Development of a Decision Support Tool for the Storage Policy of a Robotic Mobile Fulfilment System

A CEVA Logistics Den Haag Case Study

M. K. Baran

Master Thesis

Development of a Decision Support Tool for the Storage Policy of a Robotic Mobile Fulfilment System

A CEVA Logistics Den Haag Case Study

by

M. K. Baran

Master Thesis

In partial fulfilment of the requirements for the degree of

Master of Science
in Mechanical Engineering

at the Department Maritime and Transport Technology of Faculty Mechanical, Maritime and Materials Engineering of Delft University of Technology
to be defended publicly on Monday September 4th, 2023 at 14.00 PM

Student number:	4481232	
MSc track:	Multi-Machine Engineering	
Report number:	2023.MME.8852	
Supervisor:	Dr. F. Schulte,	TU Delft
Supervisor:	S. Vogelaar,	CEVA Logistics
Thesis committee:	Dr. F. Schulte,	TU Delft committee Chair, 3ME
	Dr. A. Napoleone,	TU Delft committee member, 3ME
	X. Tang,	TU Delft committee member, 3ME
	S. Vogelaar,	company Supervisor, CEVA Logistics
Date:	July 9th, 2023	

An electronic version of this thesis is available at http://repository.tudelft.nl/.

It may only be reproduced literally and as a whole. For commercial purposes only with written authorization of Delft University of Technology. Requests for consult are only taken into consideration under the condition that the applicant denies all legal rights on liabilities concerning the contents of the advice.

TU Delft

Abstract

This paper focuses on the effect of different storage policies on the performance of Robotic Mobile Fulfilment Systems (RMFSs). The research is conducted under the instructions of the Technical University of Delft and CEVA Logistics the Hague. The aim of the research is to develop a decision support tool that can aid companies in the choice of the storage policy to use for their RMFS. RMFSs have multiple different levels of decision problems that need to be solved. The performance of a RMFS highly depends on the algorithms that are applied to solve these decision problems. This research focuses only on the storage policies of a RMFS. In this case storage policy refers to the decision in which pod items should be stored and where on the storage area the pod should be positioned. In order to gain a good understanding of the effect of such a policy on the overall performance of a RMFS, experiments should be performed. Physical experiments are however very hard and costly to perform. This research therefore makes use of a simulation study to test different storage policies in different scenarios. The simulation model used is an adaptation on an agent-based semi-open queuing network framework model by Merschformann et al. (2018a). In the experiments four different storage policies are investigated under three different storage layouts. The results of the simulations are analysed via the throughput, the pile-on and the distance travelled during the simulation. After this a score is given to both the picking as well as the replenishment side of the system. It is important to investigate both sides of the system since the flaws on one side can negatively affect the other side. The scores of the replenishment and picking processes are then averaged to gain a final score which indicates the overall performance of the policies. Finally a fifth policy has been developed where each pod can contain multiple different sized compartments. Unfortunately testing and verification of this policy was not possible in the given time frame. For this reason it has been left out of the experiments.

M. K. Baran
Delft, January 2022

Contents

List of Figures vii

List of Tables ix

List of abbreviations xi

1 Introduction 1
- 1.1 Problem definition 1
- 1.2 Goals and subjects 2
- 1.3 Research questions 2
- 1.4 Report structure 3

2 Literature Research 5
- 2.1 History of the RMFS 5
- 2.2 Decision problems in a RMFS 6
- 2.3 Existing research 7
- 2.4 Research gap 9

3 Conceptual Model 13
- 3.1 Methodology 13
- 3.2 Key Performance Indicators 15
- 3.3 Storage policies 16
  - 3.3.1 Policy 1 - fixed 16
  - 3.3.2 Policy 2 - dynamic, random 17
  - 3.3.3 Policy 3 - dynamic, closest 17
  - 3.3.4 Policy 4 - dynamic, zone based 18
  - 3.3.5 policy 5 - dynamic, heterogeneous locations 18
- 3.4 Assumptions & Boundaries 18
  - 3.4.1 Algorithms 18
  - 3.4.2 Order and replenishment generation 19
  - 3.4.3 Model settings 19

4 Simulation Model Setup 21
- 4.1 Model framework 21
- 4.2 Model parameters 21
  - 4.2.1 Layout 22
  - 4.2.2 System 22
  - 4.2.3 Item generation 23
  - 4.2.4 Control 23
  - 4.2.5 Policy 1 24
  - 4.2.6 Policy 2 25
  - 4.2.7 Policy 3 26
  - 4.2.8 Policy 4 28
  - 4.2.9 Policy 5 28
- 4.3 Input data tool 28

5 Verification & Validation 33

5.1 Input data . . . 33
5.2 Verification - policy 1 . . . 34
5.3 Policy 2 . . . 39
5.4 Verification - policy 3 . . . 39
5.5 Verification - policy 4 . . . 40
5.6 Verification - Policy 5 . . . 43
5.7 Model validation . . . 43

6 Experiments **47**
6.1 Layout . . . 47
6.2 Input data . . . 47
6.3 General settings . . . 48
6.4 KPI evaluation . . . 48
6.4.1 Number of AMRs . . . 49

7 Results **53**
7.1 Long layout . . . 53
7.2 Square layout . . . 55
7.3 Wide layout . . . 57

8 Discussion **61**
8.1 Replenishment activation/deactivation . . . 61
8.2 Pile-on & distance travelled . . . 61
8.3 Throughput . . . 62
8.4 Overall scores . . . 63

9 Conclusion & Future Work **65**
9.1 Recommendations future work . . . 66

Appendices **67**

Appendix A Input data tool: Python script **69**

Appendix B Result processing: Python script **73**

Appendix C Simulation model settings **77**

List of Figures

1.1 AMRs by Geek+ that can be used in RMFSs to transport pods (Robotics 24/7 Staff 2021). 2

2.1 Reconstruction of what a granary in the Jordan Valley would have looked like (Kuijt and Finlayson 2009) 5
2.2 Schematic of the workflow inside a RMFS (Merschformann et al. 2019) 6
2.3 An overview of the decision making levels (Merschformann et al. 2019). 7

3.1 DEGREE research methodology for a simulating study according to Rosetti (2021). 15
3.2 Example schematic of three different zones in a RMFS top down view. 18
3.3 Example schematic of the pod layout for policy 5 compared to policies 1 - 4, where blue represents the pod's frame and the coloured boxes are different sized item bundles. 19

4.1 Probability and volume data of the acquired order history. 31

5.1 Verification of the input data tool. 33
5.2 First verification tests for policy 1. 36
5.3 Second verification tests for policy 1. 38
5.4 Verification test for policy 3 showing that pods converge to the stations as intended. 39
5.5 Verification tests for policy 3. 40
5.6 Verification tests for policy 4. 42
5.7 Results of policy 3, using 1 compartment and policy 2 with the same simulation settings. 43
5.8 44
5.9 The throughput of all policies using the layout and input data from the RMFS for two different weeks. 46

6.1 The three different layout configurations used in the experiments, with yellow being the replenishment stations, red the picking stations and blue the pods. 48
6.2 The throughput of different policies using different amounts of AMRs. 50
6.3 The relation between the number of AMRs and the number of picking stations according to Gong et al. (2020). 51

7.1 Throughput of the system for all policies using the long layout. 54
7.2 Pile-on of the system for all policies using the long layout. 54
7.3 Distance travelled of all policies for the long layout using seed 0. 55
7.4 Throughput of the system for all policies using the square layout. 56
7.5 Pile-on of the system for all policies using the square layout. 56
7.6 Distance travelled of all policies for the square layout using seed 0. 57
7.7 Throughput of the system for all policies using the wide layout. 58
7.8 Pile-on of the system for all policies using the wide layout 58
7.9 Distance travelled of all policies for the Wide layout using seed 0. 59

List of Tables

1 List of abbreviations, part A xi
2 List of abbreviations, part B xii

2.1 A table highlighting the subjects discussed in some of the most relevant papers covering the RMFS. 11

3.1 Table with advantages and disadvantages of each storage policy 17

4.1 Table showing the relative capacities of the storage classes 28

5.1 The SKU characteristics used as input data for the verification of policy 1. 34
5.2 Monitored 6 pods and their initial/final location/contents, applying policy 1. 35
5.3 Monitored 9 pods and their initial/final location/contents, applying policy 1 with its fixes. . . . 37
5.4 Monitored pods and their initial/final location/contents, applying policy 4. 41
5.5 Table showing the relevant parameters used for the validation simulation. 44
5.6 Table showing the relevant parameters used for the validation simulation. 45
5.7 The average throughput per hour for the historical data and the simulation model. 45

6.1 SKU properties of the input data. 47
6.2 Table showing the relevant parameters and their values used across all simulations. 49
6.3 The throughput of the policies using different amounts of AMRs. 50

7.1 All KPI results and the final score for the different policies using the long layout. 53
7.2 All KPI results and the final score for the different policies using the Square layout. 55
7.3 All KPI results and the final score for the different policies using the Wide layout. 57

C.1 Table showing all layout parameters used in the model part A. 78
C.2 Table showing all layout parameters used in the model part B. 79
C.3 Table showing all system parameters (excluding the order generation) used in the model. 79
C.4 Table showing the parameters used to generate the item list. 80
C.5 Table showing the parameters used to generate order from the item list during the simulation. . 81

List of abbreviations

RMFS	Robotic Mobile Fulfilment system
AMR	Autonomous Mobile Robot
B2C	Business to Customer
B2B	Business to Business
FTE	Full Time Employee
SKU	Stock Keeping Unit
OA	Order Assignment
TC	Task Creation
TA	Task Allocation
PP	Path Planning
POA	Pick Order Assignment
ROA	Replenishment Order Assignment
PS	Pod Selection
PSA	Pod Storage Assignment
PPS	Pick Pod Selection
RPS	Replenishment Pod Selection
COSLAPP	Collaborative Optimisation of Storage Location Assignment and Path Planning
SOQN	Semi-Open Queuing Network
OQN	Open Queuing Network
CQN	Closed Queuing Network
DES	Discrete Event Simulation
TP	Throughput
FT_a	Average order Flow Time
I	Total amount of processed item during a measurement
T	Total time of a measurement, in hours
TA_i	Arrival Time of order i, in seconds
TD_i	Departure Time of order i, in seconds

Table 1: List of abbreviations, part A

O	Total amount of Orders over a time interval
$C_{utlised}$	The total utilised capacity, in %
C_f^n	Filled capacity of pod n, in m^3
C_p	Total capacity of a pod, in m^3
P	Total amount of pods in the system
KPI	Key Performance Indicator

Table 2: List of abbreviations, part B

1 Introduction

The warehouse is a very important component in the logistics chain. Some of its most important functions are: storing of items, picking of orders and shipping and transporting of goods. The most labour intensive task of such a warehouse often lies in the order picking. This is also its most crucial task when it comes to customer response and satisfaction (Cai et al. 2021). Manual order picking usually covers around 60% of a warehouse's labour and 30-40% of its operation time (Bassan et al. 1980)(Tompkins et al. 2010). This is mostly due to the fact that pickers constantly have to move from the storage area to the picking area to fulfil orders.

For this and many other reasons e-commerce companies are looking in to warehouse automation more and more. Increasing productivity, decreasing operational costs and increasing customer satisfactions are all reasons why e-commerce companies are eager to delve in to warehouse automation technologies. The Robotic Mobile Fulfilment System (RMFS) is one of these developments in the warehouse automation industry. In a RMFS, small Autonomous Mobile Robots (AMRs) are deployed that eliminate the need for movement by employees. These AMRs are capable of lifting storage pods and bringing them from the storage area to the replenishment/picking stations. The stations are in turn designed so that an employee will have access to all the necessary equipment and will thus no longer have to walk. This can potentially double the picking productivity according to Wurman et al. (2008).

1.1. Problem definition

CEVA Logistics Den Hague is adding another structure inside their warehouse, a picking tower, specific for a new client who specialises in low value jewellery and ad-dons. A picking tower is a structure inside a warehouse, often with multiple floors, where the replenishment and storing of items and picking of orders take place. The tower should support two flows, B2B (Business to Business) for supplying retail and B2C (Business to Customer) for e-commerce purposes. One requirement from the client is to minimise the dependency on FTEs (Full Time Employees), resulting in picking and packing to become partially automated, or in other words a RMFS.

The workflow in the new picking tower will be as follows. Items that enter the warehouse through the inbound section will first be transported to replenishment stations of the new picking tower via a conveyor system. Here the items are stored in pods, which are brought from the central storage area to the replenishment stations by AMRs. When the pods have been filled they are moved back to the storage area, again by an AMR. See figure 1.1 for an example of such AMRs developed by Geek+. Then whenever an order enters the system, the pod containing the relevant items is transported by an AMR to a picking station. A picking station is responsible for putting together orders for customers, by picking the right quantity of items from the storage pods and putting the items in a carton to form the order. When the order is completely picked, the carton will be transported via a conveyor system to a box closing machine and will eventually end up at a parcel sorter.

Figure 1.1: AMRs by Geek+ that can be used in RMFSs to transport pods (Robotics 24/7 Staff 2021).

To reach the greatest efficiency possible, it is necessary to optimise all steps of the RMFS. This includes the replenishment process, the storage location and storage quantity of every item that enters the system, the picking process and the routing of the AMRs. Decreasing travel time of the robots, minimising employee waiting times, maximising pick hits per pod (when items that are being stored in the same pod are ordered together) and fully utilising the available storage capacity are some of the system's main challenges. With many replenishment and picking stations, hundreds of different items and possible pod configurations this becomes a very complex optimisation problem.

CEVA Logistics Den Haag does not currently have a solid way of determining the storage policy for their new RMFS. Storage policy referring in this case in which location an item should be stored during the replenishment process and to which storage space the pod should return after it has been used at a station. At the moment this decision is based on a client's general product information, such as size of the products and order frequency, and on what has worked best in the past. This is mostly based on intuition and not on any concrete model. This research is thus performed in order to create a scientific tool which can assist companies in the decision of the best storage policy for any new client in a RMFS.

1.2. Goals and subjects

The main goal of this thesis is to develop a model that can be used to gain insight on the performance of different storage policies based on a clients respective product and order data and to be able to give a well argued advise on which policy to choose when adopting a new client in a RMFS. This goal can be separated in to three smaller goals. The goals of this thesis then are:

- To be able to interpret product and order data from any client for its most relevant characteristics.
- To develop a model that shows the performance of different storage policies based on the respective product and order data.
- To be able to give a well-argued advise on which storage policy to use for any client.

1.3. Research questions

The previously mentioned research goals will be achieved by answering a main research question and a few relevant sub questions. The main research question will be as follows:

- What is the best storage policy to use in a RMFS to maximise performance in the picking phase whilst minimising losses at the replenishment phase?

This research question will be aided by some sub-questions, which can be read down below. The first sub-question covers the different policies that will be researched. The second question is about product and order data and how to model this in a simulation. The third and fourth sub-questions cover the developing

of the model. The last two sub-questions will cover the final details of the model and the verification and validation.

1. What are the different storage policies that can be applied to a RMFS?
2. What modelling methodology is the most suitable for this problem?
3. What are all relevant parameters that can influence the system?
4. How to model demand rates and product information to use in a simulation model?
5. How can the model be made adaptable in such a way that it can be used for different product and order distributions?
6. How can the model be verified and validated using historical data?

1.4. Report structure

The research will be structured in the following manner. First the literature will be explored in chapter 2, covering robotic mobile fulfilment systems (RMFSs). In this chapter some background information is given on the history of the RMFS, followed by information on the working mechanics of a RMFS. Then the already existing research on RMFSs will be discussed and finally the research gap, or what is still missing from the existing literature and what this research will add to this.

Next in chapter 3 the methodology of this research is explained. What steps have to be followed to conduct a successful simulation study. Then the key performance indicators (KPIs) shall be discussed that are used to measure the performance of the simulations. After this the actually investigated storage policies are covered followed by some model assumptions and boundaries. Then in chapter 4 the full setup of the simulation model is discussed. The used framework is shortly covered and all the different parameters that are important for the simulations. Also the pseudo code for all the different storage policies is shown in this chapter followed by some information on the built python tool to translate company order data in to usable input data for the model. After the model is built, chapter 5 covers the verification and validation of the newly built storage policies and the adjusted framework model. The verification experiments are mostly performed using the integrated GUI of the framework model. For the validation actual data from an existing RMFS is used to compare to the model.

Now that the model is fully verified and validated, chapter 6 will cover the experiments that are going to be performed with the model. This chapter will cover the different simulation settings that will be used and in what manner the results are going to be evaluated. Also some more detailed information on the selection of the amount of AMRs is discussed. After this, in chapter 7, the results from the simulations are posted. These results are then evaluated in chapter 8 and an answer to the main research question is given. Finally chapter 9 will conclude the research and give some recommendations for future work.

2
Literature Research

The RMFS seems like a simple system at first glance, but there is actually a lot of decision logic that is necessary for a RMFS to operate smoothly. In this chapter the history and core mechanics of the RMFS and some of its most important jargon will be discussed. Furthermore existing researches and literature are analysed in order to create a better grasp of what fields have already been investigated and what fields are still left untouched.

2.1. History of the RMFS

To start explaining the concept of a RMFS it is first necessary to understand what a general warehouse is and how it works. RMFSs are namely a type of automated warehouse. Nowadays warehouses are a very important part of almost any supply chain. Their main purpose is to buffer goods in order to facilitate variability within such a supply chain. This can be due to product batching or seasonality for example (Gu et al. 2006).

The concept of warehousing started a long time ago. The first type of warehouses came in the form of granaries. A granary is a building or room in which supplies can be stored. Its main purpose was often for storing grains and food. The oldest granaries which have been discovered are over 11,000 years old and are located in the Jordan Valley (Kuijt and Finlayson 2009). An illustrative reconstruction of what such a granary looked like can be seen in figure 2.1.

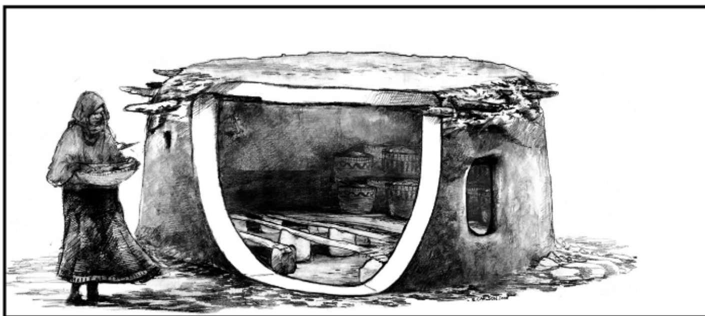


Figure 2.1: Reconstruction of what a granary in the Jordan Valley would have looked like (Kuijt and Finlayson 2009)

In the past, during the industrial revolution, storage was a very important aspect of the economy. Labour was cheap and manpower widely available. This however caused warehouses to be inefficient and not very well organised. It was only after world war II that increasing attention to the operational and managerial efficiency of warehouses was starting to be paid. The main reasons for this were the higher labour costs and the technological developments at the time (Ashayeri and Gelders 1981).

Nowadays warehouses can be manual, automated or semi-automated. Meaning a warehouse operates completely on employees doing labour, with employees doing labour in combination with machines, or that machines have completely replaced employees on the work floor. A RMFS is a type of semi-automated ware-

house. An RMFS operates using dedicated replenishment and picking stations. These stations are located on the edge of a larger storage area which is usually filled with shelves that contain products. These shelves are usually referred to as 'pods' and the products are referred to as 'Stock Keeping Units (SKUs)'. In a manual warehouse, employees will have to walk through this storage area to replenish or pick SKUs from the pods and then bring them to the relevant station to form a complete order. However in a RMFS the storage pods are carried from the storage area to the replenishment or picking stations and back by Automated Mobile Robots (AMRs). This eliminates the need for employees to walk. As mentioned in the introduction, this can potentially double the picking productivity of a warehouse. The complete workflow of a RMFS can be seen in figure 2.2.

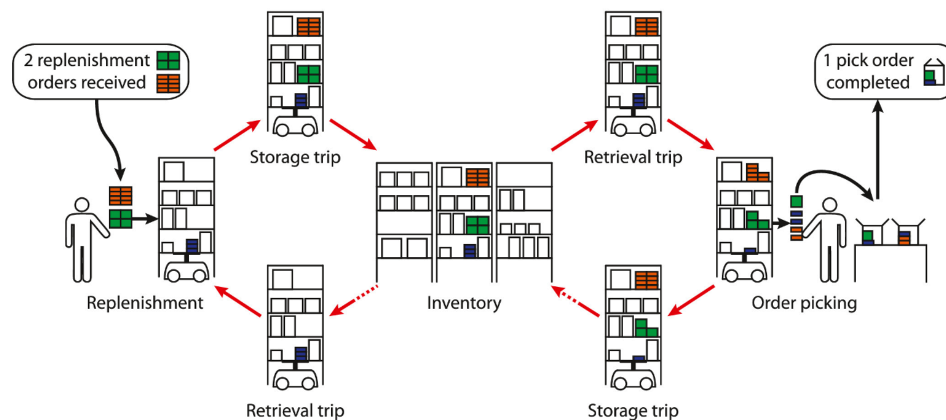


Figure 2.2: Schematic of the workflow inside a RMFS (Merschformann et al. 2019)

The idea of a RMFS was first conceptualised by Jünemann (1989), however it was only patented by Mountz et al. (2008) under the company KIVA Systems Inc. (now Amazon Robotics), which is now known as the founder of the RMFS. Nowadays many other companies have taken over the concept such as: Geek+, CarryPick™, Swisslog and Butler™ (Azadeh et al. 2019).

2.2. Decision problems in a RMFS

Behind the relatively simple workflow of a RMFS, is a lot of very complex decision making. At first when a pod has to be replenished, the respective replenishment policy has be to determined. Which pod should be retrieved and in which location in the pod should a SKU be put, are both questions that need to be answered. After a pod has been replenished, the pod has to return to the storage area. This is yet another decision that has to be made. Fairly similar decisions can be made for the picking process. Finally the AMRs themselves require a lot of decision making. In order to prevent the AMRs from colliding, making sure they follow efficient routes and that they retrieve the right pods are all examples of decisions that have to be made (Merschformann et al. 2019). In this subsection the most important decision making processes of a RMFS and some important jargon will elaborated.

The aim of the decision making logic in a RMFS is to keep the employees at the stations busy whilst minimising the required resources (AMRs) to do so. The decision problems in a RMFS can generally be divided in to three levels: strategic, tactical and operational level.

First the strategic decisions for a RMFS are made at time of construction. Decisions such as the layout of the storage area, the placement of the work stations and type of robots to use are made at this level. Then after the RMFS is constructed it is in the tactical level. Now it is time to make decisions such as: What will be the number of pods per SKU? How many replenishment or pick stations should be active? What is the storage level beneath which replenishment should start? The final level is the operational level. At this moment the RMFS is in operation and decisions are made in real time. The decisions in the operational level can be divided in to four steps.

1. Order Assignment (OA)
2. Task Creation (TC)

3. Task Allocation (TA)
4. Path Planning (PP)

The first step OA, describes the decisions involved to allocate a Pick Order Assignment (POA) or Replenishment Order Assignment (ROA) to a workstation. A POA is a request to fulfil a certain order i.e. to pick the right amounts of the required SKUs from the pods that arrive at the picking station to fill a package and complete the order. A ROA is a request to fill a pod that arrives at the replenishment station with the right SKUs. The second step TC, describes the creation of certain tasks for the AMRs. These tasks are Pod Selection (PS) and Pod Storage Assignment (PSA). PS is the task of selecting a pod that has to be transported to a workstation. PSA is the task of returning that pod to a specific storage location. The PS task is also divided in to two subcategories: Pick Pod Selection (PPS) and Replenishment Pod Selection (RPS). This is needed because the selection processes for picking and replenishing differ, whereas PPS is reliant on due times. TA then uses all aforementioned tasks to create a 'trip' for an AMR. Basically TA builds a sequence of tasks that the AMR needs to follow. This sequence of tasks is then the input for the PP, where the exact route the AMR will follow is generated (ibid.). An overview of the three levels and the different tasks in the operational level can be seen in figure 2.3. This research will be performed at the operational level and focuses in particular on the second step, TC.

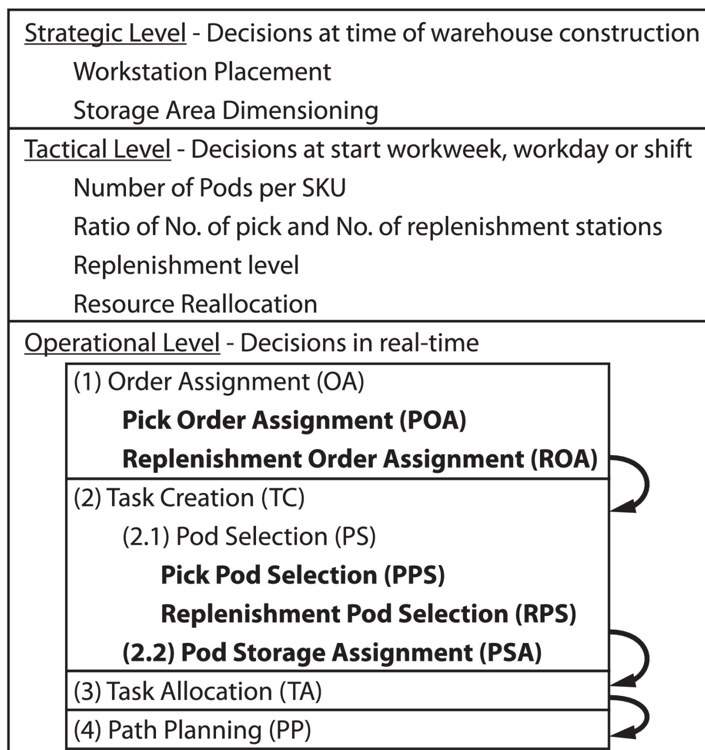


Figure 2.3: An overview of the decision making levels (Merschformann et al. 2019).

2.3. Existing research

Enright and Wurman (2011) mentions that resource allocation is a largely understudied subject for semi-automated warehouses. Aspects such as inventory pod selection, storage location and order allocation all influence the efficiency of such a system. This is however a study from more than 10 years ago making the relevance of the mention disputable.

Looking more recently, multiple studies can be found covering location assignment in RMFS. For example Keung et al. (2021) presents a location assignment algorithm using machine learning. Here they make use of a clustering algorithm. Meaning that certain SKUs are allocated to certain fixed zones. This should decrease the travelling costs of the AMRs and increase the total throughput. This makes the system not fully dynamic

since zones have to be predetermined. The determination of the zones is however done through a classification algorithm making human interaction obsolete. The paper furthermore mentions quite some relevant studies in the storage location field, which seem to focus on the following subjects:

1. Sum of similarity values of SKUs in a rack (Kim et al. 2020).
2. Energy consumption aware evaluation method (Li et al. 2020).
3. Zone assignment strategy in single deep scenario (Roy et al. 2019).
4. Optimising the number of movable racks per Stock Keeping Unit (SKU), the ratio of replenishment stations to picking stations and a new form of semi-open queuing network model (Tessensohn et al. 2020).

Another recent paper by Cai et al. (2021) presents a location assignment model based on the path length. Calling it the collaborative optimisation of storage location assignment and path planning (COSLAPP) model. Here it optimises the storage location taking in to account path length and pick hits. Pick hits are SKUs that are often required in the same order, thus making it efficient to put those SKUs in the same pod. So what is determined in this research is where SKUs should be stored in the system and how the AMRs should drive to reach a minimal spent time. To simplify the model multiple assumptions are made. The most important assumptions regarding the storage of SKUs, so not including the path planning, are the following:

1. Each rack/pod is homogeneous. Meaning the size, specification and number of goods per inventory is the same for all racks.
2. Each storage space can only be occupied by 1 type of item.
3. An order can only contain a maximum variety of goods and a rack containing goods that is moved at a certain time must meet the demand required for that specific order.
4. The order in which items are picked is always the same within 1 operational cycle.

To continue in the space of path planning. A study by Merschformann et al. 2018a covers multiple path planning algorithms to see their effect on the performance of a RMFS. According to the authors conventional path planning algorithms are not suitable to handle some of the constraints that arise when working with robots. Kinematic constraints such as maximum velocity limits and turning speeds for example. The goal of this research is to develop a new path planning algorithm that is able to deal with these added constraints, based on conventional algorithms. The paper concludes the research by stating that a modification of the conventional A-star algorithm, WHCA (Windowed Hierarchical Cooperative A) algorithm, performs the best for their specific test case, but that it does not scale very well. The FAR (Flow Annotation Replanning) algorithm however performs well even for very large instances.

The amount of AMRs used in a RMFS is also a very important parameter. Gong et al. (2020) performed a study in which was investigated what the relation is between the amount of AMRs, the number of pickers and the picking throughput. They discovered that bottlenecks can be created when utilising either too few AMRs or too few pickers. To account for this, system managers should develop contour lines which can indicate where the optimal combination lies between the number of AMRs and the number of pickers. The paper concludes by stating that more research should be performed using different storage policies and on the AMR charging issue that was encountered.

In 2019 a somewhat more recent literature review has been conducted by Azadeh et al. (2019). Here multiple papers covering the efficiency of RMFSs are discussed. The biggest take away from this literature review is that there has been a lot of research done on efficient picking systems and different storage strategies. A research by Xie et al. 2021 for example studies the effect of splitting orders at picking stations to increase the efficiency. The literature review recommends future researchers to look at the dynamicity of the system. Most current papers rely on a static demand policy whilst in reality this is very dynamic. Also pod inventories are, whilst optimally sorted, often regarded as static. The paper recommends the use of Artificial Intelligence to help decide which SKUs should be stored where and where the pods should be stored after they have been stocked.

Some other very recent work exists in the form of a master thesis by Galliussi (2022). She did a simulation study on the performance of a RMFS using three different type of storage strategies. The strategies used are:

1. Each item is always stored in the same place.
2. SKUs are stored randomly and can be allocated anywhere in the storage area.
3. SKUs are assigned to a zone in which they can be stored randomly.

She concluded the research by stating that the best strategy is spreading SKUs among multiple pods. Mainly because this increases the odds of orders containing multiple items from a single pod, which results in less Automated Guided Vehicle (AGV) movement. Furthermore she mentions that for future research more work should be put on studying the performance of RMFSs and that the use of actual real input data will increase the accuracy of the results.

Finally Merschformann et al. 2019 conducted a study on different decision making algorithms in RMFSs. In this study a focus was put on a bunch of different OA, PS and PSA algorithms. They tested a total of 1600 different scenarios. Each includes multiple RCs (Rule Configurations) and WSs (Warehouse Scenarios). The research was split in to two phases, where phase 1 uses eight performance measures to investigate which scenarios perform the best. Phase 2 includes the best performing scenarios from phase 1 and 6 benchmark scenarios. These benchmark scenarios are configured in such a way that each decision making algorithm is present in at least one of them. After the experiments that followed, the research states that a high pile-on and a short total distance travelled by the AMRs are in general very important statistics for the success of the decision making algorithm of a RMFS. Pile-on in these kind of systems usually refers to the amount of units that can be picked from a single pod visiting a picking station. It also concludes that the throughput shows similar variations to other performance indicators, meaning that throughput on its own can be used as a well enough metric to measure the success of a RMFS. The last important mention is that primarily the POA decision is responsible for fluctuations in the throughput, so this should be prioritised by system engineers and warehouse operators.

2.4. Research gap

All the aforementioned studies measure the performance of a RMFS by the performance of the picking process. However such systems consist of many more processes. One of these processes, that is very important in a RMFS, is the replenishment process. Although the performance of this process is often not used as a direct measure of how well a RMFS is doing, it should not be neglected when measuring the overall performance. This subject is however disregarded in most researches. Most papers focus on the optimal storage locations of the SKUs and the picking process hereafter. They do not take in to account how this affects the replenishment process.

The replenishment process impacts the entire system. It is easy to say where SKUs should be stored to reach an optimal picking efficiency, but this is often not in line with what is optimal for the replenishment process. One can imagine that it is very efficient for two SKUs that are often bought together, to be stored next to each other in the same pod. However at the replenishment station these items might not arrive one after the other. Items at the replenishment stations often arrive in a more random order. The two items can thus in this case not be stored in the same pod without the pod experiencing extra waiting times or with the pod having to travel to the replenishment station twice, which is both unpractical. This is then also where the discrepancy lies between an efficient layout of the pods, optimal for picking, and an efficient replenishment process. For example the solution proposed by of Cai et al. (2021) should be extended with a study that includes the replenishment phase to see what the actual performance of the system would be when optimising the storage in the respective manner.

The research by Roy et al. (2019) is to the authors knowledge the only other study that takes the replenishment process in to account for their analysis. In this paper the effect of utilising dedicated AMRs versus pooled AMRs is studied. The used storage policy in this case is a dynamic homogeneous policy. Meaning that items can be placed in any pod and that all locations in a pod are of equal size. Furthermore single versus multiple zone storage is investigated. The results show that using pooled AMRs, the picking time reduces up

to 66%. However, what most studies ignore, is that the replenishment time increases by up to three times. The paper suggests to investigate the problem in the future using more than two storage zones. Applying different storage strategies might also help reduce the increased replenishment time.

Currently multiple policies exist for storing SKUs in a RMFS, with every policy having its advantages and disadvantages. Which policy is best for which situation is often hard to determine and based on pure insight. Studies such as that of Galliussi (2022) and Lamballais et al. (2017b) show that certain policies are work better than others. Zoning policies in specific seem to perform well. However these studies do not take the difference between clients in to account. More specifically their order information. Size of the products and pick hits between SKUs are two examples of characteristics that can change what the optimal storage policy would be. For this reason part of the main goal of this research will be to give a well-argued advise on the optimal storage policy for any RMFS based on the order and product data for that specific client. The performance in this case should thus not only be judged through the picking performance but also using the replenishment performance. In table 2.1 below, a summary can be read on the subjects covered by the aforementioned papers. In this table it is also clearly visible that the replenishment phase is an often neglected subject in research.

Paper \ Subject	Literature review	Storage zone clustering	Other storage strategies	Pick hits	Path planning	Replenishment performance	Picking performance	Machine learning
Enright and Wurman (2011)	x							
Azadeh et al. (2019)	x							
Keung et al. (2021)		x			x		x	x
Cai et al. (2021)		x	x				x	
Galliussi (2022)		x		x	x		x	
Kim et al. (2020)			x				x	
Li et al. (2020)		x	x	x			x	
Roy et al. (2019)		x			x	x	x	
Tessensohn et al. (2020)			x	x			x	
Merschformann et al. (2018a)		x	x				x	
Merschformann (2017)			x				x	
Merschformann et al. (2019)					x		x	
Xie et al. (2021)							x	
Wang et al. (2022)		x					x	
Gong et al. (2020)			x				x	
Lamballais et al. (2017b)		x					x	
Lamballais et al. (2017a)			x	x			x	
This research			**x**			**x**	**x**	

Table 2.1: A table highlighting the subjects discussed in some of the most relevant papers covering the RMFS.

3
Conceptual Model

Often when doing a simulation study it is very useful, and sometimes needed, to first describe the conceptual model. This means to describe the most important aspects that the model will include without actually building the model. In this chapter information on the used methodologies, performance indicators, assumptions and boundaries and the different to be investigated policies will be elaborated.

3.1. Methodology

The goal of this research is to theoretically evaluate real world operations. There are usually two options when trying to imitate real world situations. The first is to make a physical (scale) model and the second option is to make a digital model. More often than not a problem is however to big or complicated to easily make a physical model. In these cases a digital one might be the only viable solution. A digital copy of a real world operation or process, that is used for evaluating different scenarios, is what is called a simulation model. Also for this research a complete physical model will not be viable, therefore a simulation model is the perfect solution (TWI Global 2023)(AnyLogic 2022).

From literature it is clear that queuing networks are often used to simulate RMFSs. Specifically, Semi-Open Queuing Networks (SOQN). Queuing networks can be separated in to three main types:

1. Open Queuing Networks (OQN)
2. Closed Queuing Networks (CQN)
3. Semi-Open Queuing Networks (SOQN)

OQN can be explained as a system where customers arrive from outside the systems, undergo service at some nodes and then leave the system. CQN is a system where nothing goes in or out of the system and the number of customers is fixed. SOQN is a combination of OQN and CQN. Here customers can enter the system from outside and leave once they have received their relevant services, which resembles an OQN. There is however also a capacity constraint on the services within the system, which resembles a CQN. In this case, whenever a customer enters the system and there is no service available the customer has to enter an external queue (Azadeh et al. 2017)(Chen and Yao 2001)(Roy 2016). When looking at an RMFS, the customers are the products that continuously flow in and out of the system and the services are in this case the AMRs that deliver a transportation service to the products. For this reason a SOQN is the most appropriate for simulating an RMFS.

Furthermore systems such as RMFSs can be represented by Discrete Event Simulations (DES). DES represents systems in a non-continuous manner. In such a simulation events occur at fixed moments causing the state of the system to change. Therefore only these state changes are recorded at certain time instances, making it a discrete simulation (OCW TU Delft 2022).

The research methodology of this study will be according to the proposed DEGREE methodology by Rosetti

(2021). This methodology revolves around a series of steps that can be used during the general process of solving a problem. The steps are the following:

1. Define the problem
2. Establish measures of performance for evaluation
3. Generate alternative solutions
4. Rank alternative solutions
5. Evaluate and iterate during process
6. Execute and evaluate the solution

The first step is meant to ensure that one is solving the correct problem. The second step ensures that the problem is solved for the correct reason. The third and fourth step are particularly meant for one to evaluate different solutions of the problem. This is to ensure that the final solution is indeed the best solution. The fifth step is where someone should evaluate how the process is going and should allow iterations. It is important to understand that a problem solving process may be repeated to ensure that the model will reach its destined goal. According to Rosetti (2021) it is better to start the model at a level that allows for quick implementation without having to create the entire model all at once. The final step is the actual practical implementation of the model. The outcome of a simulation model is often used as a solution to a real world problem, however this should preferably always be tested first in practise.

In general the DEGREE methodology should be of great support when solving a problem. However when modelling a simulation there are a number of specific actions that need to be performed to which the DEGREE methodology should be adapted. In principle all of the steps from the DEGREE methodology are also incorporated in the five phases for using simulation. Below the complete list of phases and sub-steps that should be followed according to Rosetti (ibid.) for applying simulation to a problem can be read and in figure 3.1 a full overview of all the steps and potential iterations can be seen.

1. Problem formulation
   (a) Define the problem
   (b) Define the system
   (c) Establish performance metrics
   (d) Build conceptual model
   (e) Document model assumptions
2. Simulation model building
   (a) Model translation
   (b) Input data modelling
   (c) Verification
   (d) Validation
3. Experimental design and analysis
   (a) Preliminary runs
   (b) Final experiments
   (c) Analysis of results
4. Evaluate and iterate
   (a) Documentation
   (b) Model manual
   (c) User manual
5. Implementation

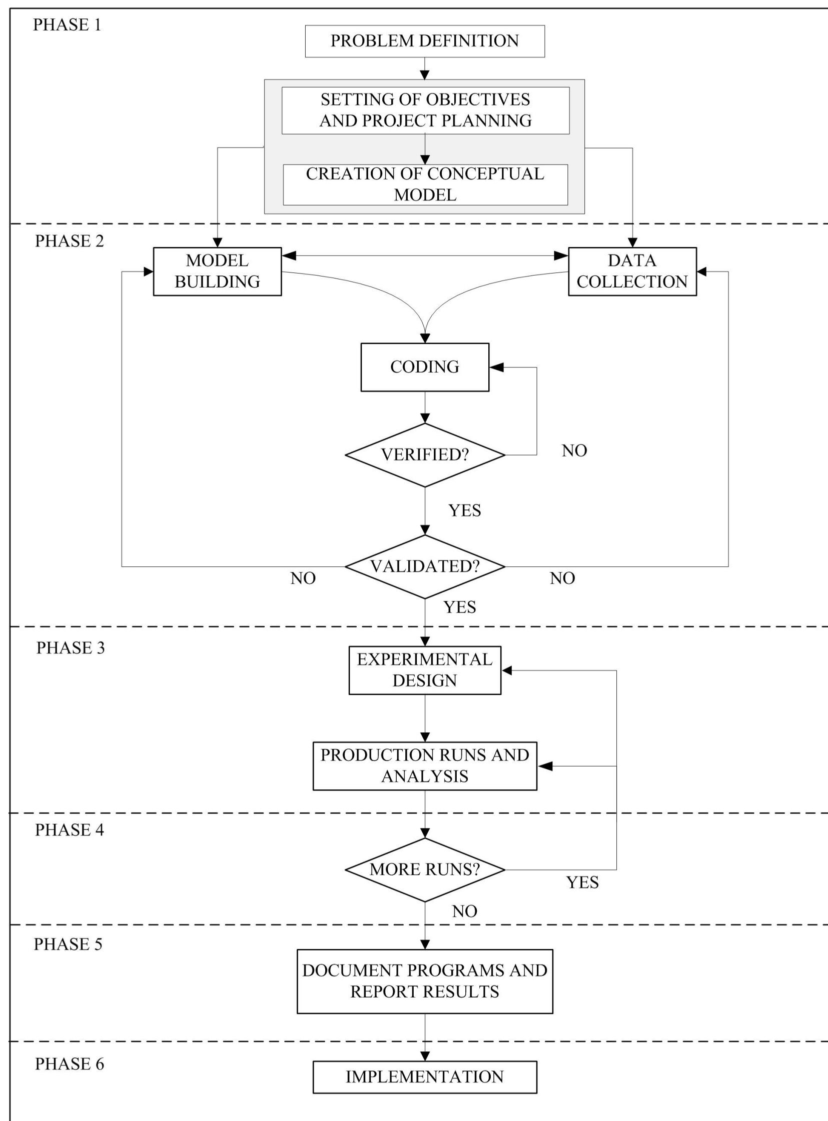


Figure 3.1: DEGREE research methodology for a simulating study according to Rosetti (2021).

3.2. Key Performance Indicators

A lot of performance factors, or Key Performance Indicators (KPIs), can be determined from the operation of a RMFS. However some KPIs can give a much better idea of the effectiveness of a RMFS's decision making policies as has been discovered by Merschformann et al. (2019), two of which are the item pile-on (PO) and the total distance travelled (DT) by the AMRs. Item pile-on in the case of a RMFS is introduced by the paper as the amount of items that can be picked from a single pod in a single visit to a pick station. The pile-on is then averaged over all the pod visits that were done during the simulation horizon. This measure is a great representation of the amount of pod movements that need to be made over a specific amount of time. A higher pile-on rate will decrease the amount of pod movements between the pick stations and the storage area. This KPI can also be applied to the replenishment side of the system. In this case the pile-on represents how many bundles can be placed inside a single pod per single visit to a replenishment station. The pile-on can be written as follows:

$$PO_p = \frac{\sum_{v=1}^{V} I_v}{V} \tag{3.1}$$

Where:
PO_p = The averaged picking pile-on rate (items/visit).

I = The amount of items picked from a pod per visit v.
V = The total amount of visits to the picking stations over the simulation horizon.

$$PO_r = \frac{\sum_{v=1}^{V} B_v}{V} \tag{3.2}$$

Where:
PO_r = The averaged replenishment pile-on rate (bundles/visit).
B = The amount of bundles stored in pod p per visit v.
V = The total amount of visits to the replenishment stations over the simulation horizon.

The second KPI that tells a lot about the performance of a decision making policy in an RMFS is the average distance travelled by each AMR. The amount of distance travelled is of course related to the amount of time spent. During this time items can neither be picked nor replenished to that pod and the AMR is unavailable to perform other tasks. In other words the longer an AMR spends travelling the less efficient the system is. For this reason the average distance travelled is a good indicator of the performance of a RMFS. For this research the averaged distance travelled per AMR will be used as the measure. This KPI can be written as:

$$DT = \frac{\sum_{a=1}^{A} D_a}{A} \tag{3.3}$$

Where:
DT = The average distance travelled per AMR (m/AMR).
A = The total amount of AMRs in the system.
D_a = The total distance travelled per AMR a (m).

Both the aforementioned KPIs are highly correlated with the throughput of the system. The throughput of a RMFS is often its most interesting performance factor, while it is a great indicator of the revenue that is made (Archambault 2020). The throughput in a RMFS is a value that tells the amount of items that flow through the system over a given period of time. In the case of the picking side, it is described as the amount of orders per hour that are fulfilled. In the case of the replenishment side it is usually described as the amount of bundles that are stored each hour. It can be calculated using the following formula (Stauffer 2022):

$$TP = \frac{I}{T} \tag{3.4}$$

Where:
TP = The throughput of the system (orders/hour or bundles/hour).
I = The total amount of orders/bundles that were processed during the measurement.
T = The total time of the measurement (hours).

This research focuses on the effect of different storage policies on the performance of both the picking process as the replenishment process. These KPIs are therefore able to cover both these parts of the system. It should however be noted that in case of the picking the KPIs are measured in items or orders and in case of the replenishment the KPIs are measured in bundles.

3.3. Storage policies

The goal of this research is to investigate the effect of different storage policies in a RMFS. It is therefore necessary to conceptualise the policies that are going to be evaluated in the experiments. The policies that will be utilised in the experiments are elaborated in the following subsections. Policies 1, 2 and 5 are particularly chosen since these are the policies that the case study currently considers whenever they acquire a new customer. The third policy is mainly chosen for its simplicity and the fourth policy is chosen while it is a policy that is proven in literature to be better performing than other common policies, such as in Cai et al. (2021), Galliussi (2022) and Azadeh et al. (2019). A quick overview of the most important advantages and disadvantages of each policy can be read in Table 3.1 below.

3.3.1. Policy 1 - fixed

The first policy is going to be the most simple. With this policy every item has a fixed location and all location are of the same size. Advantages of this policy are: SKUs are always stored in the most optimal location and

Policy	Advantages	Disadvantages
1. Fixed	Optimal item locations, Constant process times	Pods always travel to/from the same location, Non optimal use of storage capacity
2. Dynamic, random	Simple, Replenishment to any pod, Good space utilisation	Chaotic storage, Non optimal use of storage capacity, Diverse process times
3. Dynamic, closest	Replenishment to any pod, More optimal use of storage capacity	More complex, Chaotic storage, Diverse process times
4. Dynamic, zone based	Prioritises high demand SKUs, Moderate optimal item location, Decent storage utilisation	Moderately complex, Diverse process times
5. Dynamic, heterogeneous	Replenishment to any pod, More optimal use of storage capacity	More complex, Chaotic storage, Diverse process times

Table 3.1: Table with advantages and disadvantages of each storage policy

processing times remain relatively constant throughout operation. Disadvantages include: pods always have to travel from and to the same location and the storage space can not be optimally occupied.

3.3.2. Policy 2 - dynamic, random

The second policy is a more dynamic one. This means that SKUs do not have a fixed place in the storage system. While replenishing, items can thus be stored in any random available pod/location instead of a pod that is defined specifically for that SKU. Advantages therefore are: the policy is very simple and pods replenishment can be done to any pod. This also increases the storage space utilisation (ibid.). However this results in a somewhat 'chaotic' storage area, where SKUs are stored randomly throughout the system making process times diverse. Another disadvantage is again the non optimal use of the available storage space due to the homogeneous locations.

3.3.3. Policy 3 - dynamic, closest

The third policy is for some the most obvious and simple. This policy involves always picking the closest available location to store an item bundle and always pick the closest available storage space to store the pod. Closest in this case can have several meanings. It can be either based on distance or on estimated travel time. Also for this policy all the locations are of the same size. Advantages of this policy include: It is very simple and it has a very efficient replenishment process. Disadvantages are: No good utilisation of the storage floor.

3.3.4. Policy 4 - dynamic, zone based

Policy four is somewhat of a combination between policy one and two. Instead of having fixed locations for every pod, this policy works with fixed zones. The zones can be separated in such a way that SKUs with the highest order frequency are located closest to the picking stations, zone A for example, and SKUs that are less often ordered can be stored further away from the picking stations, in zone C for example. See figure 3.2 for a visual representation. however in the zones themselves items can be stored anywhere, in a 'chaotic' manner. Furthermore to keep the policy more simple, homogeneous locations are chosen such as in policy one and two.

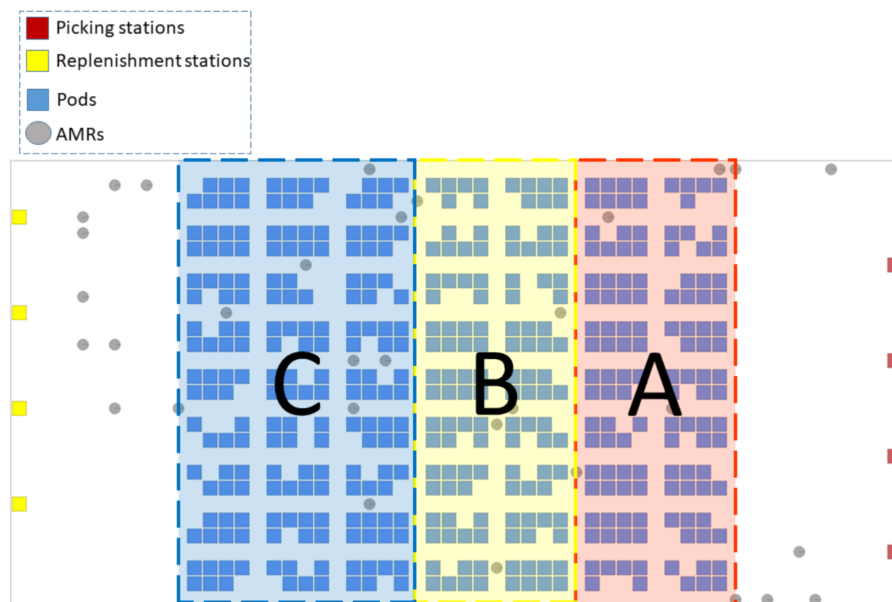


Figure 3.2: Example schematic of three different zones in a RMFS top down view.

3.3.5. policy 5 - dynamic, heterogeneous locations

The fifth policy is very similar to the second policy. The items can be stored in any random location in any random pod. There is however a distinct difference. Where the other policies all make use of locations of the same size, this policy encompasses different sized locations or heterogeneous locations. Now item bundles can be stored in storage locations that match their volume whenever possible. This increases the possibility to optimally occupy the available storage space which can be very useful for clients with a lot of diverse product volumes. For extra clarification see figure 3.3. Unfortunately this policy turned out to be very hard to implement in the used framework model. For this reason it was not completely finished within the available time frame and therefore excluded from the final experiments.

3.4. Assumptions & Boundaries

The complete system of a RMFS can get very complex. As mentioned before there are a lot of decision problems that need to be managed and some of these also interact with each other. In order to keep this research feasible it is necessary to do assumptions and set boundaries before building a model. In the following subsections the made assumptions and boundaries will be elaborated.

3.4.1. Algorithms

This research focuses for the most part on the inventory storage and the replenishment process. For this reason it is assumed that both the picking process and the path finding of the AMRs are ideally working systems. To realise this the picking process will have a fixed processing time and the path finding of the AMRs will be determined by an already existing algorithm. Also other processes such as, task allocation and order assignment will be handled by already existing algorithms.

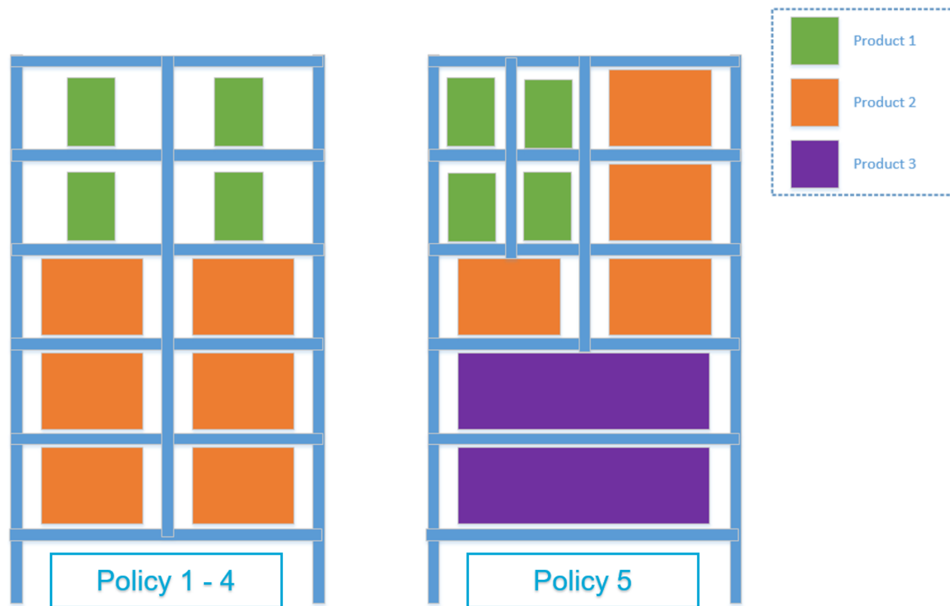


Figure 3.3: Example schematic of the pod layout for policy 5 compared to policies 1 - 4, where blue represents the pod's frame and the coloured boxes are different sized item bundles.

3.4.2. Order and replenishment generation

The generation of an order is assumed to follow a certain probability distribution. This is of course not exactly similar to the real world but this is necessary in order to test new scenarios. Further order assumptions that are made include that orders are only generated of in stock items to prevent stock out situations and that an order is always instantly replaced by a new one after it is completed. During the simulation, the generation of replenishment orders can also be paused by the system in order to avoid over-stocking the storage. For example when the storage capacity reaches a level of 90% filled, the replenishing will stop until the storage capacity reaches 75% again, after which the replenishing will continue (Merschformann et al. 2018b). Furthermore the probability of an incoming replenishment order is drawn from the same distribution as the order generation. This means that it is unknown in which exact order replenishment orders will arrive. Finally the time an employee at a pick or replenishment station needs to handle one unit is deterministic and fixed throughout the experiments.

3.4.3. Model settings

There are many assumptions that have to be made regarding the layout. These assumptions include the following: All aisles are 1 directional, The RMFS will only include 1 layer, Size of the buffer area is sufficiently large, AMRs do not have to recharge and AMR velocities and turn speeds are fixed and predetermined.

The system will furthermore be bounded in size. The floor dimensions, size of the pods and amount of pods of the new picking tower at CEVA Logistics Den Haag will be used as the guideline. The exact dimensions can be found in the next chapter. There are some other dimensional layout decisions, such as the number of replenishment/pick stations, that can not match the case study exactly however these will be kept fixed for all experiments. This way they should have a negligible effect on the final results.

4 Simulation Model Setup

The simulation model used for this research is based on a framework model by Merschformann et al. (2018b). This framework was developed in order to aid future research on RMFSs while this is still a largely new and unstudied field. It is an agent-based discrete event simulation framework. The framework is written in the 'C#' programming language and the full source code is available on: "https://www.rawsim-o.de". In this chapter the framework model itself and the changes that were made for this research will be elaborated.

4.1. Model framework

The model by Merschformann et al. (ibid.) is an extension on the simulation model 'ALPHABET SOUP' by Hazard et al. (2006), who researched multi-vehicle warehousing systems. According to the authors the model was developed to frame control and coordination issues, mainly focusing on issues in resource allocation and robot motion planning.

The model follows for the most part the steps explained in figure 2.3 in chapter 2. For each decision making problem an algorithm is selected. The model then goes step by step through the RMFS applying the algorithms to perform each task. It starts by generating the initial layout of the RMFS, also called the 'instance'. This includes everything from the size of the storage area and the location of the replenishment/picking stations to the position of the AMRs and the initial inventory of the pods. These resemble the first two decision levels and are completely determined by the user. When the instance is fully generated, the simulation can start. Now the model will start producing pick orders and replenishment orders, which can be recognised as the OA decision problems. A pick order contains one or more lines, where each line describes a certain quantity of a SKU. A replenishment order refers to a single bundle, which in this case means a fixed amount of a specific SKU. These pick and replenishment orders are then allocated to one of the stations that is the most suitable at the respective moment in time. When allocating replenishment or pick orders to stations, the model tries to bundle some of these together in order to be more efficient in the next step, pod selection. When the order is ready to be processed the model will look for a pod that fits the requirements for that task. This can be recognised as the PPS and RPS. When a pod has been handled at a station the model will find an appropriate storage location within the storage area that meets the requirements for the relevant algorithm. This can be recognised as the PSA. The final two steps of the model are allocating the above mentioned tasks to a suitable AMR and deciding how this AMR will actually travel around the RMFS. These can be recognised as the TA and PP decision problems.

4.2. Model parameters

As for all simulation models it is important that the model resembles the real world implementation as much as possible. In this case that does not only mean that the storage policies should be mimicked but that all other parameters that influence the system should be determined. In this section all the parameters that have to be determined in the simulation model will be elaborated. A more elaborate overview of the values of the parameters used in the experiments will be given in section 6. The parameters can be divided in to multiple categories:

- Layout parameters
- General system parameters
- Item generation parameters
- Control parameters

4.2.1. Layout

The layout parameters refer to the layout of the RMFS. This means first of all the size of the operational floor. In other words the complete length, width and height of the system. This should furthermore include everything from the number of pods, the number of aisles, the number of floors and the number of picking/replenishment stations. Once this has been determined it is important to choose the number of AMRs. This decision can have a large effect on the efficiency of different policies in the system. For this reason the number of AMRs will be discussed in chapter 6.

Each of the above mentioned elements includes some relevant characteristics. First of all the size of the system. In the model this can be mainly described by the number of horizontal and vertical aisles. Between these aisles the pods will then be generated. In this manner 'blocks' of pods will be generated. The length of these blocks is also another parameter that needs to be determined. The blocks are however always two wide, to represent a single-deep scenario. Another parameter that needs be be set is the number of floors. Some RMFSs or manual warehouse systems make use of multiple floors to increase the space utilisation of the work floor. However for this research only 1 floor will be used. The height between each floor is therefore also not important. One final parameter is important for the actual size of the system and that is the percentage of the storage spaces that will actual be occupied by pods. Some spaces in the system are left empty to create places for bots to go idle. This parameter is given as a percentage of the total available storage spaces.

After the size of the system is determined, the pods require some more information. Most importantly the capacity of the pods. This will determine how many items can be stored inside each pod. In the model this capacity is described as a 1 dimensional number that represents the total volume or weight capacity of the pod. Another parameter that is important is the time it takes to pick up a pod or have it set down by an AMR. This can have a small effect on the speed of the system. Finally the pod radius has to be determined. These last two parameters however will not have a major effect on the experiments and are just set as the general settings of the framework model.

Now that it is determined where the SKUs are stored and how many can be stored, it is time to set the number of stations and their characteristics. Most important is of course the number of pick stations. This is one of the main drivers that determines the maximum throughput that an RMFS can deliver. However the system also needs to contain enough products to keep up with the demand that is required from the picking stations, which is what the replenishment stations are needed for. These stations do need more information to work properly. The time it takes to pick an item and process it and the time it takes to store a bundle of items inside a pod are two very important characteristics of these stations. These therefore also need to be determined for the simulation model. The last station parameter is the station capacity. For both type of stations it should be determined how many items can be temporary stored at the station. In case of the replenishment station this temporary storage is required while bundles have to wait for the right pod to arrive. In the case of the picking stations some products might have to wait a bit until all the right products to form the order have arrived.

These parameters are based on the case study as much as possible. However some parameters can either not be adjusted or not be perfectly similar to the case study due to the nature of the framework. Nevertheless since the layout parameters are fixed throughout the experiments they should not interfere with the result in a significant manner.

4.2.2. System

Now that the layout of the model has been determined, the simulation settings should be defined. These setting describe mostly general information about the simulation such as: the length of the simulation (simulation horizon), the seed used for the generation of the layout and the orders, how long the simulation should run before starting to record data (warm up period), etc. The simulation time should be set in such a way that

there is enough time for the system to run through multiple replenishment cycles. A replenishment cycle in this case refers to the pausing and restarting of the generation of replenishment orders. This happens when the total capacity of the system reaches a certain percentage. When it gets too high the replenishment orders will stop generating, when it gets too low the replenishment orders will start generating again. The seed used for every simulation should vary to ensure that none of the experiments are exactly the same. A warm up period is not necessary if the simulation horizon is long enough. The latter has been determined by analysing test results and noticing that a warm up period is not needed. The framework model also includes some extra settings that are specific to the model, the output of the model and the visual representation of the model, these will not be discussed further. A full list of the system parameters that are used for the experiments and a short description can be seen in table C.3.

4.2.3. Item generation

The framework allows the model to generate a list of items (SKUs) that are consequently used for the generation of orders during the simulation. In order to generate the list, multiple parameters and probability distributions have to be set. These parameters determine first of all how many SKUs the list will include and furthermore determine for each SKU aspects such as: order frequency, weight (which is a replacement for a SKU's size/volume), bundle size and pick-hit rate. In the table it can be seen that not all parameters are filled in, this is because they are not used.

After this list has been created it is also necessary to set parameters for the generation of orders. Parameters such as: the number of lines in an order and the minimum and maximum amount of items per line. A 'line' in e-commerce jargon usually refers to the order of an arbitrary amount of 1 single SKU (Koster et al. 2007). If, for example, in one order four items a, one item b and 2 items c are ordered, this order contains 3 lines. A line in this case could refer to 'four items a'. The model then combines the information corresponding to all the SKUs and order parameters to generate orders during the simulation. A complete list of the order parameters can be seen in tab C.5. Again some parameters are not filled in since they are not used.

Even though the model provides a very intuitive and simple manner of generating orders for a simulation, this research requires that third party order data can be used as an input for the model. Therefore the input data for this research has been acquired from a real client of the case study. As mentioned the input for the simulation model requires a few specific product characteristics of each SKU in the system. These are: A random identification number, the order frequency, a weight number to represent the volume/weight of the product, the bundle size and pick-hit rate. First of all the random identification number ensures that the model can recognise each unique product. Then the model needs to know the order frequency and weight of each product. It uses the order frequency for two things, the generation of orders and for generating the initial inventory. The initial inventory is however also dependant on the weight of the SKUs. It needs this information since each pod is limited in capacity. Also the bundle sizes of each product are important. In a real life RMFS the replenishment often occurs using boxes (bundles) that contain a certain amount of a specific SKU. For example a box filled with 10 pairs of shoes. This means that during generation and replenishment, SKUs are assigned to a pod only as a complete bundle of items. The items can however later be picked in smaller amounts than the bundle size. The last piece of information that can be added to the model is the pick-hit rate or co-weight. In the model this is represented by a two dimensional plane on which the SKUs are randomly distributed. The distance between two products on this plane then determines their respective co-weight. Further details on the exact analysis of the input data can be found in section 4.3.

4.2.4. Control

The controller parameters of the model include some of the most important parameters for this research. In the controller configuration all the control methods can be specified. I.e. the path planning algorithm, task allocation algorithm, the item storage algorithm, the pod storage algorithm, etc. In the case of this research specifically the item storage configuration and pod storage configuration are of most importance. The item storage configuration determines which pod is requested whenever an item bundle is ready for replenishment and thus determines in which pod an item is being stored. The pod storage configuration is the algorithm that determines to which storage location a pod should return after it has been replenished or picked. This algorithm thus determines how the storage floor gets utilised. Since these configurations have countless of variations, the chosen configurations for the examined policies mentioned in chapter 3 will be elaborated in the following subsections. Two other parameters that are also necessary for the chosen policies

that are going to be investigated are storage zones and the pod layout. Not all the parameters and control configuration that are required for the chosen storage policies are by default programmed in to the model framework. Some have to be added manually. How this is done can be read in the next subsections.

Each storage policies thus consists of two algorithms. One for the item storage part and one for the pod storage part of the system. The item storage policy normally consists of two main functions:

1. SelectPodForInitialInventory
2. DecideAboutPendingBundles

The first function is responsible for the distribution of items during the initial generation of the simulation. The second function determines where items are stored while the simulation is running. Policy 4 has a few extra functions that are necessary for the determination of the zones, which is also used for the initial generation in policy 1.

The pod storage policy usually contains only 1 main function:

- GetStorageLocationForPod

This function does exactly what it says. It determines the best location for the current pod to be stored. Also here the pod storage manager for policy 4, thus also 1, contain some extra functions to account for the zones.

Below a more in depth explanation of the modelling process of each policy will be given together with a pseudo code representation of the most important functions. Some of these functions will refer to some default components from the framework model. These will not be elaborated further in this research. For the complete model and a guide on how to use the framework 'RAWSimO' please be referred to `https://github.com/Mitchel703/RAWSimO-Test`.

4.2.5. Policy 1

This policy is characterised by its simplicity as described in the previous chapter. The main control mechanisms are: Pods are always retrieved and sent back to/from the same storage location and each product has fixed pods to which they are assigned. To achieve this there are two requirements that have to be met. The first one is that items can only be replenished to a pod where that item is already present, unless there are no pods that contain the specific item. This was achieved by a tracker that investigates which item descriptions are present within the location of the pod, based on its coordinates, radius and height. All pods that are available for replenishment are then filtered based on the presence of that specific item. After which these pods are ordered from the ones that include the least amount of the specified item to the one with the most. The pod with the least amount is then selected for the replenishment action of this specific item bundle. The point of this policy is that a floor manager can put items with a high order frequency closer to the picking stations than items with a lower order frequency. This means that the model should also be able to do this before the simulation starts. This is achieved with a similar method as the zone based policy which will be elaborated further on. The main functions of the item storage and pod storage policies of policy 1 in pseudo code can be seen in algorithms 1 and 2.

Algorithm 1 Policy 1 - Fixed Item Storage

```
function SelectPodForInitalInventory(Instance pods, Bundle)
    set chosenPod equal to call ChoosePod(Bundle)    ▷ The definition of function ChoosePod is given in policy 4
    if chosenPod is equal to Null then
        print "Could not find a pod for the given bundle"
    else
        return chosenPod
    end if
end function

function DecideAboutPendingBundles
    for all bundle in pendingBundles do
        set desiredStorageClass equal to DetermineStorageClass(bundle)
        set chosenPod equal to the list of all pods in the desiredStorageClass
            Where FitsForReservation(Pod, Bundle) is True
            OrderBy if pod already contains the bundle description or not, 0 if True OR 1 if False
            ThenBy the amount of bundle items currently contained
            Select The first pod from the list or return Null

        if chosenPod is not equal to Null then
            call AddToReadyList(Bundle, chosenPod)
        else
            continue
        end if
    end for
end function
```

Algorithm 2 Policy 1 - Fixed Pod Storage

```
set positions equal to the initial locations of all pods

function GetStorageLocationForPod(Pod)
    return call positions(Pod)
end function
```

4.2.6. Policy 2

The main mechanism of this policy is randomness. In other words pods are allowed to be retrieved and sent back to/from any position in the storage area. Products are therefore also spread out across multiple pods, located in different areas of the storage area. In terms of the model this means that both the item storage and the pod storage can happen completely random. The only requirement is that, similar to policy 1, pods should have enough storage space. One extra function that is added to this algorithm is that the model prefers a pod that is on the same tier (floor level). The pseudo code can be read in algorithms 5 and 6.

Algorithm 3 Policy 2 - Random Item Storage

```
function SelectPodForInitalInventory(Instance pods, Bundle)
    return a list of all pods
      Where FitsForReservation(Pod, Bundle) is True
      OrderBy randomize all pods in the list
      Select The first pod from the list
end function

function DecideAboutPendingBundles
    for all bundle in pendingBundles do
        set chosenPod equal to the list of all pods
          Where FitsForReservation(Bundle) is True
          OrderBy randomize all pods in the list
          Select The first pod from the list

        if chosenPod is not equal to Null then
            call AddToReadyList(Bundle, chosenPod)
        end if
    end for
end function
```

Algorithm 4 Policy 2 - Random Pod Storage

```
function GetStorageLocationForPod(Pod)
    if PreferSameTier is True && Amount of tiers is higher than 1 then
        set locationsOfThisTier equal to all unused pod locations on that tier
        if locationsOfThisTier contains any then
            return a random value from the list locationsOfThisTier
        end if
    end if

    if There are zero unused pod locations then
        print "There was no suitable storage location on this tier"
        return a random location of all the unused pod locations
    end if
end function
```

4.2.7. Policy 3

Policy 3 seems very simple at first glance, however turning this in to code is a bit more difficult. The main mechanism of this policy is that pods and storage locations are chosen based on either distance or travel time from a picking or replenishment station. In terms of the model this means that the algorithm has to calculate the distance/travel time from a station to each individual pod, every time a pick or replenishment order is processed. In this case it is chosen that this distance is calculated using a Euclidean distance. The pseudo code for this policy looks as follows:

Algorithm 5 Policy 3 - Closest Item Storage

```
function SelectPodForInitalInventory(Instance pods, Bundle)
    return a list of all pods
        Where FitsForReservation(Pod, Bundle) is True
        OrderBy randomize all pods in the list
        Select The first pod from the list
end function

function DecideAboutPendingBundles
    for all bundle in pendingBundles do
        set inputStation equal to all inputStations
            OrderBy randomize all inputStations
            Select The first inputStation from the list

        set chosenPod equal to the list of all pods
            Where FitsForReservation(Bundle) is True
            OrderBy WrongTierPenalty plus call CalculateEuclidean(inputStation, pod, WrongTierPenalty)
            Select The first pod from the list

        if chosenPod is not equal to Null then
            call AddToReadyList(Bundle, chosenPod)
        end if
    end for
end function
```

Algorithm 6 Policy 3 - Closest Pod Storage

```
function GetStorageLocationForPod(Pod)
    set minDistance is equal to infinite
    set bestStorageLocation to null
    set podLocation equal to call GetClosesWayPoint(pod.Tier, pod.X, pod.Y)

    for all storageLocations in UnusedPodStorageLocations do
        distance is equal to call CalculateEuclidean(pod, storageLocation, WrongTierPenalty)
    end for

    if distance is lower then minDistance then
        set minDistance equal to distance
        set bestStorageLocation equal to storageLocation
    end if

    if bestStorageLocation is equal to null then
        print "There was no suitable storage location for this pod"
    end if

    return bestStorageLocation
end function
```

4.2.8. Policy 4

The main mechanism of this policy involves storage zones. This means that items are only allowed to be stored in a specific area within the complete storage area based on their order frequency. Within these zones the storage will be completely random. The storage area will be divided in to three zones. In the model this will be achieved by assigning storage locations to a certain class (class 0, 1 and 2). The locations will be ordered according to their distance from the picking stations, where class 0 will be closest to the picking stations and class 2 furthest from the picking stations. During initialisation of the simulation, pods receive the same class as the location that they are generated on. The capacity of the classes relative to the total capacity of the system can be seen in table 4.1 which follows the margins from Lamballais et al. (2017b). As can be seen from the pseudo code below, this policy involves a few extra functions, the ChoosePod and ChoosePodByConfig function. These functions are responsible for choosing storage locations that are of the same class as the item bundle that has to be stored, but will also make sure that another suitable storage location is picked whenever it is not possible to store an item bundle in the same class. The pseudo codes of this policy can be seen in algorithms 7 and 8.

	Relative capacity
Class 0	20%
Class 1	30%
Class 2	50%

Table 4.1: Table showing the relative capacities of the storage classes

4.2.9. Policy 5

This policy revolves mainly around the use of different sized locations inside the pods. The pods can however still be retrieved and sent back to/from any storage location but items should now prioritise locations that closely match their bundle size/weight to increase efficient use of the total storage. To implement this in the framework model, a switch had to be made from a singular value for the pod capacity to a list of compartments or locations with each its own capacity for the total pod's capacity. However the pod capacity turned out to be so far integrated in the model as a singular value that changing this correctly took too much time. As mentioned before it was therefore chosen to exclude this policy from further use in the research. However eventually it does look like a functioning model has been achieved. The model can be viewed from: `https://github.com/Mitchel703/RAWSimO-Mitchel`. It has to be mentioned however that this model still lacks a lot of verification and validation.

4.3. Input data tool

For any simulation applies that the main goal of using input data is to drive the simulation according to Ungureanu et al. (2005). There are three steps that need to be taken before one can use input data in a simulation model. These are:

- Collecting input data
- Analysing input data
- Using analysed input data

One of the goals of this research is that it should be adaptable. Meaning that it can be used for any RMFS and any respective client. For this reason it is important that order data can be formulated in a consistent manner. To achieve this a tool was developed in the Python program which transforms historical order data in to a usable text file. This text file can then in turn easily be loaded in to the simulation to adapt the model relevant to the specific client.

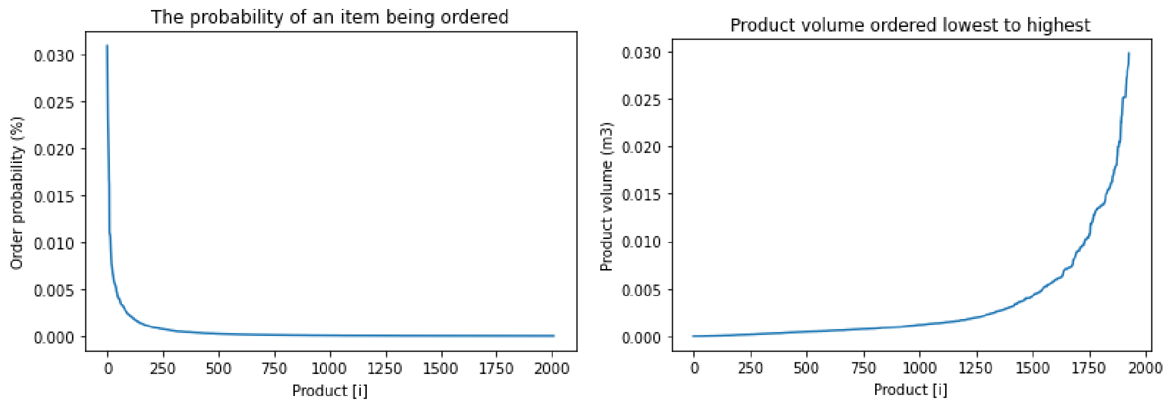
To produce this text file a few steps are taking within the tool. First of all the data is filtered to only include a specific time period. Often times such order data includes months and months of orders, which is not only unnecessary but also computationally very heavy. In this case it was chosen to limit the data to one week of

Algorithm 7 Policy 4 - Zone Item Storage

```
function SelectPodForInitialInventory(Instance pods, Bundle)
    set chosenPod equal to ChoosePod(Bundle)
    if chosenPod is equal to Null then
        print "Could not find a pod for the given bundle"
    else
        return chosenPod
    end if
end function

function DecideAboutPendingBundles
    for all bundle in pendingBundles do
        set chosenPod equal to ChoosePod(Bundle)
        if chosenPod is not equal to Null then
            call AddToReadyList(Bundle, chosenPod)
        end if
    end for
end function

function ChoosePod(Bundle)
    set DesiredStorageClass equal to call DetermineStorageClass(bundle)
    set CurrentClassLow equal to DesiredStorageClass && CurrentClassHigh equal to DesiredStorageClass
    while True do
        if CurrentClassLow is lower than ClassCount then
            set chosenPod equal to call ChoosePodByConfig(CurrentClassLow, Bundle)
        end if
        if chosenPod is not Null then
            break
        end if
        if CurrentClassHigh is higher or equal than zero && not equal to CurrentClassLow then
            set chosenPod equal to call ChoosePodByConfig(CurrentClassHigh, Bundle)
        end if
        if chosenPod is not Null then
            break
        end if
        CurrentClassLow + 1 & CurrentClassHigh - 1
        if CurrentClassHigh is lower than zero && CurrentClassLow is higher than ClassCount then
            break
        end if
    end while
    return chosenPod
end function

function ChoosePodByConfig(classId, Bundle)
    set chosenPod equal to Null
    if LastChosenPod is of the same Class && FitsForReservation(Bundle) is True then
        set chosenPod equal to LastChosenPod
    else
        Create a list of all pods within the Class
        where where FitForReservation(Pod, Bundle) is True
        OrderBy (CapacityInUse + CapacityReserved) / Capacity
        Select The first pod from the list
        if chosenPod is not equal to Null then
            call AddToReadyList(Bundle, chosenPod)
        end if
    end if
end function
```

Algorithm 8 Policy 4 - Zone Pod Storage

```
function GetStorageLocationForPod(Pod)
    set chosenLocation equal to call ChooseStorageLocation(pod)
    if chosenLocation is equal to Null then
        print "There was no suitable storage location for this pod"
    else
        return chosenLocation
    end if
end function

function ChooseStorageLocation(pod)
    set desiredStorageClass equal to call DetermineStorageClass(pod)
    set CurrentClassLow equal to equal to DesiredStorageClass && CurrentClassHigh equal to DesiredStor-
ageClass
    set chosenStorageLocation equal to Null

    while True do
        if CurrentClassLow is lower than ClassCount then
            set chosenStorageLocation equal to call GetClassStorageLocations(CurrentClassLow)
               Where call IsStorageLocationClaimed is false
               OrderBy distance to the output stations
               Select First from the list
        end if
        if chosenPod is not Null then
            break
        end if

        if CurrentClassHigh is higher or equal than zero && not equal to CurrentClassLow then
            set chosenStorageLocation equal to call GetClassStorageLocations(CurrentClassHigh)
               Where call IsStorageLocationClaimed is false
               OrderBy distance to the output stations
               Select First from the list
        end if

        if chosenPod is not Null then
            break
        end if

        CurrentClassLow + 1 & CurrentClassHigh - 1

        if CurrentClassHigh is lower than zero && CurrentClassLow is higher than ClassCount then
            print "There was no storage location available"
        end if
    end while

    return chosenStorageLocation
end function
```



(a) The probability of each product from the case study's input data

(b) The volume (m^3) of each product from the case study's input data

Figure 4.1: Probability and volume data of the acquired order history.

orders. The next step is to then group all the orders based on their SKU name or number. This ensures that for every SKU the total amount of items sold gets added up and that every row now only contains one unique SKU. In order to get the order probability of each SKU, the total orders of a SKU should be divided by the total number of sold items in the respective time frame. The simulation model however uses probability weights instead of actual probabilities. Therefore only the ratio in order probability between the different SKUs is important. In this case it was chosen to multiply the order probabilities by 100 to increase the readability of the probabilities. Now the weights/volumes, bundle sizes and co-weights should be added for every SKU. After this is done the input text file for the simulation can be written. This is simply a summation of the data for every SKU.

For the case study, order probability and volume data was widely available. These can also be seen in figure 4.1. Note that the products on the horizontal axis do not match the same order between the two graphs. Bundle size and co-weight data are however somewhat scarce. For this reason some assumptions have to be made. First of all the bundle sizes are determined based on the size of a product. For larger items it was decided that at least 5 of them should be able to fit in a single location. In this case locations have a volume of 0.15 m^3. This means that products with a volume bigger than 0.03 m^3 are not allowed to be stored in the system and are thus filtered from the input data. Then based on the average size of the products it was decided to determine the bundle sizes in the following manner:

- $V[i] < 5*10^{-4} m^3 \Rightarrow B[i] = \frac{0.15m^3}{5*10^{-4}m^3} = 300$
- $5*10^{-4} \geq V[i] < 1*10^{-3} m^3 \Rightarrow B[i] = \frac{0.15m^3}{1*10^{-3}m^3} = 150$
- $1*10^{-3} \geq V[i] < 5*10^{-3} m^3 \Rightarrow B[i] = \frac{0.15m^3}{5*10^{-3}m^3} = 30$
- $5*10^{-3} \geq V[i] < 1*10^{-2} m^3 \Rightarrow B[i] = \frac{0.15m^3}{1*10^{-2}m^3} = 15$
- $1*10^{-2} \geq V[i] < 3*10^{-2} m^3 \Rightarrow B[i] = \frac{0.15m^3}{3*10^{-2}m^3} = 5$

Where:
V[i] = The volume of product i
B[i] = the bundle size of product i

Unfortunately co-weight data is still very rare. This is often a dynamic characteristic and hard to put an actual number on. For this reason it has been decided not to take pick-hits in to account for this research. The complete python script that is used for the analyses and translation of the input data can be seen in Appendix B.

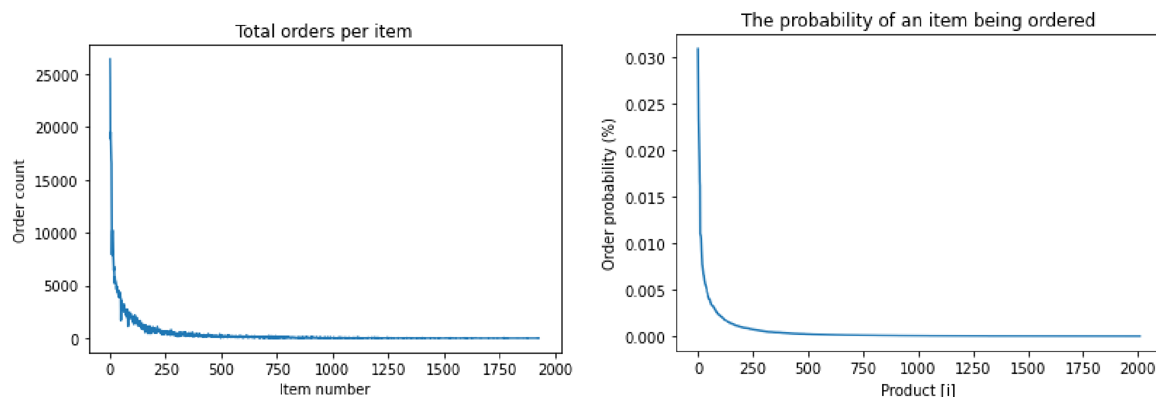
5

Verification & Validation

The framework model in itself has already been verified and validated in multiple researches and by the developer himself (Merschformann et al. 2018b)(Merschformann et al. 2019). However the new changes made to the model should still be verified and validated as well as the tool to translate input data. To do this for each of the changes some experiments are performed to ensure that each works properly. The model has a build in Graphical User Interface (GUI) which can be perfectly used to analyse the model during the experiments. Furthermore the input data shall be compared to measurements that the model performs during the simulation.

5.1. Input data

Currently the main source for input data is existing order data from a manual picking system at the case study. The tool that has been developed to convert this raw data in to usable input data has to be verified and validated. The simulation model is able to monitor how often SKUs are ordered in total over the simulation horizon. For this verification experiment a simulation horizon of 7 days was chosen. This total order count is then plotted for each SKU along the vertical axis. The SKUs are shown on the horizontal axis. This graph can then be compared to a graph showing the order distribution of each SKU that was generated using the tool. The shape of the graphs should now look similar. Both graphs can be seen in figure 5.1. It is clear that both graphs indeed show a very similar shape. This means that the order probabilities that were used as input are well translated in to the model. The weights and bundle sizes that belong to each item are exactly copied in to the model, therefore these do not need have to be verified.



(a) The total amount ordered per SKU over a simulation horizon of 7 days. (b) The order distribution as generated by the input data model.

Figure 5.1: Verification of the input data tool.

5.2. Verification - policy 1

The first policy ensures that products can be placed in fixed locations on the storage floor based on their order frequency. This means that SKUs are always replenished to the same pods and that the pods always return to their specific storage location. In order to test this, two different effects have to be monitored. First of all the spread of products on the storage floor at the start of the simulation should be checked. For this policy it is important that SKUs with a higher ordering frequency are placed closer to the pick stations than products that are ordered less often. The second effect is that this spread of products should remain as constant as possible over any amount of time.

To create a simulation experiment that is able to verify these effects multiple simulation settings should be determined. First of all a simulation is set up with the appropriate algorithms for policy 1. Secondly a relatively small product data file, containing 10 SKUs with each distinctly different order probabilities, is used as input data for the simulation. The SKU characteristics of the input data can be seen in Table 5.1. To ensure that the test remains simple, the capacity of each pod will be reduced to 17, in order to only contain a maximum of 3 item bundles. Other settings of the simulation or not so important for this verification since they do not have an effect on the position of the items. In this case the initial settings of the framework are used since these have been verified in other literature as mentioned before.

	Weight	Bundle size	Order weight
SKU 0	1.0	5	10.0
SKU 1	1.0	5	9.0
SKU 2	1.0	5	8.0
SKU 3	1.0	5	7.0
SKU 4	1.0	5	6.0
SKU 5	1.0	5	5.0
SKU 6	1.0	5	4.0
SKU 7	1.0	5	3.0
SKU 8	1.0	5	2.0
SKU 9	1.0	5	1.0

Table 5.1: The SKU characteristics used as input data for the verification of policy 1.

After all the settings have been determined the simulation gets generated using the GUI. Now from each storage zone 2 pods are checked for its contents. This has two reasons. Firstly to see whether or not items are actually in the right zones based on their probability and secondly to be able to track the contents and locations over the duration of the simulation. The simulation is then executed with a simulation horizon of 24 hours. After the simulation has finished, again the locations and the contents of the same 6 pods are recorded and compared against their initial location and content to see if they remained similar or not. The 6 pods, their initial position, final position, initial contents and final contents can be seen in Table 5.2.

Looking at Table 5.2 a few things can be noticed. The contents of pods that are generated in zone 1 consist only of product 0 and 1. This is as expected since the algorithm is defined in such a way that 20% of the items with the highest order frequency should be assigned to the zone closest to the picking stations, which is zone 1 in this case. Zone 2 should then contain the next 30% of items based on the order frequency. Since there are only 10 items used as input data this should mean that items 2, 3 and 4 should be generated in zone 2. As can be seen from the table this is pretty much the case with one exception (product 1 in pod 416). Lastly zone 3 should only contain the remaining items, which is also accurate except for one exception (product 4 in pod 351). These exceptions can however be explained by some decisions that are made within the algorithm. When a bundle of items is generated, the algorithm will assign it a desired zone. It will then look for an ap-

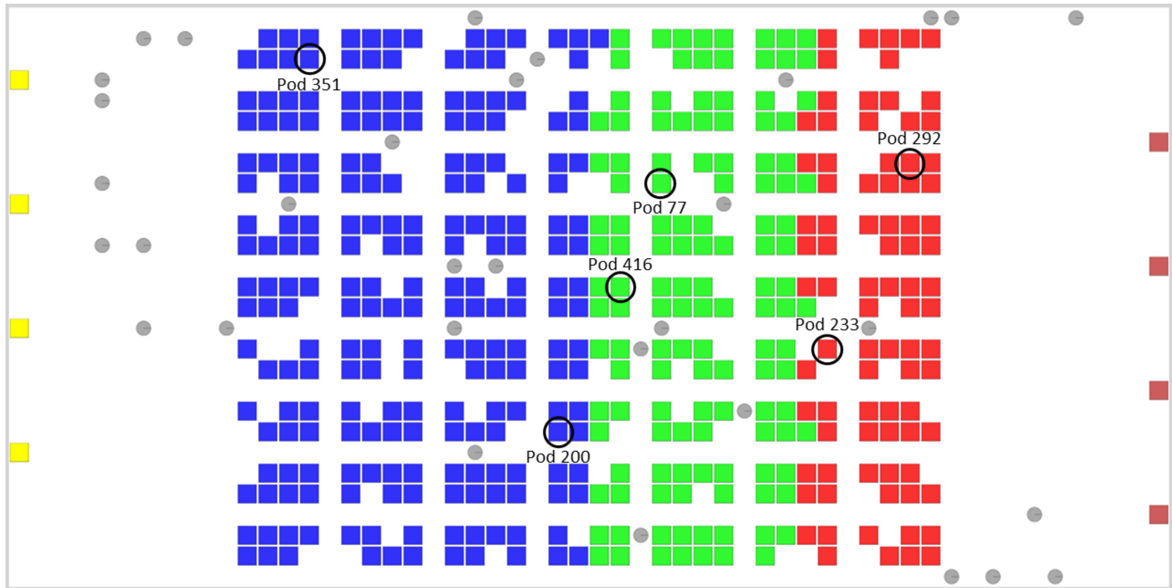
Pod	Storage zone	Initial position (X/Y)	Final position (X/Y)	Initial contents (Product No. (amount))	Final contents (Product No. (amount))
292	1	43.50/20.50	43.50/20.50	0(10) 1(5)	0(15)
233	1	39.50/11.50	39.50/11.38	0(5) 1(10)	Empty
416	2	29.50/14.50	53.50/10.50	1(5) 3(5) 4(5)	2(5) 4(5)
77	2	31.50/19.50	31.50/19.50	2(5) 3(5) 4(5)	2(16)
351	3	14.50/25.50	14.50/25.50	4(5) 5(10)	3(5) 9(10)
200	3	26.50/7.50	26.50/7.50	5(5) 7(5) 8(5)	0(15)

Table 5.2: Monitored 6 pods and their initial/final location/contents, applying policy 1.

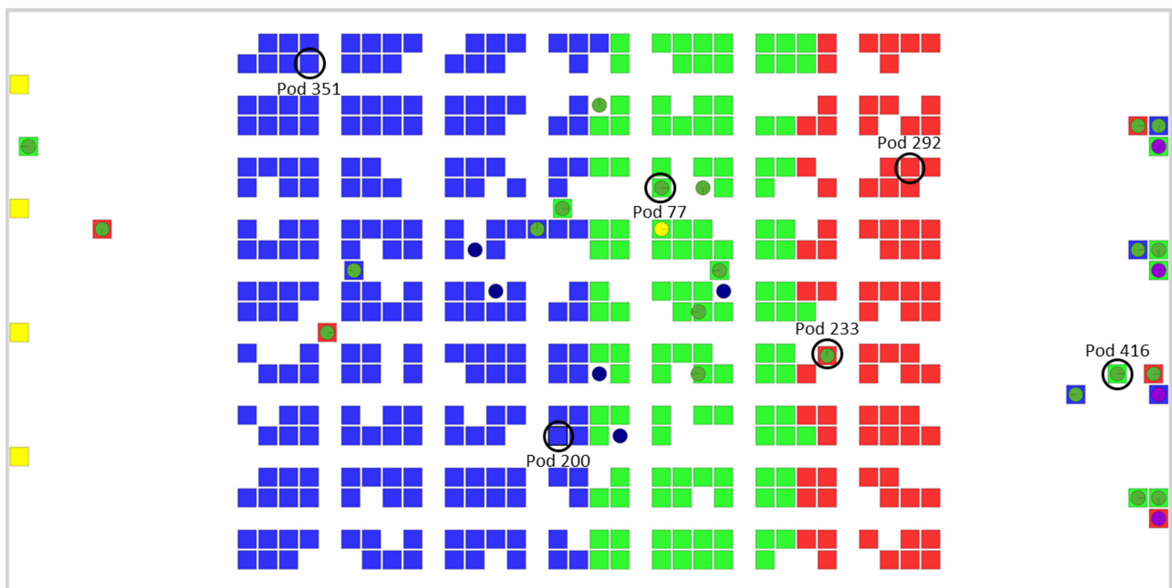
propriate pod in that specific zone to allocate the bundle to. However if this is not possible for some reason, i.e. the zone is full, it will allocate the bundle to a neighbouring zone. This is done to prevent bundles that can not be placed in their specific zone from clogging up the input stations capacity.

Next, when comparing the initial contents of the pods to their final contents, it can be seen that these are a bit less consistent. Some similarities can be found in the type of products, such as in pod 292 and pod 77. However other pods show a completely different content at the end of the simulation than in the beginning. This is probably also a result of items no longer fitting in their respective zone and thus being allocated to another. That is most likely also why product 0 is located in pod 200 at the end of the simulation, which is a zone 3 location but a zone 1 product.

Finally in the table not all initial and final locations are similar, however from the GUI it was clear that the pods, where the final location was different then their initial location, were being moved by an AMR when the simulation finished. A good example is pod 233 which was just about to leave its fixed location, which is why the y coordinates are slightly different between the initial and the final location of this pod. The locations of the pods are thus indeed fixed. This can also be seen in the screenshots of the GUI from the initial generation (Figure 5.2a) and after the simulation had finished (Figure 5.2b).



(a) The locations of the 6 pods at the start of the verification simulation using policy 1.



(b) The location of the 6 pods after the verification simulation ended using policy 1.

Figure 5.2: First verification tests for policy 1.

As mentioned the initial and final contents of the pods are not very similar when testing this policy. The main problem being that items that belong to higher order frequency zones end up in lower frequency zones due to a lack of space. This is however not the idea of this policy. This policy is supposed the be absolutely fixed. Meaning that items can only be stored in pods where, beforehand, was determined that those items belong. For this reason some fixes had to be made to the model's algorithm and further verification should be performed. The fix that was added to the policy is that now items will first try to be assigned to a pod in the relevant zone which already contains the respective item. Only if this is not possible, an empty pod in the relevant zone will be selected and else the item will just wait at the replenishment station until an appropriate pod becomes available. This does however have some effect on the efficiency of the replenishment. In order

to counteract this, the way the initial simulation gets generated is changed as well. Instead of assigning a fixed percentage of the items to a certain zone, the items now get divided according a weighted demand. This weighted demand is the average of an item's relative order probability and relative weight. In this way it is possible to leave some capacity empty in each zone, which should make it easier for the replenishment side to always find a suitable pod in the right zones. The formula for the weighted demand looks as follows:

$$WeightedItemDemand = \frac{OrderProbabilityWeight}{TotalProbabilityWeight} + \frac{Weight}{overallWeight} \tag{5.1}$$

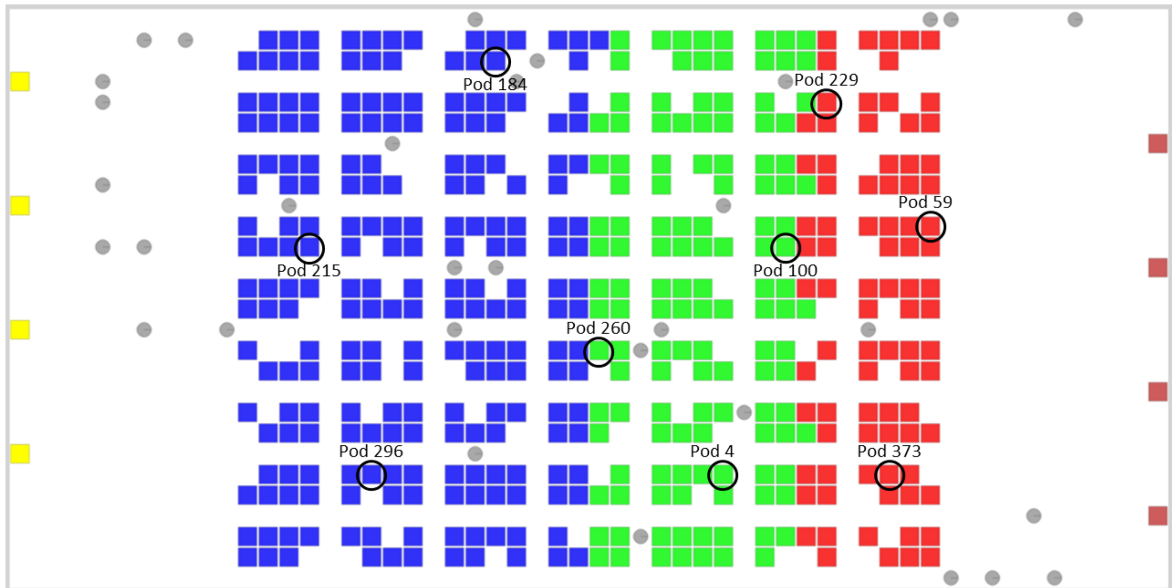
Now the same verification test is repeated but this time 9 pods in total are selected, 3 randomly from each zone. This is done to increase the overall capability of accessing the distribution of items and whether or not they remain in their original assigned pods and/or zones. The results of the test can be seen in Table 5.3.

Pod	Storage zone	Initial position (X/Y)	Final position (X/Y)	Initial contents (Product No. (amount))	Final contents (Product No. (amount))
59	1	44.50/17.50	44.50/17.50	0(5) 1(10)	0(15)
373	1	52.50/5.50	52.50/5.50	0(10) 1(5)	0(14)
229	1	39.50/23.50	39.50/23.50	0(5) 1(10)	0(5) 1(10)
100	2	37.50/16.50	37.50/16.50	2(5) 3(5) 4(5)	2(5) 3(5) 4(5)
4	2	34.50/5.50	34.50/5.50	2(5) 4(10)	4(15)
260	2	28.50/11.50	28.50/11.50	1(5) 3(10)	2(15)
184	3	23.50/25.50	23.50/25.50	5(5) 8(10)	8(16)
296	3	17.50/5.50	17.50/5.50	Empty	6(14)
215	3	14.50/16.50	14.50/16.50	Empty	Empty

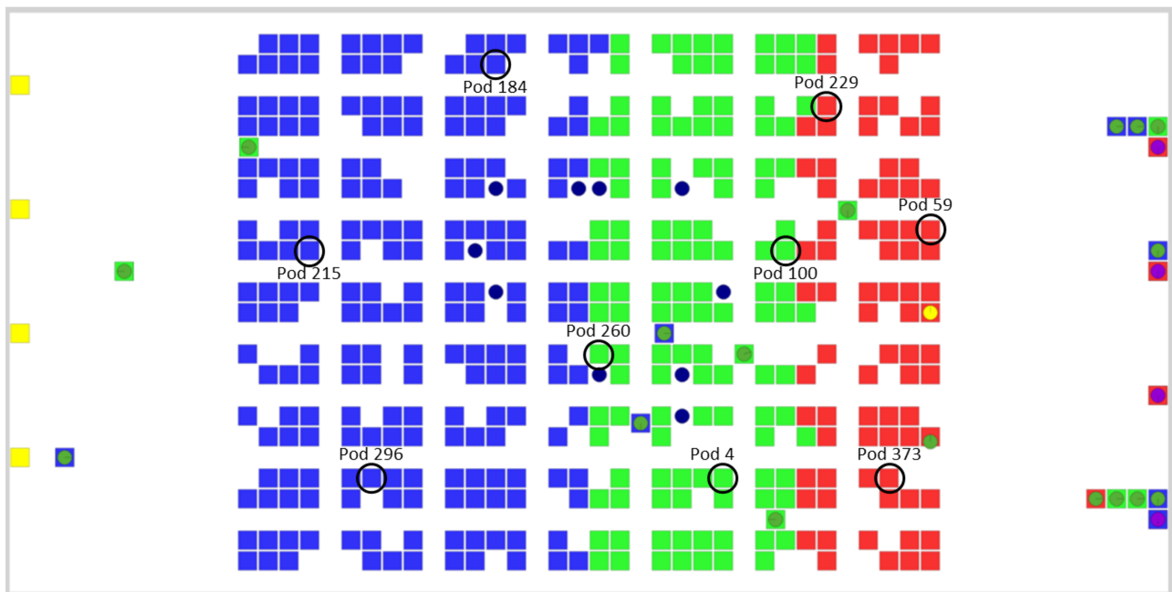
Table 5.3: Monitored 9 pods and their initial/final location/contents, applying policy 1 with its fixes.

From the table it is clear that the locations of the pods are still perfectly consistent. So it can concluded that this part of the policy works as intended. Also the distribution of the items over the zones seems to be as it is supposed to be. There is one exception and that is item 1 being located in pod 260, which is in zone 2. This can however be explained by the fact that the generation of the simulation has not changed. Only the decisions made during the run of the simulation. So in this case item 1 was still moved to zone 2 because zone 1 was probably full.

Looking at the final contents of the pods it is clear to see that most pods contain items similar to what what their initial content was. Also here there are some exceptions. Pod 260 and 296 both contain items that were not part of their original content. However in both cases the 'new' item they contain does belong to the respective zones of the pods. This can happen when no other pod that already contains the relevant item is available so another pod from the same zone, that has capacity left, is selected for the storage of the item. In figures 5.3a and 5.3b again the initial and final states of this verification simulation can be seen.



(a) The locations of the 9 pods at the start of the verification simulation using policy 1.



(b) The location of the 9 pods after the verification simulation ended using policy 1.

Figure 5.3: Second verification tests for policy 1.

With the added fixes policy 1 seems to work as intended now. The only drawback is the fact that there is some effect on the efficiency of the replenishment process due to some bundles having to be kept idle for longer at the replenishment stations waiting for a suitable pod.

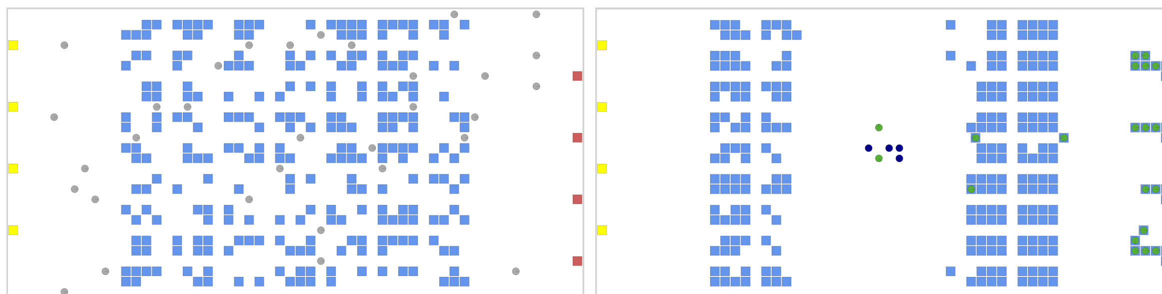
5.3. Policy 2

The second policy consists solely of existing algorithms in the framework model that have already been used and investigated by Merschformann et al. 2018a. It can therefore be assumed that this policy does not need additional verification and validation.

5.4. Verification - policy 3

The third policy also consist of algorithms that were already present in the framework model. However during initial testing some bugs were discovered. These have been fixed manually but some verification should be performed. This policy relies on the model always searching for the closest available storage location for either an item bundle or a pod. In order to test this the 5 first replenishment actions from a simulation are followed to see whether or not the closest storage locations are actually used. The system settings will remain similar to the settings used for the verification of policy 1, however the storage area is reduced to make sure that it is easier to follow by eye. Furthermore the relevant algorithms will be changed to fit policy 3. The initial simulation can be seen in Figure 5.5a. Here the pods that have room left for at least one extra bundle and are in the closest section of pods are circled and given a number for recognition. The green number represents the remaining capacity that the pod has. Now the simulation is started until the first 5 pods have visited a replenishment station. Figure 5.5b shows the locations of the pods after these 5 replenishment actions. The numbers in red specifies pods that have been replenished and how much capacity they have left. It furthermore indicates to which storage location the pods were returned after the replenishment. From these results it is clear to see that pods return to storage locations that are as close as possible to the input station. One interesting decision however can be noticed. That is the fact that pod 2 has been replenished whilst pod 6 has not been replenished yet. Pod 6 is clearly closer to the input station than pod 2. This can be explained by the fact that the model also prioritises pods in which it can store more than one bundle. The exact order of which pod was replenished first is the following: 3 - 4 - 7 - 5 - 2.

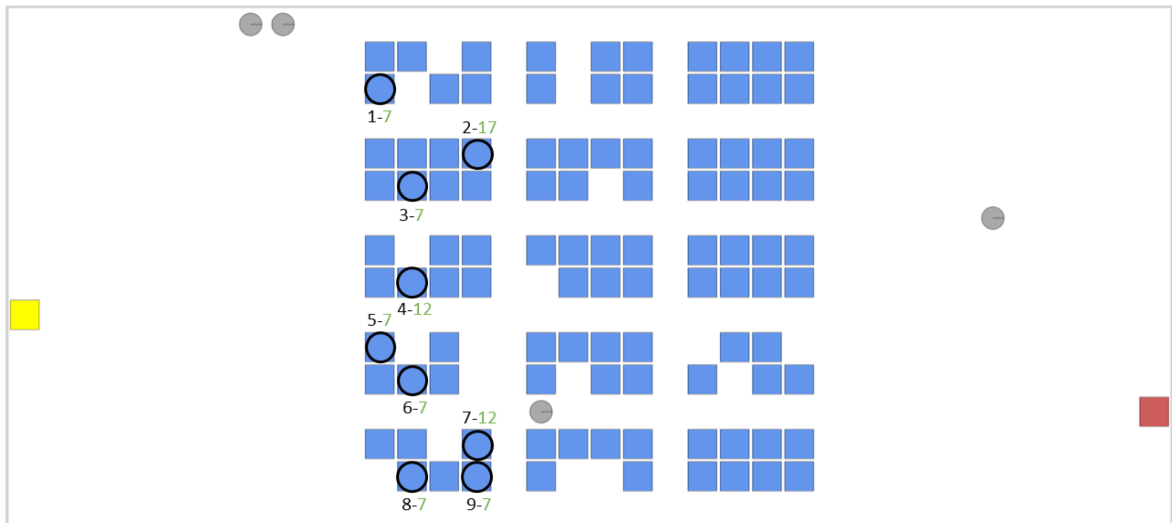
Another test that can be performed is to perform a simulation with a similar storage layout as for policy 1. Only this time reducing the amount of pods by almost half. The initial positions of the pod are completely random. So after a while the storage floor should segregate towards the picking and replenishment stations leaving the centre of the storage area empty. The results of this test can be seen in Figure 5.4. from this figure it can be clearly seen that after an hour of simulation time the pods have indeed segregated towards the stations.



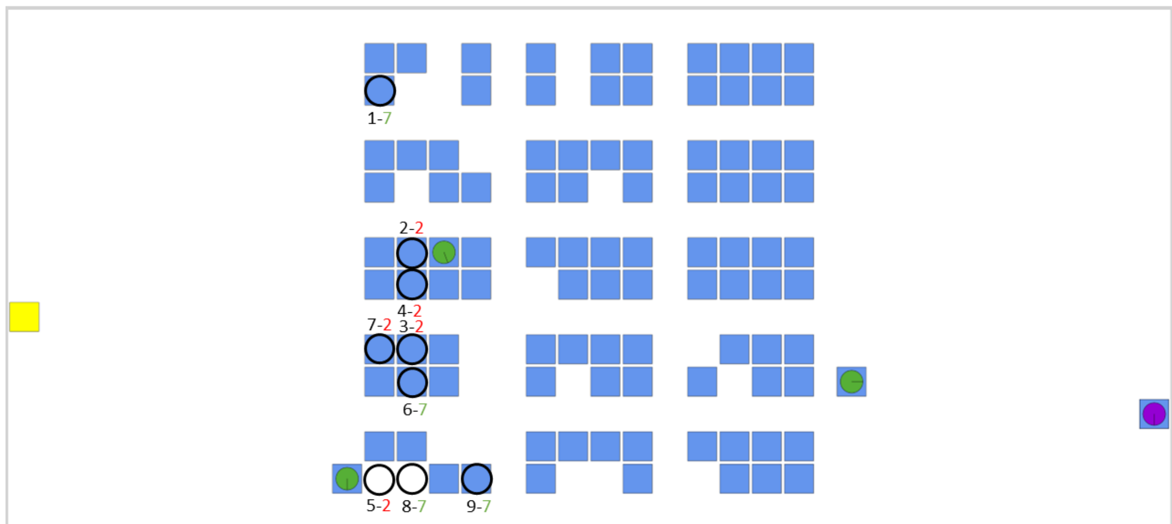
(a) Initial overview of the simulation and the location of the pods using policy 3.

(b) Final overview of the simulation and the location of the pods using policy 3.

Figure 5.4: Verification test for policy 3 showing that pods converge to the stations as intended.



(a) The location of all pods that are not full in the first section of pods at the start of the verification simulation using policy 3.



(b) The location of the 9 pods after the verification simulation ended using policy 1.

Figure 5.5: Verification tests for policy 3.

5.5. Verification - policy 4

The fourth policy also consists for the most part of algorithms that previously existed in the framework model. However during testing at least two issues were found in these algorithms. Therefore some verification should still be performed. One issue with the algorithm was the way in which storage zones were allocated to items. The algorithm decides on the 'priority' of an item using a function that includes both order count and the actual weight of an item, which is similar to formula 5.1 from policy 1. The order count is the actual amount of times an item has been ordered. This way both the order count, which gets updated throughout the simulation, and the physical weight/volume are used to determine the weighted demand of an item when assigning it to a storage zone. This then gets compared to the capacity of a storage zone to see whether the item fits

in the zone. However in some extreme cases where a certain item has a much higher order count or a much higher weight than the other items, the algorithm decides there is not enough capacity in a single storage zone, leaving the zone empty in the simulation. In most cases however, the input data includes a lot of different items, resulting in the WeightedItemDemand of each item to be relatively low. For this research only 3 zones are used in this policy, ensuring that the capacity of each zone is always enough to accommodate for multiple items no matter their order count or weight. This way of assigning items to storage zones using the weighted item demand and the order count can thus remain the same while it does make the algorithm very dynamic. This is also preferable since in normal operations the order count of items is also dynamic over a certain time period. Furthermore this way of assigning items to storage zones ensures that a zone is never completely filled which leaves room to more randomly distribute items across each zone.

The second issue found was in the way the storage zones were allocated on the storage floor. The algorithm allows the user to determine the borders of the storage zones manually. For example 0.2, 0.6, 1.0 means that the first zone will include the first 20% of pods (Seen as euclidean distance from the picking stations). Zone 2 will then include the next 40% of pods (0.2 - 0.6) and the final zone, zone 3, will include the last 40% of pods. This function did however not work properly but was quickly fixed by adjusting an error in the code.

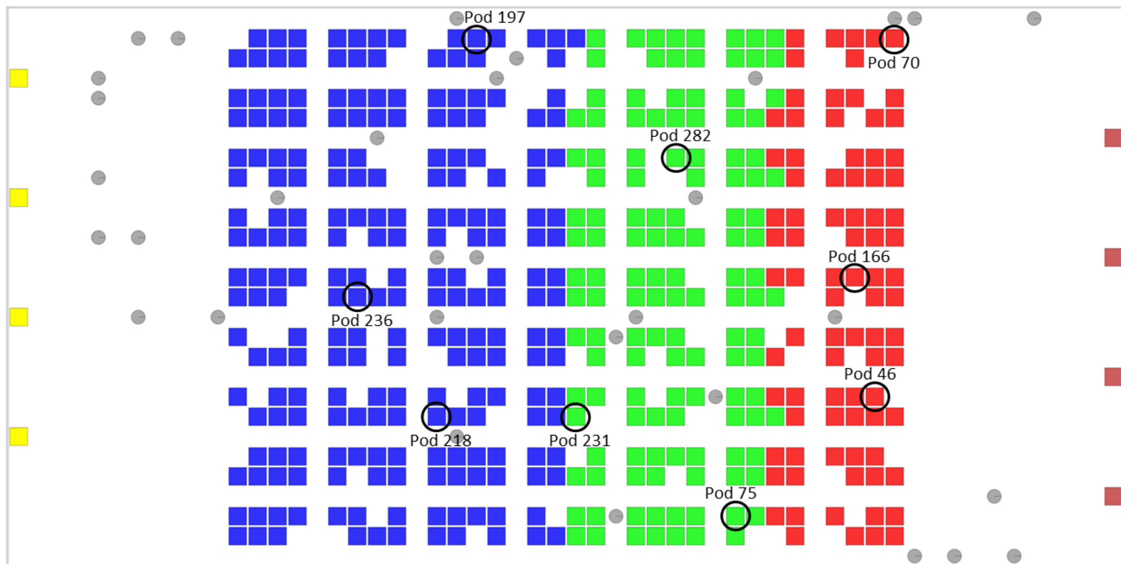
The first verification experiment that is performed is exactly the same as for policy 1. Also the input data used is similar. This way it can be ensured that the distribution of items over the storage area is as intended. So again a simulation was generated using the GUI. However this time applying the algorithms corresponding to policy 4. From each zone 3 pods are chosen at random in order to monitor its location and contents. This manner can give a good overview of the items located in each zone. An overview of the results can be found in Table 5.4.

Pod	Storage zone	Initial position (X/Y)	Final position (X/Y)	Initial contents (Product No. (amount))	Final contents (Product No. (amount))
70	1	55.40/26.50	41.50/8.50	0(15)	0(15)
46	1	43.50/8.50	42.50/11.50	0(15)	0(5)
166	1	42.50/14.50	42.50/25.50	0(15)	0(15)
282	2	33.50/20.50	34.50/1.50	2(15)	1(10) 2(5)
75	2	36.50/2.50	55.50/15.50	1(10) 2(5)	1(10) 2(3)
231	2	28.50/7.50	36.50/16.50	Empty	1(10) 2(5)
218	3	21.50/7.50	11.50/26.50	4(10) 8(5)	3(5) 6(5) 8(4)
197	3	23.50/26.50	17.50/16.50	4(5) 6(5) 8(5)	3(5) 4(10)
236	3	17.50/13.50	16.50/1.50	3(5) 4(5) 5(5)	3(5) 5(5)

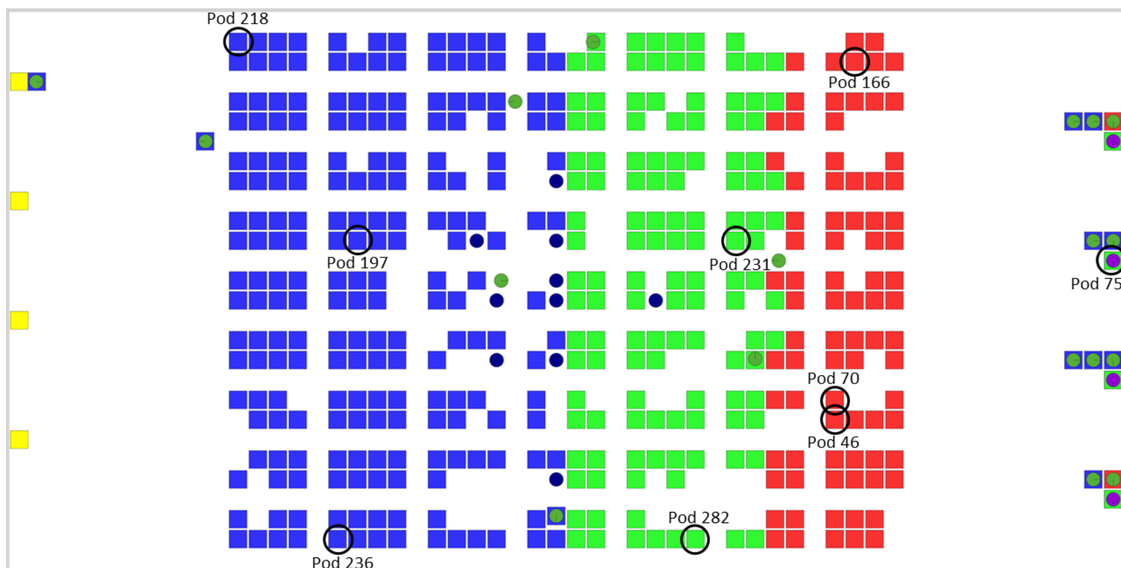
Table 5.4: Monitored pods and their initial/final location/contents, applying policy 4.

The first noticeable difference between this policy and policy 1 is the distribution of items right after the initial generation of the simulation. This time the weighted item demand is being used to determine to which zone an item has to be assigned. Resulting in only item 0 being stored in zone 0 (closest to the picking stations). Zone 1 includes only items 1 and 2 and zone 2 contains all other items. Another effect that now occurs is that none of the zones are completely filled. Each zone has a dozen of pods that are left empty. Looking at the final contents of each pod it is clear that the distribution of items over the storage area remained very similar. Each pod still contains the items that were allocated to its respective zone at the initial generation. This is to be expected even though this policy is dynamic, since the input data only includes

10 items with each relatively big differences in order frequency. Lastly the final locations of the pods are completely different from the initial locations, except for the fact that they have all remained in their initial zones. This is also as intended while the pods may be assigned to a random free storage space in its respective zone. The initial and final state of the simulation and the position of the pods can be seen in figure 5.8.



(a) The locations of the 9 pods at the start of the simulation.



(b) The location of the 9 pods after the simulation ended.

Figure 5.6: Verification tests for policy 4.

5.6. Verification - Policy 5

This policy is by far the most complex. Mainly because a lot of code has been adjusted and/or added. The main concept of this policy is that pods now consists of multiple compartments with each its own capacity instead of just a singular overall capacity. In order to verify if everything is in working order, multiple tests have to be performed. Unfortunately this was not possible in the given time frame of the research. Still one verification test was performed to see whether or not the model still behaves properly.

This verification test is to see whether the model still acts as it should. The original model uses only one single capacity for each pod. When testing the new model which incorporates compartments, using only 1 compartment, the results should be the same for the old model as for the new model. To test this a simple experiment is set up. A simulation will be ran for both the models. Since policy 3 uses a random item and pod storage the results shall be compared to policy 2, which also uses random item and pod storage. The simulation horizon will be 24 hours. All other settings will also be the same for both simulations. The results of both simulations can be seen in the combined graph in figure 5.7. It can be seen that both models resemble each other very well. There is a slight difference in replenishment throughput during the first full cycle. This may be caused by the fact that the decision algorithm for item storage in policy 5 has been altered a little to accommodate for compartments.

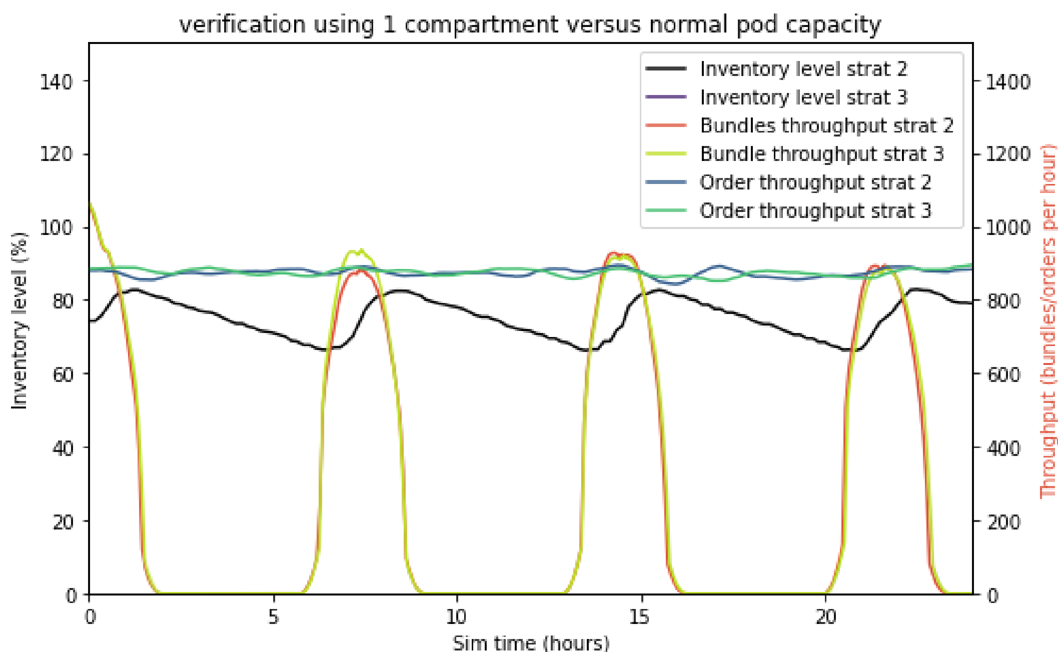


Figure 5.7: Results of policy 3, using 1 compartment and policy 2 with the same simulation settings.

5.7. Model validation

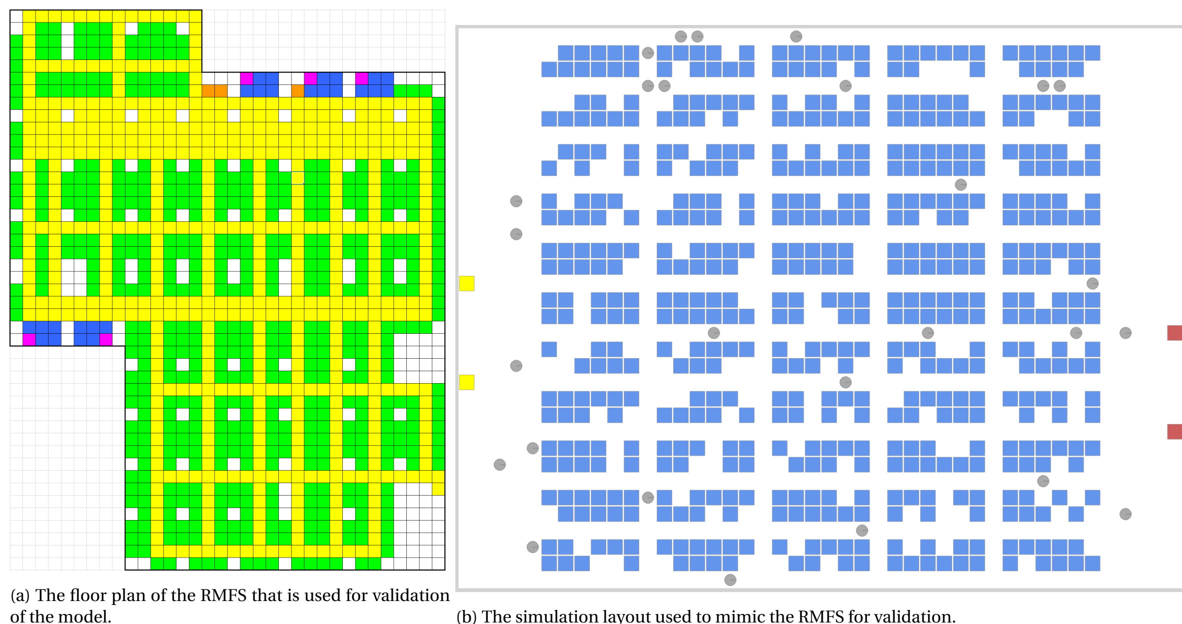
For the validation of the model some actual historical data has been acquired from a RMFS. The data ranges from January 2023 until March 2023. For the most part it contains information about the orders that have been completed within this time frame. Using the input data tool, 1 random week from this data is analysed and an input data file is created. Unfortunately information about SKU weights and bundle sizes were not available, these are thus determined to be some arbitrary distribution. The total amount of unique items sold in this week came down to 1740. The SKU weights are randomly picked from a normal distribution with parameters $\mu_W = 6$ and $\sigma_W = 1$ and are bounded between 1 and 11. The bundle sizes are randomly picked from a normal distribution with parameters $\mu_B = 5$ and $\sigma_B = 1$ and are bounded between 2 and 8. It was furthermore given that pods contained 12 compartments, which means that 12 bundles can be stored inside a pod at any given time. This results in a maximum pod capacity of: $PodCapacity = 12 * \mu_W * \mu_B = 360$. Other information that is extracted from the order data is the average amount of lines per order and the

amount of items that are ordered per line. All final input data can be seen in Table 5.5 below. The policy used at this RMFS is a fully random one, which is similar to policy 2 of this research. This data can thus be perfectly used to validate policy 2 of the model. However since this is the only available data from an operational RMFS also the other policies shall be simulated and compared to this data.

Input data parameter	Distribution	Value	Boundaries
Number of items	-	1740	-
Item description	Normal	$\mu_B = 200,\ \sigma_B = 20$	-
SKU weights (W)	Normal	$\mu_B = 6,\ \sigma_B = 1$	1 < W < 11
Bundle sizes (B)	Normal	$\mu_B = 5,\ \sigma_B = 1$	2 < B < 8
Probability weight	Exponential	$\lambda = 1$	-
Items per line (I)	Normal	$\mu_B = 2.28,\ \sigma_B = 1.96$	1 < I < 15
Lines per order (L)	Normal	$\mu_B = 8.31,\ \sigma_B = 7.32$	1 < L < 48

Table 5.5: Table showing the relevant parameters used for the validation simulation.

In order to be able to get an accurate comparison it is important to mimic the RMFS as closely as possible in the model. In figure 5.8a the floor plan of the RMFS can be seen. Here the green squares represent the pods, the blue segments are stations and the yellow squares are floor tiles. In this case there are two pick stations located at the north side of the storage area and 2 replenishment stations at the south side of the storage area. It is not possible to exactly copy this layout using the model due to the discussed boundaries. Therefore some simplifications have to be made. First of all the amount of pods should be exactly the same. In this case the RMFS contains 535 pods. This property for the most part determines how many aisles the simulation should contain and what the pod amount should be. The actual shape of the RMFS can unfortunately not be recreated in the model so it has been designed in such a way that the maximum distance from a pod to a pick station is very similar. The simulation model is also always single-deep whereas the RMFS also has pod segments which are multi-deep. Furthermore the amount of AMRs that are used is 27. The final simulation model can be seen in figure 5.8b. A list of all the relevant parameters specific to this validation simulation is shown in Table 5.6.



(a) The floor plan of the RMFS that is used for validation of the model.

(b) The simulation layout used to mimic the RMFS for validation.

Figure 5.8

Layout parameter	Value
Simulation horizon	7 days
Seed	0
BotCount	27
PodCapacity	360
PodAmount	0.735
Number of picking stations	2 stations
Number of replenishment stations	2 stations
ItemTransferTime	15 seconds
ItemPickTime	8 seconds
ItemBundleTransferTime	20 seconds
ReplenishmentStationCapacity	720
PickingStationCapacity	8 orders
NrHorizontalAisles	10 aisles
NrVerticalAisles	4 aisles
HorizontalLengthBlock	6 pods

Table 5.6: Table showing the relevant parameters used for the validation simulation.

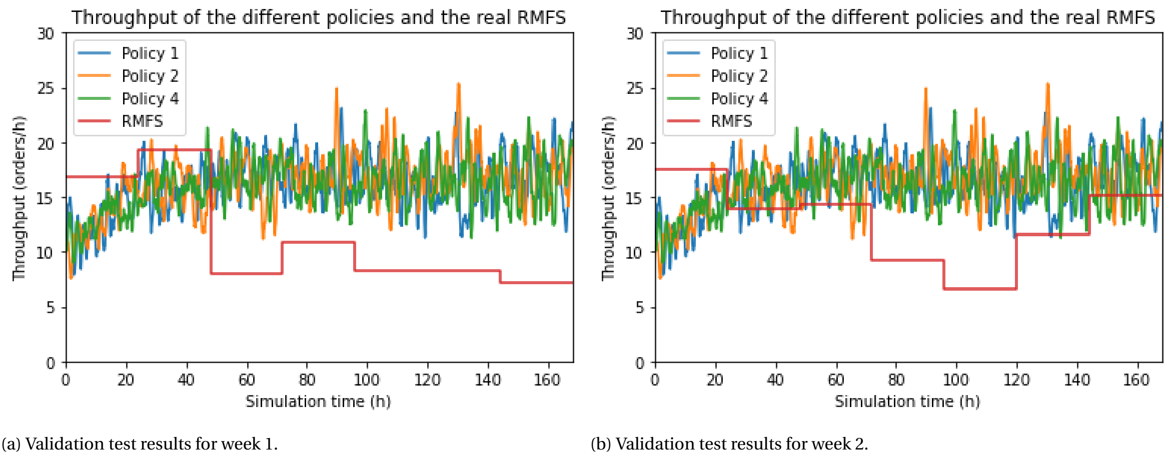
Now from both the historical order data and the simulation results the average throughput can be calculated. From the historical data 2 different weeks were selected and analysed. This order data unfortunately did not include any times, only the dates of the orders. It is therefore only possible to compute the average throughput per day in this case and then divide that by 24. This resulted in a throughput of 6.4 orders per hour for the historical data and 16.2 orders for the simulation model. It is however important to note that the physical RMFS only operates 14 hours a day, which increases the actual throughput per hour. The final results of the average throughput per hour can be seen in Table 5.7 and in Figure 5.9.

Throughput source	Orders/hour
Historical data week 1	11.24
Historical data week 2	12.60
Simulation policy 1	16.34
Simulation policy 2	16.67
Simulation policy 4	16.20

Table 5.7: The average throughput per hour for the historical data and the simulation model.

From the results two things can be noticed. The first thing is that all three policies result in pretty much similar numbers for the throughput. This can have a number of explanations, with the most plausible being that there are too many AMRs. This causes the system to be 'saturated', meaning that all policies have reached a maximum possible throughput, as will be explained in section 6.4.1.

The second thing that is noticeable is that the results from the simulation are on average about 38% higher than the actual RMFS throughput. However looking at Figures 5.9a and 5.9b, it can be seen that on some days



(a) Validation test results for week 1.

(b) Validation test results for week 2.

Figure 5.9: The throughput of all policies using the layout and input data from the RMFS for two different weeks.

the throughput is very similar between the model and the RMFS and on other days it is quite different. It is thus clear that the throughput of the RMFS is very inconsistent. This can have a number of explanations. The first issue can be that the pick and replenishment processing times are a lot more inconsistent in the practical case. Every employee operates at a different speed and some days the orders might be more complex than others whilst the simulation model is very consistent. Definitive data about these processing times is unfortunately not available, therefore in the model these had to be assumed. Another cause could be that there was some unexpected downtime of the RMFS on certain days which causes the throughput to be lower. Anyway it seems that there are definitive similarities between the throughput of the model and the throughput of the RMFS to successfully validate the model.

6 Experiments

Now that the model and all the added/changed policies have been verified and validated, the actual experiments can be performed. The four different policies will be tested using a generated set of data from the developed input data tool and using multiple different storage floor layouts.

6.1. Layout

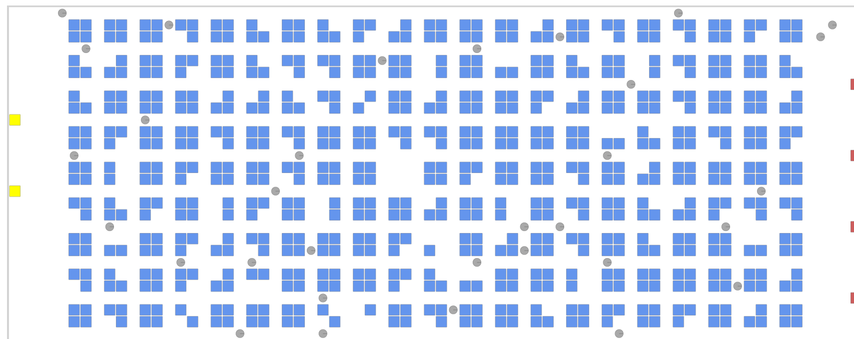
To gain good insight on the impact of different storage policies on the system multiple system layouts will be used in the experiments. The current model is limited to rectangular shapes. Therefore 3 different layouts are chosen. A wide storage floor, a long storage floor and a square storage floor. Wide in this case means that the replenishment stations and the pick stations are relatively close to one another whereas long means that the replenishment and pick stations are relatively far apart. For all layouts it is important to keep the amount of pods similar so this will not have an effect on the system performance. A total of 642 pods is chosen for each layout configuration. The three configurations can be seen in figure 6.1.

6.2. Input data

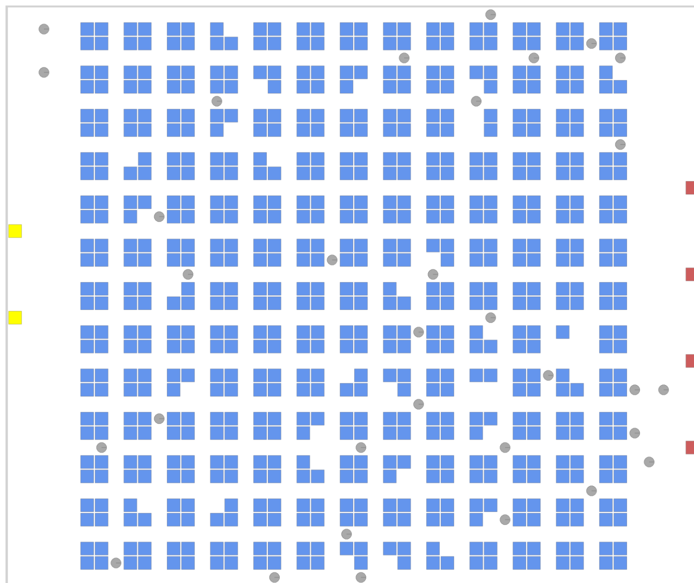
The input data will contain a total of 1000 SKUs. The weight of each SKU will be randomly picked from a normal distribution, with $\mu = 6$ and $\sigma = 1$. The distribution will also be cut of at values below 2 and values above 10. The bundle sizes will follow a similar distribution but with $\mu = 5$ and $\sigma = 1$ and limited to values between two and eight. Finally the order probability weight of each SKU will be randomly chosen from an exponential distribution with $\lambda = 1$ since this closely resembles a typical ABC curve in e-commerce (Merschformann et al. 2019). Finally the amount of lines per order and the amount of items per line should be determined. These will also follow a normal distribution. The values of the used distributions and their boundaries can be seen in Table 6.1.

Pod	Distribution	Parameters	Boundaries
Number of SKUs	-	1000	-
SKU description	Normal	$\mu = 200$, $\sigma = 20$	-
Weight (W)	Normal	$\mu = 6$, $\sigma = 1$	$1 > W > 10$
Bundle size (B)	Normal	$\mu = 5$, $\sigma = 1$	$2 > B > 8$
Probability weight (P)	Exponential	$\lambda = 1$	-
Lines/order (L)	Normal	$\mu = 1$, $\sigma = 1$	$1 > L > 4$
Items/line (I)	Normal	$\mu = 1$, $\sigma = 0.3$	$1 > I > 3$

Table 6.1: SKU properties of the input data.



(a) The long layout configuration, 20x8 aisles.



(b) The square layout configuration, 12x12 aisles.

(c) The wide layout configuration, 8x20 aisles.

Figure 6.1: The three different layout configurations used in the experiments, with yellow being the replenishment stations, red the picking stations and blue the pods.

6.3. General settings

For each policy 5 repeating simulations will be performed to decrease the effect of outliers (Merschformann et al. 2018a)(Merschformann et al. 2019). Each simulation run will use a different seed. The seed in a simulation model refers to the generation of random variables. Differing this seed every simulation ensures that each time new variables are generated for all the probability distributions present in the model. In this case this mainly refers to the generation of orders and the initial inventory generated by the model. Each simulation will furthermore contain a continuous simulation horizon of 72 hours. This ensures that there are more than enough storage cycles, i.e. replenishment order breaks and start ups, to be able to gain good insight in the actual replenishment and picking performance. The most relevant model settings which are used in every simulation can be seen in Table 6.2.

6.4. KPI evaluation

After all the simulations have been performed a decision should be made on which policy performs the best for the given input data. To do this the KPIs have to be analysed in such a way that a score can be given to each policy. Since the pile-on and the throughput of a RMFS should be maximised and the distance travelled by the AMRs should be minimised for the best performance, a simple scoring formula can be set up. Specifically for this research however, not only the picking side but also the replenishment side of the system should receive a score. These scores are then averaged to find the overall score for the policy. It should be noted however

Layout parameter	Value
Simulation horizon	72 hours
BotCount	2 per station
PodCapacity	360
CompartmentCapacity	-
NrHorizontalAisles	(8, 12, 20) aisles
NrVerticalAisles	(20, 12, 8) aisles
HorizontalLengthBlock	2 pods
Number of picking stations	4 stations
Number of replenishment stations	2 stations
ItemTransferTime	15 seconds
ItemPickTime	8 seconds
ItemBundleTransferTime	20 seconds
ReplenishmentStationCapacity	720
PickingStationCapacity	8 orders

Table 6.2: Table showing the relevant parameters and their values used across all simulations.

that the throughput and the pile-on consist of different units for the replenishing and the picking. In order to compensate for this the replenishing throughput is multiplied by the average bundle size and the picking throughput is multiplied by the average amount of items per order. The scores will be determined as follows:

$$S_p = \frac{TP_p * PO_p}{DT} * 100 \tag{6.1}$$

$$S_r = \frac{TP_r * PO_r}{DT} * 100 \tag{6.2}$$

$$Score = \frac{(S_p * i) + (S_r * b)}{2} \tag{6.3}$$

Where:
S_p / S_r = The score for the picking side or the replenishment side respectively.
i = The average amount of items per order.
b = The average bundle size.

6.4.1. Number of AMRs

The amount of AMRs in an RMFS is paramount for the performance of the entire system. Having too few AMRs causes pick and replenishment stations to stay idle for longer amounts of time. This can happen while all AMRs are currently occupied. Having too many AMRs will cause long queues at the stations and a higher chance of collisions among them. Both cases result in inefficient behaviour of the system. So the question remains: What number of AMRs should one choose? This is not a simple question and there are a lot of factors that influence this decision. Factors such as: the used storage policy, the type of SKUs that are in the system, the amount of different SKUs, the size and/or weight of the SKUs, the average size of orders, etc. In order to gain a better understanding on how to choose the right number of bots, some tests are run with the simulation model. Using the same simulation setup as described in the sections above but only the 'long' layout of the system, multiple simulations are run. However each time with a different amount of AMRs. The AMRs are chosen as 1, 2, 3, 4 and 5 per station. This means that the lowest amount of AMRs used is 6 and the highest amount is 30. The effect of the amount of AMRs will only be measured using the order throughput of the system. The results can be read in Table 6.3 and seen in figure 6.2. From the results it is visible that

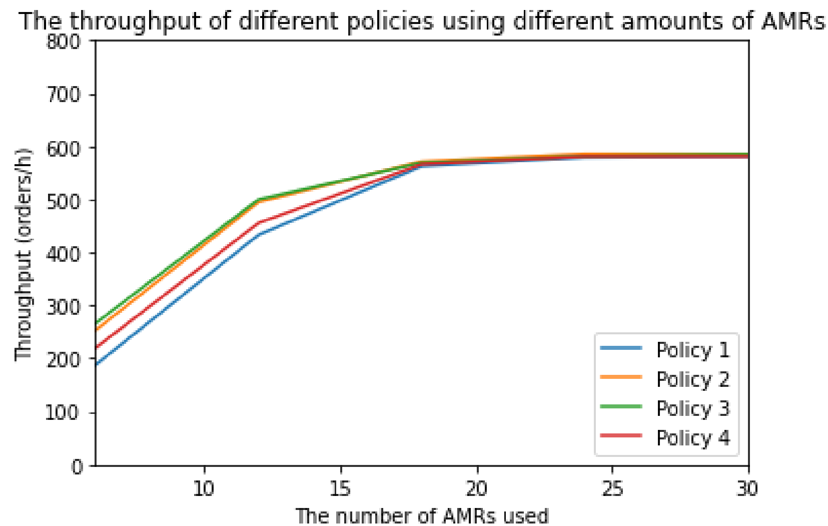


Figure 6.2: The throughput of different policies using different amounts of AMRs.

the throughput of the system reaches a maximum after a certain number of AMRs. Furthermore the system seems to saturate for 18 AMRs and beyond. Meaning that each policy starts to show similar results because the system is overfull with AMRs. For this reason the number of AMRs for the experiments should be picked lower than 18 in order to more clearly see the effects of the different policies on the system.

Average throughput (Orders/hour)	6 bots	12 bots	18 bots	24 bots	30 bots
Policy 1	187	433	562	579	581
Policy 2	253	495	571	585	585
Policy 3	266	500	569	581	584
Policy 4	219	455	565	581	581

Table 6.3: The throughput of the policies using different amounts of AMRs.

To reinforce the findings of the previously mentioned tests, a study is found by Gong et al. (2020), who has developed a relation showing the effect on the throughput of a RMFS, whilst varying the number of AMRs and pick stations. This relation can be seen in Figure 6.3. Also from this graph it becomes clear that the maximum throughput will saturate when increasing the number of AMRs whilst keeping a fixed number of pickers. For example when 5 pickers are used, the system is saturated when 10 or more AMRs are used. This relation is however developed with a small RMFS layout and a random storage policy, so the relation may differ for other systems. Combining the relation found by Gong et al. (ibid.) and the results previously discussed, two AMRs will be used per pick station for the experiments of this research. Also for each replenishment station two AMRs will be added.

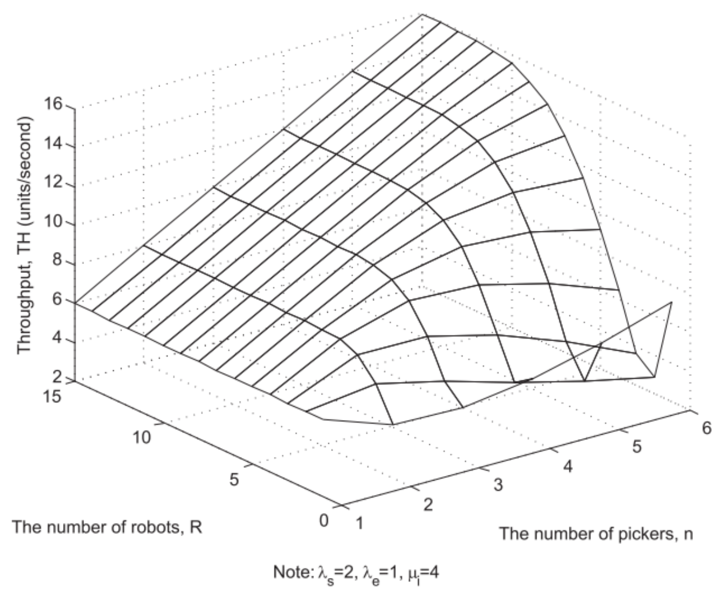


Figure 6.3: The relation between the number of AMRs and the number of picking stations according to Gong et al. (2020).

7 Results

The results from the experiments mentioned in chapter 6 will be shown in this chapter. For each of the chosen layouts (long, square and wide) the KPIs, averaged over 5 simulations, of the first four policies will be shown in tables. Also the standard deviation between the different seeds is added. Furthermore the gathered data for each policy will be shown in graphs.

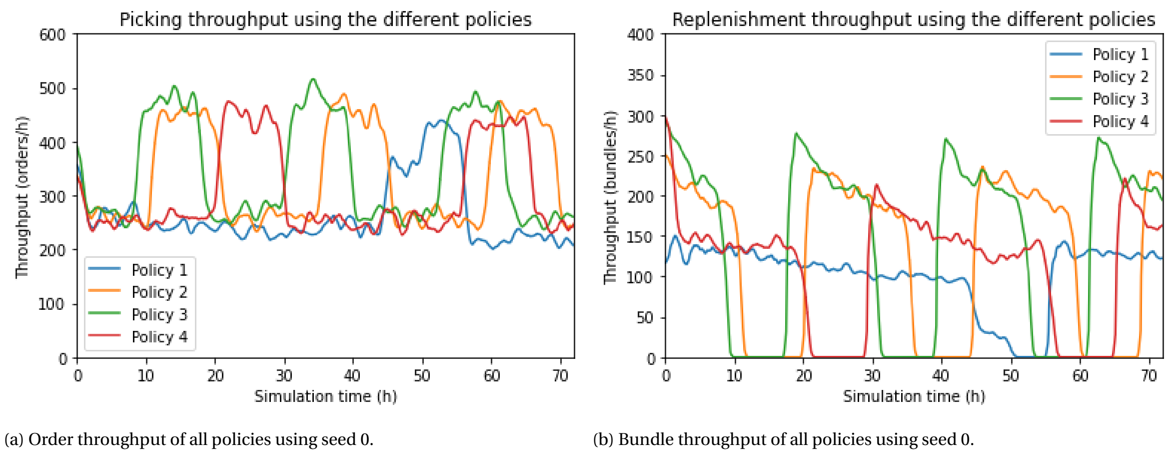
7.1. Long layout

In Table 7.1 the results can be seen from the experiments using the long version of the storage area. In this table the averages, over 5 different seeds, and the standard deviation are posted for each KPI. From this table it is clear that policy 3 scores the highest overall. Policy 1 scores the lowest and shows the largest deviations among similar simulation runs. Figures 7.1a, 7.1b, 7.2a, 7.2b and 7.3 show the exact data that was received from the simulations using seed 0 for each KPI. The data in these graphs is always represented as an average value per hour that was measured during the simulation.

KPI	Policy 1	Policy 2	Policy 3	Policy 4
Picking throughput (orders/h)	282 ± 2.82	362 ± 1.51	372 ± 2.40	326 ± 2.19
Replenishment throughput (bundles/h)	116 ± 5.77	198 ± 1.49	219 ± 1.70	156 ± 1.05
Picking pile-on (items/pod visit)	2.96 ± 0.07	3.38 ± 0.02	3.13 ± 0.04	3.08 ± 0.01
Replenishment pile-on (bundles/pod visit)	1.45 ± 0.08	2.58 ± 0.03	2.61 ± 0.05	1.98 ± 0.02
Distance travelled (m/h)	21.6e3 ± 73.9	18.2e3 ± 68.8	16.2e3 ±78.3	19.9e3 ± 54.1
Score picking	3.9	6.7	7.2	5.0
Score replenishment	0.8	2.8	3.5	1.5
Score	**3.9**	**10.4**	**12.4**	**6.4**

Table 7.1: All KPI results and the final score for the different policies using the long layout.

Multiple things can be distinguished from the graphs in Figure 7.1. On the left the throughput of the picking side, in orders per hour, is shown. On the right is the replenishment throughput which is represented as bundles per hour. In both graphs some form of periodicity can be recognised. There are moments when the order throughput increases for a short period of time. Comparing these moments to the bundle throughput, it can be seen that this occurs when the bundle throughput is at zero. The vice versa is also true. When the bundle throughput is not at zero, the order throughput decreases a bit. Furthermore it can be seen that policy 3 has the highest frequency, of around once every 22 hours, when talking about these fluctuating order

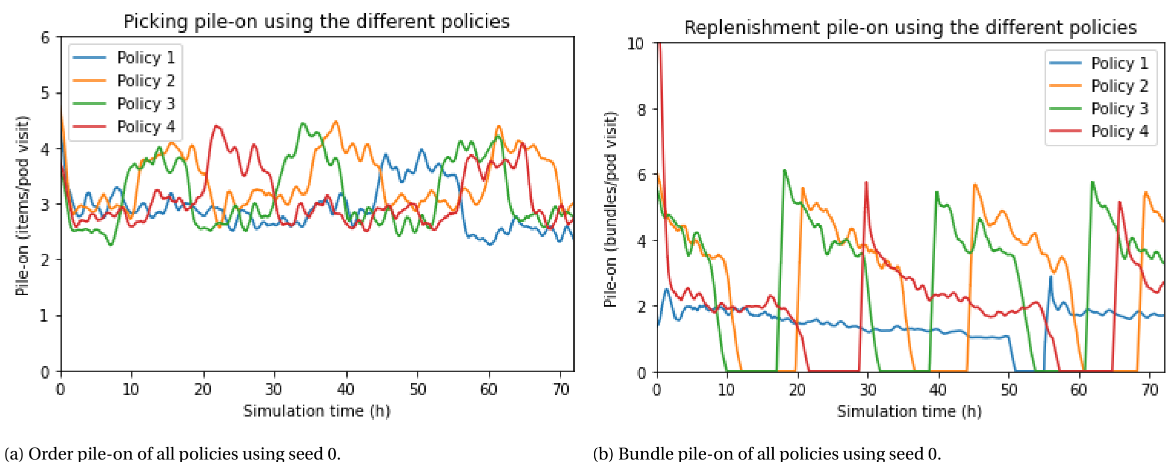


(a) Order throughput of all policies using seed 0.

(b) Bundle throughput of all policies using seed 0.

Figure 7.1: Throughput of the system for all policies using the long layout.

throughput moments. Policy 1 has by far the lowest frequency, while only 1 moment is recorded where the order throughput is increased. Because of this it is impossible to say what this frequency would be. Also the values of the order throughput follow these same differences. Policy 3 reaches the highest values, both in the increased parts as in the lower parts of the graph. Policy 3 reaches maximum order throughput values of 510 orders per hour and minimum values of 250 orders per hour. Policy 1 clearly reaches the lowest values with minimum values of a little over 200 orders per hour. It should be mentioned however that policy 2, 3 and 4 reach very similar values, whereas policy 1 reaches clearly lower values. The same can be said about the bundle throughput. Policy 3 has the highest values and the highest frequency. The frequency is similar to the picking throughput at 22 hours. The maximum and minimum values for policy 3 are 280 and 0 bundles per hour respectively. Another effect that is visible is that the period of increased bundle throughput is the shortest for policy 3, while it is the longest for policy 1. Policy 1 also has the lowest values and frequency, with a maximum of 150 bundles per hour. Another interesting effect that can be noticed is that when the bundle throughput increases from zero there is first always a peak, after which the throughput slowly decreases and then reduces all the way to zero. At the start of the simulation this peak is even the highest for policy 4.



(a) Order pile-on of all policies using seed 0.

(b) Bundle pile-on of all policies using seed 0.

Figure 7.2: Pile-on of the system for all policies using the long layout.

The pile-on of both the orders and the bundles, as seen in Figure 7.2, show very similar effects as the throughput. The differences in picking pile-on value are however a little less noticeable. All policies have fluctuating values between around 2.5 and 4.8 items per pod visit. The frequency differences are also comparable to the throughput results. The differences in the replenishment pile-on values and frequencies are also very similar as the the replenishment throughput. With maximum values reaching 6 bundles per pod visit. One noticeable effect is that policy 4 has a very large peak at the start of the simulation of over 10 bundles per pod visit.

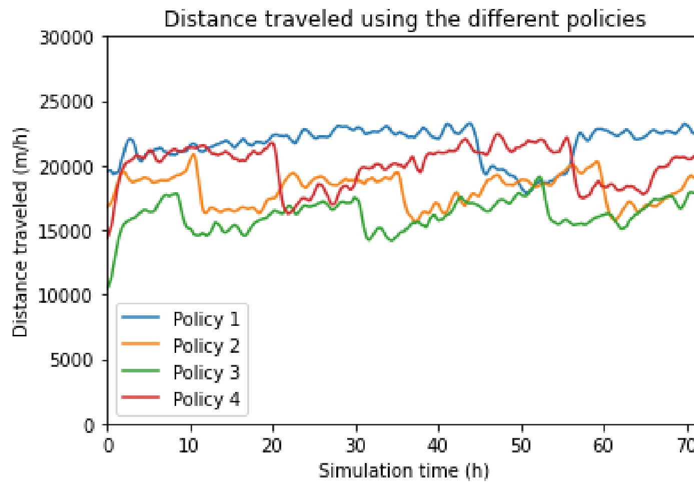


Figure 7.3: Distance travelled of all policies for the long layout using seed 0.

Figure 7.3 shows the average distance travelled per hour or each policy. At the start of the simulation this value increases rapidly for each policy after which the values again seem to be somewhat periodic. The same distribution of the frequency as mentioned for the throughput and the pile-on can be noticed. Also the minimum and maximum values of the policies are relatively the same as before. Policy 1 clearly has the highest average distance travelled with maximum values of around 22.5 kilometres per hour and policy 3 has the lowest values with minima reaching 15 kilometres per hour, ignoring the start of the simulation.

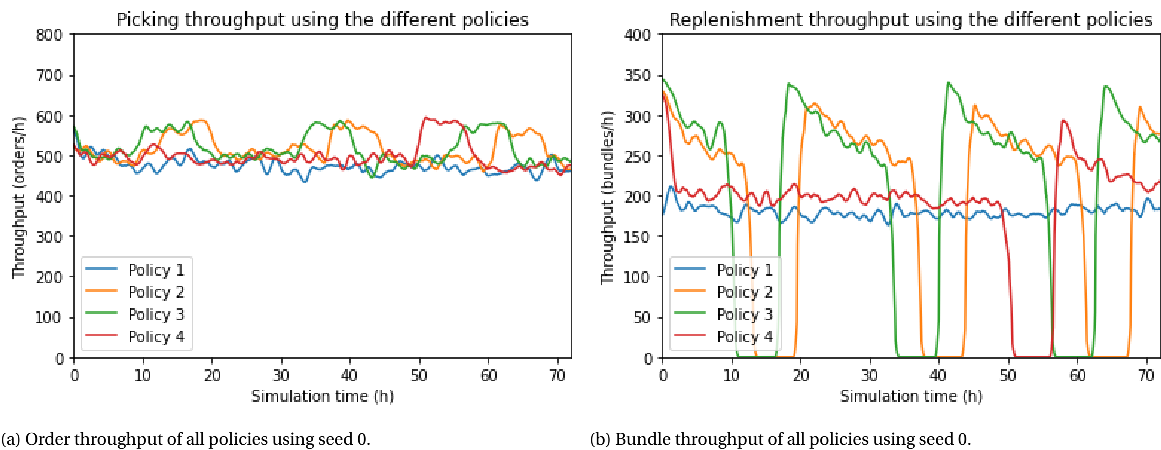
7.2. Square layout

Table 7.2 shows the results from the experiments using the square version of the storage area. Again policy 3 has the highest overall score and policy 1 the lowest. This time however the standard deviations of policy 3 are the highest instead of those of policy 1. What is furthermore noticeable is that the replenishment scores for policy 2 and 3 are closer together than for the long layout. Figures 7.4a, 7.4b, 7.5a, 7.5b and 7.6 show the exact data that was received from the simulations using seed 0 for each KPI. The data in these graphs is again represented as an average value per hour that was measured during the simulation.

KPI	Policy 1	Policy 2	Policy 3	Policy 4
Picking throughput (orders/h)	476 ± 0.99	519 ± 0.77	524 ± 2.24	501 ± 3.43
Replenishment throughput (bundles/h)	181 ± 0.51	267 ± 1.37	281 ± 1.71	209 ± 1.88
Picking pile-on (items/pod visit)	3.78 ± 0.026	4.13 ± 0.013	4.01 ± 0.039	4.03 ± 0.035
Replenishment pile-on (bundles/pod visit)	1.68 ± 0.006	2.81 ± 0.022	2.68 ± 0.034	2.04 ± 0.027
Distance travelled (m/h)	25.6e3 ± 89.8	20.4e3 ± 33.3	18.2e3 ± 171.0	23.4e3 ± 82.8
Score picking	7.0	10.5	11.5	8.6
Score replenishment	1.2	3.7	4.1	1.8
Score	**6.5**	**14.4**	**16.1**	**8.9**

Table 7.2: All KPI results and the final score for the different policies using the Square layout.

The throughput data of the simulations using the square layout of the storage area can be seen in Figure 7.4. When comparing the graph of the picking throughput (Figure 7.4a) to the picking throughput with the

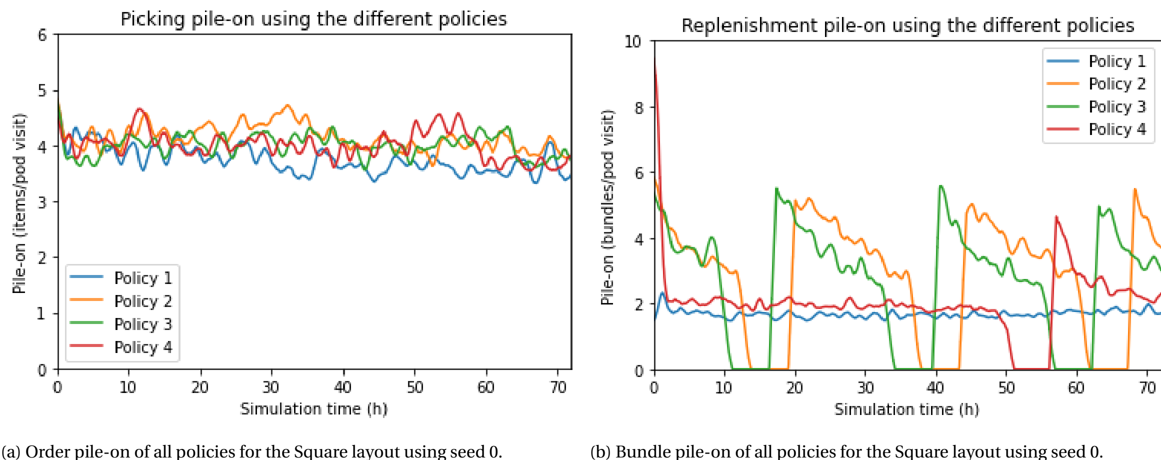


(a) Order throughput of all policies using seed 0.

(b) Bundle throughput of all policies using seed 0.

Figure 7.4: Throughput of the system for all policies using the square layout.

long layout (Figure 7.1a) the first noticeable thing is that policy 1 now has a more constant level of throughput, whereas with the long layout it showed a noticeable increase at a specific point. This behaviour is also mimicked in the replenishment throughput, Figure 7.4b. Another difference can be seen in the height of the values. For both the picking and the replenishment the throughput has increased by about 50%. Furthermore the range of the values is much smaller for the picking throughput of these simulations than with the long layout. With maxima reaching about 600 orders per hour and minima of around 450 orders per hour for the picking throughput and maxima for the replenishment throughput reaching 350 bundles per hour. Lastly the periodicity of policy 2 and 3 seems fairly similar to before. The frequency of policy 1 and 4 seems a lot lower.



(a) Order pile-on of all policies for the Square layout using seed 0.

(b) Bundle pile-on of all policies for the Square layout using seed 0.

Figure 7.5: Pile-on of the system for all policies using the square layout.

Figure 7.5 shows the pile-on for the picking and replenishment process. For the picking process the pile-on seems very similar for all policies. Only policy 1 seems to have a slightly lower pile-on after the start of the simulation. The values fluctuate between 3.5 and 4.8 items per pod visit. The pile-on for the replenishment process seems very similar to the pile-on in Figure 7.2b. The only difference is that the average value of the pile-on for policy 1 and 4 have become closer to one another and they seem more or less constant over the duration of the simulation. Again policy 4 has a very large peak at the start of the simulation for the replenishment pile-on.

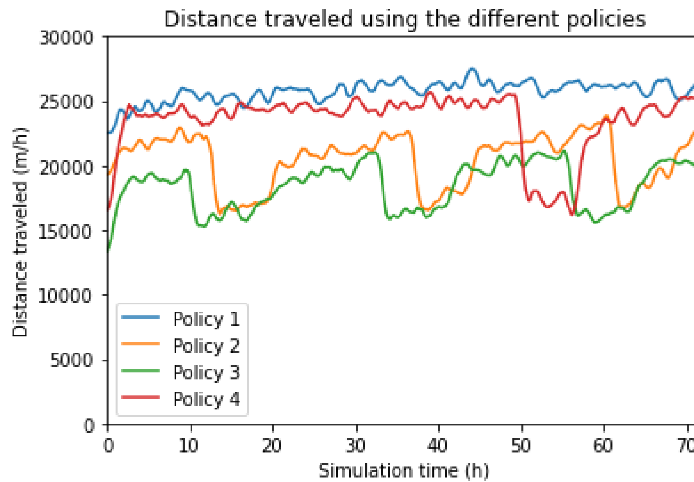


Figure 7.6: Distance travelled of all policies for the square layout using seed 0.

The average distance travelled per hour of each policy is shown in Figure 7.6. Also here policy 1 and 4 are for the most part constant with some fluctuations. Policy 1 has an average value of a bit over 25 kilometres per hour. The distance travelled for policy 2 and 3 seems very similar to Figure 7.3 with policy 3 reaching minima of a little over 15 kilometres per hour.

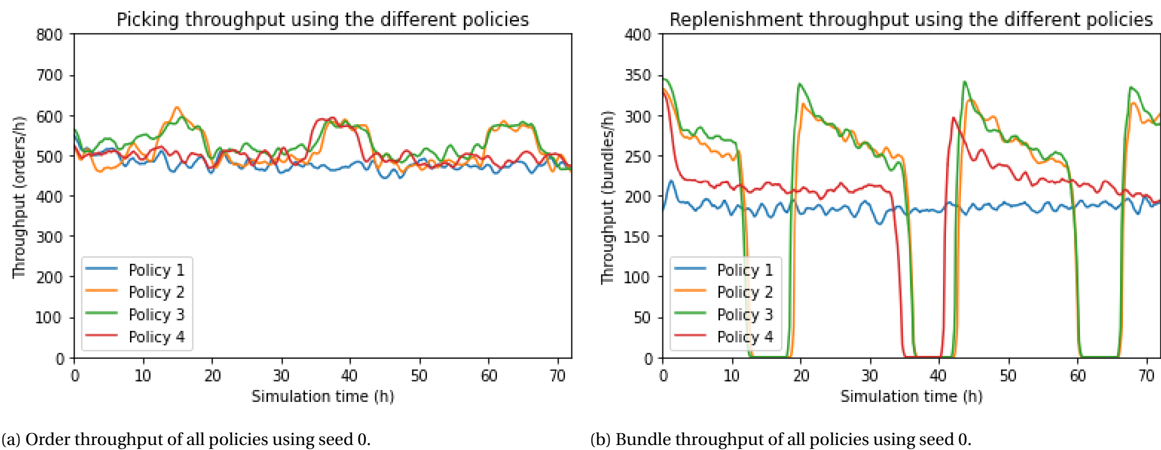
7.3. Wide layout

Finally Table 7.3 shows the results from the simulations using the wide layout. Also here policy 3 reaches the highest total score. This time however policy 1 and 4 show the largest variations between different tests, resulting in higher standard deviations for the KPIs. Again noticeable is that the replenishment scores for policy 2 and 3 are even closer together than in the previous results. Figures 7.7a, 7.7b, 7.8a, 7.8b and 7.9 show the exact data that was received from the simulations using seed 0 for each KPI. The data in these graphs is again represented as an average value per hour that was measured during the simulation.

KPI	Policy 1	Policy 2	Policy 3	Policy 4
Picking throughput (orders/h)	482 ± 3.00	516 ± 1.39	534 ± 1.71	506 ± 3.73
Replenishment throughput (bundles/h)	186 ± 1.38	269 ± 0.85	277 ± 1.31	220 ± 1.30
Picking pile-on (items/pod visit)	3.84 ± 0.055	4.08 ± 0.025	4.06 ± 0.014	3.99 ± 0.031
Replenishment pile-on (bundles/pod visit)	1.66 ± 0.019	2.82 ± 0.026	2.47 ± 0.024	2.04 ± 0.023
Distance travelled (m/h)	25.1e3 ± 179.9	20.7e3 ± 81.3	17.2e3 ± 76.8	22.3e3 ± 196.4
Score picking	7.4	10.2	12.6	9.0
Score replenishment	1.2	3.7	4.0	2.0
Score	**6.8**	**14.3**	**16.3**	**9.6**

Table 7.3: All KPI results and the final score for the different policies using the Wide layout.

Figure 7.7 shows the throughput of the different policies for both the picking side as the replenishment side of the system using the wide layout of the storage area. Looking at Figure 7.7a, policy 1 again has a more or less constant level of order throughput of around 480 orders per hour. Policy 2, 3 and 4 all have moments where the throughput is increased for a short period of time. The throughput seems to reach a maximum of about 600 orders per hour and a minimum of around 450 orders per hour. Interesting is that one of these

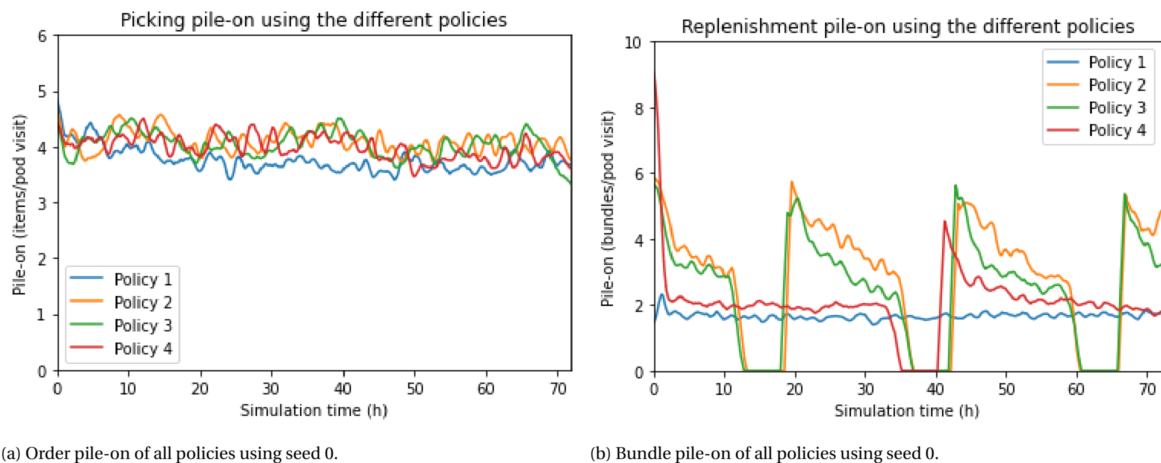


(a) Order throughput of all policies using seed 0.

(b) Bundle throughput of all policies using seed 0.

Figure 7.7: Throughput of the system for all policies using the wide layout.

periods seems to overlap for these three policies. The replenishment throughput, as seen in Figure 7.7b, is very similar to the situation with the square layout, Figure 7.4b.



(a) Order pile-on of all policies using seed 0.

(b) Bundle pile-on of all policies using seed 0.

Figure 7.8: Pile-on of the system for all policies using the wide layout

The pile-on of the picking and the replenishment process can be seen in Figure 7.8. The results for both these KPIs look very similar to Figure 7.5. The picking pile-on looks to fluctuate between 3.8 and 4.8 items per pod visit for policy 2, 3 and 4. Again policy 1 reaches slightly lower values for the picking pile-on and has clearly the lowest values for the replenishment pile-on with 2 bundles per pod visit.

The average distance travelled by the AMRs using the wide layout can be seen in Figure 7.9. These results also look very similar to 7.6. However policy 3 seems to reach slightly lower values with minima of around 14 kilometres per hour.

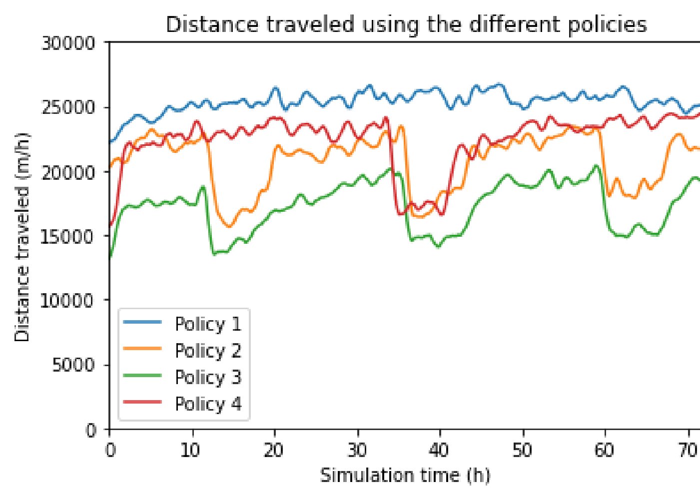


Figure 7.9: Distance travelled of all policies for the Wide layout using seed 0.

8 Discussion

In this chapter the results from the performed experiments are going to be discussed and evaluated. The chapter will be structured as follows: First the effect of each policy on the KPIs will be discussed, then any possible relations between the KPIs and finally any other effects that might have occurred during the experiments. Furthermore the main research question should be answered in this chapter. It should be noted that the results in chapter 7 are structured following the different storage layouts that were used. For the discussion however first the effect of replenishment pauses will be discussed, than the effects of the different layouts will be discussed for each KPI, starting with the pile-on and the distance travelled. The throughput and the final scores of the policies will be discussed last.

8.1. Replenishment activation/deactivation

Before the KPIs will be discussed individually it is important to mention that there is a similar phenomenon across almost all results. From all graphs it is clear that most of the KPIs show some periodicity. The frequency of this periodicity can differ but this will be discussed later. The cause of this periodicity however can be related to what is discussed in chapter 3 section 3.4.2, the deactivating and reactivating of replenishment orders. For all layouts and all policies some relation can be seen between the replenishment side of the system and the picking side of the system. This is caused by the fact that replenishment orders are stopped during the simulations when the inventory level reached a certain percentage. This in turn causes the picking side of the system to have access to more AMRs and thus have a somewhat increased performance for that period of time. This effect occurs in most results and will therefore also be mentioned multiple times.

8.2. Pile-on & distance travelled

Looking first at the pile-on for the picking process it is clear that policy 2 has the highest values overall. Only using the long layout policy 3 ends up slightly higher. This is somewhat unexpected since policy 1 and 4 group products based on their popularity. It would be logical that these policies have a higher pile-on rate than the more chaotic or random policies. There can be multiple reasons that has caused these results. One reason could be the used input data and order generation settings. For the experiments, order generation is set to using two normal distributions. One for the amount of lines that an order consists of and one for the amount of items that are in a line. Both of these were set pretty low, at a μ of 1 and σ of 1, following the research by Merschformann et al. (2019). This means that most orders only contain a few SKUs and most likely only 1 item of each SKU. This decreases the chance that multiple of the same SKU are needed at a picking station, which is what policy 1 and 4 are specifically designed for. Furthermore the used input data consist of a 1000 SKUs using an exponential distribution. This could be too many SKUs for policy 1 and 4 to actually have clear distinction in the popularity.

Another effect that can be noticed is that the pile-on of the picking process for the long layout of the storage floor scores a lot lower than for the square and wide layouts. On average almost a full item is being picked less per pod using the long layout. Looking also at the graphical data, one aspect is very noticeable that is most likely the cause for this lower pile-on rate. The pile-on for the square and wide layouts are more or less constant. All policies in this layout hover around some value for the picking pile-on. However using the long

layout some clear periodicity can be seen for all policies. This means that there are periods of time where the value for the pile-on is relatively low and periods where it is relatively high. Comparing the graphs of all the layouts, it can be seen that the 'high' moments using the long layout correspond to the 'constant' level that is achieved with the other layouts. This can be most likely explained by the fact that the AMRs have to travel further distances using the long layout, taking longer to fill up the total inventory of the system. Therefore experiencing longer effect of the moments where replenishment is active or not.

The pile-on of the replenishment process does not show this same effect when looking at the average values. All policies show very similar values in each scenario. The graphs also show that value wise there is not much difference. The only real difference is that the periodicity of the data changes for each scenario. Again this can be explained by the travel distance differences in the different scenarios. Furthermore policy 1 always has the lowest value for the replenishment pile-on followed by policy 4, than by policy 3 and finally policy 2 which has the highest values. This can be explained by the complexity of the replenishment process for each policy. Policy 1 and 4 have more complex rules for the storage of SKUs. Both policies for example only allow specific SKUs for certain pods. This causes replenishment stations to have to ignore empty storage locations in a pod whenever it visits the station. Policy 3 is a lot less complex since it only tries to pick the closest empty storage location and then puts the pod back on the closest storage space. This means that a lot of the pods close to the replenishment station have at least some capacity filled. Policy 2 is fully random and thus has a higher chance of picking completely empty pods. This is most likely the reason that this policy shows the highest replenishment pile-on.

As already mentioned the distance between the stations and the pods are larger using the long layout. However from the results it can be concluded that using the long layout the AMRs travel less distance overall. This is a very counter intuitive result. The only way to correctly explain this is by looking at the results for all the KPIs. In most occasions travelling less distance is good for the overall performance of a RMFS, however this only applies if the other KPIs do not worsen. In this case all other KPIs are also a lot lower. This points out that, although less distance is travelled, overall less actions are performed using the long layout. Probably because each AMR takes a lot longer to perform a single task. So even though the distance travelled using the long layout is better than using the other layouts, it does not mean that the overall performance is also better. Lastly it can be seen that policy 1 performs the worst in all scenarios, looking at the distance travelled. Policy 4 performs the second worst, policy 2 second best and policy 3 performs the best in all scenarios. The difference between the policies can be best explained by the complexity of the replenishment process.

8.3. Throughput

The first thing that is noticed from the result tables is that the throughput for both the picking and the replenishment side are a lot lower for the long layout compared to the other two layouts. This is most likely caused by the before mentioned reason, that travel distances are much further using the long layout. This corresponds to what one would expect. Looking further at the graphs some things can be said about the values of the throughput for the different policies. First of all the frequency differences of the policies. Clearly policy 1 has the lowest frequency, meaning that this policy has the slowest rate of filling the total inventory level of the system whilst replenishing is activated. This can also be substantiated by the fact that the picking throughput is fairly similar for all policies while replenishing is active, though the replenishing throughput is significantly different. Policy 1 clearly has the lowest replenishing throughput, which has a direct inverse correlation with the average picking throughput. In comparison policy 3 has the highest values for the replenishing throughput, resulting also in higher average values for the picking throughput. This conclusion can also be drawn from the fact that policy 3 has the shortest periods between its peaks, or the highest frequency. In case of this research it is thus fair to say that more complex policies, which should be optimal for the picking process, have an indirect negative effect on the picking performance.

Another phenomenon that should be discussed is the gradually decreasing replenishment throughput during active replenishment periods. At the start of each replenishment cycle a peak can be seen for each of the policies. A potential cause for this could be that at this moment in time the inventory of the system is relatively empty. This could make it easier to replenish pods. This can also explain the same phenomenon seen in the replenishment pile-on data. Especially policy 4 shows an exceptionally large peak at the start of every simulation. It is unsure why this peak is so much bigger than the others put this should not have a significant

effect on the total performance of the system while this happens only for a very short period.

8.4. Overall scores

Looking at the overall scores for each of the tested policies, it is clear that policy 3 performs the best in each scenario. Policy 2 is a close second with higher values for the pile-on in some scenarios. Policy 1 definitely performs the worst in each scenario. As mentioned this can have multiple reasons, such as: The number of AMRs, The used input data, The order generation settings, etc. All these parameters differ for every practical case and it is thus impossible to say whether or not policy 3 will out perform the other policies in different scenarios, however for the scenarios used in this research, a storage policy that always stores bundles in the closest available location and always returns pods to the closest available storage space seems to be the best choice.

9 Conclusion & Future Work

The performance of a robotic mobile fulfilment system is highly dependant on the decision algorithms that it operates on. This research has focused mainly on the replenishment pod selection and the pod storage assignment policies of such a RMFS. These are the two policies that together determine how the available storage space is utilised. This chapter will conclude all the findings of this research and give some recommendation for future research.

The main goal of this research was to help identify the best performing storage policy to use in a given warehousing scenario. This goal was supported by multiple smaller goals. The first one being able to interpret generic order data from a company and be able to translate it to useful input data for a simulation model. To achieve this a tool has been created using the Python programming language. This tool can extract all the necessary information from order data and generates a text file that can be used as input data for the created simulation model. The second goal was to actually develop a simulation model that is able to test different policies using this created input data. This was done by the use of a framework model which was adapted to fit the needs of this research. The final goal was to be able to give a well-argued advise on which storage policy would be the best to use for any given client. The question that is within this goal has been partially answered, while the answer is different for every client. However the goal is met by using the developed tools and methodology in this research. This way it is possible to give a well-argued advise for any set of order data and most generic storage layouts on which storage policy to use. This research was performed on behalf of the Technical University of Delft and the company CEVA Logistics the Hague.

The storage policies that were studied in this research are:

1. Fixed policy
2. Dynamic, random policy
3. Dynamic, closest policy
4. Dynamic, zone based policy
5. Dynamic, heterogeneous policy

These policies were chosen for multiple reasons. The first one being that policy 1, 2 and 5 were at the time being considered by the case study company as potential policies for their future RMFS. The second reason is that policy 4 has been well regarded in literature as being one of the most efficient algorithms for RMFSs and the last reason being that policy 3 is a very simple policy and is logically a first choice. Adjusting the framework model to be able to use policy 5 unfortunately was not managed in time. The other policies however have been fully verified and validated. These were then tested using the simulation model.

For the experiments a lot of parameters had to be determined. This included the layout of the storage floor, the number of autonomous mobile robots, the number of pick and replenishment stations, the order generation and more. Most of these parameters were drawn from other literature. For the storage floor it was

decided to use three different types of layouts in order to see the effect of this parameter on the results. These included a relatively long, a square and a wide type of layout. Furthermore the number of AMRs was set to 2 per station. This was determined based on multiple tests and another literature study. Then the number of stations had to be determined. Based on some testing it was clear that replenishing happens a lot faster than picking, therefore it was decided to have more pick stations than replenishment stations. Furthermore with the help of literature it was decided to use 2 replenishment and 4 pick stations. Finally the order generation settings were for the most important parameters also received from the created input data tool. With all the parameters set, the experiments were conducted. Each policy was simulated using the different layout scenarios. Each simulation run was repeated 5 times with different seeds to decrease the influence of outliers.

The results of the simulations were then evaluated using the throughput, the pile-on and the distance travelled during the simulation. The results of all the policies in the different scenarios were then compared and scored using a developed formula. For both the picking process and the replenishment process, policy 3 reached the highest scores in all scenarios and policy 1 the lowest. This was not expected in all scenarios, but it turns out that the complexity of the policies plays a huge role in the replenishment performance, which in turn can affect the picking performance. It can thus confidently be said that policy 3 is the best performing policy in the scenarios tested in this research. However it was also determined that a lot of parameters can influence the performance of these policies, such as the amount of AMRs or the input data, therefore experiments should thus always be repeated for new scenarios.

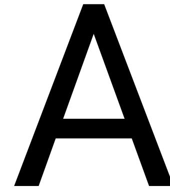
9.1. Recommendations future work

First of all every future research should take replenishment performance in to consideration when studying RMFS picking performance. As shown in this research some policies may not produce the expected results. This can have many causes, however one could be the complexity of the system. The effect a policy has on the replenishment process might also be negatively impacting the picking performance.

Secondly the model that was developed for policy 5, although it has not been fully verified and tested, should be close to complete. It would be highly recommended to continue the work done with this model. This type of RMFS model where pods can be fully equipped with different sized compartments does currently not exist in literature and might be useful to create more detailed storage policies.

Lastly as mentioned multiple times there are a lot of parameters that influence the system of a RMFS. In this research a lot of parameters were taken either from literature or assumed by the framework model. For a more complete research, a sensitivity analysis should be performed on the effect of these parameters on the replenishment and picking processes. Furthermore this research focused on 4 different storage policies. There are many more policies that exist in literature. Future research could extend this work by adding more unique policies and comparing overall scores for different scenarios.

Appendices



Input data tool: Python script

```
#writing a .xgenc file

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Loading in data depending on pc

# HOME PC DIRECTORY data
data = pd.read_excel(r'file_location', usecols=[12, 26, 30, 38])
data_volume = pd.read_excel(r'file_location', usecols=[1, 8])

#%% Filtering the data for the necesarry parts only
# Read headers
SKU = data.columns[0] # The SKU names
DATE = data.columns[1] # The dates
ORDER = data.columns[2] # The reference ID for the order
QTY = data.columns[3] # The quantities
VLM = data_volume.columns[1] # The volumes

# Filter date column to only show date and not time
data[DATE] = data[DATE].str[:9]

# Select the date of the first day
FirstDay = data[DATE][0]

# Selecting the orders from the first week UTC data
Orders = data.loc[data[DATE].isin(['01-DEC-18',
                                   '03-DEC-18',
                                   '04-DEC-18',
                                   '05-DEC-18',
                                   '06-DEC-18',
                                   '07-DEC-18',
                                   '08-DEC-18'])]

# Selecting only the product SKU ID, Reference ID and quantity columns
Orders_ColumnFilter = pd.DataFrame(Orders, columns=[SKU, ORDER, QTY])

# Drop all rows with NaN values
```

```
41 Orders_NoNaN=Orders_ColumnFilter.dropna()
42 Orders_NoNaN=Orders_ColumnFilter.dropna(axis=0)
43
44 # Reset index after drop
45 Orders_NoNaN=Orders_NoNaN.dropna().reset_index(drop=True)
46
47 # Group products with the same stock number and add the quantities
48 GroupBySKU_ID = Orders_NoNaN.groupby([SKU], axis=0).sum()
49
50 # Sort the data from highest quantity to lowest for a nice plot
51 Orders_Grouped = GroupBySKU_ID.sort_values(QTY, ascending=False)
52 Orders_Grouped = Orders_Grouped.reset_index()
53
54 # Merge the orders with quantity data with orders with volume data and sort data again
55 Orders_Merged = pd.concat((Orders_Grouped, data_volume), axis=0)
56 Orders_Merged = Orders_Merged.groupby([SKU], axis=0).sum()
57 Orders_Merged = Orders_Merged.sort_values(QTY, ascending=False)
58
59 # Filter out all values that are under a certain quantity for cleaner results
60 Orders_Filtered = Orders_Merged[Orders_Merged[QTY] > 0] # Filter out quantities that have
61     # not been sold in this week
62 Orders_Filtered = Orders_Filtered[Orders_Filtered[VLM] < 0.03] # Filter out weigths above
63     # 0.03 since we account for 16 locations per pod (2.4 m3) and want to be able to fit atleast
64     # 5 items in a location
65 Orders_Filtered = Orders_Filtered[Orders_Filtered[VLM] > 0] # Filter out weights of zero
66     # since these are not correct
67
68 # Extracting the quantities and weights to singular
69 Quantity = Orders_Filtered[QTY].values
70 Weights = Orders_Filtered[VLM].values
71 Weights_Ordered = np.sort(Weights)
72
73 #%% Calculations on the average lines and products in an order
74 # Sort the dataframe based on similar reference Ids
75 OrderLinesSorted = pd.DataFrame(Orders_NoNaN, columns=[ORDER, QTY])
76     .sort_values(ORDER, ascending=True)
77 OrderLinesSorted[QTY] = 1
78 OrderLinesGrouped = OrderLinesSorted.groupby([ORDER], axis=0).sum()
79 MeanOrderLines = OrderLinesGrouped[QTY].mean()
80
81 # Fit a normal distribution to the data and retrieve the min and max
82 LineOrders = OrderLinesGrouped[OrderLinesGrouped[QTY] < 50]
83 mu_Lines, std_Lines = norm.fit(LineOrders[QTY])
84 LinesMin = LineOrders.min()[0]
85 LinesMax = LineOrders.max()[0]
86
87 # Fit a normal distribution to the quantity that is ordered per line
88 ItemOrders = Orders_NoNaN[Orders_NoNaN[QTY] < 100]
89 mu_Items, std_Items = norm.fit(ItemOrders[QTY])
90 ItemsMin = ItemOrders[QTY].min()
91 ItemsMax = ItemOrders[QTY].max()
92
93 print("----------------- Order amounts and order lines -----------------")
94 print("The Amount of items per line can be written as a normal distribution with mu = %.2f and
95     "std = %.2f with lower bound = %.2f and upper bound = %.2f" % (mu_Items, std_Items,
96     ItemsMin, ItemsMax))
```

```
print('\n')
print("The Amount of lines can be written as a normal distribution with mu = %.2f and std = %.2f with
    "lower bound = %.2f and upper bound = %.2f" % (mu_Lines, std_Lines, LinesMin, LinesMax))
print("-----------------------------------------------------------------")

AvThroughput = len(LineOrders)/(7*24)
print("The average throughput = ", AvThroughput, " orders per hour")
#%% Calculations and plots
# Find the probability of an item being ordered by dividing over the total orders
Total_QTY = sum(Quantity)
Prob_Quantity = (Quantity/Total_QTY)
Quantity_Weights = Prob_Quantity*100 #Times 100 to give better readable values

plt.plot(range(0,len(Prob_Quantity)), Prob_Quantity)
plt.xlabel('Item number')
plt.ylabel('Probability')
plt.title('Order frequency distribution')
plt.show()

plt.plot(range(Weights_Ordered.size), Weights_Ordered)
plt.xlabel('Product [i]')
plt.ylabel('Product volume (m3)')
plt.title('Product volume ordered lowest to highest')
plt.show()

#%% Calculating bundle sizes
PodVolume = 2.4 #m3
NmbLocations = 16
LocationVolume = PodVolume/NmbLocations

n = range(Quantity_Weights.size) #Actual sample size
# n = np.array(range(0, 10)) #Used for testing smaller sample sizes
# Prob_Quantity = [20, 10, 5, 1, 0.01] #Used for testing extreme probability differences

BundleSize = np.zeros(shape=(Quantity_Weights.size, 1))

for i in n:
    if  Weights[i] < 5e-4:
        BundleSize[i] = LocationVolume/5e-4
    elif Weights[i] >= 5e-4 and Weights[i] < 1e-3:
        BundleSize[i] = LocationVolume/1e-3
    elif Weights[i] >= 1e-3 and Weights[i] < 5e-3:
        BundleSize[i] = LocationVolume/5e-3
    elif Weights[i] >= 5e-3 and Weights[i] < 1e-2:
        BundleSize[i] = LocationVolume/1e-2
    else:
        BundleSize[i] = 5

BundleSize_Ordered = np.sort(BundleSize, axis=0)
plt.plot(range(BundleSize_Ordered.size), BundleSize_Ordered)
plt.xlabel('Product [i]')
plt.ylabel('Bundle size')
plt.title('Bundle sizes ordered lowest to highest')
plt.show()
#%% writing the .xgenc file

```

```
153 # HOME PC DIRECTORY
154 path = 'file_location'
155
156 f = open(path, 'w')
157 f.write('<?xml version="1.0" encoding="utf-8"?>\n')
158 f.write('<SimpleItemGeneratorConfiguration xmlns:xsd="http://www.w3.org/2001/XMLSchema"
159     "xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">\n')
160 f.write('  <Name>Test-10</Name>\n')
161 f.write('  <DefaultWeight>0.01</DefaultWeight>\n')
162 f.write('  <DefaultCoWeight>0.2</DefaultCoWeight>\n')
163 f.write('  <ProbToUseCoWeight>0</ProbToUseCoWeight>\n')
164
165 f.write('  <ItemDescriptions>\n') # Write a generic description (random number) for every item
166 for i in n:
167     f.write('    <SkvpOfInt32Double>\n')
168     f.write('      <Key>' + str(i) + '</Key>\n')
169     f.write('      <Value>' + str(np.random.normal(200,20)) + '</Value>\n')
170     f.write('    </SkvpOfInt32Double>\n')
171 f.write('  </ItemDescriptions>\n')
172
173 f.write('  <ItemDescriptionWeights>\n') #Write weights (volume) of items
174 for i in n:
175     f.write('    <SkvpOfInt32Double>\n')
176     f.write('      <Key>' + str(i) + '</Key>\n')
177     f.write('      <Value>' + str(Weights[i]) + '</Value>\n')
178     f.write('    </SkvpOfInt32Double>\n')
179 f.write('  </ItemDescriptionWeights>\n')
180
181 f.write('  <ItemDescriptionBundleSizes>\n') # Write replenishment bundle sizes
182 for i in n:
183     f.write('    <SkvpOfInt32Int32>\n')
184     f.write('      <Key>' + str(i) + '</Key>\n')
185     f.write('      <Value>' + str(BundleSize[i, 0]) + '</Value>\n')
186     f.write('    </SkvpOfInt32Int32>\n')
187 f.write('  </ItemDescriptionBundleSizes>\n')
188
189 f.write('  <ItemWeights>\n') # Write item order probabilities
190 for i in n:
191     f.write('    <SkvpOfInt32Double>\n')
192     f.write('      <Key>' + str(i) + '</Key>\n')
193     f.write('      <Value>' + str(Quantity_Weights[i]) + '</Value>\n')
194     f.write('    </SkvpOfInt32Double>\n')
195 f.write('  </ItemWeights>\n')
196
197 f.write('  <ItemCoWeights />\n')
198 f.write('</SimpleItemGeneratorConfiguration>')
199 f.close()
```

B

Result processing: Python script

```
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
from tabulate import tabulate
import statistics as statistics

#%%
Policies = ['1', '2', '3', '4']#, '5'] # Policies with results
Seeds = ['0', '1', '2', '3', '4']

dataThroughputOrders = []
dataThroughputBundles = []
dataPileonOrders = []
dataPileonBundles = []
dataDistanceTraveled = []

for n in Policies:
    for s in Seeds:
        dataThroughputOrders.append(np.genfromtxt(r'file_location\CustomPlotDataOrdersHandledHour1200
            '.dat'))
        dataThroughputBundles.append(np.genfromtxt(r'file_location\CustomPlotDataBundlesHandledHour
            '1200.dat'))
        dataPileonOrders.append(np.genfromtxt(r'C:file_location\CustomPlotDataItemPileonHour1200
            '.dat'))
        dataPileonBundles.append(np.genfromtxt(r'file_location\CustomPlotDataBundlePileonHour1200
            '.dat'))
        dataDistanceTraveled.append(np.genfromtxt(r'file_location\CustomPlotDataDistanceTraveledHour
            '1200.dat'))
#%%

AvThroughputOrdersIndividual = []
AvThroughputBundlesIndividual = []
AvPileonOrdersIndividual = []
AvPileonBundlesIndividual = []
AvDistanceTraveledIndividual = []

for n in range(len(Policies) * len(Seeds)):
    AvThroughputOrdersIndividual.append(sum(dataThroughputOrders[n][:,1])/
        len(dataThroughputOrders[n][:,0]))
    AvThroughputBundlesIndividual.append(sum(dataThroughputBundles[n][:,1])/
```

```
41          len(dataThroughputBundles[n][:,0]))
42      AvPileonOrdersIndividual.append(sum(dataPileonOrders[n][:,1])/len(dataPileonOrders[n][:,0]))
43      AvPileonBundlesIndividual.append(sum(dataPileonBundles[n][:,1])/len(dataPileonBundles[n][:,0]
44      AvDistanceTraveledIndividual.append(sum(dataDistanceTraveled[n][:,1])/
45          len(dataDistanceTraveled[n][:,0]))
46
47  AvThroughputOrders = []
48  AvThroughputBundles = []
49  AvPileonOrders = []
50  AvPileonBundles = []
51  AvDistanceTraveled = []
52  StdAvThroughputOrders = []
53  StdAvThroughputBundles = []
54  StdAvPileonOrders = []
55  StdAvPileonBundles  = []
56  StdAvDistanceTraveled = []
57
58  for n in range(len(Policies)):
59      # Calculating the means
60      AvThroughputOrders.append(statistics.mean(AvThroughputOrdersIndividual[n * len(Seeds) :
61          ((1 + n) * len(Seeds))]))
62      AvThroughputBundles.append(statistics.mean(AvThroughputBundlesIndividual[n * len(Seeds) :
63          ((1 + n) * len(Seeds))]))
64      AvPileonOrders.append(statistics.mean(AvPileonOrdersIndividual[n * len(Seeds) : ((1 + n) *
65          len(Seeds))]))
66      AvPileonBundles.append(statistics.mean(AvPileonBundlesIndividual[n * len(Seeds) : ((1 + n)
67          * len(Seeds))]))
68      AvDistanceTraveled.append(statistics.mean(AvDistanceTraveledIndividual[n * len(Seeds) :
69          ((1 + n) * len(Seeds))]))
70      # Calculating the standard deviations
71      StdAvThroughputOrders.append(statistics.stdev(AvThroughputOrdersIndividual[n * len(Seeds) :
72          ((1 + n) * len(Seeds))]))
73      StdAvThroughputBundles.append(statistics.stdev(AvThroughputBundlesIndividual[n * len(Seeds)
74          : ((1 + n) * len(Seeds))]))
75      StdAvPileonOrders.append(statistics.stdev(AvPileonOrdersIndividual[n * len(Seeds) :
76          ((1 + n) * len(Seeds))]))
77      StdAvPileonBundles.append(statistics.stdev(AvPileonBundlesIndividual[n * len(Seeds) :
78          ((1 + n) * len(Seeds))]))
79      StdAvDistanceTraveled.append(statistics.stdev(AvDistanceTraveledIndividual[n * len(Seeds) :
80          ((1 + n) * len(Seeds))]))
81
82  #%% Plots
83  Seed = 0 # The seed to plot --> Seed = 0, plots all data of the policies using seed 0.
84  Policies = ['Policy 1', 'Policy 2', 'Policy 3', 'Policy 4']#, 'Policy 5']
85
86  plt.figure(1)
87  plt.title("Picking throughput using the different policies")
88  plt.xlabel("Simulation time (h)")
89  plt.ylabel("Throughput (orders/h)")
90  plt.xlim([0, 72])
91  plt.ylim([0, 600])
92  for n in range(len(Policies)):
93      plt.plot(dataThroughputOrders[Seed + n*len(Seeds)][:,0], dataThroughputOrders[Seed +
94          n*len(Seeds)][:,1])
95  plt.legend(Policies)
96
```

```
plt.figure(2)
plt.title("Replenishment throughput using the different policies")
plt.xlabel("Simulation time (h)")
plt.ylabel("Throughput (bundles/h)")
plt.xlim([0, 72])
plt.ylim([0, 400])
for n in range(len(Policies)):
    plt.plot(dataThroughputBundles[Seed + n*len(Seeds)][:,0], dataThroughputBundles[Seed +
        n*len(Seeds)][:,1])
plt.legend(Policies)

plt.figure(3)
plt.title("Picking pile-on using the different policies")
plt.xlabel("Simulation time (h)")
plt.ylabel("Pile-on (items/pod visit)")
plt.xlim([0, 72])
plt.ylim([0, 6])
for n in range(len(Policies)):
    plt.plot(dataPileonOrders[Seed + n*len(Seeds)][:,0], dataPileonOrders[Seed + n*len(Seeds)]
        [:,1])
plt.legend(Policies)

plt.figure(4)
plt.title("Replenishment pile-on using the different policies")
plt.xlabel("Simulation time (h)")
plt.ylabel("Pile-on (bundles/pod visit)")
plt.xlim([0, 72])
plt.ylim([0, 10])
for n in range(len(Policies)):
    plt.plot(dataPileonBundles[Seed + n*len(Seeds)][:,0], dataPileonBundles[Seed + n*len(Seeds)][:,1])
plt.legend(Policies)

plt.figure(5)
plt.title("Distance traveled using the different policies")
plt.xlabel("Simulation time (h)")
plt.ylabel("Distance traveled (m/h)")
plt.xlim([0, 72])
plt.ylim([0, 30000])
for n in range(len(Policies)):
    plt.plot(dataDistanceTraveled[Seed + n*len(Seeds)][:,0], dataDistanceTraveled[Seed + n*len(Seeds)][
plt.legend(Policies)

#%% Calculating KPI scores
Sp = []
Sr = []
Score = []
for n in range(len(Policies)):
    Sp.append(AvThroughputOrders[n] * AvPileonOrders[n] / AvDistanceTraveled[n] * 100)
    Sr.append(AvThroughputBundles[n] * AvPileonBundles[n] / AvDistanceTraveled[n] * 100)
    Score.append( ((Sp[n] * 1) + (Sr[n] * 5)) / 2)

Headers = Policies
Index = ['Throughput orders', 'Throughput bundles', 'Pile-on orders', 'Pile-on bundles',
    'Distance traveled', 'Picking score', 'Replnishment score', 'Total Score']
Table = [AvThroughputOrders, AvThroughputBundles, AvPileonOrders, AvPileonBundles, AvDistanceTraveled,
    Sp, Sr, Score]
```

```
153 print(tabulate(Table, headers=Headers, tablefmt='fancy_grid', showindex=Index))
154
155 Headers = Policies
156 Index = ['std Throughput orders', 'std Throughput bundles', 'std Pile-on orders',
157     'std Pile-on bundles', 'std Distance traveled']
158 Table = [StdAvThroughputOrders, StdAvThroughputBundles, StdAvPileonOrders, StdAvPileonBundles,
159     StdAvDistanceTraveled]
160 print(tabulate(Table, headers=Headers, tablefmt='fancy_grid', showindex=Index))
```

C
Simulation model settings

Layout parameter	Value	Description
TierCount	1	The number of floors.
TierHeight	4m	The distance between floors.
BotCount	12	The amount of AMRs.
BotRadius	0.35m	The radius of an AMR.
MaxAcceleration	$1m/s^2$	Maximum acceleration of an AMR.
MaxDeceleration	$1m/s^2$	Maximum deceleration of an AMR.
MaxVelocity	$1.5m/s$	Maximum speed of an AMR.
TurnSpeed	2.5rpm	Constant turn speed of an AMR.
CollisionPenaltyTime	0.5s	Time penalty when two AMRs collide.
PodTransferTime	2.2s	Time it takes for an AMR to pick up/put down a pod.
PodAmount	0.85/0.95%	Percentage of storage locations that include a pod.
PodRadius	0.45m	Radius of a pod.
PodCapacity	360	Capacity of a pod measured in a one-dimensional weight.
StationRadius	0.45m	Radius of a pick/replenishment station.
ItemTransferTime	15s	
ItemPickTime	8s	Time it takes for an employee to pick an item from a pod.
ItemBundleTransferTime	20s	Time it takes for an employee to replenish an item bundle to a pod.
IStationCapacity	720	Maximum capacity of a pick station measured in a one-dimensional weight.
OStationCapacity	8	Maximum capacity of a replenishment station measured in number of pick orders.
ElevatorTransportationTimePerTier	10s	Time it takes for an AMR to ascend/descend 1 floor using an elevator.
AisleLayoutType	Tim	
AislesTwoDirectional	False	Whether the aisles are one or two directional.
SingleLane	True	
NameLayout	CEVA-Layout	Name of this layout structure.
NrHorizontalAisles	8/12/20	Number of horizontal oriented aisles.
NrVerticalAisles	20/12/8	Number of vertical oriented aisles.
HorizontalLengthBlock	2	Amount of pods in one block in the horizontal direction.
VerticalLengthBlock	Read only	Amount of pods in one block in the vertical direction.
WidthRingway	Read only	
WidthHall	2m	Distance between picking stations and storage area.
WidthBuffer	2m	Length of the buffer area for pods at stations.

Table C.1: Table showing all layout parameters used in the model part A.

Layout parameter	Value	Description
DistanceEntryExitStation	3m	
CounterClockWiseRingwayDirection	True	
NPickStationWest	0	Number of picking stations on the west side.
NPickStationEast	4	Number of picking stations on the east side.
NPickStationSouth	0	Number of picking stations on the south side.
NPickStationNorth	0	Number of picking stations on the north side.
NReplenishmentStationWest	2	Number of replenishment stations on the west side.
NReplenishmentStationEast	0	Number of replenishment stations on the east side.
NReplenishmentStationSouth	0	Number of replenishment stations on the south side.
NReplenishmentStationNorth	0	Number of replenishment stations on the north side.
NElevatorWest	0	Number of elevators on the west side.
NElevatorEast	0	Number of elevators on the east side.
NElevatorSouth	0	Number of elevators on the south side.
NElevatorNorth	0	Number of elevators on the north side.

Table C.2: Table showing all layout parameters used in the model part B.

System parameter	Value	Description
Name	CEVA-Settings	Name of the setting configuration.
SimulationWarmUpTime	0s	Time spend to warm up the simulation.
SimulationDuration	172800s	Duration of the simulation.
Seed	0 - 4	Seed used for procedural generation.
LogLevel	Info	Level for output messages.
LogFileLevel	All	Which output files to generate.
MonitorWellSortedness	False	Whether to monitor the sortedness of the system.
DebugMode	RealTimeAndMemory	Current debug mode.
Tolerance	0.2m	Distance between pod and station.
UseAcceleration	True	Whether to visually show acceleration.
UseTurnDelay	True	Whether to visually show turn dela.y
RotatePods	False	Whether to visually show pods rotating.
QueueHandlingEnabled	True	Whether queues are used in the path finding.
StationShutdownTresholdTime	600s	Time after which a station is idle.
CorrelativeFrequencyTracking	True	Enables correlative frequency tracking between items
IntenseLocationPolling	False	Enables more frequent logging of the AMRs position

Table C.3: Table showing all system parameters (excluding the order generation) used in the model.

Item Parameter	Value	Description
ItemDescriptionCount	1000	Amount of unique items
ProbToUseCoWeight	0	Likelihood to use co-weight over the normal weight
DeafultWeight	1	Default weight to use when no other is given
DefaultCoWeight	0.2	Default co-weight (pick-hit probability) to use when no other is given
ProbWeightDistributionType	Exponential	Distribution for order weight/frequency items
ProbabilityWeightConstant	-	Parameter for constant distribution
ProbabilityWeightUniformMin	-	Parameter for uniform distribution
ProbabilityWeightUniformMax	-	Parameter for uniform distribution
ProbabilityWeightNormalMu	-	Parameter for normal distribution
ProbabilityWeightUNormalSigma	-	Parameter for normal distribution
ProbabilityWeightExpLambda	1.0	Parameter for exponential distribution
ProbabilityWeightGammaK	-	Parameter for gamma distribution
ProbabilityWeightGammaTheta	-	Parameter for gamma distribution
ProbabilityWeightLB	0	Lower bound for the generation SKU weights/frequencies
ProbabilityWeightUB	Infinity	Upper bound for the generation SKU weights/frequencies
WeightDistributionType	Normal	Distribution for weight of items
ItemWeightMU	5	Parameter for normal weight distribution
ItemWeightSigma	1	Parameter for normal weight distribution
ItemWeightLB	2	Lower bound for the generation physical SKU weights
ItemWeightUB	8	Lower bound for the generation physical SKU weights
SupplyBundleSize	True	Whether below distribution is used
BundleSizeDistribution	Normal	Distribution for the size of bundles
BundleSizeMu	6	Parameter for normal weight distribution
BundleSizeSigma	1	Parameter for normal weight distribution
BundleSizeLB	3	Lower bound for the size
BundleSizeUB	11	Lower bound for the size
GivenCoWeights	0	Percentage of items that will actually receive a co-weight

Table C.4: Table showing the parameters used to generate the item list.

Order Parameter	Value	Description
ItemType	SimpleItem	Type of item list
OrderMode	Fill	How orders are generated
SubmitBatches	False	Whether orders can be batched together
InitialInventory	0.7	Initial occupation of the storage
IgnoreCapacityForBundleGeneration	False	Whether bundles can be generated regardless of capacity
BufferBundlesUntilInventoryLoad	1.1	How many bundles should be buffered when inventory is full
WarmupOrderCount	0	Amount of orders to process during warm up
ItemWeightMin	5	Minimum item weight if no distribution is set
ItemWeightMax	5	Maximum item weight if no distribution is set
BundleSizeMin	6	Minimum bundle size if no distribution is set
BundleSizeMax	6	Maximum bundle size if no distribution is set
ReturnOrderProbability	0	Probability of returning orders
PositionCountMean	1	Average amount of items ordered per line
PositionCountStdDev	0.3	Respective standard deviation
PositionCountMin	1	Respective minimum
PositionCountMax	3	Respective maximum
OrderPositionCountMean	1	Average amount of lines per order
OrderPositionCountStdDev	1	Respective standard deviation
OrderPositionCountMin	1	Respective minimum
OrderPositionCountMax	4	Respective maximum
DueTimePriorityMode	True	Whether orders are generated with priority
DueTimePriorityOrderProbability	0.2	Probability of an order having priority
DueTimePriorityOrder	1800s	Maximum allowed cycle time of a priority order
DueTimeOrdinaryOrder	7200s	Maximum allowed cycle time of a normal order
DueTimeOffsetMean	10800s	
DueTimeOffsetStdDev	900s	
DueTimeOffsetMin	7200s	
DueTimeOffsetMax	14400s	

Table C.5: Table showing the parameters used to generate order from the item list during the simulation.

Bibliography

AnyLogic (2022). Why use simulation modeling? url: https://www.anylogic.com/use-of-simulation/.

Archambault, C. (2020). Determining Throughput Meaning vs. Profit. url: https://www.worximity.com/en/blog/determining-throughput-meaning-vs-profit.

Ashayeri, J. and L. F. Gelders (1981). "Warehouse design optimization". In: European Journal of Operational Research 21(1985), pp. 285–294. doi: https://doi.org/10.1016/0377-2217(85)90149-3.

Azadeh, K., R. de Koster, and D. Roy (2017). "Robotized Warehouse Systems: Developments and Research Opportunities". In: ERIM Report Series REsearch in Management 1(1), pp. 1–55.

Azadeh, K., R. de Koster, and D. Roy (2019). "Robotized and Automated Warehouse Systems: Review and Recent Developments". In: Transportation Science 53(4), pp. 917–945. doi: https://doi.org/10.1287/trsc.2018.0873.

Bassan, Y., Y. Roll, and M. J. Rosenblatt (1980). "Internal Layout Design of a Warehouse". In: AIIE Transactions 12(4), pp. 317–322. doi: 10.1080/05695558008974523.

Cai, J. et al. (2021). "Collaborative Optimization of Storage Location Assignment and Path Planning in Robotic Mobile Fulfillment Systems". In: Sustainability 13(10), p. 5644. doi: https://doi.org/10.3390/su13105644.

Chen, H. and D. Yao (2001). Fundamentals of Queueing Networks: Performance, Asymptotics, and Optimization. 46th ed. Stochastic Modeling and Applied Probability. Springer. isbn: 978-1-4757-5301-1.

Enright, J. J. and P. R. Wurman (2011). "Optimization and coordinated autonomy in mobile fulfillment systems". In: AAAI Workshop - Technical Report, pp. 33–38.

Galliussi, D. (2022). "Robotic mobile fulfillment systems (RMFS): a simulative study with three different managements". MA thesis. Padova, Italië: University of Padua.

Gong, Y., M. Jin, and Z. Yuan (2020). "Robotic mobile fulfilment systems considering customer classes". In: International Journal of Prodcution Research 1(1), pp. 1–20. doi: 10.1080/00207543.2020.1779370.

Gu, J., M. Goetschalck, and L. F. McGinnis (2006). "Research on warehouse operation: A comprehensive review". In: European Journal of Operational Research 177(2007), pp. 1–21. doi: https://doi.org/10.1016/j.ejor.2006.02.025.

Hazard, C. J., P. R. Wurman, and R. D'Andrea (2006). "Alphabet Soup: A Testbed for Studying Resource Allocation in Multi-vehicle Systems". In: Proceedings of AAAI Workshop on Auction Mechanisms for Robot Coordination, pp. 23–30.

Jünemann, R. (1989). Materialfluß und Logistiek. Logistik in Industrie, Handel und Dienstleistungen. Springer: Berlin. isbn: 978-3-662-08532-5.

Keung, K. L., C. K. M. Lee, and P. Ji (2021). "Data-driven order correlation pattern and storage location assignment in robotic mobile fulfillment and process automation system". In: Advanced Engineering Informatics 50(1). doi: https://doi.org/10.1016/j.aei.2021.101369.

Kim, H., C. Pais, and Z. M. Shen (2020). "Item Assignment Problem in a Robotic Mobile Fulfillment System". In: IEEE Transactions on Automation Science and Engineering 17(4), pp. 1854–1867. doi: 10.1109/TASE.2020.2979897.

Koster, R. de, T. Le-Duc, and K. J. Roodbergen (2007). "Design and control of warehouse order picking: A literature review". In: European Journal of Operational Research 182(2), pp. 481–501. doi: https://doi.org/10.1016/j.ejor.2006.07.009.

Kuijt, I. and B. Finlayson (2009). "Evidence for food storage and predomestication granaries 11,000 years ago in the Jordan Valley". In: Anthropology 106(27), pp. 10966–10970. doi: https://doi.org/10.1073/pnas.0812764106.

Lamballais, T., D. Roy, and M. B. M. de Koster (2017a). "Iventory Allocation in Robotic Mobile Fulfillment". In: IISE Transactions 52(1), pp. 1–17. doi: https://doi.org/10.1080/24725854.2018.1560517.

Lamballais, T., D. Roy, and M. B. M. De Koster (2017b). "Estimating performance in a Robotic Mobile Fulfillment System". In: European Journal of Operational Research 256(3), pp. 976–990. doi: https://doi.org/10.1016/j.ejor.2016.06.063.

Li, X. et al. (2020). “Storage assignment policy with awareness of energy consumption in the Kiva mobile fulfilment system”. In: Transportation Research Part E: Logistics and Transportation Review 144(1). doi: https://doi.org/10.1016/j.tre.2020.102158.

Merschformann, M. (2017). “Active Repositioning of Storage Units in Robotic Mobile Fulfillment Systems”. In: Operations Research Proceedings 2017 1(1), pp. 379–385. doi: https://doi.org/10.1007/978-3-319-89920-6_51.

Merschformann, M., L. Xie, and D. Erdmann (2018a). “Multi-Agent Path Finding with Kinematic Constraints for Robotic Mobile Fulfillment Systems”. In: 1(2), pp. 1–38. doi: https://doi.org/10.48550/arXiv.1706.09347.

Merschformann, M., L. Xie, and H. Li (2018b). “RAWSim-O: A Simulation Framework for Robotic Mobile Fulfillment Systems”. In: Logistics Research 11(8), pp. 1–11. doi: DOI_10.23773/2018_8.

Merschformann, M. et al. (2019). “Decision rules for robotic mobile fulfillment systems”. In: Operations Research Perspectives 6(1), pp. 1–17. doi: https://doi.org/10.1016/j.orp.2019.100128.

Mountz, M. C. et al. (2008). “Inventory system with mobile drive unit and inventory holder”. U.S. pat. US7402018B2. Amazon Robotics Inc.

Robotics 24/7 Staff (2021). Geek+ and GEODIS Optimize Warehouse Logistics With Autonomous Mobile Robots in Hong Kong. url: https://www.robotics247.com/article/geek_geodis_optimize_warehouse_logistics_autonomous_mobile_robots_hong_kong.

Rosetti, M. D. (2021). Simulation Modeling and Arena. 3rd ed. Github.

Roy, D. (2016). “Semi-open queuing networks: a review of stochastic models, solution methods and new research areas”. In: International Journal of Production Research 54(6), pp. 1735–1752. doi: https://doi.org/10.1080/00207543.2015.1056316.

Roy, D. et al. (2019). “Robot-storage zone assignment strategies in mobile fulfillment Systems”. In: Transportation Research Part E: Logistics and Transportation Review 122(1), pp. 119–142. doi: https://doi.org/10.1016/j.tre.2018.11.005.

Stauffer, B. (2022). How to Calculate Throughput. url: https://study.com/learn/lesson/throughput-rate-time-formula-calculate.html#:~:text=Throughput%5C%20Formula,-The%5C%20throughput%5C%20formula%5C&text=This%5C%20concept%5C%20is%5C%20applied%5C%20to,is%5C%20TH%5C%20%5C%3D%5C%20I%5C%20%5C%2F%5C%20T..

Tessensohn, T. L., D. Roy, and R. de Koster (2020). “Inventory allocation in robotic mobile fulfillment systems”. In: IISE Transactions 52(1), pp. 1–17. doi: https://doi.org/10.1080/24725854.2018.1560517.

Tompkins, J. A. et al. (2010). FACILITIES PLANNING. 4th ed. John Wiley & Sons. isbn: 978-0-470-44404-7.

TWI Global (2023). WHAT IS SIMULATION AND WHAT DOES IT MEAN? (DEFINITION AND EXAMPLES). url: https://www.twi-global.com/technical-knowledge/faqs/faq-what-is-simulation#:~:text=Why%5C%20is%5C%20Simulation%5C%20Used%5C%3F,comparing%5C%20alternative%5C%20solutions%5C%20and%5C%20designs..

Ungureanu, D. et al. (2005). “SIMULATION MODELING, INPUT DATA COLLECTION AND ANALYSIS”. In: Electronics, pp. 1–8.

Wang, K. et al. (2022). “Performance evaluation of a robotic mobile fulfillment system with multiple picking stations under zoning policy”. In: Computers & Industrial Engineering 169(1), pp. 1–21. doi: https://doi.org/10.1016/j.cie.2022.108229.

Wurman, P. R., R. D’Andrea, and M. Mountz (2008). “Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses”. In: AI Magazine 29(1). doi: https://doi.org/10.1609/aimag.v29i1.2082.

Xie, L. et al. (2021). “Introducing split orders and optimizing operational policies in robotic mobile fulfillment systems”. In: European Journal of Operational Research 288(1), pp. 80–97. doi: https://doi.org/10.1016/j.ejor.2020.05.032.