned, MMAS strongly favours the best ant, so we update pheromone values only for the best solution in the current iteration. To kick-start the pheromone values after a reinitialization, or simply reinforce the global best solution's path, MMAS will update pheromone values using the global best solution σ^* instead of the iteration best solution with probability p_{σ^*}. All pheromone values that correspond to the given solution's path are updated using the following formula:

$$\tau'_{\mathcal{B},m} = \min\left(\tau_{\max}, \tau_{\mathcal{B},m} + \frac{1}{|\sigma|}\right) \tag{5.5}$$

In Equation 5.5 we increase the pheromone value $\tau_{\mathcal{B},m}$ by the reciprocal value of the number of moves in the given solution σ. In case the new pheromone value is greater than $\tau_{\max}$, the new pheromone value will be equal to $\tau_{\max}$.

Ant construction algorithm

The ant construction algorithm is modelled according to guidelines Dorigo and Stützle (2004), Chapter 3. All ants start from an initial state $\mathcal{B}$ and construct a path σ that renders the initial state a valid final state. In each step the ants calculate probabilities for all possible moves $\mathcal{M}$ as a combination of the heuristic η and pheromone τ values. The heuristic values are calculated using one of the previously described heuristic algorithms (2D-LPFH, random and greedy construction heuristic and PILOT method). We call the heuristic to find a valid solution starting from the current state $\mathcal{B}$. If the heuristic finds a valid solution, the number of moves between the current state $\mathcal{B}$ and final solution is used as the heuristic value. A move is then chosen using a probabilistic action choice rule called the *random proportional rule*. These steps are repeated until a solution is reached. If a valid solution cannot be found, the ant returns a partial solution.

The ant construction algorithm is given in Algorithm 5.3. It takes two arguments: an initial state $\mathcal{B}$ and an evaluation algorithm f. In each iteration, we first determine all possible moves $\mathcal{M}$ for the current state $\mathcal{B}$. If $\mathcal{M}$ contains a move $\hat{m}$ that leads to a valid final state, we append $\hat{m}$ to our list of moves σ and return σ (line 5 to 8).

For each move m in $\mathcal{M}$, we calculate a probability $p_{\mathcal{B},m}$ using the *random proportional rule* (line 9):

$$p_{\mathcal{B},m} = \frac{[\tau_{\mathcal{B},m}]^{\alpha}[\eta_{\mathcal{B},m}]^{\beta}}{\sum_{l \in \mathcal{M}_{\mathcal{B}}}[\tau_{\mathcal{B},l}]^{\alpha}[\eta_{\mathcal{B},l}]^{\beta}}, \text{if } m \in \mathcal{M}_{\mathcal{B}} \tag{5.6}$$

Algorithm 5.3 Ant construction algorithm

```
Input: $\mathcal{B}$: initial state, $f$: evaluation function
 1: $\sigma \Leftarrow$ empty list
 2: $f_e \Leftarrow f(\mathcal{B})$
 3: while $f_e > 0 \vee$ other stopping criteria do
 4:   $\mathcal{M} \Leftarrow$ set of all possible moves in state $\mathcal{B}$
 5:   if $\mathcal{M}$ contains a move $\hat{m}$ leading to a final state then
 6:     append $\hat{m}$ to $\sigma$
 7:     return $\sigma$
 8:   end if
 9:   $P \Leftarrow$ calculate probabilities for all $m \in \mathcal{M}$
10:   $p_{sum} \Leftarrow 0$
11:   $r \Leftarrow$ random number between 0 and 1
12:   for each $m \in \mathcal{M}$ do
13:     $p_{sum} \Leftarrow p_{sum} + P_m$
14:     $m' \Leftarrow m$
15:     if $p_{sum} \geq r$ then
16:       break
17:     end if
18:   end for
19:   append $m'$ to $\sigma$
20:   apply move $m'$ to $\mathcal{B}$ yielding new state $\mathcal{B}$
21:   $f_e \Leftarrow f(\mathcal{B})$
22: end while
23: return $\sigma$
Output: list of moves
```

where $\tau_{\mathcal{B},m}$ refers to the pheromone value of move m from state $\mathcal{B}$ and $\eta_{\mathcal{B},m}$ refers to the heuristic value of move m from state $\mathcal{B}$. α and β define the relative influence of pheromone and heuristic values in the probability calculation.

In the next step we select the next move. A random value r between 0 and 1 is chosen (line 11). We select the next move m' by summing up the probability values P_m until the sum p_{sum} reaches or passes the chosen random value r (lines 12 to 18). The last move m whose probability P_m was added to the sum p_{sum} is chosen as the next move m' (line 14). We add m' to our list of moves σ and apply m' to $\mathcal{B}$ (line 20). This procedure is repeated until a valid final solution, time limit or maximum number of moves has been reached. Finally, we return a list of moves.

The an construction algorithm has an average runtime of $\mathcal{O}(|\mathcal{M}|)$ because we have to iterate over all moves to check if any move leads to a final state as well as calculate the pheromone and heuristic value of each move. The average runtime can also be express in relation with the number of stacks w using Equation 4.6, thereby making the average runtime equal to $\mathcal{O}(w^2)$. Unfortunately, we also have to take into account the runtime of the heuristic algorithm. The worst performing heuristic algorithm is the Greedy construction heuristic $\mathcal{O}(w^2)$, thereby making the worst case runtime of the ant construction heuristic $\mathcal{O}(w^4)$.

Each ant runs this algorithm and all ants are run in parallel. Given that the underlying hardware can support it, we can run an arbitrary number of ants without a major impact on the running time. This is allowed by the design of the algorithm, since the only shared resource is the read-only pheromone model.

Using a strategic heuristic and compound moves

It is important to note that if a strategic heuristic is used for calculating the heuristic values, we apply the same changes as described in Section 5.1. A move m is replaced by a compound move $\overline{m}$, and the set of possible moves $\mathcal{M}$ (line 4) is replaced by the set of possible compound moves $\overline{\mathcal{M}}$ that are returned by the heuristic algorithm. Further, this change is propagated to the pheromone model, where a move m is replaced by a compound move $\overline{m}$, so now a state is $\mathcal{B}$ is paired with a compound move $\overline{m}$. Thereby, pheromone values would be denoted as $\tau_{\mathcal{B},\overline{m}}$.

Once more, these changes do not change the behaviour of Algorithm 5.4, but rather the implementation, so we do not reflect these changes in the pseudocode.

MMAS algorithm

The detailed steps of the MMAS are given in Algorithm 5.4. The algorithm takes four parameters: the initial state $\mathcal{B}$, the number of ants n, the number of maximum consecutive iterations without improvement $i_{\max}$ and p_{σ^*}. Before the algorithm starts iterating we need to initialize the pheromone values (line 2). In each iteration the algorithm constructs n ant solutions (line 4) that are subsequently improved using a local search procedure (line 5). The best constructed solution σ is selected from the set of constructed solutions $\{\sigma_i\}$ (line 6). In case σ provides an improvement, the global best solution σ^* is updated and the no improvement counter i is reset to 0 (line 8), otherwise we increase i by 1. In case the number of consecutive iterations without improvement i exceeds the threshold $i_{\max}$ (line 10), the pheromone values are reinitialized (line 11) and i is reset to 0. Finally the pheromone values are updated in lines 16 to 21. The decision whether to use the iteration best or global best solution is based on the probability p_{σ^*}. These steps are repeated until either the number of iterations or time limit has been reached. The algorithm returns the best found solution σ^*.

Algorithm 5.4 $\mathcal{MAX}-\mathcal{MIN}$ Ant System for 2D-PMP

```
Input: $\mathcal{B}$: initial state; $n$: ant count; $i_{\max}$: maximum number of consecutive iterations without
    improvement; $p_{\sigma*}$: probability for using the global best solution for pheromone update
$i \Leftarrow 0$
initialize pheromone values
while stopping criteria not satisfied do
    $\{\sigma_i\} \Leftarrow$ construct set of $n$ ant solutions from $\mathcal{B}$
    $\{\sigma_i\} \Leftarrow$ apply local search to all solutions in $\{\sigma_i\}$
    $\sigma \Leftarrow$ find best solution in $\{\sigma_i\}$
    if $\sigma < \sigma^*$ then
        $\sigma^* \Leftarrow$ set $\sigma$ as new global best solution
        $i \Leftarrow 0$
    else if $i > i_{\max}$ then
        reinitialize pheromone values
        $i \Leftarrow 0$
    else
        $i \Leftarrow i + 1$
    end if
    $r \Leftarrow$ choose random number in $[0, 1]$
    if $r < p_{\sigma*}$ then
        update pheromones with global best solution
    else
        update pheromones with iteration best solution
    end if
end while
return $\sigma^*$
Output: list of moves
```

CHAPTER 6

Local search algorithm

In this chapter we present a local search procedure that uses our path representation of the solution from the ACO to find shortcuts in the path.

6.1 Shortcut heuristic

This local search algorithm explores all possible connections between two states of a solution. If a connection between two states is discovered, this means that the other moves in between those two states can be "cut out", thereby making the solution shorter.

Given a solution σ, the heuristic tries to find 'shortcuts' in the solution. More specifically, the algorithm tries to detect cases where two states $\mathcal{B}_1$ and $\mathcal{B}_2$ that are connected with moves $\{m_j, ..., m_k\} \in \sigma$ can be connected with a single move $\hat{m}$. In this case, the sequence $\{m_j, ..., m_k\}$ can be replaced by $\hat{m}$ in the solution and thereby shorten the solution σ for $k - j - 1$ moves. Fig. 6.1 illustrates such a shortcut between states $\mathcal{B}_2$ and $\mathcal{B}_5$, eliminating moves m_2, m_3, m_4, shortening the solution by two moves.

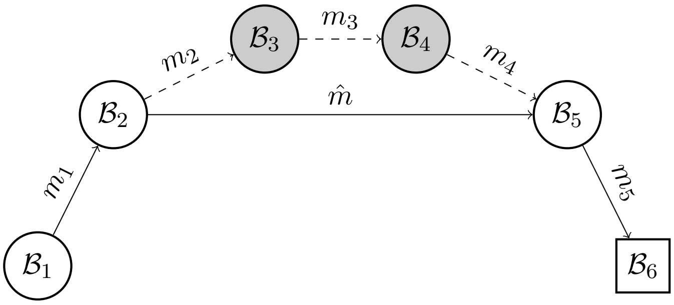


Figure 6.1: Shortcut heuristic depiction.

We define the neighbourhood of a state as the set of all states that are reachable with one move. The local search procedure starts by pairwise comparison of neighbourhoods of all states in a solution. If there is an intersection between two neighbourhoods this means that there is a move that connects the two states and that we found a *shortcut*. We want the shortcut to start as close as possible to the initial state and end as close as possible to the final state to "cut out" as many states as possible.

Algorithm 6.1 Shortcut heuristic - Local search improvement heuristic

```
Input: $\mathcal{B} \Leftarrow$ initial state; $\sigma \Leftarrow$ solution
 1: for all $m \in \sigma$ ascending do
 2:     $\mathcal{S}_\mathcal{B} \Leftarrow$ set of all reachable states from state $\mathcal{B}$
 3:     $\mathcal{S}_\sigma \Leftarrow$ set of all states in $\sigma$ after $\mathcal{B}$
 4:     if $\mathcal{S}_\mathcal{B} \cap \mathcal{S}_\sigma \neq \emptyset$ then
 5:         $\hat{m} \Leftarrow$ move between $\mathcal{B}$ and the $\mathcal{B}' \in \mathcal{S}_\mathcal{B} \cap \mathcal{S}_\sigma$ with maximal distance to $\mathcal{B}$
 6:         replace moves between $\mathcal{B}$ and $\mathcal{B}'$ in $\sigma$ with $\hat{m}$
 7:         apply move $\hat{m}$ to $\mathcal{B}$ yielding new state $\mathcal{B}$
 8:     else
 9:         apply move $m$ to $\mathcal{B}$ yielding new state $\mathcal{B}$
10:     end if
11: end for
12: return $\mathcal{B}$
Output: improved or equal solution
```

The shortcut heuristic is outlined in Algorithm 6.1, where we iterate over the set of moves in the given solution k in lines 1 to 11. In each of those iterations we check for intersections between the set of all reachable states $\mathcal{S}_\mathcal{B}$ in the current state $\mathcal{B}$ and the set of all states $\mathcal{S}_\sigma$ of the current solution σ occurring after the current state $\mathcal{B}$. In case the intersection $\mathcal{S}_\mathcal{B} \cap \mathcal{S}_\sigma$ is not an empty set (line 4), then the state(s) in $\mathcal{S}_\mathcal{B} \cap \mathcal{S}_\sigma$ can be reached by one move from $\mathcal{B}$. Therefore, we choose the state $\mathcal{B}'$ in $\mathcal{S}_\mathcal{B} \cap \mathcal{S}_\sigma$ that has the biggest distance from $\mathcal{B}$ (i.e. the state $\mathcal{B}'$ that is closest to the final state), and replace the moves between $\mathcal{B}$ and $\mathcal{B}'$ in the solution with the move m that connects $\mathcal{B}$ to $\mathcal{B}'$ (line 6). The final result is an improved or equal solution.

When not implemented well this algorithm will run too long to be a useful local search procedure that can be used between iterations of another algorithm. Since we have pairwise comparison with neighbourhood intersection at each pair, the average runtime is $\mathcal{O}(n^3)$ where n is the number of moves within the solution σ. The pairwise comparisons need to be efficient and we need to enable an early return from the local search procedure. This is done by starting from the first state and comparing it with all states from the last one to the state after the current one. Thereby we can stop as soon as we find a shortcut because it will be the longest possible shortcut. For further performance improvement we also encoded all states as strings and performe moves by shuffling characters within a string to reduce the number of operations in heavily used functions like state comparison and applying moves to states. Unfortunately all mentioned optimizations will not reduce the time complexity of our local search algorithm, but they will reduce the number of preformed operations and thereby take less time to execute the algorithm.

CHAPTER 7

Experimental evaluation

In this chapter we present the experimental evaluation of the developed algorithms. We start by presenting the set of instances used for testing and our testing environment. This is followed by multiple sections in which we explain the experiment and present the results one after the other.

The first sections describing experiments and their results cover the random, greedy and 2D-LPFH heuristics, followed by the PILOT method and MMAS. In the last section, we offer a comparative analysis of MMAS and 2D-LPFH.

7.1 Instances

Our experiments are conducted on a predefined set of instances. We divide the instances into categories based on the number of stacks and their occupancy rate. Each category contains 50 instances.

Our set of instances consists of container bays with $h = 4$ tiers, $w \in \{4, 6, 8, 10\}$ stacks and $q \in \{50\%, 75\%\}$ occupancy rate. An occupancy rate of, e.g. $q = 75\%$ means that a container bay configuration with $h = 4$ tiers and $w = 8$ stacks will have $|\mathcal{C}| = h \cdot w \cdot q = 24$ containers. Different combinations of the number of stacks and occupancy rate allow for *8 categories* of instances. Within each category there are no further levels of difficulty. All containers within an instance have unique priorities and are randomly positioned. This set of instances is referred to as the *regular set* and it contains 400 instances.

For running preliminary tests we use a *sample set* that consists of 50% of instances of each category in the regular set. Since all instances are randomly generated and do not have a particular ordering, we select the first 25 instances of each category from the regular set as the subset of instances for the sample set. This set contains 200 instances.

After preliminary testing result, we discovered that a category of instances with more stacks, could provide more insight into the performance of MMAS and 2D-LPFH. Therefore we define four further categories with $h = 4$ tiers, $w^+ \in \{12, 14\}$ stacks and the previously defined occupancy rates $q \in \{50\%, 75\%\}$. This set is referred to as the *BIG set* and it contains 200 instances.

h	w	$\|\mathcal{C}\|$	q	Category name	Sample	Regular	Big
4	4	8	50%	q50w04	25	50	–
	6	12		q50w06	25	50	–
	8	16		q50w08	25	50	–
	10	20		q50w10	25	50	–
	12	24		q50w12	–	–	50
	14	28		q50w14	–	–	50
4	4	12	75%	q75w04	25	50	–
	6	18		q75w06	25	50	–
	8	24		q75w08	25	50	–
	10	30		q75w10	25	50	–
	12	36		q75w12	–	–	50
	14	42		q75w14	–	–	50

Table 7.1: An overview of instance categories, their short names and number of category instances in the three sets of instances.

All instance categories are listed in Table 7.1. For easier reference, we define a short name for each category. The short name is an abbreviation specifying the number of stacks and occupancy rate. E.g., q75w10 stands for a container bay configuration with $q = 75\%$ occupancy rate and $w = 10$ stacks. The table also states the number of instance from each category included in different test sets.

For generating instances we use the PMP instance generator provided by Expósito-Izquierdo et al. (2012). In the instance generator we only set the number of bays, tiers, stacks and containers to generate one of the previously described categories. The number of groups and levels is set to one since we use unique priorities and we cannot use the difficulty levels predefined for PMP due to different constraints of the problems. All instances are generated randomly and they are valid and unique.

We do not use virtual tiers in our experiments. A *virtual tier* is an additional tier that is added on top of the existing container bay. The algorithm is allowed to use a virtual tier as a regular tier and the solution can contain containers in a virtual tier. All our instances are $h = 4$ tiers tall and the algorithm is not allowed to place containers above the top tier as it is done by Caserta and Voß (2009); Expósito-Izquierdo et al. (2012); Prandtstetter (2013) and other authors. The reason for this is that adding two virtual tiers on top of a $q = 75\%$ occupancy bay creates a $q' = 50\%$ occupancy bay with $h' = 6$ tiers. Not using these tiers allows us to stress-test our algorithms by leaving only one free tier for container manipulations. E.g., in a container bay configuration with $h = 4$ tiers, $w = 6$ stacks and an occupancy rate of $q = 75\%$ this leaves six slots for manipulating 18 containers.

Figures from A.1 to A.10 show examples of different sizes of container bays. The largest category of instances q75w14 (Figure A.10) is more than five times bigger than the smallest category q50w04 (Figure A.1).

7.2 Testing environment

All experiments are carried out on a machine with four Intel Xeon (E5645) processors, each with six cores at 2.40 GHz, along with 200 GB of RAM. The underlying system is Ubuntu 13.04. with Java 1.7. Please note that our experiments do not require this much computational power and memory to achieve good results. Our goal is to provide a testing environment where we do not have to deal with hardware limitations and can evaluate our algorithms full potential.

Our implementation of MMAS runs all ant construction algorithms of one iteration in parallel, each in its own thread. This way we utilize multiple cores and significantly decrease the computation time.

At the time of running our experiments there are no other processes running on our testing machine, except for the necessary operating system processes that cannot be disabled. Therefore all 24 cores were fully available for our tests. Due to long run times of most of our experiments, we run them in parallel, but always ensuring that at least two thirds of the computational resources were free to avoid slowing down any of the experiments.

7.3 Experiments

We perform experiments with *Greedy* and *Random* construction heuristics, *2D-LPFH*, *PILOT method* and *MMAS*. All mentioned algorithms, except for the greedy construction heuristic, are of stochastic nature. Therefore we run all test cases ψ times for each instance.

An overview of all experiments is provided in Table 7.2 where we specify the algorithm and aspect of that algorithm that is explored in this experiment, along with the set of instances the experiments were conducted on. The last column specifies ψ. In each experiment there are multiple test cases, 34 test cases in total.

Experiment		**Instance set**			
Algorithm	Aspect	Sample	Regular	Big	ψ
Greedy	–		✓	✓	1
Random	–		✓	✓	150
2D-LPFH	lambda values		✓	✓	150
2D-LPFH	extended runtime	✓			1
PILOT	greedy		✓	✓	1
PILOT	random		✓	✓	150
PILOT	2D-LPFH		✓	✓	150
MMAS	ant count	✓			5
MMAS	pheromone values	✓			5
MMAS	heuristic	✓			5

Table 7.2: Overview of conducted experiments and the used sets of instances

To ensure all experiments terminate within a reasonable amount of time, they are terminated in case no solution is found after a time limit of five minutes (wall clock time) or 500 moves of a single run, whichever is reached first. In case a solution is found the experiments terminate themselves. The same termination rules are used for MMAS, i.e. it terminates after 500 iterations or five minutes. The only difference is that the ant construction algorithm has a lower move limit of 250 moves since we do not want extremely bad solutions appearing in the pheromone trails. As you will notice in the results, 250 and 500 moves is a generous move limit since we expect solutions with less than 100 moves for the biggest categories of instances ($h = 14$, $q = 0.75$).

All experiments use the same evaluation function, *Penalty evaluation function* (Equation 4.4) with parameters $\alpha_1 = 1.0$, $\alpha_2 = 0.5$ and $\alpha_3 = 1.0$. After each algorithm is finished, the local search procedure *Shortcut heuristic* (Section 6.1) is applied.

Result analysis

Results of all experiments are stored in CSV files and analysed using the statistical computing language *R*[1]. We store the following information for the best runs: instance details (name, stacks, tiers, containers...), algorithm name, time and date of test start, evaluation value and duration (in milliseconds), move count and list of moves.

We always report only the best solution out of all runs. Results of instances with partial solutions are excluded from the results and we ONLY evaluate valid solutions (i.e. the evaluation value is zero). All reported average values and standard deviations include all valid solutions for the observed group. For comparison between two approaches we use the Wilcoxon paired rank sum test (WPRST) (Wilcoxon, 1945) with a significance level of $\alpha = 0.05$.

Below, we define the meaning of each abbreviation appearing in the coming sections. This meaning is preserved throughout the remainder of this work, unless stated otherwise.

N number of solved instances for given category and algorithm

Average average number of moves for given category and algorithm

St.dev standard deviation from the average number of moves for given category and algorithm

7.4 Random and greedy construction heuristics experiment

The random and greedy construction heuristics were used to observe how two completely naive construction heuristics would perform for the 2D-PMP. Tests were run on the whole set of instances. Unfortunately, they were barely able to solve any instances. The greedy construction heuristic solved two q50w04 instances with 73 and 68 moves. The random construction heuristic solved all 50 q50w04 instances with an average of 34.34 moves and a standard deviation of ± 23.37 as well as one q50w06 instance with 164 moves. Results are also provided in Table 7.3.

These naive construction heuristics failed at solving 2D-PMP instances because the solution requires an exact ordering of moves (a path) to be a valid final solution. The random construction

---

[1] `http://www.r-project.org/`

Algorithm	Category	N	Average	St.dev.	Duration	Dur.st.dev.
greedy	q50w04	2	70.50	3.54	1.00	0.00
random	q50w04	50	34.34	23.37	21.08	2.92
random	q50w06	1	164.00	–	38.00	

Table 7.3: Results for naive approaches over whole test set.

heuristic performed better because it was restarted multiple times, as opposed to the greedy construction heuristic that always takes the same path and does not need to be restarted. The evaluation function used in the greedy construction heuristic can easily lead the algorithm to run in loops (e.g. around a local minimum), which are not allowed, and thereby cause the algorithm to halt. If we give the random construction heuristic unlimited time and moves and allow infinite loops, the random algorithm would solve all instances. However, under the same conditions as all algorithms in this work, it was unable to do so. Another problem for these construction heuristics is a large search space.

7.5 2D-LPFH experiments

2D-LPFH has only two parameters, λ_2 and λ_3. We test these parameters, choose the two best parameter configurations and allow them to run equally long as MMAS to see if a longer run time yields a better result and to have a benchmark for comparison with MMAS.

Table 7.4 lists all 2D-LPFH experiments. All test cases are numbered and have a specific aspect of the algorithm that is being evaluated. We then specify the parameter values of λ_{2-3} as well as the number of iterations and the time-out where applicable.

Nr.	Experiment aspect	λ_2	λ_3	ψ	$t_{\max}$(s)
1	lambda values	1	1	150	–
2	lambda values	2	2	150	–
3	lambda values	2	3	150	–
4	lambda values	3	3	150	–
5	lambda values	5	5	150	–
6	lambda values	10	10	150	–
7	time-out	2	2	–	300
8	time-out	10	10	–	300

Table 7.4: List of 2D-LPFH experiments. The table shows the parameters used in individual experiments.

Lambda values

The lambda parameters, λ_2 and λ_3 of 2D-LPFH regulate its "greediness". A short reminder, λ_2 regulates the number of possible destination stacks for the current container and λ_3 regulates the number of possible temporary stacks for interfering containers. If we restrict these parameters to the lowest value, i.e. one, we observe a greedy and consistent behaviour. This is because the algorithm is reduced to a greedy version of 2D-LPFH and it always makes the same choice. Using higher values for these parameters allows the algorithm to make a random choice among the top λ_2 or λ_3 options and thereby explore the solution space. Multiple runs yield multiple solutions.

Although 2D-LPFH is an adapted version of the LPFH, the parameters retain the same meaning and recommended values as in the original algorithm because only necessary changes were made to satisfy the new constraints imposed by 2D-PMP. Therefore, we are able to use a similar testing strategy as in Expósito-Izquierdo et al. (2012). We tested 2D-LPFH by taking the best run of 150 runs. Since we wanted to find an overall best configuration we decided to run each test case on the whole set of instances.

We define six test cases for the lambda values where we gradually increase the values of both parameters. Experiments in Table 7.4 marked with 1 through 6 refer to the lambda test cases. The first case is the greedy parameter value one, followed by the three slightly "relaxed" cases. The last two cases are the "loose" parameter values of five and ten.

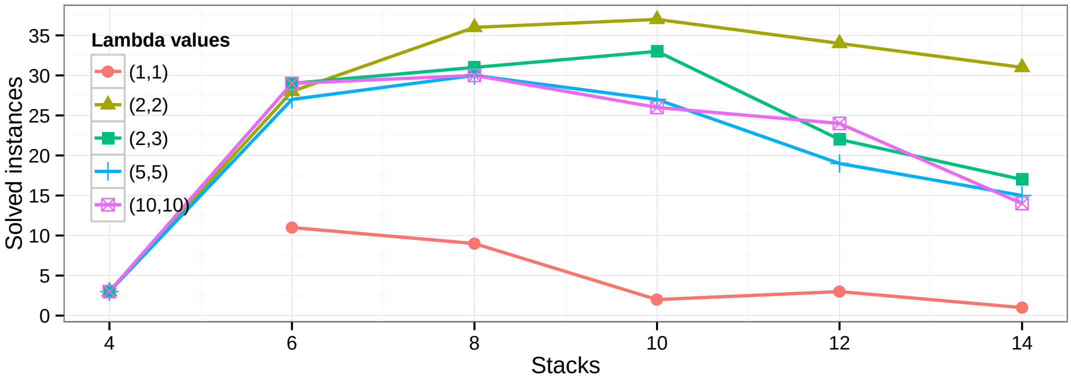


Figure 7.1: Number of solved instances per category for 2D-LPFH lambda test cases. Only values for $q = 0.75$ are shown.

All 2D-LPFH lambda test cases have successfully solved all instances from the $q = 50\%$ categories. The number of solved instances for $q = 75\%$ categories is shown in Figure 7.1. Only 3 instances have been solved for the q75w04 category by all test cases except the greedy test case, $\lambda_{2,3} = 1$ that solved none.

As we will see in further results, this is a common occurrence with all algorithms due to lack of space. If we look at a sample instance of category q75w04 in Figure A.2, we can see that there is only four empty slots available for manipulation of twelve containers. E.g., if 2D-LPFH wants to move a container from second tier of stack s to the second tier of stack s' in an instance

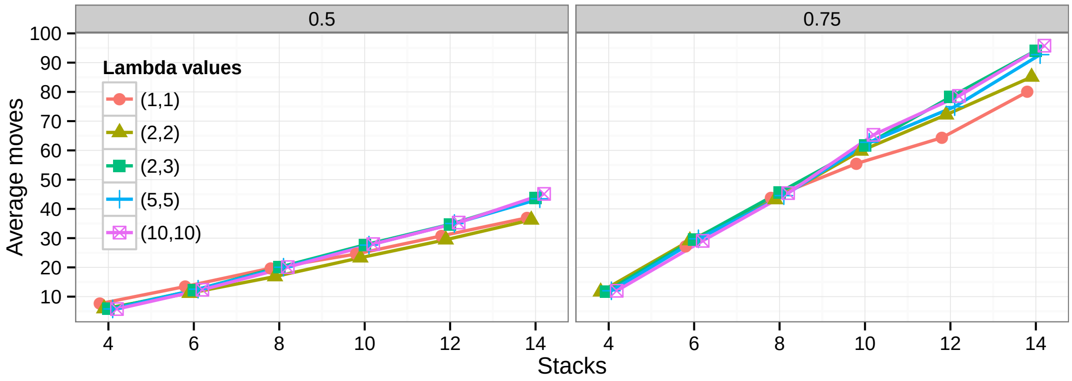


Figure 7.2: Average number of moves per category for 2D-LPFH lambda test cases split by occupancy rate. Values for $q = 0.50$ are shown on the left and $q = 0.75$ on the right.

of category q75w04, it will run out of temporary stacks for interfering containers. Virtual tiers could resolve this problem, but of our goals was to stress-test our algorithm.

Further, the number of solved instances slowly grows for most test cases with the number of stacks, until it reaches instances with more than ten stacks where it slowly declines. The reason for this decline is a very large search space where the algorithm starts making single loops until it exhausts all possible moves and blocks. Tests have shown that the test case for $\lambda_{2,3} = 2$ was able to solve the most instances, 94% of the whole set of instances.

Figure 7.2 shows the average number of moves for all test cases, split by the occupancy rate value. For the $q = 50\%$ categories, we can see two groups slowly splitting towards the higher number of stacks. In the $q = 75\%$ categories we can see that most test cases perform equally well, except for $\lambda_{2,3} = 1$ and $\lambda_{2,3} = 2$ that have a lower move average. The $\lambda_{2,3} = 1$ test case result can be disregarded because of the very low number of solved instances, thereby making the $\lambda_{2,3} = 2$ the best performing test case. Full test results are given in Table 7.7 where we provide the number of solved instances and average number of moves per category with the standard deviation. The standard deviation values are more than 10% of the average number of moves, thereby overlapping with results of other categories. This occurs because the instances within a category are randomly generated and thereby the number of non-well-located containers is also random. To prevent such results we would have to create a specialized instance generator that can fine tune the difficulty level within categories.

Table 7.5 shows the results of the WPRS test over all test cases. The table can be read row by row where a checkmark in each row means that the test case of the current row has a significantly lower number of moves than the test case in the given column. The test case with most checkmarks in a row is the best performing test case, while the test case with most checkmarks in a column is the worst performing test case. Thereby, the previous conclusion that $\lambda_{2,3} = 2$ is the best performing test case is confirmed.

Further, in Table 7.6 and Figure 7.3 we present the average run time in milliseconds. The average run times are between 5 and 50 milliseconds per instance, depending on the category.

a < b	(1,1)	(2,2)	(2,3)	(5,5)	(10,10)
(1,1)		–	✓	✓	✓
(2,2)	✓		✓	✓	✓
(2,3)	–	–		–	✓
(5,5)	–	–	–		✓
(10,10)	–	–	–	–	

Table 7.5: Results of WPRS test over all test cases for the number of moves. A checkmark means that the results for the test case of this row are significantly smaller than the result for the test case of the corresponding column.

The original LPFH has run times within the same interval. The standard deviation reveals the same fact as previously mentioned, since the instances within categories are randomly generated, the number of non-well-placed containers is also random. Thereby the large standard deviation present in the average number of moves is reflected in the run time standard deviation.

In conclusion, the $\lambda_{2,3} = 2$ test case performed best, followed by the $\lambda_{2,3} = 5$ and $\lambda_2 = 2, \lambda_3 = 3$ test cases. These results confirm the same behaviour described in Expósito-Izquierdo et al. (2012). The $\lambda_{2,3} = 1$ test case has once again shown that a greedy approach is not good for the 2D-PMP since it is unable to solve most of the $q = 75\%$ instances.

Extended run time

To provide 2D-LPFH the same conditions as MMAS we run it consecutively for five minutes on each instance and take the best solution. To clarify, this means that instead of running each instance 150 times and taking the best result, we repeatedly solved the same instance for five minutes. While running this experiment, there were no other processes running on the testing machine that could interfere with the run times. This experiment will be executed on the sample set of instances.

We choose the two best performing lambda configurations of 2D-LPFH, $\lambda_{2,3} = 2$ and $\lambda_{2,3} = 5$ for this experiment. One more reason for choosing the $\lambda_{2-3} = 5$ configuration was to see if a bigger search space would perform better with a longer running time. If we take the average of the two average run times for the chosen test cases, we get an average run time of $(44.32 + 52.15)/2 = 48.23$ milliseconds. Further, if we divide five minutes with the average run time, we can calculate that each instance will be run an average of $5 * 60 * 1000/48.23 = 6219$ times. Results are provided in Figures 7.4 and 7.5.

As in the previous experiment, 2D-LPFH solved all $q = 50\%$ instances. Figure 7.4 shows the number of solved instances for the $q = 75\%$ categories compared with the regular time test cases with the same configuration. We can see a noticeable improvement for both test cases. The $\lambda_{2-3} = 2$ test case solved on average five more instances, which is a 20% improvement. The $\lambda_{2-3} = 5$ test case solved almost twice as much instances for $q = 75\%$ and $w > 6$ categories.

Figure 7.5 shows the average number of moves for the two extended run time test cases compared with the same configurations presented in the previous chapter, split by occupancy

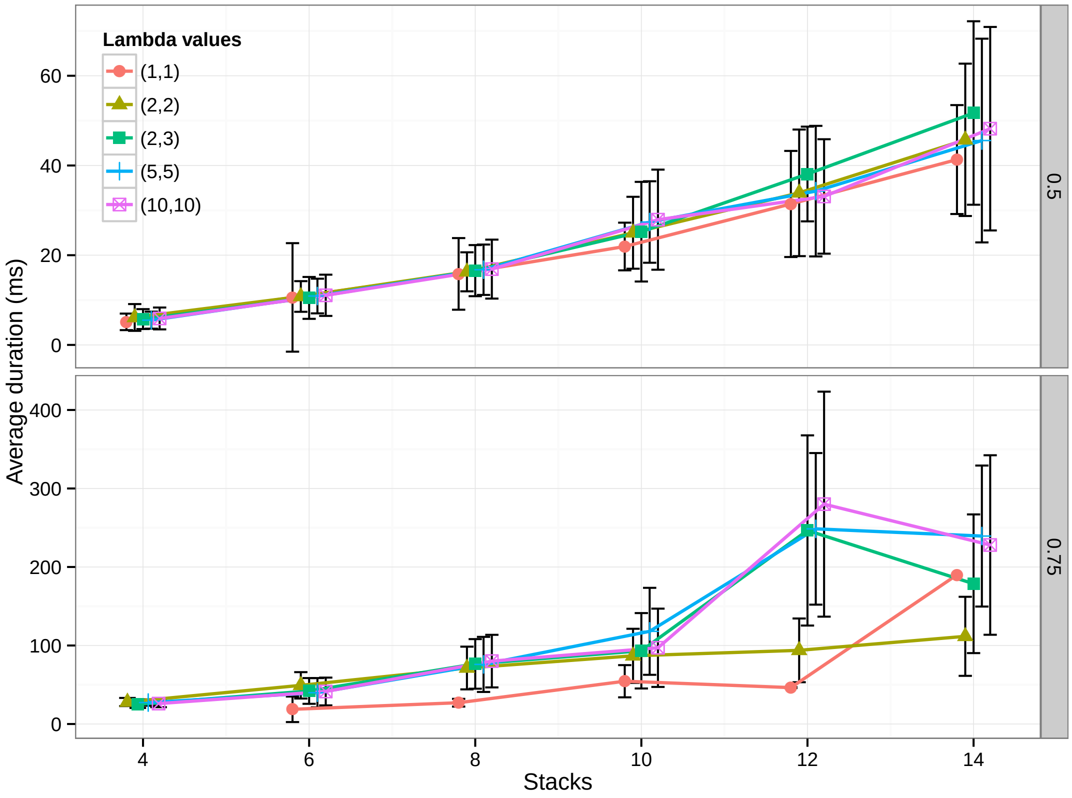


Figure 7.3: Average duration in milliseconds (ms) per category for 2D-LPFH lambda test cases split by occupancy rate. Standard deviation is represented with black bars. Values for $q = 50\%$ are shown above and $q = 75\%$ below.

factor. The extended run time test cases have a significant reduction in the average number of moves with respect to the regular run time test cases. For larger $q = 50\%$ instances there is a five move reduction in average, whereas for the $q = 75\%$ instances there is a five to twelve moves reduction in average. Thereby we may conclude that given a longer run time (more repetitions) the 2D-LPFH can find a better solution.

If we compare only the two extended run time test cases we will see that the $\lambda_{2-3} = 2$ test case performs slightly better and the WPRS test with an alternative hypothesis $(\lambda_{2-3} = 2) < (\lambda_{2-3} = 5)$ confirms this with a p-value lower than $\alpha = 0.05$.

Full results are available in Table 7.8 where we provide the number of solved instances and the average number of moves with the standard deviation.

	Category	Lambda values				
		(1,1)	(2,2)	(2,3)	(5,5)	(10,10)
Average	q50w04	5.14	6.12	5.76	5.52	5.90
	q50w06	10.59	10.80	10.48	10.90	11.06
	q50w08	15.83	16.30	16.56	16.76	16.90
	q50w10	21.94	25.00	25.24	27.40	27.92
	q50w12	31.42	33.90	38.10	34.28	33.10
	q50w14	41.32	45.72	51.70	45.56	48.20
	q75w04	–	28.00	25.00	27.33	26.00
	q75w06	18.73	49.25	42.10	39.85	41.41
	q75w08	27.11	71.31	76.55	75.90	80.10
	q75w10	54.50	86.84	93.27	118.00	97.08
	q75w12	46.33	93.82	246.50	248.60	280.00
	q75w14	190.00	111.70	178.60	239.40	228.00
	Total	22.16	44.32	51.95	52.15	54.63
St.dev.	q50w04	1.84	2.99	2.22	1.90	2.44
	q50w06	12.09	3.42	4.67	3.86	4.60
	q50w08	7.99	4.35	5.70	5.62	6.57
	q50w10	5.31	8.02	11.11	9.09	11.16
	q50w12	11.83	14.11	10.55	14.55	12.75
	q50w14	12.14	16.98	20.45	22.71	22.68
	q75w04	–	5.20	4.58	4.04	4.58
	q75w06	16.24	16.81	16.40	18.66	17.68
	q75w08	4.94	27.20	31.59	35.09	33.50
	q75w10	20.51	34.44	48.02	55.41	49.86
	q75w12	0.58	40.59	121.10	96.54	143.30
	q75w14	–	50.35	88.34	89.81	114.30
	Total	18.24	39.95	68.99	71.58	81.86

Table 7.6: Average run time in milliseconds with standard deviation of 2D-LPFH lambda test cases.

	Category	Lambda values				
		(1,1)	(2,2)	(2,3)	(5,5)	(10,10)
N	q50w04	50	50	50	50	50
	q50w06	49	50	50	50	50
	q50w08	48	50	50	50	50
	q50w10	49	50	50	50	50
	q50w12	50	50	50	50	50
	q50w14	50	50	50	50	50
	q75w04	0	3	3	3	3
	q75w06	11	28	29	27	29
	q75w08	9	36	31	30	30
	q75w10	2	37	33	27	26
	q75w12	3	34	22	19	24
	q75w14	1	31	17	15	14
	Total	322	469	435	421	426
Average	q50w04	7.60	5.92	5.84	5.80	5.76
	q50w06	13.47	11.20	12.16	12.56	12.36
	q50w08	19.73	16.88	20.04	20.06	20.10
	q50w10	24.57	23.30	27.68	27.62	27.96
	q50w12	30.78	29.46	34.70	35.00	35.32
	q50w14	36.94	36.24	43.68	43.46	45.12
	q75w04	–	11.67	11.67	12.00	12.00
	q75w06	27.18	29.18	29.55	29.78	29.07
	q75w08	43.78	43.19	45.65	44.53	45.40
	q75w10	55.50	59.81	61.70	62.78	65.31
	q75w12	64.33	72.18	78.23	74.95	78.62
	q75w14	80.00	85.10	94.12	92.73	95.79
	Total	23.77	33.82	34.18	33.04	34.03
St.dev.	q50w04	2.900	1.861	1.777	1.738	1.744
	q50w06	4.184	2.955	3.310	3.315	3.391
	q50w08	3.541	2.512	2.927	3.047	3.442
	q50w10	3.385	3.046	3.706	3.669	3.870
	q50w12	3.688	3.309	4.062	3.681	3.888
	q50w14	4.017	2.911	4.028	3.840	3.526
	q75w04	–	1.528	1.528	2.000	2.000
	q75w06	6.431	4.839	6.294	6.116	5.732
	q75w08	3.898	5.497	6.509	6.196	5.963
	q75w10	7.778	5.849	6.410	6.053	6.565
	q75w12	1.155	8.314	11.380	6.294	7.058
	q75w14	–	8.837	10.100	7.146	10.610
	Total	12.29	23.45	22.89	21.69	22.90

Table 7.7: Number of solved instances and average number of moves with standard deviation for 2D-LPFH lambda test cases.

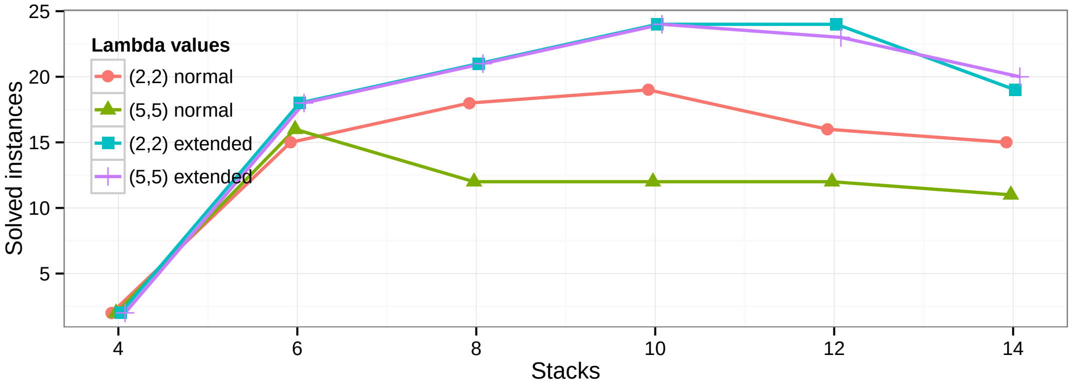


Figure 7.4: Number of solved instances per category for 2D-LPFH extended run time experiment. only values for $q = 0.75$ are shown.

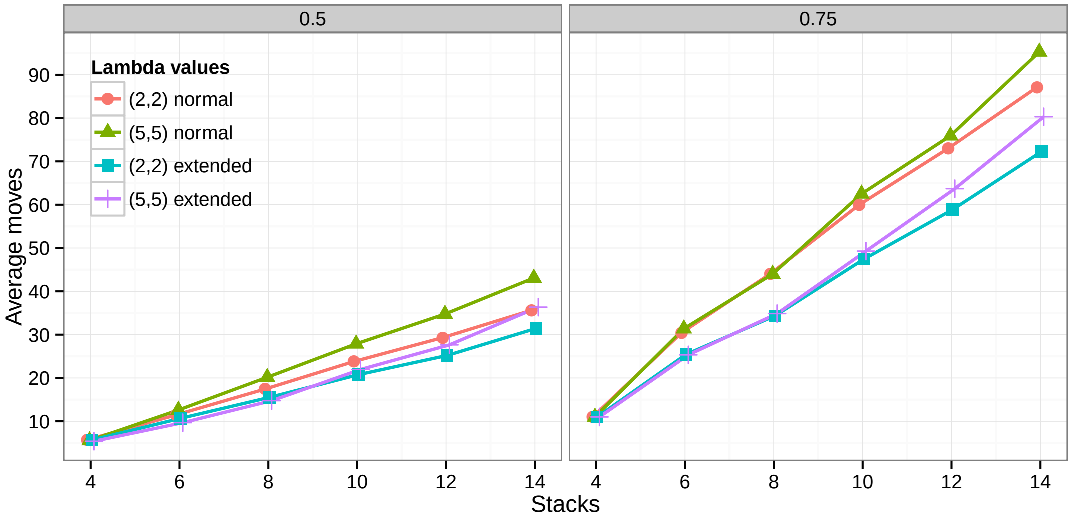


Figure 7.5: Average number of moves per category for 2D-LPFH extended run time experiment split by occupancy rate. Values for $q = 0.50$ are shown on the left and $q = 0.75$ on the right.

	Categories	N		Average		St.dev.	
(λ_2, λ_3)		(2,2)	(5,5)	(2,2)	(5,5)	(2,2)	(5,5)
	q50w04	25	25	5.68	5.400	1.773	1.472
	q50w06	25	25	10.68	9.680	2.462	1.994
	q50w08	25	25	15.52	14.76	2.023	2.278
	q50w10	25	25	20.76	21.96	2.067	2.937
	q50w12	25	25	25.20	27.64	1.658	2.706
	q50w14	25	25	31.44	36.36	2.329	2.691
	q75w04	2	2	11.00	11.00	1.414	1.414
	q75w06	18	18	25.44	25.33	5.147	5.179
	q75w08	21	21	34.33	34.86	3.307	3.705
	q75w10	24	24	47.42	49.29	6.775	5.607
	q75w12	24	23	58.88	63.70	5.590	5.182
	q75w14	19	20	72.26	80.30	4.280	4.791
	Total	258	258	30.45	32.40	19.55	21.89

Table 7.8: Number of solved instances and average number of moves with standard deviation for 2D-LPFH extended run time test cases.

7.6 PILOT experiments

The PILOT method has two important parameters we can manipulate: the used sub-heuristic and the number of lookahead steps. We run the PILOT method experiments with LPFH, greedy and random construction heuristics as sub-heuristics and $k \in \{2, 4, 5, 6\}$ construction steps for each heuristic. The 2D-LPFH sub-heuristic test cases were extended to include more lookahead steps $k^+ \in \{1, 2, 3, 4, 5, 6, 7\}$.

Nr.	Heuristic	k	ψ
1-4	Greedy	$\{2,4,5,6\}$	150
5-8	Random	$\{2,4,5,6\}$	150
9-15	2D-LPFH	$\{1,2,3,4,5,6,7\}$	150

Table 7.9: List of PILOT experiments. The table shows the parameters used in individual experiments.

Table 7.9 lists all 15 PILOT test cases. The second column specifies the used sub-heuristic and the third column specifies the lookahead steps. These test cases were executed on the whole set of instances.

Random and Greedy sub-heuristics

Using random and greedy construction heuristics as sub-heuristic of PILOT method yielded similar results to the previous random and greedy construction heuristic results (Section 7.4).

Algorithm	k	Category	N	Average	St.dev.	Duration	Dur.st.dev.
Greedy	2	q50w04	2	3.50	0.71	3.00	0.02
	4	q50w04	1	3.00		3.02	–
	5	q50w04	1	3.00		3.70	–
	6	q50w04	1	3.00		5.20	–
Random	2	q50w04	15	13.47	11.69	2.12	0.40
	4	q50w04	21	12.48	7.53	3.00	0.72
		q50w06	1	4.00		1.70	–
		q75w04	1	21.00		12.00	–
	5	q50w04	16	12.88	14.08	3.08	0.92
		q50w06	1	3.00		3.62	–
	6	q50w04	21	14.62	14.46	4.21	0.50
		q50w06	1	3.00		4.17	–

Table 7.10: Results for PILOT method using random and greedy construction heuristics as sub-heuristics. Duration is expressed in milliseconds.

PILOT method was able to solve two q50w04 instances using two lookahead moves and one instance using $k \in \{4, 5, 6\}$ lookahead moves. The random construction heuristic performed slightly better, solving an average of 19 instances per test case, out of 600. Once again, the random and greedy construction heuristics have shown to be unusable for solving the 2D-PMP. Table 7.10 offers the results.

2D-LPFH sub-heuristic

For this experiment we choose the best performing 2D-LPFH configuration, $\lambda_{2,3} = 2$ as PILOT method's sub-heuristic for $k \in \{1, 2, 3, 4, 5, 6, 7\}$. As described in Section 5.1, 2D-LPFH can perform a different set of moves than the greedy and random construction heuristic and it performs compound moves. Therefore PILOT method explores only the set of compound moves $\overline{\mathcal{M}}$ provided by the algorithm.

In Table 7.12 we provide the number of solved instances and average number of moves per category with the standard deviation. PILOT method was able to solve all $q = 50\%$ instances using 2D-LPFH as sub-heuristic. For the $q = 75\%$ categories, the PILOT method was unable to solve all instances, but all test cases perform similarly. The average number of moves reveals a slightly difference between all approaches, but does not provide substantial evidence to make a decision on the best test case. Therefore we use the WPRS test and the results are provided in Table 7.11. The results tell us that the $k = \{5, 6, 7\}$ test cases perform better than the remaining test cases, but equally well compared to each other.

a < b	1	2	3	4	5	6	7
1		–	–	–	–	–	–
2	–		–	–	–	–	–
3	–	–		–	–	–	–
4	–	–	–		–	–	–
5	✓	✓	✓	✓		–	–
6	✓	✓	✓	✓	–		–
7	✓	✓	✓	✓	–	–	

Table 7.11: Results for WPRS test with alternative hypothesis $a < b$ over PILOT 2D-LPFH test cases results. The checkmark marks pairs where the alternative hypothesis holds.

To make a final decision on the best test case, we compare the average number of moves for $k = \{5, 6, 7\}$ on a per category basis. Based on this observation, the $k = 7$ can be selected as the best performing test case with an insignificant difference towards $k = \{5, 6\}$.

	Category	Lookahead moves						
		1	2	3	4	5	6	7
N	q50w04	50	50	50	50	50	50	50
	q50w06	50	50	50	50	50	50	50
	q50w08	50	50	50	50	50	50	50
	q50w10	50	50	50	50	50	50	50
	q50w12	50	50	50	50	50	50	50
	q50w14	50	50	50	50	50	50	50
	q75w04	2	3	3	3	3	3	3
	q75w06	31	32	32	32	30	30	32
	q75w08	39	40	40	41	41	38	38
	q75w10	41	41	39	41	40	42	43
	q75w12	37	40	35	39	41	40	43
	q75w14	31	34	36	38	36	38	38
	Total	481	490	485	494	491	491	497
Average	q50w04	5.86	5.74	5.74	5.84	5.96	6.26	6.44
	q50w06	11.48	11.18	11.20	10.94	10.88	11.02	11.04
	q50w08	17.26	17.14	17.18	17.32	17.06	17.10	16.94
	q50w10	23.56	23.70	23.40	23.62	23.20	23.30	22.94
	q50w12	29.82	29.88	29.46	29.54	29.74	29.20	29.20
	q50w14	36.90	37.30	37.26	37.08	37.18	37.14	37.26
	q75w04	11.00	11.67	11.67	11.67	11.67	11.67	11.67
	q75w06	26.94	26.91	27.81	26.84	26.23	26.87	27.28
	q75w08	39.69	39.33	40.05	40.29	39.98	39.13	39.21
	q75w10	55.07	54.54	55.08	54.83	53.95	54.21	53.79
	q75w12	67.92	69.00	68.46	69.21	67.98	67.78	68.23
	q75w14	84.10	82.03	82.53	83.18	80.75	81.97	79.16
	Total	33.32	33.68	33.51	34.15	33.63	33.87	33.89
St.dev.	q50w04	1.830	1.688	1.601	1.683	1.761	1.925	2.168
	q50w06	2.915	3.015	2.928	2.729	2.700	2.615	2.523
	q50w08	3.036	3.097	2.869	3.054	2.637	2.697	2.645
	q50w10	3.494	3.644	3.435	3.162	3.356	3.157	3.040
	q50w12	3.778	4.163	4.001	3.913	3.590	3.557	3.676
	q50w14	3.352	3.138	3.250	3.002	2.731	2.935	3.573
	q75w04	1.414	1.528	1.528	1.528	1.528	1.528	1.528
	q75w06	5.144	4.914	5.361	5.106	4.523	5.111	5.163
	q75w08	5.105	4.509	5.657	6.266	5.812	5.041	5.292
	q75w10	6.101	5.608	4.568	5.049	4.830	4.922	5.910
	q75w12	6.508	6.349	7.663	8.430	6.944	8.141	8.026
	q75w14	7.960	7.412	9.188	7.326	7.897	6.934	6.109
	Total	22.24	22.38	22.53	22.99	22.30	22.60	22.24

Table 7.12: Number of solved instances and average number of moves with standard deviation for PILOT method with 2D-LPFH sub-heuristic test cases.

7.7 MMAS experiments

ACO metaheuristic has multiple important parameters: n, α, β and ρ (all except n are pheromone model related parameters). Additionally we can select an implementation of the pheromone model and heuristic function. MMAS introduces $\tau_{\min}$, $\tau_{\max}$, p_{σ^*} and $i_{\max}$. As a starting point, we used recommended values from Dorigo and Stützle (2004) (Box 3.1 on page 71 and Box 3.3 on page 96) and decided to use the *move based pheromone model*. During preliminary testing we found the offered values to produce good results.

We test the following aspects of MMAS: n, α, β and the choice of heuristic function. All other parameters are set to the following values, which are taken from Dorigo and Stützle (2004): $\rho = 0.02$, $i_{\max} = 75$ and $p_{\sigma^*} = 0.05$. Table 7.13 lists all test cases for MMAS. The test cases are divided into three aspects: ant count, pheromone and heuristic values and heuristic function.

Nr.	Experiment aspect	n	α	β	Heuristic function
1	ant count	2	1	2	2D-LPFH
2	ant count	4	1	2	2D-LPFH
3	ant count	6	1	2	2D-LPFH
4	ant count	8	1	2	2D-LPFH
4	α and β values	8	1	2	2D-LPFH
5	α and β values	8	1	6	2D-LPFH
6	α and β values	8	2	1	2D-LPFH
7	α and β values	8	0	1	2D-LPFH
8	heuristic function	8	1	2	2D-LPFH
9	heuristic function	8	1	2	PILOT 2D-LPFH

Table 7.13: List of MMAS test cases and their parameters.

These test cases were executed on the sample set of instances. We use the set of parameters that performed best during preliminary tests in place of variables whose values will be explored in future tests. For (α, β) we use values $(1, 2)$ and for the heuristic function we use 2D-LPFH. The best test case is later run again on the whole set of instances to create a benchmark for comparison with other test cases.

Ant count

The number of ants can strongly influence the result quality. One reason is that more ants allow more exploration and quicker discovery of better results. We have shown in previous test cases that more runs of 2D-LPFH significantly boost performance, therefore we assume more ants will yield better results. For other versions of ACO, the number of ants influences the pheromone values as well because more than one ant can deposit pheromone values. In our case only the best ant deposits pheromone values.

We define 4 test cases with $n \in \{2, 4, 6, 8\}$. Detailed results are provided in Table 7.16. For the $q = 50\%$ instances, all test cases solved all instances, and for $q = 75\%$ all instances performed almost equally well, solving 62 out of 100 instances. Only test case $n = 6$ solved 63 instances. If we analyse the average number of moves we will face a similar situation with minimal difference between the average number of moves per category over all test cases. However, we notice that the average numbers of moves is slowly declining with increasing number of ants. To prove this trend we perform a series of statistical tests. Results of the WPRS test are provided in Table 7.14 and tell us that test cases $n' = \{6, 8\}$ performed significantly better than $n'' = \{2, 4\}$, but there is no significant difference between test cases for six and eight ants.

a < b	2	4	6	8
2		–	–	–
4	–		–	–
6	✓	✓		–
8	✓	✓	–	

Table 7.14: Results for WPRS test with alternative hypothesis $a < b$ over MMAS ant count test cases results. The checkmark marks pairs where the alternative hypothesis holds.

To make a final decision on the best performing test case, we once again analyse the average number of moves for n' test cases. We can see that the test case with eight ants in comparison with six ants has almost consistently a smaller or equal average number of moves in all instance categories and over all instances. Thereby we choose $n^* = 8$ as the best performing test case.

The reason for all test cases to perform so similarly can be found in the behaviour of 2D-LPFH and the fact that MMAS only considers compound moves the heuristic function returns as possible which reduces the search space in each step significantly. Each possible compound move returned by 2D-LPFH at any given step reduces the number of non-well-located containers by one. Thereby, even random selection of moves at every step will lead to a valid final solution. For this reason we can only notice minimal differences between different configurations. On an exceptional basis, the algorithm might encounter a step that is not a valid final solution and has no possible moves. This kind of behaviour requires us to find the best possible algorithm parameters and take even small insignificant improvements into account.

Alpha and beta values

α and β values are used in the ant construction algorithm to decide how much influence do the pheromone and heuristic values have in choosing the next step, respectively. The recommended value for (α, β) in Dorigo and Stützle (2004) is $(1, 2)$. We test further values to see if and how much influence the pheromone model has on the result quality and whether more heuristic influence will yield a better result. We test the MMAS for the following set of values of $(\alpha, \beta) \in \{(1, 2), (1, 6), (0, 1), (2, 1)\}$.

Results are provided in Table 7.17. As in the previous experiment, all $q = 50\%$ instances are solved in all test cases, while all test cases solved almost the same amount of $q = 75\%$ instances, approximately 63 instances. We move move further to the average number of moves. Again, we can notice only minimal difference in the average number of moves per category for the same reason as described in the previous experiment.

To narrow the decision for the best performing test case, we use statistical tests. Results of a series of WPRS tests are provided in Table 7.15. The results indicate that the test cases with $\alpha = 1$ and $\beta \in \{2, 6\}$ yield s significantly smaller number of moves than the remaining test cases. Again, to make a final decision we again analyse the average number of moves and notice that the average number of moves over all instances is lower for test case $(1, 2)$, thereby making it the best performing test case. Additionally, we choose a lower β value to preserve a higher, yet moderate influence of pheromone values. Tests have shown that configurations where pheromones have a moderate influence perform significantly better than those where pheromones have none or too much influence in choosing the next move.

a < b	(1,2)	(1,6)	(2,1)	(0,1)
(1,2)		–	✓	✓
(1,6)	–		✓	✓
(2,1)	–	–		–
(0,1)	–	–	–	

Table 7.15: Results for WPRS test with alternative hypothesis $a < b$ over MMAS pheromone and heuristic values test cases results. The checkmark marks pairs where the alternative hypothesis holds.

Heuristic algorithms

Given that only 2D-LPFH was able to solve a reasonable number of instances, we have a limited choice of heuristic algorithms. We choose the best performing test cases from 2D-LPFH, $\lambda_{2,3} = 2$ and PILOT method with 2D-LPFH as sub-heuristic and $k = 7$ lookahead moves.

After running the both test cases on a sample set of instances, we were not able to distinguish almost any difference between the results of the two approaches. Therefore we run the test on the whole set of instances. Test results are provided in Table 7.18. Both test cases solved 511 instances and have an average number of 29.4 moves. Unfortunately, even after running tests on the whole set of instances both approaches performed so closely, that we are not able to distinguish a big enough difference to make a decision on the better test case. Therefore we decide to choose the simpler approach, without PILOT method as the better approach.

	Category	Ant count			
		2	4	6	8
N	q50w04	25	25	25	25
	q50w06	25	25	25	25
	q50w08	25	25	25	25
	q50w10	25	25	25	25
	q75w04	2	2	2	2
	q75w06	18	18	18	18
	q75w08	19	19	20	19
	q75w10	23	23	23	23
	Total	162	162	163	162
Average	q50w04	5.40	5.40	5.40	5.40
	q50w06	10.44	10.44	10.44	10.44
	q50w08	15.28	15.20	15.16	15.24
	q50w10	20.68	20.80	20.64	20.48
	q75w04	11.00	11.00	11.00	11.00
	q75w06	25.33	25.28	25.28	25.28
	q75w08	33.16	32.63	33.35	32.37
	q75w10	44.35	44.43	44.09	43.91
	Total	21.13	21.08	21.16	20.93
St.dev.	q50w04	1.472	1.472	1.472	1.472
	q50w06	2.382	2.382	2.382	2.382
	q50w08	2.189	2.160	2.154	2.185
	q50w10	2.340	2.500	2.307	2.257
	q75w04	1.414	1.414	1.414	1.414
	q75w06	5.145	5.120	5.120	5.120
	q75w08	3.184	2.650	5.163	2.852
	q75w10	4.355	4.409	4.111	4.220
	Total	12.96	12.93	12.99	12.75

Table 7.16: Number of solved instances and average number of moves with standard deviation for MMAS ant count test cases.

	Category	(α, β)			
		(1,6)	(1,2)	(2,1)	(0,1)
N	q50w04	25	25	25	25
	q50w06	25	25	25	25
	q50w08	25	25	25	25
	q50w10	25	25	25	25
	q75w04	2	2	2	2
	q75w06	18	18	18	18
	q75w08	20	19	20	19
	q75w10	23	23	23	23
	Total	163	162	163	162
Average	q50w04	5.40	5.40	5.40	5.40
	q50w06	10.40	10.44	10.44	10.40
	q50w08	15.16	15.24	15.16	15.32
	q50w10	20.48	20.48	20.92	20.80
	q75w04	11.00	11.00	11.00	11.00
	q75w06	25.22	25.28	25.33	25.28
	q75w08	33.60	32.37	34.10	32.58
	q75w10	44.04	43.91	44.74	44.43
	Total	21.15	20.93	21.39	21.09
St.dev.	q50w04	1.472	1.472	1.472	1.472
	q50w06	2.380	2.382	2.382	2.380
	q50w08	2.154	2.185	2.154	2.340
	q50w10	2.365	2.257	2.532	2.466
	q75w04	1.414	1.414	1.414	1.414
	q75w06	5.151	5.120	5.145	5.120
	q75w08	5.807	2.852	7.305	2.912
	q75w10	4.073	4.220	4.126	3.918
	Total	13.04	12.75	13.36	12.90

Table 7.17: Number of solved instances and average number of moves with standard deviation for MMAS pheromone and heuristic values test cases.

	Category	MMAS heuristic function	
		2D-LPFH	PILOT 2D-LPFH
N	q50w04	50	50
	q50w06	50	50
	q50w08	50	50
	q50w10	50	50
	q50w12	50	50
	q50w14	50	50
	q75w04	3	3
	q75w06	34	33
	q75w08	42	43
	q75w10	44	44
	q75w12	46	46
	q75w14	42	42
	Total	511	511
Average	q50w04	5.68	5.68
	q50w06	10.36	10.34
	q50w08	14.86	14.82
	q50w10	20.08	20.12
	q50w12	25.54	25.44
	q50w14	31.74	32.50
	q75w04	11.67	11.67
	q75w06	24.65	24.24
	q75w08	32.36	32.19
	q75w10	44.57	44.36
	q75w12	55.93	55.41
	q75w14	68.57	68.17
	Total	29.47	29.41
St.dev.	q50w04	1.634	1.634
	q50w06	2.371	2.370
	q50w08	2.232	2.210
	q50w10	2.311	2.327
	q50w12	2.667	2.589
	q50w14	2.732	2.866
	q75w04	1.528	1.528
	q75w06	4.478	4.154
	q75w08	3.275	3.923
	q75w10	4.217	3.912
	q75w12	4.763	4.750
	q75w14	5.735	4.504
	Total	18.63	18.48

Table 7.18: Number of solved instances and average number of moves with standard deviation for MMAS heuristic function test cases.

7.8 Comparative analysis of best configurations

In this section we compare the four best approaches from previous experiments showing a progressive improvement in results. From the 2D-LPFH experiments we choose the best performing lambda test case, $\lambda_{2,3} = 2$ and the same configuration in the extended run time experiment. From the PILOT experiments we choose the 2D-LPFH sub-heuristic with $k = 7$ lookahead moves. Finally from the MMAS experiments we choose the $n = 8, \alpha = 1, \beta = 2$ parameters with 2D-LPFH as the heuristic function. All chosen test cases are executed over the whole set of instances.

All chosen test cases solved all $q = 50\%$ instances. Figure 7.6 shows the number of solved instances per test case for $q = 75\%$ categories. We notice that the MMAS and extended run time 2D-LPFH solved almost the same number of instances with the extended run time 2D-LPFH in a slight lead. They are followed by the PILOT method test case and 2D-LPFH respectively.

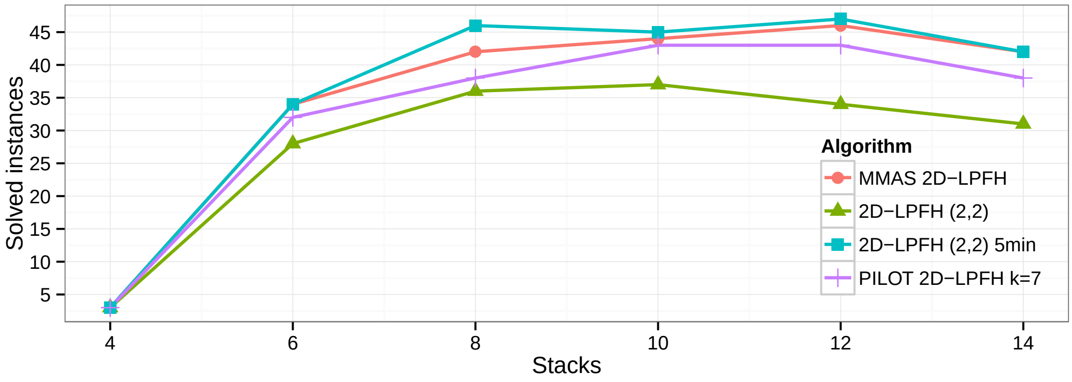


Figure 7.6: Number of solved instances per category for chosen best test cases. Only values for $q = 0.75$ categories are shown.

Figure 7.7 shows the average number of moves. For the $q = 50\%$ categories there is an overlapping for the MMAS and extended run time 2D-LPFH test case that perform better than the 2D-LPFH and PILOT method test case that also overlap. The average number of moves for the $q = 75\%$ categories reveals a similar ordering between test cases as in the number of solved instances with an exception of the MMAS test case performing better than the extended run time 2D-LPFH test case.

Since the chosen test cases have very different run times (milliseconds vs minutes vs fixed run time), we are unable to perform comparisons with respect to run time. Table 7.20 offers a full overview of the number of solved instances and average number of moves with the standard deviation. Table 7.21 offers an overview of the runtimes expressed in milliseconds. Using only the average number of moves and number of solved instances we can clearly separate MMAS and extended run time 2D-LPFH as the overall best performing test cases. Using WPRS test we provide further evidence for this conclusion. Table 7.19 shows the result of a series of statistical tests. The results clearly point out MMAS as significantly better than any other test

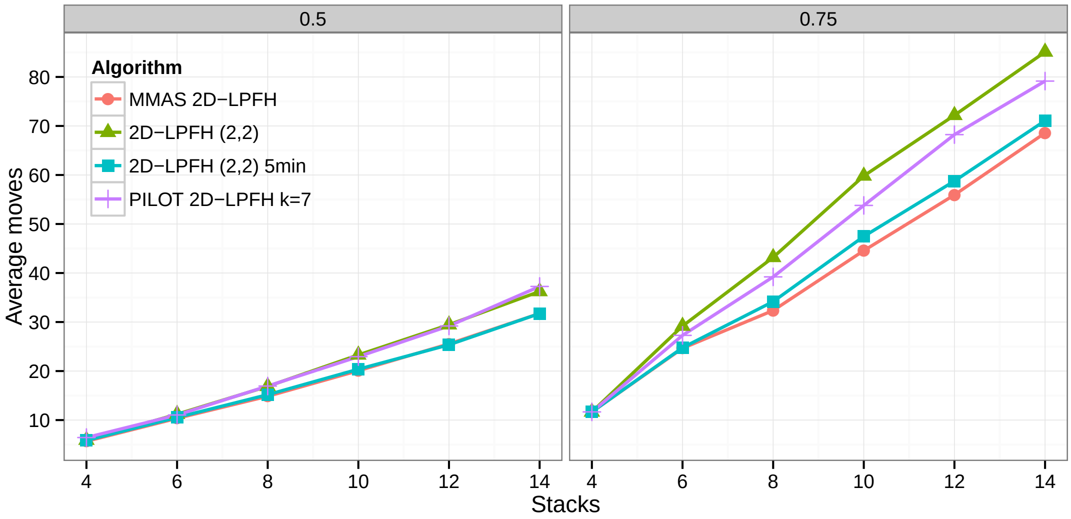


Figure 7.7: Average number of moves per category for best algorithm configurations split by occupancy rate. Values for $q = 0.50$ are shown on the left and $q = 0.75$ on the right.

case. Following MMAS is the extended run time 2D-LPFH that performs better than PILOT method using 2D-LPFH as sub-heuristic and 2D-LPFH, respectively.

a < b	MMAS 2D-LPFH	2D-LPFH 5min	PILOT 2D-LPFH	2D-LPFH
MMAS 2D-LPFH		✓	✓	✓
2D-LPFH 5min	–		✓	✓
PILOT 2D-LPFH	–	–		✓
2D-LPFH	–	–	–	

Table 7.19: Results for Wilcoxon paired rank sum test with alternative hypothesis $a < b$ over four best test cases results. The checkmark marks pairs where the alternative hypothesis holds.

	Category	MMAS 2D-LPFH	2D-LPFH 5min	PILOT 2D-LPFH	2D-LPFH
N	q50w04	50	50	50	50
	q50w06	50	50	50	50
	q50w08	50	50	50	50
	q50w10	50	50	50	50
	q50w12	50	50	50	50
	q50w14	50	50	50	50
	q75w04	3	3	3	3
	q75w06	34	34	32	28
	q75w08	42	46	38	36
	q75w10	44	45	43	37
	q75w12	46	47	43	34
	q75w14	42	42	38	31
	Total	511	517	497	469
Average	q50w04	5.68	5.86	6.440	5.92
	q50w06	10.36	10.54	11.04	11.20
	q50w08	14.86	15.18	16.94	16.88
	q50w10	20.08	20.38	22.94	23.30
	q50w12	25.54	25.36	29.20	29.46
	q50w14	31.74	31.74	37.26	36.24
	q75w04	11.67	11.67	11.67	11.67
	q75w06	24.65	24.76	27.28	29.18
	q75w08	32.36	34.17	39.21	43.19
	q75w10	44.57	47.47	53.79	59.81
	q75w12	55.93	58.77	68.23	72.18
	q75w14	68.57	71.07	79.16	85.10
	Total	29.47	30.53	33.89	33.82
St.dev.	q50w04	1.634	1.784	2.168	1.861
	q50w06	2.371	2.367	2.523	2.955
	q50w08	2.232	2.067	2.645	2.512
	q50w10	2.311	2.221	3.040	3.046
	q50w12	2.667	2.562	3.676	3.309
	q50w14	2.732	2.465	3.573	2.911
	q75w04	1.528	1.528	1.528	1.528
	q75w06	4.478	4.513	5.163	4.839
	q75w08	3.275	3.946	5.292	5.497
	q75w10	4.217	5.358	5.910	5.849
	q75w12	4.763	5.704	8.026	8.314
	q75w14	5.735	4.729	6.109	8.837
	Total	18.63	19.57	22.24	23.45

Table 7.20: Number of solved instances and average number of moves with standard deviation for selected best test cases.

	Category	MMAS 2D-LPFH	2D-LPFH 5min	PILOT 2D-LPFH	2D-LPFH
Average	q50w04	14.9×10^3	6	300.0×10^3	16
	q50w06	65.0×10^3	11	299.9×10^3	63
	q50w08	97.9×10^3	16	298.4×10^3	142
	q50w10	219.9×10^3	25	281.6×10^3	267
	q50w12	360.9×10^3	34	254.7×10^3	413
	q50w14	624.7×10^3	46	231.9×10^3	652
	q75w04	50.0×10^3	28	300.0×10^3	25
	q75w06	643.8×10^3	49	265.6×10^3	110
	q75w08	796.0×10^3	71	212.4×10^3	202
	q75w10	1147.0×10^3	87	166.7×10^3	315
	q75w12	1210.2×10^3	94	163.7×10^3	451
	q75w14	1211.9×10^3	112	166.0×10^3	546
	Total	551.2×10^3	44	242.1×10^3	287
St.dev.	q50w04	4.5×10^3	3	0.0×10^3	7
	q50w06	25.0×10^3	3	0.5×10^3	19
	q50w08	24.7×10^3	4	3.5×10^3	31
	q50w10	46.0×10^3	8	37.6×10^3	38
	q50w12	59.0×10^3	14	70.7×10^3	78
	q50w14	109.9×10^3	17	78.2×10^3	86
	q75w04	17.0×10^3	5	0.0×10^3	9
	q75w06	585.1×10^3	17	53.1×10^3	31
	q75w08	347.1×10^3	27	64.5×10^3	41
	q75w10	123.7×10^3	34	88.8×10^3	79
	q75w12	9.3×10^3	41	87.0×10^3	197
	q75w14	7.9×10^3	50	89.2×10^3	201
	Total	483.8×10^3	40	80.7×10^3	221

Table 7.21: Average runtime in milliseconds of solved instances with standard deviation for selected best test cases.

CHAPTER 8

Conclusion

This chapters provides a short overview of achieved goals, after which we sum up our conclusions from the previous chapters and make suggestions for future work. Finally, in the last section we describe a proposed local search method that attempts to create a network of solutions for one instance and perform local search by finding the shortest path within the built network.

8.1 Critical reflection

In this work, we define a new extension of the classical PMP, discuss its computational complexity and then we approximately solve it by successfully adapting LPFH, a PMP strategic heuristic, whose results are further improved by embedding it with PILOT method and MMAS. We also show that naive algorithms are able to solve only the simplest category of instances within a reasonable amount of time and moves. All this is supported by a series of experiments, iteratively improving algorithm performance.

The main challenge of this work was to implement a working heuristic for 2D-PMP that would then be improved using the PILOT method and MMAS. Implementing 2D-LPFH was not the hardest task. After finding the main differences between the classical PMP and 2D-PMP and establishing *models of adequate stack assignment*, the remaining work on 2D-LPFH was easy. The next big challenge was representing 2D-PMP as a path construction problem in order to apply MMAS to it, along with finding a working pheromone model. These problems were all successfully solved.

The final and biggest challenge was using 2D-LPFH as a sub-heuristic for PILOT method and heuristic for MMAS. In the first version we ignore 2D-LPFH's strategy and choose only the first move from a series of planned moves made by 2D-LPFH, thereby yielding a series of unnecessary moves and unexpectedly bad results. After careful reconsideration we reimplement the PILOT method and MMAS to respect 2D-LPFH's strategy and accept compound moves. This has proven to be a working fix for the described problem. However, it introduces a new problem: a too limited search space that allows to only perform moves that the 2D-LPFH strategy allows. We do not know for certain that this subset of the search space includes the optimal solution and thereby we could be facing a local optimum we cannot escape without changing the underlying heuristic.

Finally, we achieved a significant performance improvement with respect to the initial 2D-LPFH results, showing that 2D-LPFH still has room for improvement. Unfortunately, the optimum solutions are unknown for our set of instances and problem.

2D-LPFH lambda test cases prove a greedy 2D-LPFH configuration narrows the heuristic's search space too much and does not allow it to find a valid final solution. The PILOT method shows it can successfully navigate through a bigger search space with a *"tempered-greedy"* approach. MMAS introduces a new element, joint knowledge of multiple construction heuristics that has shown most successful in "guiding" 2D-LPFH to construct better results. These same results are available in 2D-LPFH's search space, but could not be reached, even after 5 minutes of consecutive re-reruns on each instance.

In conclusion, the pure 2D-LPFH should be used for $q = 50\%$ with a run time that is in relation to the size of the instance. The exact relation is a topic for future work. However, for the $q = 75\%$ instances, even though the extended runtime 2D-LPFH solved more instances than MMAS, MMAS always had a lower number of moves and should therefore be used for $q = 75\%$ instances. From the runtime perspective, 2D-LPFH will most probably reach a better solution before MMAS and improve it faster, but MMAS will find a better solution given a longer runtime. In our experiments we limited both run times to five minutes and discovered that this time limit was enough for the MMAS to outperform 2D-LPFH. For lower time limits, we could observe the 2D-LPFH performing better than MMAS.

Unfortunately, the PILOT method could not measure up to MMAS and extended runtime 2D-LPFH. The PILOT method is only good enough if we are looking for a slight performance improvement compared to the 2D-LPFH without a significant runtime increase. As the two naive approaches were not able to solve most of the instances, we can not recommend them for any size of instance.

8.2 Future work

The first task should be implementing a 2D-PMP instance generator that allows fine tuning of categories. One way is to use the same category fine tuning methods described in Expósito-Izquierdo et al. (2012), applied to 2D-PMP, or to define new methods, e.g. number of non-well-placed containers.

The implemented heuristic, 2D-LPFH performs well. Following only its compound moves, allows us to find a small subset of available solutions. A heuristic that finds lower quality solutions and does not limit the search space as much, leaves room for improvement by, e.g. using a sophisticated local search algorithm or metaheuristic. On the other hand, an exact algorithm like A^* would be a interesting tool for benchmarking current and future implementations, but it requires an exact heuristic, which could be hard to prove, given the additional horizontal constraints introduced in 2D-PMP.

MMAS should be compared against another metaheuristic, e.g. simulated annealing or the corridor method (Caserta and Voß, 2009).

The binary encoding described in Caserta et al. (2009) could be used to make quick calculations on a matrix and perhaps easily reveal implicit information about states that could be used as heuristic information or a local search.

8.3 Solution Network Analysis Procedure

The Solution Network Analysis Procedure (SNAP) combines all discovered solutions for a single instance into a network. Since a solution can be represented as a path, as described in section 5.3, we can overlay all known solutions into a network. All equivalent states are represented with a unique node and edges between those nodes represent moves performed in all of the overlayed solutions.

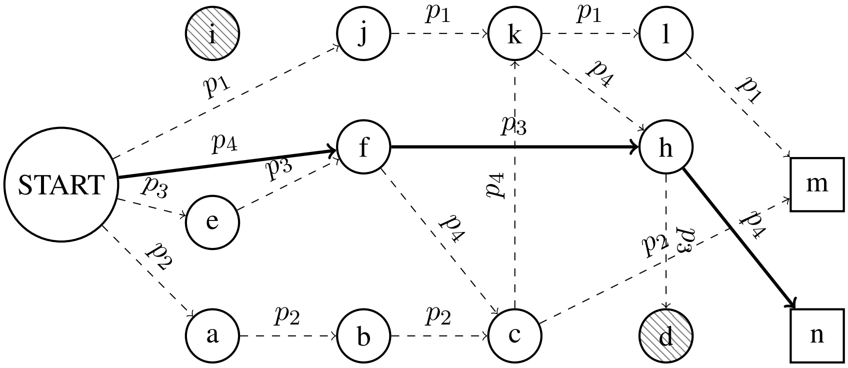


Figure 8.1: A network of solutions.

This network has only one origin and destination. The shortest path between these two points is the best solution with the given knowledge. Each time we want to perform a local search within our network, we run a shortest path algorithm like Dijkstra. Each edge is assigned a cost of one, and all final nodes are connected into one ultimate final node with a cost of zero. If the new path is unknown and shorter than any known path, that means we found a better solution than any of the known solutions.

Figure 8.1 shows a depiction of a solution network. All solutions' paths are marked with s_i and the shortest path is marked with thick lines.

Due to time limitations, we were not able to implement this algorithm, but we propose it as a good addition to a metaheuristic like ACO that produces many solutions and works well with local search procedures.

APPENDIX A

Instance samples

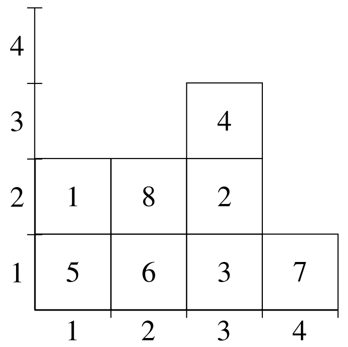


Figure A.1: Example of instance with 4 tiers, 4 stacks and 50% occupancy rate.

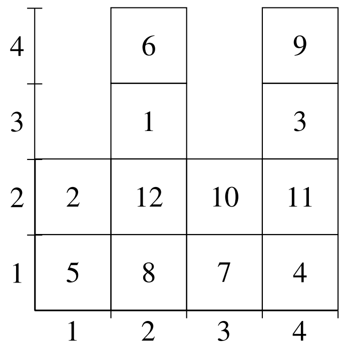
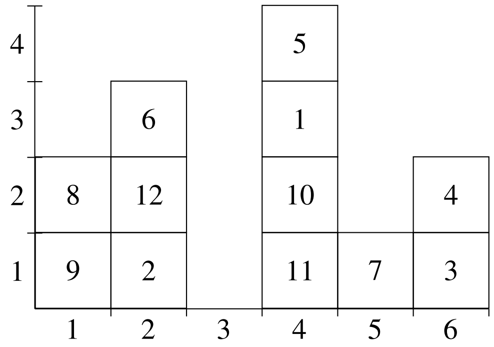
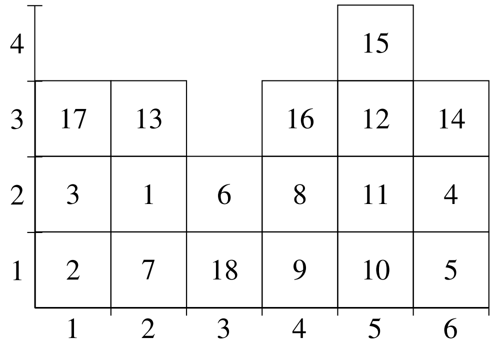


Figure A.2: Example of instance with 4 tiers, 4 stacks and 75% occupancy rate.



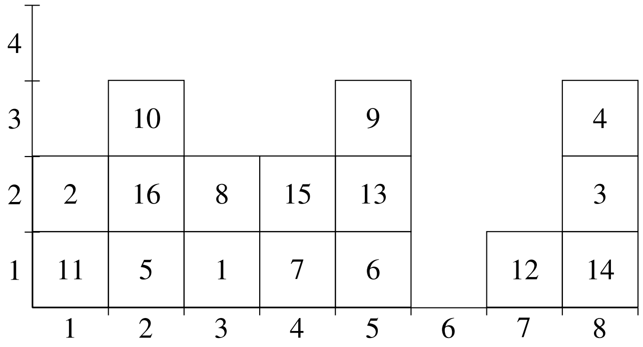
4				5		
3		6		1		
2	8	12		10		4
1	9	2		11	7	3
	1	2	3	4	5	6

Figure A.3: Example of instance with 4 tiers, 6 stacks and 50% occupancy rate.



4					15	
3	17	13		16	12	14
2	3	1	6	8	11	4
1	2	7	18	9	10	5
	1	2	3	4	5	6

Figure A.4: Example of instance with 4 tiers, 6 stacks and 75% occupancy rate.



4								
3		10			9			4
2	2	16	8	15	13			3
1	11	5	1	7	6		12	14
	1	2	3	4	5	6	7	8

Figure A.5: Example of instance with 4 tiers, 8 stacks and 50% occupancy rate.

4		1						14
3	24	10	8		9	3		15
2	20	5	18	17	6	21	12	13
1	2	19	11	16	22	7	23	4
	1	2	3	4	5	6	7	8

Figure A.6: Example of instance with 4 tiers, 8 stacks and 75% occupancy rate.

4								3		
3					2			10	5	
2		7	17	19	6			4	11	
1	14	1	15	9	13	20	18	16	8	12
	1	2	3	4	5	6	7	8	9	10

Figure A.7: Example of instance with 4 tiers, 10 stacks and 50% occupancy rate.

4		19				11		16		23
3	2	7	17	26		14		22		10
2	3	8	4	25		1	12	15	21	6
1	30	27	20	18	5	13	24	29	28	9
	1	2	3	4	5	6	7	8	9	10

Figure A.8: Example of instance with 4 tiers, 10 stacks and 75% occupancy rate.

4								17						
3				27	4			14	24			5		1
2		7		28	15			6	3	18		22	9	11
1	20	10	26	19	8		25	12	13	23		2	21	16
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure A.9: Example of instance with 4 tiers, 14 stacks and 50% occupancy rate.

4	27				23			33		17	35			
3	21		28	9	20		32	31		13	36	34		
2	7	24	22	10	5	30	38	15	29	39	37	4	42	26
1	41	19	1	8	2	6	18	11	12	40	14	3	16	25
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure A.10: Example of instance with 4 tiers, 14 stacks and 75% occupancy rate.

Bibliography

Andreas Bortfeldt and Florian Forster. A tree search procedure for the container pre-marshalling problem. *European Journal of Operational Research*, 217(3):531–540, 2012.

Marco Caserta and Stefan Voß. A corridor method-based algorithm for the pre-marshalling problem. In *Proceedings of the EvoWorkshops 2009 on Applications of Evolutionary Computing*, volume 5484 of *Lecture Notes in Computer Science*, pages 788–797. Springer-Verlag, 2009.

Marco Caserta, Silvia Schwarze, and Stefan Voß. A new binary description of the blocks relocation problem and benefits in a look ahead heuristic. In *Evolutionary Computation in Combinatorial Optimization*, pages 37–48. Springer, 2009.

Marco Caserta, Silvia Schwarze, and Stefan Voß. Container rehandling at maritime container terminals. In *Handbook of Terminal Planning*, volume 49 of *Operations Research/Computer Science Interfaces Series*, pages 247–269. Springer New York, 2011a.

Marco Caserta, Stefan Voß, and Moshe Sniedovich. Applying the corridor method to a blocks relocation problem. *OR spectrum*, 33(4):915–929, 2011b.

Marco Caserta, Silvia Schwarze, and Stefan Voß. A mathematical formulation and complexity considerations for the blocks relocation problem. *European Journal of Operational Research*, 219(1):96–104, 2012.

Young-Tae Chang, Sang-Yoon Lee, and Jose L Tongzon. Port selection factors by shipping lines: Different perspectives between trunk liners and feeder service providers. *Marine Policy*, 32 (6):877–885, 2008.

J-L Deneubourg, Serge Aron, Simon Goss, and Jacques Marie Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of insect behavior*, 3(2):159–168, 1990.

Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

Cees Duin and Stefan Voß. The pilot method: A strategy for heuristic repetition with application to the steiner problem in graphs. *Networks*, 34(3):181–191, 1999.

Christopher Expósito-Izquierdo, Belén Melián-Batista, and Marcos Moreno-Vega. Pre-marshalling problem: Heuristic solution method and instances generator. *Expert Systems with Applications*, 39(9):8337–8349, 2012.

Florian Forster and Andreas Bortfeldt. A tree search procedure for the container relocation problem. *Computers & Operations Research*, 39(2):299–309, 2012.

Fred Glover. Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

Fred Glover. Tabu search-part ii. *ORSA Journal on computing*, 2(1):4–32, 1990.

Simon Goss, Serge Aron, Jean-Louis Deneubourg, and Jacques Marie Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76(12):579–581, 1989.

Shan-Huen Huang and Tsan-Hwan Lin. Heuristic algorithms for container pre-marshalling problems. *Computers & Industrial Engineering*, 62(1):13–20, 2012.

Kap Hwan Kim and Gyu-Pyo Hong. A heuristic rule for relocating blocks. *Computers & Operations Research*, 33(4):940–954, 2006.

Scott Kirkpatrick, MP Vecchi, et al. Optimization by simmulated annealing. *Science*, 220(4598): 671–680, 1983.

Yusin Lee and Shih-Liang Chao. A neighborhood search heuristic for pre-marshalling export containers. *European Journal of Operational Research*, 196(2):468–475, 2009.

Yusin Lee and Nai-Yun Hsu. An optimization model for the container pre-marshalling problem. *Computers & Operations Research*, 34(11):3295–3313, 2007.

Matthias Prandtstetter. A dynamic programming based branch-and-bound algorithm for the container pre-marshalling problem. Technical report, AIT Austrian Institute of Technology, submitted to European Journal of Operational Research, 2013.

Moshe Sniedovich and S Viß. The corridor method: a dynamic programming inspired metaheuristic. *Control and Cybernetics*, 35:551–578, 2006.

Thomas Stützle and Holger H. Hoos. Max-min ant system. *Future Generation Computer Systems*, 16(9):889–914, June 2000.

Stefan Voß, Andreas Fink, and Cees Duin. Looking ahead with the pilot method. *Annals of Operations Research*, 136(1):285–302, 2005.

Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1(6):80–83, 1945.